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

REALISTICKÉ ZOBRAZOVÁNÍ VOXELOVÝCH SCÉN V REÁLNÉM ČASE

REAL-TIME PHOTOREALISTIC RENDERING OF VOXEL SCENES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. PETR FLAJŠINGR**
AUTHOR

VEDOUCÍ PRÁCE **Ing. TOMÁŠ MILET**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

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

Zadání diplomové práce

Student: **Flajšingr Petr, Bc.**
Program: Informační technologie
Obor: Počítačová grafika a multimédia
Název: **Realistické zobrazování voxelových scén v reálném čase**
Real-Time Photorealistic Rendering of Voxel Scenes
Kategorie: Počítačová grafika
Zadání:

1. Nastudujte techniky fotorealistického zobrazování v reálném čase, zobrazování voxelových modelů, API Vulkan a akcelerace pomocí GPU.
2. Navrhněte aplikaci umožňující realistické zobrazování voxelových modelů v reálném čase s využitím GPU.
3. Implementujte navrženou aplikaci.
4. Proměřte a zhodnoťte.
5. Vytvořte demonstrační video.

Literatura:

- Dle doporučení vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2 a kostra aplikace.

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

Vedoucí práce: **Milet Tomáš, Ing.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 30. července 2021
Datum schválení: 30. října 2020

Abstrakt

Tato práce se zabývá implementací realistického zobrazování voxelových scén za využití grafické karty. Práce vysvětluje základy realistického zobrazování a voxelovou reprezentaci vizuálních dat. Představuje také některé hierarchické struktury použitelné pro akceleraci. Popisuje návrh řešení zaměřující se na reprezentaci voxelových dat a jejich vykreslování. Práce popisuje knihovny vytvořené v rámci práce na projektu a aplikované algoritmy. Vyhodnocuje časovou a paměťovou náročnost aplikace a její grafické výstupy.

Abstract

The subject of this thesis is an implementation of realistic rendering of voxel scenes using a graphics card. This work explains the fundamentals of realistic rendering and voxel representation of visual data. It also presents selected hierarchical structures usable for acceleration and describes the desing of a solution focusing on the representation of voxel data and their rendering. The thesis describes libraries created as part of the project and algorithms. It also evaluates time and memory requirements of the application along with graphical output.

Klíčová slova

Vulkan, Voxel, Ray casting, Octree, gpu, Sparse voxel octree, Light field probes

Keywords

Vulkan, Voxel, Ray casting, Octree, gpu, Sparse voxel octree, Light field probes

Citace

FLAJŠINGR, Petr. *Realistické zobrazování voxelových scén v reálném čase*. Brno, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Milet

Realistické zobrazování voxelových scén v reálném čase

Prohlášení

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

. . . .
Petr Flajšingr
27. července 2021

Poděkování

Chtěl bych poděkovat vedoucímu práce panu Ing. Tomáši Miletovi za vedení práce a rozsáhlé odborné konzultace. Dále děkuji Anně Mašátové a Dominikovi Dvořákovi za pomoc s úpravou tohoto textu.

Obsah

1 Úvod **2**

2 Teorie **3**
2.1 Realistické zobrazování . . . 3
2.2 Voxel . . . 9
2.3 Hierarchické struktury . . . 14
2.4 Vulkan API . . . 16

3 Návrh řešení **18**
3.1 Reprezentace voxelových dat . . . 18
3.2 Vykreslování . . . 19
3.3 Struktura vykreslovacího řetězce . . . 28

4 Implementace **30**
4.1 Použité knihovny a nástroje . . . 30
4.2 Knihovna pf_common . . . 31
4.3 Knihovna pf_glfw_vulkan . . . 31
4.4 Knihovna pf_imgui . . . 32
4.5 Struktura programu . . . 33
4.6 Načítání modelů a jejich transformace . . . 34
4.7 Výpočet průsečíku se scénou . . . 35
4.8 Správa modelů . . . 36
4.9 Demonstrační aplikace . . . 39

5 Vyhodnocení **41**
5.1 Převod vstupních dat na octree . . . 41
5.2 Paměťová náročnost . . . 42
5.3 Výsledky . . . 44

6 Závěr **48**

Literatura **50**

A Obsah přiloženého paměťového média **53**

B Konfigurační soubor **54**

Kapitola 1

Úvod

Nejpopulárnějším způsobem modelování a reprezentace dat pro vykreslování je v současné době využití trojúhelníkových sítí, ale existují i další přístupy. Jednou z poměrně populárních alternativ je využití voxelů. I přes to, že použití voxelové reprezentace v real-time grafice není tolik rozšířené, lze nalézt spoustu zdrojů a vědeckých článků o této problematice.

Vzhledem ke změnám možností využití grafické karty, především existenci compute shaderů, je práce s voxely mnohem jednodušší, než tomu bylo v dřívějších dobách. Není nutné provádět převod na trojúhelníky a je možné pracovat s voxelovými primitivy přímo. Tím se odemyká možnost využití optimalizovaných algoritmů. Tato práce se zabývá zobrazováním voxelových scén za pomocí moderních metod a nástrojů, které jsou vývojářům poskytnuty.

Cílem této práce je navrhnout a implementovat systém pro vykreslování voxelových scén. Z mnoha dostupných metod jsou zde využity dvě – sparse voxel octree ray tracing z článku [21] a light field probes z článku [26]. Pro light field probes byla vytvořena modifikace k výpočtu a vykreslení nepřímého osvětlení ve scéně.

První kapitola si dává za cíl seznámit čtenáře s řešenou problematikou. Vysvětluje základní principy realistického zobrazování a některé populární metody, jimiž je toho možné dosáhnout. Také popisuje, co jsou voxely, jak se s nimi dá pracovat v kontextu rozdělení prostoru a základní metody vykreslování. Dále je zde rozebrána problematika vybraných hierarchických struktur, pomocí kterých lze urychlit průchod voxelovým prostorem. V poslední řadě obsahuje krátký popis moderního API pro práci s grafickými kartami: Vulkan.

Kapitola druhá se zabývá návrhem řešení. Je zde popsán způsob reprezentace voxelových dat s využitím akcelerační struktury a její detaily. Kapitola také vysvětluje navrhovaný způsob vykreslování voxelových scén s využitím ray casting přístupu pro primární paprsky a představuje způsob výpočtu nepřímého osvětlení za pomocí sond. V poslední sekci je vysvětleno jak do sebe jednotlivé části zapadají ve větším celku.

Následující kapitola se zabývá specifikami implementace. Nejprve jsou uvedeny nástroje a knihovny využité v samotné implementaci. Dále jsou popsány knihovny, které byly v rámci práce vytvořeny: knihovna obsahující obecné nástroje, knihovna pro usnadnění práce s Vulkan a GLFW, a knihovna pro vytváření a interakci s uživatelským rozhraním. Také je zde popsána základní struktura výsledného programu společně se specifikami implementace jako je transformace modelů či správa modelů v paměti. Poslední sekce krátce popisuje uživatelské rozhraní demonstrační aplikace.

Poslední kapitola se věnuje vyhodnocením dosažených výsledků. Obsahem je analýza doby převodu scén do interní reprezentace, paměťová náročnost využitých datových struktur a také vyhodnocení dosažených grafických výstupů implementace.

Kapitola 2

Teorie

Cílem této kapitoly je seznámit čtenáře s koncepty nutnými pro pochopení vlastního řešení. Kapitola nejprve definuje a vysvětluje, co je to voxel a jaká jsou jeho hlavní využití v počítačové grafice, a dále také popisuje fotorealistické zobrazování a vykreslování voxelových scén obecně. Je zde taktéž popsána charakteristika často používaných akceleračních struktur pro vykreslování voxelových modelů.

2.1 Realistické zobrazování

Tato sekce popisuje princip a použití různých technik realistického zobrazování používaného v současné počítačové grafice. Jsou zde uvedeny pouze relevantní informace související s tématem a předpokládá se čtenářova znalost základních termínů.

Realistické zobrazování je v knize [14] popsáno jako tvorba obrazu podle definovaného modelu scény a v ní přítomného osvětlení. Jednotlivé pixely ve vytvořeném snímku (angl. rendered image) se dají chápat jako množství světla procházejícího podél paprsků ve scéně, což odpovídá integrálu vstupující světelné energie v bodě, případně v regionu.

Dle knihy [31] je základní komponentou zobrazování v počítačové grafice zobrazovací řetězec (angl. rendering pipeline). Funkcí zobrazovacího řetězce je vygenerování dvourozměrného snímku daného virtuální kamerou, scénou obsahující vykreslované modely a zdroji světla.

V článku [17] je představena zobrazovací rovnice. Jedná se o integrální rovnici, jež zobecňuje přenos světelných paprsků ve scéně. Tato rovnice je uvedena v rovnici 2.1,

$$I(x,x') = g(x,x')\left[\varepsilon(x,x') + \int_S \rho(x,x',x'')I(x',x'')dx''\right] \tag{2.1}$$

kde $I(x,x')$ je intenzita světla z bodu x' do bodu x, $g(x,x')$ je geometrický term reprezentující zastínění povrchu mezi body x a x', $\varepsilon(x,x')$ značí intenzitu emitujícího světla z bodu x' do bodu x a $\rho(x,x',x'')$ představuje intenzitu rozptýleného světla z bodu x'' do bodu x přes plochu povrchu x'. Integrál je vypočten přes $S = \bigcup S_i$, tedy jako sjednocení všech povrchů, kdy S_0 je dostatečně velká hemisféra uzavírající celou scénu.

Jak je dále uvedeno v článku [17], rovnice vychází z fyzikální rovnice radiozity. Popisuje přenos intenzity světla z jednoho povrchového bodu do jiného jako sumu emitovaného osvětlení a celkové intenzity rozptýleného světla v bodě x od všech okolních povrchových bodů.

Podobná zobrazovací rovnice byla představena souběžně s dříve zmíněnou rovnicí v článku [16]. Tato rovnice je popsána pomocí vektorů a je v literatuře používána častěji. Nejznámější formu zobrazovací rovnice popisuje kniha [14], je uvedena v rovnici 2.2,

$$L_{out}(P,\omega_0) = L_e(P,\omega) + \int_{\omega_i \in S^2(p)} L_{in}(P,-\omega_i) f_s(P,\omega_i,\omega_0)(\omega_i \cdot \mathbf{n}_P) d\omega_i \tag{2.2}$$

kde $L(P,\omega)$ je příchozí (L_{in}) či odchozí (L_{out}) světlo v bodě P ve směru ω, $\mathbf{n}_P$ je normála povrchu v bodě P, ω_i značí příchozí směr světla, ω_0 analogicky odchozí směr světla a S je sjednocení všech povrchů. L_{out} je světlo vyzářené pryč z bodu P ve směru ω_0 (tedy směrem k pozorovateli), L_e je emitované světlo, L_{in} je světlo přicházející z ω_i a f_s je obousměrná distribuční funkce odrazu světla (BRDF; světlo odražené z ω_i k ω_0).

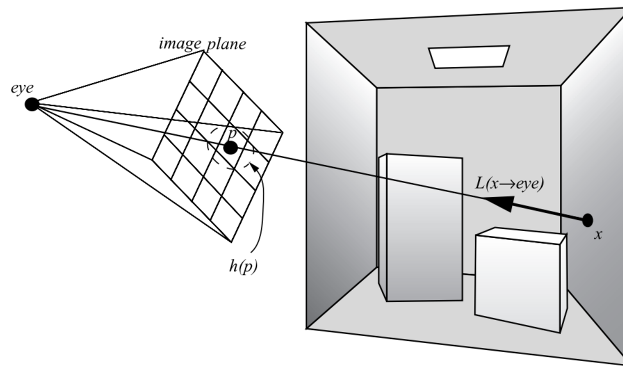
Existuje mnoho metod realistického zobrazování, přičemž následující část textu obsahuje základní popis vybraných metod.

2.1.1 Ray tracing

Ray tracing, neboli sledování paprsku, je pravděpodobně nejznámější metodou globálního osvětlení. Kniha [30] popisuje ray tracing jako algoritmus počítající radiační hodnotu L_{pixel} pro každý pixel ve výsledném obrázku. Tato hodnota je váženou hodnotou osvětlení ve scéně obsaženou v cestě paprsku. Výpočet popisuje rovnice 2.3,

$$L_{pixel} = \int_{imageplane} L(p \to eye)h(p)dp = \int_{imageplane} L(x \to eye)h(p)dp \tag{2.3}$$

kde p je bod na rovině obrázku, $h(p)$ váhová či filtrační funkce a x je viditelný bod od kamery skrze p.



Obrázek 2.1: **Vrhání paprsku do scény.** Počátkem paprsku je *eye*, nebo-li umístění kamery. Paprsek dále prochází *image plane*, což je rovina odpovídající výslednému vykreslenému obrázku. Převzato z Advanced global illumination [30].

Jak napovídá již samotný název – ray tracing používá paprsky k výpočtu barvy výsledného obrázku. Paprsek je polopřímka a je tedy definován počátečním bodem $\vec{o}$ (origin) a směrem $\vec{d}$ (direction). Nutnou funkcí pro funkčnost tohoto algoritmu je výpočet průsečíku paprsku s primitivy, pomocí kterých je scéna vytvořena. Možnosti výpočtu průsečíku paprsku s voxelem jsou uvedeny v sekci 2.2.1. Jednoduchý ray tracing je popsán algoritmem 1. Takto zjednodušená implementace je v praxi samozřejmě velice neefektivní a pro běžně používané scény je nutné použít akcelerační struktury pro urychlení hledání průsečíku jak uvádí kniha [8]. Pro realistické zobrazování jsou kromě primárních paprsků generovány i paprsky sekundární. Tyto paprsky jsou použity například pro odrazy, refrakci a stíny.

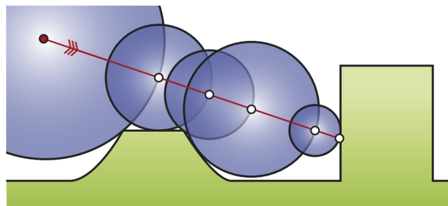
Ve své standardní formě ray tracing neumožňuje generování měkkých stínů a spousty dalších sekundárních efektů. Pro dosažení kýženého efektu lze využít například **Monte carlo ray tracing** (stochaistic ray tracing nebo distributed ray tracing) [4]. Namísto jediného paprsku pro výpočet stínů, odrazů a refrakcí je využito paprsků více a výsledky jejich výpočtu jsou následně průměrovány. Metoda umožňuje vytvoření mnoha dalších efektů, mezi které patří například hloubka pole, rozmazání pohybu a další.

Algoritmus 1: Ray tracing

```
1 ray = build_ray(camera.position, image_plane);
2 min_distance = MAX;
3 hit_primitive = false;
4 foreach primitive in scene do
5     (intersected, distance) = intersect(ray, primitive);
6     if intersected and distance < min_distance then
7         hit_primitive = primitive;
8         min_distance = distance;
```

Ray marching

Další metoda, kterou by bylo vhodné zmínit, je taktéž algoritmus ray marching (sphere tracing). Namísto přímého výpočtu průsečíku se scénou – jak tomu je u spousty ray tracing implementací – prochází paprsek scénou postupně, dokud nedojde k průsečíku.



Obrázek 2.2: **Ray marching.** Kružnice zobrazují vzdálenost současné pozice paprsku ke scéně, což odpovídá kroku paprsku v současné iteraci. Převzato z Enhanced Sphere Tracing [19].

Článek [12] popisuje algoritmus ray marching následovně: pro funkčnost algoritmu je podmínkou, aby existovala možnost vypočítat z každého bodu ve scéně vzdálenost k jejímu povrchu. Pro tento účel lze využít takzvaných signed distance functions (SDF). Každý paprsek, pro něž jeho generování probíhá stejně jako u již zmíněného ray tracing, se iteruje napříč scénou, dokud do ní paprsek nenarazí, nepřekoná maximální počet iterací či maximální vykreslovací vzdálenost (algoritmus 2).

Algoritmus 2: Ray marching

```
1  i = 0;
2  t = t_min;
3  while t < t_max and i < MAX_ITERATIONS do
4      distance_to_scene = dist_func(ray);
5      if distance_to_scene < EPSILON then
6          return t;
7      t += distance_to_scene;
8      i += 1;
9      ray.origin += distance_to_scene;
10 return t_max;
```

Pro tento algoritmus existuje značné množství optimalizací [19], jako je například "over-relaxation", kdy dochází k záměrně většímu kroku a případnému návratu zpět. Dalším příkladem je namísto sledování paprsku použít kužel, což výrazně snižuje množství iterací algoritmu.

Light field probes

Metoda globální iluminace využívající ray tracing byla představena v publikaci [26]. Pracuje na předpokladu, že kontinuální světelné pole scény $\mathcal{L}(x, \omega)$, kde $x \in \mathcal{R}^3$ je bod v prostoru a $\omega \in S$ je odchozí směr, reprezentuje distribuci osvětlení ve scéně pro všechny body a směry ve scéně. Light field probes reprezentuje tuto distribuci pomocí diskrétního vyobrazení.

Prostor ve scéně je rozdělen pomocí pravidelné mřížky. Do každé diskrétní pozice x' je umístěna instance jedné sondy (probe), která mapuje směry ω kolem x' na: intenzitu osvětlení $\mathcal{L}(x, \omega)$, normály $\vec{n}_x$ v bodech x'' nejblíže bodu x' a hloubkovou mapu mezi x' a body x''. K výpočtu těchto hodnot dochází při přípravě scény.

Obrázek 2.3: **Umístění sond ve scéně.** Pro obecné případy jsou sondy rozmístěny ve scéně rovnoměrně ve formě mřížky. Převzato z Real-Time Global Illumination Using Precomputed Light Field Probes [26].

Algoritmus vykreslování postupuje napříč připravenými sondami. Při průchodu paprskem dochází k výběru sondy, trasování uvnitř ní a následně je tento proces opakován napříč scénou, dokud nedojde k jistému protnutí nebo minutí geometrie scény.

2.1.2 Radiozita

Metody radiozity byly vyvinuty již v padesátých letech minulého století pro simulaci tepelného přenosu. Později byla popsána varianta pro vykreslování v článku [10]. Metoda je založena na jednoduchém principu – vzhledem k tomu, že každý povrch ve scéně může odrážet světlo, je možné tento povrch považovat za zdroj světla. Radiozita povrchu je dána rovnicí 2.7,

$$B_i = E_i + \rho_i \sum_{j=1}^{N} B_j F_{ij} \text{ pro } i = 1 \text{ do } N \tag{2.4}$$

kde B je celkové množství energie vyzařované z povrchu, E je množství energie vyzařované z povrchu bez vlivu okolí, ρ je faktor reflexivity, F je faktor určující jaká část energie dorazila na povrch a N je počet povrchů ve scéně.

Z autorova průzkumu vyplývá, že je radiozita primárně používána pro předvýpočet globální iluminace v některých scénách a dále není opakovaně počítána. Důvodem je nejspíše poměrně značná náročnost algoritmu.

2.1.3 Materiály

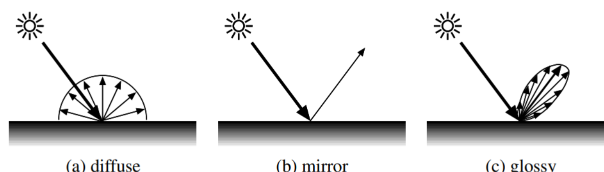
Pro simulaci interakce světla s povrchy je nutné mít tyto povrchy popsány určitými parametry. Použité parametry závisí na osvětlovacím modelu a také renderovacím algoritmu. Publikace [22] popisuje interakci světla s povrchem rovnicí 2.5,

$$(x, y, \theta, \phi, t, \lambda)_{in} \rightarrow (x, y, \theta, \phi, t, \lambda)_{out} \tag{2.5}$$

kde levá strana reprezentuje foton interagující s povrchem a pravá strana foton vycházející ven, (x, y) je pozice na povrchu, (θ, ϕ) je příchozí/odchozí směr, t je čas interakce a λ je vlnová délka. Pro zjednodušení lze z rovnice odstranit čas, čímž je předpokládáno, že se vzhled povrchu s časem nemění. Při diskretizaci vlnových délek je možné dosáhnout dalšího zjednodušení, produkující takzvanou BSSRDF (bidirectional scattering surface distribution function). Dalším zjednodušením může být ignorování podpovrchového rozptylu světla (subsurface scattering), jehož výsledkem je již známé BRDF (bidirectional reflectance distribution function) (rovnice 2.6),

$$\begin{aligned} f_r(\vec{c}, \hat{\omega_i} \rightarrow \hat{\omega_0}) &= \frac{dL_0(\vec{x}, \hat{\omega_0})}{dE(\vec{x}, \hat{\omega_i}} \\ &= \frac{dL_0(\vec{x}, \hat{\omega_0})}{L_i(\vec{x}, \omega_i) \cos\hat{} \theta_i d\omega_i} \end{aligned} \tag{2.6}$$

kde L_0 je podíl intenzity světla vycházející z povrchu na bodu $\vec{x}$ ve směru $\hat{\omega_0}$ a intenzity světla příchozího do bodu $\vec{x}$ ze směru $\hat{\omega_i}$.



Obrázek 2.4: **Typy odrazu světla.** (a) difuzní, (b) zrcadlové, (c) lesklé. Převzato z Realistic Materials in Computer Graphics [22].

Dle knihy [15] lze materiály dělit do několika základních kategorií podle typu interakce se světlem, přičemž tyto typy jsou uvedeny v tabulce 2.1. Vizualizace některých možných odrazů světla znázorňuje obrázek 2.4.

materiál	dominantní distribuce
průhledný nemetalický	difuzní odraz
metalický	zrcadlový odraz
průsvitný	difuzní přenos
průhledný	běžný přenos

Tabulka 2.1: **Typy materiálů a jejich interakce se světlem.**

Dalším podstatným parametrem je tvar povrchu, který výrazně mění to, jak dochází k odrazu světla. Vzhledem k potenciální složitosti této vlastnosti se používají například následující metody:

- Mapování normál (normal mapping)
- Parallax mapping

Příklady některých možných vlastností materiálů:

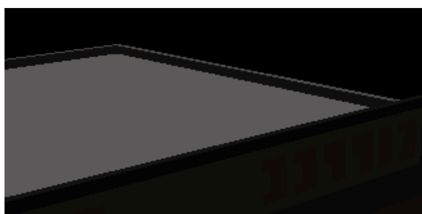
- barva
- hrubost
- metalické
- mapa normál
- emisivita
- průhlednost
- odrazivost
- spousta dalších...

2.1.4 Antialiasing

Aliasing je nechtěný efekt při zpracování signálu, kdy dochází ke vzniku artefaktů při vzorkování. Při vykreslování se jedná především o problém při vzorkování textur a artefakty na hranách objektu či přechodu mezi nimi (obrázek 2.5). Existuje množství metod pro odstranění tohoto efektu. Podle článku [25] lze metody pro antialiasing dělit do následujících kategorií:

- Full-Scene Anti-Aliasing (FSAA)
- Image Post-Processing Anti-Aliasing (IAA)
- Geometric Anti-Aliasing (GAA)

Každá z těchto kategorií má mírně odlišný přístup k problému aliasingu s různými limitacemi.



Obrázek 2.5: **Aliasing.** Ukázka aliasingu vznikajícího kvůli nedostatečnému rozlišení výsledného obrázku.

Příkladem FSAA je **Super Sampling Anti-Aliasing (SSAA)**, kde je počet pixelů pro renderování navýšen oproti cílové velikosti renderovaného obrázku. Při dokončení jsou přilehlé pixely a jejich barva/hloubka zprůměrovány. Tato metoda dosahuje výborných výsledků, ale je velmi výpočetně náročná. Optimalizovanou alternativou je **Multisample Anti-Aliasing (MSAA)**. Jedná se o metodu fungující na stejném principu, ale namísto navýšení rozlišení celého renderovaného snímku dochází k výběru oblastí, u kterých je velká pravděpodobnost vzniku aliasingu a super sampling probíhá pouze v těchto oblastech.

Z rodiny IAA uveďme **Morphological Anti-Aliasing (MLAA)**. Tato metoda si dává za cíl minimalizovat aliasing z okrajů a siluet. Algoritmus detekuje podezřelé oblasti podle rozdílu sousedních pixelů a používá rozmazání s okolím. Detekce aliasingu může být i složitější, jako například vyhledávání specifických tvarů.

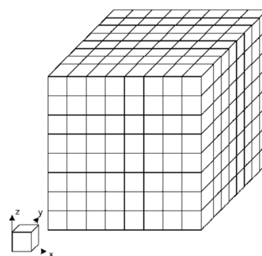
Vybraný zástupce GAA je **Geometric PostProcessing AA (GPAA)**. Při renderování obrazu dochází k ukládání informací o geometrii scény do separátního bufferu. Podle vzdálenosti sousedních pixelů ve finálním kroku dochází k rozhodnutí, zda je nutné provádět anti-aliasing. Pokud ano, jsou vypočteny hodnoty okolních pixelů a dochází k rozmazání.

Obrázek 2.6: **Rozdíl kvality po použití SSAA.** Levá polovina obrázku obsahuje snímek, ve kterém není využit anti-aliasing. V pravé části bylo využito SSAA.

2.2 Voxel

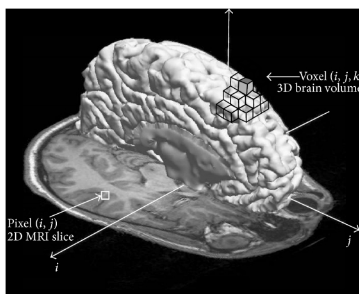
Jak uvádí kniha [14], voxel, neboli **vo**lume **el**ement, reprezentuje hodnotu na pravidelné mřížce ve 3D prostoru (obrázek 2.7). Díky tomu, že je prostor rozdělen mřížkou pravidelně, lze voxel definovat pomocí tří-složkového vektoru (rovnice 2.7). Pro účely vykreslování jsou voxelům přiřazovány další vlastnosti, jako například barva nebo materiál.

$$\vec{pozice}_{voxel} \in \mathbb{Z}_3 \tag{2.7}$$



Obrázek 2.7: **3D mřížka.** Převzato z Applied Mathematics and Computation [9].

Voxely jsou často využívány v medicíně [1], například pro výstupy magnetické resonance (obrázek 2.8).



Obrázek 2.8: **Zobrazení výsledku magnetické resonance pomocí voxelů.** Převzato z Computational and Mathematical Methods in Medicine [5].

Dalším častým využitím je modelování terénu, kde voxely přináší možnost reprezentovat převisy, čímž je terén značně realističtější než je tomu při aplikaci často používaných výškových map. Pravděpodobně nejznámější software používající voxely pro terén je **Minecraft** (obrázek 2.9). Za zmínku stojí také hra **Teardown** (2020)[1], kde je celý herní svět vytvořen pomocí voxelů a umožňuje téměř neomezenou destrukci.

Obrázek 2.9: **Minecraft.** Převzato z https://forums-cdn.spongepowered.org/uploads/default/original/2X/8/83abd20efc6cf5104c2f8c5459808bcd1addef7a.jpg.

[1] https://www.teardowngame.com/

2.2.1 Vykreslování voxelových modelů

Níže jsou popsány základní metody vykreslování voxelových modelů. Jedná se jen o základní příklady. Ve skutečnosti existuje mnohem více metod, než by bylo rozumné zde popisovat.

Rasterizace

Pro vykreslování voxelových scén pomocí rasterizace je nutné ji převést na trojúhelníkovou reprezentaci. Toho se dá dosáhnout několika způsoby.

Instanced rendering je velice primitivní metoda. Pro každý voxel, který je reprezentován přímo svojí pozicí a případně dalšími parametry (barva...), je vykreslena krychle o předem určené délce hran. Před samotným vykreslováním musí docházet k odstranění takových voxelů, které nejsou viditelné. Pokud by byl tento krok vynechán, bylo by vykreslování velice náročné. [28]

Algoritmus 3: Instancované vykreslování

```
1 voxel_array = cull_voxels(all_voxels);
2 foreach voxel in voxel_array do
3     render_cube(voxel.position);
```

Marching cubes [23] je příkladem algoritmu pro extrakci povrchu z voxelových dat. Původně představen pro vizualizaci dat v medicínském odvětví, je současně využíván například pro vizualizaci terénu [27]. Pro každou oblast v mřížce prostoru je zjištěno, který z rohů voxelu se nachází uvnitř či vně tělesa. Na základně toho je vypočten tvar a pozice generovaných trojúhelníků. V některých úpravách algoritmu je možné značně snížit počet vygenerovaných trojúhelníků a snížení redundantnosti dat. Po vygenerování často dochází k minimalizaci počtu trojúhelníku. Algoritmus 4 obsahuje zjednodušenou verzi této metody.

Algoritmus 4: Marching cubes

```
1 foreach voxel in area do
2     case = calculate_case(voxel);
3     triangles = generate_triangles(voxel.position, case);
4     result.add(triangles);
```

Existují další metody, jako například **marching tetrahedra**, ale tato práce rasterizačních metod nevyužívá a proto zde nebudou dále zmiňovány.

Ray casting

Při vykreslování pomocí paprsků lze problém rozdělit na dvě části. První z nich je výpočet průsečíku s voxelem a druhým je využití hierarchické struktury pro minimalizaci počtu navštívených voxelů. Hierarchické struktury jsou obsaženy v samostatné sekci.

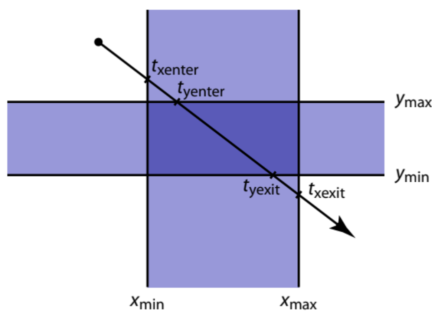
Průsečík paprsku s voxelem lze počítat mnoha způsoby. Primitivní přístup k řešení tohoto problému je počítat průsečík s každou rovinou, která reprezentuje voxel a následně vybrat ten nejbližší (algoritmus 5). Tohle řešení je ale náročné a využívá podmínky, což může podle článku [11] značně zpomalit výpočet.

Algoritmus 5: Primitivní výpočet průsečíku s voxelem

```
1  corner1 = voxel.position;
2  corner2 = voxel.position + voxel_length;
3  coeffs[0] = (corner1.x - ray.origin.x) / ray.direction.x;
4  coeffs[1] = (corner1.y - ray.origin.y) / ray.direction.y;
5  coeffs[2] = (corner1.z - ray.origin.z) / ray.direction.z;
6  coeffs[3] = (corner2.x - ray.origin.x) / ray.direction.x;
7  coeffs[4] = (corner2.y - ray.origin.y) / ray.direction.y;
8  coeffs[5] = (corner2.z - ray.origin.z) / ray.direction.z;
9  hit = false;
10 distance = inf;
11 foreach coef in coefs do
12     if coef >= 0 then
13         hit = true;
14         hit_point = ray.origin + ray.direction * coef;;
15         if is_in_box_bounds(corner1, hit_point) then
16             distance = coef;
```

Vhodnější alternativu pro výpočet průsečíku paprsku s voxelem lze najít v publikaci [18]. Tato metoda využívá dlaždic (slabs), kdy je voxel považován za průsečík tří z nich. Na obrázku 2.13 je vizualizace výpočtu. Algoritmus pro 2D je popsán v rovnici 2.8.

$$
\begin{aligned}
x_{min} &= p_x + t_{xmin} d_x \\
t_{xmin} &= \frac{(x_{min} - p_x)}{d_x} \\
\\
y_{min} &= p_y + t_{ymin} d_y \\
t_{ymin} &= \frac{(y_{min} - p_y)}{d_y} \\
\\
t_{xenter} &= \min(t_{xmin}, t_{xmax}) \\
t_{xexit} &= \max(t_{xmin}, t_{xmax}) \\
\\
t_{yenter} &= \min(t_{ymin}, t_{ymax}) \\
t_{yexit} &= \max(t_{ymin}, t_{ymax}) \\
\\
t_{exter} &= \max(t_{xenter}, t_{yenter}) \\
t_{exit} &= \min(t_{xexit}, t_{yexit})
\end{aligned}
\tag{2.8}
$$

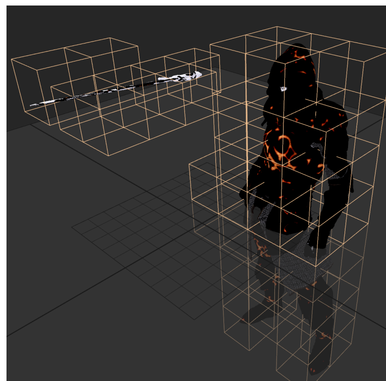


Obrázek 2.10: **Průsečík s obdélníkem metodou dlaždic.** Vizualizace výpočtu ve 2D. Ve 3D se výpočet provádí velice podobně. Převzato z Ray Tracing: intersection and shading [24].

2.2.2 Používané formáty

Pro ukládání voxelových scén existuje poměrně velké množství formátů. Bohužel však neexistuje žádný standardizovaný formát, i když nějaké pokusy o standardizaci se již objevily (například VolDat[2]). Následující část textu popisuje vybrané formáty.

Vox formát byl vytvořen pro aplikaci MagicaVoxel[3], což je modelovací software pro voxelové scény. Jedná se o binární formát, kde jsou barvy zakódovány pomocí palety. Formát podporuje také různé typy materiálů, ač s malým množstvím parametrů. Scéna je složena z částí (modelů), které mají maximální velikost 256x256x256 voxelů – při načítání je tedy nutné rozhodnout, zda je celý soubor jedním modelem, nebo je rozdělen na chunky. Specifikace formátu jsou v [6].



Obrázek 2.11: **VOX model rozdělený na chunky.** Dělba modelů na chunky, zobrazeno pomocí MagicaVoxel.

SVX (Simple Voxels) je archivový formát. Archiv obsahuje soubor manifest.xml, který popisuje velikost mřížky, velikost voxelů, paletu materiálů a další metadata. Samotné voxely jsou popsány pomocí obrázků v jednokanálovém formátu PNG, přičemž každý pixel slouží jako odkaz do palety modelů. Tento formát je používán především pro 3D tisk. Specifikaci formátu lze nalézt na [13].

Existuje spousta dalších formátů, značné množství pro 3D tisk, ale také pro využití v medicíně, jak již bylo zmíněno dříve. Dle autorova průzkumu je nejčastějším přístupem při volbě formátu pro vykreslovací engine vytvoření vlastního formátu.

[2] http://www.volumesoffun.com/voldat-format/

[3] http://ephtracy.github.io/

VolDat byl vytvořen jako pokus o jistou formu standardizace voxelových formátů. Tento formát byl popsán v [33]. Scéna je rozdělena na 2D části (průřezy napříč osou Y) a každý tento průřez je uložen v obrázku. Pixely obrázku reprezentují jednotlivé voxely, jejich hodnota vyjadřuje barvu a také odkaz do separátního textového souboru, kde jsou uloženy doplňující data, jako například informace o materiálech.

2.3 Hierarchické struktury

Jak již bylo zmíněn – pro efektivní práci s velkým množstvím voxelů je potřeba využít akceleračních struktur. V této sekci je obsažen popis některých struktur, které se dají pro voxely použít.

2.3.1 Mřížka

Mřížka, nebo také grid, je poměrně jednoduchá struktura pro dělbu prostoru. Na nejnižší úrovni může mřížka odpovídat té, pomocí které dělí prostor samotné voxely. V takovém případě by však nedošlo k žádnému zrychlení. Pro akceleraci je prostor obsahující voxely rozdělen do několika částí (tzv. chunks). V těchto oblastech jsou voxely sdružovány a při hledání požadované položky je možné nejdříve vybrat správný chunk a teprve následně vyhledávat v omezené množině voxelů. Samotná struktura má minimální paměťové nároky, ale nepřináší velké zrychlení, alespoň ve srovnání s ostatními metodami. Jednoduchá trojrozměrná mřížka je na obrázku 2.7.

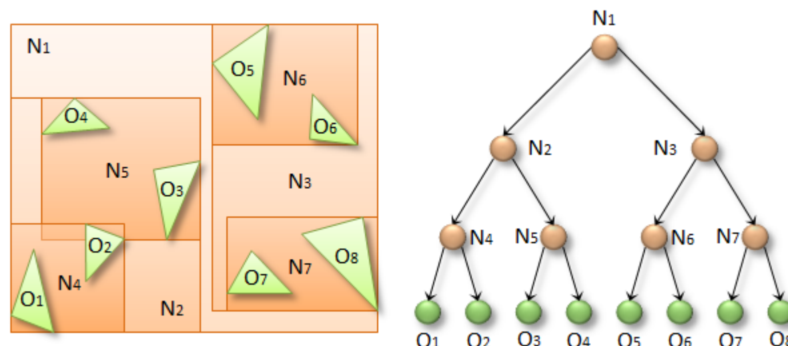
2.3.2 Hierarchie obalových těles

Obalové těleso (bounding volume) je jednoduchý geometrický objekt, který obaluje jeden či více objektů s větší komplexitou [7]. Důležitým faktorem pro vhodnost tělesa k využití v hierarchii obalových těles (bounding volume hierarchies) pro ray tracing je náročnost výpočtu průsečíku s tělesem. Jako příklad lze uvést kouli (rovnice 2.9), osově zarovnaný obdélník (axis aligned box) nebo konvexní obálku.

$$\begin{gathered} R(t) = \mathbf{o} + t\mathbf{d} \\ (\mathrm{P} + \mathrm{t}\mathbf{d} - \mathrm{C})\,(\mathrm{P} + \mathrm{t}\mathbf{d} - \mathrm{C}) = \mathrm{r}^2 \end{gathered} \tag{2.9}$$

$$\begin{aligned} R &= \text{paprsek} \\ \mathbf{o} &= \text{počátek paprsku} \\ t &= \text{vzdálenost průsečíku} \\ \mathbf{d} &= \text{směr paprsku} \\ C &= \text{střed koule} \\ r &= \text{poloměr koule} \end{aligned}$$

Samotná hierarchie těchto těles je tvořena jako stromová struktura, kde každý nelistový uzel obsahuje zpravidla 2 potomky. Obalové těleso reprezentující nelistové uzly je vytvořeno tak, aby ohraničovalo pouze prostor nutný k obsažení jeho potomků.



Obrázek 2.12: **2D hierarchie obalových těles.** *O* reprezentuje objekty ve scéně a *N* uzly stromu. Převzato z https://developer.nvidia.com/blog/thinking-parallel-part-ii-tree-traversal-gpu/.

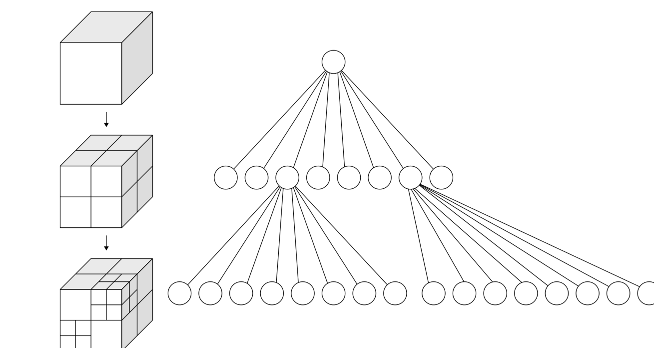
Díky snadnosti výpočtu průsečíků a možnosti netestovat průsečíky s nižšími úrovněmi stromu – a tím i komplexními objekty – může dojít k výraznému zrychlení výpočtu průsečíku se scénou. Algoritmus 6 popisuje jednoduchý průchod binárním stromem za pomoci zásobníku.

Algoritmus 6: Průchod BVH stromem pro ray tracing [32]

```
1  current_node = root;
2  stack = empty();
3  exit = false;
4  while not exit do
5      if isInternalNode(current_node) then
6          if intersectNode(current_node) then
7              pushStack(stack, current_node.first_child);
8              current_node = current_node.second_child;
9      else
10         intersectLeaf(current_node);
11         (current_node, exit) = popStack(stack);
```

2.3.3 Octree

Octree, poprvé představen v knize [20], je popsán jako hierarchický N-dimenzionální binární strom, jenž reprezentuje N-dimenzionální objekt. Pro účely tohoto textu je podstatná pouze jeho 3D varianta. Každý uzel stromu obsahuje 8 potomků na další úrovni. Tyto uzly reprezentují oblast v prostoru. Pokud uzel kompletně popisuje oblast, kterou reprezentuje, jedná se o list nebo koncový uzel. V opačném případě musí obsahovat 8 potomků pro podoblasti. Při hledání objektů v prostoru tedy lze procházet pouze velmi omezenými oblastmi a neplýtvat výpočetním výkonem.



Obrázek 2.13: **Octree.** Převzato z https://en.wikipedia.org/wiki/Octree.

Sparse voxel octree

Jedná se o renderovací metodu využívající ray casting/tracing pro vizualizaci voxelových scén. Důležitou částí generování octree pro tento algoritmus je minimalizace uzlů, které nejsou viditelné, či spojení stejných uzlů do jednoho bloku na vyšší úrovni [21]. Výhodou je velmi snadná aplikace level of detail napříč stromem. Podle velikosti pixelu je možné zastavit průchod stromem před dosažením nejnižší úrovně a vypočítat výslednou barvu z uzlu na vyšší úrovni. Algoritmus 7 popisuje jednoduchou verzi algoritmu.

Algoritmus 7: Sparse voxel octree ray casting

```
1  voxel = tree.get_root();
2  while not terminated do
3      (hit, t) = intersect_cube(ray, voxel);
4      if hit then
5          if is_voxel_small(voxel, pixel_size) or is_voxel_leaf(voxel) then
6              return t;
7          stack.push(voxel);
8          voxel = select_child(voxel);
9          continue;
10     else
11         voxel = stack.pop();
12 return false;
```

2.4 Vulkan API

Vulkan je API pro 3D grafiku a výpočty pomocí GPU [29]. Je produktem Khronos Group. Jedná se o nízkoúrovňové rozhraní s nízkou režií, které je zároveň multiplatformní. Produkty postavené na tomto API lze tedy spouštět na velkém množství různých systémů. Oproti OpenGL, které je jistým způsobem předchůdcem Vulkanu, je ve Vulkanu nutné nastavovat výrazně větší množství parametrů. Díky této vlastnosti má programátor podstatně větší kontrolu nad stavem grafické karty. Nevýhodou ovšem je značné zvýšení komplexity programu oproti jednodušším API. Některé výhody, které Vulkan poskytuje oproti jiným grafickým API:

- Jednotné API jak pro mobilní zařízení, tak pro PC.

- Dostupnost na velkém množství operačních systému (podobně jako OpenGL).
- Nízký overhead.
- Pro shadery používá binární formát SPIR-V[4], díky čemuž mohou vývojáři distribuovat pouze binární formu shaderů.
- Sjednocení grafického (graphics pipeline) a výpočetního (compute shaders) API.
- Ray tracing pomocí rozšíření (tuto funkci podporuje i DirectX12).

Současnou verzí API je Vulkan 1.2.184[2].

[4] https://www.khronos.org/opengl/wiki/SPIR-V

Kapitola 3

Návrh řešení

Následující část práce popisuje návrh aplikace pro realistické zobrazování voxelových scén. Kapitola je rozdělena na část popisu reprezentace voxelových dat a dále jejich vykreslování pomocí sparse voxel octree algoritmu.

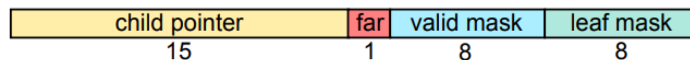
3.1 Reprezentace voxelových dat

Jak již bylo popsáno v sekci 2.2 – voxely jsou reprezentovány svou pozicí a jejich materiální vlastnosti různými parametry. Logickým krokem je rozdělení těchto dvou vlastností. V některých strukturách je lze kódovat implicitně a nemusíme tedy plýtvat paměťovým prostorem.

Pro implicitní zakódování pozice lze využít octree (sekce 2.3.3). Nejnižší úroveň stromu reprezentuje samotné voxely a jediným nutným parametrem je počáteční pozice stromu a velikost prostoru, který obaluje. Samotný strom by mohl být vytvořen se statickým rozestupem rodičů a potomků, kde by se výpočet indexu následníka mohl řídit rovnicí:

$$I_{child_j} = (i * 8) + j \tag{3.1}$$

kde I_{child_j} je index j-tého potomka, i je index současného uzlu. Při použití tohoto rozložení by ovšem musela být celá oblast uvnitř octree rozdělena na nejmenší bloky a docházelo by k obrovskému plýtvání paměti.



Obrázek 3.1: **Položka reprezentující uzel octree.** Převzato z [21].

Úsporný způsob pro reprezentaci octree byl představen v článku [21]. Metoda kóduje informace o uzlu bitově na data o velikosti 4 bytů. Vizualizace je na obrázku 3.3. Každý nelistový uzel je reprezentován svými daty a listové uzly jsou implicitně kódovány pomocí masek. `Child pointer` určuje pozici potomků uzlu pro daný uzel, přičemž je uložen jako rozdíl od pozice současného uzlu, čímž je možno ho reprezentovat menším rozsahem. Přepínač `far` určuje, zda `child pointer` odkazuje na pozice potomků, či na 32 bitový ukazatel na ně - při velkých stromech nemusí být 15 bitů dostačující. Tento pointer je uložen v oddělené části bufferu. `Valid mask` je bitová maska o velikosti 8 bitů, pokud je hodnota nastavena na 1, pak je ve stromu potomek na tomto indexu reprezentován. `Leaf mask` odpovídá funkci `valid mask`. Dává nám ovšem vědět, jestli je po-

tomek na indexu terminálním uzlem. Pokud je tedy nastavena na 1, tak je vyhledávání ve stromu na této úrovni ukončeno.

Kromě informací o obsazenosti prostoru je nutné ukládat i informace o materiálu. První možností by bylo rozšířit záznamy ve stromu o tato data, ale s velkým množstvím parametrů by velikost stromu rapidně narostla. Proto by bylo vhodnější tato data ukládat separátně. Tyto informace také nejsou relevantní při procházení stromu a pokud by byla uložena ve struktuře, došlo by ke zhoršení prostorové lokality paměti. Vhodným řešením je za oblastí stromu vytvořit oblast s ekvivalentní strukturou, kde jsou uloženy odkazy na materiálová data. Výpočet offsetu do struktury je triviální, jelikož je pozice dat na stejném indexu jako je index uzlu. Nalezené položky slouží k vyhledávání specifického záznamu v separátním bufferu. Výhodou tohoto přístupu je možnost snadno ukládat samostatné materiály pro vyšší úrovně stromu, což může být použito pro aplikaci level of detail.

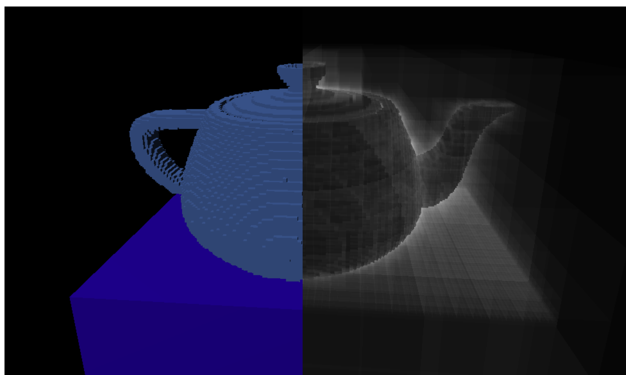
Ukládání voxelových dat Pro načítání modelů je využit formát VOX, který byl popsán v sekci 2.2.2. Při načítání je ovšem nutné data transformovat do formátu octree popsaného výše, což může způsobit výrazné zpomalení aplikace. Namísto opakované transformace by bylo tedy vhodné data ukládat tak, aby docházelo pouze k načtení – tím by došlo k výraznému zrychlení. Vzhledem k tomu, že je důležitá spíše rychlost a ne využití místa na disku, je dostačující serializovat strukturu popsanou výše přímo do binárního souboru. Zpravidla dojde k malému zvětšení oproti VOX, ale doba načítání se zkrátí na zlomek času.

3.2 Vykreslování

Následující sekce popisuje metody využité pro vykreslování scény v této práci – jmenovitě ray marching pomocí sparse voxel octrees, light field probes a jejich modifikace pro nepřímé osvětlení.

3.2.1 Sparse voxel octree

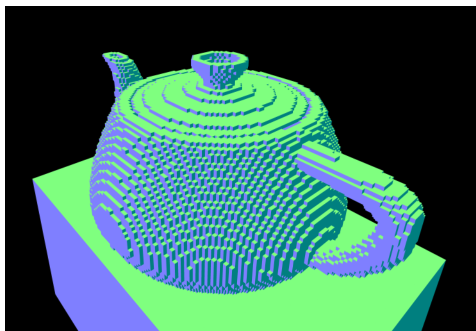
Pro rendering voxelové scény existuje několik možností, přičemž vhodným kandidátem je přístup sparse voxel octree (sekce 2.3.3). Při vržení paprsku dochází k procházení stromu a vcházení pouze do podoblastí, které mohou obsahovat vyhledávaný voxel. Algoritmus je velice úsporný oproti primitivním metodám. Tato metoda je popsána v [21].



Obrázek 3.2: **Render voxelizované "Utah teapot".** Levá část vyobrazuje běžné vykreslení, pravá obsahuje počet iterací algoritmu v daném pixelu – plně bílá barva je v tomto případě ~128 iterací.

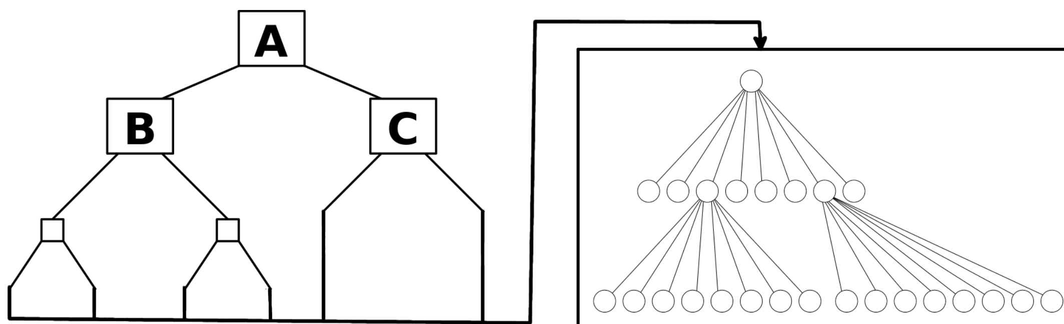
Vzhledem ke způsobu uložení dat představenému v předešlé kapitole je v rámci algoritmu nutné dopočítávat množství indexů uzlů na základě masek (`valid mask` a `leaf mask`). Jedná se ale zpravidla pouze o operace sčítání či odčítáním, tedy operace s nízkou výpočetní náročností.

Pro výpočet průsečíku s voxelem je vhodné využít optimálnější metody popsané v sekci 4. Důležitou součástí pro stínování je také získání normály povrchu. Vzhledem k tomu, že cílem autora je zachovat "krychlovitý"tvar voxelů, je nalezení normály při využití slab metody pro nalezení průsečíku poměrně triviální.



Obrázek 3.3: **Vizualizace normál pro model "Utah teapot".**

Pro usnadnění práce s modely a úsporu výpočetního výkonu při jejich transformaci má každý model svoji vlastní octree strukturu. Nutnost otestovat průsečíky se všemi těmito modely by byla velice náročná a proto je vhodné použít další akcelerační strukturu k urychlení nalezení průsečíku. Autor práce zvolil hierarchii obalových těles (sekce 2.3.2) s využitím osově zarovnaných obdélníku. Každý listový uzel této hierarchie tedy obsahuje jeden octree, který definuje voxely modelu.



Obrázek 3.4: **Rozložení hierarchií scény.** Levá část reprezentuje hierarchii obalových těles, kde se v každém listu nachází octree.

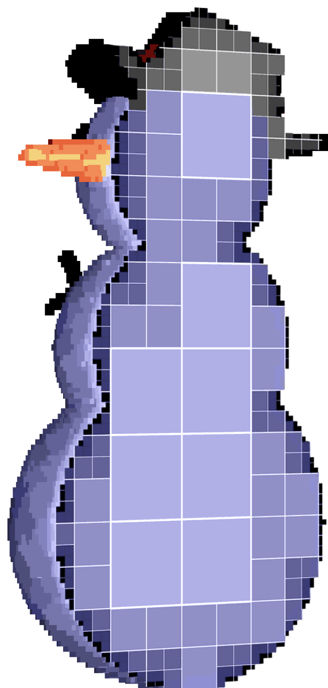
Regularita dat uložených pomocí octree je klíčovým faktorem umožňujícím efektivní sledování paprsku. Většina dat asociovaných s voxelem je uložena v jeho rodiči a samotný voxel je tedy reprezentován indexem jeho rodiče a indexem voxelu v rozsahu $[0, 7]$. Jelikož struktura neukládá prostorové informace, je také nutné v průběhu sledování paprsku skrze octree udržovat informace o velikosti a pozici zkoumaného voxelu. Voxel je v prostoru tedy reprezentován vektorem $\vec{pos}$, který je v rozsahu $[0, 1]$ v každé dimenzi a pozitivním celým číslem $scale$, které určuje délku hran voxelu. Vzhledem k využití těchto vlastností se model vždy nachází v prostoru souřadnic v rozsahu $[0, 1]$ a pro aplikaci transformací a následné efektivní sledování paprsku je nutné paprsek transformovat pomocí inverzní transformace do tohoto prostoru.

Definujme paprsek jako $p_t(t) = p + t \cdot d$. Cílem je získat hodnotu t, tedy vzdálenost k průsečíku paprsku a jednoho z voxelů patřících do octree. Pro osově zarovnanou rovinu získáme rovnici $t_x(x) = \frac{1}{d_x}x + -\frac{p_x}{dx}$ pro osu x. Obdobné rovnice jsou získány taktéž pro osy y a z. Osově zarovnanou krychli je možné reprezentovat jako dvojici rohů (bodů v prostoru) (x_0, y_0, z_0) a (x_1, y_1, z_1) tak, že $t_y(y_0) \leq t_y(y_1)$, $t_y(y_0) \leq t_z(z_1)$ a $t_z(z_0) \leq t_z(z_1)$. S touto definicí je rozsah t hodnot protnutých s krychlí dán rovnicemi $tc_{min} = \max(t_x(x_0), t_y(y_0), t_z(z_0))$ a $tc_{max} = \max(t_x(x_1), t_y(y_1), t_z(z_1))$.

Při průchodu voxely na stejné úrovni lze získat následující voxel stejné škály porovnáním hodnot $t_x(x_1), t_y(y_1)$ a $t_z(z_1)$ proti tc_{max}. Algoritmus se posune na další voxel v osách, ve kterých jsou si hodnoty rovné.

Jelikož je využitá octree struktura *sparse* – tedy neobsahuje informace o prázdném prostoru – je nutné provádět inkrementální průchod hierarchií. Algoritmus provádí prohledávání do hloubky, čili se snaží co nejdříve dostat na listové uzly pro nejrychlejší konvergenci k výsledku. Při každé iteraci může docházet k jednomu ze tří způsobů výběru dalšího zkoumaného voxelu:

- *PUSH* - Posun do potomka uzlu.
- *ADVANCE* - Posun do sourozeneckého uzlu.
- *POP* - Posun do uzlu, který byl uložen na zásobník.



Obrázek 3.5: **Reprezentace modelu pomocí sparse octree.** V obrázku je vidět, že oblasti pro které není nutné udržovat detail jsou velice úsporné.

Pro průchod octree je tedy nutné využít datové struktury zásobník. Kdykoli, kdy algoritmus postoupí do nižší úrovně stromu (*PUSH*), je potenciálně uložen předchozí rodič. Pokud je současná větev nevhodná pro další výpočty, je vyvolán *POP* a tím se posune algoritmus v hierarchii výše.

Při postupu v hierarchii níže dochází k volbě vhodného potomka. Volba je založena na vyhodnocení t_x, t_y a t_z a jejich porovnání s tc_{min}.

Tuto metodu autor zvolil pro výpočet primárních paprsků kvůli vysoké přesnosti a poměrně nízké výpočetní náročnosti tohoto algoritmu. Metoda pro výpočet sekundárních paprsků je popsána v následující sekci.

3.2.2 Light field probes

Light field probes byly již krátce představeny v sekci 2.1.1. Každá sonda je reprezentována svou pozicí a texturou, která obsahuje následující informace:

- Radiální vzdálenost ke geometrii.
- Normálu.
- Informace o materiálu, barvě či jiná data využitá v další práci se sondami.

Tato textura má velikost 1024x1024 a každý texel reprezentuje dvě float hodnoty (`rg32f`). Vzhledem k tomu, že paměťová náročnost je poměrně vysoká, je nutné ukládaná data nějakým vhodným způsobem komprimovat. Radiální vzdálenost ke geometrii je možné logaritmovat pro zachování vyšší přesnosti pro bližší objekty a také kvantovat na snížení paměťové náročnosti. Logaritmizace a linearizace hloubky obsahuje rovnice 3.2,

$$
\begin{aligned}
encoded_depth &= \frac{log(C \cdot depth + 1)}{log(C \cdot Far)} \\
decoded_depth &= \frac{(C \cdot Far + 1)^{encoded_depth} - 1}{C}
\end{aligned}
\tag{3.2}
$$

kde C je konstanta modifikující rozložení přesnosti, často nastavena na 1, $depth$ je hloubka a Far je maximální vzdálenost. Po logaritmizaci je hodnota transformována na 16 bitů a uložena do textury sondy.

Dalším parametrem jsou normály. Ty lze zakódovat na 2 float hodnoty mnoha metodami, jak je uvedeno v [3]. Autor zvolil metodu projekce na osmistěn. Po projekci jsou složky výsledného vektoru kvantovány na 8 bitů, čímž je celková potřebná paměť pro uložení těchto hodnot snížena na 16 bitů. Rovnice 3.3 popisuje zakódování normály a 3.4 její dekódování.

$$
\begin{aligned}
\vec{p} &= normal_{xy} \frac{1}{|normal_x| + |normal_y| + |normal_z|} \\
encoded\vec{_}normal &= \begin{cases} (1 - |p_{yx}|) \cdot sign(\vec{p}), & \text{pokud } normal_z \leq 0.0 \\ \vec{p}, & \text{jinak} \end{cases}
\end{aligned}
\tag{3.3}
$$

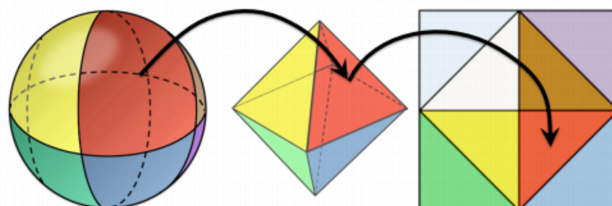
Kde $normal$ je normála, p je pomocný vektor k projekci a $encoded_normal$ je zakódovaná normála.

$$
\begin{aligned}
z &= 1 - |encoded_normal_x| - |encoded_normal_y| \\
\vec{v} &= \langle encoded_normal_{xy}, z \rangle \\
decoded\vec{_}normal &= \begin{cases} \langle (1 - |v_{yx}|) \cdot sign(v_{xy}), v_z \rangle, & \text{pokud } v_z < 0.0 \\ normalize(\vec{v}), & \text{jinak} \end{cases}
\end{aligned}
\tag{3.4}
$$

Kde $encoded_normal$ je zakódovaná normála, z je vypočtená třetí složka výsledné normály, v je pomocný vektor a $decoded_normal$ je dekódovaná normála.

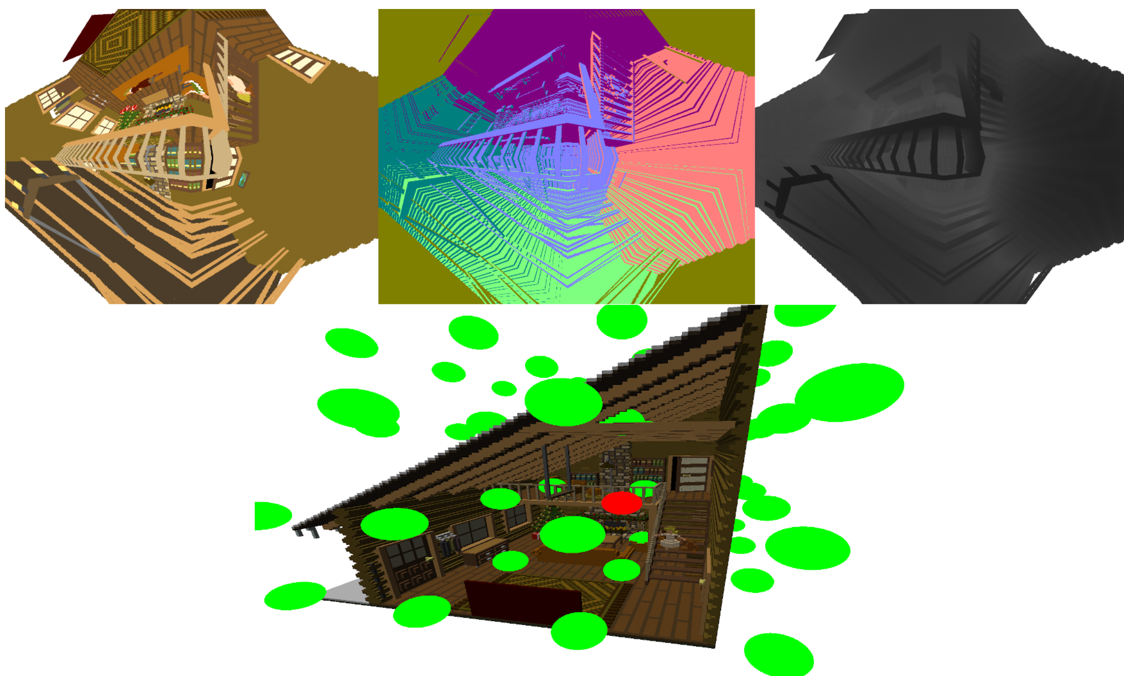
Poslední položkou uloženou v textuře sondy je radiance. Pro případné další rozšíření je uložena ve formátu `R11G11B10` pro podporu HDR.

Jelikož sonda reprezentuje světelné pole ve svém okolí jakožto kouli, je nutné data nějakým způsobem transformovat na texturu, tedy rovinu. Toho lze dosáhnout mapováním koule na osmistěn a následné projekce na rovinu, jak je znázorněno na obrázku 3.6. Pro tuto projekci a její inverzní operaci je možné využít rovnic 3.3 a 3.4.



Obrázek 3.6: **Vizualizace projekce koule na osmistěn a následně na rovinu.**

Výpočet informací o geometrii je prováděn pomocí ray tracingu představeném v sekci 3.2. Na obrázku 3.7 je ukázka dat jedné ze sond.



Obrázek 3.7: **Vizualizace dat uložených pro sondu.** V horní polovině obrázku je vyobrazena radiance, normály a hloubka. Dolní část ukazuje pozici sondy ve scéně (červená).

3.2.3 Renderování scény pomocí light field probes

Pomocí atlasu textur, který byl vytvořen pro všechny sondy pokrývající scénu, je možné provádět trasování paprsku. Tato metoda byla popsána v publikaci [26]. Díky radiální vzdálenosti a normálám zakódovaných v sondách je možné vypočítat průsečíky nesouvislých paprsků.

V prvé řadě je nutné pro paprsek vybrat vhodnou sondu, ve které zahájíme jeho sledování. Pokud tato sonda není schopná poskytnout nám jistý *miss* (minutí scény) nebo *hit* (protnutí scény)

je nutné vybrat alternativní sondu, která má nějakou šanci paprsek sledovat. Sledování paprsku v rámci jedné sondy tedy vrací tři stavy:

- *miss* – minutí scény
- *hit* – průsečík se scénou
- *unknown* – neznámý výsledek

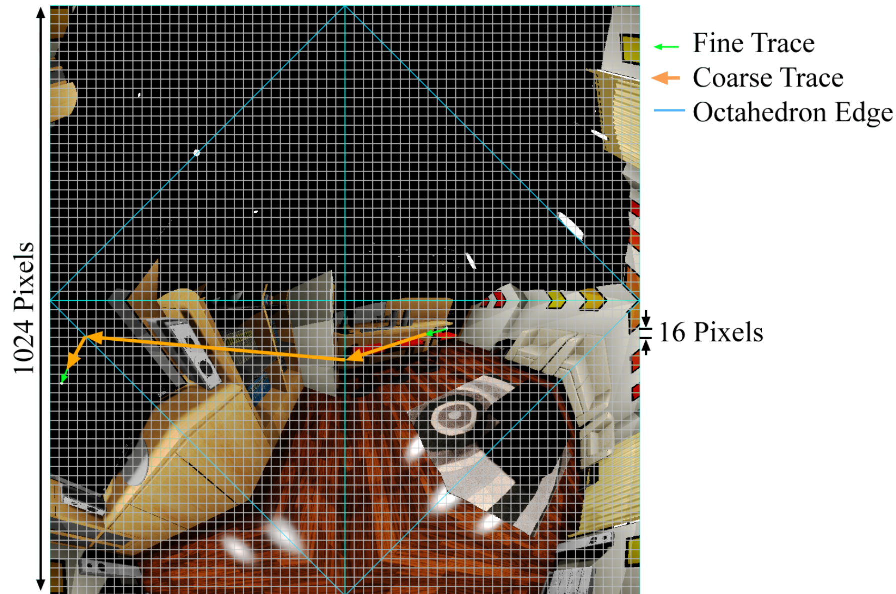
Stav *unknown* nastane pouze v případě, kdy paprsek prochází ve větší vzdálenosti, než jaká je v dané pozici radiální hloubka. Jinými slovy: pokud sonda "ví", že nemá dostatečné informace o scéně v dané pozici, nemůže být paprsek spolehlivě vyhodnocen pouze pomocí informací z této sondy.

Sledování paprsku začíná převedením jeho počátku do prostoru sondy. Následně je paprsek transformován pomocí projekce na osmistěn do souřadnic textury. Dochází tedy k transformaci přímky z R^3 do křivky v R^2. Tato křivka má až čtyři segmenty, kde každý segment leží v jiné stěně osmistěnu. Následně algoritmus postupuje po jednotlivých texelech nacházejících se na této křivce a vyhodnocuje, jestli dochází k průsečíku pomocí porovnávání uložené hloubkové mapy se vzdáleností paprsku k sondě. Pokud je paprsek ve stejné vzdálenosti nebo je dále, dochází k průsečíku nebo paprsek prochází za geometrií. Algoritmus pro výpočet průsečíku obsahuje algoritmus 8.

Algoritmus 8: Sledování paprsku v rámci jedné sondy

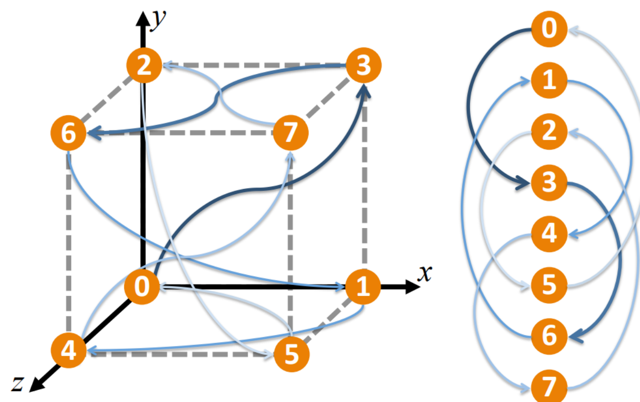
```
1  segments = compute_segments(ray, probe);
2  foreach segment in segments do
3      coords_on_segment = create_coords(segment);
4      foreach coord in coords_on_segment do
5          state = compare_ray_distance_to_radial(ray, coord, probe);
6          if state == HIT then
7              return (HIT, coord);
8          if state == HIDDEN_BEHIND_SURFACE then
9              return (UNKNOWN, coord) ;
10 return (MISS, last_point_on_segments);
```

Pro urychlení tohoto algoritmu je pro každou sondu vytvořena další textura, která má 16krát menší velikost – tedy 64x64 – a ve které jsou uloženy v každém texelu nejbližší radiální vzdálenosti. Jedná se prakticky o mipmapu. Při procházení pixelů segmentů tedy není nutné provádět algoritmus na maximálním rozlišení, čímž dochází k výraznému zrychlení. Pouze při nalezení potenciálního průsečíku je trasování prováděno na vysokém rozlišení. Na obrázku 3.8 je vizualizace průchodu paprsku jednou sondou.



Obrázek 3.8: **Sledování paprsku v textuře sondy.** Oranžová část paprsku je prováděna pouze na textuře s nižším rozlišením, zelená je prováděna na vysokém rozlišení pokud dochází k detekci potenciálního průsečíku. Převzato z Real-Time Global Illumination using Precomputed Light Field Probes [26].

Výběr sondy. Nejdůležitějším krokem je výběr vhodné sondy pro zahájení algoritmu. Jedním možným řešením je výběr sondy čistě podle vzdálenosti k počátku paprsku, což doporučuje taktéž článek [26], s tím, že výběr dalších sond se řídí pořadím uvedeným v obrázku 3.9. Pokud se paprsek nenachází v jiné skupině sond po dokončení průchodu této skupiny, je hledání ukončeno jako neúspěšné, jinak je prováděno v nové skupině. V případě autorova řešení docházelo k velkému množství *unknown* výsledků a proto se rozhodl implementovat jiné řešení.



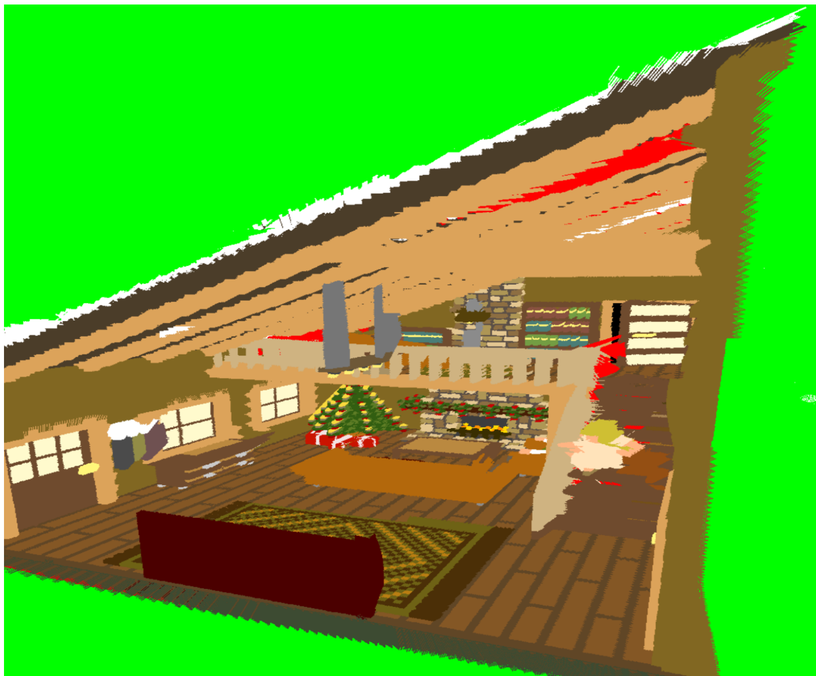
Obrázek 3.9: **Pořadí výběru sondy pro lokální skupinu.** Převzato z Real-Time Global Illumination using Precomputed Light Field Probes [26].

Namísto výběru sondy tak, jak je popsáno v předchozím odstavci, je možné rozdělit prostor, jenž je sondami sledován, na voxelové pole (mřížku). Toto pole obsahuje pro každý voxel až čtyři sondy, pro které se daný voxel vyskytuje blíže, než je jejich radiální vzdálenost ke scéně. Namísto výběru nejbližší sondy je tedy sonda vybrána dle toho, ve kterém voxelu se paprsek současně nachází. Sledování je ukončeno pouze tehdy, pokud žádná ze sond uložená v tomto poli nenalezla řešení a sledovaný bod paprsku se neposunul do jiného voxelu.

Algoritmus 9: Sledování paprsku skrze light field

```
1 result = UNKNOWN;
2 while result == UNKNOWN do
3     probe = select_probe_from_voxel_field(ray);
4     if not is_valid_probe(probe) then
5         break;
6     result, endpoint = trace_single_probe(ray, probe);
7     ray.origin = endpoint;
8 return result;
```

Na obrázku 3.10 je zobrazena scéna vykreslená čistě pomocí výše popsané metody. Stínování zde není aplikováno.

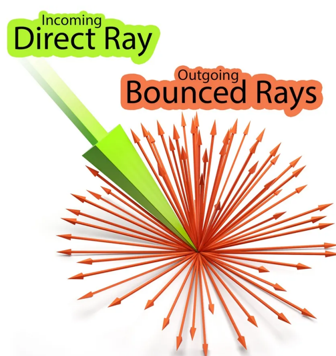


Obrázek 3.10: **Scéna vykreslena algoritmem sledování paprsku skrze light field.** Zelené oblasti značí *miss*, červené *unknown*. Je vidět, že dochází k mnoha nepřesnostem a ve scéně, kde je spousta objektů dochází ke vzniku míst, kde sondy nenaleznou řešení.

3.2.4 Nepřímé osvětlení

Původním záměrem autora bylo využít výše popsané sondy k výpočtu sekundárních paprsků ve scéně online, ale ukázalo se, že pro takový přístup je tato metoda příliš pomalá. Bylo tedy nutné vymyslet alternativní řešení.

Pro již zmíněnou metodu se nabízí modifikace, která by umožnila přidat funkčnost nepřímého osvětlení tak, že do atlasu textur sond je vypočteno nepřímé osvětlení při jejich inicializaci a posléze jsou aplikovány při renderování scény. K výpočtu je použit ray tracing implementovaný pomocí metody popsané v první části sekce. Pro každý texel, který reprezentuje difuzní materiál, je vysláno několik paprsků z místa prvního průsečíku scény. Tyto paprsky se dále odráží ve scéně a sbírají světlo z vnějších zdrojů/emitujících materiálů.

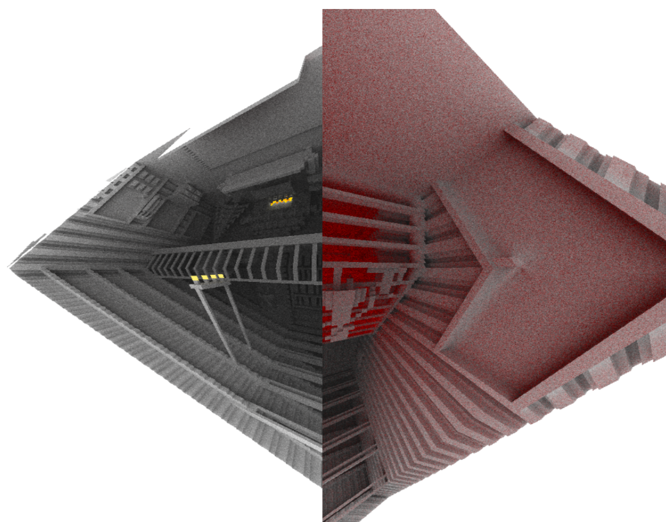


Obrázek 3.11: **Paprsky odražené z difuzního materiálu.** Převzato z https://renderstuff.com/tutorials/indirect-illumination-in-v-ray-tutorial-176/.

Jelikož informace o materiálech poskytované zdrojovými modely jsou poměrně stručné, odraz paprsku je prováděn náhodně v polokouli určené normálou, jak je demonstrováno na obrázku 3.12. Výpočet příspěvku osvětlení je popsán v algoritmu 10.

Algoritmus 10: Výpočet nepřímého osvětlení

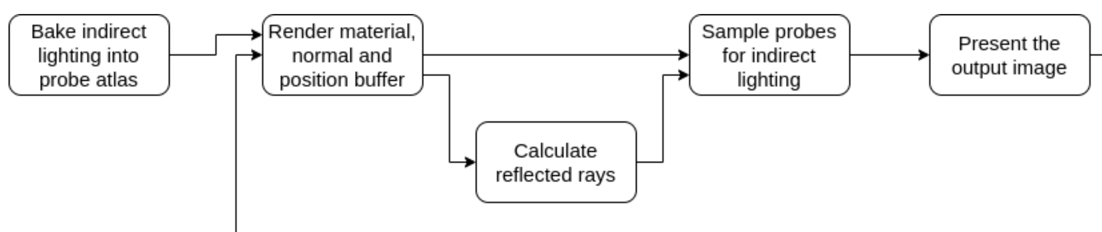
```
1  result = vector(1, 1, 1);
2  for i in range(0, MAX_BOUNCES) do
3      (is_hit, position, normal, material_type) = trace_ray(ray);
4      if not is_hit then
5          break;
6      if material_type == DIFFUSE then
7          result *= attentuation;
8          ray.origin = position;
9          ray.direction = random_point_on_hemisphere(normal);
10     if material_type == METALLIC then
11         ray = bounce_ray_for_metallic(ray, position, normal);
12     if material_type == EMIT then
13         add_emissive_light(result);
14         break;
15 return result;
```



Obrázek 3.12: **Textura sondy obsahující informace o nepřímém osvětlení.** Na levé straně je zdrojem pouze vnější osvětlení, vpravo je vidět vliv emitujícího materiálu.

3.3 Struktura vykreslovacího řetězce

K vykreslení finálního snímku je samozřejmě potřeba zkombinovat metody popsané v předchozí sekci. Jak je vidět z obrázku 3.10, používat light field probes jako primární vykreslovací algoritmus není vhodné, jelikož dochází k vzniku velkého množství nepřesností. Z tohoto důvodu je pro první fázi využit ray tracing ve sparse voxel octree – popsáno v sekci 3.2.1.



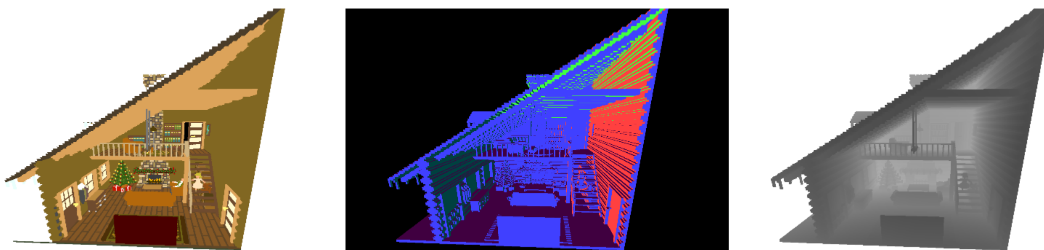
Obrázek 3.13: **Postup vykreslování jednotlivých snímků.** Výpočet nepřímého osvětlení pro sondy je prováděn pouze při inicializaci, případě jako reakce na požadavek uživatele.

Před zahájením vykreslování snímku je nutné připravit atlas sond. Pro "online"výpočet sekundárních paprsků by bylo dostačující připravit do atlasu radianci, normály a hloubku, vzhledem k vysoké časové náročnosti tato metoda nebude použita. Namísto radiance je tedy v atlasu sond uložen příspěvek nepřímého osvětlení (obrázek 3.12). Tento proces, s ohledem na jeho náročnost, může trvat delší dobu, ale je dostačující ho provést pouze při inicializaci scény. Celý řetězec je na obrázku 3.14.

Další krok je již opakován periodicky pro každý snímek. Pomocí sparse voxel octree je vykreslen buffer, který obsahuje informace o místě průsečíku scény, index materiálu a normálu – tento buffer je zobrazen v obrázku 3.14.

Následně je na základě hodnot v bufferu dohledáno ze sond a jejich atlasu nepřímé osvětlení. Díky použití mřížky k urychlení nalezení potenciálně vhodné sondy (sekce 3.2.3) lze očekávat, že tato operace nezabere příliš mnoho času. Také je možné, že pro daný fragment existuje několik

různých zdrojů dat o nepřímém osvětlení, jelikož tato data mohou být uložena ve více sondách, čímž se efektivně výrazně zvyšuje počet sekundárních paprsků. V tomto kroku paralelně též probíhá výpočet odražených paprsků pro metalické materiály.



Obrázek 3.14: **Složky bufferu vykreslené na začátku rámce.** Zleva se jedná o index materiálu (zobrazeno jako barva materiálu), normály, pozici ve světe (zobrazeno jako hloubka).

Kapitola 4

Implementace

Tato kapitola obsahuje seznam použitých nástrojů a knihoven. Popisuje implementaci knihoven vytvořených při práci na praktické části tohoto projektu, také algoritmy použité ke tvorbě octree a uživatelské rozhraní demonstrační aplikace.

4.1 Použité knihovny a nástroje

Pro implementaci autor zvolil jazyk `c++`, standard 20[1], kvůli autorově pokročilé znalosti tohoto jazyka a také jako možnost procvičit nově přidané funkce v poslední revizi. Pro překlad projektu byl použit `CMake`[2] v kombinaci s `Ninja`[3] a package management nástroji `CPM.cmake`[4] a `Hunter`[5]. Jako vývojové prostředí bylo použito `CLion`[6]. Kód shaderů je vytvořen v GLSL[7]. Při práci na projektu byly taktéž využity tyto knihovny třetích stran:

- `spdlog`[8]
- `range-v3`[9]
- `magic_enum`[10]
- `argparse`[11]
- `toml++`[12]
- `backward-cpp`[13]
- `cppcoro`[14]
- `{fmt}`[15]
- `stb`[16]
- `glm`[17]
- `Dear ImGui`[18]

Dále byly využity následující nástroje pro statickou analýzu kódu, formátování atp.:

[1] https://en.cppreference.com/w/cpp/20
[2] https://cmake.org/
[3] https://ninja-build.org/
[4] https://github.com/TheLartians/CPM.cmake
[5] https://hunter.readthedocs.io/en/latest/
[6] https://www.jetbrains.com/clion/
[7] https://www.khronos.org/opengl/wiki/Core_Language_(GLSL)
[8] https://github.com/gabime/spdlog
[9] https://github.com/ericniebler/range-v3
[10] https://github.com/Neargye/magic_enum
[11] https://github.com/p-ranav/argparse
[12] https://marzer.github.io/tomlplusplus/
[13] https://github.com/bombela/backward-cpp
[14] https://github.com/lewissbaker/cppcoro
[15] https://github.com/fmtlib/fmt
[16] https://github.com/nothings/stb
[17] https://github.com/g-truc/glm
[18] https://github.com/ocornut/imgui

- cppcheck[19]
- cpplint[20]
- sanitisers[21]
- valgrind[22]
- Ccache[23]
- clang-format[24]

4.2 Knihovna pf_common

Knihovna poskytující často používané funkce:

- Concepty a další nástroje pro template meta programování a statický polymorfimus.
- Jednoduché coroutines (iota, range...)
- Implementace výjimek se stack trace reporting.
- Některé idiomy c++ – RAII, Visitor...
- Funkce pro binární serializaci dat a práci s binárními daty.
- Obecné geometrické funkce, které nejsou poskytnuty v glm.
- Funkce pro rozšíření možnosti využití enum.
- Mnoho dalších...

Je implementována jako header-only knihovna[25].

4.3 Knihovna pf_glfw_vulkan

Tato knihovna poskytuje dvě základní funkce:

- vytvoření a správa okna pomocí GLFW[26]
- komunikace s GPU za pomocí Vulkan API[27]

Třída GlfwWindow umožňuje instanciaci GLFW knihovny a také komunikaci za pomocí událostí, které knihovna programu předává. Zprostředkovává tak uživateli možnost interagovat s aplikací za pomocí periferních zařízení (myš a klávesnice). Tato třída splňuje concept[28] Window, který je také obsažen v knihovně. Za pomoci Window je implementována komunikace mezi Vulkan a okenním systémem. Díky tomuto rozdělení si může uživatel vytvořit komunikační vrstvu mezi knihovnou a jiným okenním systémem, jako například SDL[29]. Pro tento účel implementuje knihovna vlastní notifikační systém událostí ve třídě EventDispatchImpl, který výrazně usnadňuje případnou implementaci s jiným okenním backendem.

Část knihovny pro interakci s Vulkan je o poznání rozsáhlejší. Při návrhu knihovny byl kladen zřetel především na její jednoduchost a zároveň bezpečnost jejího užívání. Z tohoto důvodu je uvnitř hojně využito std::shared_ptr. Vzhledem k tomu, že ve Vulkan je nutné často

[19] http://cppcheck.sourceforge.net/
[20] https://github.com/cpplint/cpplint
[21] https://github.com/google/sanitizers
[22] https://valgrind.org/
[23] https://ccache.dev/
[24] https://clang.llvm.org/docs/ClangFormat.html
[25] https://en.wikipedia.org/wiki/Header-only
[26] https://www.glfw.org/
[27] https://www.khronos.org/vulkan/
[28] https://en.cppreference.com/w/cpp/language/constraints
[29] https://www.libsdl.org/

vytvářet objekty v závislosti na některém z dříve vytvořených (například vk::Instance ⟶ vk::PhysicalDevice ⟶ vk::Device), si v sobě každý nově vytvořený potomek ukládá ukazatel na svého "rodiče". To zaručuje, že nemůže dojít k uvolnění objektů omylem, v případě, že budeme používat některého z jeho potomků.

Knihovna obsahuje vlastní verzi velkého množství Vulkan objektů a zároveň je plně typově bezpečná. Příkladem může být přístup do vk::Buffer, kdy k datům přistupujeme pomocí mapovacího objektu, který ve velké míře využívá template funkcí pro kontrolu offsetu a správné práce s pamětí.

Všechny objekty jsou potomkem VulkanObject, což je rozhraní, které poskytuje základní debug informace o objektu. Vytváření objektů je vždy prováděno pomocí struct konfiguračních dat. Předpokládá se užití designated initialisers[30]. Ukázka vytvoření logického zařízení je ve výpisu 4.1.

```
// tvorba instance vynechana kvuli velkemu mnozstvi argumentu
device = instance->selectDevice(DefaultDeviceSuitabilityScorer());
surface = instance->createSurface(window);
logicalDevice = device->createLogicalDevice({
   .id = "dev1",
   .deviceFeatures = vk::PhysicalDeviceFeatures{},
   .queueTypes = {vk::QueueFlagBits::eCompute},
   .presentQueueEnabled = true,
   .requiredDeviceExtensions = {VK_KHR_SWAPCHAIN_EXTENSION_NAME},
   .validationLayers = getValidationLayers(),
   .surface = *surface});
```

Výpis 4.1: Tvorba logického zařízení

Pro objekty, u nichž by konfigurační struktura vyžadovala nadměrně velké množství argumentů, jsou v knihovně dostupné builder třídy, například GraphicsPipelineBuilder nebo RenderPassBuilder.

Součástí knihovny je též rozhraní pro kompilaci shader souborů z disku nebo paměti.

4.4 Knihovna pf_imgui

pf_imgui je event-driven UI knihovna postavena na Dear Imgui. Cílem je zjednodušení práce při vytváření uživatelského rozhraní a také přidání některých funkcí. Funkce, které jsou nad rámec Dear Imgui přidané, jsou:

- Pozorování změn hodnot pomocí callback funkcí (observer pattern).
- Ukládání hodnot do konfiguračního souboru. Tyto změny jsou načteny vždy při zapnutí programu.
- Některé elementy navíc, například Memo.

Knihovna je postavena nezávisle na renderovacím backendu, v repository je poskytnut backend pro Vulkan a GLFW. Pro vlastní použití je nutné rozšířit hlavní objekt ImGuiInterface a přidat volání do backendu své volby. Například pro Vulkan by se mohlo jednat o jednoduchou funkci

[30] https://en.cppreference.com/w/cpp/language/aggregate_initialization#Designated_initializers

přidání draw command do vk::CommandBuffer. V knihovně je obsaženo podstatné množství elementů, přičemž jsou implementovány wrappery pro všechny elementy dostupné v Dear Imgui. Také jsou přidány grafy z knihoven ImGuiFlameGraph[31], implot[32] a možnost interagovat se soubory na disku pomocí ImGuiFileDialog[33].

V knihovně je také implementováno podstatné množství layoutů – grid layout, anchor layout a další.

Přidání vlastních elementů do knihovny je velice primitivní. Připravená rozhraní pokrývají spoustu potenciální funkčnosti, a to například včetně *drag and drop* a rozhraní pro nastavení stylu/barvy elementu.

Kupříkladu implementace CheckBox vyžaduje pouze ~10 řádků kódu – nepočítaje deklarace funkcí a tělo konstruktoru – s tím, že dědí z ItemElement, Labellable, Savable, ValueObservable<bool>, ColorCustomizable<..> a StyleCustomizable<..>.

Ukázka vytvoření tlačítka, které reaguje na kliknutí otevřením dialogu pro výběr txt souboru je ve výpisu 4.2.

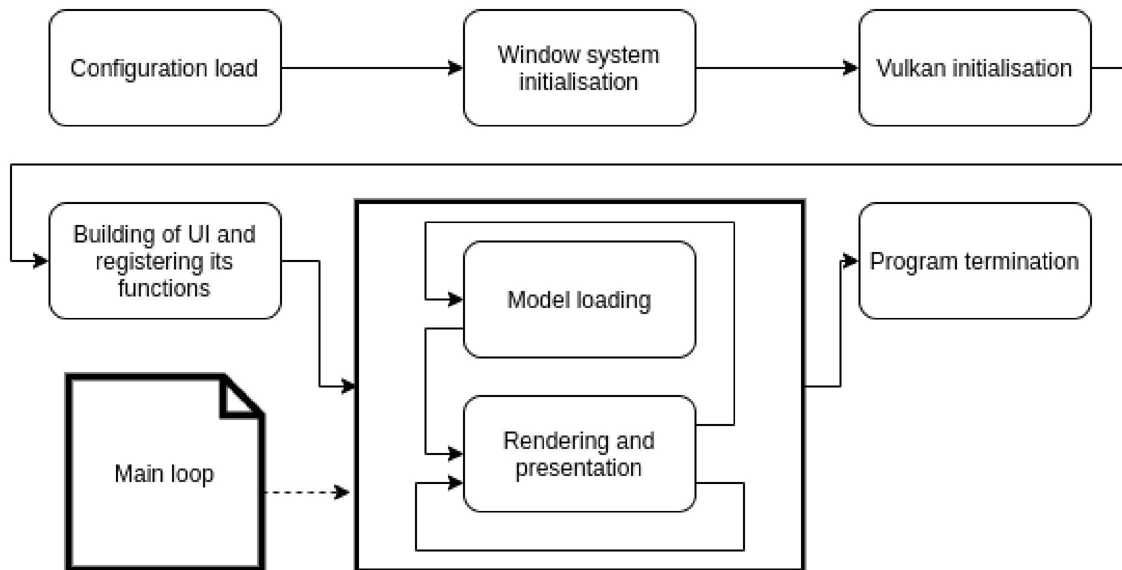
```
imgui.createChild<Button>("btn_id", "Open file")
   .addClickListener([&imgui] {
      imgui.openFileDialog("Select file",
         {.ext = {"txt"}, .description = "txt files"},
         [] (const auto &files) {
            for (const auto &file : files) { print(file); }
         }, [] { print("No file selected"); });
   }
```

Výpis 4.2: Vytvoření tlačítka pro výběr souboru

4.5 Struktura programu

Na obrázku 4.1 je diagram hlavní struktury programu. V diagramu jsou zobrazeny pouze větší části programu, kdy každá část je závislá na té předchozí. Prvním krokem je načtení konfigurace, která je blíže popsána v sekci 4.9. Následně dochází k inicializaci okenního systému za pomocí knihovny pf_glfw_imgui, stejná knihovna je využita v inicializaci Vulkan. Dále je vybudováno uživatelské rozhraní s tím, že jsou mu přiřazeny všechny potřebné funkce. Po tomto nastavení se program dostává do hlavní smyčky, kde jsou krom interakce uživatele s UI a kamerou prováděny dvě důležité operace – načítání modelů (částečně prováděno na separátním vlákně) a vykreslování společně s prezentací obrázku.

[31]https://github.com/bwrsandman/imgui-flame-graph
[32]https://github.com/epezent/implot
[33]https://github.com/aiekick/ImGuiFileDialog



Obrázek 4.1: **Základní rozdělení programu.**

4.6 Načítání modelů a jejich transformace

Pro vykreslování modelů je nutné je transformovat do octree. Jelikož neexistuje žádný standardní formát pro voxely, je potřeba implementovat import pro různé formáty. Aplikace je prozatím schopna importovat formát `vox` (sekce 2.2.2)).

Načítání dat je rozděleno na dvě části. V první části jsou ze vstupního souboru načteny využité materiály a je vytvořen seznam voxelů. Tohle je jediná část, kterou je nutné implementovat speciálně pro nové formáty. Zbytek převodu je popsán v algoritmu 11.

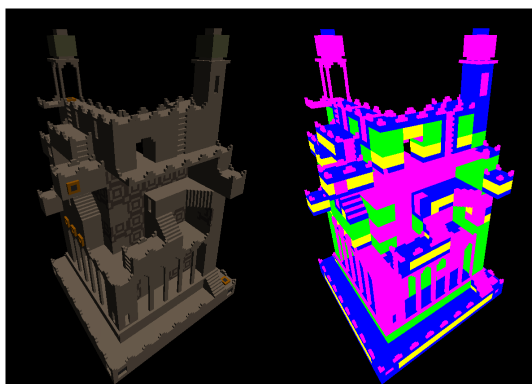
Algoritmus 11: Převod voxelů do octree

```
1  voxels.sort();
2  tree_depth = calculate_tree_depth(voxels);
3  root = tree.init_root();
4  foreach voxel in voxels do
5      node = root;
6      for level = 0; level < tree_depth; level++ do
7          index_for_level = getIndexForLevel(voxel.position, level, tree_depth);
8          child_node = node.getOrCreateNod(index_for_level);
9          // nastaveni hodnot voxelu (material...)
10         node = child_node;
11     node.is_terminal = true;
```

Po vytvoření stromové reprezentace je dalším krokem minimalizace stromu. Pokud jsou některé oblasti zaplněné, je možné snížit jeho složitost – označíme uzel na vyšší úrovni jako terminální a jeho potomky odstraníme.

Algoritmus 12: Minimalizace octree

```
1 Function IsNodeFilled(node):
2     return node.child_count == 8 and all_of(node.children, IsFilled);
3 Function MinimizeTree(node):
4     if IsNodeFilled(node) then
5         node.children.clear();
6         node.is_terminal = true;
7     else
8         foreach child in node.children do
9             MinimizeTree(child);
```



Obrázek 4.2: **Vizualizace hloubky voxelů ve stromu.** Levá část obsahuje render, vpravo jsou barevně odlišené úrovně octree.

Finálním krokem je transformace vytvořeného stromu do jeho binární reprezentace popsané v sekci 3.1.

4.7 Výpočet průsečíku se scénou

Paprsky vycházející z kamery jsou generovány podle rovnice 4.1. $\vec{c}$ značí normalizovanou pozici současného pixelu, $\vec{near}$ určuje pozici blízké roviny (near plane), $\vec{far}$ roviny vzdálené (far plane) a p je výsledný paprsek pro pozici na obrazovce.

$$
\begin{aligned}
\vec{c} &= calculateNormalizedScreenCoords() \\
\vec{near} &= inverseProjectionView \cdot < c_{xy}, -1, 1 > \\
\vec{far} &= inverseProjectionView \cdot < c_{xy}, 1, 1 > \\
\vec{origin} &= \frac{near_{xyz}}{near_w} \\
\vec{direction} &= normalize(\frac{far_{xyz}}{far_w} - origin) \\
p &= (\vec{origin}, \vec{direction})
\end{aligned}
\tag{4.1}
$$

Algoritmus 13 popisuje průchod BVH stromem za využití zásobníku. Důležitou součástí tohoto algoritmu je řazení potomků uzlu podle jejich vzdálenosti od počátku paprsku. Díky tomu jsou uzly

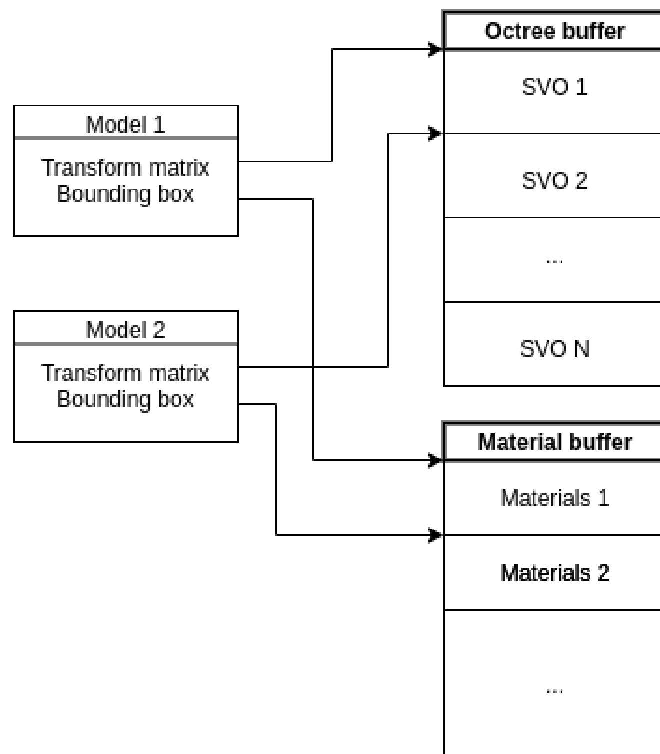
stromu procházeny v pořadí od nejbližšího, lze tedy od určité chvíle naprosto ignorovat modely, jejichž obalové těleso (AABB) se nachází dále než prozatím nejbližší průsečík scény. Je zde také menší pravděpodobnost, že bude docházet k větvení kvůli pořadí průchodu.

Algoritmus 13: Průchod paprsku hierarchií obalových těles

```
1  Function TraverseBVH(ray):
2      node = root;
3      stack = EmptyStack();
4      calculate intersection with root;
5      while intersected do
6          while intersected and not intersected is leaf do
7              intersection_a = IntersectNode(node.left_child);
8              intersection_b = IntersectNode(node.right_child);
9              swap intersections so that the closer one is in instersection_a;
10             if intersection_b hit and intersection_b is closer than nearest geometry hit then
11                 push(stack, node.right_child);
12             if neither node hit and not stack.Empty() then
13                 intersection_a = pop(stack);
14         if intersected is leaf then
15             if intersection is closer than nearest geometry hit then
16                 calculate model intersection and save distance if hit;
17         if empty(stack) then
18             intersected = false;
19         else
20             node = pop(stack);
```

4.8 Správa modelů

Pro modely ve scéně jsou použity tři samostatné struktury. Pro uložení základních informací, jako je transformační matice či obalové těleso, má každý model samostatnou strukturu. Data modelu – octree a materiály – jsou uloženy ve dvou bufferech, přičemž jsou tyto buffery sdíleny napříč všemi modely. Informační struktura obsahuje offsety pro umístění dat modelu ve zmíněných bufferech. Rozmístění těchto dat je na obrázku 4.3.



Obrázek 4.3: **Informace o modelu a vztahy k datovým bufferům.**

Pro usnadnění práce s alokací GPU paměti byla vytvořena třída `BufferMemoryPool`, která spravuje předaný buffer. Třída umožňuje zvolit zarovnání bloků při propůjčení paměti a zaručuje, že nedojde k přepisu již zapůjčených dat. Zapůjčené paměťové bloky využívají RAII[34] ke znemožnění úniku paměti. Ukázka použití této třídy je ve výpisu 4.3.

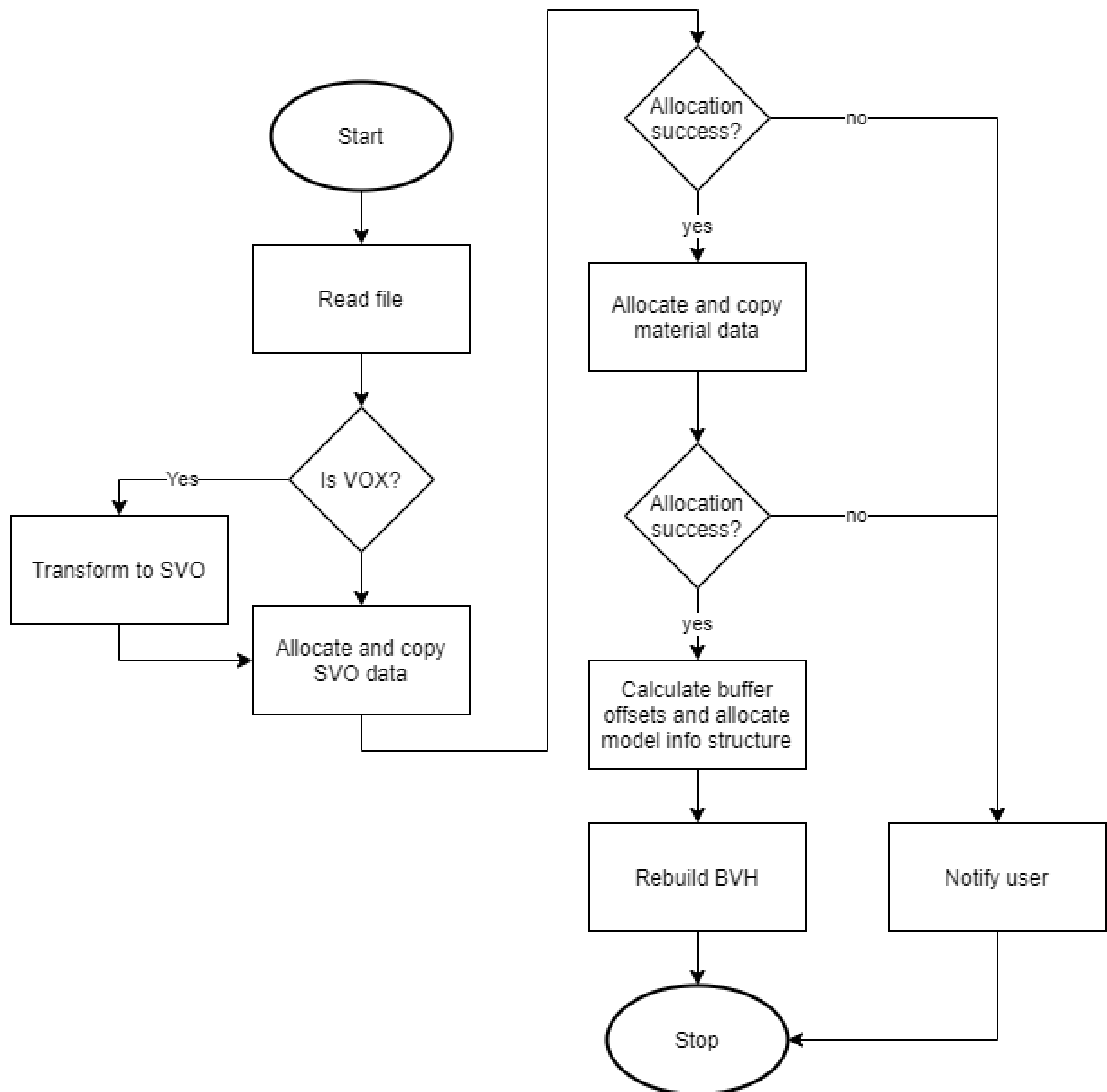
```
auto buffer = createBuffer(100_MB);
auto memoryPool = BufferMemoryPool{buffer, ALIGNMENT};

auto memoryBlock = memoryPool.leaseMemory(2_MB);
modelInfo.setMemoryOffset(memoryBlock.getOffset());
memoryBlock.mapping().set(data);
// ...
memoryPool.returnMemory(memoryBlock);
```

Výpis 4.3: Ukázka využití memory pool

Celý proces načtení nového modelu z disku je popsán v obrázku 4.4. Program umožňuje načítat dva typy formátů – `VOX`, jehož data je nutné transformovat do octree a `PF_VOX`, jež má již vhodnou podobu. Po načtení dochází k alokaci grafické paměti a odeslání dat do ní. Finálním krokem je aktualizace hierarchie obalových těles.

[34] https://en.cppreference.com/w/cpp/language/raii



Obrázek 4.4: **Diagram načítání dat modelů.**

4.8.1 Načítání a ukládání scén

Pro účel zjednodušení práce s modely aplikace podporuje jednoduchý konfigurační soubor pro uložení rozložení scény. Formátem tohoto souboru je `TOML`, podobně jako u konfiguračního souboru aplikace. Soubor definice scény je rozdělen na dvě části.

První částí je definice pozice a rozložení pole sond (light field probes). Položka `probeGridPos` určuje pozici počátku mřížky sond a `probeGridStep` definuje vzdálenost jednotlivých sond. `proximityGridSize` určuje rozdělení prostoru pokrytým sondami na voxely, které obsahují informace o sondách pro tento podprostor nejvhodnějších.

Další částí je pole modelů ve scéně obsažených. Záznamy modelů obsahují cestu k souboru pro načtení dat modelu a také vektory k jeho transformaci.

Ukázka jednoduchého souboru scény je ve výpisu 4.4.

```
probeGridPos = [ -1.0, 0.0, -1.0 ]
probeGridStep = 1.5
proximityGridSize = [ 64, 64, 64 ]

[[models]]
path = '/chair.vox'
rotateVec = [ 0.0, 0.0, 0.0 ]
scaleVec = [ 4.0, 4.0, 4.0 ]
translateVec = [ 0.0, 0.0, 0.0 ]
```

Výpis 4.4: Definice jednoduché scény

V implementaci je i téměř dokončená logika pro načítání scén připravených pro hru Teardown. Modely ve formátu, které hra využívá, jsou na nejnižší úrovni ve `VOX` souborech. Definice scény je v samostatném `XML` souboru, který určuje transformace modelů a umožňuje definovat objekty "otexturované"jiným modelem.

4.9 Demonstrační aplikace

Demonstrační aplikace slouží k vizualizaci výsledků a práci s modely/scénami. Aplikace může pracovat ve dvou režimech:

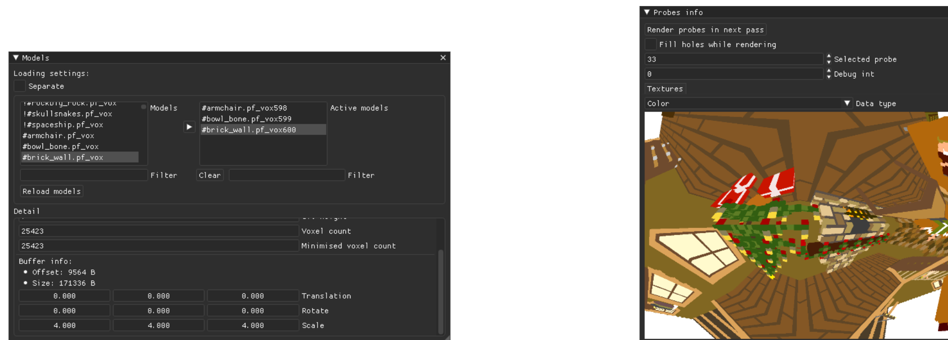
- Editační režim – v tomto režimu je scéna vykreslována pouze pomocí sparse voxel octree ray marchingu a slouží primárně k vytvoření scény a jejímu uložení na disk. Také je zde možné sledovat stav sond.
- Renderovací režim – slouží k načítání scén vytvořených v předchozím režimu a demonstraci výsledného algoritmu.

Uživatelské rozhraní je implementováno pomocí knihovny `pf_imgui` představené v sekci 4.4. Většina prvků uživatelského rozhraní obsahuje tooltipy pro snadné zjištění funkce jednotlivých komponent a UI okna lze přetáhnout ven z hlavního okna pro lepší přehlednost. Bližší popis se nachází na přiloženém paměťovém médiu, základní přehled zde ale bude uveden.

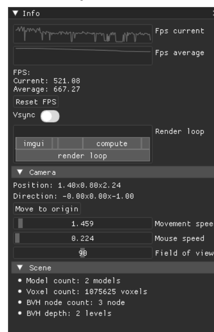
UI aplikace je rozděleno do následujících částí:

- Info – zobrazení statistik vykreslování (počet snímků za sekundu, flame graph), informace o kameře a její parametry, a také informace o právě vykreslované scéně.
- Render settings – výběr typu zobrazení (stínování, normály, počet iterací, hloubka ve stromě...), ovládání světla.
- Debug – memo pro výstup logů, připravený Chaiscript pro přidávání složitějších funkcí.
- Debug images – okno pro zobrazení textury reprezentující počet iterací při průchodu octree.
- Shader controls – ovládání parametrů pro shadery.
- Probe grid controls – ovládání mřížky sond.

- Models – ovládání modelů zobrazených ve scéně.
- Probes info – vizualizace atlasu sond a renderování scény pomocí něj.



(a) **Okno pro přidávání a manipulaci s modely** (b) **Okno pro zobrazení atlasu textur sond**



(c) **Informační okno**

Obrázek 4.5: **Ukázka oken demonstrační aplikace.**

Konfigurace

Důležitou součástí aplikace je konfigurační soubor, který je blíže popsán v příloze B. Tento soubor obsahuje primárně informace o cestách ke zdrojům (cesta k modelům, shaderům...). Také je v něm obsažena definice velikosti okna demonstrační aplikace a velikost pracovní skupiny pro vykreslování za pomoci octree. Aplikace automaticky ukládá data, která uživatel nastavil v uživatelském rozhraní.

Typ souboru zvolený pro konfiguraci je `TOML`. Tento formát byl zvolen primárně z toho důvodu, že je velice snadno čitelný pro uživatele a není tedy nutné vytvářet separátní program pro manipulaci s konfigurací. Samozřejmě existují podobné, uživatelsky přívětivé formáty, jako například `YAML` nebo `INI`, ale `TOML` byl zvolen také kvůli knihovně `toml++`, díky které je manipulace s konfiguračním souborem relativně pohodlná.

Kapitola 5

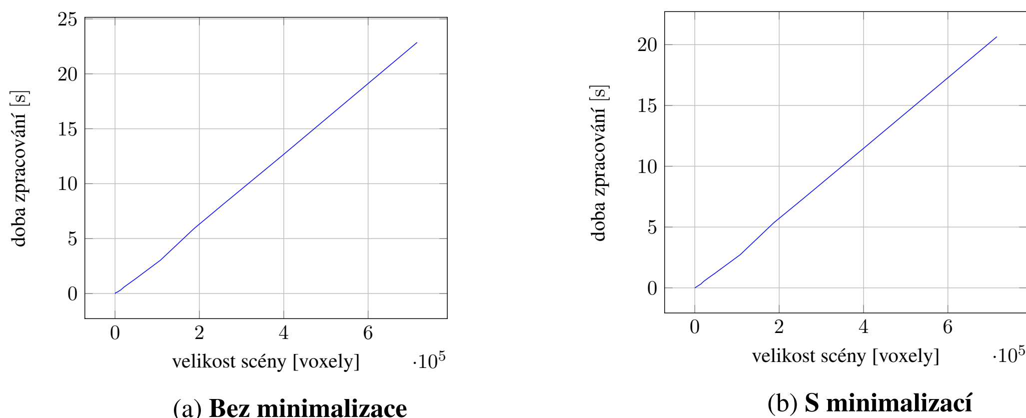
Vyhodnocení

V této kapitole je krátký popis dosažených výsledků týkající se doby převodu vstupních dat a také analýza využití paměti na grafické kartě. Poslední sekce se zabývá grafickými výstupy implementace. Pro měření byla použita následující PC konfigurace:

- CPU: AMD Ryzen 5 3600, 6-core, 4.2 GHz.
- GPU: ASUS GeForce ROG STRIX RTX 2070S.

5.1 Převod vstupních dat na octree

Převod vstupních dat na octree byl popsán v sekci 4.6. Převod je prozatím prováděn sekvenčně na jednom vlákně, ale v budoucí verzi jej autor plánuje předělat tak, aby využíval více vláken procesoru. Na obrázku 5.1a je graf závislosti času převodu na počtu vstupních voxelů. Graf 5.1b zobrazuje stejné měření s aktivovanou minimalizací. Měření byla prováděna s nejvyšší úrovní optimalizací kompilace. Pro měření bylo použito CPU `Ryzen 3600` a probíhalo na jednom vláknu za využití knihovny `nanobench`[1]. Model, který byl v testech použit, byly plné koule o různých poloměrech.



(a) **Bez minimalizace**

(b) **S minimalizací**

Obrázek 5.1: **Doba tvorby octree.**

[1] https://github.com/martinus/nanobench

I když to není z grafu příliš znatelné, transformace s minimalizací je o několik procent rychlejší, jelikož není nutné vytvářet tak velký finální strom. Pokud by vstupní data byla pro minimalizaci nepříznivá, byla by tato metoda pomalejší.

5.2 Paměťová náročnost

V této sekci je popsána paměťová náročnost dvou hlavních částí částí programu. Jedná se o vyhodnocení využití VRAM.

Octree

Každý uzel octree zabírá v prvé řadě 4 bajty (15 bitů `child pointer` + 1 bit `far` + 8 bitů `valid mask` + 8bitů `leaf mask`). Dále pro dohledání materiálů a ostatních parametrů další 4 bajty (24 bitů `value pointer` + 8 bitů `mask`). Tyto dva záznamy nejsou vytvářeny pro listové uzly. Pokud budeme předpokládat strom, který má prostor obsazený přesně z 50 % a je obsazen pouze každý druhý voxel, s hloubkou stromu 6 – tedy 37448 voxelů – celková obsazená paměť zmíněnými záznamy bude 18724 bajtů. Při popsaném rozložení voxelů se samozřejmě jedná o nejhorší možný případ.

hloubka stromu	obsazená paměť [B]
1	4
2	36
3	292
4	2340
5	18724
6	149796

Tabulka 5.1: **Využití paměti podle hloubky stromu při maximální nepříznivosti podmínek.**

Tímto jsou pokryta základní data popisující strom a vyhledávací strukturu do parametrů. Další paměť je využita `far` pointery. V pesimistickém případě je může potřebovat ~5 % uzlů, ovšem jsou potřeba až pro hloubku stromu vyšší než 7, a to kvůli množství potomků v úrovni.

Poslední data vyžadující velké množství paměti jsou samotné parametry voxelů. Samozřejmě záleží na využitých parametrech, pro phongovo stínování je dostačující barva, tedy 4 bajty (RGBA formát, kde je každý kanál reprezentován 8 bity). Samozřejmě se data dají zakódovat efektivněji.

Celková paměťová náročnost pro octree se zmíněnými parametry je tedy:

hloubka stromu	využití paměti [B]
1	20
2	164
3	1316
4	10532
5	84260
6	674084

Tabulka 5.2: **Využití paměti podle hloubky stromu při využití barevných parametrů pro voxely.**

Ve všech případech se jedná o nejhorší možný případ. Realisticky použitelné modely mají paměťovou náročnost výrazně nižší a také u nich dochází k minimalizaci stromu (která v uvažovaném nejhorším případě není možná).

Light field probes

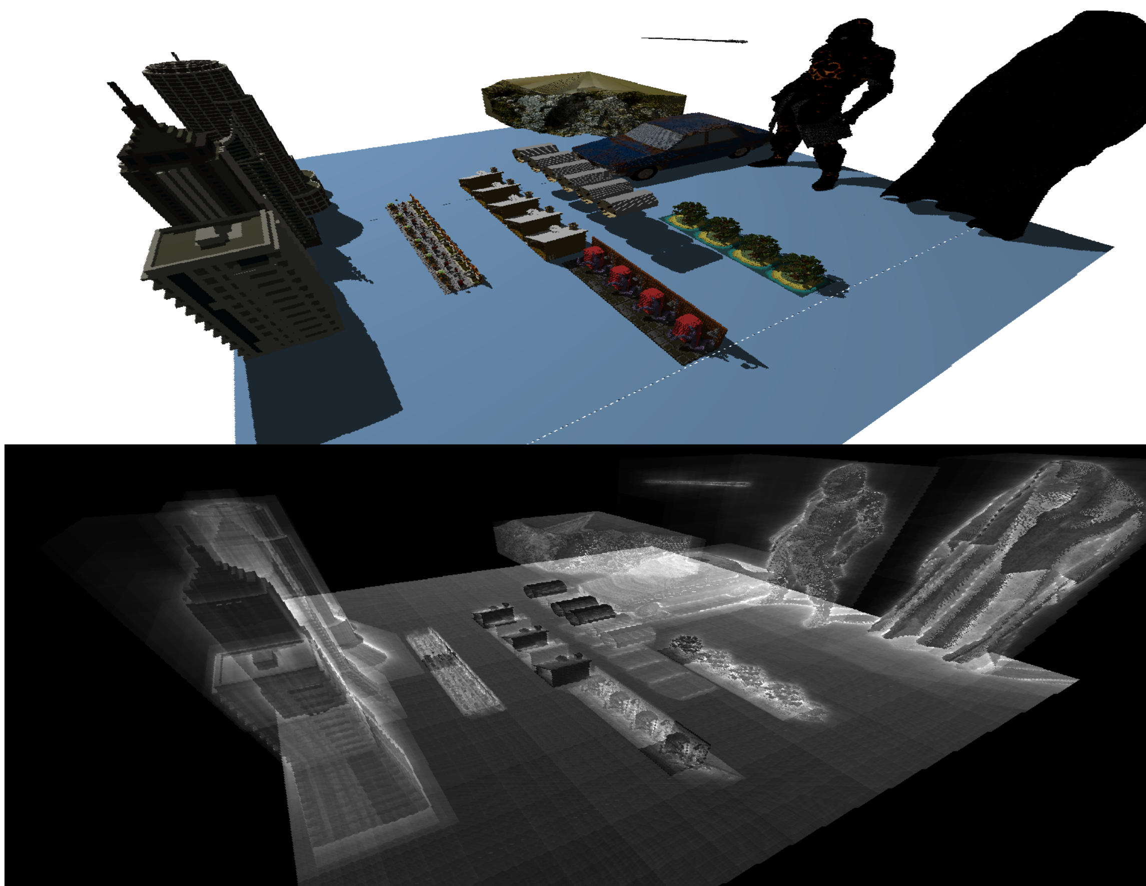
Jak bylo zmíněno v sekci 3.2.2, každá sonda sestává ze dvou textur. První z nich má rozlišení 1024x1024 s datovým typem `RG32F`, tedy využívá 8 bajtů paměti. Z těchto 8 bajtů jsou 4 využity pro radianci/nepřímé osvětlení, 2 pro normály a 2 pro hloubku. Druhá textura má $\frac{1}{16}$ rozlišení hlavní textury a obsahuje pouze informace o hloubce. Datovým typem této textury je `R16F`, což znamená, že každý texel zabírá 2 bajty. Využití paměti jedné sondy je tedy 8396800 bajtů (~8.4 MB).

Samozřejmě není použita jen jedna sonda. Množství sond je kvůli implementaci algoritmů s nimi pracujících omezeno na mocniny dvou v každé ose. V následující tabulce je seznam využití paměti pro množství sond, které se nejpravděpodobněji využijí.

počet sond	využití paměti [MB]
1	8.4
4	33.5
16	134.3
128	1074.8
512	4299.1

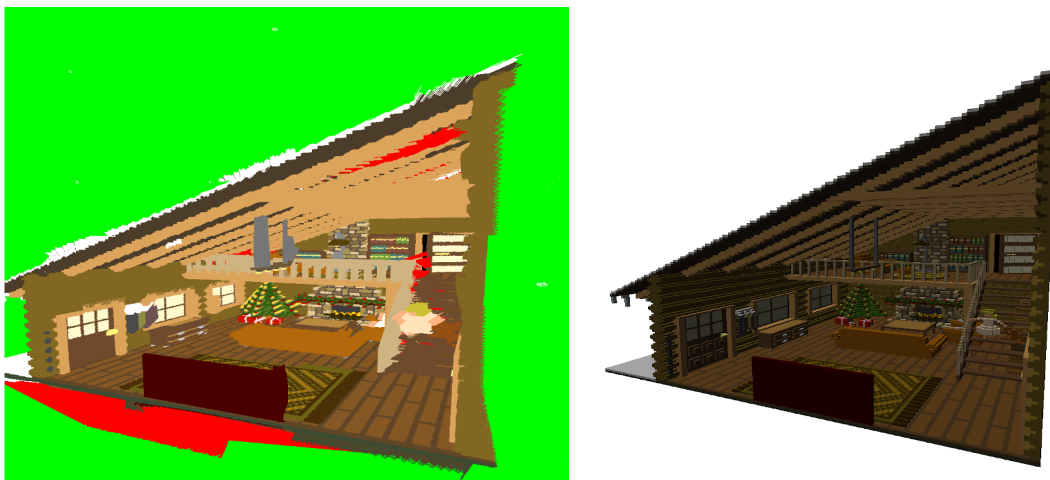
Tabulka 5.3: **Využití paměti pro časté velikosti mřížky.**

5.3 Výsledky



Obrázek 5.2: **Scéna vykreslena pomocí sparse voxel octree ray tracing.** V horní části obrázku se nachází vykreslená scéna se stíny, v dolní části je počet iterací nutný k výpočtu hodnot jednotlivých pixelů – bílá barva v tomto případě znamená 200 iterací.

Na obrázku 5.2 je vykreslena scéna pomocí sparse voxel octree ray tracing. Scéna obsahuje 36 modelů složených z 12-ti milionů voxelů. Hierarchie obalových těles této scény obsahuje 71 uzlů, má tedy hloubku 7 úrovní. Nejvyšší počet iterací nutných k výpočtu průsečíku zde bylo ~200 iterací (kroků průchodu stromu). Ve scéně je prováděn výpočet ostrých stínů. Na hardware, který byl zmíněn na začátku kapitoly, a při rozlišení 1920x1080 pixelů byla tato scéna vykreslena rychlostí 100 snímků za sekundu. Při vypnutí paprsků pro stíny došlo ke zrychlení o 20%, tedy 120 snímků za sekundu. Samozřejmě na snímku dochází k nejhoršímu možnému případu – všechny modely jsou viditelné a plocha, která je naspod, je velice neefektivní, jelikož se jedná o velký model, který má výšku pouze jednoho voxelu. Dělba prostoru pomocí octree je v tomto případě neefektivní. Po jeho odstranění dojde ke zrychlení na 300 snímků za sekundu.

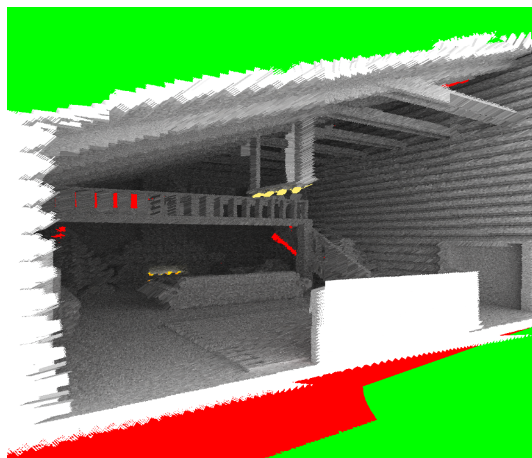


Obrázek 5.3: **Srovnání scény vykreslené pomocí light field probes a sparse voxel octree.** V levé části obrázku je scéna vykreslena pomocí light field probe ray tracing. Zelené oblasti zobrazují minutí scény, červené nemožnost zjistit výsledek pro paprsek. Pravá část je stejná scéna vykreslena pomocí sparse voxel octree.

Obrázek 5.3 obsahuje porovnání vykreslení scény z pohledu kamery pomocí light field probe ray tracing a sparse voxel octree ray tracing. Z obrázku je jasně vidět, že metoda není vhodná jako primární vykreslovací algoritmus. Výsledek obsahuje spoustu pixelů, kde nebyl nalezen výsledek a dochází k mnoha nepřesnostem, jak je lépe vidět na obrázku 5.4. Tyto problémy mohou vznikat buď nedostatečným množstvím informací v sondách – žádná ze sond není schopna poskytnout optimální výsledek – nebo nevhodným výběrem sondy.

Obrázek 5.4: **Přiblížení na artefakty light field probes ray tracing.**

Jak bylo popsáno v sekci 3.2.4, namísto radiance zobrazené výše je v sondách vykreslen příspěvek nepřímého osvětlení. Obrázek 5.5 zobrazuje nepřímé osvětlení ve scéně.



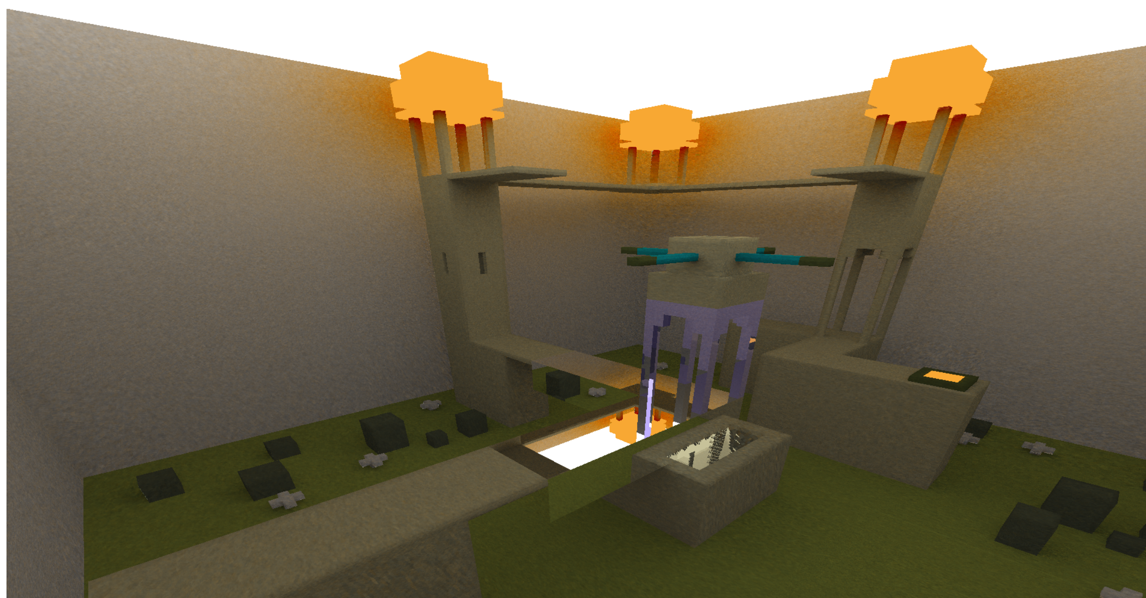
Obrázek 5.5: **Scéna vykreslena pomocí light field probes s nepřímým osvětlením.**

Při kombinaci zmíněných algoritmů je vykreslena scéna pomocí sparse voxel octree a ze sond je získáno nepřímé osvětlení. Výsledek je na obrázku 5.6. Z obrázku je vidět, že ne u všech fragmentů došlo k úspěšnému dohledání hodnot pro nepřímé osvětlení. Důvodem jsou dříve zmíněná "neviditelná"místa, kdy žádná z kandidátních sond "nevidí"na pozici.

Obrázek 5.6: **Scéna vykreslena kombinací light field probes a sparse voxel octree.**

Tento přístup je tedy vhodnější pro scény, kde nedochází k častému překryvu, jako ve scéně výše. Na obrázku 5.7 je bližší snímek vánočního stromu z obrázku 5.6. Žádná ze sond na toto místo nevidí a vznikají tím nehezké artefakty. Je možné pokusit se zakrýt tyto problémy čtením dat ze sondy pouze ve směru místa průsečíku namísto přesné vzdálenosti, to ale přináší podobné problémy.

Obrázek 5.7: **Artefakty v prostoru kde sonda "nevidí".**



Obrázek 5.8: **Ukázka scény s emisivním a reflektivním materiálem.** Oranžové objekty jsou zdroji světla. Ve středu spodní části obrázku se nachází reflektivní materiál. Ve scéně se nenachází zdaleka tolik artefaktů jako v té předchozí díky její jednoduchosti.

Kapitola 6

Závěr

Cílem práce bylo navrhnout systém pro zobrazování voxelových modelů v reálném čase za využití GPU akcelerace a implementovat jej. K dosažení tohoto cíle bylo nejdříve nutné nastudovat již existující algoritmy řešící podobné problémy. Algoritmů pro realistické vykreslování existuje poměrně velké množství, některé z nich byly představeny v první kapitole. Bylo také nutné se seznámit s možnostmi reprezentace voxelových dat a možné interakce s nimi – například výpočet průsečíku. V neposlední řadě bylo potřeba představit užitečné datové struktury pro dělbu prostoru, díky kterým je možné optimalizovat různé algoritmy.

Následný návrh se zabýval specifickou reprezentací voxelových dat k možnosti efektivního nalezení průsečíku s paprskem. Metoda reprezentace těchto dat byla založena na článku [21]. Jako způsob uložení voxelových dat byla navržena úsporná struktura sparse voxel octree. Dále zde byly představeny některé metody pro vykreslování scén. První z nich je založena na sledování paprsku ve výše zmíněné datové struktuře, přičemž bylo navrženo rozšíření využívající hierarchie obalových těles ke snazší práci s modely a urychlení jejich vykreslování. Dalším důležitým zdrojem byl článek [26], který představil koncept light field probes. V návrhu byl popsán způsob reprezentace dat sond za pomocí atlasu textur obsahující informace pro možnost sledování paprsku skrze tyto struktury. Byla vysvětlena též zmíněná metoda pro vykreslování a vlastní optimalizace za použití voxelového pole, která umožňuje algoritmus zrychlit a zlepšuje jeho úspěšnost pro nalezení průsečíku se scénou. Pro přípravu nepřímého osvětlení scény byla navržena metoda založená na light field probes, která využívá atlasu textur k uložení dat o nepřímém osvětlení – tato data lze využít v dalším algoritmu. Byla též popsána struktura výsledného vykreslovacího řetězce, která byla rozdělena na samotné vykreslení scény a aplikace nepřímého osvětlení ze sond.

Pro implementaci aplikace bylo využito API Vulkan společně s mnoha dalšími knihovnami zmíněnými v kapitole 4. V průběhu implementace návrhu vznikly tři knihovny, které byly krátce popsány. Specificky se týkaly obecné funkčnosti, práce s okny a grafickou kartou a uživatelského rozhraní. Kapitola specifikovala některé zajímavé části implementace, jako je budování octree pro modely a správa modelů uvnitř programu.

Výsledná demonstrační aplikace umožňuje uživateli vytvořit si vlastní scénu či nahrát již scény připravené. Aplikace též zobrazuje informace o zobrazovaných modelech a umožňuje uživateli s nimi manipulovat. Také zobrazuje metriky scény a rychlost vykreslování. Demonstruje funkčnost sparse voxel octree ray tracing, pro který zobrazuje i informace pro evaluaci scény, a také light field probes, kdy je možné zkoumat obsah jednotlivých sond. V alternativním režimu vykresluje scény kombinací těchto dvou algoritmů.

Z pohledu efektivity algoritmů je načítání scén z formátů třetích stran poměrně pomalé, uživatel může čekat i několik vteřin u větších souborů, díky implementaci vlastního formátu je ale možné tomuto zpomalení předejít. Náročnost algoritmů na paměť je pro reprezentaci voxelových modelů

je přijatelná, lze vykreslovat scény o velikosti desítek milionů voxelů s využitím paměti, se kterou by neměly problémy ani starší grafické karty – například scéna obsahující ~12 milionů voxelů využívá ~92 MB paměti. Paměťová náročnost sond je výrazně vyšší, realistické využití s dalšími strukturami je pro maximální velikost mřížky sond přibližně 8x8x8. Paměťové omezení dané sondami implikuje, že navržený systém je vhodnější spíše pro scény v menší škále, ovšem s vysokou úrovní detailu.

Výsledné grafické výstupy tedy mohou obsahovat rozsáhlé scény, dochází ovšem ke vzniku artefaktů. Tyto artefakty jsou ovšem rušivé a s využitím použitých metod jim nejspíše nelze předejít. Alternativou k použitému přístupu by mohla být metoda uvedena v článku *Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields*, která navíc umožňuje využívat dynamických scén, na rozdíl od metody použité zde, která zvládá pouze scény statické. Mezi další rozšíření by mohla patřit implementace anti-aliasingu, specificky nějaká temporální metoda by byla zajímavá. Dalším vhodným rozšířením by byla implementace datové struktury, která by umožňovala optimalizované streamování dat scény z disku, aby bylo možné zobrazovat rozsáhlé scény bez nutnosti čekat na načtení modelů.

Literatura

[1] *Medical Image Computing and Computer Assisted Intervention - MICCAI 2019: 22nd international conference, Shenzhen, China, October 13-17, 2019: proceedings–Part II*. Springer, 2019.

[2] *Vulkan® 1.2.166 - A Specification* [online]. Leden 2021 [cit. 2021-07-21]. Dostupné z: https://www.khronos.org/registry/vulkan/specs/1.2/html/vkspec.html.

[3] CIGOLLE, Z. H., DONOW, S., EVANGELAKOS, D., MARA, M., MCGUIRE, M. et al. A Survey of Efficient Representations for Independent Unit Vectors. In:. 2014.

[4] 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.

[5] DESPOTOVIĆ, I., GOOSSENS, B. a PHILIPS, W. MRI Segmentation of the Human Brain: Challenges, Methods, and Applications. *Computational and Mathematical Methods in Medicine*. Květen 2015, sv. 2015, s. 1–23. DOI: 10.1155/2015/450341.

[6] EPHTRACY. *MagicaVoxel .vox File Format* [online]. Říjen 2016 [cit. 2021-07-21]. Dostupné z: https://github.com/ephtracy/voxel-model/blob/master/MagicaVoxel-file-format-vox.txt.

[7] ERICSON, C. Collision Detection Design Issues. *Real-Time Collision Detection*. 2005, s. 75–125. DOI: 10.1016/b978-1-55860-732-3.50007-3.

[8] FUJIMOTO, A. a IWATA, K. Accelerated Ray Tracing. In: KUNII, T. L., ed. *Computer Graphics*. Tokyo: Springer Japan, 1985, s. 41–65. ISBN 978-4-431-68030-7.

[9] GIATILI, S. a STAMATAKOS, G. A detailed numerical treatment of the boundary conditions imposed by the skull on a diffusion–reaction model of glioma tumor growth. Clinical validation aspects. *Applied Mathematics and Computation*. Květen 2012, sv. 218, s. 8779–8799. DOI: 10.1016/j.amc.2012.02.036.

[10] GREENBERG, D., COHEN, M. a TORRANCE, K. Radiosity: A method for computing global illumination. *The Visual Computer*. Září 1986, sv. 2, s. 291–297. DOI: 10.1007/BF02020429.

[11] HAN, T. D. a ABDELRAHMAN, T. Reducing branch divergence in GPU programs. In:. Leden 2011, s. 3. DOI: 10.1145/1964179.1964184.

[12] HART, J. Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces. *The Visual Computer*. Červen 1995, sv. 12. DOI: 10.1007/s003710050084.

[13] HUDSON, A. *SVX Format* [online]. Říjen 2014 [cit. 2021-07-21]. Dostupné z: https://abfab3d.com/svx-format/.

[14] HUGHES, J. F., DAM, A. V., MCGUIRE, M., SKLAR, D. F., FOLEY, J. D. et al. *Computer graphics principles and practice*. Pearson, 2019.

[15] HUNTER, R. S. a HAROLD, R. W. *The measurement of appearance*. J. Wiley, 1987.

[16] IMMEL, D., COHEN, M. a GREENBERG, D. A Radiosity Method for Non-Diffuse Environments. In:. Srpen 1986, sv. 20, s. 133–142. DOI: 10.1145/15922.15901.

[17] 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.

[18] KAY, T. L. a KAJIYA, J. T. Ray Tracing Complex Scenes. In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: Association for Computing Machinery, 1986, s. 269–278. SIGGRAPH '86. DOI: 10.1145/15922.15916. ISBN 0897911962. Dostupné z: https://doi.org/10.1145/15922.15916.

[19] KEINERT, B., SCHAEFER, H., KORNDÖRFER, J., GANSE, U. a STAMMINGER, M. Enhanced Sphere Tracing. In: *STAG*. 2014.

[20] LABORATORY, R. P. I. I. P. a MEAGHER, D. *Octree Encoding: a New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer*. 1980. Dostupné z: https://books.google.cz/books?id=CgRPOAAACAAJ.

[21] LAINE, S. a KARRAS, T. Efficient Sparse Voxel Octrees – Analysis, Extensions, and Implementation. In:. 2011.

[22] LENSCH, H. P. A., GOESELE, M., CHUANG, Y.-Y., HAWKINS, T., MARSCHNER, S. et al. Realistic Materials in Computer Graphics. In: *ACM SIGGRAPH 2005 Courses*. New York, NY, USA: Association for Computing Machinery, 2005, s. 1–es. SIGGRAPH '05. DOI: 10.1145/1198555.1198601. ISBN 9781450378338. Dostupné z: https://doi.org/10.1145/1198555.1198601.

[23] LORENSEN, W. E. a CLINE, H. E. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *SIGGRAPH Comput. Graph.* New York, NY, USA: Association for Computing Machinery. Srpen 1987, sv. 21, č. 4, s. 163–169. DOI: 10.1145/37402.37422. ISSN 0097-8930. Dostupné z: https://doi.org/10.1145/37402.37422.

[24] MARSCHNER, S. *CS 4620 Lecture 3, Ray Tracing: intersection and shading*. FCornell University, září 2013.

[25] MAULE, M., COMBA, J. L. D., TORCHELSEN, R. a BASTOS, R. Transparency and Anti-Aliasing Techniques for Real-Time Rendering. In: *2012 25th SIBGRAPI Conference on Graphics, Patterns and Images Tutorials*. 2012, s. 50–59. DOI: 10.1109/SIBGRAPI-T.2012.9.

[26] McGuire, M., Mara, M., Nowrouzezahrai, D. a Luebke, D. Real-Time Global Illumination Using Precomputed Light Field Probes. In: *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. New York, NY, USA: Association for Computing Machinery, 2017. I3D '17. DOI: 10.1145/3023368.3023378. ISBN 9781450348867. Dostupné z: https://doi.org/10.1145/3023368.3023378.

[27] Nguyen, H. Chapter 1. Generating Complex Procedural Terrains Using the GPU. In: *GPU gems 3*. Addison-Wesley, 2008.

[28] Nousiainen, O. *Performance comparison on rendering methods for voxel data*. Červenec 2019. Dostupné z: https://aaltodoc.aalto.fi/bitstream/handle/123456789/39899/master_Nousiainen_Oskari_2019.pdf?sequence=1&isAllowed=y.

[29] Olson, T., Koch, D., Shyshkovtsov, O. Zdravkovic, A., Fuller, J. et al. *Vulkan - Industry Forged*. Březen 2015. Dostupné z: https://www.khronos.org/vulkan/.

[30] Philip, D., Bala, K. a Bekaert, P. *Advanced global illumination*. CRC Press, 2020.

[31] T. Akenine Moller, N. H. A. P. M. I. S. H. *Real-Time Rendering*. 4th Edition. CRC Press, 2018. ISBN 9781138627000. Dostupné z: http://gen.lib.rus.ec/book/index.php?md5=00E321284115574DDD4DD98E3B43BD0E.

[32] Vaidyanathan, K., Woop, S. a Benthin, C. Wide BVH Traversal with a Short Stack. In: *High Performance Graphics*. 2019.

[33] Williams, D. *VolDat: A standard format for exchanging volume data* [online]. Březen 2013 [cit. 2021-07-21]. Dostupné z: http://www.volumesoffun.com/voldat-format/.

Příloha A

Obsah přiloženého paměťového média

- `README.md` - Obsahuje informace o aplikaci, jak ji přeložit a používat.
- `doc/` - Složka obsahující dokumentaci.
  - `thesis/` - Text a zdrojové soubory této práce.
  - `Thesis.pdf` - Vygenerovaná dokumentace .
- `src/` - Zdrojové soubory.
- `bin/` - Složka se spustitelným souborem pro systém Linux.
- `video/` - Složka s prezentačním videem.

Příloha B

Konfigurační soubor

```
[rendering.compute]
local_size_x = <int>
# doporuceno 8
local_size_y = <int>
# doporuceno 8

[resources]
path_models = <path string>
# cesta do slozky assets/vox v adresari projektu
path_shaders = <path string>
# cesta do slozky src/shaders v adresari projektu

[ui.imgui]
path_icons = <path string>
# cesta do slozky assets/icons v adresari projektu

# v teto sekci se nachazi take aplikaci
# vygenerovana konfigurace hodnot uzivatelskeho rozhrani

[ui.window]
height = <int>
width = <int>
```