VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ZOBRAZOVÁNÍ VOXELOVÝCH SCÉN POMOCÍ RAY TRACINGU V REÁLNÉM ČASE

RENDERING OF VOXEL-BASED SCENES USING REAL-TIME RAY TRACING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAKUB MENŠÍK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MICHAL MATÝŠEK

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2020/2021

Zadání diplomové práce

Student: **Menšík Jakub, Bc.**
Program: Informační technologie Obor: Počítačová grafika a multimédia
Název: **Zobrazování voxelových scén pomocí ray tracingu v reálném čase**
Rendering of Voxel-Based Scenes Using Real-Time Ray Tracing
Kategorie: Počítačová grafika
Zadání:

1. Seznamte se s rozhraním DXR, případně s vybraným enginem využívajícím toto rozhraní.
2. Prozkoumejte možnosti uplatnění ray tracingu při vizualizaci rozsáhlých voxelových scén.
3. Navrhněte efektivní postup využívající ray tracing pro interaktivní vizualizaci voxelových scén.
4. Implementujte tento postup (v rámci vybraného enginu). Zvolte si a implementujte množinu efektů (minimálně stíny a ambient occlusion) pomocí ray tracingu.
5. Zhodnoťte dosažené výsledky, vytvořte krátké prezentační video a diskutujte možnosti budoucího vývoje.

Literatura:

- Dle doporučení vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3, rozpracovaný bod 4.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Matýšek Michal, Ing.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 19. května 2021
Datum schválení: 30. října 2020

Abstrakt

Cílem této práce bylo vytvořit program k vizualizaci voxelových scén v reálném čase s využitím ray tracingu. Součástí bylo studium různých metod takového vykreslování se zaměřením na stíny. K řešení bylo využito enginu Unity a experimentálních balíčků Unity Jobs a Burst. Práce představuje různé ray tracing průchody a metodu SVGF, která slouží především k filtrování vzniklého šumu se zachováním hran. Podařilo se vytvořit program, který vykresluje tvrdé stíny, měkké stíny a ambient occlusion rychlostí přibližně padesát snímků za sekundu.

Abstract

The aim of this work was to create a program to visualize voxel scenes in real time using ray tracing. It included the study of various methods of such a rendering with a focus on shadows. The solution was created using Unity engine and experimental packages Unity Jobs and Burst. The thesis presents multiple ray tracing passes and SVGF technique, that is used to turn a noisy input into full edge-preserving image. The final program is able to render hard shadows, soft shadows, and ambient occlusion at speed of fifty frames per second.

Klíčová slova

DirectX 12, sledování paprsku, sledování cest, vykreslování v reálném čase, sledování paprsku v reálném čase, voxel, procedurální generování, Unity, Unity Jobs, Burst, volumetrický terén, ambient occlusion, stínování, stíny, ostré stíny, měkké stíny, SVGF, šum, filtrování, paralelismus, reprojekce, variance, globální osvětlení.

Keywords

DirectX 12, ray tracing, path tracing, real-time rendering, real-time ray tracing, voxel, procedurl generation, Unity, Unity Jobs, Burt, volumetric terrain, ambient occlusion, shading, shadows, hard shadows, soft shadows, SVGF, noise, filtering, parallelism, reprojection, variance, global illumination.

Citace

MENŠÍK, Jakub. *Zobrazování voxelových scén pomocí ray tracingu v reálném čase*. Brno, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Michal Matýšek

Zobrazování voxelových scén pomocí ray tracingu v reálném čase

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Michala Matýška.

....
Jakub Menšík
17. května 2021

Poděkování

Rád bych poděkoval panu Ing. Michalu Matýškovi za odborné vedení a okamžitou pomoc kdykoliv bylo potřeba.

Obsah

1 Úvod **2**

2 Principy vykreslování pomocí ray tracingu **3**
- 2.1 Matematické vyjádření paprsku . . . 3
- 2.2 Druhy ray tracingu . . . 4
- 2.3 Grafické efekty . . . 8
- 2.4 DirectX 12 vykreslovací řetězec pro ray tracing . . . 10
- 2.5 Ray tracing v reálném čase . . . 12

3 Voxelové scény **15**
- 3.1 Procedurální generování terénu . . . 15
- 3.2 Reprezentace voxelové scény . . . 16

4 Návrh řešení **18**
- 4.1 Unity a SRP . . . 18
- 4.2 Struktura aplikace . . . 18
- 4.3 Generování terénu . . . 20

5 Implementace **22**
- 5.1 Tvorba SRP . . . 22
- 5.2 Předzpracování . . . 22
- 5.3 Ray tracing průchody . . . 25
- 5.4 Filtrování metodou SVGF . . . 29
- 5.5 Generování terénu . . . 32

6 Vyhodnocení **36**
- 6.1 Měření . . . 36
- 6.2 Výsledné snímky . . . 38

7 Závěr **41**

Literatura **42**

A Obsah přiloženého média **44**

B Ovládání aplikace **45**

Kapitola 1

Úvod

Žijeme ve světě, ve kterém se moderní technologie neustále vyvíjejí. Při sledování některých filmů či počítačových her lze jen stěží rozeznat, zda se jedná o realitu nebo uměle vytvořené efekty. Tato práce se zabývá takovými technologiemi, které se snaží reálnému zobrazování co nejvíce přiblížit. Fungují na principu ray tracingu neboli sledování paprsku. Rozhlédněte se po pokoji a vyberte si místo na stěně. Představte si, že k danému místu nakreslíte čáru. Od tohoto místa nakreslíte další čáru směrem ke světlu. Takto zjednodušeně se dá pohlížet na ray tracing. Jedná se o sledování šíření světla pomocí paprsků.

Tato technika se využívá již několik let ve filmovém průmyslu. Prvním animovaným filmem, který využil ray tracing pro vykreslení všech stínů a světel, byla Univerzita pro příšerky z roku 2013. Ve filmu je přibližně 24 snímků za sekundu a film trvá 104 minut, což vychází odhadem na 150 tisíc snímků. Vykreslení jednoho snímku v tomto filmu trvalo 29 hodin, uvedl technický ředitel Sanjay Bakshi [16]. Díky velkému množství zařízení, na kterých se film vytvářel, trvalo vytvoření filmu pouze čtyři roky. Od té doby se technologie vyvinuly a v dnešní době je již možné využívat ray tracing v reálném čase, například v počítačových hrách. Díky modernímu hardwaru je možné vykreslit jeden snímek přibližně za 20 milisekund.

Kapitola 2

Principy vykreslování pomocí ray tracingu

Tato kapitola popisuje teoretické základy vykreslování pomocí ray tracingu. Věnuje se základnímu vyjádření paprsku, které je nezbytné pro pochopení dalších podkapitol. Dále se zabývá vývojem různých metod pro sledování paprsku až po metody používané v dnešní době. Pomocí ray tracingu je možné dosáhnout různých efektů, které jsou v této kapitole také zmíněny. Ray tracing je možné využívat v reálném čase především díky nejnovějším grafickým kartám společnosti Nvidia, proto je zde podkapitola věnovaná rozhraní DirectX 12, které je pro řízení grafických karet využito.

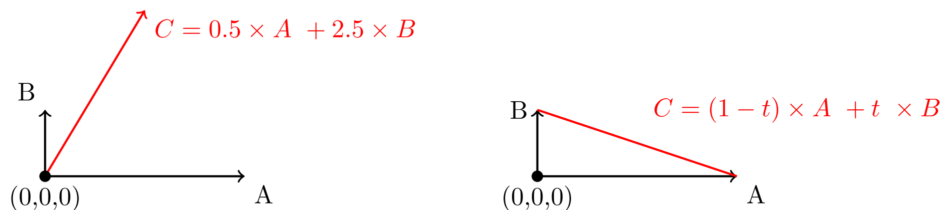
2.1 Matematické vyjádření paprsku

Tato sekce vychází z kurzu *Úvod do DirectX ray tracingu* [18]. Ještě předtím, než se tato práce začne zabývat sledováním paprsků a grafickými efekty, kterých lze sledováním paprsků dosáhnout, je dobré si definovat samotný paprsek. Nejlépe se dá představit jako laserové ukazovátko. Je to tedy dvojice – bod v 3D prostoru, který může být nazván jako počáteční, a směr, kterým se paprsek z daného bodu šíří. Tato reprezentace je ale v počítačové grafice nevyhovující, protože je často potřeba najít místo průniku paprsku a jiného objektu ve scéně, a k tomu je nezbytné vědět vzdálenost od počátečního bodu.

První možností reprezentace paprsku je průnik dvou rovin. Tato verze se však v programování sledování paprsků téměř nevyskytuje. Další variantou je lineární kombinace dvou bodů v 3D prostředí. Body v 3D prostředí bývají reprezentovány jako směrový vektor z počátku O[0,0,0] do samotného bodu. Mějme tedy dva body A a B. Pomocí lineární kombinace se dá vyjádřit bod C, ležící v rovině AB, s počátečním bodem O[0,0,0], viz obrázek 2.1.

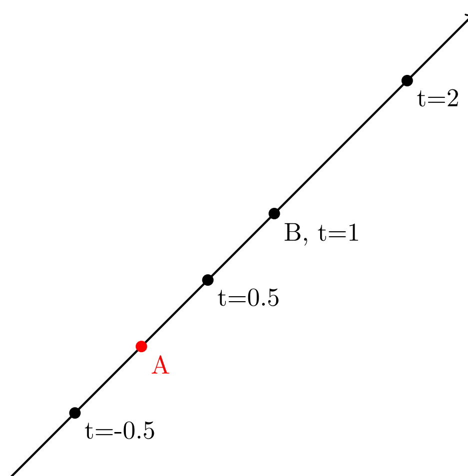
Cílem není vyjádřit bod C na rovině AB, ale na přímce AB, která reprezentuje paprsek. K tomu je potřeba upravit parametry pro lineární kombinaci tak, aby bod C ležel pouze na této přímce. Toho se dá dosáhnout tím, že součet parametrů využitých v lineární kombinaci bude roven jedné. Na obrázku 2.1 je znázorněna upravená varianta pro parametr t, který může nabývat pouze hodnot v intervalu $< 0, 1 >$.

Parametrické vyjádření paprsku se získá tak, že se zvolí bod A jako počáteční bod paprsku, dále směr, kterým se bude paprsek šířit jako vektor $B - A$ a parametr t bude moci nabývat libovolných hodnot. Pro kladné hodnoty parametru t bude bod C ve směru šíření



Obrázek 2.1: Vlevo je znázorněná lineární kombinace dvou vektorů v rovině reprezentujících body A a B. Vpravo je upravený tvar pro lineární kombinaci dvou vektorů, kde parametr t může nabývat pouze hodnoty v intervalu $< 0, 1 >$.

paprsku z bodu A, pro záporné hodnoty bude za počátečním bodem A, viz obrázek 2.2. Další výhodou této reprezentace je možnost určit maximální délku paprsku pomocí parametru t, protože nekonečně dlouhé paprsky jsou často z výpočetního hlediska nevyhovující.



Obrázek 2.2: Parametrické vyjádření paprsku s počátkem v bodě A, směrovým vektorem $B - A$ a různými hodnotami parametru t, určujícími aktuální pozici na paprsku.

2.2 Druhy ray tracingu

V roce 1980 byly v oblasti ray tracingu tři hlavní nevyřešené problémy – jak dosáhnout stínů, jak dosáhnout reflexí a jak dosáhnout refrakcí. Využíval se *ray casting* [13] neboli vržení paprsku, tedy metoda, která šířila paprsek k nejbližšímu objektu a následně vyhodnotila v místě dopadu paprsku barvu, která se vykreslila na obrazovku. Nejvyužívanějším osvětlovacím modelem byl Phongův osvětlovací model [12]. Problém byl v tom, že byl empirický a nedokázal simulovat některé fyzikální efekty, například lesklé odrazy. Tato sekce popisuje, jak se následně metody a osvětlovací modely pro ray tracing vyvíjely až do dnešní doby.

Whittedův ray tracing

Následující sekce vychází z článku *An Improved Illumination Model for Shaded Display* [17]. Turner Whitted v něm představil vylepšený osvětlovací model, který dokázal řešit všechny

tři výše zmíněné problémy. Jeho algoritmus se liší od ray castingu především tím, že nekončí v momentě, kdy paprsek protne nejbližší těleso ve scéně, ale z místa protnutí vysílá další paprsky. Jak lze vidět na obrázku 2.3, paprsky se z místa protnutí šíří do každého světelného zdroje nebo ve směrech $\vec{R}$ a $\vec{P}$, dle povrchu. Paprsky šířené ke světelným zdrojům se nazývají stínové a slouží k určení, zda je dané místo osvětleno nebo je zastíněno. Pokud stínový paprsek protne nějaký objekt scény dříve než světelný zdroj, je dané místo zastíněno. Další dva paprsky se nazývají sekundární paprsky. Paprsky šířené ve směru $\vec{R}$ slouží k zjištění odrazů, jsou to tedy odražené paprsky. Směr těchto paprsků se dá zjistit jednoduše na základě zákona o odrazu vlnění – úhel odrazu je roven úhlu dopadu. A konečně paprsky šířené ve směru $\vec{P}$ slouží k zjištění refrakcí, říká se jim lomené paprsky. Směr těchto paprsků se počítá na základě Snellova zákona, který popisuje šíření vlnění přecházejícího z jednoho prostředí do jiného, viz rovnice 2.1:

$$\eta \cdot \sin\theta = \eta' \cdot \sin\theta', \tag{2.1}$$

kde θ a θ' jsou úhly k normálám a η a η' jsou indexy lomu obou prostředí, které je zapotřebí znát (například index lomu vzduchu je 1.0, index lomu vody je 1.333, atd.). Whittedův vylepšený model pro výpočet intenzity světla je dán rovnicí 2.2:

$$I = I_a + k_d \sum_{j=1}^{Is} (\vec{N} \cdot \vec{L}_j) + k_s S + k_t T, \tag{2.2}$$

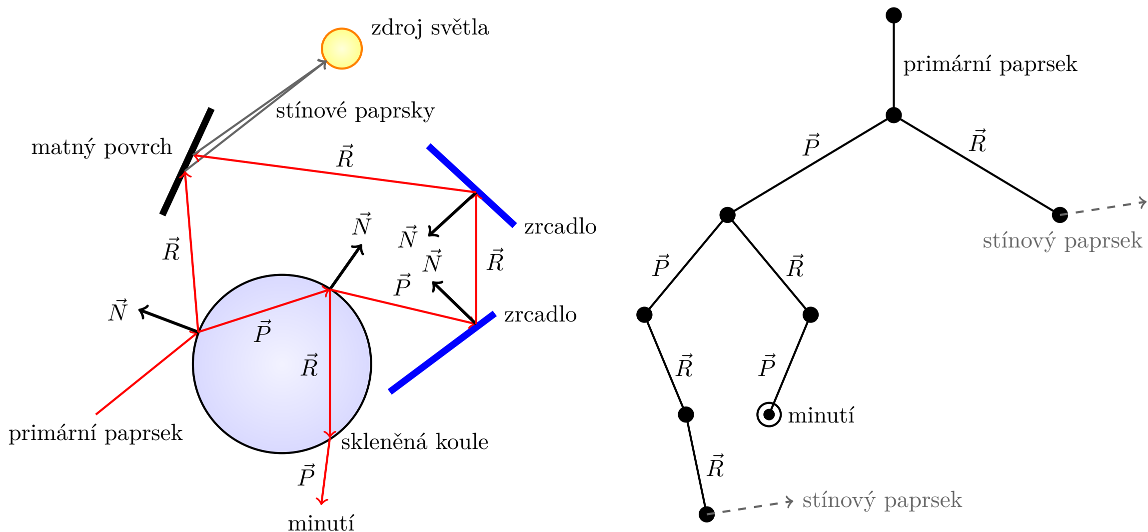
kde k_d, k_s a k_t jsou koeficienty pro výpočet rozptýleného světla, odrazu a lomu, I_a je intenzita ambientního světla, $\vec{L}_j$ je směrový vektor k j-tému zdroji světla, $\vec{N}$ je normála na povrchu v místě protnutí, S je intenzita světla navrácená z paprsku pro reflexi, T je intenzita světla navrácená z paprsku pro refrakci, Is je počet světelných zdrojů ve scéně.

Tento algoritmus se stal základem rekurzivního sledování paprsků a přiblížil se fyzikálně založenému vykreslování. To se podepsalo na náročnosti výpočtů. V tehdejší době trvalo renderování jednoho fotorealistického obrázku desítky minut. V dnešní době je už možné vykreslovat pomocí tohoto algoritmu v reálném čase, viz článek [9]. K tomu je však nutné nastavit maximální hloubku rekurze. V prostředí se spoustou průhledných objektů by bez tohoto omezení rostl počet generovaných paprsků exponenciálně, z každého nového místa protnutí by byly vygenerovány další dva nové sekundární paprsky – lomený a odražený. Ke znázornění generování sekundárních paprsků se často využívá tzv. *tree of rays* neboli strom paprsků. Na něm je vidět, jaké sekundární paprsky se generovaly v jednotlivých hloubkách rekurze. Na obrázku 2.3 lze vidět generování paprsků a odpovídající strom paprsků.

Cookův Ray tracing

Následující sekce vychází z článku *Distributed ray tracing* [2]. Tento článek z roku 1984 umožňuje použití dalších efektů, kterých Whittedův ray tracing není schopen dosáhnout. Jedná se především o *shadow penumbras* (měkké stíny), *glossy reflections* (neostré odrazy), *depth of field* (hloubka ostrosti), *motion blur* (rozmazání pohybu) a *antialiasing*. Právě řešení antialiasingu umožňuje dosáhnutí dalších efektů. Místo vržení jednoho paprsku daným pixelem se jich vrhne více, a to na základě distribuční funkce. Směry vržených paprsků jsou náhodné. K výpočtu se využívá metoda *Monte Carlo*.

Celková intenzita světla odrážející se od povrchu pod úhlem (ϕ_r, θ_r) je dána jako integrál nad funkcí osvětlení L, což je dopadající intenzita světla pod úhlem (ϕ_i, θ_i), a funkcí



Obrázek 2.3: Obrázek vlevo znázorňuje rekurzivní generování paprsků z místa průniku s objekty. Primární paprsek vychází z kamery ve směru jednoho z pixelů plátna, protíná skleněnou kouli a následně se generují dva paprsky – lomený $\vec{P}$ a odražený $\vec{R}$. Tyto paprsky dále procházejí scénou a generují další paprsky, dokud není dosaženo maximální hloubky rekurze nebo dokud u všech paprsků nedojde k *minutí*, což znamená, že paprsek neprotíná žádné těleso ve scéně. Následně se zpětně vyhodnotí výsledná barva pixelu. Z důvodu přehlednosti nejsou v obrázku zaznačeny všechny generované paprsky. Obrázek vpravo znázorňuje *tree of rays* vytvořený z obrázku vlevo. Přerušovanou čarou jsou naznačeny stínové paprsky ke světelným zdrojům.

odrazivosti R, viz rovnice 2.3. Přesné řešení tohoto integrálu by bylo výpočetně náročné, a proto se zde využije náhodné vzorkování. Aritmetickým průměrem všech vzorků se dá přiblížit přesnému výsledku.

$$I(\phi_r, \theta_r) = \int_{\phi_i} \int_{\theta_i} L(\phi_i, \theta_i) R(\phi_i, \theta_i, \phi_r, \theta_r) d\phi_i d\theta_i \tag{2.3}$$

Neostrých odrazů je dosaženo tak, že místo vržení jednoho paprsku ve směru dokonalého odrazu se jich vrhne několik v okolí paprsku s dokonalým odrazem. Tímto způsobem se získají i odlesky. Pokud odražený paprsek protne světelný zdroj, přičte jeho intenzitu k lesklé složce.

Na světelné zdroje se při distribuovaném ray tracingu nenahlíží jako na body, ale jako na tělesa, typicky koule. Bodové světelné zdroje jsou schopny produkovat pouze ostré stíny. Z bodu na objektu se vrhne paprsek ke světelnému zdroji a pokud cestou neprotne jiné těleso, bod na objektu je osvětlen, v opačném případě není. Distribuovaný ray tracing vrhá několik paprsků směrem ke kouli, která reprezentuje zdroj světla, a hodnota zastínění se vypočítá jako poměr paprsků, které dorazily až ke světelnému zdroji, a počtu celkově vržených paprsků. Tento způsob dokáže produkovat měkké stíny.

K dosažení co nejvíce realistické scény vytváří distribuovaný ray tracing také hloubku ostrosti. Bez ní jsou všechny objekty scény perfektně ostré, což neodpovídá reálnému zobrazení. Hloubka ostrosti udává rozsah vzdáleností ve scéně, v němž se objekty zobrazí dostatečně zaostřené. Tento rozsah je kolem ohniskové roviny, která udává množinu perfektně

zaostřených bodů. Hloubky ostrosti se dosáhne přidáním čočky před plátno, viz obrázek 2.4. Průměr čočky je F/n a je zaostřená v ohniskové vzdálenosti P. Vzdálenost plátna od čočky V_P je dána vztahem 2.4

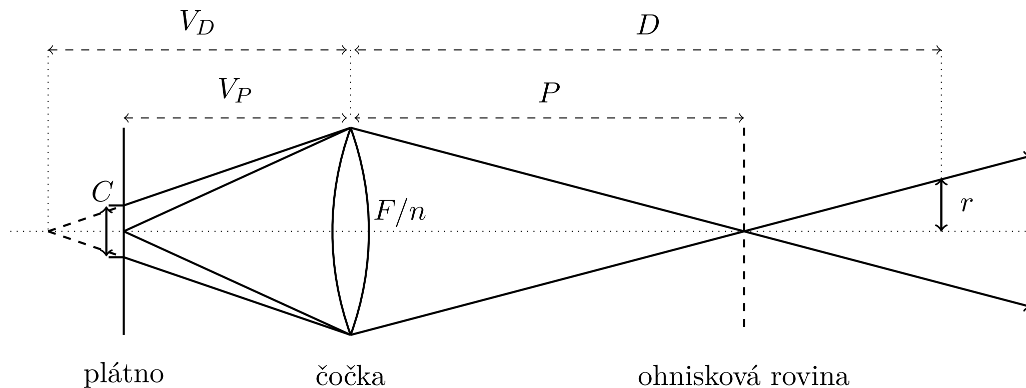
$$V_P = \frac{FP}{P - F} \text{ pro } P > F. \tag{2.4}$$

Body scény ve vzdálenosti D od čočky se zobrazí až za plátnem, ve vzdálenosti V_D od čočky. To znamená, že protínají plátno nikoli v jednom bodě (což by znamenalo ostré zobrazení), ale ve více bodech, a tvoří tak zvaný *circle of confusion* (kruh zmatku). Tím, že se jeden bod scény promítne na plátně jako kruh, dochází k jeho rozmazání na všechny body plátna v daném kruhu. Čím je bod ve scéně dál za ohniskovou rovinou, tím větší kruh na plátně vytvoří. Průměr C vzniklého kruhu je dán vztahem 2.5

$$C = |V_D - V_P| \frac{F}{nV_D}. \tag{2.5}$$

Pro každý bod na plátně je tedy nutné vrhat více než jeden paprsek. Vrhané paprsky dohromady tvoří tvar kuželu. Mějme bod I na plátně. Průměr kuželu, ve kterém se vrhají paprsky z bodu I, je ve vzdálenosti D od čočky dán vztahem 2.6. Všechny body na nebo uvnitř kuželu ovlivňují barvu výsledného bodu I, protože je bod I uvnitř nebo na hraně jejich circle of confusion.

$$r = \frac{1}{2} \frac{F}{n} \frac{D - P}{P}. \tag{2.6}$$



Obrázek 2.4: Po zavedení čočky je každý bod scény, který neleží v ohniskové rovině, zobrazen na plátně jako kruh, kterému se říká *circle of confusion*. Jeho velikost závisí na vzdálenosti bodu a vlastnostech čočky. Body v ohniskové rovině jsou na plátně zobrazeny také jako body, takže jsou vidět perfektně ostře. Obrázek překreslen z [2].

Posledním zmíněným efektem je motion blur neboli rozmazání pohybu. Toho se dosáhne distribucí vzorků v čase. K tomu je nezbytná schopnost spočítat přesnou pozici objektů v daném čase – není potřeba přidávat další paprsky, ale stačí scénu a pozice objektů postupně aktualizovat po malých částech. Tato metoda potřebuje velmi dobrý antialiasing, jinak by docházelo k nežádoucím efektům.

Kajiyaův Ray tracing

Následující sekce vychází z článku *The rendering equation* [5]. Kajiyaův ray tracing, často také nazývaný jako *Monte Carlo ray tracing* nebo *path tracing*, se na první pohled velmi podobá distribuovanému ray tracingu, je to jeho zobecněná verze. Metoda vychází ze zobrazovací rovnice 5.4

$$I(x,x') = g(x,x')[\epsilon(x,x') + \int_S \rho(x,x',x'')I(x,x'')dx''], \tag{2.7}$$

kde $I(x,x')$ je celková vyslaná zář z bodu x' do bodu x, $g(x,x')$ je geometrický člen, $\epsilon(x,x')$ je zář emitovaná zdrojem z bodu x' do bodu x a $\rho(x,x',x'')$ je zář z bodu x'' do bodu x odražená přes bod x'. To, jak se paprsky odráží od matných povrchů, je dáno funkcí distribuce obousměrného rozptylu, zkráceně *BSDF* (*bidirectional scattering distribution function*). BSDF modeluje vlastnosti materiálu, vstupem jsou dva úhly – úhel dopadajícího paprsku a úhel odraženého paprsku. Pomocí BSDF se vypočítá poměr mezi příchozí a odchozí světelnou energií, čímž ovlivní váhu jednotlivých paprsků. V článku [5] není ještě o BSDF zmínka, ale obecně se při path tracingu používá.

Celý algoritmus funguje tak, že se vrhne paprsek skrz náhodný bod uvnitř pixelu (případně více paprsků, které se pak zprůměrují), při dopadu se paprsek odrazí v několika směrech. Na rozdíl od distribuovaného ray tracingu, kde se paprsky vrhají jen v oblasti kuželu, se zde vrhají v oblasti celé hemisféry. Směry paprsků se získají stochasticky metodou Monte Carlo. Tímto se dosáhne efektu globální iluminace, protože se paprsky odrážejí v oblasti celé hemisféry. Váha jednotlivých paprsků se získá přes BSDF. Dalších efektů zmíněných u distribuovaného ray tracingu se dá dosáhnout stejným způsobem – přidáním čočky (hloubka ostrosti), času (motion blur) atd. Path tracing vyžaduje nadvzorkování v rámci jednoho pixelu, ale i tak dochází často ke vzniku šumu, proto se path racing doplňuje vhodným algoritmem pro *denoising* (odstranění šumu) [7].

2.3 Grafické efekty

Tato sekce stručně popisuje několik vybraných grafických efektů, kterých lze pomocí ray tracingu dosáhnout.

Ostré stíny

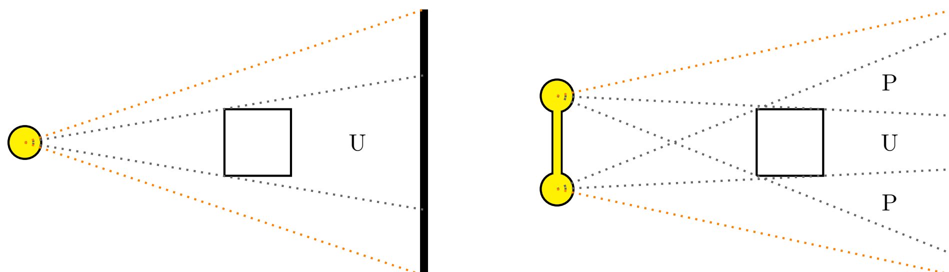
Stíny patří při vykreslování k důležitým efektům, protože dávají scéně mnohem lepší atmosféru a pozorovatel snadněji odhadne velikosti a umístění objektů. První typ stínů, takzvané ostré stíny, vychází z toho, že je scéna osvětlena bodovým světlem. Světelné paprsky k pozorovanému místu buď dopadají, nebo ne. Tím dochází k tomu, že má stín ostré hrany, není tam žádný postupný přechod. Tento způsob vykreslování stínů neodpovídá realitě, protože nebere v potaz měkké stíny, které jsou popsány níže. Ukázka ostrých stínů je na obrázku 2.5.

Efekt ostrých stínů je výpočetně poměrně jednoduchý, jelikož stačí vyslat z pozorovaného místa jen jeden paprsek k bodovému světlu. Pokud po cestě protne jiný objekt, je pozorované místo zastíněno, jinak je osvětleno.

Měkké stíny

V reálném světě nebývají světla bodová, ale plošná. Taková světla vytváří tři druhy stínů – *umbra*, *penumbra* a *antumbra*, viz obrázek 2.5. Umbra je část stínu, která je před celým plošným světlem plně zakryta. Penumbra je taková část stínu, která je zakryta pouze částí plošného světla, takže je stín jemnější. Antumbra je oblast, kde se spojují dvě penumbry.

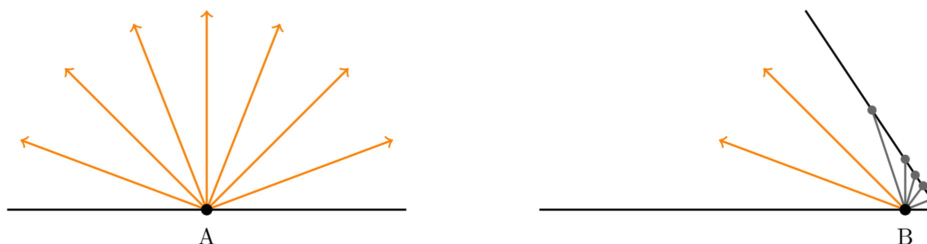
Dosáhnutí měkkých stínů je výpočetně obtížnější než těch ostrých. Nestačí zde vyslat pouze jeden paprsek, ale je jich potřeba několik na různá místa plošného světla. Následně se spočítá podíl paprsků, které dosáhly světla, a celkového počtu vyslaných paprsků. Podle toho se vypočítá zastínění pozorovaného místa.



Obrázek 2.5: Rozdíl mezi bodovým světlem a plošným světlem. U bodového světla vzniká pouze část umbra (označena U), a proto jsou stíny ostré. U plošného světla vzniká i část penumbra (označena P), kde je postupný přechod z úplně zastíněné do nezastíněné části, a stíny jsou měkké.

Ambient occlusion

Tento efekt slouží k vytvoření jemnějších stínů ve scéně a spadá do skupiny metod aproximujících globální osvětlení. Díky své jednoduchosti patří k velmi oblíbeným efektům. Z pozorovaného místa na objektu se vyšlou paprsky ve všech směrech přes polokouli danou normálovým vektorem $\vec{n}$. Paprsky, které nic neprotnou, nemají na osvětlení vliv. Paprsky, které protnou jiný objekt indikují, že je z daného směru místo zastíněno. Na základě poměru těchto paprsků se určí osvětlení daného místa. K místům, která jsou ve velké míře obklopena jinými objekty, nedopadá tolik globálního světla. Ukázka vysílání paprsků pro výpočet tohoto efektu je na obrázku 2.6.



Obrázek 2.6: Ukázka výpočtu efektu ambient occlusion. Vlevo leží bod A na rovném povrchu, vyslané paprsky neprotínají žádný objekt, a tudíž k místu dopadne více globálního světla než k bodu B vpravo, který leží v rohu a pouze dva vyslané paprsky neprotnou žádný objekt.

2.4 DirectX 12 vykreslovací řetězec pro ray tracing

Tato sekce vychází z kurzů *Úvod do DirectX ray tracingu a Nvidia RTX* [15, 18]. Tradiční vykreslovací řetězec pro rasterizaci obsahuje množství různých typů shaderů[1], například vertex shader, fragment shader, geometry shader a další. Vykreslovací řetězec pro ray tracing uvádí pět nových shaderů:

- ray generation shader,
- intersection shader,
- miss shader,
- closest-hit shader,
- any-hit shader.

Ray generation shader je vstupním bodem celého řetězce, je vždy spuštěn jako první a slouží ke generování primárních paprsků. K tomu se využívá funkce `TraceRay()`. Tato funkce spustí proces trasování paprsku, který podle výsledku aktivuje další shadery. Nejprve je nutné nastavit parametry paprsku, které jsou popsány v sekci 2.1. Typicky se paprsek šíří z kamery skrz daný pixel obrazovky a hledá se průsečík s objekty ve scéně. Paprsků se může skrz jeden pixel šířit i více. Důležité je také inicializovat *ray payload*, což je datová struktura, do které se ukládají informace o paprsku (výsledná barva, hloubka zanoření a další).

Pokud je při trasování paprsku v rámci akcelerační struktury nalezen potenciální průnik s tělesem, nastane jedna s těchto možností. První možnost je, že se provede výchozí *ray-triangle intersection test*, který ověří průnik paprsku s daným objektem na úrovni trojúhelníků. Druhou možností je, že se aktivuje naimplementovaný *intersection shader*, který ověří, zda došlo k průniku s určitým typem geometrie (například s koulí). Tento shader je dobrovolný.

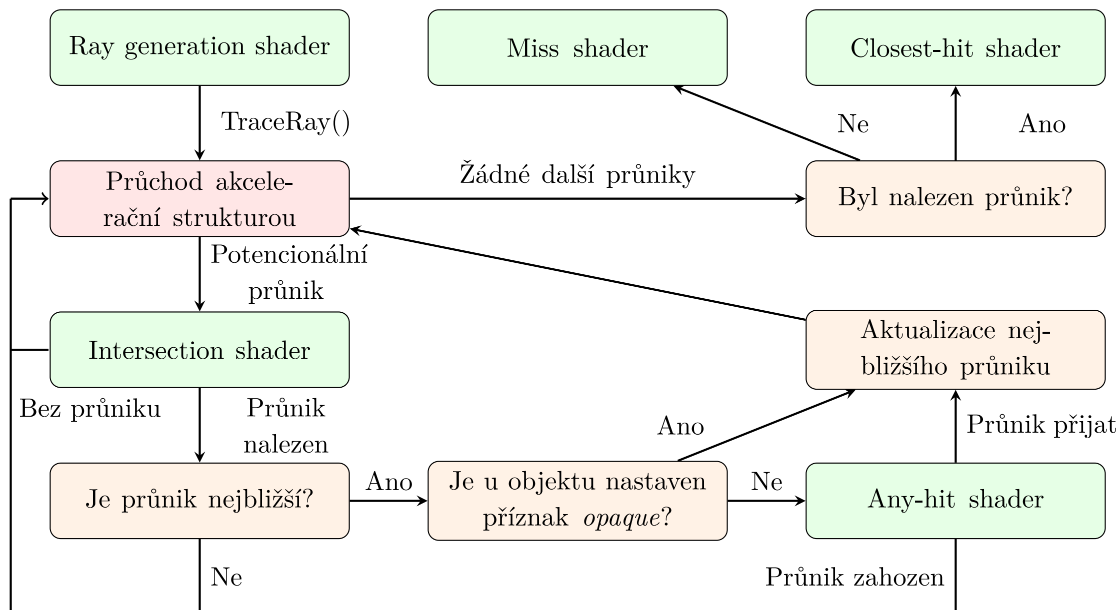
Jakmile je nalezen průnik s tělesem, aktivuje se *any-hit shader*, pokud existuje. V tomto shaderu se rozhodne, zda se daný průnik přijme nebo zahodí. Tento shader je vhodný pro *alpha testing*, který ověřuje, zda je textura protnutého tělesa v místě průniku průhledná nebo ne. Pokud je průhledná, průnik s tělesem se ignoruje. Tento shader je také dobrovolný a při jeho absenci se automaticky přijmou všechny průniky k dalšímu vyhodnocení.

Když se dokončí trasování paprsku, zjistí se, zda byl nalezen průnik s nějakým tělesem. Pokud průnik nalezen byl, aktivuje se *closest-hit shader*, jehož funkce se dá přirovnat k fragment shaderu. Typicky se zde vyhodnocuje výsledná barva na základě materiálu tělesa, nanesené textury a dalších vlastností materiálu. Také se zde kontroluje, zda nedošlo k dosažení maximální hloubky rekurze. Pokud ne, je možné v tomto shaderu dále volat funkci `TraceRay()` a šířit další paprsky. Tento shader je povinný pro všechna tělesa ve scéně.

Pokud při trasování paprsku nebyl nalezen žádný průnik, aktivuje se *miss shader*. V tomto shaderu se může například vyhodnotit barva pozadí, případně je možné, stejně jako z closest-hit shaderu, šířit další paprsky. Tento shader musí být pro využitý vykreslovací řetězec definovaný právě jednou, stejně jako ray generation shader. Celé schéma DirectX 12 vykreslovacího řetězce je znázorněno v obrázku 2.7.

Vykreslovací řetězec obsahuje zkompilované shader programy a informace o tom, jak si jednotlivé shadery vyměňují data po spuštění. Pro propojení s objekty ve scéně se využívá tabulka shaderů. Objekty scény jsou reprezentovány akcelerační strukturou.

[1] Shader – program na grafické kartě, řídí chod programovatelných částí vykreslovacího řetězce.



Obrázek 2.7: Schéma znázorňující DirectX 12 vykreslovací řetězec. Zeleně jsou vybarveny jednotlivé shadery, oranžově jsou vybarveny bloky s podmínkami a červeně je vybarven průchod akcelerační strukturou, kde se hledají potenciální průniky paprsku s objekty ve scéně.

Akcelerační struktura

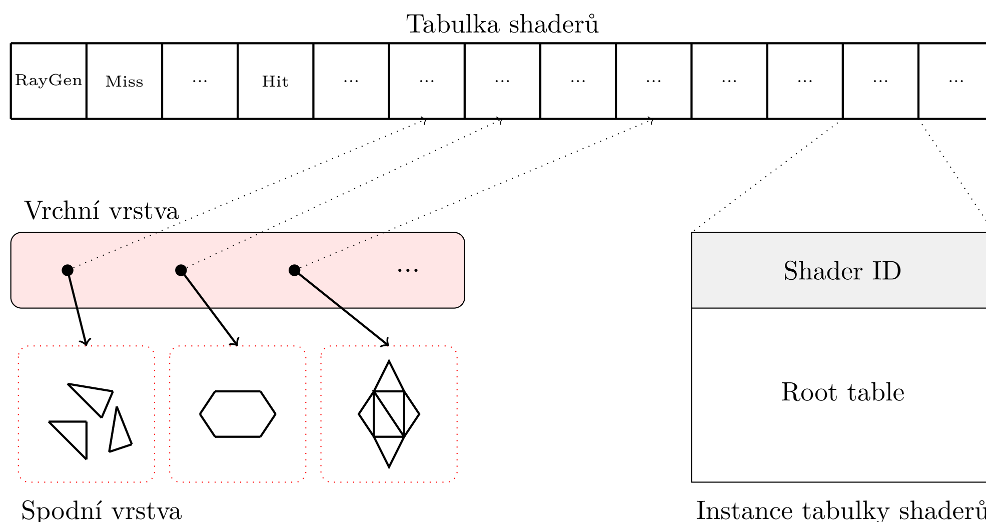
Důležitou součástí vykreslovacího řetězce je akcelerační struktura a její efektivní prohledávání. Cílem je redukovat počet testů pro nalezení průniků s šířeným paprskem na minimum. Nvidia RTX nabízí vysoce optimalizovanou konstrukci a prohledávaní akcelerační struktury v reálném čase [15]. Ke konstrukci slouží funkce `BuildRaytracingAccelerationStructure()`, která nabízí různé optimalizační možnosti pro statické či animované scény.

Akcelerační struktura je rozdělena do dvou vrstev – vrchní a spodní. Spodní vrstva reprezentuje geometrická primitiva, typicky trojúhelníky. Tato primitiva jsou reprezentována *vertex* a *index bufferem*. Vrchní vrstva slouží k referenci instancí ve spodní vrstvě. Zároveň nese informace o transformační matici pro transformaci primitiv do prostoru scény a offset do tabulky shaderů pro nalezení potřebných shaderů a dat k vykreslení daného objektu [15]. Na instance spodní vrstvy se lze odkazovat vícekrát, stačí mít různé transformační matice ve vrchní vrstvě. Tímto způsobem se redukují duplicity primitiv ve spodní vrstvě a dosáhne se lepšího výkonu – čím méně primitiv ve spodní vrstvě, tím lepší výkon [6]. Dvouvrstvá hierarchie akcelerační struktury je znázorněna na obrázku 2.8.

Tabulka shaderů

Tabulka shaderů slouží k propojení shaderových programů, které jsou součástí vykreslovacího řetězce, a akcelerační struktury, kde jsou uloženy informace o objektech. Tabulka obsahuje vždy právě jeden ray generation shader a alespoň jeden miss shader. Za nimi následují any-hit a closest-hit shadery. Aplikace většinou obsahují v tabulce shaderů velké množství closest-hit shaderů, protože každý objekt může mít jinak specifikované chování při průniku s paprskem. Předem se nedá určit, které objekty budou protnuty, a tudíž je zapotřebí mít v tabulce informace o všech shaderech a jejich zdrojích. Typicky existuje pro

jeden geometrický objekt jeden záznam v tabulce [15]. Záznam v tabulce nemusí obsahovat informace pouze o jednom shaderu, ale může obsahovat informace o skupině shaderů (*hit group*), které jsou určeny pro daný objekt. Tabulku shaderů, její instance a propojení s akcelerační strukturou lze vidět na obrázku 2.8.



Obrázek 2.8: Obrázek znázorňující propojení akcelerační struktury s tabulkou shaderů. Instance vrchní vrstvy nesou odkaz na primitiva ve spodní vrstvě, stejně tak nesou offset do tabulky shaderů. Instance tabulky shaderů je tvořena identifikátorem shaderu, který aplikace získá po zkompilování, a tak zvanou *root table*[3], která slouží k definování zdrojů, které shader využívá.

2.5 Ray tracing v reálném čase

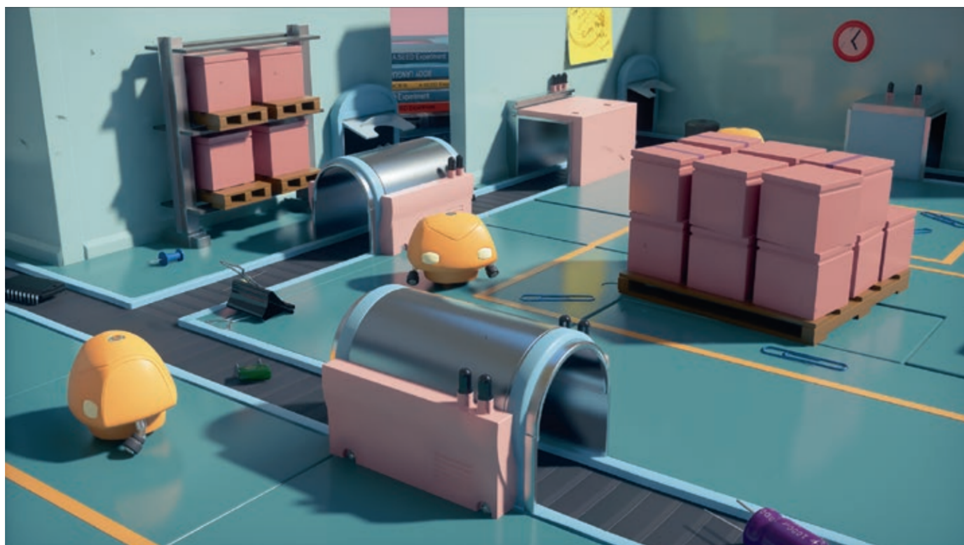
Tato sekce vychází z článku *NVDIA Turing GPU Architecture* [8]. Ray tracing je výpočetně velmi náročná technika renderování, a proto se pro real-time aplikace, hlavně v herním průmyslu, využívala rasterizace. K umožnění využití ray-tracingu v real-time aplikacích bylo potřeba téměř desetileté spolupráce mezi výzkumnými týmy z Nvidie, GPU návrháři a dalšími softwarovými firmami. Jejich spolupráce přinesla hardwarově akcelerované enginy RT Cores, které v kombinaci s RTX technologií a dalšími algoritmy umožňují real-time ray tracing.

Hybridní vykreslování

Jednou z možností, jak využít ray tracing v real-time aplikacích, je hybridní vykreslování. Tento přístup kombinuje využití rasterizace v místech, kde je efektivní, a ray tracing pro efekty, ve kterých rasterizace nedosahuje tak kvalitních výsledků (reflexe, refrakce a stíny). Výsledky hybridního vykreslování lze vidět například na modelu PICA PICA, viz obrázek 2.9.

Rasterizace je v tomto typu vykreslování využívaná hlavně pro zjištění viditelnosti objektů, protože dokáže zpracovat velmi rychle velké množství trojúhelníků. Je možné efek-

[3]Možnosti nastavení *root table* k nalezení na `https://docs.microsoft.com/en-us/windows/win32/direct3d12/specifying-root-signatures-in-hlsl`.



Obrázek 2.9: Snímek z modelu PICA PICA, který využívá hybridní vykreslování. Převzato z [3].

tivně využít *z-buffer*, který do jisté míry nahrazuje primární paprsky. Následně se pomocí ray tracingu šíří sekundární paprsky, kterými se dosáhne přesných reflexí, refrakcí a stínů. Tímto způsobem se sníží počet paprsků a tím se zvýší efektivita vykreslování.

Denoising

Aby bylo možné využívat ray tracing v real-time aplikacích, je nutné omezit počet paprsku na potřebné minimum. Tím dochází ke vzniku nežádoucích artefaktů, mezi které patří především vznik šumu. Proto se denoising stal jedním z klíčových směrů ve vývoji ray tracingu. Klasickým filtrům pro odstranění šumu dnes konkurují denoisery s umělou inteligencí.

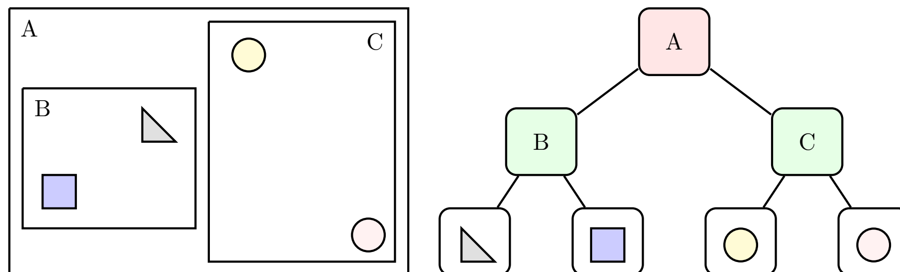
BVH

Další možností pro zrychlení aplikace je snížení počtu testů pro průnik paprsku s objekty. K tomu je nutné využít vhodnou akcelerační strukturu. Takovou strukturou je *BVH* (bounding volume hierarchy). Jedná se o stromovou akcelerační strukturu, která funguje na principu obalových těles (bounding box).

Kořen stromu reprezentuje jedno velké obalové těleso, které se dále dělí na menší obalová tělesa. Ty jsou postupně uložena v uzlech stromu. Průchod takovým stromem začíná od kořene a postupně se testuje průnik paprsku s obalovými tělesy v uzlech, až dokud se nenarazí na geometrická primitiva, která jsou uložena v listech stromu. Tímto způsobem se při hledání průniku s paprskem nemusí kontrolovat každý objekt scény, což by trvalo velmi dlouho, ale pouze některé objekty. Na obrázku 2.10 je znázorněna jednoduchá BVH akcelerační struktura.

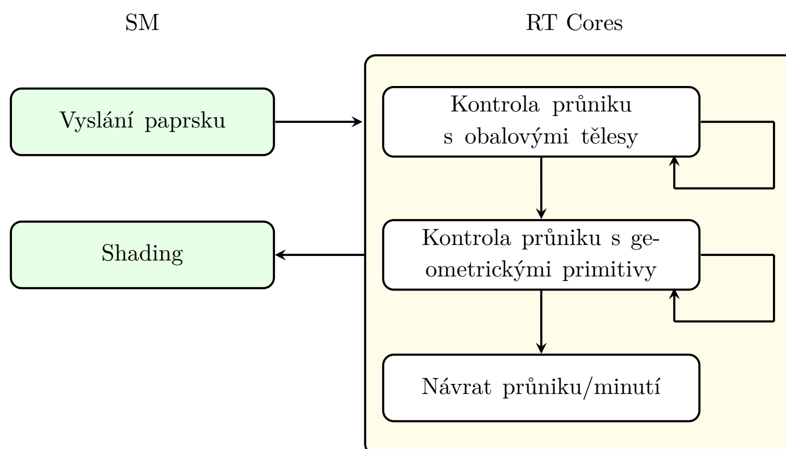
RT Cores

Zavedením BVH akcelerační struktury se zrychlí hledání průsečíku s objekty, ale ani to samo o sobě nestačí k docílení real-time ray tracingu. Je potřeba hardwarové podpory.



Obrázek 2.10: Ukázka jednoduché BVH akcelerační struktury ve 2D prostoru. Červeně je zobrazen kořenový uzel, zeleně uzly stromu a v listech, které mají bílou barvu, jsou uložena jednotlivá geometrická primitiva.

Firma Nvidia uvedla grafické karty s architekturou *Turing*, které obsahují nový prvek – *RT Cores*. Ty mají za úkol odlehčit *SM* (streaming multiprocessors), které se mohou věnovat jiným úkolům, například *shadingu*. Tyto multiprocesory by byly zahlceny hledáním průniku pro každý paprsek scény. RT Cores mají dvě jednotky – jedna se stará o hledání průniku s obalovými tělesy, ta druhá ověřuje průniky s geometrickými primitivy. SM mají za úkol vyslat paprsek, RT Cores projde akcelerační strukturu a vrátí zpátky SM výsledek, zda došlo k průniku či nikoliv. Tato komunikace je znázorněna na obrázku 2.11.



Obrázek 2.11: Ukázka rozdělení práce mezi SM a RT Cores, kde hlavním úkolem RT Cores je odlehčit SM od hledání průsečíků. Obrázek překreslen z [8].

Kapitola 3

Voxelové scény

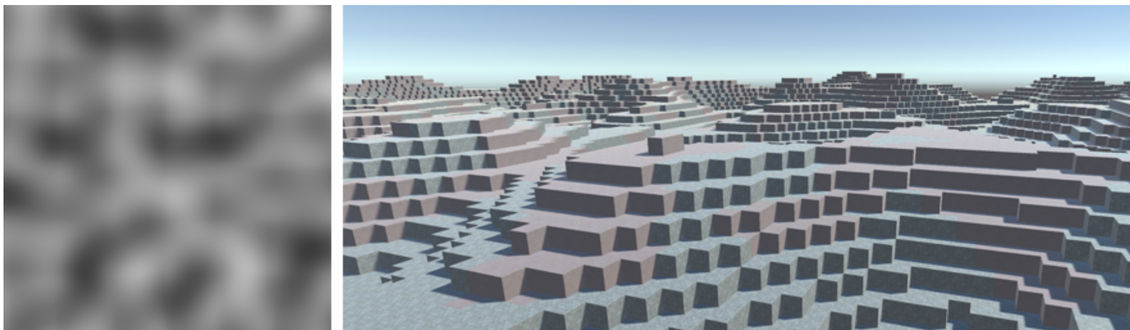
Tato kapitola se zabývá voxelovými scénami. Pojmem voxel se označuje hodnota v pravidelné mřížce v 3D prostoru, typicky má tvar krychle. Jedná se o analogii pixelu ve 2D prostoru. Voxely se často používají v lékařství, reprezentují se pomocí nich například výsledky tomografu. Populární jsou také v herním průmyslu a to hlavně díky oblíbené voxelové hře Minecraft. S voxelovými scénami se často pojí procedurální generování prostoru, které je popsáno v sekci 3.1. Z důvodu velké rozsáhlosti scén je důležitá efektivní reprezentace voxelů ve scéně. Různé druhy reprezentace jsou uvedeny v sekci 3.2.

3.1 Procedurální generování terénu

Pod pojmem procedurální generování se dá představit algoritmické vytváření obsahu. Toto generování je většinou založeno na generátoru náhodných čísel. Náhodná čísla se dají rozdělit na dva druhy – opravdu náhodná čísla a pseudonáhodná čísla. Opravdu náhodných čísel není tak jednoduché dosáhnout, počítač je založen na zpracovávání přesných instrukcí, a proto je pro opravdu náhodná čísla potřebný specializovaný hardware, který měří nějaký fyzikální jev, o kterém se dá prohlásit, že je náhodný. Z těchto důvodů se většinou používají pseudonáhodná čísla. Jedná se o posloupnost čísel, která je vygenerovaná podle nějakého vzorce a na první pohled se zdá být opravdu náhodná. Aby tato posloupnost nebyla pro jeden generátor vždy stejná, využívá se tak zvaný *seed* (semínko), což je hodnota, která se vloží do generátoru a ten na jejím základě změní výslednou posloupnost. Pro stejný seed tedy dostaneme vždy stejnou sekvenci čísel, což se využívá například ve výše zmíněné hře Minecraft, kde si hráči mohou pomocí seedu vygenerovat oblíbenou mapu. Vygenerovaná sekvence čísel se nazývá šum.

Perlinův šum

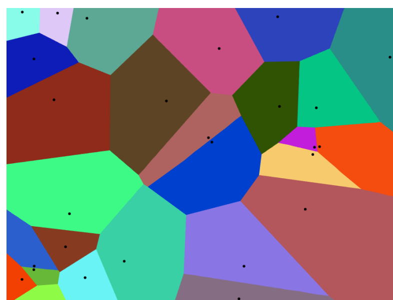
Samotný náhodný šum pro procedurální generování terénu nestačí. Aby se terén podobal reálnému terénu, je potřeba, aby byly hodnoty šumu na sobě závislé. Pak se bude měnit výška terénu postupně a terén bude vypadat organicky. K tomu se využívá Perlinův šum, který vynalezl Ken Perlin [10, 11]. K tvorbě Perlinova šumu se využívá několika funkcí, které se liší frekvencemi a amplitudami. Počet těchto funkcí je dán počtem oktáv, typicky mají funkce v každé další oktávě dvojnásobnou frekvenci. Tyto funkce jsou následně sečteny do výsledné komplexní šumové funkce. Na obrázku 3.1 lze vidět ukázku Perlinova šumu a terén vygenerovaný tímto šumem.



Obrázek 3.1: Na obrázku vlevo lze vidět Perlinův šum. Na obrázku vpravo je terén, který je podle tohoto Perlinova šumu vygenerovaný.

Voroného diagram

Tato podsekce vychází z článku [1]. Mějme několik náhodně zvolených bodů ve 2D prostoru, nazvěme je řídící body. Voroného diagramy rozdělí prostor do takových regionů, které obsahují právě jeden řídící bod a všechny ostatní body tohoto regionu mají k danému řídícímu bodu blíž než k jiným řídícím bodům. Tato metoda tedy funguje na principu nejbližších sousedů. Na obrázku 3.2 je takové rozdělení prostoru znázorněno. Jednotlivé vzniklé regiony mají každý svou barvu, řídící body jsou černě. Voroného diagramy slouží k dělení prostoru, ale dají se využít také při procedurálním generování. Díky jejich buněčnému vzhledu se dají využít ke generování textur či terénu, mohou se také kombinovat s Perlinovým šumem.



Obrázek 3.2: Na obrázku lze vidět Voroného diagram, podle kterého se dá generovat terén, který má buněčný vzhled.

3.2 Reprezentace voxelové scény

Při vykreslování voxelových scén je důležité zvolit vhodnou strukturu pro ukládání dat. Pokud by procedurálně vygenerovaný voxelový terén nebyl interaktivní, šlo by se ukládani dat vyhnout. Terén by se pomocí šumové funkce vygeneroval vždy stejně a dále by se do něj nezasahovalo. Přesto by bylo nutné ho pokaždé generovat znovu, což není potřeba pokud se data uloží. Tato práce bude uvažovat s interaktivním voxelovým terénem – voxely se můžou volně přidávat a odebírat. Datová struktura je tedy nezbytná minimálně pro ukládaní změn terénu. Existují dvě často používané struktury – pole a *octree*.

Pole

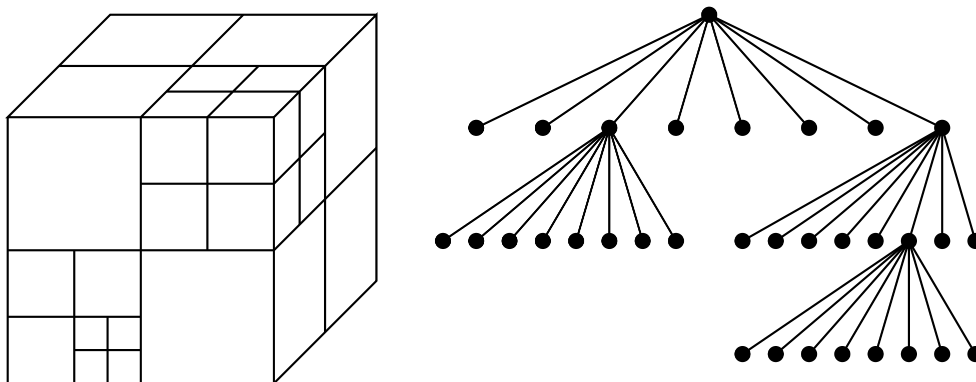
První zmíněnou datovou strukturou je pole. Výhodou této struktury je rychlost. Hledá-li se voxel na pozicích x, y, z, stačí se v trojrozměrném poli podívat na daný index, který odpovídá pozicím voxelu. Nalezení voxelu je tedy v konstantním čase. Zároveň voxely nepotřebují ukládat své souřadnice, protože odpovídají indexům v poli a zbytečně by zabíraly místo.

Nevýhodou této datové struktury je její prostorová složitost. Do pole by se musel ukládat každý voxel scény. Pokud by tyto voxely obsahovaly další informace o materiálu, normálách a tak dále, velikost pole by byla obrovská. Zároveň pole neumožňuje vynechat prázdné části scény, kde nejsou žádné voxely. Z těchto důvodů se častěji využívá *octree*.

Octree

Datová struktura *octree* je analogií ke *quadtree* ve 2D prostoru. Výchozím uzlem je krychle, která reprezentuje celou scénu. Jakmile se do scény přidá voxel, výchozí krychle se rozdělí na osm menších krychlí, z nichž jedna obsahuje vložený voxel. Pokud se do krychle, která už obsahuje nějaký voxel, vloží další voxel, krychle se rozdělí na dalších osm menších. Tyto krychle reprezentují jednotlivé uzly stromu. Každý uzel je tedy buď prázdný nebo obsahuje právě jeden voxel (takový uzel se nazývá list) nebo obsahuje ukazatele na dalších osm uzlů. Tímto způsobem se můžou do scény přidávat další objekty. Výhodou této struktury je možnost reprezentovat velké prázdné oblasti scény pouze jedním uzlem, čímž se značně sníží nároky na paměť. Ukázka struktury octree je na obrázku 3.3.

Nevýhoda oproti poli je v horší časové složitosti. Nalezení náhodného voxelu je v logaritmickém čase.



Obrázek 3.3: Obrázek znázorňující octree. Vlevo je scéna rozdělena krychlemi a vpravo je její reprezentace ve stromové struktuře.

Kapitola 4

Návrh řešení

Tato kapitola popisuje strukturu aplikace, použitý engine[1] a návrh generování voxelového terénu. Pro vytvoření aplikace s ray tracingem je nutné využít API[2] DirectX 12. Přistupovat k tomuto rozhraní lze například skrze některý engine. Jako vývojové prostředí byl zvolen herní engine Unity. Díky vestavěným funkcím pro komunikaci s grafickou kartou se dá zaměřit více na samotný ray tracing a optimalizační techniky pro lepší výkon.

4.1 Unity a SRP

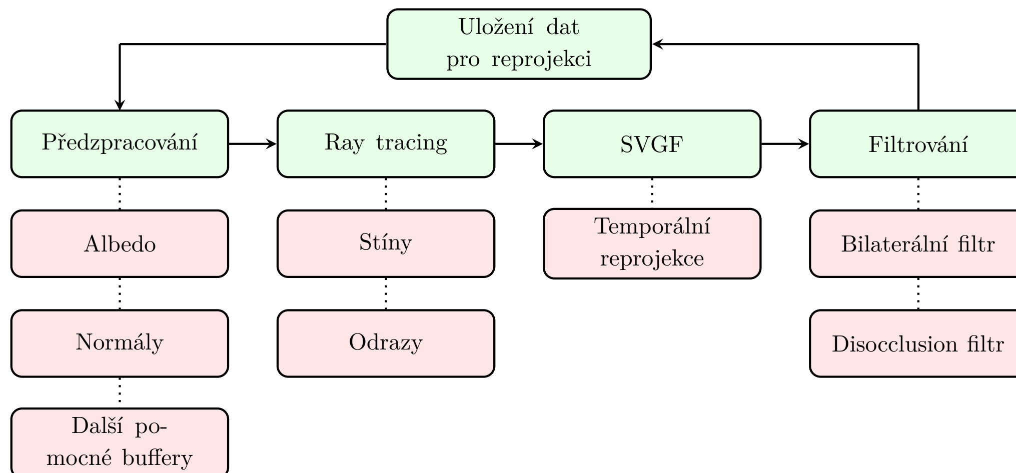
Unity je multiplatformní herní engine pro tvorbu 2D a 3D her. Podpora pro ray tracing je od verze Unity 2019.3, kde se objevily dvě šablony pro vykreslovací řetězec – HDRP[3] a URP[4]. HDRP je šablona určena pro zařízení s velmi dobrým grafickým hardwarem, jako jsou PC nebo Xbox, a je zaměřena na fyzikálně reálné vykreslování. Oproti URP nabízí ray tracing, globální osvětlení, ambient occlusion, odrazy a další efekty. URP využívá několik funkcí podobných těm v HDRP, ale je zaměřena spíše na výkon a multiplatformost, proto není její grafický výstup na úrovni HDRP. Oproti HDRP umožňuje navíc 2D osvětlení. Obě tyto šablony jsou postaveny na SRP – scriptable rendering pipeline (skriptovatelný vykreslovací řetězec). SRP je modulární vykreslovací systém. Umožňuje vývojářům modifikovat vykreslovací řetězec přímo pomocí skriptů. Tímto způsobem se zajistí, že se v pozadí neprovádí žádné operace a výpočty, které by v dané aplikaci nebyly potřebné a zbytečně by snižovaly výkon. Vývojář si všechny moduly řetězce může naprogramovat sám, což je pro tuto práci výhodné. Cílem práce bude vytvořit takový vykreslovací řetězec, který se bude blížit HDRP, ale bude využívat pouze ty moduly, které se hodí pro vykreslování voxelových scén.

4.2 Struktura aplikace

Struktura aplikace musí být taková, aby se pro vykreslování scény využívalo co nejméně paprsků při zachování co největší výstupní kvality. Cílem je dosáhnutí efektů stínů a ambient occlusion. Oba efekty potřebují pro výpočet velké množství paprsků, proto jsou využity techniky pro snížení počtu paprsků na nezbytné minimum. Jedná se především o využití

[1] Engine – jádro aplikace, řídící část softwaru.
[2] API – application programming interface (rozhraní pro programování aplikací).
[3] HDRP – high definition rendering pipeline.
[4] URP – universal rendering pipeline.

dat z předběžného zpracování a denoising. Navrhovaná struktura aplikace je zobrazena na obrázku 4.1.



Obrázek 4.1: Schéma navrhované struktury aplikace. Aplikace je rozdělena do pěti hlavních částí – předzpracování, ray tracing, SVGF, filtrování a uložení dat pro reprojekci. Pod jednotlivými částmi jsou červeně znázorněny kroky, které daná část provádí.

Předzpracování

V této fázi se získávají data a informace o scéně, které se využívají v dalších fázích. Často je tato fáze označována jako *deferred shading* (odložený shading) – nedochází k žádným výpočtům výsledné barvy pixelů, pouze se pomocí jednoduchých shaderů získávají informace o scéně. Většinou se využívá rasterizace, protože je pro tyto zpracování rychlejší než ray tracing. Informace jsou vepsány do textur, které tvoří tak zvaný *G-buffer* (geometry buffer). Důležité informace k dalšímu zpracování jsou albedo barva, normály, identifikátory objektů, materiály, pozice ve světě a hloubka scény.

Ray tracing

V této fázi se šíří paprsky z kamery skrz pixely plátna pomocí funkce `TraceRay()` (popsáno v sekci 2.4). Důležitými parametry v této fázi jsou počet paprsků vyslaných skrz jeden pixel a maximální hloubka rekurze u paprsků. Hloubku rekurze je vhodné upravovat podle rychlosti aplikace. Počet paprsků vyslaných skrz jeden pixel je možné nastavit jako konstantní pro každý pixel, nebo se může využít adaptivní vzorkování [4]. Pomocí adaptivního vzorkování se určí místa, kde je výhodnější vyslat více paprsků. Většinou se jedná o místa, kde je stín nebo odraz světla a je zapotřebí více paprsků k získání lepšího detailu v daném místě. Ray tracing dále slouží k vykreslení stínů či reflexí.

SVGF

SVGF neboli spatio-temporal variance guided filter je metoda, která slouží k rekonstrukci obrazu na základě temporálních dat z předchozích snímků [14] a k filtrování obrazu, které je popsáno níže. Metoda je určena především pro snížení šumu v metodách globálního osvětlení a v jejich aproximacích, což je v této práci přímé sluneční osvětlení, nepřímé osvětlení a ambient occlusion. Principem této metody je promítnutí hodnot stínů z předchozích

snímků na ten aktuální, a to na správné pozice, které aktuálnímu snímku odpovídají. To znamená, že hodnota stínu z předchozího snímku na konkrétním místě na povrchu objektu musí být promítnuta na stejné místo daného objektu v aktuálním snímku. K této reprojekci se používají pomocné buffery – normálový, hloubkový, pohybový a rozdílový. Následně se smíchá hodnota stínu z minulých snímku s tou aktuální. Někdy se může stát, že nastane tak zvaný *ghosting effect*, což znamená, že se hodnoty z minulých snímku nepromítly na správné místo v tom aktuálním. K tomu může dojít hlavně u scén s rychle se pohybujícími objekty. Z tohoto důvodu je vhodné na výsledek použít filtr.

Filtrování

Tato fáze slouží především k odstranění šumu. Filtrování se využívá jak na výsledný obraz, tak i na získané mezivýsledky. Nejčastěji využívané filtry jsou bilaterální filtr nebo à trous filtr. Filtrování je typicky prováděno několikrát během jednoho snímku, proto je důležitá efektivní implementace.

K filtrování se může využít fragment shader, který má možnost získat hodnoty vedlejších pixelů a provést výpočet. Lepší je ale využít compute shader, který umožňuje lepší paralelní zpracování. Jak je uvedeno v knize *Practical Rendering and Computation with Direct3D 11* [19], lze některé filtry zpracovat ve dvou krocích. Takové filtry se nazývají separabilní. Jejich výhodou je, že se dají rozdělit na dva menší filtry, jeden pro vertikální průchod a druhý pro horizontální průchod. To znamená, že lze rozdělit například filtr o velikosti 3x3 na dva menší – 1x3 a 3x1. Takové rozdělení zajistí menší počet operací a tedy rychlejší výpočet. Další výhodou compute shaderu je sdílená paměť skupin vláken, která při výběru vhodných skupin zajistí lepší přístup k hodnotám vedlejších pixelů.

Uložení dat pro reprojekci

V této fázi se zkombinují všechny mezivýsledky z předchozích kroků a výsledný obraz se vykreslí. Dále se uloží informace, například aktuální hodnoty ambient occlusion, pro reprojekci v dalším snímku.

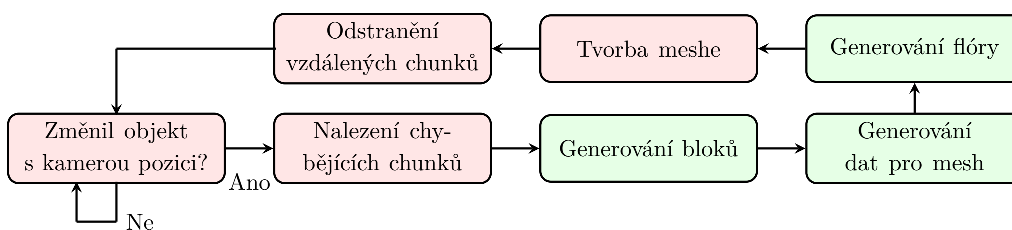
4.3 Generování terénu

Pro možnost postupného generování terénu je ideální terén rozdělit na menší části – *chunky*. Tyto chunky jsou tvořeny základními voxelovými objekty – bloky. Většina voxelových aplikací používá velikost jednoho chunku $16 \times 16 \times 16$ bloků. Ve hře Minecraft je blok reprezentován pomocí identifikátoru, který určuje o jaký blok se jedná, a dalších dat, které definují chování bloku. Pro tuto aplikaci bude pro jednoduchost blok reprezentován pouze identifikátorem.

Generování terénu je vhodné rozdělit do několika fází. Nejprve je nutné zjistit, zda se objekt s kamerou nachází na stejném nebo novém chunku oproti předchozímu snímku. Pokud se nachází na novém chunku, je potřeba zjistit, které chunky je nutné vygenerovat. Následně se pro jednotlivé nové chunky vygenerují bloky, jejichž pozice a identifikátor se vypočítá pomocí šumové funkce, což je typicky Perlinův šum, viz podkapitola 3.1. V další fázi se vypočítají vrcholy trojúhelníků a *uv* souřadnice pro textury. Z těchto dat se pak vytvoří pro daný chunk *mesh*. Je důležité, aby se při tvorbě meshe nevytvořily stěny, které nejsou vidět a pouze by zpomalovaly běh aplikace. Jakmile je dokončena tvorba meshe,

vygeneruje se také flóra. V poslední fázi se odstraní chunky, od kterých se objekt s kamerou vzdálil za určenou hranici.

Jelikož je metoda ray tracingu výpočetně velmi náročná, je nezbytné, aby bylo generování terénu rychlé a efektivní. Z tohoto důvodu je terén generován pomocí vláken. K tomu je využit balíček *Unity Job System*. Tento balíček umožňuje bezpečné generování chunků na všech jádrech, které CPU nabízí. Pro ještě větší optimalizaci je využit balíček *Burst*. Jedná se o kompilátor, který je určen především pro efektivní kompilaci kódu vytvořeného pomocí Unity Job System a pracujícího paralelně. Burst překládá kód z jazyka C# na vysoce optimalizovaný nativní strojový kód. Jednotlivé kroky generování terénu jsou znázorněny na obrázku 4.2. Paralelizované části jsou vybarveny zelenou barvou.



Obrázek 4.2: Schéma navrhovaného procesu generování terénu. Zeleně jsou vyznačené části, které jsou generovány paralelně pomocí balíčku Unity Job System a kompilátoru Burst.

Kapitola 5

Implementace

Tato kapitola popisuje podrobnou implementaci aplikace. Jednotlivé podkapitoly popisují tvorbu SRP, předzpracování a tvorbu G-bufferu, výpočet stínů, implementaci metody SVGF a generování terénu. Aplikace je vytvořena v enginu Unity v jazycích C# a HLSL. Jednotlivé bloky skriptovatelného vykreslovacího řetězce spravuje třída `CameraRenderer`. Generování světa a správu akcelerační struktury má na starost třída `SceneManager`.

5.1 Tvorba SRP

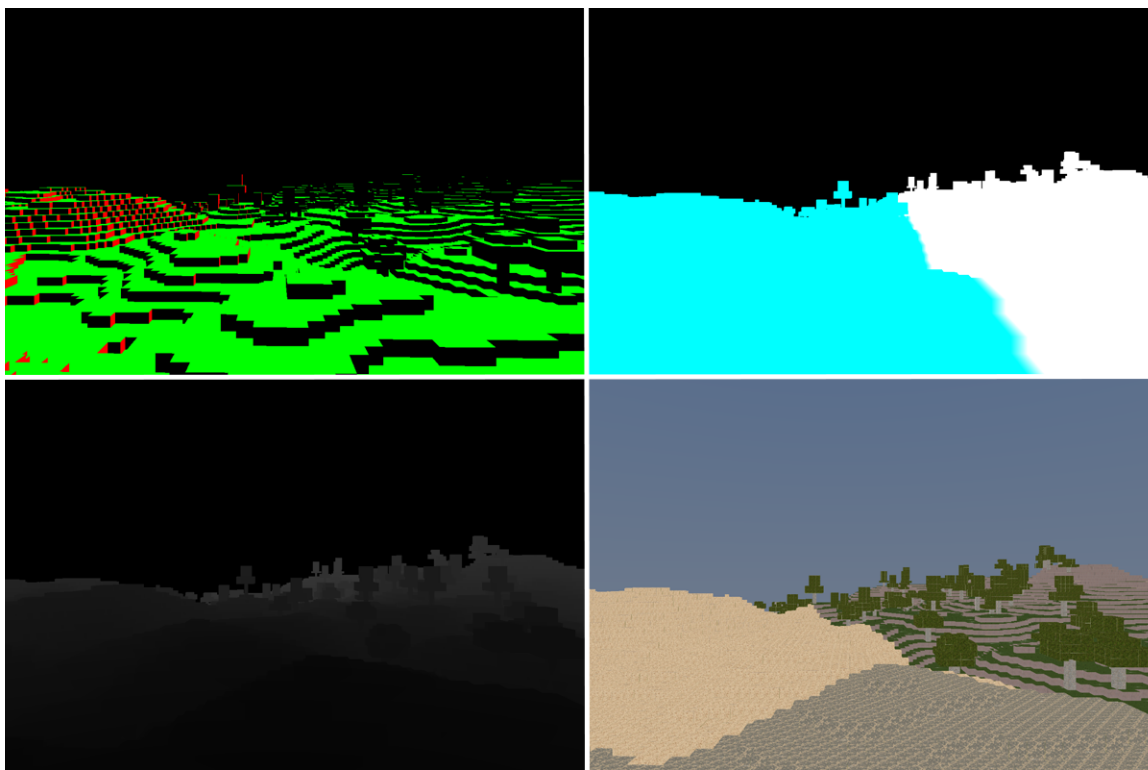
Jak již bylo zmíněno dříve, v této práci se využívá SRP neboli scriptable rendering pipeline. SRP umožňuje plánování a upravování vykreslovacích příkazů. Jednotlivými příkazy se dá vytvořit vlastní vykreslovací řetězec. Každý SRP řetězec vyžaduje dvě věci - *Render Pipeline Asset* a *Render Pipeline Instance*. Render Pipeline Asset slouží k uchování dat a informací pro daný vykreslovací řetězec. Tyto informace a data se přiřazují přímo v inspektoru Unity. V tomto projektu slouží především k uložení informací o compute shaderech. Render Pipeline Instance je třída, která obsahuje metodu `Render`, což je vstupní bod celého vykreslovacího řetězce. V tomto projektu se tato třída nazývá `VoxelRenderPipelineInstance`. Jednotlivé příkazy vykreslovacího řetězce jsou pak ve třídě `CameraRenderer`, která obsahuje také funkci `Render`, kterou `VoxelRenderPipelineInstance` volá pro vykreslení každého snímku.

5.2 Předzpracování

Cílem předzpracování je získat G-buffer s informacemi o scéně, které urychlí a usnadní další výpočty. Předzpracováním se získá celkem sedm různých informací. Všechny se dají získat pomocí rasterizace. V této práci však bylo cílem využít pouze ray tracing, takže se, i za cenu nepatrně pomalejšího přístupu, rasterizace nevyužívá. Jednotlivé informace o scéně se získají jedním průchodem ray tracingu nazvaným `GBuffer`. Paprsky se šíří z kamery skrz každý pixel výsledného obrazu. Níže jsou blíže specifikované jednotlivé informace, které se získávají. Celý G-buffer je znázorněn na obrázku 5.1.

Normály

Normály se počítají v místě průniku paprsku s objektem ve scéně. Nejprve se získá normála v prostoru objektu pomocí barycentrických souřadnic, následně se převede do prostoru



Obrázek 5.1: Obrázek znázorňující několik informací uložených v G-bufferu. Vlevo nahoře jsou normály, vpravo nahoře jsou pozice ve světě, vlevo dole je hloubka a vpravo dole je albedo. Dále jsou v G-bufferu uloženy informace o materiálech, identifikátorech objektů a motion vector.

scény. K uložení jsou potřebné tři hodnoty. Využívají se při několika výpočtech. U přímého osvětlení, nepřímého osvětlení a u výpočtu ambient occlusion slouží k výpočtu směru šíření dalšího paprsku nad polokoulí, jejíž natočení je dáno právě normálou. Dále jsou využívány v reprojekci, kde se porovnávají s normálami z předchozího snímku a kontrolují, zda jsou body pro reprojekci stejné. Další využití je při filtrování, kde se kontrolují normály okolních pixelů v tak zvané edge-stopping funkci pro zachování hran.

Pozice ve světě

Další informace, která se po protnutí nejbližšího tělesa získává, je pozice průniku ve světě. K uložení souřadnic jsou potřebné opět tři hodnoty. Pozice ve světě se využívají u všech dalších průchodů - nahrazují primární paprsky. To znamená, že pro přímé osvětlení, nepřímé osvětlení a ambient occlusion se už nemusí paprsky vysílat z kamery skrz každý pixel, ale přímo z míst průniků s nejbližšími objekty ve scéně, což ušetří čas. Dále jsou pozice ve světě, stejně jako normály, využity také při reprojekci, kde se porovnávají s pozicemi ve světě z předchozího snímku. Stejně tak jsou využity i při filtrování, kde kontrolují pozice ve světě okolních pixelů v edge-stopping funkci pro zachování hran.

Hloubka

Hloubka se získává tak, že se vypočítá jako podíl vzdálenosti od kamery do místa průniku s maximální délkou paprsku. Pokud paprsek neprotne žádný objekt, hloubka je nastavena na nulu. K uložení stačí jedna hodnota. Hloubka se využívá především ke kontrole, zda došlo k minutí. Pokud ano, není potřeba v rámci daného pixelu šířit paprsky pro přímé sluneční osvětlení, nepřímé osvětlení ani ambient occlusion. Zároveň není potřeba u takového pixelu provádět reprojekci ani filtrování. Hloubka se také využívá k úpravě koeficientu reprojekce u hodně vzdálených objektů.

Identifikátory

Identifikátory objektů jsou získávány kvůli reprojekci, kde se, stejně jako normály a pozice ve světě, porovnávají s identifikátory z předchozího snímku. Identifikátory lze získat v místě dopadu funkcí `InstanceID()`, kterou Unity nabízí. Problém nastává v momentě, kdy se do akcelerační struktury přidají nové objekty – identifikátory části objektů jsou změněny. Z tohoto důvodu se při vkládání nových objektů do akcelerační struktury vynechává při reprojekci kontrola identifikátorů. K uložení stačí jedna hodnota.

Materiály

V této práci je implementováno několik materiálů. U každého se počítá osvětlení jiným způsobem. Proto je nutné znát, s jakým materiálem se zrovna počítá. K uložení stačí jedna hodnota.

Albedo

Poslední informace, která se při tomto průchodu získává, je albedo reprezentující barvy objektů bez stínování. K uložení jsou potřebné tři hodnoty. Barva v místě dopadu paprsku je vypočítaná z textury, která je pro všechny materiály stejná. Tato hromadná textura obsahuje textury všech použitých bloků. Aby nedocházelo ke vzniku nežádoucích artefaktů, je při výpočtu barvy nutné zohlednit *LOD* neboli *level of detail* (úroveň detailu). K tomu se využívá *mipmapping*. Velikost textury v místě dopadu paprsku, ze které se počítá barva, je vybrána na základě délky paprsku, kterou paprsek cestoval z kamery do místa průniku s objektem. Pokud dojde k minutí, je nutné vypočítat barvu oblohy. Ta se mění podle části dne.

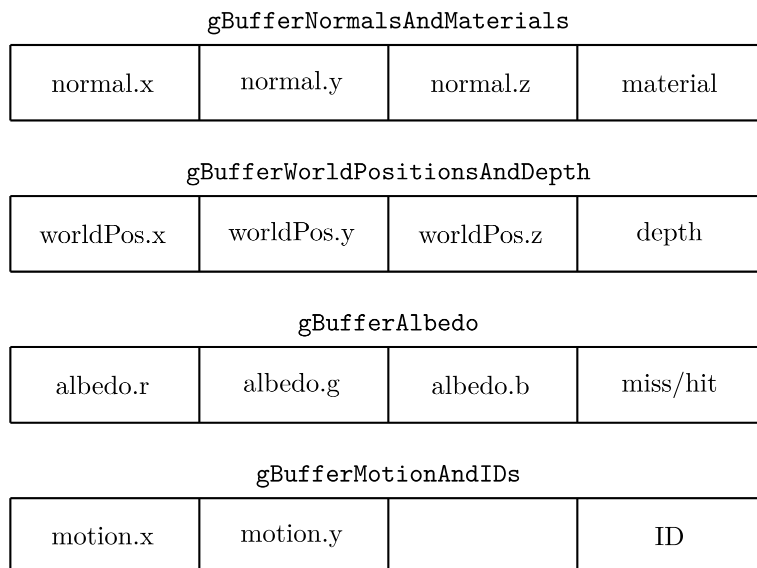
Motion vector

Motion vector se jako jediná z výše uvedených informací nezískává při průchodu `GBuffer`. Počítá se paralelně v compute shaderu `MotionVector.compute`. Využívá se při reprojekci k určení, jaký pixel reprezentoval aktuální pozici ve světě v předchozím snímku. K uložení jsou tedy potřebné dvě hodnoty, jejichž přičtením k pozici aktuálního pixelu se získá pozice pixelu v předchozím snímku pro daný bod ve scéně.

Způsob uložení

Každá informace potřebuje k uložení jiný počet hodnot. Bylo by zbytečné vytvářet pro každou informaci svůj buffer. Proto jsou všechny informace vhodně zkombinovány do několika

textur, aby se využilo co nejméně bufferů a minimalizoval se nevyužitý prostor. Výsledné uložení a názvy textur G-bufferu jsou na obrázku 5.2.



Obrázek 5.2: Znázornění způsobu uložení informací v G-bufferu do jednotlivých textur.

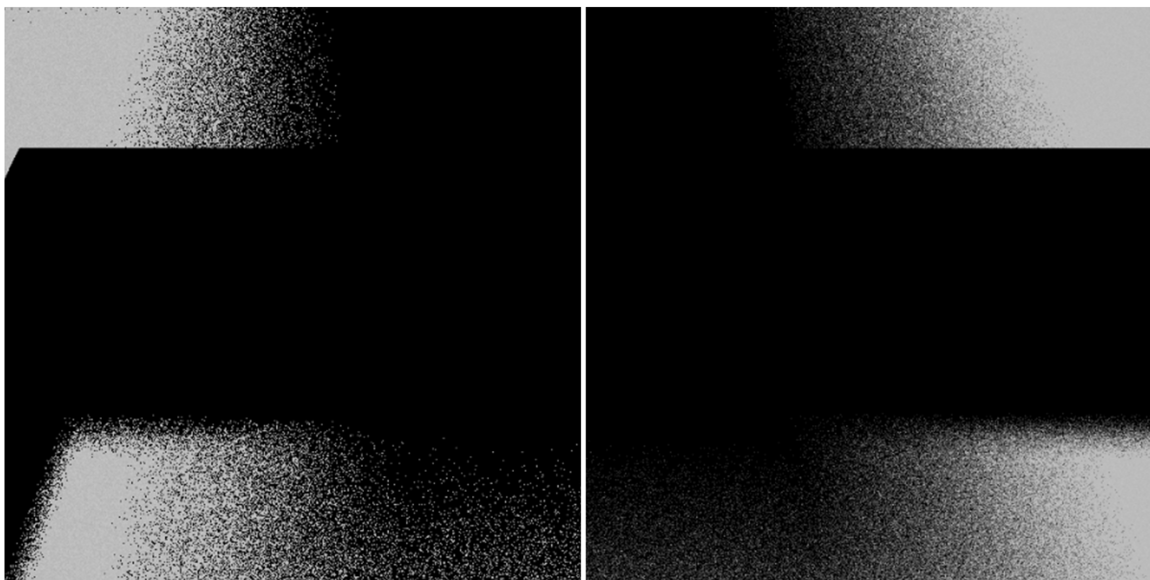
5.3 Ray tracing průchody

V práci jsou tři ray tracing průchody. První z nich se nazývá `GBuffer` a byl již popsán v sekci o předzpracování 5.2. Druhý průchod se nazývá `DirectAndIndirectLighting`, který získává přímé sluneční osvětlení, nepřímé osvětlení nebo oba typy osvětlení zároveň, záleží na nastavení parametrů programu. Posledním průchodem je `AO`, který počítá ambient occlusion. Closest-hit shadery, které jsou v této sekci zmíněny, jsou implementovány v shaderech pro jednotlivé materiály.

Přímé sluneční osvětlení

Výpočet přímého slunečního osvětlení probíhá v shaderu `RayTrace.raytrace`. Díky pozicím ve světě, které se získávají při průchodu `GBuffer`, není nutné šířit primární paprsky skrz každý pixel obrazovky, ale stačí je šířit z daných pozic ve scéně. Nejprve se kontroluje hodnota depth bufferu daného pixelu. Pokud je rovna nule, primární paprsek minul a hodnota přímého slunečního osvětlení je nastavena na hodnotu jedna ve všech složkách a výpočet tímto končí. Ve zbylých případech se z daného bodu vysílají ke slunci takzvané stínové paprsky. Nejprve je vybrán náhodný vzorek na slunci (kouli). Následně je k tomuto bodu šířen paprsek, který nese informaci o viditelnosti. Výchozí hodnota viditelnosti je nastavena na nulu. Paprsek má dále stanovenou masku, která při průchodu akcelerační strukturou ignoruje objekt slunce, protože důležité jsou pouze průniky s ostatními objekty. Dále je nastaven příznak, který vynechá vyhodnocení closest-hit shaderu. Je to z toho důvodu, že pokud paprsek narazí na objekt, hodnota viditelnosti je nula, což je už výchozí hodnota, není ji tedy potřeba měnit. Naopak pokud dojde k minutí, je volán miss shader a hodnota viditelnosti je nastavena na jedna, protože paprsek dorazil až ke slunci. Jelikož při vyslání právě jednoho paprsku z každé pozice ve světě vzniká velký šum, který by mohl

způsobovat problémy při reprojekci a filtrování, šíří se z každého bodu čtyři paprsky, jejichž hodnoty viditelnosti jsou zprůměrovány. Rozdíl mezi jedním a čtyřmi paprsky je vidět na obrázku 5.3.



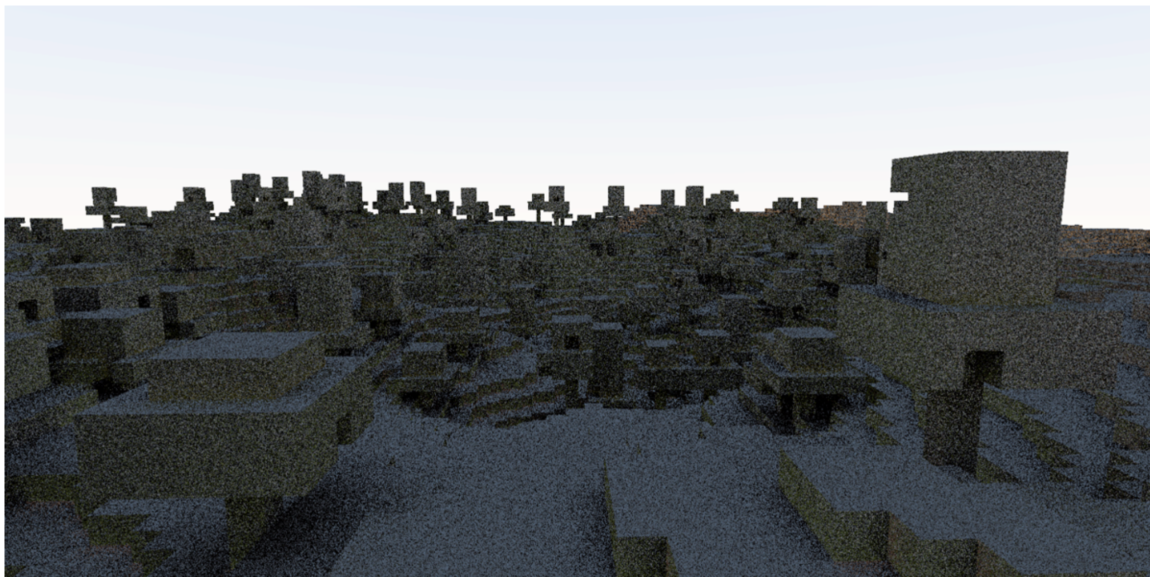
Obrázek 5.3: Na obrázku lze vidět rozdíl mezi jedním a čtyřmi stínovými paprsky v daných místech scény. Vlevo je jeden paprsek, vpravo čtyři. Šum vpravo je jemnější, zatímco při jednom paprsku jsou hodnoty pixelů nula nebo jedna.

Nepřímé osvětlení

Nepřímé osvětlení se počítá ve stejném průchodu jako přímé sluneční osvětlení. Pokud je hodnota depth bufferu větší než nula, šíří se paprsky nad hemisférou, která je dána normálou v daném bodě. Vzorek na hemisféře se vypočítá tak, že se vytvoří souřadnicový systém, jehož osa y je zarovnána s normálou. Následně se v tomto systému vypočítá vzorek na hemisféře. Vektor udávající směr k vzorku je poté převeden zpátky do souřadnicového systému scény do daného bodu. Získaným vzorkem se šíří paprsek, který pokud protne objekt scény, aktivuje closest-hit shader, kde se vypočítá barva v místě dopadu a také přímé sluneční osvětlení. Jelikož jsou v této práci difúzní objekty, je potřeba vynásobit vrácenou světelnou hodnotu podle Lambertova zákona skalárním součinem normály a vektoru, který udává směr ke vzorku. Lambertův zákon totiž říká, že množství odraženého světla je větší, když se směr dopadu blíží normále. Paprsky pro nepřímé osvětlení jsou šířeny čtyři. Jejich hodnoty jsou zprůměrovány. Konečná hodnota je ještě vydělena hodnotou *PDF* neboli probability density function (hustota rozdělení pravděpodobnosti). Výsledné nepřímé osvětlení je vidět na obrázku 5.4.

Ambient occlusion

Výpočet ambient occlusion probíhá v shaderu `AO.raytrace`. Stejně jako u předešlých dvou technik, i zde lze díky pozicím ve světě v G-bufferu přeskočit primární paprsky. Pokud primární paprsek neminul, tak se vypočítá náhodný vzorek na hemisféře, která je dána



Obrázek 5.4: Obrázek znázorňující nepřímé osvětlení scény. Hodnoty nepřímého osvětlení jsou tmavé, pokud by byla využita větší hloubka rekurze, hodnoty by byly světlejší. Větší hloubka by však značně zpomalila program.

normálou v daném bodě scény. Následně se z bodu šíří paprsek skrz vypočítaný vzorek. Paprsek je téměř totožný se stínovým paprskem využívaným u přímého slunečního osvětlení. I zde paprsek nese informaci o viditelnosti, která je na začátku nastavena na nulu. Díky tomu se může vynechat closest-hit shader. Rozdíl je ovšem v maximální vzdálenosti, kterou paprsek urazí. Zatímco u přímého slunečního osvětlení je vzdálenost vypočítaná přímo k bodu na slunci, u ambient occlusion stačí velmi krátká vzdálenost, protože je důležité zakrytí daného místa v blízkém okolí. Síla efektu ambient occlusion se dá upravit pomocí parametrů, které upravují vzdálenost paprsků, čímž se efekt zesílí nebo zeslabí. Celkem jsou vyslány čtyři paprsky, jejichž hodnota se zprůměruje. Výstup tohoto průchodu je vidět na obrázku 5.5.

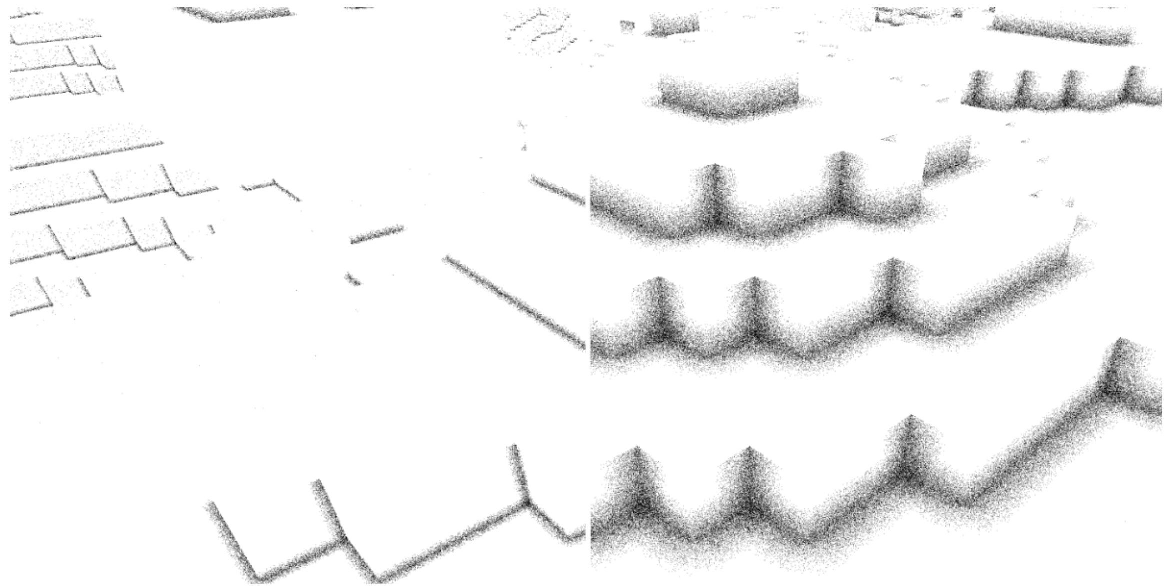
Kombinace stínů

Po zmíněných ray tracing průchodech jsou stíny uloženy v těchto bufferech:

- `directLightBuffer`: obsahuje buffer s přímým slunečním osvětlením,
- `indirectLightBuffer`: obsahuje buffer s nepřímým osvětlením,
- `ambientOcclusionBuffer`: obsahuje buffer s ambient occlusion.

Dalším krokem ve vykreslovacím řetězci je metoda SVGF. Zde je možné postupovat několika způsoby. Ve hře *Quake II RTX*[1] je metoda SVGF použita na každý buffer zvlášť, pouze s tím rozdílem, že pro nepřímé osvětlení není řízená variancí. V této práci byl ovšem po testování zvolen jiný přístup, a to že se jednotlivé buffery se stíny zkombinují do jednoho a na něj

[1] Odkaz na video, kde tvůrci Quake II RTX popisují jednotlivé ray tracing průchody a jejich filtrování – `https://www.youtube.com/watch?v=FewqoJjHR0A`.



Obrázek 5.5: Obrázek znázorňující nejslabší a nejsilnější stupně efektu ambient occlusion. Síla tohoto efektu je dána délkou paprsků, které se z jednotlivých bodu scény šíří.

je použita metoda SVGF. Výsledky se více podobaly takzvané *ground truth*, což označuje přesné reálné osvětlení. Jelikož jsou materiály difúzní, ke kombinaci stínů je využita funkce *BRDF* (bidirectional reflectance distribution function), což je odrazivá složka funkce BSDF, která je popsána v podsekci 2.2. Podle této funkce je výsledná barva pixelu dána vzorcem

$$color[x,y] = (\frac{L_I}{PDF} + L_D) \cdot \frac{A}{\pi}, \tag{5.1}$$

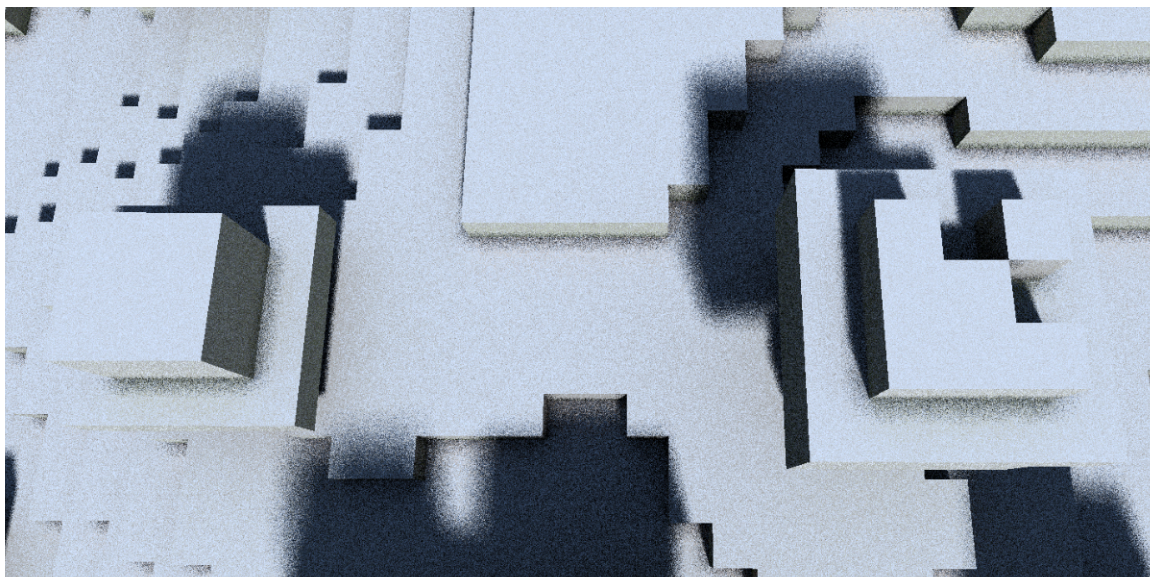
kde $color[x,y]$ je výsledná barva pixelu, L_I je nepřímé osvětlení, PDF je hustota rozdělení pravděpodobnosti, L_D je přímé osvětlení a A je albedo. Hodnota PDF byla zvolena $\frac{1}{2\pi}$. Po jejím dosazení se získá

$$color[x,y] = (\frac{L_I}{\frac{1}{2\pi}} + L_D) \cdot \frac{A}{\pi}, \tag{5.2}$$

což se dá dále upravit na výsledný tvar

$$color[x,y] = (L_I \cdot 2 + \frac{L_D}{\pi}) \cdot A. \tag{5.3}$$

Protože se před metodou SVGF kombinují pouze stíny, násobení albeda je provedeno až po filtrování jako poslední krok vykreslovacího řetězce. Ambient occlusion není ve funkci BRDF zohledněno z toho důvodu, že se jedná pouze o aproximaci globálního osvětlení. Jeho hodnota je proto přidána do přímého slunečního osvětlení tak, že se hodnoty navzájem vynásobí. Výsledný buffer se stíny před filtrováním je vidět na obrázku 5.6.



Obrázek 5.6: Obrázek znázorňující zkombinované přímé sluneční osvětlení, nepřímé osvětlení a ambient occlusion před voláním metody SVGF. Buffer se stíny obsahuje velký šum, který se následně metodou SVGF odstraní.

5.4 Filtrování metodou SVGF

Filtrování metodou SVGF probíhá ve třech krocích, které jsou popsány níže. Vstupem je výstup z ray tracingu, který obsahuje, z důvodu nízkého počtu paprsků na jeden pixel, velký šum. Dále je na vstupu množství pomocných bufferů z předzpracování. Výstupem je vyfiltrovaný obraz, který se co nejvíce podobá reálnému osvětlení.

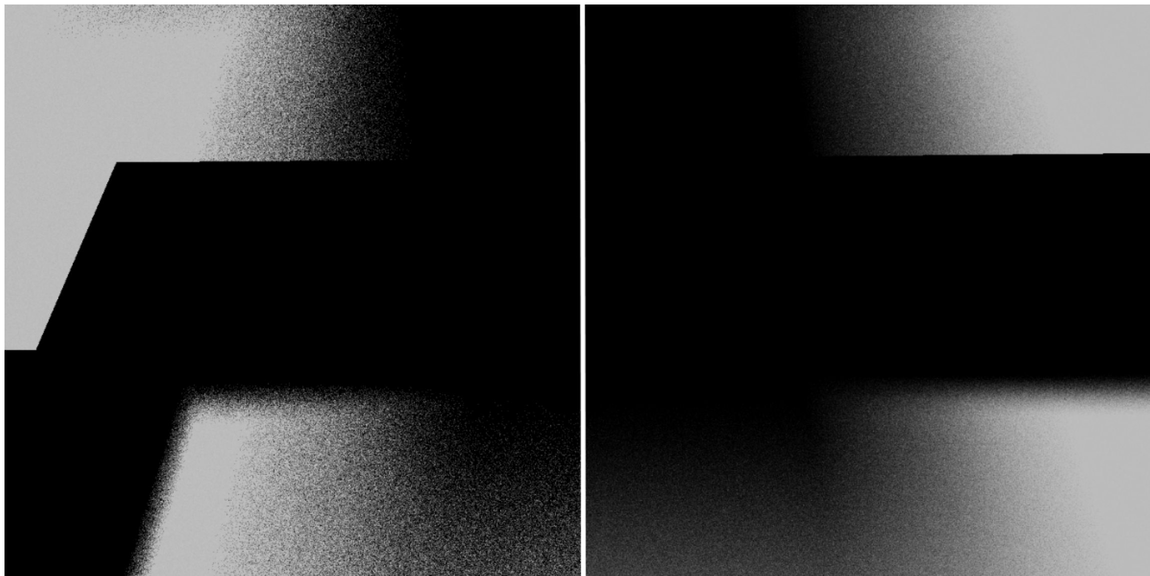
Reprojekce

Prvním krokem metody SVGF je reprojekce. Ta má na vstupu několik pomocných informací – pozice ve světě, normály, identifikátory objektů a jejich hodnoty z předchozího snímku. Vstupem je také výstup z ray tracingu, motion vector a výsledek reprojekce z předchozího snímku. Cílem reprojekce je získat informace o scéně, které již byly vypočítány v minulém snímku – spousta paprsků totiž dopadá na ta stejná místa. Pokud se kamera nehýbe vůbec a scéna je statická, dopadají všechny paprsky na ta stejná místa. Jelikož je snaha o co nejmenší počet paprsků, je dobré tato data z minulých snímku využít.

Reprojekce se počítá paralelně v compute shaderu `Reprojection.compute` pro každý pixel obrazovky. Nejprve se zjistí, zda daný pixel nese nějakou informaci, nebo zda paprsek na daném pixelu neprotl žádný objekt scény. Tato informace je uložena v depth bufferu. Následně se pomocí motion vectoru zjistí, který pixel minulého snímku odpovídá pixelu tohoto snímku. Poté se na těchto pozicích porovnají hodnoty normál, identifikátory objektů a vzdálenost pozic ve světě. Pokud tyto hodnoty aktuálního a předchozího snímku nesouhlasí, na výstup se zapíše hodnota aktuálního snímku. Pokud souhlasí, výstup reprojekce je dán vzorcem

$$Output[x, y] = k \cdot LastFrame[x, y] + (1 - k) \cdot CurrentFrame[x, y], \quad (5.4)$$

kde x a y určují pozice pixelů, $LastFrame$ a $CurrentFrame$ jsou buffery s hodnotami z předchozího a aktuálního snímku a k je koeficient reprojekce, který byl v této práci nastaven na hodnotu 0,8. Tato hodnota byla určena na základě článku [14]. Ukázka reprojekce je na obrázku 5.7.



Obrázek 5.7: Obrázek vlevo znázorňuje vstup reprojekce a obrázek vpravo její výstup. Stíny jsou jemnější, což značně urychluje a vylepšuje filtrování.

Výpočet variance

Druhým krokem metody SVGF je výpočet variance. Určuje, jak se hodnoty množiny liší od jejich střední hodnoty. Jinými slovy udává, jak moc jsou hodnoty v množině rozptýleny. Toto lze uplatnit na lokální úrovni při filtrování – v místech, kde je nízká, není nutnost filtrovat tak moc, jako v místech, kde se hodnoty hodně liší. Nejedná se tedy o výpočet variance nad množinou všech pixelů najednou, ale pro každý pixel se počítá zvlášť v jeho blízkém okolí. Její další výhodou je zachování ostrosti stínů těsně za hranami objektů. Tyto stíny se často vlivem několika iterací filtru rozmazaly a výsledný stín za hranou byl nehezký.

Výpočet probíhá paralelně v compute shaderu `Variance.compute`. Z důvodu, že se variance v metodě SVGF typicky počítá na okolí o velikosti 7×7 či 9×9, lze výpočet rozdělit na dva průchody, vertikální a horizontální, a provést ho separabilně, čímž dojde ke zrychlení. Variance je dána vzorcem

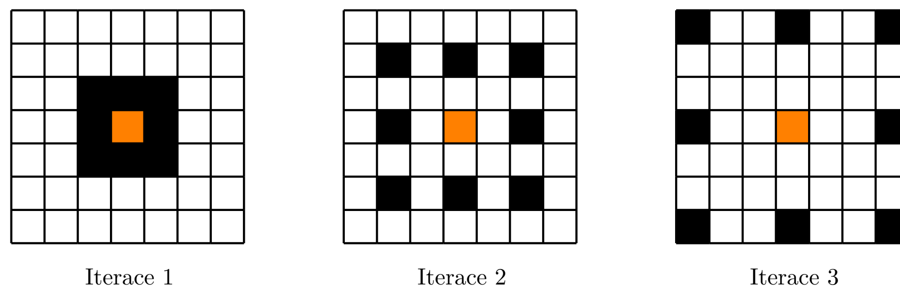
$$\sigma^2 = \left(\frac{1}{n}\sum_{i=1}^{n} Y_i^2 - \overline{Y}^2\right) \tag{5.5}$$

kde n je počet pixelů v daném okolí, Y_i je hodnota pixelu a $\overline{Y}$ je střední hodnota daného okolí. Na výslednou hodnotu je dále aplikována Besselova korekce, která opravuje odchylku ve varianci. Tato odchylka vzniká z důvodu, že střední hodnota není počítána pro všechny pixely obrazovky, ale pouze pro vybrané okolí. Výsledná hodnota variance pro jednotlivé pixely je uložena v pomocném bufferu. Finální výpočet je dán vzorcem

$$\sigma^2 = (\frac{1}{n}\sum_{i=1}^{n} Y_i^2 - \overline{Y}^2) \cdot \frac{n}{n-1} \tag{5.6}$$

Filtrování

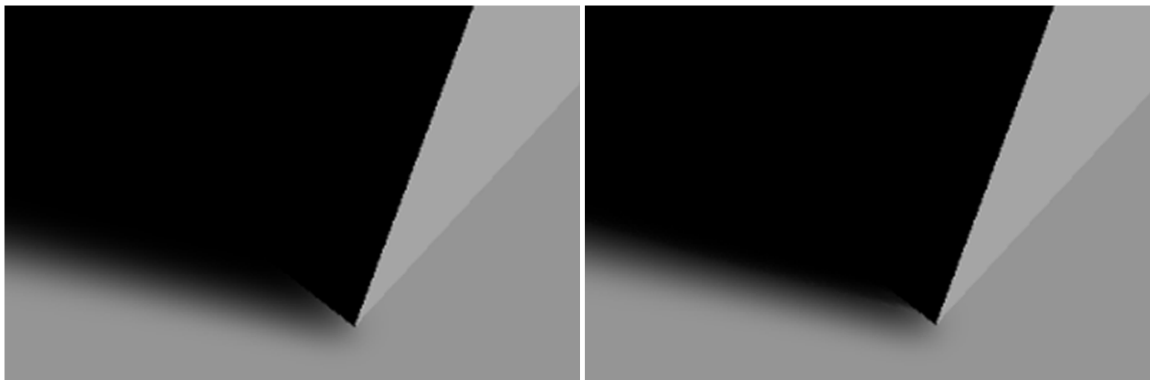
Posledním a nejdůležitějším krokem SVGF metody je filtrování. Vstupem je buffer se stíny, který stále, i po reprojekci, obsahuje velké množství šumu. Výstupem je buffer se stíny, které se přibližují reálnému stínování. Filtrování se většinou provádí pomocí à trous filtrů. Tyto filtry jsou specifické tím, že se filtruje v několika iteracích a při výpočtech se vynechávají některé pixely. Jak je znázorněno na obrázku 5.8, s každou iterací se zvyšuje velikost skoku a je počítáno se vzdálenějšími pixely. Tímto způsobem se dá dosáhnout filtrování ve větším okolí pixelu bez potřeby zbytečně velkého konvolučního jádra. V této práci jsou využity jádra o velikosti 3×3 a 5×5.



Obrázek 5.8: Obrázek znázorňující tři iterace à trous filteru o velikosti 3×3 pro oranžový pixel.

Filtrování se provádí paralelně v compute shaderu `ATrous.compute`. Vstupní funkce tohoto shaderu se nazývá `ATrous`. Pokud se v předchozím kroku počítala variance, filtrování je takzvaně řízené variancí. V takovém případě se získá hodnota variance z pomocného bufferu. Pokud je hodnota menší než nastavená varianční konstanta, filtrování se neprovádí a do výstupního bufferu se zapíší hodnoty přímo ze vstupu. Hodnota varianční konstanty není pevně dána, záleží na nastavení celé SVGF metody a může se lišit například při zapnuté nebo vypnuté reprojekci. Filtrování řízené variancí je značně rychlejší a umožňuje zachování tvrdých stínů za hranami objektů. Nevýhodou je ztráta postupného přechodu mezi tvrdými a měkkými stíny, kde je vidět větší skok. Tento přechod se dá zjemnit provedením alespoň dvou iterací à trous filtru. Rozdíl mezi filtrováním řízeným a neřízeným variancí je vidět na obrázku 5.9.

Zbylé pixely, kde variance přesahuje varianční konstantu, se filtrují pomocí kernelů o velikosti 3×3 a 5×5. Je nezbytné, aby byly při filtrování zachovány hrany. Pokud by nebyly, stíny různých objektů by byly rozmazané mezi všechny objekty. Je tedy potřeba určit pixelům v okolí 3×3 či 5×5 váhu, která udává, jak moc se daný pixel při filtrování započítá. Největší váhu budou mít ty pixely, které jsou na stejném objektu jako centrální filtrovaný pixel a mají stejnou normálu. K určení váhy slouží několik edge-stopping funkcí neboli funkcí zachovávajících hrany. První z nich kontroluje vzdálenost bodů ve scéně okolních pixelů od centrálního pixelu. K tomu se využije buffer `gBufferWorldPositionsAndDepth`, kde jsou uloženy pozice ve světě. Pokud je vzdálenost větší než stanovená hodnota, zkoumaný okolní pixel bude mít při filtrování váhu nula a do



Obrázek 5.9: Obrázek znázorňující rozdíl mezi filtrováním neřízeným a řízeným variancí. Kromě ostřejších stínů za hranami objektů slouží variance také ke zrychlení filtrování.

filtrování se nezapočítá. Další funkce kontroluje normály okolních pixelů a centrálního pixelu, které jsou uloženy v `gBufferNormalsAndMaterials`. Váha zkoumaného pixelu z okolí je dána vzorcem

$$w_n = \max(n_p \cdot n_q, 0)^{k_n}, \tag{5.7}$$

kde n_p je normála centrálního pixelu, n_q je normála pixelu v okolí a k_n je koeficient normálové funkce zachovávající hrany, který je v práci nastaven na 128. Poslední funkce určuje váhu filtrování na základě barev okolních pixelů a centrálního pixelu. Je dána vzorcem

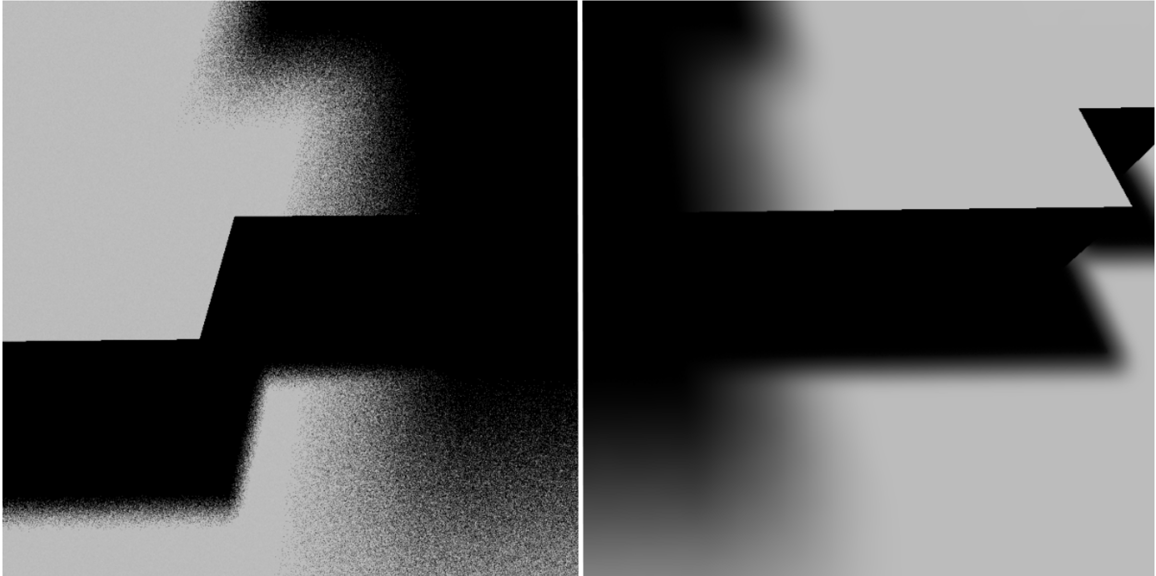
$$w_c = \min(\exp(-(c_p \cdot c_q)), 1), \tag{5.8}$$

kde c_p je barva centrálního pixelu a c_q je barva pixelu v okolí. Výsledná váha okolních pixelů je tedy dána násobkem w_n, w_c a váhou à trous filtru na daném pixelu.

Důležitým parametrem při filtrování je počet iterací. Po testování různých hodnot se došlo k počtu sedm, při kterém zcela zanikne šum vzniklý malým počtem paprsků na jeden pixel. Pokud je ovšem využita i variance, počet iterací se liší právě na základě hodnoty variance pro dané pixely. Pokud variance pro daný pixel klesne pod stanovenou hodnotu, není potřeba provádět další iterace. Problém s počtem iterací nastává při filtrování bodů ve velké vzdálenosti, při kterém dochází ke zmenšení viditelnosti jemných stínů či naopak ke zvýraznění větších stínů. Je to dáno především tím, že je větší plocha scény reprezentována malým počtem pixelů. Toto se dá ošetřit menším počtem iterací či změnou parametrů u funkcí zachovávajících hrany. Tím se ale znovu objeví nechtěný šum. Výsledek celé metody SVGF je vidět na obrázku 5.10.

5.5 Generování terénu

Primárním zaměřením této práce je vykreslování pomocí ray tracingu, které je výpočetně velmi náročné. Proto byl kladen důraz na rychlé a efektivní generování terénu. Jak již bylo zmíněno v návrhu, ke generování terénu jsou využity dva balíčky - Unity Job System a Burst. Tyto balíčky umožňují paralelní generování terénu pomocí vysoce optimalizovaného nativního strojového kódu.



Obrázek 5.10: Na obrázku vlevo lze vidět vstup do metody SVGF, která má za úkol především odstranění šumu. Na obrázku vpravo je její výstup.

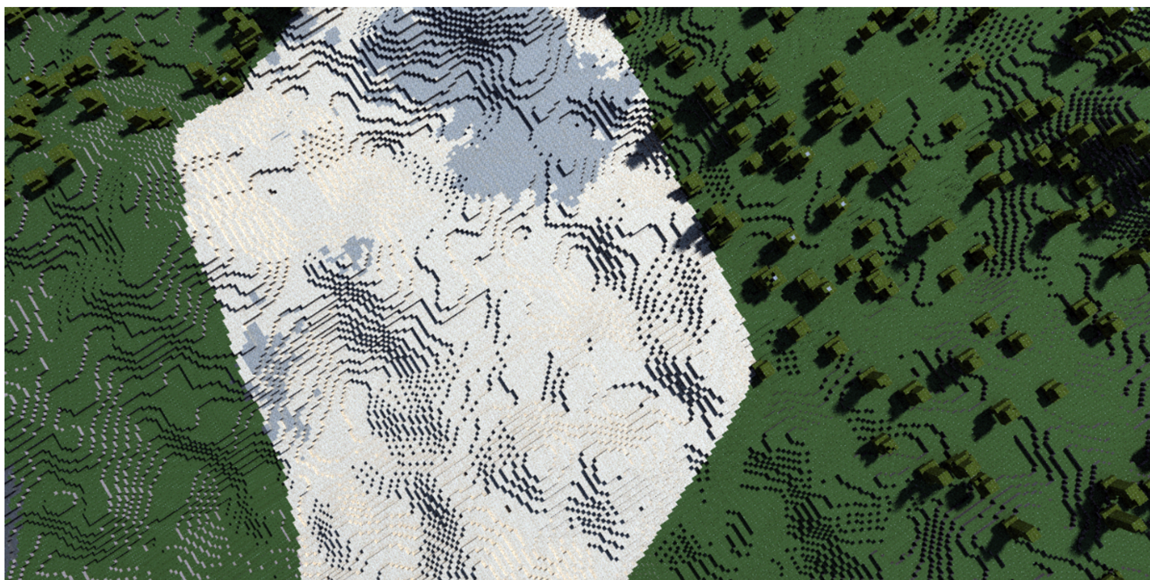
Generování terénu je řízeno třídou `SceneManager`, jednotlivé chunky jsou reprezentovány třídou `Chunk`. Generování probíhá v několika fázích:

- `Resting` (odpočívání): ve fázi odpočívání se kontroluje, zda se kamera nachází na stejném chunku jako v předchozím snímku. Pokud se chunk změnil, je volána funkce `FindMissingChunks()`, která v zadaném poloměru nalezne chunky, které jsou potřeba vygenerovat. Tyto chunky se uloží do pole `_toGenerate`.
- `GeneratingBlocks` (generování bloků): během této fáze se paralelně generují informace o blocích u chunků v poli `_toGenerate`. Počet paralelních prací je dán vhodným násobkem počtu procesorů v počítači. Generování probíhá ve třídě `GenerateBlocksJob` a je podrobně popsáno v sekci 5.5.
- `GeneratingMeshData` (generování dat pro mesh): jakmile jsou vygenerovány informace o blocích, začne generování dat pro mesh, které probíhá také paralelně ve třídě `GenerateMeshData`. Data se do meshe v této fázi ještě nepřiřazují, pouze se generují. Podrobně je generování dat pro mesh popsáno v sekci 5.5.
- `GeneratingFlora` (generování flóry): po naplnění meshů se zkontroluje, zda na jednotlivých chuncích budou stromy – pokud ano, stromy se paralelně generují na pozicích, které byly určeny při generování bloků.
- `FillingMeshes` (naplnění dat do meshe): v této fázi se přiřazují do meshe vrcholy, jejich indexy a uv souřadnice pro textury. Pokud se na daném chunku nachází stromy, zkombinují se data o vygenerovaných blocích a stromech a až poté se přiřadí do meshe. Jakmile je mesh dokončen, vloží se do akcelerační struktury.
- `RemovingChunks` (odstranění chunků): po dokončení generování nových chunků se odstraní ty chunky, které jsou mimo dosah kamery. Následuje fáze odpočívání.

Generování bloků

Bloky se generují paralelně po chuncích ve třídě `GenerateBlocksJob`. Maximální počet chunků na sobě je nastaven na tři. Každý chunk obsahuje $16 \times 16 \times 16$ bloků. Aby nevznikaly zbytečné stěny mezi jednotlivými chunky, je v další fázi při tvorbě meshe potřebné znát i hraniční bloky vedlejších chunků. To však kvůli paralelnímu zpracování není možné, proto se při generování bloků generují bloky v rozsahu $18 \times 18 \times 18$. Tímto způsobem zná každý chunk krajní bloky vedlejších chunků.

Při generování se nejprve pro každou souřadnici (x, z) zjistí výška terénu a výška jednotlivých typů bloků. K tomu se využije Perlinův šum, viz 3.1. Dále se určí biom, ve kterém se daný blok nachází. V práci se nachází tři biomy - trávník, les a poušť. Jsou definovány pomocí Voroného diagramů a jejich řídících bodů. Daný blok se přiřadí k tomu biomu, jehož řídící bod je mu nejblíž. Tyto řídící body se generují po spuštění programu. Pokud je biom les, generují se pozice pro stromy. K tomu je využita hashovací funkce, jejímž vstupem jsou souřadnice (x, z), což zajistí, že se stromy vygenerují vždy na stejném místě. Ukázka jednotlivých biomů je vidět na obrázku 5.11.



Obrázek 5.11: Obrázek znázorňující jednotlivé biomy, které se v této práci generují. Jedná se o trávníky, lesy a pouště. Biomy jsou tvořeny pomocí Voroného diagramů.

Generování dat pro mesh

Generování dat pro mesh probíhá paralelně po chuncích ve třídě `GenerateMeshDataJob`. Pro každý blok daného chunku se nejprve kontroluje, zda je prázdný (blok vzduchu) nebo plný. Pokud je plný, kontrolují se sousední bloky na všech šesti stranách. Ke kontrole se využívají funkce ze statické třídy `BlockUtils`. Pokud na sousední straně existuje také plný blok, není pro tuto stranu nutné přidávat data pro mesh, protože by daná strana nebyla přes sousední blok nikdy vidět. Pokud je ovšem sousední blok prázdný, je nutné přidat tuto stranu do meshe. K tomu se využívají data ze struktury `BlockData`, kde jsou definovány vrcholy bloků, jejich indexy a také uv souřadnice pro jednotlivé textury bloků. Připravená

data se po naplnění do meshe nepřidávají, nejprve se vygenerují stromy, pokud na daném chunku nějaké jsou, a až pak se naplní mesh.

Generování flóry

Generování flóry probíhá paralelně po chuncích ve třídě `GenerateTreesJob` pod podmínkou, že se na daném chunku nachází alespoň čtyři stromy. Při generování stromů se, na rozdíl od generování bloků, generují bloky i data pro mesh zároveň. Je to z důvodu, že samotné stromy nemají až tolik bloků jako terén, proto by nemělo smysl toto generování provádět paralelně ve dvou fázích. Zároveň je výška bloků pro stromy zvýšena. Stromy se tedy mohou generovat v rozsahu $16 \times 22 \times 16$ bloků. Stromy se generují metodou `GenerateTree`. Výška stromu se určí pseudonáhodně hashovací funkcí a následně se vytvoří listy, které jsou s rostoucí výškou blíž ke kmenu. Následně se vygenerují data pro mesh stejným způsobem, jakým se generují pro bloky. Vygenerovaná flóra je na obrázku 5.12.

Obrázek 5.12: Na tomto obrázku lze vidět různé náhodně vygenerované stromy. Stromy se generují pomocí Perlinova šumu a hashovací funkce.

Kapitola 6

Vyhodnocení

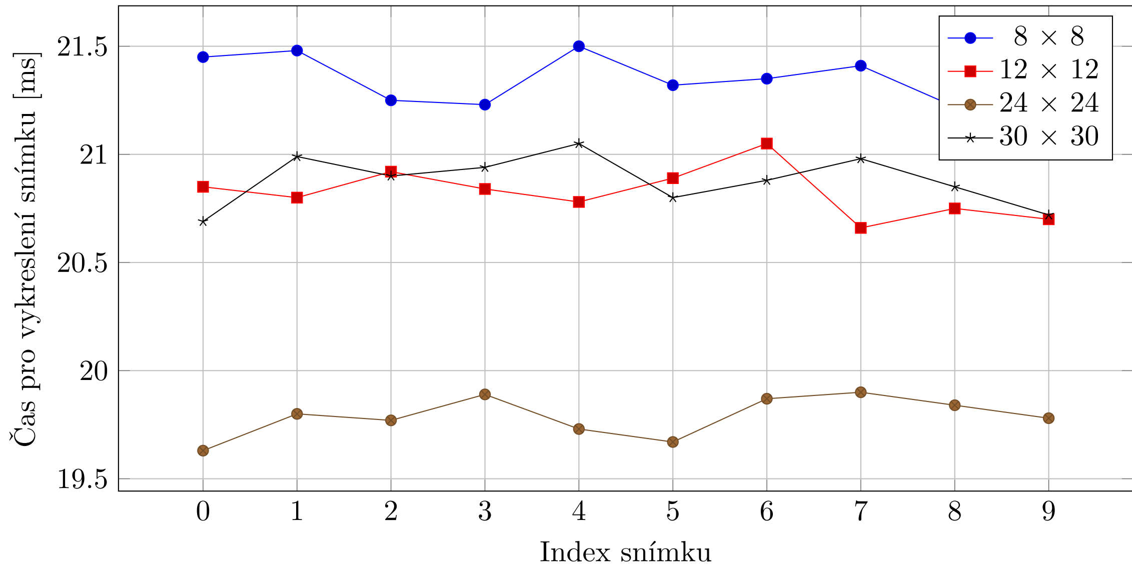
Projekt byl vyvinut na počítači, který disponuje procesorem AMD Ryzen 7 3700X a grafickou kartou NVDIA GeForce RTX 2070 SUPER. Právě tato grafická karta s architekturou Turing obsahuje jádra RT, která jsou popsána v sekci 2.11. Díky nim dosahuje výkonu více než 10 gigapaprsků za sekundu, což umožňuje ray tracing v reálném čase. Tato práce byla zaměřena především na vykreslování fyzicky přesných stínů. V této kapitole jsou naměřené výsledky a výsledné snímky z programu. Všechny výsledky a naměřené hodnoty byly dosaženy při rozlišení 1920×1080.

6.1 Měření

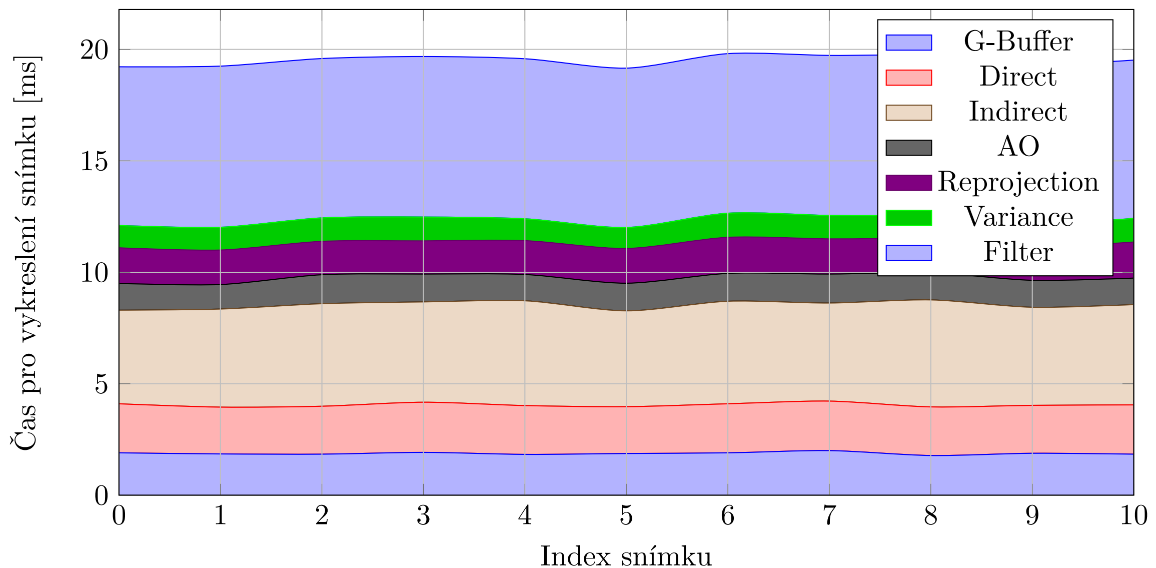
Prvním měřeným parametrem je počet vláken, které se spouští v rámci jedné skupiny vláken v compute shaderech. Vlákna jsou dána parametrem `numthreads(x, y, z)`, kde x, y a z značí počet vláken v jednotlivých dimenzích. Jelikož jsou v této práci compute shadery využity pro zpracování obrazu, vystačí pouze vlákna v dimenzích x a y, proto je jejich počet ve směru z vždy roven 1. Celkový počet nesmí kvůli hardwarovým omezením přesáhnout hodnotu 1024. Testování bylo provedeno s velikostí vláken $8 \times 8 \times 1$, $12 \times 12 \times 1$, $24 \times 24 \times 1$ a $30 \times 30 \times 1$. Zároveň při měření není zabírána obloha, protože obloha není filtrována, čímž se značně sníží doba vykreslení snímku. Výsledky jsou znázorněny v grafu 6.1.

Další důležitou věcí je podíl času jednotlivých modulů při vykreslování. Dle očekávání je nejnáročnějším modulem filtrování, protože probíhá v sedmi iteracích. Zároveň se nedá filtrovat separabilně kvůli funkcím zachovávajícím hrany. Z ray tracing průchodů je nejpomalejší průchod pro výpočet nepřímého osvětlení. To je dáno především tím, že se paprsky pro nepřímé osvětlení šíří čtyři a zároveň se pro každý musí počítat pseudonáhodný úhel nad hemisférou danou normálou. Zároveň paprsek oproti přímému slunečnímu osvětlení či ambient occlusion průchodu nese informace o všech složkách barvy, zatímco zbylé zmíněné průchody nesou pouze jedno číslo, které udává, zda je bod zastíněný nebo ne. Průchod pro získání G-bufferu je rychlejší z důvodu, že se šíří pouze jeden paprsek, nikoliv čtyři. Naměřené hodnoty jednotlivých modulů jsou vidět v grafu 6.2.

Dalším důležitým parametrem v programu je množství světa, které je dáno poloměrem, ve kterém se generují chunky v okruhu kamery. V tabulce 6.1 je uveden přibližný počet vrcholů ve scéně vzhledem k velikosti poloměru. Počet vrcholů se liší hlavně biomy, které jsou zrovna načtené. Nejvíce vrcholů je ve scéně, kde je celý svět tvořen lesem. Dále je v grafu 6.3 ukázán čas potřebný k vykreslení snímku při různých velikostech světa. Jak lze



Graf 6.1: Tento graf znázorňuje počet milisekund potřebných pro vykreslení snímku s různým počtem vláken v compute shaderech. Nejlepších výsledku program dosahuje při použití 24 × 24 vláken. Měření bylo provedeno s chunky v okruhu 13 chunků od kamery.



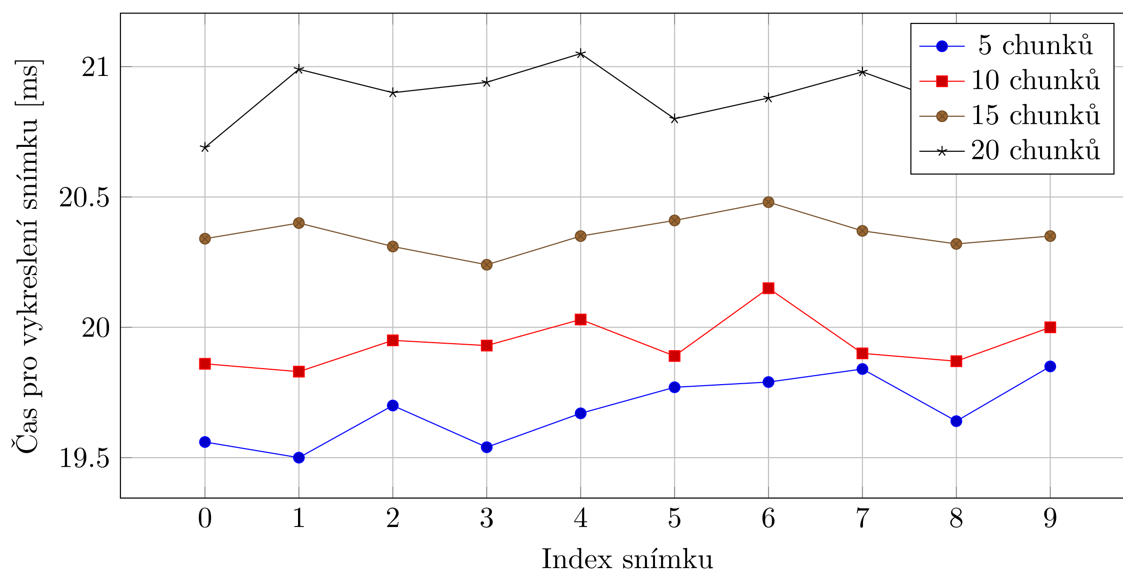
Graf 6.2: Tento graf ukazuje rozložení časové náročnosti jednotlivých modulů vykreslovacího řetězce pro vykreslení snímků. Nejnáročnější je filtrování, protože probíhá v sedmi iteracích. Z ray tracing průchodů je nejnáročnější výpočet nepřímého osvětlení. Měření bylo provedeno s chunky v poloměru 13 chunků od kamery.

z grafu vidět, množství světa až tolik neovlivňuje dobu vykreslování. To je dáno především tím, že se do akcelerační struktury vkládají celé chunky a nikoliv menší objekty. Proto procházení struktury netrvá o tolik déle.

Při nastavení většího světa však trvá generování terénu déle. Je to dáno především tím, že je na jeden snímek nastaven pevný počet prací, které se mohou na procesorech pustit.

Poloměr	Přibližný počet vrcholů
5	157000
10	580000
15	1289000
20	2223000

Tabulka 6.1: Tabulka znázorňující přibližný počet vrcholů ve scéně pro měřené poloměry chunků v okolí kamery.

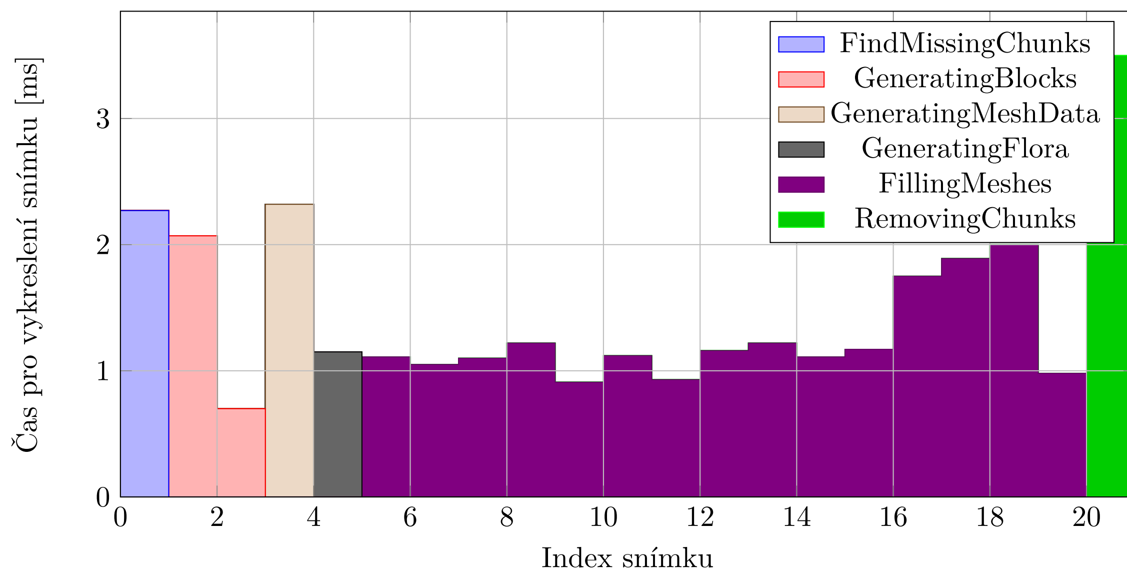


Graf 6.3: V tomto grafu lze vidět rozdíl v délce vytvoření jednoho snímku s rozdílným poloměrem chunků v okruhu kamery. Rozdíly nejsou až tak velké, protože se do akcelerační struktury vkládají celé chunky a nikoliv menší objekty.

To znamená, že čím je větší terén, tím více snímků je zapotřebí pro vygenerování nových chunků. Doba jednotlivých fází pro generování terénu, které jsou popsány v sekci 5.5, je znázorněna v grafu 6.4. Hodnota poloměru je při měření nastavena na 13, což je i výchozí hodnota programu.

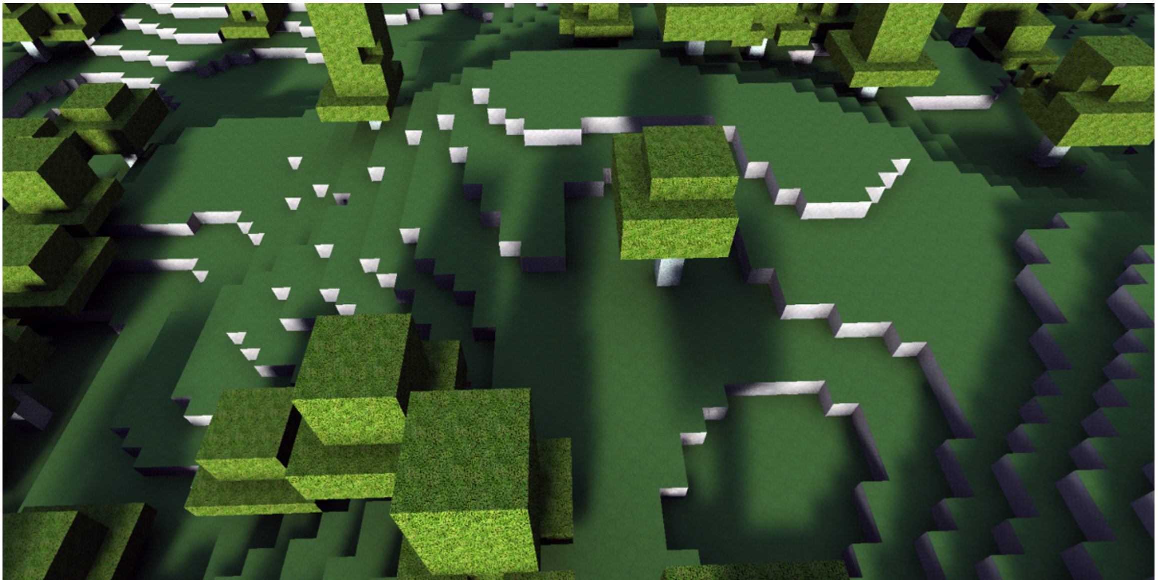
6.2 Výsledné snímky

Tato sekce obsahuje snímky z finální aplikace. Snímky jsou pořízeny během dne i během západu slunce.



Graf 6.4: Tento graf znázorňuje rozložení generování terénů na několik snímků po jednotlivých fázích generování. Cílem bylo dosáhnout generování tak, aby zabíralo přibližně dvě milisekundy na snímek a co nejméně zpomalovalo šíření snímků. Jak je v grafu vidět, jediná fáze, která hranici výrazně přesahuje, je odstraňování chunků, které již nejsou v dosahu. Časově nejnáročnější bylo ovšem naplnění meshe daty, což je rozloženo do patnácti ámců.

Obrázek 6.5: Ukázka stínů během dne.



Obrázek 6.6: Ukázka stínů během dne.

Obrázek 6.7: Ukázka stínů při západu slunce.

Kapitola 7

Závěr

Cílem této práce bylo nastudovat metody a vytvořit program k vizualizaci voxelových scén v reálném čase s využitím ray tracingu. Z grafických efektů bylo primární zaměření na vykreslování stínů. Jako engine využívající rozhraní DXR bylo využito Unity, kde je ray tracing v experimentální verzi. Výsledné grafické efekty zahrnují tvrdé stíny, měkké stíny a ambient occlusion. Efektů bylo dosaženo několika průchody ray tracingu a filtru pro odstranění vzniklého šumu. Pro filtrování byla zvolena metoda SVGF, jejíž implementace trvala v této práci nejdelší dobu. Dále bylo implementováno procedurální generování voxelového terénu, u kterého byl kladen důraz na rychlost generování, aby co nejméně zpomaloval ray tracing. Proto byly využity experimentální balíčky Unity Jobs a Burst pro optimalizovaný vícevláknový kód. Nevýhodou experimentální verze ray tracingu v Unity bylo, že některé využité funkce pro práci s akcelerační strukturou přibývaly až během práce s novými verzemi Unity. Výsledný program vykresluje vygenerovaný terén rychlostí přibližně padesát snímků za sekundu. Generování terénu je rozloženo do několika snímků tak, aby se generování pohybovalo kolem dvou milisekund na snímek.

Do práce by bylo možné přidat několik dalších efektů. Mezi nejzajímavější patří odrazy. Systém generování terénu umožňuje přidání nových typů bloků či vodní plochy, která by mohla odrážet okolí. Přidáním odrazů by se však musela změnit metoda SVGF, která pro filtrování odrazů není ideální. Dále by bylo možné do práce přidat lokální zdroje světla, což by vyžadovalo upravit metodu SVGF na její adaptivní verzi. Aktuální implementace reprojekce vytváří při rychlém pohybu kamery v hodně vzdálených místech scény nereálné pohyby stínů, což je další prostor pro vylepšení.

Nastudování a pochopení metod ray tracingu a jejich praktického využití zabralo hodně času. Nejtěžším krokem v této práci však bylo přestat dbát na často až přílišné detaily při vykreslování stínů a posunout se na další možnosti rozšíření práce. I přes to práce přinesla spoustu nových poznatků, zábavy a rozšířila povědomí o možnostech ray tracingu v reálném čase a jeho obrovském potenciálu.

Literatura

[1] Aurenhammer, F. Voronoi Diagrams—a Survey of a Fundamental Geometric Data Structure. *ACM Comput. Surv.* New York, NY, USA: Association for Computing Machinery. září 1991, sv. 23, č. 3, s. 345–405. DOI: 10.1145/116873.116880. ISSN 0360-0300. Dostupné z: https://doi.org/10.1145/116873.116880.

[2] Cook, R. L., Porter, T. a Carpenter, L. Distributed Ray Tracing. In: *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques.* New York, NY, USA: Association for Computing Machinery, 1984, s. 137–145. SIGGRAPH '84. DOI: 10.1145/800031.808590. ISBN 0897911385. Dostupné z: https://doi.org/10.1145/800031.808590.

[3] Haines, E. a Akenine Möller, T., ed. *Ray Tracing Gems.* Apress, 2019. http://raytracinggems.com.

[4] Hasselgren, J., Munkberg, J., Salvi, M., Patney, A. a Lefohn, A. Neural Temporal Adaptive Sampling and Denoising. *Computer Graphics Forum.* Květen 2020, sv. 39, s. 147–155. DOI: 10.1111/cgf.13919.

[5] Kajiya, J. T. The Rendering Equation. *SIGGRAPH Comput. Graph.* New York, NY, USA: Association for Computing Machinery. srpen 1986, sv. 20, č. 4, s. 143–150. DOI: 10.1145/15886.15902. ISSN 0097-8930. Dostupné z: https://doi.org/10.1145/15886.15902.

[6] Lefrançois, M.-K. a Gautron, P. *DX12 Raytracing tutorial - Part 1.* Jan 2020. Dostupné z: https://developer.nvidia.com/rtx/raytracing/dxr/DX12-Raytracing-tutorial-Part-1.

[7] Mara, M., McGuire, M., Bitterli, B. a Jarosz, W. An Efficient Denoising Algorithm for Global Illumination. In: *Proceedings of High Performance Graphics.* New York, NY, USA: Association for Computing Machinery, 2017. HPG '17. DOI: 10.1145/3105762.3105774. ISBN 9781450351010. Dostupné z: https://doi.org/10.1145/3105762.3105774.

[8] NVDIA. *NVIDIA Turing GPU Architecture.* Dostupné z: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[9] Overbeck, R., Ramamoorthi, R. a Mark, W. R. Large ray packets for real-time Whitted ray tracing. In: *2008 IEEE Symposium on Interactive Ray Tracing.* 2008, s. 41–48. DOI: 10.1109/RT.2008.4634619.

[10] Perlin, K. An Image Synthesizer. In: *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: Association for Computing Machinery, 1985, s. 287–296. SIGGRAPH '85. DOI: 10.1145/325334.325247. ISBN 0897911660. Dostupné z: https://doi.org/10.1145/325334.325247.

[11] Perlin, K. Improving Noise. *ACM Trans. Graph.* New York, NY, USA: Association for Computing Machinery. červenec 2002, sv. 21, č. 3, s. 681–682. DOI: 10.1145/566654.566636. ISSN 0730-0301. Dostupné z: https://doi.org/10.1145/566654.566636.

[12] Phong, B. T. Illumination for Computer Generated Pictures. *Commun. ACM.* New York, NY, USA: Association for Computing Machinery. červen 1975, sv. 18, č. 6, s. 311–317. DOI: 10.1145/360825.360839. ISSN 0001-0782. Dostupné z: https://doi.org/10.1145/360825.360839.

[13] Roth, S. D. Ray casting for modeling solids. *Computer Graphics and Image Processing*. 1982, sv. 18, č. 2, s. 109 – 144. DOI: https://doi.org/10.1016/0146-664X(82)90169-1. ISSN 0146-664X. Dostupné z: http://www.sciencedirect.com/science/article/pii/0146664X82901691.

[14] Schied, C., Kaplanyan, A., Wyman, C., Patney, A., Chaitanya, C. R. A. et al. Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination. In: *Proceedings of High Performance Graphics*. New York, NY, USA: Association for Computing Machinery, 2017. HPG '17. DOI: 10.1145/3105762.3105770. ISBN 9781450351010. Dostupné z: https://doi.org/10.1145/3105762.3105770.

[15] Stich, M. *Introduction to NVIDIA RTX and DirectX Ray Tracing*. Aug 2020. Dostupné z: https://developer.nvidia.com/blog/introduction-nvidia-rtx-directx-ray-tracing/.

[16] Takahashi, D. *How Pixar made Monsters University, its latest technological marvel*. VentureBeat, Dec 2018. Dostupné z: https://venturebeat.com/2013/04/24/the-making-of-pixars-latest-technological-marvel-monsters-university/.

[17] Whitted, T. An Improved Illumination Model for Shaded Display. *Commun. ACM.* New York, NY, USA: Association for Computing Machinery. červen 1980, sv. 23, č. 6, s. 343–349. DOI: 10.1145/358876.358882. ISSN 0001-0782. Dostupné z: https://doi.org/10.1145/358876.358882.

[18] Wyman, C., Hargreaves, S., Shirley, P. a Barré Brisebois, C. Introduction to DirectX Raytracing. In: *ACM SIGGRAPH 2018 Courses*. August 2018.

[19] Zink, J., Pettineo, M. a Hoxley, J. *Practical Rendering and Computation with Direct3D 11*. 1st. USA: A. K. Peters, Ltd., 2011. ISBN 1568817207.

Příloha A

Obsah přiloženého média

Přiložené médium obsahuje:

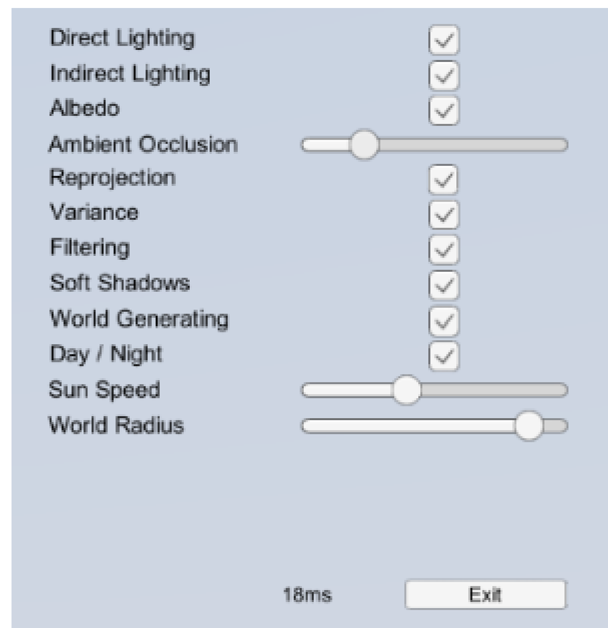
- **`demo.mp4`**: demonstrační video,
- **`exec/`**: složka s přeloženými soubory,
- **`src/`**: složka se zdrojovými soubory,
- **`tex/`**: složka se zdrojovými soubory textové zprávy,
- **`manual.pdf`**: návod k použití aplikace,
- **`README.txt`**: popis obsahu média,
- **`xmensi03.pdf`**: technická zpráva.

Příloha B

Ovládání aplikace

Aplikace se ovládá pomocí klávesových zkratek a menu. Menu je zobrazeno na obrázku B.1. Slouží k nastavení grafických efektů, množství světa či rychlosti střídání dne a noci. Použité klávesové zkratky jsou:

- **`W`**, **`A`**, **`S`**, **`D`**: pohyb do stran,
- **`Space`**: pohyb nahoru,
- **`Shift`**: pohyb dolů,
- **`M`** nebo **`Esc`**: otevření menu.



Obrázek B.1: Menu pro ovládání efektů a běhu aplikace.