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

ZOBRAZENÍ ROZSÁHLÝCH VOLUMETRICKÝCH DAT NA CPU

CPU RENDERING OF LARGE VOLUMETRIC DATA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MICHAL MAJER**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. MICHAL ŠPANĚL, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Majer Michal**
Program: Informační technologie
Název: **Zobrazení rozsáhlých volumetrických dat na CPU**
CPU Rendering of Large Volumetric Data
Kategorie: Počítačová grafika

Zadání:

1. Seznamte se s principy přímého zobrazení volumetrických dat (tzv. Volume Rendering).
2. Prostudujte současné přístupy a existující knihovny pro zobrazení velkých volumetrických dat a analyzujte jejich vhodnost pro implementaci v CPU.
3. Vyberte vhodné postupy a technologie a navrhněte CPU renderer a demo aplikaci pro zobrazení velkých volumetrických dat.
4. Experimentujte s Vaší implementací a případně navrhněte vlastní modifikace.
5. Porovnejte dosažené výsledky a diskutujte možnosti budoucího vývoje.
6. Vytvořte stručný plakát prezentující Vaši bakalářskou práci, její cíle a výsledky.

Literatura:

- Wald *et al.*, "OSPRay - A CPU Ray Tracing Framework for Scientific Visualization," in *IEEE Transactions on Visualization and Computer Graphics*, 2017 (https://dl.acm.org/doi/10.1109/TVCG.2016.2599041).
- Beyer *et al.*, "A Survey of GPU-Based Large-Scale Volume Visualization", in *EuroVis - STARs*, 2014 (https://diglib.eg.org/handle/10.2312/eurovisstar.20141175.105-123).

Pro udělení zápočtu za první semestr je požadováno:

- Splnění prvních tří bodů zadání.

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

Vedoucí práce: **Španěl Michal, Ing., Ph.D.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 2. listopadu 2021

Abstrakt

Tato práce zkoumá přímé zobrazování rozsáhlých volumetrických dat na CPU. Cílem bylo navrhnout paralelní implementaci algoritmu Ray casting v jazyce Rust a implementovat optimalizace Early Ray Termination a Empty Space Skipping pro zrychlení vykreslování. Dále jsem navrhl demo aplikaci k interaktivnímu prohlížení objemových dat, která tento algoritmus aplikuje. V rámci práce také vznikl generátor volumetrických dat.

Obě optimalizace ve výsledném řešení zrychlují výkon 12×. Paralelizace toto číslo dále zlepšuje a na testované soustavě renderuje rozsáhlý objem rychlostí 3,92 FPS.

Abstract

This work examines direct rendering of large volumetric data on the CPU. The aim was to design a parallel implementation of Ray casting algorithm in the Rust programming language and to implement Early Ray Termination and Empty Space Skipping optimizations to speed up rendering. I also designed a demo application to interactively display volumes using this algorithm. A volumetric data generator was also created as part of the work.

Both optimizations in the resulting solution offer a 12× speed up. Parallelization further improves this number and renders a large volume at 3.92 FPS on the tested system.

Klíčová slova

přímé vykreslování objemů, Ray casting, voxel, objemová data, CPU rendering, počítačová grafika

Keywords

direct volume rendering, Ray casting, voxel, volume data, CPU rendering, computer graphics

Citace

MAJER, Michal. *Zobrazení rozsáhlých volumetrických dat na CPU*. Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Michal Španěl, Ph.D.

Zobrazení rozsáhlých volumetrických dat na CPU

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Michala Španěla, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Michal Majer
8. května 2022

Poděkování

Chtěl bych poděkovat Ing. Michalovi Španělovi, Ph.D. za vedení mé bakalářské práce a za veškerou pomoc, kterou mi během vypracování poskytl.

Obsah

1 Úvod **3**

2 Volumetrická data a metody jejich zobrazení **4**
2.1 Volumetrická data 4
2.2 Přenosová funkce 6
2.3 Osvětlení objemu 7
2.4 Interpolace dat 8
2.5 Přímé zobrazování volumetrických dat 11
2.6 Ray casting 12
2.7 Optimalizace Ray castingu 14
2.8 Paralelizace Ray castingu 17
2.9 OSPRay 19

3 Návrh CPU rendereru volumetrických dat v jazyce Rust **20**
3.1 Renderování metodou Ray casting 21
3.2 Osvětlovací model 23
3.3 Zpracování volumetrických dat 23
3.4 Návrh demo aplikace 24
3.5 Návrh aplikace pro generování objemů 25

4 Implementace rendereru **27**
4.1 Použité nástroje a knihovny 27
4.2 Struktura implementace 28
4.3 Zpracování dat 29
4.4 Objemy 30
4.5 Rendering 31
4.6 Sdílení dat mezi pracovními vlákny 33
4.7 Optimalizace renderování 34
4.8 Demo aplikace 35
4.9 Generování volumetrických dat 36
4.10 Testy, benchmarky a dokumentace 37

5 Testování a vyhodnocení výsledků **39**
5.1 Testovací dataset 39
5.2 Efektivita optimalizací 39
5.3 Efektivita přístupu do paměti 40
5.4 Cena převodu vzorků 41
5.5 Kvalita vykreslování 42

5.6 Experimenty s parametry paralelního algoritmu 42

6 Závěr **45**

Literatura **46**

A Plakát **49**

B Přehled struktury datového média **50**

Kapitola 1

Úvod

Zařízení v oboru lékařství, např. CT a ultrazvuk, produkují volumetrická data, která je potřeba dále analyzovat. Užitečným nástrojem k analýze je grafické zobrazení těchto dat. Interaktivní zobrazení volumetrických dat je poptávané, zároveň ale výpočetně velmi náročné. Proto existuje úsilí tento proces maximálně zefektivnit.

Tématem této práce je zobrazení rozsáhlých objemových dat na CPU. Typicky jsou renderovací úlohy vykonávané na grafickém čipu (GPU), zobrazování dat na CPU ale má své výhody a uplatnění. Především se jedná o problém řádově menší paměti, ke které má GPU přístup. Je zcela běžné, že zobrazovaný volumetrický datový soubor svou velikostí přesahuje desítky GB a pro tyto případy je renderování na CPU jednodušší na realizaci. Dále se renderování na CPU využívá na stanicích, které nemají grafický čip, ať už stolní počítače, nebo velké clustery.

V této práci jsem zkoumal existující metody a hotová řešení zobrazování volumetrických dat a navrhl jsem aplikaci využívající vlastní implementaci renderovacího algoritmu, který vychází z existujících řešení.

Práce se věnuje metodám přímého vykreslování volumetrických dat, zejména metodě Ray casting a jejím optimalizacím. Navrhl a implementoval jsem vlastní renderovací algoritmus, aplikaci s grafickým rozhraním a aplikaci pro generování volumetrických dat. Renderer využívá paralelismu při zachování optimalizací Early Ray Termination a Empty Space Skipping. Toho dosahuje sdílením informací o průhlednostech mezi pracovními vlákny.

Kapitola 2 rozebírá principy zobrazení volumetrických dat. Kapitoly 3 a 4 popisují návrh a implementaci vlastního řešení rendereru a demo aplikace. V poslední kapitole se zaměřuji na vyhodnocení dosažených výsledků práce. Optimalizovaný renderer zobrazil objemová data o rozlišení $2k^3$ při 3,91 FPS.

Kapitola 2

Volumetrická data a metody jejich zobrazení

Vizualizace je proces získávání nových poznatků vizuálními metodami. Je to sada operací - analýza dat, filtrování a renderování. Cílem renderování je zpracovat analyzovaná data do 2D obrázků [21]. Zobrazování objemových dat je metoda extrahování užitečných informací za použití interaktivní grafiky [8].

2.1 Volumetrická data

Volumetrická data jsou definována jako množina diskrétních bodů s hodnotou [11]. Typicky jsou tyto diskrétní vzorky uspořádané do trojrozměrné (typicky pravidelné) mřížky (rasteru). Jeden takový bod se nazývá `voxel` [8]. Termín pochází ze spojení slov volume, picture a element, tedy prostorový prvek obrazu.

Pokud je objekt popsán volumetrickými daty, znamená to, že je popsán celý vnitřní objem. Tím se liší od povrchových reprezentací trojrozměrných objektů, které nesou informace pouze o povrchu objektu.

Vznik volumetrických dat

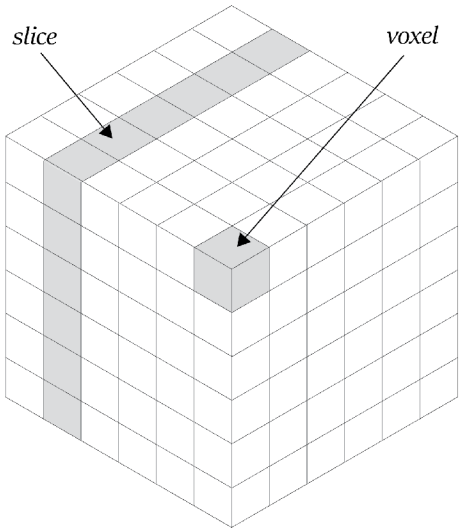
Volumetrická data jsou generována například ve zdravotnickém průmyslu [3]. Typicky vznikají složením dvourozměrných řezů produkovaných zařízeními jako CT, MRI nebo ultrazvuk. Vývojáři produktů a vědci zobrazují výsledky simulací například z oblastí aerodynamiky, medicíny a jiných. Dále se volumetrické renderování používá ve videoherním a filmovém průmyslu.

Rastr – mřížka volumetrických dat

Regulární mřížka je taková, kde se vzdálenosti mezi rovinami vzorků dají vyjádřit trojicí (d_x, d_y, d_z). Pokud jsou tyto vzdálenosti stejné, jedná se o kartézskou mřížku. Práce s pravidelnou pravoúhlou mřížkou postihuje velkou část využití v praxi. [8]

Na obrázku 2.1 je kromě voxelu vyobrazen řez (`slice`). Řez je množina všech vzorků sdílejících jednu souřadnici.

Pravidelnou mřížku vzorků lze vyjádřit 3D maticí. Nejjednodušší reprezentace 3D matice v lineární paměti počítače je souvislý blok paměti o velikosti počtu prvků matice a informace o dimenzích původní matice. Struktura hodnot matice m_{xyz} je zachována, po-



Obrázek 2.1: Vizualizace objemu. `Voxel` je jeden datový bod s hodnotou. Řez (`Slice`) je množina všech voxelů sdílejících jednu souřadnici. Voxel jako datový bod nemá rozměr, ale stejně jako pixely se běžně vizualizuje rozměrný.

kud jsou prvky v paměti uspořádány v pořadí podle os a k adresaci prvku matice je použit následující vztah:

$$i = z \ \cdot dim_x \cdot dim_y + y \cdot dim_x + x \tag{2.1}$$

kde (dim_x, dim_y, dim_z) jsou dimenze matice a i je pořadí prvku v lineárním bloku paměti. Dimenze dim_z je zde nejrychleji rostoucí souřadnice.

Formát volumetrických dat

V současnosti existují různé formáty reprezentace volumetrických dat, neexistuje však jeden dominantní standard.

K dekódování volumetrického souboru je nutné, aby soubor obsahoval hlavičku s meta-informacemi. Pokud je hlavička textová, je v některých případech možné soubor přečíst i bez externí specifikace formátu. Příkladem informací obsažených v hlavičce jsou datový typ vzorku, rozměry dat, vzdálenosti mezi vzorky a pořadí bajtů (MSB/LSB). Velikost vzorků je nejčastěji 1 bajt, což nahrává jednoduchosti zpracování a šetří paměť. [19]

Požadavky na formát se odvíjejí od oblasti vzniku a zpracování dat. V případech, kdy může být důležité surová data ručně prohlížet a upravovat, je výhodné použít textový formát.

Binární reprezentace dat má významnou výhodu. Jednotlivé řezy se dají uložit jako bezztrátově zkomprimované obrázky [19]. Komprimace obrazových dat je dobře prozkoumaná oblast a úspora dat je značná. Velkou část volumetrických souborů lze komprimovat algoritmy určenými pro šedotónový obraz, protože voxely mají jedinou hodnotu. Alternativně lze použít všeobecné kompresní techniky (zlib). Příkladem je stránka digimorph.org[1], která poskytuje volumetrická data jako soubor řezů ve formátu TIFF.

[1] http://www.digimorph.org/listallapplets.phtml?mt=6

2.2 Přenosová funkce

Volumetrická data jsou pouhým souborem vzorků, které nelze zobrazit. Přenosová funkce slouží k přiřazení optických vlastností těmto hodnotám. Získané informace, typicky barva a průhlednost, se používají ve fázi renderování.

Přenosová funkce plní roli klasifikace [17]. Určuje totiž vzorky, které jsou pro vizualizaci zajímavé.

V základní podobě vypadá převod přenosovou funkcí f následovně:

$$C_{rgba} = f(v) \tag{2.2}$$

v je hodnota vzorku z objemu a C_{rgba} jsou výsledné optické vlastnosti [18].

Jednorozměrná přenosová funkce pracuje s jediným parametrem – hodnotou datového bodu v objemu. Může být implementovaná vyhledávací tabulkou nebo libovolným funkčním předpisem. Možnost upravovat přenosovou funkci během vizualizace je žádoucí.

Přenosová funkce je navržena pro určitý vzorkovací krok. Pokud by se vzorkovací krok zmenšil bez úpravy přenosové funkce, vzrostl by počet vzorků a objem by se stal méně průhledným. Aby změna vzorkovacího kroku neovlivnila průhlednost objemu, musí se na výsledek přenosové funkce aplikovat následující korekce [7]:

$$\alpha = 1 - (1 - \alpha_0)^{s_0/s} \tag{2.3}$$

s_0 je referenční délka kroku, pro který je přenosová funkce navržena, s je nová délka kroku a α_0 je průhlednost získaná z přenosové funkce.

Vícerozměrné přenosové funkce přidávají další parametry a tím získávají mocnější vyjadřovací sílu [7].

Gradient

Typická dvourozměrná přenosová funkce je taková, kde druhým parametrem je `gradient`. Gradient je první derivací objemu a v bodě se dá interpretovat jako směr největší změny hodnoty funkce. Využívá se k odhadu normály povrchu, která je potřebná k výpočtu množství světla odraženého od objektu ve fázi stínování [8].

Mezi nejjednodušší heuristiky užívané pro odhad gradientu je rozdíl hodnot okolních vzorků podél jednotlivých os [7]:

$$Grad_x(P) = \frac{P_{x+n} - P_{x-n}}{2n} \tag{2.4}$$

$$Grad_y(P) = \frac{P_{y+n} - P_{y-n}}{2n} \tag{2.5}$$

$$Grad_z(P) = \frac{P_{z+n} - P_{z-n}}{2n}, \tag{2.6}$$

kde n je vzdálenost vzorku od výchozího bodu. Například bod P_{x+n} je od výchozího bodu P vzdálený n jednotek v kladném směru osy x.

Výsledný vektor $Grad(P)$ se dá interpretovat jako normála povrchu. Předpokladem tedy je, že významné změny hodnot značí přechod mezi dvěma různými objekty. Délka a směr gradientu se dá použít k detekci hranic objektů a k jejich stínování v osvětlovacím modelu.

Uvnitř homogenních materiálů jsou gradienty malé nebo nulové. Práce s těmito gradienty může při výpočtu světelných vlastností přinést do obrazu chyby [10]. Tyto chyby se dají redukovat aplikováním stínování jen na vzorky s gradienty od určité velikosti.

Gradient je možno vypočítat nad hodnotami vzorků i nad převedenými optickými vlastnostmi. Pokud je gradient počítán z optických vlastností, je nutné obarvit přenosovou funkcí více vzorků, a je to tedy dražší varianta.

Gradient je také možno v rámci předzpracování vypočítat pro každý bod mřížky a při běhu interpolovat[2] kromě vzorku i gradient [8]. Pro rozsáhlé objemy je uvedené řešení z důvodů potřebné paměti neschůdné. Pro objem s rozlišením $1k^3$ vzorků by totiž k uložení gradientů hodnot bylo zapotřebí 12 GB paměti.

V praxi je potřeba ošetřit vzorkování gradientu v bodech blízkých hranici objemu. Vzorkování v těchto bodech může vyžadovat přístup k bodům mimo objem. Jedním řešením je definovat `pozadí`, tedy hodnotu, kterou nabývají všechny body mimo hranice objemu. Druhým řešením je omezit rozsah vzorkování tak, aby mimo tyto hranice nezasahoval.

Získání gradientu v bodech mimo definovanou mřížku vyžaduje interpolaci (viz kapitola 2.4).

Návrh přenosové funkce

Přenosová funkce přiřazuje vzorku optické vlastnosti nezávisle na jeho pozici. To přináší obtíže při snaze o izolaci objektu, který je tvořen shlukem neunikátních datových hodnot.

Například při prohlížení CT skenu lidského těla může lékař chtít prozkoumat pouze kostní tkáň. Pro takové zobrazení je potřeba najít přenosovou funkci, která bude kostní tkáni přiřazovat neprůhlednou barvu a ostatním vzorkům maximální průhlednost.

Vytvoření přenosové funkce vyžaduje iterativní přístup. Rychlá a efektivní iterace je umožněna dobrým uživatelským rozhraním. To by mělo návrháři umožnit rychle upravovat parametry a okamžitě vidět výsledné zobrazení. Návrh vhodné přenosové funkce také vyžaduje dobrou znalost zobrazovaných dat.

I s dobrými nástroji je návrh přenosové funkce obtížný. Existující publikace se snažily alespoň částečně automatizovat hledání přenosových funkcí [9].

2.3 Osvětlení objemu

Optické vlastnosti média ovlivňují světlo přes něj procházející.

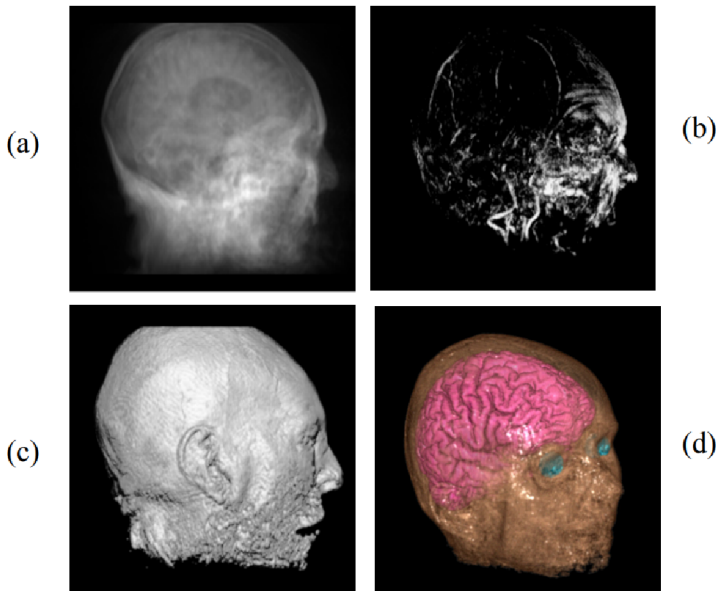
Metody z obrázku 2.2 mají několik společných vlastností. Všechny pracují s paprsky vysílanými skrze pixel a vzorkují interpolované vlastnosti. Liší se v akumulaci těchto vzorků; rentgenový mód vzorky sčítá, MIP[3] do pixelu zapíše pouze nejvyšší navzorkovanou hodnotu, Iso-surface rendering odhaduje hranice objektů a plný objemový rendering simuluje světlo procházející objemem. [8]

Realistický optický model je velmi složitý. Existují optické modely podstatně jednodušší, cenou za jednoduchost je však horší kvalita výsledného obrazu.

Pro stínování se často používá varianta Phongova osvětlovacího modelu [16, 17]. Tento model počítá lokální intenzitu světla odraženou povrchem. K tomu využívá směr a intenzitu zdroje světla a normálu povrchu, která je u volumetrických dat odhadována z gradientu. Jako lokální osvětlovací metoda ignoruje odrazy světla a stíny.

[2] Více o interpolaci v kapitole „Interpolace dat“ 2.4.

[3] Maximum Intensity Projection



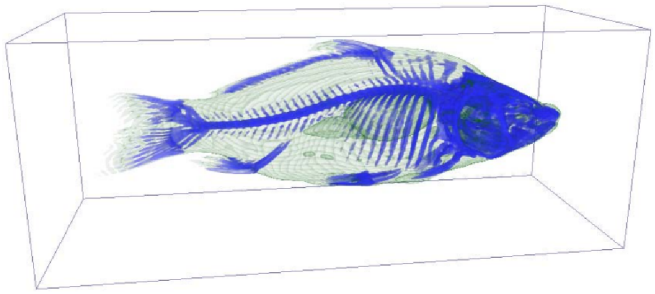
Obrázek 2.2: Model zobrazený různými technikami: (a) Rentgen, (b) MIP, (c) Iso-surface, (d) Plný objemový rendering (převzato z [8]).

Obrázek 2.3: Porovnání jednoduchého a komplexního osvětlovacího modelu. Globálně osvětlený model (vpravo) bere v potaz stíny vržené okolními objekty (převzato z [7]).

2.4 Interpolace dat

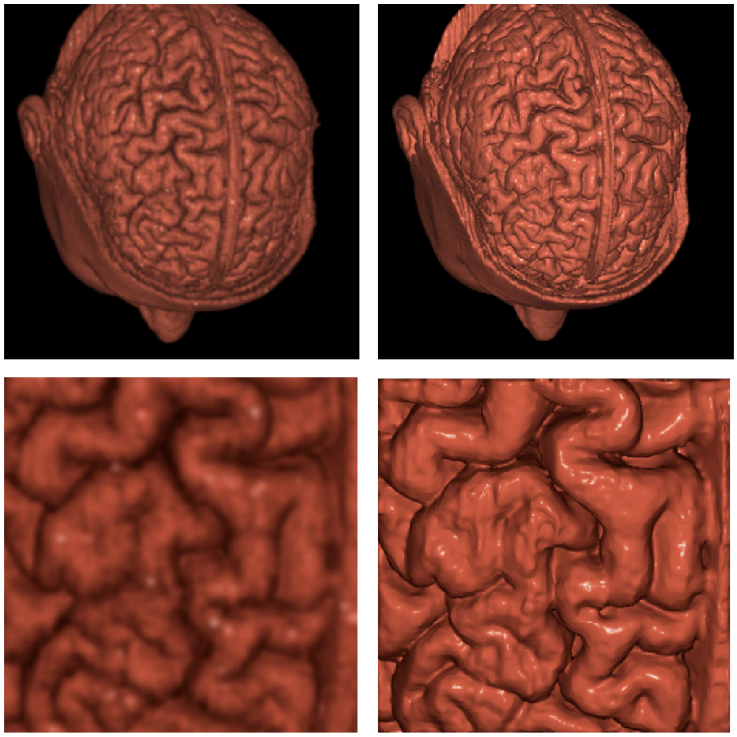
Pro techniky přímého zobrazování volumetrických dat je potřebné, aby bylo možné získat vzorek v libovolném bodě objemu. Tedy i v bodech, které v množině dat nejsou popsány. Tato absence dat je řešena `interpolací` [8]. Definuje se funkce $f(x, y, z)$, pro kterou existuje řešení na celém datovém objemu – $(x, y, z) \in \mathbb{R}^3$.

Body uvnitř objemu, které nejsou explicitně vyjádřeny původními daty, jsou funkcí f doplněny. Parametry funkce f jsou zejména body v okolí x, y, z, které mají definované hodnoty. V případě interpolace dat je nutné přenosovou funkci spočítat pro každý vzorek v každém snímku.



Obrázek 2.4: Volumetricky vyrenderovaný CT sken kapra (převzato z [4]).

Je možno interpolovat buď samotná data, a nebo až jejich optické vlastnosti získané z přenosové funkce [14]. Vzhledem k náročnosti jejího výpočtu se může znatelně prodloužit



Obrázek 2.5: Porovnání interpolace optických vlastností (vlevo) a interpolace vzorků (vpravo) (převzato z [8]).

doba vykreslování. Výhodou je detailnější výsledný obraz. Při interpolaci optických vlastností stačí spočítat přenosovou funkci v datových bodech mřížky. Tento výpočet lze navíc provést ve fázi předzpracování. Za cenu určité ztráty detailu je tato metoda rychlejší. Porovnání je vidět na obrázku 2.5.

Způsobů interpolace trojrozměrných dat v pravidelné mřížce může existovat nekonečně mnoho, v praxi se používá několik metod, které dále blíže popíšu.

Metoda nejbližšího souseda

Interpolace metodou nejbližšího souseda přiřazuje bodům hodnotu nejbližšího vzorku. Metoda nebere v úvahu ostatní okolní body. Funkce má ostré přechody mezi hodnotami, okolí definovaných bodů má stejnou hodnotu jako bod samotný. V pravidelné pravoúhlé síti jsou výsledkem homogenní kostky, výsledný obraz není dostatečně kvalitní pro použití při zobrazování.

Bodu P je přiřazena taková hodnota, jakou má vzorek na souřadnicích

$$P(x, y, z) = (round(P_x), round(P_y), round(P_z)) \tag{2.7}$$

nebo

$$P(x, y, z) = (floor(P_x), floor(P_y), floor(P_z)) \tag{2.8}$$

kde $round$ je aritmetické zaokrouhlení a $floor$ je zaokrouhlení dolů.

Trilineární interpolace

Trilineární interpolace je založena na lineární interploaci. Interpolace se aplikuje postupně po jednotlivých osách. Na pořadí aplikace interpolací nezáleží.

Vzorkovaný bod se nachází uvnitř buňky, která je definovaná 8 vrcholy. V prvním kroku se provádí 4 lineární interpolace, a to mezi dvojcemi bodů sdílející 2 souřadnice. Pokud například jako první provedu interpolace podél osy x, budu interpolovat dvojce se shodnými souřadnicemi y a z. Výsledkem jsou 4 body, se kterými se dál pracuje. Původních 8 bodů buňky, ve které se vzorek nachází, již dál neuvažujeme.

Ve druhém a třetím kroku interpolujeme dvojce bodů po zbylých osách. Výsledkem je jediná hodnota – interpolovaná hodnota. Tento postup je graficky znázorněn na obrázku 2.6.

Body jsou značeny p_{000} až p_{111}, kde trojce binárních číslic v dolním indexu značí, ve kterých rovinách bod leží. Číslicí 0 je označována rovina blíž k počátku souřadnic.

Postup výpočtu interpolace pro bod $P[x, y, z]$

Prvním krokem je nalezení osmi bodů definujících buňku, ve které se bod P nachází. Tyto body v pravidelné mřížce získáme zaokrouhlením souřadnic bodu P dolů a nahoru.

Dále stanovíme relativní vzdálenosti bodu od stěn buňky po jednotlivých osách.

$$u = \frac{x - p_{000x}}{p_{100x} - p_{000x}}, u \in \langle 0; 1 \rangle \tag{2.9}$$

$$v = \frac{y - p_{000y}}{p_{010y} - p_{000y}}, v \in \langle 0; 1 \rangle \tag{2.10}$$

$$w = \frac{z - p_{000z}}{p_{001z} - p_{000z}}, w \in \langle 0; 1 \rangle \tag{2.11}$$

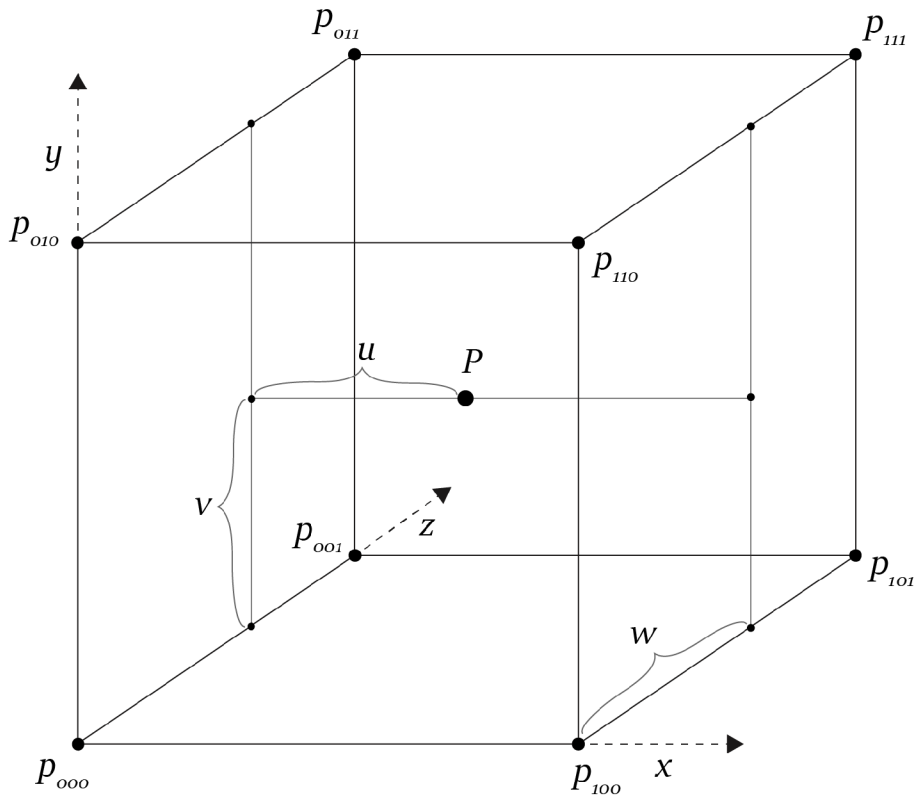
Poté aplikujeme lineární interpolaci podél jedné osy (x). Vzniknou 4 body v jedné rovině.

$$p_{00} = p_{000} \cdot (1 - u) + p_{100} \cdot u \tag{2.12}$$

$$p_{01} = p_{001} \cdot (1 - u) + p_{101} \cdot u \tag{2.13}$$

$$p_{10} = p_{010} \cdot (1 - u) + p_{110} \cdot u \tag{2.14}$$

$$p_{11} = p_{011} \cdot (1 - u) + p_{111} \cdot u \tag{2.15}$$



Obrázek 2.6: Vizualizace trilineární interpolace. Výsledné vlastnosti bodu P jsou složeny z bodů p_{xyz} pomocí vah u, v, w.

Obdobně podél os y a z:

$$p_0 = p_{00} \cdot (1 - v) + p_{10} \cdot v \tag{2.16}$$

$$p_1 = p_{01} \cdot (1 - v) + p_{11} \cdot v \tag{2.17}$$

$$p = p_0 \cdot (1 - w) + p_1 \cdot w \tag{2.18}$$

Výsledná hodnota p je výsledkem interpolace a představuje hodnotu objemu v bodě $P[x, y, z]$

$$f(x, y, z) = p \tag{2.19}$$

Zhodnocení metody

Poměr kvality výsledku a rychlosti z této metody činí dobrého kandidáta pro použití v implementaci interaktivního vykreslovacího programu.

Výpočetně tato metoda není náročná. Pokud mají mezi sebou voxely vzdálenost 1 jednotku, výpočet se stane ještě jednodušším.

2.5 Přímé zobrazování volumetrických dat

Metody zmíněné v tomto přehledu pracují přímo nad volumetrickými daty. Výsledný obraz se skládá z pixelů, jejichž hodnota se získává přímým vzorkováním volumetrických dat.

Druhá skupina metod zobrazování, kterou jsem v práci nerozvíjel, jsou metody povrchového renderování (*surface rendering*). Principem je detekce povrchů v objemu, jejich převedení na síť mnohoúhelníků a následné vykreslení tradičními renderovacími metodami. Nejznámější metodou z této skupiny je algoritmus *Marching cubes*. Výhodou je rychlost vykreslení, nevýhodou je ztráta informací. Rozdíly porovnává práce (P. M. A. van Ooijen, [15]).

2.6 Ray casting

Volumetrická varianta algoritmu Ray casting začíná vrháním paprsků do scény, jeden paprsek pro každý pixel. Podél paprsku je objem vzorkován v pravidelných krocích za použití interpolace. Vzorky se obarvují přenosovou funkcí a akumulují se do výsledné hodnoty pixelu.

Kamera

Pro získání obrazu je potřeba pro každý pixel obrazu vyslat jeden paprsek. Tyto paprsky generuje takzvaná kamera [2].

Dva základní typy projekcí kamer jsou ortografická a perspektivní.

Ortografická kamera vysílá paprsky kolmé na projekční rovinu. Ortogonální/paralelní projekce zachovává rovnoběžky, objekty vzdalující se od kamery se jeví stále stejně velké.

Perspektivní kamera vysílá paprsky sbíhající se v bodě P. Perspektivní projekce je používaná nejběžněji. Výsledky se jeví přirozenější, protože lidské oko funguje na stejném principu. Na obrazu získaném perspektivní projekcí se vzdálenější objekty jeví menší a nemusí být zachována rovnoběžnost.

Kamera navržena pro generování paprsků může mít odlišnosti od typické rasterizační kamery.

Kamera je definovaná jako pětice

$$C = (P, D, U, R, F) \tag{2.20}$$

kde P je poloha kamery, D je směr pohledu, U je orientace směru nahoru, R je rozlišení kamery a F je zorné pole ([6], kapitola 13).

Vektor orientace U pomáhá definovat přesnou orientaci kamery. Jeho dopad na výslednou orientaci se dá přirovnat k naklonění hlavy do strany. Často se volí jako *jednotkový* vektor po směru osy y ($[0, 1, 0]$, směr nahoru), což zajistí přirozenou orientaci kamery. Od vektoru U se také odvíjí orientace projekční plochy kamery, musí se proto transformovat tak, aby se směrem pohledu svíral úhel 90 stupňů. Pokud je kamera namířena směrem vektoru U ($D = \pm U$), výpočet je nestabilní.

Rozlišení kamery R je dvojce celých čísel (W, H), která vyjadřuje horizontální a vertikální počet pixelů obrazu. Rozlišení ovlivňuje poměr stran projekční plochy, nemusí ale být definováno přímo na kameře. V definici kamery se dá R zaměnit za poměr stran $aspect = \frac{W}{H}$, čímž se stává nezávislou na konkrétním rozlišení. Pro délky horizontální a vertikální hrany projekční plochy platí, že jsou vůči sobě v poměru $aspect$.

F představuje úhel zorného pole. Ten popisuje, jak široce od směru pohledu D kamera „vidí“. Bývá uváděn pouze jeden z úhlů (vertikální), protože definovat oba úhly je redundantní.

Inicializace kamery

Popsaná kamera je převzata ze zdrojového kódu knihovny OSPRay [21]. Hlavní prvky kamery jsou ilustrovány na obrázku 2.7.

Prvním krokem je vytvoření pravoúhlého systému souřadnic kamery (d, r, u).

1. $\vec{d}$ – směr kamery ($\vec{d} = D$)
2. $\vec{r}$ – směr „doprava“
3. $\vec{u}$ – směr „nahoru“

Vektor $\vec{r}$ je vypočítán jako

$$\vec{r} = \vec{d} \times \vec{U}, \quad |\vec{d}| = |\vec{U}| = 1 \tag{2.21}$$

Vektor $\vec{u}$ je vypočítán jako

$$\vec{u} = \vec{r} \times \vec{d} \tag{2.22}$$

Nyní je možné sestrojit obdélníkovou plochu, přes kterou budou paprsky vysílány, takzvanou *projekční plochu*. Projekční plocha je kolmá na směr pohledu, pro jednoduchost bývá od polohy kamery vzdálena jednu jednotku ve směru kamery. Horizontální hrany jsou rovnoběžné s vektorem $\vec{r}$ a vertikální hrany jsou rovnoběžné s vektorem $\vec{u}$. Zároveň je z bodu P obdélník pozorován pod úhlem F a střed plochy určuje průnik směru kamery.

Výšku plochy získáme jako

$$S_y = 2 \cdot \tan \frac{F_h}{2} \tag{2.23}$$

Šířku vypočítáme obdobně z horizontálního pozorovacího úhlu, nebo dopočítáme z S_y jako

$$S_x = S_y * aspect \tag{2.24}$$

Takto definovaná plocha je popsána homogenními souřadnicemi v rozsahu $\langle 0, 1\rangle \times \langle 0, 1\rangle$. Počátek těchto souřadnic určeme podle konvence jako levý horní nebo levý dolní roh. Z bodu P bude vektor dir_{00} směřovat na počátek homogenních souřadnic. Sestrojíme vektory du a dv o délkách S_x a S_y ve směrech $\vec{r}$ a $\vec{u}$.

S pomocí takto definovaných vektorů dokážeme libovolný paprsek s homogenními souřadnicemi (u, v) určit jako přímku procházející bodem P a směrem $\vec{s}$ určeným vztahem

$$\vec{s} = dir_{00} + u \ \cdot du + v \ \cdot dv \tag{2.25}$$

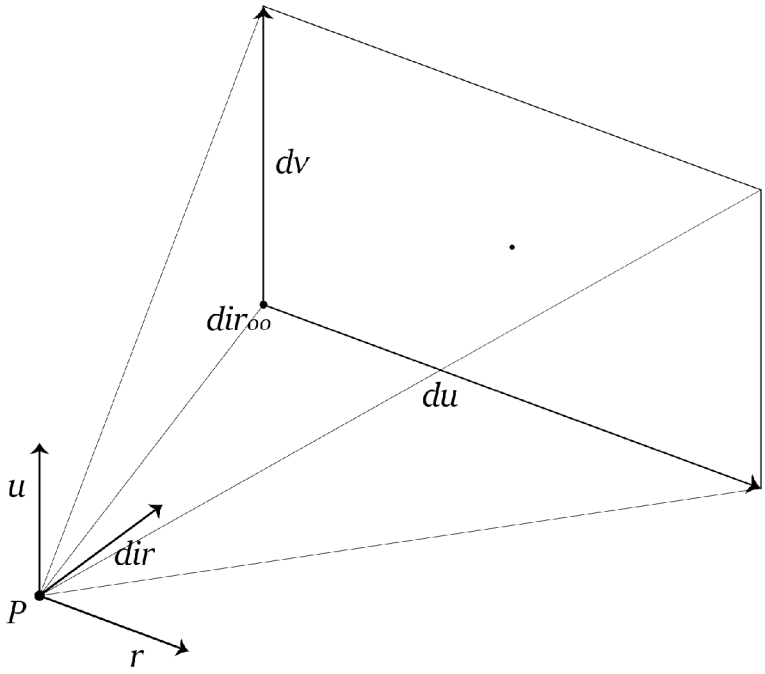
Akumulace barvy

Každý paprsek, který protíná objem, nyní tento objem vzorkuje na pravidelných intervalech.

Vzorkování může probíhat v pořadí zepředu dozadu (*front-to-back compositing*), nebo zezadu dopředu (*back-to-front compositing*). Vzorkování front-to-back umožňuje optimalizovat sběr vzorků (viz kapitola 2.7), a proto je častěji používaná. [8]

Počátek a konec vzorkovaného intervalu paprsku je zarovnán. Způsob zarovnání má vliv na artefakty při renderování, obzvlášť u hranic objemu. Vliv zarovnání je menší pro velké objemy. [20]

Obrázek 2.8 ukazuje různé typy zarovnání.



Obrázek 2.7: Ilustrace prvků definujících kameru. Vektory dir_{00}, du a dv umožňují jednoduché sestrojení paprsků pro Ray casting.

Pro získání vzorku v libovolném místě objemu se používá interpolace. Každý vzorek je převeden přenosovou funkcí na dvojici (c_s, α_s) a přičten podle následujícího vztahu [8]:

$$c_0 = 0 \tag{2.26}$$
$$\alpha_0 = 0 \tag{2.27}$$
$$c_{i+1} = c_s \alpha_s (1 - \alpha_i) + c_i \tag{2.28}$$
$$\alpha_{i+1} = \alpha_s (1 - \alpha_i) + \alpha_i \tag{2.29}$$

kde c_i je akumulovaná barva, c_s je barva vzorku, α_i je akumulovaná průhlednost a α_s je průhlednost vzorku.

Výsledná barva pixelu je vypočítána jako $c \cdot \alpha$.

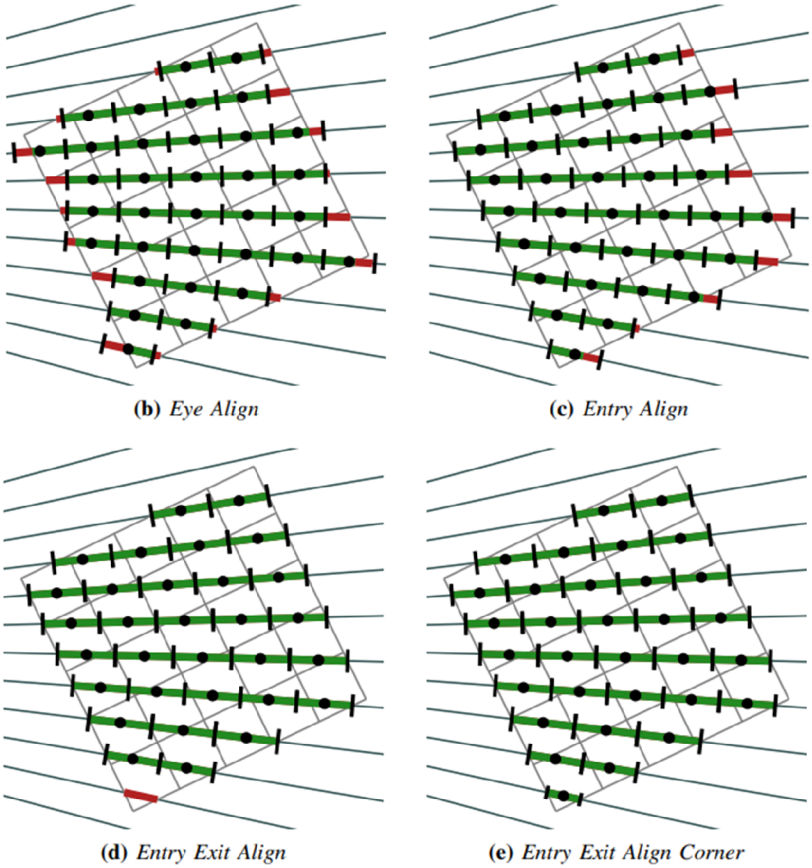
2.7 Optimalizace Ray castingu

Přímé zobrazování objemů má vysokou časovou složitost závislou na jeho rozlišení vztahem $O(n^3)$ [13]. Ve snaze tento úkol zefektivnit byly navrženy různé optimalizační techniky.

Krok vzorkování

Jednoduchou, ale efektivní optimalizací je zvětšit vzorkovací krok a tím vykreslit scénu se zlomkovým počtem vzorkování. Zvolit vhodný krok je zásadní. Příliš velký krok má značný negativní efekt na kvalitu obrazu.

Délka kroku může být adaptivní, k rozhodování o délce kroku jsou potřeba další informace o objemu.



Obrázek 2.8: Typy zarovnání vzorků vůči hranicím objemu (převzato z [20]).

ERT – Early Ray Termination

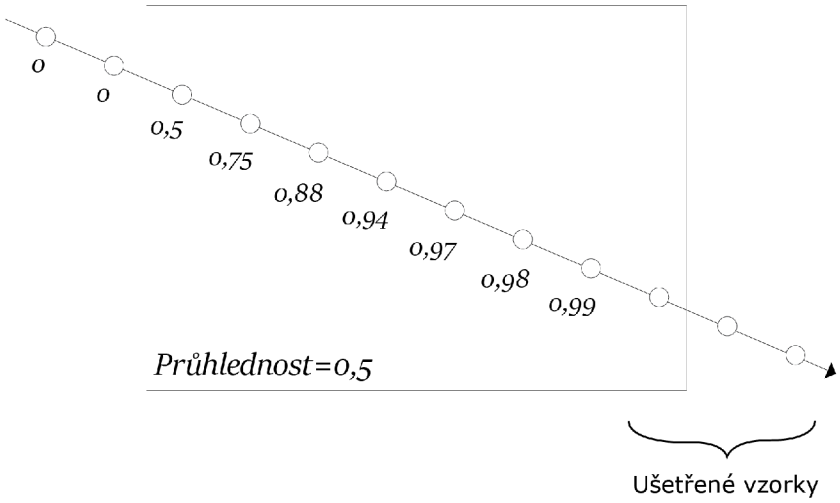
Myšlenka optimalizace ERT je taková, že akumulovaná hodnota paprsku se může ustálit. Pokud je hodnota ustálená, další akumulace je zbytečná, protože neovlivňuje výsledný obraz. Jakmile je takový bod identifikován, vzorkování lze pro daný paprsek předběžně ukončit a tím snížit počet vzorků, které je nutné vypočítat. [12]

Stálost akumulované hodnoty je závislá na akumulované průhlednosti. Ta je v průběhu akumulace sledována a prohlášena za ustálenou v případě překročení stanovené úrovně. Tato mez je typicky nastavena pevně jako hodnota blížící se 1 (např. $0,99$). Ilustrace vzorkování, které je ukončeno předčasně, je ilustrováno v obrázku 2.9.

ESS – Empty space skipping

Přeskakování prázdného prostoru je obzvláště výhodné pro objemy s velkými průhlednými zónami. Objem je rozdělen na pravidelné krychlové oblasti, informace o „prázdnosti“ oblastí se uloží do datové struktury. Struktura může být víceúrovňová, jak to navrhuje práce [12], nebo jednodušší jednoúrovňová.

K určení, jestli je oblast průhledná, je potřeba všechny její vzorky ohodnotit přenosovou funkcí. Data se musí aktualizovat po změně přenosové funkce.



Obrázek 2.9: Ilustrace ERT. Paprsek protíná objem s průhledností $0,5$. S každým vzorkem akumulovaná průhlednost roste, až přesáhne stanovenou hranici $0,99$. Tím je iterace ukončena a je ušetřeno několik vzorkování.

Preprocessing dat

Pro maximální efektivitu a dosažení interaktivního zobrazení dat (mnoho snímků za sekundu) může být vhodné data předzpracovat. Tím je myšleno buď přidání nových struktur, a nebo přeuspořádání stávajících datových struktur.

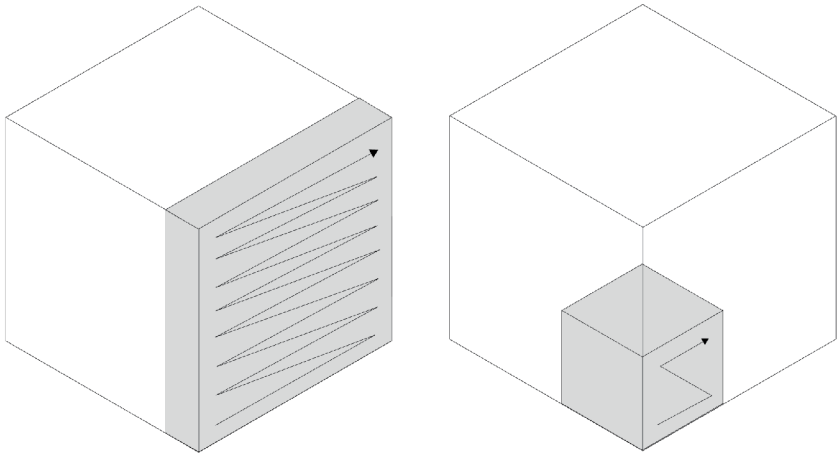
Vykreslovací algoritmus a struktura dat mezi sebou úzce souvisejí. Výše zmíněná optimalizace ESS vyžaduje předzpracování.

Rozložení v paměti

Přístup do hlavní paměti je příliš pomalý, a proto se urychluje vyrovnávacími paměťmi. Ty musí předpovídat, ke kterým datům bude procesor přistupovat, aby nedošlo k výpadkům (*Cache miss*). Mezi heuristiky, které jsou k předpovídání používány, patří *lokalita dat*. Lokalita dat je předpoklad, že pokud je přistoupeno k datům na určité adrese, pravděpodobně bude přistoupeno i k datům v okolí této adresy. [1]

Při vzorkování volumetrických dat je přistupováno k osmi v objemu sousedícím hodnotám. Fakt, že dva vzorky sousedí v objemu, neznamená, že sousedí v paměti. Naopak, jejich vzdálenost může být velká. Při pohledu na obrázek 2.10 můžeme vidět, že pokud jsou data rozložena po řezech na ose x, mezi dvěma vzorky sousedícími na ose x leží v paměti dalších $y_{max} \cdot z_{max}$ vzorků. U rozsáhlých objemů tato vzdálenost v paměti může být v řádu megabajtů. Takové přístupy mohou vyvolávat výpadky vyrovnávacích pamětí, což by negativně ovlivnilo rychlost výpočtu.

Optimalizace spočívá v přeuspořádání vzorků v paměti tak, aby jejich blízkost v objemu více korespondovala s blízkostí v paměti. Tím by se měla snížit šance na výpadek vyrovnávací paměti a zvýšit rychlost výpočtu. Jedním z vhodných rozložení je rozdělení objemu na krychlové podobjemy (bloky). Tyto bloky mají mnohem menší řezy a tedy bližší vzorky.



Obrázek 2.10: Ilustrace rozložení vzorků po řezech (vlevo) a po blocích (vpravo)

Změna typu vzorků

Jednorázový převod celočíselných vzorků na vzorky v plovoucí desetinné čárce ve stádiu předzpracování ušetří opakované provádění těchto drahých převodů při renderování. Negativem je násobně větší velikost objemu v paměti, proto tato optimalizace není vhodná pro rozsáhlé objemy.

Level of detail

Při vzorkování se dá technika interpolace vybírat pro každý vzorek zvlášť na základě zdánlivé velikosti buňky z pohledu kamery. Čím se buňka jeví menší, tím je možno použít výpočetně levnější interpolaci, aniž by byla významně ovlivněna kvalita obrazu.

Vzdálenost od pozorovatele využívá i optimalizace *adaptivního vzorkování*. Ta se zvyšující se vzdáleností od kamery prodlužuje vzorkovací krok. [23]

2.8 Paralelizace Ray castingu

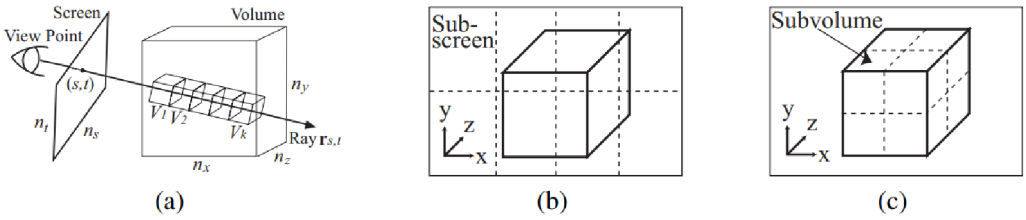
V algoritmu Ray casting se paralelizace přímo nabízí. Výpočty jednotlivých pixelů jsou na sobě nezávislé, proto nejjednodušší variantou je rozdělit úkoly renderování jednotlivých pixelů mezi dostupná vlákna a do framebufferu posbírat výsledky. Narazíme však na omezení, z nichž je největším rychlost paměti. Pro návrh efektivnějšího algoritmu musí být brány v potaz charakteristiky pamětí a cache.

Paralelní algoritmy Ray castingu se dělí na dvě skupiny: ty využívající paralelismu v obraze (*screen-parallel*) a ty využívající paralelismu v objemu (*object-parallel*), viz obrázek 2.11.

Paralelismus v obraze rozdělí obrazovku na n podobrazovek, někdy nazývané dlaždice (*tiles*). Jedno vlákno zde vzorkuje celý paprsek, proto je jednoduché implementovat ERT. Podobrazovky mohou být přidělovány tak, aby se všechna vlákna dobře vytížila. Nevýhodou však je, že každé vlákno operuje nad celým objemem, což je drahé pro rozsáhlé objemy.

Princip paralelismu v objemu spočívá v rozdělení objemu na n podobjemů. Podobjemy jsou přidělovány vláknům, která je vykreslí. Tím vznikne n překrývajících se podobrazů, které je potřeba seskládat dohromady. Paprsek je v tomto schématu vzorkován více vlákny,

čímž se značně ztěžuje globální aplikace ERT. ERT v rámci podobjemů je ovšem stále možná.



Obrázek 2.11: Paralelizace Ray castingu využívající paralelismu (b) obrazu a (c) objektu [13]

Segmented Ray Casting

Tento paralelní algoritmus je popsán v práci [5]. Celý objem je v rámci preprocessingu rozdělen do bloků, typicky krychlí. Data jsou integrována podél paprsků paralelně po jednotlivých blocích, částečné výsledky jsou skládány do výsledného obrazu.

Přehled algoritmu: Bloky objemu jsou postupně přidělovány vláknům. Vlákna vyfiltrují paprsky, které procházejí jejich blokem. Vlákna akumulují data podél paprsku na jim přiděleném bloku. Výsledek posílají řídícímu vláknu. Řídící vlákno skládá výsledky přicházející z bloků do jediného kanvasu.

Prvním krokem je určení paprsků procházejících daným blokem. Přímočará metoda spočívá ve výpočtu průniku paprsku a stěn bloku dat. Efektivnější metodou je promítnout siluetu bloku na plochu kamery (near plane). Siluetu získáme aplikací transformační matice (view matrix) na tři z pohledu kamery viditelné stěny. Rasterizační algoritmus jako například Scan-line pak získá množinu paprsků procházející blokem.

Druhým krokem je vzorkování. Úkol vzorkování jednoho bloku je přidělován vláknům. Vzorkování bloku se provádí stejně jako vzorkování celého objemu. Vzorky jsou akumulovány podél paprsku v pravidelných intervalech.

Aplikace optimalizací je komplikovaná, protože při vzorkování na bloku dat není znám kontext z okolí. Výsledek vzorkování se nemusí projevit na výsledném obrazu, čímž by byl výpočet zbytečný. Rychlost získaná paralelizací by ale měla převážit tyto ztráty efektivity.

Akumulování vzorků je asociativní [5], což znamená, že vzorky můžeme akumulovat nezávisle v rámci jednotlivých bloků.

Výsledné hodnoty barev a průhledností jsou zaslány vláknu, které má na starost výsledné složení obrazu. Výsledky částečných akumulací musí být zpracovány v pořadí viditelnosti bloků (není komutativní). Každému bloku je proto přiřazeno číslo vyjadřující pořadí překryvu. V pořadí tohoto indexu se výsledky skládají do finálního rasteru.

2.9 OSPRay

Intel® OSPRay[4] je ray tracingová renderovací knihovna pro stavění interaktivních aplikací a je představitelem existujících řešení pro renderování objemových dat [21]. Knihovna má otevřený zdrojový kód a obsáhlou dokumentaci. Je vydána pod licencí Apache 2.0[5].

Knihovna je navržena pro běh na procesorech s podporou SIMD vektorových instrukcí. Je implementována v jazycích C/C++ a využívá ISPC, rozšíření jazyka C pro snadný návrh paralelních programů.

Je zaměřena na vizualizace a výkon. Knihovna nalézá využití ve vědeckých vizualizacích, v menší míře ve filmových produkcích.

Aplikační rozhraní využívá objektově orientovaný návrh. Všechny entity (renderer, objemy, kamery, ...) jsou specializací `OSPObject`. Parametry se objektům předávají pomocí `ospSetParam`. Objekty mají transakční sémantiku, změny se projeví po volání `ospCommit(obj)`.

Scéna obsahuje množinu *skupin*. Skupina je množina objemů, geometrií a zdrojů světla. OSPRay pracuje s geometrickými primitivy i objemovými daty, podporuje tedy i renderování povrchových objektů, jako jsou polygonové sítě, koule, roviny a křivky.

Zajímavé je, že lze (metodou `ospSetRegion`) naplnit objem daty i bez znalosti jeho vnitřní struktury. To se hodí pro typ objemu využívající rozdělení na menší blokové podobjemy pro zlepšení datové lokality (`block_bricked_volume`).

Práce s volumetrickými daty je předávána knihovně OpenVKL[6], proto podporuje stejné typy volumetrických dat. Kromě typického mřížkového objemu podporuje sférické objemy (`StructuredSphericalVolume`) a nestrukturované částicové objemy (`ParticleVolume`). Objemy typu `OSPVolume` jsou obaleny strukturou `VolumetricModel`, která přidává informace o objemu relevantní k renderování, zejména přenosovou funkci.

Přenosové funkce jsou implementovány jako vyhledávací tabulky. Funkce je definována tabulkou barev, tabulkou průhledností a intervalem hodnot vzorků. Jednotlivé hodnoty barev a průhledností jsou pomocí interpolace namapovány na rozsah hodnot vzorků.

Využívají se tři typy interpolací: metoda nejbližšího souseda, trilineární a trikubická interpolace 2.4.

Rozhraní pro renderování je asynchronní. Parametry jsou 4 objekty: FrameBuffer, Renderer, Camera a World. Voláním `ospRenderFrame` se spustí renderovací úloha.

Renderování jednoho snímku se provádí paralelně po dlaždicích, využívá tedy screen-parallel renderingu. Využit je paralelismus na úrovni vláken a instrukcí, podporována je i distribuce práce mezi více počítači pomocí MPI. Rozdělování dlaždic je vyvažováno (`LoadBalancer.cpp`). Při renderování je použita optimalizace ERT.

O výpočet kolizí paprsků s objekty scény se stará knihovna Embree[7]. Paprsky protínající objemy jsou transformovány do lokálních souřadnic. Získání vzorků z objemů včetně interpolace je pak předáno knihovně OpenVKL.

OSPRay dosahuje interaktivní rychlosti vykreslování. Objem o velikosti 8 GB byl na stanici[8] vykreslen rychlostí 25,1 FPS [21].

[4] https://www.ospray.org/
[5] https://www.apache.org/licenses/LICENSE-2.0
[6] https://www.openvkl.org/
[7] https://www.embree.org/
[8] 2×Xeon 2699 v3 "Haswell", 512 GB RAM

Kapitola 3

Návrh CPU rendereru volumetrických dat v jazyce Rust

V rámci projektu se zaměřím na návrh a implementaci CPU rendereru rozsáhlých volumetrických dat v reálném čase.

Pod rozsáhlými volumetrickými daty rozumím soubory v rozlišení okolo 2000^3 vzorků. Jedná se tedy o soubory, které jsou z důvodu omezené kapacity pamětí dostupných grafických karet vhodnější ke zpracování procesorem.

Interaktivita je pro můj renderer přední požadavek. Proto jsem se rozhodl, že nejvyšší prioritu bude mít rychlost renderování. K dosažení tohoto cíle použiji efektivní paralelní algoritmus pro CPU.

Jako implementační jazyk jsem zvolil Rust. Doménou tohoto odvětví jsou především jazyky C a C++. Jelikož v jazyce Rust neexistuje rozšířený volumetrický renderer, podcílem této práce je prozkoumat, zda je k tomuto úkolu Rust vhodný. Domnívám se, že by vhodný být mohl, protože se jedná o nízkoúrovňový jazyk překládaný do nativních instrukcí a má dobrou podporu pro psaní paralelních programů.

Implementace rendereru bude mít podobu knihovny. Jako knihovna bude mít veřejné aplikační rozhraní umožňující jednoduchou interakci. Bude podporovat zpracování jednoduchých volumetrických formátů a několik reprezentací dat v paměti s možností rozšíření o podporu dalších formátů uživatelem.

Pro účely demonstrace fungování rendereru navrhnu dvě aplikace. Úkolem první aplikace bude generovat rozsáhlá volumetrická data, která využiji při vývoji rendereru a zároveň při demonstraci výsledků. Interakce bude mít formu jednoduchého CLI.

Jako druhá bude aplikace s grafickým uživatelským rozhraním, která bude umožňovat interaktivně zobrazovat volumetrická data. Rozhraní aplikací nepatří mezi cíle mé práce, proto nebudu věnovat zvláštní pozornost analýze požadavků, ergonomice a přenositelnosti.

Úspěch implementace ověřím sadou testů a benchmarků měřící rychlost a porovnávající účinek implementovaných optimalizací s různými parametry.

Přehledově se návrh dělí na následující komponenty:

- renderování – paralelní algoritmus a stínování,
- práce s daty – parsování a reprezentace objemů,
- návrh doprovodných aplikací.

3.1 Renderování metodou Ray casting

Knihovna bude podporovat renderování sekvenční i paralelní. Sekvenční renderování odpovídá popisu z kapitoly 2.6. Bude sloužit k ladění, zhodnocení optimalizací a validaci optimalizovaného paralelního řešení.

Renderery poskytnuté knihovnou budou operovat v blokujícím i neblokujícím režimu. K realizaci neblokujícího rendereru bude muset být navrženo asynchronní rozhraní pro komunikaci mezi klientem a rendererem.

Toto rozhraní bude poskytovat sdílení bufferu pro hotové snímky, komunikaci příkazů rendereru, uvědomění klienta o hotovém snímku a mechanismus ovládání kamery mezi snímky.

Přehled paralelního algoritmu

Paralelní algoritmus je inspirován prací [13]. Je založen na *object-parallel* renderingu (viz sekce 2.8) a jeho základní fáze jsou:

1. rozdělení objemu na bloky,
2. rozdělení úkolů podle viditelnosti bloků v dlaždicích,
3. paralelní renderování bloků a paralelní kompozice,
4. složení výsledného obrazu.

Algoritmus je popsán schématem na obrázku 3.1.

Optimalizace využité algoritmem jsou ERT (Early Ray Termination), ESS (Empty Space Skipping) a paralelizace.

Při klasickém *object-parallel* renderingu se bloky vykreslují individuálně, aniž by se zkoumalo, jestli daná část bloku přispěje k výslednému obrazu. Tento problém viditelnosti je řešen renderováním bloků v pořadí viditelnosti a sdílením akumulovaných průhledností, díky kterému je zachováno globální ERT. Nevýhodou tohoto přístupu je ztráta paralelismu ve směru pohledu kamery, protože bloky od kamery vzdálenější musí být renderovány až po renderování bloků bližších.

Vlákna jsou rozdělena do dvou skupin. *Renderovací Vlákna* (RV), jejichž úkolem je renderovat bloky do *dlaždic* a *Kompoziční Vlákna* (KV), která renderování řídí. Tyto druhy vláken mezi sebou mají dynamiku master/slave, kde KV zadávají renderovací úkoly pro RV.

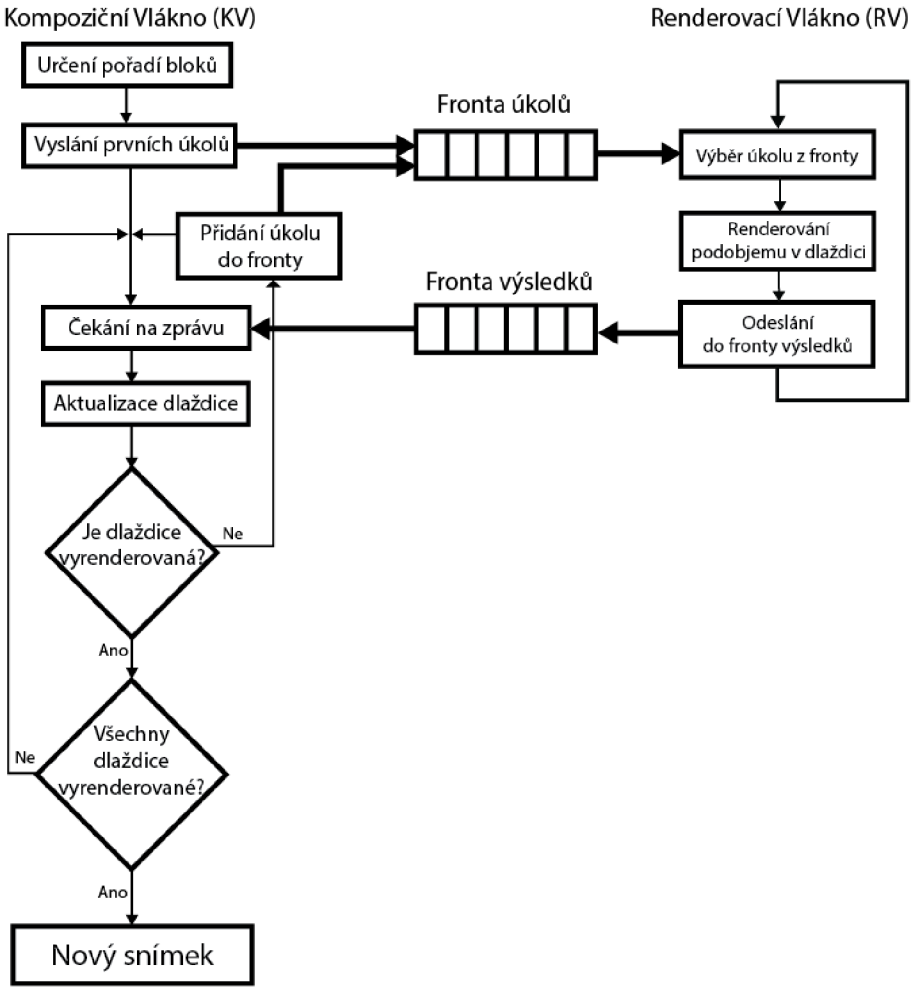
Přípravná fáze

V této fázi se objem rozdělí do bloků a viewport se rozdělí do dlaždic. Počty bloků a dlaždic závisí především na rozsáhlosti objemu a na počtu hardwarových vláken. Optimální parametry určím pomocí experimentů.

Všechny bloky budou přístupné všem vláknům ve sdílené paměti, aby mohl být úkol renderování kteréhokoliv bloku přidělen kterémukoliv RV. Přístup k blokům bude pouze čtecí, nebude tedy třeba přístupy synchronizovat.

Dále jsou bloky seřazeny podle jejich vzdálenosti od kamery od nejbližších k nejvzdálenějším. Ke každé dlaždici je sestrojena fronta, která obsahuje bloky, které jsou v ní viditelné.

V této fázi se také dají eliminovat prázdné bloky v rámci ESS (teorie 2.7). Oproti odhalení prázdných bloků při jejich renderování se tím ušetří mezivláknová komunikace.



Obrázek 3.1: Schéma renderovacího algoritmu. Dva druhy vláken jsou spojeny dvěma frontami pro zprávy.

Řazení bloků je nutné provést po každé změně polohy kamery, protože se může změnit pořadí bloků vůči kameře. Fronty dlaždic se musí budovat pro každý nový snímek.

Kompoziční vlákna

Kompoziční vlákna rozdělují úkoly renderovacím vláknům. Úkoly jsou přidány do společné fronty úkolů, ze které si RV úkoly odebírají. Úkol tedy není přiřazen konkrétnímu RV.

Po začátku renderování snímku je fronta úkolů prázdná a musí se inicializovat. Z fronty každé dlaždice je vyjmuta reference na blok a je spolu s dlaždicí zařazena do fronty úkolů. Tato inicializace může být provedena paralelně. Tím je renderování každé dlaždice zahájeno a KV přecházejí do pasivního režimu, ve kterém čekají na hotové úkoly.

Zpráva je z fronty hotových úkolů vyzvednuta a zpracována některým z Kompozičních Vláken. Obsah zprávy zahrnuje data, kterými je daná dlaždice aktualizována. Pokud fronta dlaždice obsahuje další blok, je z fronty odebrán a odeslán jako součást nového renderovacího úkolu. V opačném případě je dlaždice označena za hotovou a může být kopírována do framebufferu. KV udržují počet dokončených dlaždic, kterým zjistí, jestli je renderování ukončeno. V tom případě uvědomí řídící vlákno.

Renderovací vlákna

Renderovací vlákna odebírají renderovací úkoly z fronty, úkol zpracují a pošlou zpět informaci o dokončení.

Zpráva o renderovacím úkolu zahrnuje obsah a pozici dlaždice a odkaz na blok, který do ni mají vyrenderovat.

Samotný blok je renderován algoritmem Ray casting (sekce 2.6). Zpět posílá pozměněnou dlaždici.

Finální kompozice

Hotové dlaždice jsou do framebufferu kopírovány průběžně, aby nevznikla prodleva mezi ukončením renderování a prezentací hotového obrazu.

Při označování dlaždic za hotové KV kontroluje, zda jsou již v tomto stavu všechny dlaždice. V takovém případě KV vyšle signál značící, že renderování snímku je u konce.

3.2 Osvětlovací model

K získání obrazových informací použiji přenosové funkce (teorie 2.2), které budou převádět vzorky získané z rozhraní objektů.

Zobrazovaný objem je zapotřebí stínovat, aby se ve výsledném obrazu projevil jeho tvar. Za účelem stínování použiji Phongův osvětlovací model. Tento model je vhodný díky jeho dostatečně dobrým vizuálním výsledkům, rychlosti a jednoduchosti implementace.

Phongovo stínování používá normály hranic objektu, ale objem žádné hranice nemá. Musí se proto odhadnout, k čemuž využiji gradient (teorie 2.2).

Scéna bude mít jediné statické světlo.

3.3 Zpracování volumetrických dat

Knihovna bude umět načítat objemy ze souborů na disku. Obsah souboru bude zpracován zvoleným parserem do jednotné reprezentace v paměti. Vstupem parseru tedy bude obsah souboru a výstupem zpracovaná metadata. Metadaty zde rozumím znalosti o souboru dostatečné k vybudování některého z mnoha typů objemů. Konkrétně: rozlišení, pořadí vzorků, tvar buňek apod.

Objem bude v paměti reprezentován svými hranicemi, daty a přenosovou funkcí. Všechny druhy objemů budou sdílet rozhraní, které bude umožňovat tato data vzorkovat v libovolném bodě za použití interpolace (teorie 2.4). Renderer bude agnostický ke konkrétnímu typu objemu, veškerá práce s objemem bude probíhat přes toto uniformní rozhraní.

Typy volumetrických dat

Knihovna definuje několik typů objemů. Tyto objemy zpravidla počítají s jednobajtovými vzorky. Uživatelé knihovny budou moci definovat nové typy implementující rozhraní objemů.

Prvním typem objemu bude *lineární objem*. Jeho vzorky budou v paměti uloženy po řezech.

Blokový objem bude mít vzorky uspořádané po blocích. Takové uspořádání umožní dobrou dělbu objemu pro paralelní renderování. Vzorky, které v objemu sousedí, budou

v paměti uloženy blíž u sebe, čímž se zlepší lokalita dat a to může znamenat efektivnější renderování. Vzorky na hranicích bloků budou duplikovány, aby se nikdy nemusely interpolovat hodnoty ze dvou různých bloků. Lineární a blokový objem charakterizuje obrázek 2.10.

Obě verze těchto objemů budou mít i *mapovanou/streamovací* variantu. Ta místo načítání souboru do paměti přistupuje k souboru přímo, přes mapování do paměti. Možnosti předzpracování budou omezeny, ale vybudování urychlovacích struktur (například indexu prázdných polí) bude stále možné. Pokud budeme chtít použít paralelní algoritmus na vykreslení mapovaného souboru, musí být soubor sám v blocích uložen.

Floatový objem obsahuje vzorky typu plovoucí desetinné čárky. Vzorky jsou převedeny v rámci předzpracování. Paměť potřebná k uložení objemu bude čtyřikrát větší (z jednoho bajtu na vzorek na 4 bajty na vzorek), ale vyhneme se opakovanému převodu při vykreslování.

3.4 Návrh demo aplikace

K demonstraci výsledků práce navrhnu jednoduchou grafickou demo aplikaci. Tato aplikace bude využívat volumetrickou renderovací knihovnu skrze její veřejné rozhraní.

Obrázek 3.2 ukazuje základní rozvržení grafického rozhraní. Menu na levé straně umožní výběr objemu k zobrazení. Prvním krokem bude výběr souboru z disku. Uživatel bude muset znát formát souboru a zvolit vhodný parser tohoto formátu. Pro jednoduchost budou parsery i přenosové funkce předdefinované, tedy neupravitelné interaktivně, ale pouze ve zdrojovém kódu. Další záložka bude obsahovat přepínač přenosové funkce.

Než se objem začne zobrazovat, uživatel si ještě zvolí metodu a optimalizace renderování. Na výběr bude mít mezi jednovláknovým a paralelním renderováním, zapnout a vypnout bude moct optimalizace ERT a ESS.

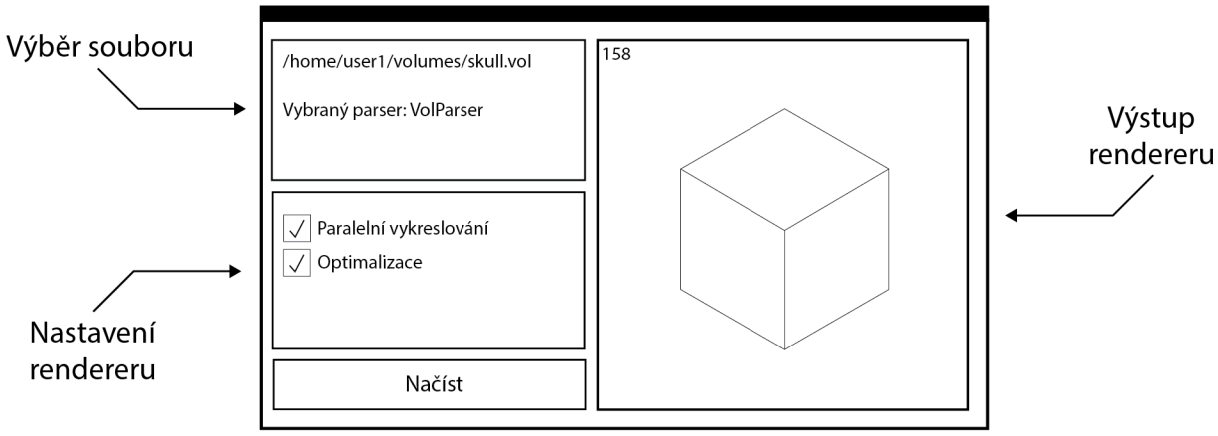
Délka vzorkovacího kroku bude volitelná mezi dvěmi přednastavenými hodnotami. Třetí možností délky kroku bude *adaptivní délka kroku*. Při manipulaci s kamerou se prodlouží vzorkovací krok Ray castingu, čímž se sníží náročnost renderování a zvýší se responzivita. Po ukončení manipulace s kamerou se vyrenderuje snímek ve vyšší kvalitě. Součástí uživatelského rozhraní bude možnost mezi módy přepínat.

Pravá část okna je využita pro obrazový výstup. Výstup rendereru bude mít fixní rozlišení 700x700. Ovládat kameru bude možné tahy kurzorem po renderovacím plátně. Při podržení levého tlačítka myši se bude měnit poloha kamery ve směru tahu a pohyb při držení pravého tlačítka způsobí otáčení kamery. Alternativním způsobem manipulace s kamerou budou tři posuvníky pohybující kamerou po osách.

Druhá karta menu bude sloužit k výběru přenosové funkce, tedy k obarvení objemu.

Aby proces renderování neblokoval interakci s uživatelským rozhraním, musí probíhat na odděleném vlákně. K tomu musím navrhnout komunikaci mezi *event loop*em a rendererem, která bude schopna přenést pohyby kamery, příkazy k renderování a výsledky rendereru. Předpokladem je, že objem je statický a proto bude potřeba renderovat nové snímky pouze při změně kamery, přenosové funkce nebo objemu.

Aplikace bude obsahovat modul udržující stav aplikace.



Obrázek 3.2: Grafický návrh okna demo aplikace a její tři hlavní části.

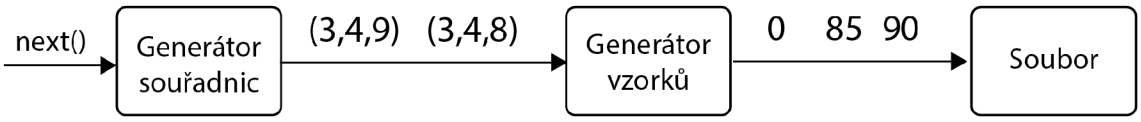
3.5 Návrh aplikace pro generování objemů

Tato aplikace bude sloužit k získání testovacích dat. Díky kontrole nad testovacími daty bude implementace rendereru snazší. Na generovaných datech také vyhodnotím výsledky práce.

Aplikace bude ovládána přes příkazový řádek. Všechny argumenty budou předány při spuštění aplikace. Program na jejich základě vygeneruje volumetrická data, která uloží do souboru. Mezi parametry generování bude patřit:

- rozlišení objemu,
- tvar voxelu,
- rozložení vzorků v paměti,
- typ objemu.

Generování objemu bude konceptuálně rozděleno do dvou úkolů, určení pořadí vzorků v paměti a generování hodnot vzorků (viz obrázek 3.3). Vznikne tak proud vzorků, který se bude zapisovat přímo do souboru.



Obrázek 3.3: Schéma aplikace. Generátor souřadnic určuje pořadí vzorků v paměti, generátor vzorků souřadnicím přiděluje hodnoty. Ty se ve stejném pořadí ukládají do souboru.

Pořadí vzorků

Prvním krokem generování je určení pořadí vzorků.

Vzorky mohou být uspořádány po řezech, nebo po blocích (viz 2.7, podsekce Rozložení v paměti). Bloky budou krychlové a budou mít libovolnou délku strany, která je v celém objemu stejná. Data bloku budou uložena po řezech a bloky budou uloženy v paměti za

sebou. Sousedící bloky budou mít překryv jeden voxel. Tato duplikace části dat představuje určitou marži, ale bude zajištěno, že žádné vzorkování nebude vyžadovat přístup k více blokům.

Entita, která bude na základě vstupních parametrů určovat pořadí vzorků, bude produkovat proud souřadnic v objemu tak, jak se vzorky mají za sebou v souboru objevit.

Získání hodnot vzorků

Generátor hodnot vzorků bude přijímat souřadnice a bude produkovat hodnoty vzorků. Půjde tedy v podstatě o „virtuální“ objem, který bude moci být vzorkován v náhodném pořadí. Vzorky obdržené z generátoru se ve stejném pořadí vpisují do souboru.

Formát souboru

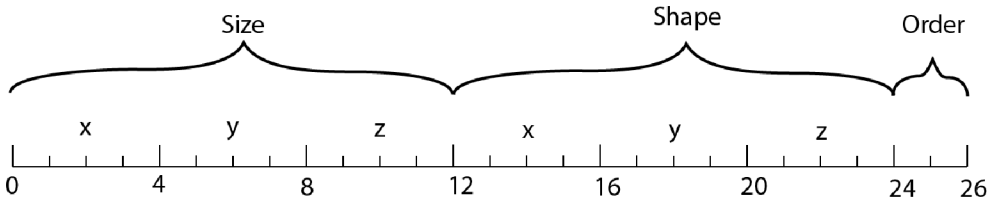
Pro generování dat jsem tento formát navrhl tak, aby minimálně splňoval mé požadavky pro testovací objemy. Formát generovaných souborů je inspirovaný formátem dat dostupných na webu UC Davis[1].

Volumetrický soubor bude začínat hlavičkou, po které budou následovat 8bitové vzorky. Nejrychleji měnící se souřadnice je souřadnice z.

Hlavička bude zabírat 26 bajtů a hodnoty budou v *Little-endian* uspořádání. Rozložení hodnot ilustruje obrázek 3.4. Rozměry objemu (`Size`) jsou celá čísla a tvar (`Shape`) je vyjádřen pomocí čísel plovoucí řádové čárky (IEEE 754 single). Poslední 2 bajty (`Order`) popisují uspořádání vzorků (viz tabulka 3.1).

Uspořádání vzorků	Hodnota bajtu	
	[24]	[25]
Po řezech	1	0
Po blocích	2	délka strany bloku (ve voxelech)

Tabulka 3.1: Význam hodnot v poli `Order`. Pro objemy uložené po blocích obsahuje velikost bloku.



Obrázek 3.4: Rozložení bajtů hlavičky volumetrického formátu. Informace vyjádřené v hlavičce jsou rozlišení a tvar objemu a pořadí vzorků v paměti.

[1] https://web.cs.ucdavis.edu/ okreylos/PhDStudies/Spring2000/ECS277/DataSets.html

Kapitola 4

Implementace rendereru

V této kapitole bude popsána implementace navržených aplikací, použité nástroje a knihovny, detaily o struktuře implementace a některé optimalizace.

4.1 Použité nástroje a knihovny

Jako implementační jazyk aplikací jsem zvolil Rust[1]. Tento jazyk jsem zvolil především kvůli požadavku na maximální rychlost programu. Rust je kompilován do nativního kódu, čímž je výkonnostně na úrovni tradičnějšího jazyku C++.

S jazykem Rust je úzce spojený program Cargo[2], software sloužící k překladu a správě projektů. Zdrojové kódy použitých knihoven jsou při překladu staženy jako balíčky z repozitáře `crates.io`[3].

Programy používají konstrukci rozhraní (*Interface*), který v jazyce Rust existuje pod jménem `trait`. Jelikož Rust není velmi známým jazykem, uvádím tuto souvislost explicitně pro větší komfort při zkoumání implementace.

Použité knihovny

Všechny použité knihovny jsou dostupné z crates.io. Jejich instalace je rozebrána v souboru `README.md` přiloženém na datovém médiu.

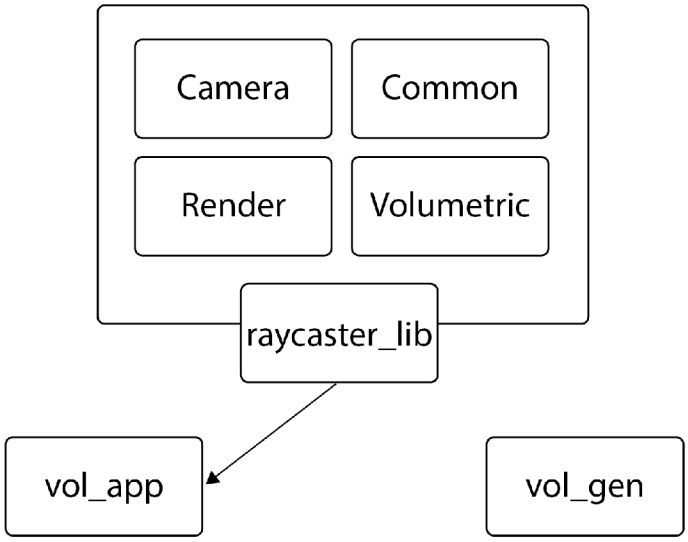
- nalgebra[4] – lineární algebra, operace s vektory a maticemi,
- nom[5] – knihovna pro vytváření parserů. Použito pro parsování volumetrických souborů,
- memmap[6] – mapování souborů do paměti. Využito v objemech typu `LinearVolume` a `BlockVolume`,
- crossbeam[7] – nástroje pro konkurentní programování. Fronty pro mezivláknovou komunikaci,

[1] https://www.rust-lang.org/
[2] https://www.rust-lang.org/tools
[3] https://crates.io/
[4] https://nalgebra.org/
[5] https://github.com/Geal/nom
[6] https://github.com/danburkert/memmap-rs
[7] https://github.com/crossbeam-rs/crossbeam

- slint[8] – GUI framework pro demo aplikaci,
- parking_lot[9] – implementace zámků (`Mutex`, `RwLock`),
- native-dialog[10] – systémová dialogová okna pro výběr souborů z disku,
- byteorder[11] – práce s pořadím bajtů při čtení z paměti (*little-endian*, *big-endian*),
- clap[12] – parsování argumentů příkazové řádky,
- fastrand[13] – generování náhodných čísel (`RNG`),
- criterion.rs[14] – benchmarkovací framework,
- rayon[15] – paralelizační knihovna,
- indicatif[16] – progress bar.

4.2 Struktura implementace

Cargo podporuje sdružení souvisejících projektů do *Workspace*. Členy této skupiny jsou tři aplikace nazývané `raycaster_lib`, `vol_app` a `vol_gen`. Obrázek 4.1 znázorňuje vysokoúrovňové dělení projektu na jeho podčásti.



Obrázek 4.1: Struktura projektu. Demo aplikace (`vol_app`) využívá rozhraní knihovny (`raycaster_lib`).

[8] https://slint-ui.com/
[9] https://github.com/Amanieu/parking_lot
[10] https://github.com/balthild/native-dialog-rs
[11] https://github.com/BurntSushi/byteorder
[12] https://github.com/clap-rs/clap
[13] https://github.com/smol-rs/fastrand
[14] https://github.com/bheisler/criterion.rs
[15] https://github.com/rayon-rs/rayon
[16] https://github.com/console-rs/indicatif

- `raycaster_lib` je renderovací knihovna, jádro celé implementace. Její podčásti jsou:
  - modul `Camera` implementující kameru,
  - modul `Common` s pomocnými strukturami jako paprsek a rozsahy,
  - modul `Render` obsahující jednovláknový i paralelní renderer a rozhraní pro komunikaci s renderery,
  - a modul `Volumetric`, ve kterém jsou definovány typy objemů a rozhraní pro práci s nimi.
- `vol_app` je demo aplikace s grafickým uživatelským rozhraním. Obsahuje výchozí hodnoty, správu stavu aplikace a definici uživatelského rozhraní v jazyce `Slint`.
- `vol_gen` je aplikace pro generování objemových dat. Obsahuje předpis pro parsování argumentů příkazové řádky do popisu objemu a generátory objemů.

4.3 Zpracování dat

Volumetrická knihovna poskytuje několik parserů, které také poslouží jako příklady pro implementaci vlastních parsovacích funkcí. Parsování souborů s objemovými daty je totiž rozšiřitelné uživatelským kódem.

Data nemusí pocházet ze souboru, ale mohou být i generována nebo definována v kódu. Dat definovaných v kódu využívám v testech. Aby data měla konzistentní rozhraní bez ohledu na původ, jsou spravována strukturou `DataSource`. Soubory nejsou načítány do paměti, ale mapovány. Tím je umožněno rozhodnout o reprezentaci objemu v paměti v následujících stádiích tvoření objemu.

`DataSource` je vstupem parseru. Jeho výstupem je struktura popisující soubor tak, aby byla samonosná, tedy dostatečná k vytvoření volumetrického objemu. K vybudování konkrétního objemu je pak nutné, aby struktura implementovala generické rozhraní `BuildVolume<T>`, kde `T` je výstupem parseru.

Knihovna sama poskytuje strukturu `VolumeMetadata`, výstup parseru, pro který je rozhraní `BuildVolume` implementováno na všech předdefinovaných typech objemu. Uživatel si však může vybrat vlastní reprezentaci metadat a implementaci tohoto rozhraní může jednoduše doplnit.

Uživatelům je poskytnuta zjednodušující funkce, která přečte soubor, aplikuje parser a vybuduje strukturu objemu. Její kód je uveden ve výpisu 4.1.

```
pub fn from_file<P, T, M, PF>(path: P, parser: PF, tf: TF)
-> Result<T, &'static str>
where
    P: AsRef<Path>,
    T: BuildVolume<M> + Volume,
    PF: Fn(DataSource<u8>) -> Result<VolumeMetadata<M>, &'static str>,
{
    let ds: DataSource<u8> = DataSource::from_file(path)?;
    let mut metadata = parser(ds)?;
    metadata.set_tf(tf);
    BuildVolume::build(metadata)
```

```
12 }
```

Výpis 4.1: Funkce `from_file` zlepšuje ergonomiku práce s knihovnou vytknutím časté sekvence příkazů do funkce. Zároveň slouží jako příklad budování objemu ze souboru.

4.4 Objemy

Objemem je v knihovně každý typ, který implementuje `trait Volume`. Nejdůležitější funkcí rozhraní je metoda `sample_at_gradient(pos)`, která vzorkuje objem v bodě `pos` a vrací interpolovaný vzorek a odhadnutý gradient v bodě.

Paralelní algoritmus vyžaduje objem rozdělený na bloky. K rozlišení takových objemů existuje `trait Blocked`. Rozhraní poskytuje metody k získání množiny bloků a informace o tom, které bloky jsou viditelné. V rozhraní je typ bloku definován jako takzvaný *asociovaný typ*.

Objemy se vzorky po řezech představují `LinearVolume` a `FloatVolume`. Blokové objemy jsou reprezentovány typy `BlockVolume` a `FloatBlockVolume`.

Mapování do paměti

Stejně jako čtení souborů, i paměť pro vzorky je spravována typem `DataSource`. Ten poskytuje společné rozhraní, ať už konkrétní objem pracuje s mapovaným souborem, nebo vektorem s daty. Práci s mapovanými soubory zapouzdřuje knihovna `memmap`.

Hranice objemu

Objem je ohraničen stěnami kvádru zarovnanými podél os. Důvodem tohoto omezení je efektivita výpočtu průniku paprsku a objemu. Struktura `BoundBox` je v kódu definovaná dvěma body, které značí dolní a horní vrchol kvádru.

Nejdůležitější funkcí této struktury je výpočet průniku s paprskem. Algoritmus je převzat z práce [22]. Jeho výsledkem je nejen informace, jestli průnik existuje, ale i úsek paprsku, který objemem proniká. Úsečka je definovaná dvojicí čísel, `t0` a `t1`. Body jsou z těchto čísel vypočítány vztahem

$$p = pos + t \cdot dir \tag{4.1}$$

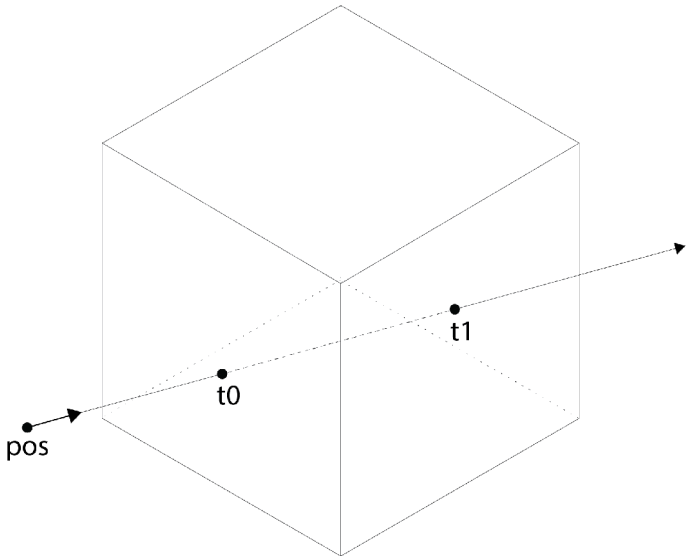
kde *pos* je počátek paprsku, *dir* je *normalizovaný* směr paprsku. viz ilustrace 4.2. Díky tomu se značně zefektivnilo vzorkování objemu, protože je možné předem určit maximální počet kroků po paprsku místo kontrolování pozice po každém kroku.

Návratový typ metody je `Option<(f32, f32)>`. Tento typ jsem zvolil, aby bylo možné vyjádřit možnost, že průnik neexistuje (hodnotou `None`).

EmptyIndex

`EmptyIndex` je akcelerační struktura poskytující informace o viditelnosti vzorků v určité oblasti. Tato oblast je krychlová a její velikost je určena generickou konstantou. Například `EmptyIndex<4>` uchovává informaci o viditelosti voxelů v blocích 4^3. Index je jednoúrovňový. Experimentoval jsem i s víceúrovňovým indexem, ten byl ale nejen složitější, ale také přinesl zhoršení výkonu.

Tato urychlovací struktura je užita u objemů, které nejsou děleny na podbloky. Blokové objemy místo toho používají informaci o viditelnosti samotných bloků.



Obrázek 4.2: Výstup funkce `BoundBox::intersect` – body průniku paprsku s objemem. Bod je z čísel t_0 a t_1 získán rovnicí 4.1.

Tento index musí být vybudován na základě konkrétní přenosové funkce. Pokud je kterýkoliv z voxelů bloku viditelný, je blok označen jako viditelný a musí se vykreslit. Pro navštívení všech bodů bloku využívám iterátor `VolumeBlockIter`.

Rychlost vybudování indexu závisí na jeho rozlišení, pro rozsáhlé objemy znatelně zpomaluje načítání objemu. Jako optimalizaci jsem navrhl tzv. *charakteristiku přenosové funkce*. Ta udává, které rozsahy hodnot vzorků by byly obarveny neprůhledně. Vzorky se pak při budování indexu nemusí transformovat přenosovou funkcí, ale pouze se porovnají s její charakteristikou. I tak zůstává budování indexu kvůli své nutnosti analyzovat celý objem dat velkou částí načítání nového objemu. Dobrým vylepšením do budoucna by bylo tento proces paralelizovat.

4.5 Rendering

Souřadný systém počítá s rastrem voxelů, kde základní vzdálenost rovin voxelů je jedna jednotka. Tyto vzdálenosti jsou ovlivněny tvarem buněk; pokud je rozměr buňky na ose x 1.5, potom je vzdálenost mezi vrcholy buněk na ose x v souřadném systému 1.5 j.

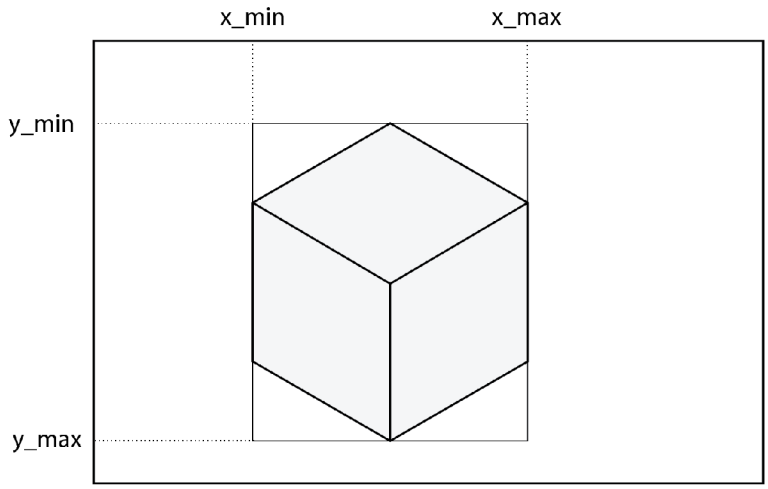
Základní pozice objemů bývá počátek souřadné soustavy.

Kamera

Kamera je implementována po vzoru popisu z kapitoly 2.6. Typ `PerspectiveCamera` popisuje polohu, směr a zorné pole kamery.

Rozhraní kamery umožňuje manipulaci s její polohou a orientací, stejně tak změnu rozlišení a zorného pole. Metoda `get_ray` konstruuje paprsek tak, jak to popisuje rovnice 2.25. Vlastnosti kamery jsou popsány v globálních souřadnicích, proto i paprsek vyprodukovaný kamerou je vyjádřený v globálním souřadném systému.

Kromě vrhání paprsků plní kamera ještě jednu funkci. Vypočítává informace o objemech, jmenovitě vzdálenost objemu od kamery a průmět objemů do homogenních souřadnic viewportu. Za pomoci těchto informací Kompoziční Vlákna určují pořadí renderovacích úkolů.



Obrázek 4.3: Ohraničení (pod)objemu ve viewportu. Hodnoty x_{min}, x_{max}, y_{min} a y_{max} jsou offsety od počátku viewportu v pixelech. Počátek souřadnic je levý horní roh.

Kameru je nutné sdílet mezi vlákny. Toho jsem v původní verzi docílil zámkem `RwLock`, kde mezi renderováním měl právo k zápisu klientský kód, který manipuloval s kamerou. Při renderování měla čtecí přístup všechna pracovní vlákna. Zamykaní kamery při každém snímku se ukázalo jako neefektivní, proto jsem přešel k přístupu, kde je při začátku renderování kamera zkopírována a poslána rendereru.

Osvětlení

Nastřádaná barva pixelu je získána metodou `collect_light`, která je volána pro každý pixel v ohraničení objemu (obrázek 4.3).

Prvním krokem je transformace paprsku do souřadného systému objemu. K tomu je sestrojena transformační matice, která nejprve aplikuje translaci a poté změnu měřítka. Touto maticí je převeden výchozí bod i směr paprsku.

```
// Correct opacity for size of step
let opacity_corrected = 1.0 - (1.0 - color_b.w).powf(step_ratio);

// Accumulate color
rgb += (1.0 - opacity) * opacity_corrected * sample_rgb;
opacity += (1.0 - opacity) * opacity_corrected;
```

Výpis 4.2: Akumulace barvy podle rovnic 2.28, 2.29 a 2.3. `rgb` akumuluje barvu v číslech s plovoucí desetinnou částí, na tříbajtovou hodnotu RGB se převádí až při ukládání do framebufferu. `step_ratio` je poměr referenčního a nového vzorkovacího kroku.

Následuje Phongovo stínování. Stínování je aplikováno podmíněně, a to pouze pokud délka gradientu přesahuje určitou mez. Tím je stínování aplikováno jen na vzorky s velkým rozdílem okolních hodnot, pro které předpokládám, že mohou ležet na hranici objemu.

Rozhraní rendereru

Sekvenční renderer lze instanciovat a provozovat i na hlavním vlákně. To ale nevyhovuje požadavkům na neblokující renderování, a proto ve většině použití uživatelé knihovny renderer spustí na samostatném vlákně. Mohou tak učinit sami, ale knihovna poskytuje systém pro správu rendereru v jiném vlákně včetně mezivláknové komunikace.

Komunikaci s renderovacím vláknem implementuje typ `RendererFront`. Tato struktura drží reference na kameru a sdílený framebuffer, kanály pro komunikaci s rendererem a referenci na renderovací vlákno.

Klientský kód může rendereru posílat zprávy. Definované jsou dva příkazy:

- `StartRendering` – Zahaj renderování snímku. Obsahuje parametry `sample_step` – délku kroku vzorkování a `camera` – kopie kamery.
- `ShutDown` – Ukončovací signál.

Příkazy nejsou přijímány během renderování, staví se do fronty. Klient typicky volá příkaz pro zahájení renderování po pohybu kamery. Po zaslání příkazu k ukončení renderování (`ShutDown`) může klient referenci na vlákno (*thread handle*) použít pro synchronizaci s hlavním vláknem. Od rendereru je přijímán jediný signál, kterým dává najevo, že je dokončeno renderování snímku.

Samotné posílání zpráv je implementováno dvěma jednosměrnými kanály. Kanál funguje jako mezivláknová fronta zpráv s odesilateli a přijímači. Při instanciaci jsou vytvořeny dvě struktury, `Sender` a `Receiver`. Tyto struktury se dají kopírovat, což znamená, že kanál může mít více než jednoho producenta i příjemce zpráv.

Aby byl tento model komunikace rozšiřitelný (i uživateli) o další implementace rendererů, `RendererFront` s renderery komunikuje přes rozhraní `trait RenderThread`. Toto rozhraní slouží především k provázání rendereru a komunikační struktury.

4.6 Sdílení dat mezi pracovními vlákny

Rastrová data jsou při paralelním renderování držena strukturou `Canvas` ve sdílené paměti. `Canvas` obsahuje informace o rozměrech plátna a vektor dlaždic. Dlaždice má dva buffery pro střádání barvy a průhlednosti, informaci o své pozici v plátně a frontu objemů, které jsou v dlaždici viditelné.

Tato fronta musí být před každým snímkem pro každou dlaždici vybudována. Tuto práci vykonává řídící vlákno metodou `build_queues` předtím, než probudí KV a RV.

Přehled metody `build_queues`:

1. Vyfiltrování podobjemů, které nejsou se současnou přenosovou funkcí viditelné.
2. Seřazení podobjemů podle vzdálenosti od kamery.
3. Projekce podobjemů do rastru kamery.
4. Zjištění dlaždic, do kterých každý podobjem zasahuje (`get_affected_tiles`).
5. Přidání podobjemu do front všech dlaždic, do kterých zasahuje

Původní implementace spočívala v kopírování dat mezi vlákny a posílání upravených kopii zpět. V novém řešení je sdílen ukazatel na dlaždici, a to bez uzamykání. Právo k přístupu k dlaždici je konceptuálně předáváno frontami úkolů a výsledků renderování.

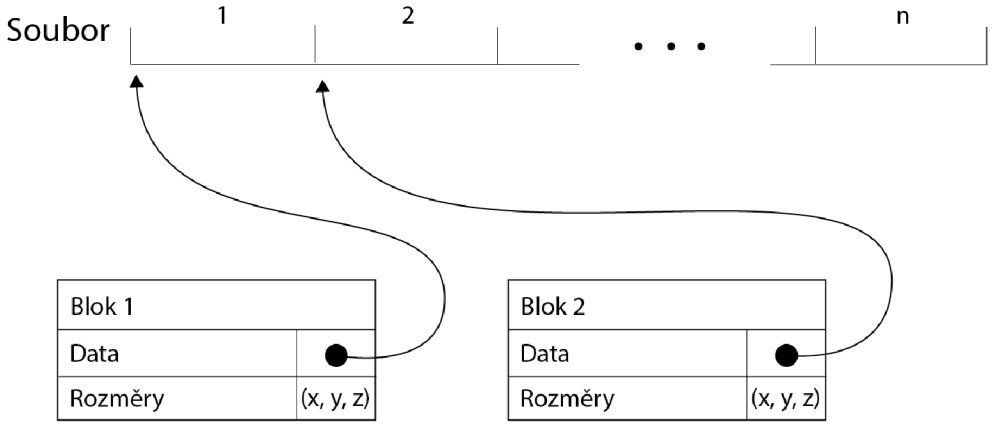
Kromě sdílené paměti jsou veškeré ostatní informace posílány *kanály*. Kanál je struktura pro jednosměrnou komunikaci. Jeho inicializace produkuje dva objekty, příjemce a odesilatele. Kanály jsou typu *mpmc* (Multi-producer multi-consumer), takže jedním kanálem může komunikovat n entit. Pomocí kanálů jsou realizovány i fronty úkolů a výsledků.

Všechny dlaždice jsou na začátku renderování rozděleny mezi KV. KV z fronty dlaždice odebere referenci na blok a spolu s referencí na dlaždici je pošle do fronty renderovacích úkolů. Tím vlastnictví dlaždice předává RV, které úkol dostane. Pokud je fronta prázdná, sníží se počítadlo zbývajících dlaždic. Toto počítadlo je implementováno atomickým typem `AtomicU32`. Pokud klesne na nulu, znamená to, že všechny dlaždice jsou vyrenderovány a renderování snímku je u konce.

Renderovací vlákno provede renderování jednoho bloku v dlaždici a výsledky renderování průběžně vpisuje přímo do bufferů dlaždice. Po dokončení úkolu pošle zprávu do fronty hotových úkolů, čímž předává vlastnictví dlaždice zpět některému z KV. Práce s bloky spočívá pouze ve čtení, a proto není nutné přístup k nim synchronizovat.

Implementace v Rustu vyžaduje `unsafe` bloky kódu, protože není možné staticky ověřit, jestli v dlaždicích nedochází k souběhu přístupů. Dlaždice jsou proto obaleny v typu `Unsafecell`, který ruší garanci jazyka o neměnnosti paměti (*interior mutability*). Každý přístup k dlaždici potom musí být uvnitř `unsafe` bloku.

Čisté ukazatele také nemohou být z důvodů bezpečnosti paměti posílány mezi vlákny a musí se manuálně povolit.



Obrázek 4.4: Blokový objem s mapováním souboru do paměti. Blok obsahuje ukazatel do mapované paměti.

4.7 Optimalizace renderování

Relevantní část canvasu

Protože hranice objemu jsou známy, kamera může zjistit nejmenší výsek viewportu, ve kterém je celý objem vidět. Obdelníkový výsek viz obrázek 4.3. Vysílat paprsky mimo tento obdelník nemá smysl, protože nemohou objem protnout. Tím se ušetří mnoho výpočtů průniku.

Early Ray Termination

Implementace ERT spočívá v porovnávání nastřádané průhlednosti v průběhu sbírání vzorku podél paprsku s předem stanovenou hranicí. Jako hranici pro ukončení střádání jsem stanovil $0,99$.

Empty Space Skipping

Optimalizace ESS je implementována dvěma způsoby. Pro objemy se vzorky po řezech je vybudována akcelerační struktura `EmptyIndex`. Informace o viditelnosti určitého bodu je zjištěna přes rozhraní `Volume`, viz kód 4.3.

Pro blokové objemy je u každého bloku uchována informace o jeho viditelnosti. Neviditelné bloky nejsou uvažovány při stavění fronty úkolů dlaždic.

```
// Empty space skipping
if self.render_options.empty_space_skipping && self.volume.is_empty(pos) {
    pos += step;
    continue;
}
```

Výpis 4.3: Zjištění informace o průhlednosti přes rozhraní `Volume`. Pokud je vzorkovací bod v průhledné oblasti, nevzorkuje se.

4.8 Demo aplikace

Grafické rozhraní je implementováno knihovnou Slint. GUI je popsáno značkovacím jazykem Slint. K události uživatelského rozhraní je možné zaregistrovat funkci, která bude v reakci na danou událost vyvolána z hlavní smyčky (*callback*).

Jádrem aplikace je struktura `State`, která uchovává veškerý stav a logiku. Stav se dá členit do dvou oborů, data uživatelského rozhraní a data o renderování. Data o uživatelském rozhraní zahrnují stav vstupů.

Parametry a události jsou v Rustu přístupné přes automaticky generované metody. Tyto metody potřebují referenci na stav aplikace, aby jej mohly pozměnit. Řešením byla struktura `Rc<RefCell<State>>`, která umožňuje vícenásobné vlastnictví paměti a dynamickou kontrolu práva k zápisu. Právo k zápisu je vždy uděleno, protože struktura je manipulována pouze z hlavního vlákna, takže nemůže dojít k souběhu přístupů.

Stav renderování je vyčleněn do struktury `RenderState`, která je součástí `State` a mezi její zodpovědnosti patří manipulace s kamerou a řízení rendereru. Ke komunikaci s rendererem využívá `RendererFront` z kapitoly 4.5. Také měří délku renderování (*Frame time*) a tuto informaci zobrazuje v uživatelském rozhraní.

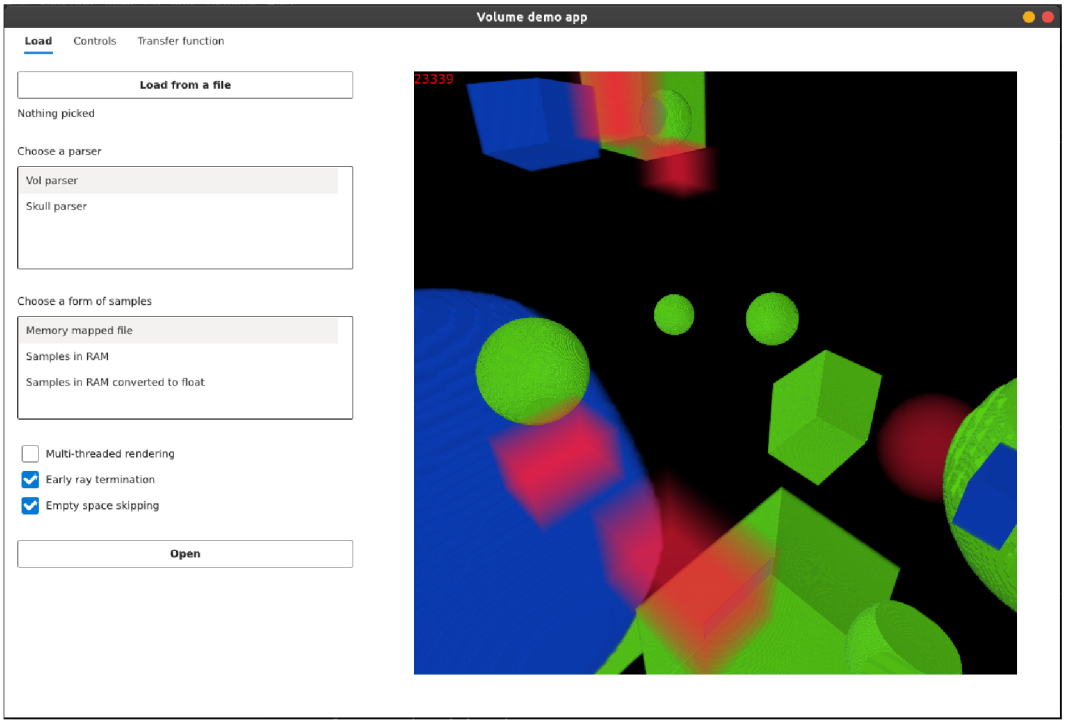
Renderer vždy pracuje v samostatném vlákně, aby neovlivnil odezvu grafického rozhraní. Na zprávu o dokončeném snímku aplikace nejdříve aktivně čekala. To bylo později vylepšeno a nyní na dokončený snímek pasivně čeká zvláštní vlákno, které vyvolá událost v hlavní smyčce.

Původní řešení používalo sdílenou kameru. Protože během aktivního renderování ke sdílené kameře přistupuje renderer, není s ní možné manipulovat. Uživatelské vstupy ale musí být zpracovány. Řešením bylo ukládat povely pro kameru do vyrovnávací fronty a po znovuzískání přístupu ke kameře je ve správném pořadí aplikovat. Časová prodleva, která vznikala zpracováváním těchto příkazů pouze mezi snímky, zhoršovala interaktivitu,

navíc bylo toto řešení poměrně složité. Proto jsem implementoval efektivnější řešení, kde je uchována kopie kamery, manipulace se na kameru aplikují okamžitě a rendereru se v případě potřeby předá kopie kamery v současném stavu.

Výchozí parametry aplikace jsou uloženy v kódu, shromážděny v modulu `defaults`. Mezi tyto parametry patří cesta k objemu, který se zobrazí po spuštění aplikace, výchozí poloha kamery a stav uživatelského rozhraní.

Při pohybu kamery je obraz renderován v nižší kvalitě, aby bylo dosaženo lepší responzivity. Metoda `check_render_conditions` na základě stavu rozhoduje, zda přikázat renderování dalšího snímku a jestli v plné, nebo horší kvalitě. V uživatelském rozhraní se dá tato preference upravit a vynutit vykreslování pouze ve vysoké kvalitě.



Obrázek 4.5: Výsledné grafické rozhraní demo aplikace. Levý sloupec slouží k otevření nového objemu. Záložky skrývají přepínač přenosové funkce.

4.9 Generování volumetrických dat

Parametry z příkazového řádku jsou parsovány knihovnou `clap`. Výsledky jsou dále zpracovány do struktury `Config` představující veškeré nastavení generování.

Dva hlavní moduly implementace jsou `generators` a `orders`. ty definují rozhraní `OrderGenerator` a `SampleGenerator`. Algoritmus vytváření objemu je pak díky těmto rozhraním generický vůči pořadí vzorků i vůči generátoru objemu.

Soubor je otevřen (případně nejdříve vytvořen), je do něj zapsána hlavička a poté sekvence vzorků.

Generování rozsáhlých volumetrických dat může být dlouhý proces, proto se v konzoli objeví zpětná vazba v podobě progress baru. Tu jsem implementoval za pomoci knihovny `indicatif` a rozšířením rozhraní pro generátory pořadí o metodu `get_progress`.

Optimalizace

Prvotní implementace byla velmi pomalá. Měření času odhalilo, že většina času běhu byla trávena čekáním na vstupně-výstupní operace. Důvodem bylo naivní posílaní každého bajtu k zápisu zvlášť funkcí `write`. Implementoval jsem proto optimalizaci dávkování. Nyní generuji najednou velké množství souřadnic, které následně generátor navzorkuje. Až poté je dávka vzorků zapsána do souboru. Jako velikost dávky jsem zvolil 32768 vzorků.

S dávkami potom bylo jednoduché implementovat paralelizaci vzorkování. Knihovna `Rayon` poskytuje funkce pro paralelizaci iterátorů, takže celá paralelizace se stává z jednoho příkazu (4.4).

```
let output_samples: Vec<u8> = batch
            .par_iter()
            .map(|&pos| sample_generator.sample_at(pos))
            .collect();
```

Výpis 4.4: Generování vzorků objemu. Každá pozice z `batch` je paralelně navzorkována a ve správném pořadí uložena do `output_samples`.

Původní implementace generovala objem o rozlišení 256^3 25.57 s, optimalizovaná varianta za pouhých 0.41 s.

Zajímavým rozšířením by byla podpora řídkých souborů. Ty se svou povahou pro tento úkol hodí, protože volumetrická data v praxi obsahují velké množství prázdných (nulových) vzorků.

4.10 Testy, benchmarky a dokumentace

Testy

Jazyk Rust obsahuje podporu pro testování a doporučení, kterými jsem se řídil. Unit testy jsou obsaženy ve stejném souboru, jako kód, který testují, a to v modulu `tests`.

Unit testy využívají pomocné funkce definované v `test_helpers.rs`. Jde o funkce konstruující objemy k testování, čímž redukují opakující se kód.

Integrační testy jsou umístěny ve složce `raycaster_lib/tests`, která je automaticky detekována programem Cargo. Všechny testy proběhnou zadáním příkazu `cargo test`.

Cargo také překládá a spouští všechny příklady z dokumentace.

Benchmarky

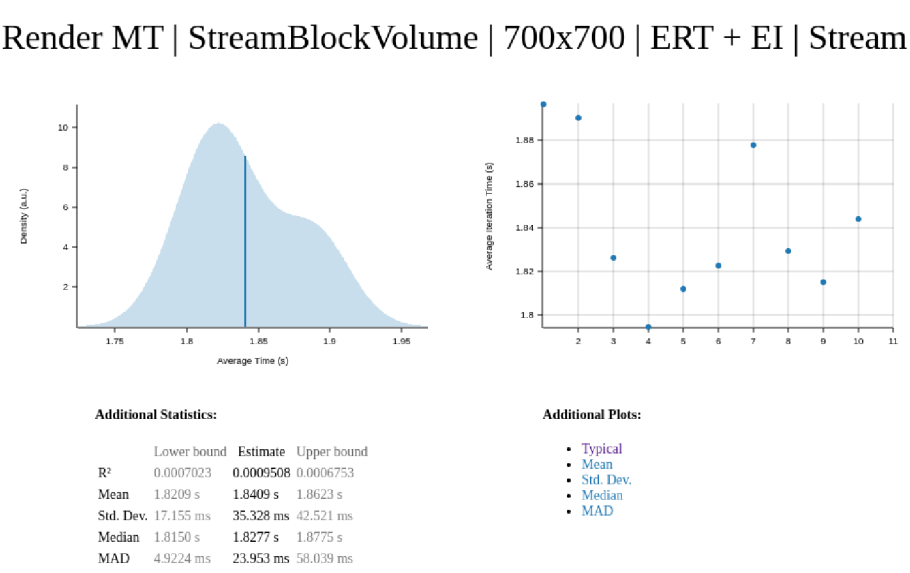
Cargo má základní podporu pro měření výkonu. Pro účely práce jsem se rozhodl pro rozvinutější testovací framework *Criterion*.

Benchmarky knihovny jsou umístěn ve složce `raycaster_lib/benches`. Pro minimální opakování kódu je jeden test definován instancí struktury `BenchOptions`. Ta zahrnuje předvolby renderování, pozice kamery, a typ objemu.

Výstupem měření je detailní statistické zpracování v html, které je generováno do umístění `target/criterion/report/index.html`. Příklad z výstupu je na obrázku 4.6.

Dokumentace

Dokumentace ke knihovně i k přidruženým aplikacím je generována nástrojem `rustdoc`. Tento nástroj čte dokumentační komentáře a převádí je do formy dokumentace v html.



Obrázek 4.6: Ukázka z výstupu měření. Report generovaný frameworkem `Criterion-rs`.

Formát dokumentačních komentářů je založen na *Markdown*. Výstup je generován do složky `target/doc`.

Výchozí nastavení generuje dokumentaci i ke všem použitým balíčkům.

Kapitola 5

Testování a vyhodnocení výsledků

Během vývoje byla aplikace testována a profilována za použití nástrojů *criterion-rs* a *cargo flamegraph*.

Testy proběhly na počítači s následujícími parametry:

CPU:	AMD Ryzen 5 3600
RAM:	16 GB DDR4
OS:	20.04 LTS
Disk:	Samsung 970 EVO 500GB

V tabulkách budu používat zkratky optimalizací (ERT a ESS). Termínem *Stream* v tabulkách myslím mapování souborů do virtuální paměti.

Objemy byly vzorkovány krokem 0.2, který jsem v sekci 5.5 vyhodnotil jako dobrý kompromis mezi kvalitou a rychlostí vykreslování.

5.1 Testovací dataset

Pro testovací účely vznikl generátor volumetrických dat. Pomocí něj jsem vytvořil 4 objemy o dvou rozlišeních a dvou typech uložení vzorků. Na těchto objemech budu testovat rychlost vykreslování. Rozlišení jsou 800^3 a 2000^3. Typy uložení jsou po řezech a po blocích.

Objemy obsahují desítky objektů, z nichž některé jsou téměř průhledné a některé neprůhledné. Detaily o objemech i s příkazy k jejich vygenerování jsou přiloženy na datovém médiu v souboru `README.md`.

5.2 Efektivita optimalizací

Účinnosti optimalizací algoritmu jsem vyhodnotil ze dvou úhlů pohledu – jejich dopad na počet vzorků a rychlost renderování.

Sada obrázků 5.1 vizuálně porovnává dopad optimalizací na počet vzorků. Intuitivně lze pozorovat, že dochází k významnému snížení počtu potřebných vzorků.

Tabulka 5.2 sleduje účinky optimalizací na celkový čas renderování snímku. Tato měření proběhla v jednovláknovém režimu. Jako významnější optimalizace se jednoznačně jeví *Empty space skipping* (ESS).

Obrázek	Metoda	Počet vzorků	Redukce
5.1 a)	Bez optimalizací	67 793 820	-
5.1 b)	ERT	41 476 827	38.82 %
5.1 c)	ESS	11 173 501	83.52 %
5.1 d)	ERT + ESS	2 284 194	96.63 %

Tabulka 5.1: Doprovodná tabulka k obrázku 5.1. Objem je neprůhledný a o rozlišení $68x256x256$. Vzorkovací krok 0.2. Redukce počtu vzorků v posledním sloupci je relativní vůči vzorkování bez optimalizací.

Metoda	Paměť	Průměrný čas	Směrodatná odchylka	Zrychlení
Bez optimalizací	RAM	22.681 s	121.110 ms	–
	Stream	22.376 s	39.212 ms	–
ERT	RAM	18.552 s	64.115 ms	1.22
	Stream	18.537 s	15.269 ms	1.21
ESS	RAM	2.585 s	2.640 ms	8.78
	Stream	2.622 s	734.26 us	8.53
ERT + ESS	RAM	1.878 s	1.340 ms	12.08
	Stream	1.873 s	652.39 us	11.95

Tabulka 5.2: Srovnání doby renderování podle optimalizací a typu přístupu do paměti. Zrychlení je bráno vůči vykreslení bez optimalizací se stejným typem uložení dat.

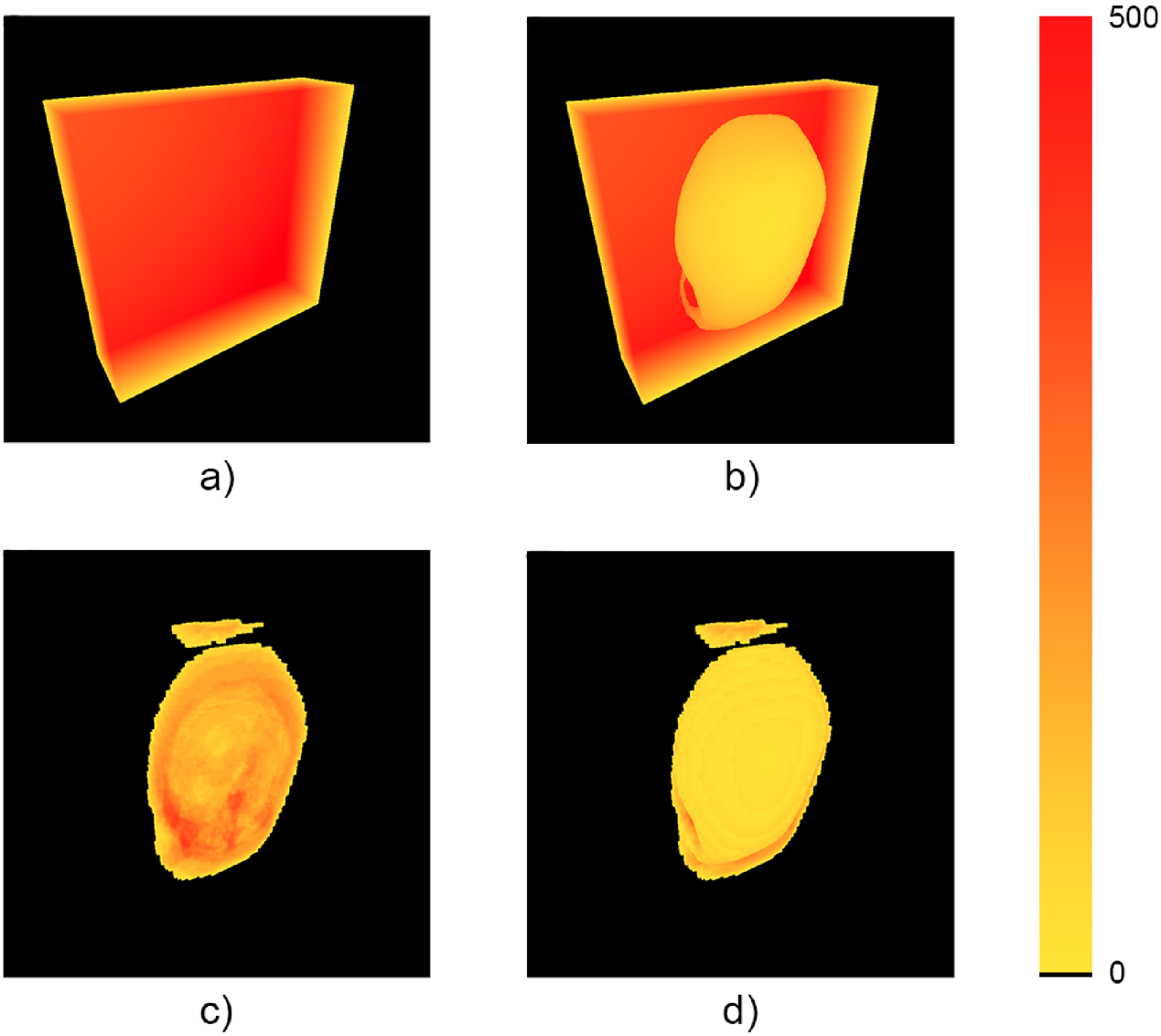
Metoda	Paměť	Průměrný čas	Směrodatná odchylka
Jednovláknová, bez optimalizací	RAM	96.889 s	207.99 ms
Jednovláknová, s optimalizacemi	RAM	6.4599 s	220.36 ms
Vícevláknová, s optimalizacemi	Soubor	255.29 ms	5.5230 ms

Tabulka 5.3: Porovnání rychlosti vykreslování objemu s rozlišením $2k^3$.

5.3 Efektivita přístupu do paměti

Porovnáním rychlosti vykreslování objemu z mapovaného souboru a objemu z RAM zjistíme, že mapované soubory se renderují nepatrně rychleji, ne ale o tolik, aby se nemohlo jednat o šum v datech. Podle měření mají mapované objemy menší rozptyl doby renderování, avšak také natolik malou, aby byla uživateli neznatelná.

Mapování souborů do virtuální paměti se osvědčilo jako lepší řešení, pokud není zapotřebí data předzpracovávat. Rychlostní benchmarky nezachytily nižší dobu načítání objemu. Velkou výhodou je možnost zpracovávání objemů větších, než je dostupná paměť počítače.



Obrázek 5.1: Počty vzorků při použití různých optimalizací. Barva pixelu značí počet vzorků pro daný pixel podle škály vpravo. Detaily viz tabulka 5.1.

5.4 Cena převodu vzorků

V rámci předzpracování je možné vzorky v paměti převést na čísla v plovoucí řádové čárce. Tato operace je poměrně drahá a jejím přesunutím do fáze předzpracování jsem si sliboval zrychlení vykreslování. Nevýhodou je však násobně vyšší nárok na paměť, proto optimalizace není dobře kompatibilní se zobrazováním rozsáhlých volumetrických dat.

Tabulka 5.4 ukazuje výsledky na objemu 800^3.

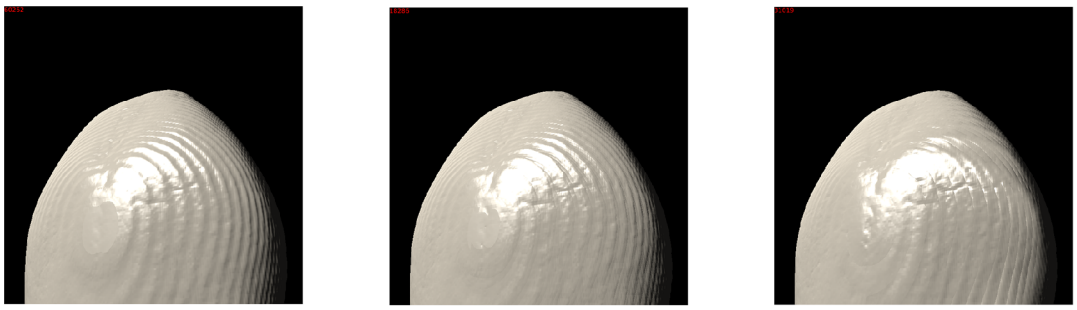
Metoda	Průměrný čas	Směrodatná odchylka	Zrychlení
Bez optimalizací	20.449 s	158.29 ms	1.11
ERT	16.809 s	10.623 ms	1.10
ESS	2.4045 s	1.1559 ms	1.07
ERT + ESS	1.7461 s	503.65 us	1.08

Tabulka 5.4: Renderování objemu se vzorky uloženými jako float32. Zrychlení je relativní vůči odpovídajícímu renderu s jednobajtovými vzorky. Protože je vyžadován preprocessing a data zabírají násobně více paměti, dá se využít pouze pro menší objemy.

5.5 Kvalita vykreslování

Délka vzorkovacího kroku

Některé parametry renderování ovlivňují vizuální kvalitu výsledného obrazu. Jedním z nich je délka vzorkovacího kroku. Obrázek 5.2 ukazuje detail lebky vyrenderovaný různou délkou kroku.



Obrázek 5.2: Kvalita obrazu při různých délkách kroku (`step`). Hodnoty $0,2$ a $0,9$ jsou použity v demo aplikaci a poskytují výsledek porovnatelný s jemnějším krokem.

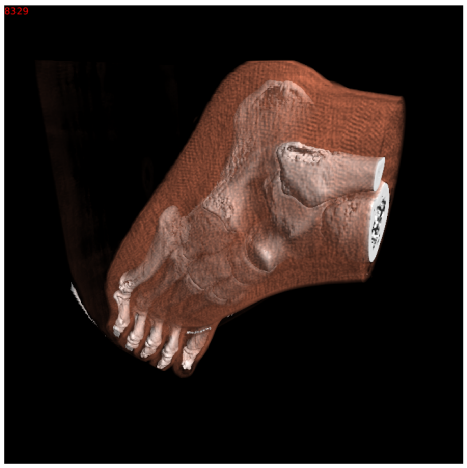
Příklady vyrenderovaných objemů

Obrázek 5.3 ukazuje příklady výsledné výstupy rendereru v rozlišení 700x700.

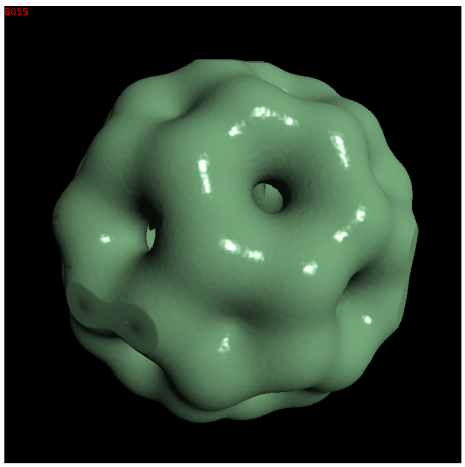
5.6 Experimenty s parametry paralelního algoritmu

Vyhodnocení probíhalo na procesoru s 6 jádry (12 vlákny). Parametry pro paralelní renderer jsem určil specificky pro tento procesor. Automatické určení vhodného počtu pracovních vláken je předmětem případného rozšíření knihovny.

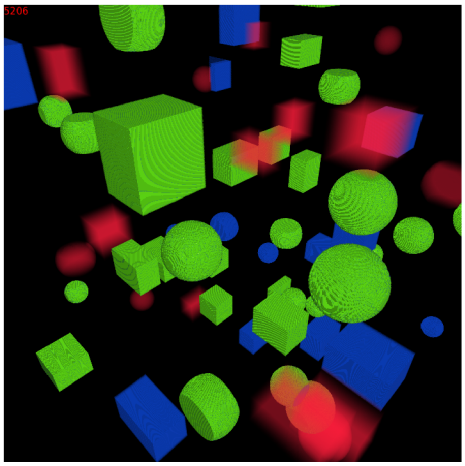
Experimentem bylo zapotřebí určit 4 parametry:



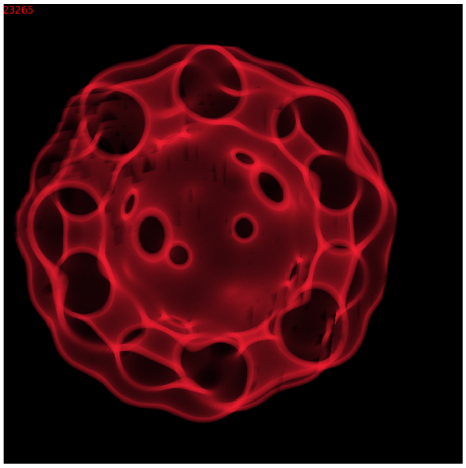
(a)



(b)



(c)



(d)

Obrázek 5.3: Příklady vyrenderovaných modelů. Modely (a), (b) a (d) jsou převzaty z webu UC Davis[1].

1. Velikost podbloků objemu
2. Velikost dlaždic canvasu
3. Počet renderovacích vláken
4. Počet kompozičních vláken

Sledování potvrdilo, že jediné Kompoziční Vlákno (KV) dokáže udržet frontu úkolů naplněnou. Protože KV jinak k rychlosti renderování nepřispívá, renderer používá jedno KV.

Vyzkoušením různých počtů Renderovacích Vláken (RV) jsem došel na jejich optimální množství, a to devět. Toto číslo je opodstatněné, pokud uvážíme, že testovací procesor disponuje dvanácti hardwarovými vlákny, z nichž 3 jsou využity klientskou aplikací (testovacím nebo grafickým rozhraním), řídícím vláknem renderování a kompozičním vláknem.

[1] https://web.cs.ucdavis.edu/~okreylos/PhDStudies/Spring2000/ECS277/DataSets.html

Pro výkon je nežádoucí, aby aktivní vlákna soutěžila o strojový čas, proto nejlépe vychází, když jejich počet v softwaru odpovídá počtu hardwarových vláken. Vhodnou optimalizací by bylo delegovat řízení renderování na KV, ušetřit vlákno a tím umožnit navýšení počtu RV.

Experimentálně jsem poté určil velikost dlaždic na $10x10$ a velikost bloků na 16^3.

Optimalizace úspěšně redukují počty vzorků nutných k vykreslení objemů, a to až o 90 %.

Z testů jako nejrychlejší vyšlo paralelní renderování se všemi optimalizacemi povolenými. Proto bych toto nastavení při používání rendereru doporučil, zejména v situacích, kdy CPU disponuje větším množstvím jader. Objem o rozlišení $2k^3$ zobrazuje rychlostí 255.29 ms za snímek (3,92 FPS).

Převedení vzorků do formátu s plovoucí čárkou získá malé urychlení (asi 10 %), pro rozsáhlé objemy ho ale z důvodů požadavků na paměť nedoporučuji.

Kapitola 6

Závěr

Cílem této práce bylo nastudovat principy a současný stav přímého zobrazování rozsáhlých volumetrických dat, navrhnout volumetrický renderer pro CPU a demo aplikaci. Tyto cíle se mi podařilo splnit.

Pro lepší pochopení problematiky jsem experimentoval s jednoduchou implementací. Na základě získaných znalostí jsem navrhl vlastní renderer. Ten využívá optimalizace Early Ray Termination a Empty Space Skipping, které poskytují více než 10× lepší výkon oproti naivnímu řešení. Renderer také využívá paralelismu na úrovni vláken, čímž bylo podle mého testování dosaženo dalšího skoku ve výkonu.

Demo aplikace zobrazuje rozsáhlá volumetrická data v dobré kvalitě a za udržení plynulosti. Uživatelské rozhraní umožňuje prohlížení objemů a prozkoumání dopadu optimalizací na rychlost vykreslování. Svůj účel demonstrace rendereru splňuje.

Rust se jako jazyk implementace CPU rendereru osvědčil. Výsledná implementace je efektivní a proces vývoje rozhodně ulehčil. Jako nevýhodu jsem považoval mladší ekosystém okolo jazyka, který se projevil obtížnější volbou knihovny pro implementaci grafické demo aplikace. Dovolím si tvrdit, že je pro tento typ úlohy minimálně stejně vhodný jako v této oblasti dominantní C++.

Výsledné řešení jsem otestoval sadou benchmarků na objemech, které jsem vygeneroval. Renderer s optimalizacemi objem o rozlišení $2k^3$ zobrazuje rychlostí 3,92 FPS, což považuji za dostatečné pro interaktivní prohlížení.

V pokračování této práce bych chtěl renderer vylepšit o podporu vícerozměrných a interaktivních přenosových funkcí. Zajímavá by také byla schopnost zobrazit řez objemu.

Literatura

[1] *Cache Memory - Locality of reference*. San Joaquin Delta College, 2021 [cit. 2022-26-04]. Dostupné z: https://eng.libretexts.org/Courses/Delta_College/Operating_System%3A_The_Basics/01%3A_The_Basics_-_An_Overview/1.7_Cache_Memory/1.7.2_Cache_Memory_-_Locality_of_reference.

[2] Acquisto, P. a Groller, E. A Distortion Camera For Ray Tracing. In:. 1993, sv. 5 [cit. 2022-28-03]. DOI: 10.2495/VID930081. Dostupné z: https://www.witpress.com/elibrary/wit-transactions-on-information-and-communication-technologies/5/13648.

[3] Calhoun, P., Kuszyk, B. S., Heath, D. G., Carley, J. C. a Fishman, E. K. Three-dimensional volume rendering of spiral CT data: theory and method. *Radiographics : a review publication of the Radiological Society of North America, Inc.* 1999, 19 3, s. 745–64.

[4] Engel, K., Hadwiger, M., Kniss, J. M. et al. Real-Time Volume Graphics. In:. New York, NY, USA: Association for Computing Machinery, 2004, s. 29–es. SIGGRAPH '04. DOI: 10.1145/1103900.1103929. ISBN 9781450378017. Dostupné z: https://doi.org/10.1145/1103900.1103929.

[5] Hsu, W. Segmented ray casting for data parallel volume rendering. In: *Proceedings of 1993 IEEE Parallel Rendering Symposium*. 1993, s. 7–14 [cit. 2022-14-02]. DOI: 10.1109/PRS.1993.586079. Dostupné z: https://ieeexplore.ieee.org/document/586079.

[6] Hughes, J. F., Dam, A. van et al. *Computer Graphics: Principles and Practice*. Pearson Education, Inc., 2014 [cit. 2022-28-03]. http://www.cs.ucy.ac.cy/courses/EPL426/courses/eBooks/ComputerGraphicsPrinciplesPractice.pdf(visited 2022-14-01).

[7] Ikits, M., Kniss, J., Lefohn, A. a Hansen, C. GPU Gems. 2007, [cit. 2022-28-03]. Dostupné z: https://developer.nvidia.com/gpugems/gpugems/part-vi-beyond-triangles/chapter-39-volume-rendering-techniques.

[8] Kaufman, A. a Mueller, K. Overview of Volume Rendering. In:. Prosinec 2005, sv. 7, s. 127–XI. DOI: 10.1016/B978-012387582-2/50009-5. ISBN 9780123875822.

[9] Kindlmann, G. a Durkin, J. Semi-automatic generation of transfer functions for direct volume rendering. In: *IEEE Symposium on Volume Visualization (Cat. No.989EX300)*. 1998, s. 79–86. DOI: 10.1109/SVV.1998.729588.

[10] Kniss, J., Premoze, S., Hansen, C. et al. A model for volume lighting and modeling. *IEEE Transactions on Visualization and Computer Graphics*. 2003, sv. 9, č. 2, s. 150–162. DOI: 10.1109/TVCG.2003.1196003.

[11] Levoy, M. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*. 1988, sv. 8, č. 3, s. 29–37. DOI: 10.1109/38.511.

[12] Levoy, M. Efficient Ray Tracing of Volume Data. *ACM Trans. Graph.* New York, NY, USA: Association for Computing Machinery. jul 1990, sv. 9, č. 3, s. 245–261. DOI: 10.1145/78964.78965. ISSN 0730-0301. Dostupné z: https://doi.org/10.1145/78964.78965.

[13] Matsui, M., Ino, F. a Hagihara, K. Parallel Volume Rendering with Early Ray Termination for Visualizing Large-Scale Datasets. In:. Prosinec 2004, s. 245–256 [cit. 2022-29-03]. DOI: 10.1007/978-3-540-30566-8_30. ISBN 978-3-540-24128-7.

[14] Max, N. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*. 1995, sv. 1, č. 2, s. 99–108, [cit. 2022-19-02]. DOI: 10.1109/2945.468400. Dostupné z: https://ieeexplore.ieee.org/document/468400.

[15] Ooijen, P. M. A. van, Geuns, R. J. M. van, Rensing, B. J. W. M. et al. Noninvasive Coronary Imaging Using Electron Beam CT: Surface Rendering Versus Volume Rendering. *American Journal of Roentgenology*. 2003, sv. 180, č. 1, s. 223–226, [cit. 2022-28-03]. DOI: 10.2214/ajr.180.1.1800223. PMID: 12490509. Dostupné z: https://doi.org/10.2214/ajr.180.1.1800223.

[16] Phong, B. T. Illumination for Computer Generated Pictures. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. jun 1975, sv. 18, č. 6, s. 311–317, [cit. 2022-28-03]. DOI: 10.1145/360825.360839. ISSN 0001-0782. Dostupné z: https://doi.org/10.1145/360825.360839.

[17] Ray, H., Pfister, H., Silver, D. a Cook, T. Ray casting architectures for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*. 1999, sv. 5, č. 3, s. 210–223. DOI: 10.1109/2945.795213.

[18] Shen, F., Wang, K. et al. Visualization using histogram based transfer functions for 3D cardiac volume data set. In: *2012 IEEE International Conference on Information and Automation*. 2012, s. 977–980 [cit. 2022-28-03]. DOI: 10.1109/ICInfA.2012.6246958. Dostupné z: https://ieeexplore.ieee.org/document/6246958.

[19] Sramek, M. a Dimitrov, L. F3d - A File Format and Tools for Storage and Manipulation of Volumetric Data Sets. In:. Leden 2002, s. 368–371 [cit. 2022-28-03]. DOI: 10.1109/TDPVT.2002.1024085. Dostupné z: https://www.researchgate.net/publication/221625939_f3d_-_A_File_Format_and_Tools_for_Storage_and_Manipulation_of_Volumetric_Data_Sets.

[20] Steneteg, P., Jönsson, D., Falk, M. a Hotz, I. *Volume Raycasting Sampling Revisited*. OSF Preprints, Mar 2020. DOI: 10.31219/osf.io/u9qnz. Dostupné z: osf.io/u9qnz.

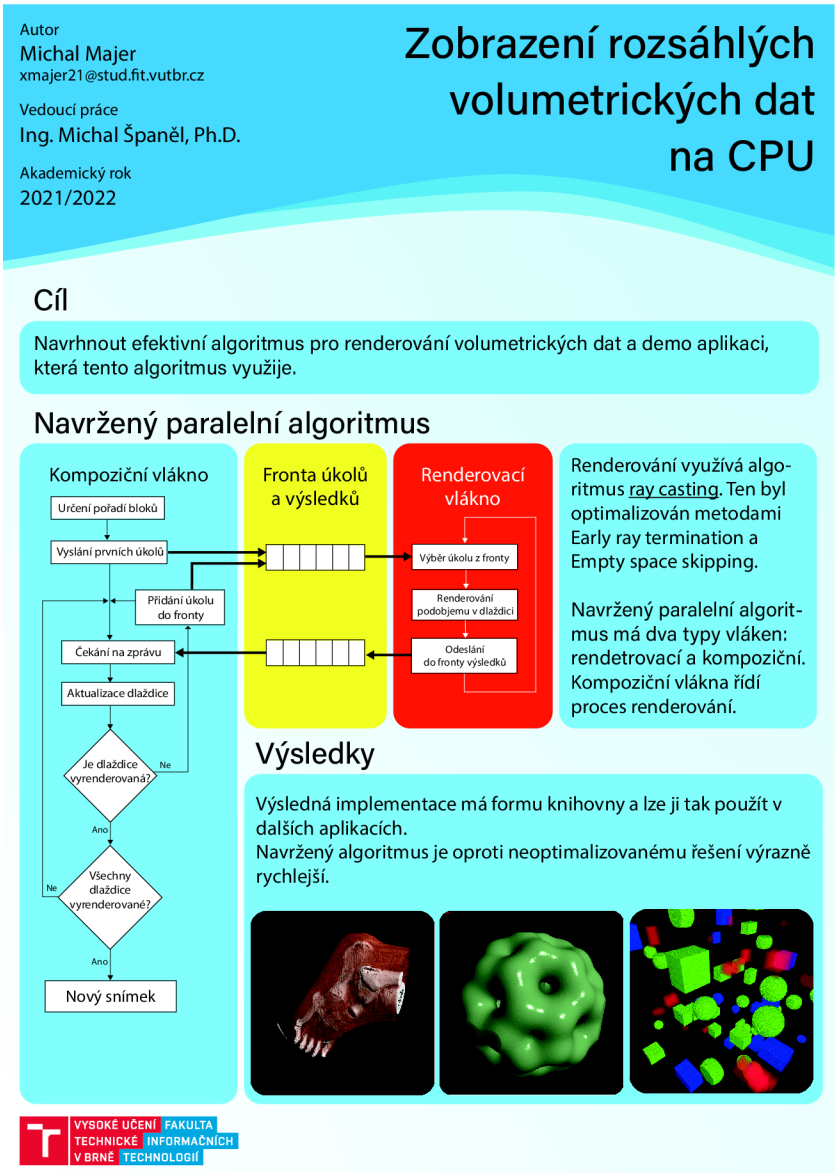
[21] Wald, I., Johnson, G., Amstutz, J. et al. OSPRay - A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics*. 2017, sv. 23, č. 1, s. 931–940, [cit. 2022-28-03]. DOI: 10.1109/TVCG.2016.2599041. Dostupné z: https://ieeexplore.ieee.org/document/7539599.

[22] Williams, A., Barrus, S., Keith, R. a Shirley, M. P. An efficient and robust ray-box intersection algorithm. *Journal of Graphics Tools*. 2003, sv. 10, s. 54.

[23] Zhang, C., Yin, H. a Xiao, S. Adaptive Sampling for GPU-Based 3-D Volume Rendering. In: *Proceedings of the 2nd International Symposium on Image Computing and Digital Medicine*. New York, NY, USA: Association for Computing Machinery, 2018, s. 27–31. ISICDM 2018. DOI: 10.1145/3285996.3286002. ISBN 9781450365338. Dostupné z: https://doi.org/10.1145/3285996.3286002.

Příloha A

Plakát



Příloha B

Přehled struktury datového média

`BP_xmajer21.pdf`	Text bakalářské práce
`BP_xmajer21_tisk.pdf`	Text bakalářské práce (verze pro tisk)
`plakat.pdf`	Plakát
`/src`	Adresář všech zdrojových souborů aplikací
`  /raycaster_lib`	Zdroje renderovací knihovny
`    /benches`	Benchmarky
`    /src`	
`    /tests`	
`  /vol_app`	Zdroje demo aplikace
`  /vol_gen`	Zdroje generátoru objemových dat
`  /Cargo.toml`	Manifest projektu
`/bin`	Přeložené spustitelné soubory
`  /vol_app`	
`  /vol_gen`	
`/docs`	Vygenerovaná dokumentace
`  /raycaster_lib`	
`    /index.html`	Vstupní bod dokumentace
`/volumes`	Objemové soubory k zobrazení
`  /Skull.vol`	
`/bp_tex`	Zdrojové soubory textové části BP
`README.md`	Instalační pokyny