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 **JAN SVOBODA**
AUTHOR

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

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

141572

Ústav:	Ústav počítačové grafiky a multimédií (UPGM)
Student:	**Svoboda Jan**
Program:	Informační technologie
Specializace:	Informační technologie
Název:	**Zobrazení rozsáhlých volumetrických dat na CPU**
Kategorie:	Počítačová grafika
Akademický rok:	2022/23

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 nebo video 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.

Při obhajobě semestrální části projektu 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, prof. Dr. Ing.
Datum zadání:	1.11.2022
Termín pro odevzdání:	10.5.2023
Datum schválení:	3.11.2022

Abstrakt

Tato práce se zabývá návrhem a implementací systému, který umožňuje zobrazovat rozsáhlá volumetricka data v reálném čase na CPU běžného počítače. Práce si klade za cíl řešit jak problematiku samotného zobrazování, kdy tato data často nelze celá umístit do operační paměti stroje, tak i problematiku úložiště těchto dat, kdy v případě rozsáhlých datasetů může být jejich uchovávání v úložišti cílového počítače nežádoucí. Navržené řešení sestává ze dvou aplikací, klientské a serverové. Serverová část slouží jako vzdálené úložiště volumetrických dat, která jsou po malých blocích a v různých kvalitách poskytována klientské aplikaci. Klientská aplikace tato data zobrazuje metodou vrhání paprsků a dle vytvořených strategií řeší načítání a uchovávání potřebných bloků v lokální paměti. Při implementaci klientské aplikace byl kladen důraz na paralelizaci klíčových procesů pro dosažení vysokého výkonu. Výsledný systém umožňuje uživateli zobrazovat rozsáhlé datasety uložené na serverovém úložišti a provádět jejich správu pomocí jednoduchého grafického uživatelského rozhraní.

Abstract

This thesis deals with design and implementation of a system that allows displaying large volumetric data in real time on the CPU of a conventional computer. The thesis aims to solve two biggest problems. Firstly, it aims to solve the problem with rendering itself, where this amount of data often cannot be placed into the main memory of a target computer. Secondly, it aims to solve the problem of storing of this data, where, in the case of large datasets, storing them in the storage of a target computer may not be desirable. The proposed solution contains two applications – the server one and the client one. The server part is used as a remote storage of volumetric data that is provided to the client application in small blocks and in different qualities. The client application renders this data by the ray casting method and, according to the created strategies, performs loading and storing of required blocks in the local memory. In order to achieve high performance, the client application was implemented with an emphasis on parallelization of the main processes. The resulting system allows a user to display large datasets stored on a server's storage and to manage the datasets using a simple graphical user interface.

Klíčová slova

volumetrická data, zobrazování, klient-server, jednotková voxelová mřížka, vrhání paprků, paralelizace, rozdělení do bloků, rozsáhlá data

Keywords

volumetric data, rendering, client-server, unit-sized voxel grid, ray casting, parallelization, block division, large data

Citace

SVOBODA, Jan. *Zobrazení rozsáhlých volumetrických dat na CPU*. Brno, 2023. 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.

.....
Jan Svoboda
4. května 2023

Poděkování

Tímto bych rád poděkoval panu Ing. Michalu Španělovi, Ph.D. za skvělé vedení této práce a za poskytnutí cenných odborných informací.

Obsah

1 Úvod **2**

2 Problematika rozsáhlých volumetrických dat **3**
2.1 Rozsah volumetrických dat a jejich paměťové nároky 4
2.2 Metody zobrazování volumetrických dat 4
2.3 Metody serializace volumetrických dat 9

3 Koncept systému pro zobrazování rozsáhlých volumetrických dat **11**
3.1 Příklady existujících klient-server přístupů 11
3.2 Koncept vlastního řešení 12

4 Návrh datového serveru **14**
4.1 Příprava a dekompozice dat 14
4.2 Ukládání dat a metadat 16
4.3 Webové API 16
4.4 Výsledný návrh 18

5 Návrh klientské aplikace **19**
5.1 Získávání dat prostřednictvím webového API 19
5.2 Paměť dat 19
5.3 Zobrazování metodou vrhání paprsků 23
5.4 Výsledný návrh 27

6 Implementace **29**
6.1 Implementace datového serveru 29
6.2 Implementace klientské aplikace 33

7 Testování a vyhodnocení **39**
7.1 Testovací datasety 39
7.2 Vyhodnocení vlivu velikosti bloku 40
7.3 Vyhodnocení vlivu počtu zobrazovacích vláken 43
7.4 Účinnost strategií pro načítání dat a uvolňování paměti 45

8 Závěr **47**

Literatura **48**

A Obsah paměťového média **51**

Kapitola 1

Úvod

Volumetrická data jsou využívána napříč širokou škálou technických oborů, ve kterých mají nejrůznější uplatnění. Nejčastěji se lze setkat s volumetrickými daty, která reprezentují výstupy zobrazovacích metod ve vědě a lékařství, jako jsou například počítačová tomografie, magnetická rezonance nebo kryoelektronová mikroskopie, dále jsou využívány ve vědeckých simulacích. Své uplatnění taktéž nalézají ve filmovém i herním průmyslu.

S tím, jak se s technickým pokrokem zvyšuje kvalita zobrazovacích metod a simulací, úměrně také roste rozsah jimi produkovaných volumetrických dat, což zvyšuje náročnost jejich zobrazování. Dnes tak není vzácností setkat se s daty překračujícími velikost jednoho terabajtu, což naráží jak na problémy samotného zobrazení, tak i na kapacitu úložiště cílového počítače. Z těchto důvodu je stále oblíbenější využití klient-server architektur, kdy server může sloužit buď jako pouhé vzdálené úložiště volumetrických dat, nebo může vykonávat i samotné zobrazováni.

Tato práce se zabývá návrhem a implementací systému, který umožňuje zobrazování takto rozsáhlých volumetrických dat na běžném počítači v reálném čase. Systém sestává z klientské a serverové části, kdy server slouží jako úložiště volumetrických dat sdílené několika klientskými aplikacemi. Klientská aplikace pak získaná data zobrazuje metodou vrhání paprsků na CPU. Tato architektura značně zjednodušuje manipulaci s rozsáhlými datasety a snižuje nároky na úložiště klientského počítače, avšak za předpokladu propojení pomocí vysokorychlostní internetové sítě.

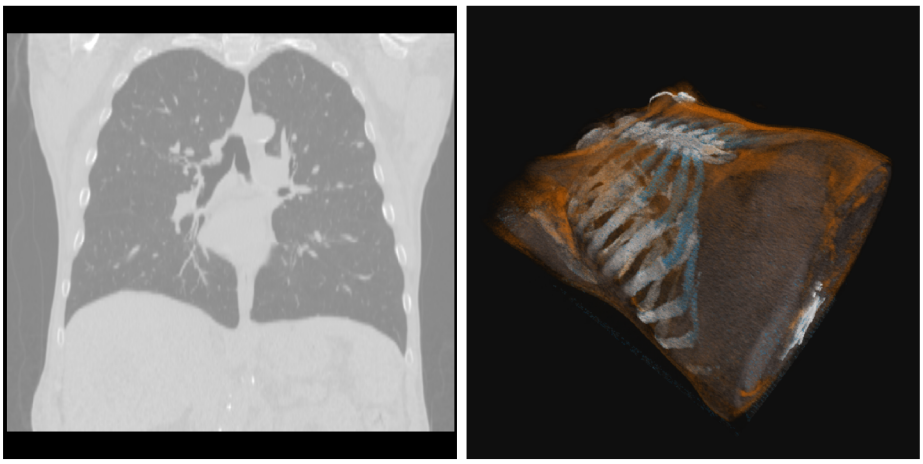
V kapitole 2 je vysvětlena problematika související se zobrazováním rozsáhlých volumetrických dat. Na základě těchto poznatků a již existujících řešení je poté v kapitole 3 představen koncept systému, který umožní zobrazování rozsáhlých volumetrických dat na *CPU* za použití klient-server architektury. Kapitoly 4 a 5 se věnují detailnímu návrhu datového serveru a klientské aplikace pro zobrazování rozsáhlých volumetrických dat. Implementace těchto dvou částí je vysvětlena v kapitolách 6.1, 6.2, kdy výsledky a měření této implementace jsou zhodnoceny v rámci kapitoly 7.

Kapitola 2

Problematika rozsáhlých volumetrických dat

Volumetrická data jsou nejčastěji chápána jako množina vzorků, které reprezentují hodnoty určité veličiny ve 3D prostoru. Obecně mohou být vzorky získávány v náhodných bodech tohoto prostoru, ale ve většině případů se jedná o vzorky získávané po konstantních intervalech podél tří ortogonálních os [16]. Samotný vzorek může mít obecně různou povahu v závislosti na dané veličině, nejčastěji se však jedná o jedno číslo (například úroveň propustnosti určité vlnové délky elektromagnetického záření), méně častěji se můžeme setkat s daty, kde jednotlivé vzorky reprezentují barvu. V určitých případech mohou být volumetrická data závislá i na čase, v těchto případech tedy data mají čtyřrozměrnou povahu [16]. Jednotlivé vzorky volumetrických dat se nazývají voxely.

Volumetrická data mohou být taktéž chápána jako sekvence dvourozměrných snímků, a někdy je k nim i takto přistupováno při jejich zobrazování. Zobrazování dat po jednotlivých snímcích je mnohem méně výpočetně náročné oproti trojrozměrnému zobrazení. Trojrozměrné zobrazení oproti tomu umožňuje zobrazit větší množství informace současně, což však nemusí být vždy žádoucí. V praxi se využívá obou těchto přístupů. Jednotlivé typy zobrazení stejného datasetu znázorňují obrázky 2.1.



(a) Zobrazení volumetrických dat po snímcích.

(b) Trojrozměrné zobrazení volumetrických dat.

Obrázek 2.1: Různé typy zobrazení datasetu *CTChest* (dataset dostupný z [1, 6]).

2.1 Rozsah volumetrických dat a jejich paměťové nároky

Kvůli trojrozměrné povaze volumetrických dat často dochází k situacím, kdy data v surové podobě mohou vyžadovat značné množství paměti úložiště. V praxi se tak běžně vyskytují i datasety s paměťovou velikostí stovek gigabajtů, lze se však setkat i s daty výrazně rozsáhlejšími. To způsobuje problémy jak se samotnou manipulací s daty, kdy úložiště běžného osobního počítače často není schopné pojmout ani jeden takto rozsáhlý dataset, tak i se zobrazováním těchto dat, kdy často nepřipadá v úvahu umístění celého datasetu do operační ani grafické paměti. Je proto třeba hledat chytřejší přístupy a návrhy, které umožní zobrazení i takto rozsáhlých dat na běžném počítači.

Při zobrazování rozsáhlého datasetu většinou nedochází k situacím, kdy by bylo nutné zobrazit dataset celý najednou a v plné kvalitě. Displej počítače má konečné rozlišení a ne vždy jsou viditelné všechny části volumetrické scény. Při perspektivní projekci se navíc úměrně v závislosti na vzdálenosti od kamery objekty opticky zmenšují, což poskytuje další prostor k optimalizacím. Pro daný pohled tedy není většinou nutné disponovat kompletními daty, ale postačí zde data značně menší. Tyto skutečnosti tedy umožňují při správném návrhu umístit do operační, nebo grafické paměti všechny potřebné části scény v dostatečném rozlišení, nebo alespoň jejich potřebnou část.

Při řešení problematiky ukládání volumetrických dat je však situace komplikovanější, jelikož data musí být uchovávána celá. Tato situace se dá v některých případech řešit kompresí těchto dat, ať už ztrátovou, nebo bezztrátovou. Pro bezztrátovou kompresi lze využít běžně používaných formátu, jako jsou například *ZIP*, *GZIP* a další. Pro ztrátovou kompresi lze vyžít kompresních metod využívaných pro dvourozměrné snímky, což je ale méně efektivní, než komprese pracující nad 3D objemem. Pro kompresi nad 3D objemem je možné využít například trojrozměrnou vlnkovou transformaci [18]. Jelikož volumetrická data lze také chápat jako sekvenci snímků, lze provést taktéž kompresi pomocí běžných standardů pro kompresi videa [19]. Ty navíc většinou podporují hardwarovou akceleraci na grafických kartách. Ne vždy je však ztrátová komprese tolerována, jelikož narušuje autentičnost samotných dat. Uvažujme například situaci, kdy by příliš vysoká úroveň ztrátové komprese způsobila ztrátu informace o malém nádoru při lékařském vyšetření.

Pokud dostupný prostor úložiště cílového počítače neumožňuje uchovávat celý dataset, musí být situace řešena umístěním úložiště vně tohoto počítač. To lze řešit různými přístupy, například připojením externích diskových jednotek, nebo použitím klient-server architektury. Obě zmíněné metody mají své výhody i svá negativa. Použití externích disků může poskytnout rychlejší čtení dat a možnost s daty pracovat bez připojení k internetu, avšak za cenu složitější manipulace. Klient-server architektura oproti tomu poskytuje jednodušší manipulaci (data není nutné mít fyzicky u sebe) a možnosti sdíleného přístupu, avšak za podmínky kvalitního propojení obou zařízení přes internetovou síť. Server zde může sloužit buď pouze jako sdílené úložiště volumetrických dat, nebo může realizovat i samotné zobrazování.

2.2 Metody zobrazování volumetrických dat

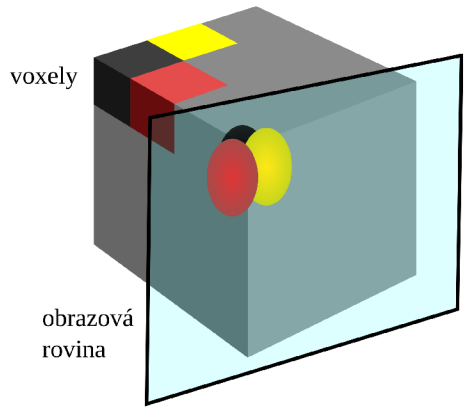
Ve většině případů samotná volumetrická data přímo nereprezentují grafickou informaci, ale pouze hodnoty nějakého měření. Je tedy většinou nutné provést mapování hodnot jednotlivých voxelů na hodnoty, které je možné danou metodou zobrazit. Nejjednodušší je zobrazení dat po snímcích, které v praxi nalézá velké uplatnění. Jelikož data jsou uložena často právě jako sekvence snímků, je nejjednodušší zobrazit data právě takto. Touto meto-

dou se však zobrazují i libovolné řezy, které nemusí korespondovat s fyzickým uložením dat v paměti. Nespornou výhodou je nízká velikost současně zobrazovaných dat.

Trojrozměrné zobrazování je oproti tomu značně výpočetně náročnější. Existuje mnoho metod pro zobrazování volumetrických dat, které je obecně možné rozdělit tří skupin. První skupinou jsou tzv. *object-order* metody, které fungují na principu přímého mapování volumetrických dat na obrazovou rovinu [16]. Do těchto metod patří například *voxel splatting*. Další skupinou jsou metody typu *image-order*, kdy tyto metody pracují na principu paprsků vrhaných jednotlivými pixely skrze volumetrická data [16]. Jedná se například o hojně využívanou metodu *ray casting* (vrhání paprsků). Poslední metodou jsou metody řadící se do skupiny *domain-based*, kdy volumetrická data jsou transformována do alternativní domény, která je poté zobrazována [16]. V praxi se taktéž vyskytují i metody, které využívají různé kombinace těchto přístupů [16].

Voxel splatting

Metoda je založena na přímém zobrazování jednotlivých voxelů, které jsou promítány na obrazovku buď v pořadí od nejvzdálenějších po nejbližší, nebo v některých implementacích od nejbližších po nejvzdálenější. Nejčastěji je takto promítnut pouze střed daného voxelu, a samotný voxel je poté rasterizován jako dvourozměrný obrazec o určitém rozměru. Největším problémem je nutnost řazení jednotlivých voxelů tak, aby mohly být promítány ve správném pořadí [20]. Pro řešení tohoto problému existují různé přístupy a optimalizace, které jsou také často závislé na typu projekce, kdy nejjednodušší na optimalizace je projekce ortogonální. Taktéž musí být vhodně zvoleny parametry vykreslovaného obrazce, aby nedocházelo k příliš velkým mezerám, nebo překryvům [14]. Nevýhodou voxel splattingu (stejně jako jiných object-oreder metod) je, že nelze snadno zabránit vykreslování voxelů, které budou v konečném snímku překryty jinými voxely [14]. Obecný princip metody znázorňuje obrázek 2.2.



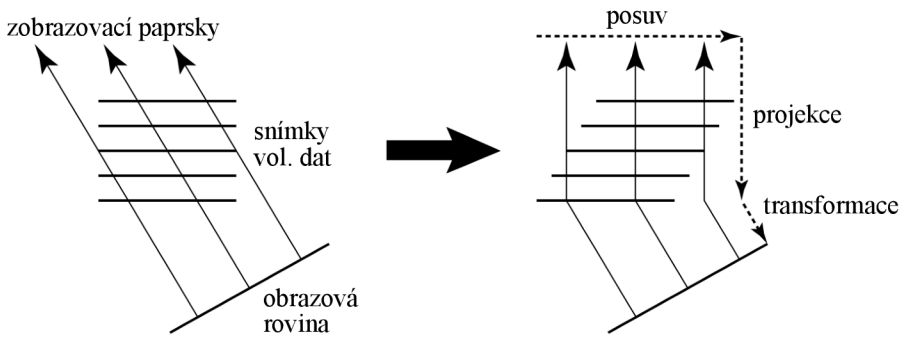
Obrázek 2.2: Metoda voxel splatting zobrazuje volumetrická data promítáním jednotlivých voxelů na obrazovku ve správném pořadí.

Výhodou některých implementací je možnost procházet voxely v pořadí, v jakém jsou uloženy fyzicky v paměti, avšak při určitých změnách pohledu musí být data v paměti přeřazena, nebo musí být uložena duplicitně v několika variantách řazení, což nemusí být v závislosti na velikosti dat vždy proveditelné. Například se využívají data uspořádaná do snímků, nejčastěji zarovnanými s osami prostoru [20]. Tyto snímky jsou přímo zobrazovány na obrazovku od nejvzdálenějšího po nejbližší. Vždy jsou zobrazovány snímky podél nejrov-

noběžnější osy se směrem pohledu, kdy pokud se při změně pohybu stane nejrovnoběžnější osou jiná osa, jsou použity snímky podél této osy [20]. To tedy vyžaduje mít k dispozici snímky ve třech různých variantách.

Shear-warp

Metoda pracuje na principu zobrazování dat pomocí vhodně transformovaných snímků. Metoda se snaží eliminovat problém *object-order* metod, kdy složitost mapování z objektového prostoru do prostoru obrazu komplikuje jejich efektivitu [14]. Tato metoda problém řeší transformací volumetrických dat do meziprostoru, který umožňuje efektivní projekci do 2D obrazu [14]. Jedinou podmínkou tohoto prostoru je, že všechny zobrazovací paprsky jsou v něm rovnoběžné se třetí souřadnicovou osou [14]. Nejdříve dochází ke transformaci jednotlivých snímků, poté jsou převzorkovány a sloučeny do mezisnímku ve směru třetí souřadnicové osy a následně dochází k transformaci tohoto mezisnímku do výsledného obrazu [14]. Princip metody pro ortogonální projekci je znázorněn na obrázku 2.3. Výhodou je i mimo jiné možnost implementace předčasného ukončení paprsku [14]. Nevýhodou je, stejně jako u některých implementací metody *voxel splatting*, nutnost mít k dispozici snímky ve třech různých variantách.



Obrázek 2.3: Princip shear-warp metody pro ortogonální projekci. (Převzato a přeloženo z [14].)

Zobrazování založené na texturách

Skupina metod, které využívají 2D, nebo 3D textur pro efektivní zobrazování na grafického akcelerátoru pomocí *image-order* metod. Především využívají grafické akcelerace trilineární interpolace a možnosti vysoké paralelizace [22]. Jejich hlavní nevýhodou je omezení grafických karet na velikost textur, které mohou pojmout.

Metoda vrhání paprsků

Ray casting, nebo také vrhání paprsků, je *image-order* metoda, která pracuje na principu vrhání paprsků skrze jednotlivé obrazové pixely. Podél těchto paprsků je prováděno vzorkování volumetrických dat, kdy hodnoty jednotlivých vzorků jsou agregovány do výsledné barvy. Hlavní nevýhodou metody je, že nepřistupuje k datům v pořadí, v jakém jsou uložená v paměti. Ve výsledku pak metoda tráví více času výpočtem umístění vzorkovacích bodů a prováděním adresovací aritmetiky, než jiné používané metody [14]. Tento problém se dá

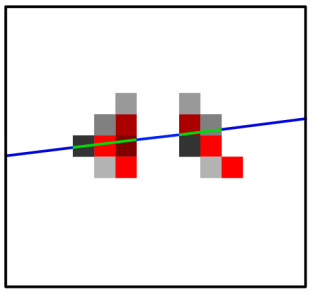
zmírnit využitím metod serializace dat, které splňují podmínku prostorové lokality. Metody serializace jsou popsány v sekci 2.3. Výhodou však je skutečnost, že metoda nevyžaduje data ve více variantách.

Předčasné ukončení paprsku

Jednou z nejčastěji využívaných optimalizačních technik je předčasné ukončení paprsku. Jelikož při agregaci hodnot vzorku dochází často ke stavu, kdy další hodnota již výsledek neovlivní, je zbytečné pokračovat ve vykonávání dalšího vzorkování tohoto paprsku. Implementace této optimalizace je u metody vrhání paprsků triviální v porovnání s *object-order* metodami, ve kterých je její implementace značně komplikovaná a často nevede k jejich vyšší efektivitě [14].

Přeskakování prázdných prostorů

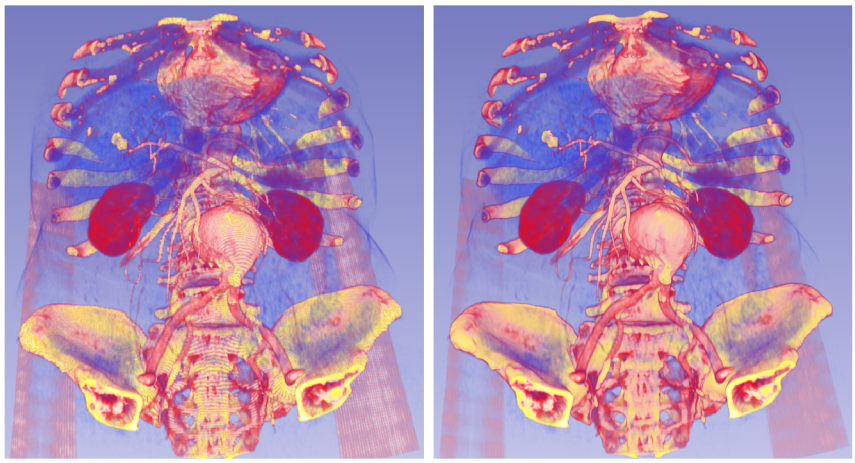
Další možnou optimalizací je přeskakování prázdných prostorů, neboli částí scény, které jsou při daném zobrazení plně průhledné. Situaci ilustruje obrázek 2.4. Existuje množství různých implementací této metody. Některé například fungují na principu rozdělení dat do oktalových stromů, kdy každý uzel obsahuje informaci o tom, zdali jsou data plně průhledná [9]. Tento přístup však není příliš efektivní pro data obsahující tenké struktury [9]. Existují taktéž implementace využívající *k-d stromy*, které s tenkými strukturami pracují lépe, avšak není příliš výhodná pro aplikace, kde dochází k častým změnám viditelnosti, jelikož přizpůsobení *k-d stromu* je poměrně pomalé [9].



Obrázek 2.4: Obrázek znázorňuje paprsek procházející volumetrickými daty, kdy pouze jeho zelená část má vliv na výsledný obraz.

Trilineární interpolace

Při vzorkování volumetrických dat je možné hodnotu vzorku získat buď čtením hodnoty nejbližšího voxelu (metoda nejbližšího souseda), nebo trilineární interpolací okolních voxelů. Druhá možnost poskytuje výrazně plynulejší přechody, avšak za cenu nižšího výkonu, jelikož pro jeden vzorek je potřeba výpočet založený na hodnotách osmi voxelů. Vizuální rozdíl těchto dvou technik znázorňují obrázky 2.5.

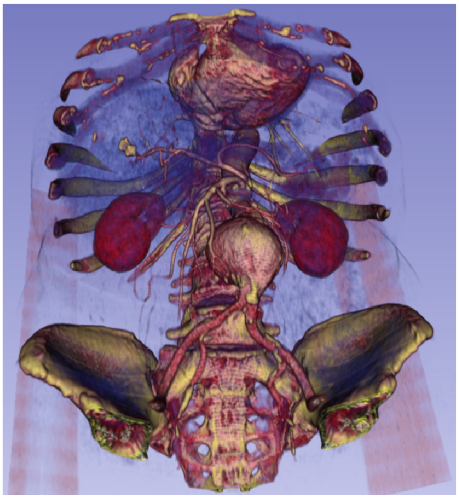


(a) Nejbližší soused. (b) Trilineární interpolace.

Obrázek 2.5: Vliv trilineární interpolace na kvalitu zobrazení. Zobrazeno na testovacím datasetu *CTA abdomen* v programu *3D Slicer* (viz [1, 6]).

Osvětlovací modely

Pro kvalitnější a realističtější zobrazení volumetrických dat může být taktéž využito různých osvětlovacích modelů, které se liší svými výsledky a svou náročností. Jelikož volumetrická data jako taková nereprezentují žádné plochy, je nutné využít jiné techniky. Nejčastěji se pro výpočet světelných efektů využívá metod založených na gradientech [12]. Výpočet gradientu je možné provádět během samotného zobrazování, nebo je možné jeho hodnoty předpočítat. Předpočítané gradienty však mohou zaobírat značné množství paměti. Zobrazení s použitím osvětlovacího modelu je ukázáno na obrázku 2.6.



Obrázek 2.6: Ukázka zobrazení s použitím osvětlovacího modelu a trilineární interpolace. Zobrazeno na testovacím datasetu *CTA abdomen* v programu *3D Slicer* (viz [1, 6]).

2.3 Metody serializace volumetrických dat

Jelikož data jsou v paměti počítače fyzicky uložena jako posloupnost bitů, musí být řešeno, jak provést transformaci volumetrických dat na tuto posloupnost. Je zde tedy prováděno mapování vícerozměrného pole na pole jednorozměrné. Výběr vhodné metody serializace volumetrických dat je velmi důležitý primárně z hlediska optimalizace přístupu do paměti, kdy je snaha zajistit, aby docházelo k co nejlepšímu využití rychlých vyrovnávacích pamětí procesoru, a nedocházelo příliš často k přístupu přímo do operační paměti, která se projevuje mnohonásobně vyšší latencí. Výběr vhodné metody je závislý především na způsobu zobrazování volumetrických dat, jelikož různé metody mohou mít odlišnou povahu přístupu do paměti. Cílem je, aby po sobě čtená data ležela v paměti co nejblíže, musí být však brána v úvahu také složitost samotného přepočtu prostorových souřadnic na pozici v paměti.

Důležitou vlastností specifickou pro některé serializační metody je prostorová lokalita, která popisuje schopnost metody uchovávat prostorově blízká data blízko i ve fyzické paměti. V tomto ohledu vykazují dobré výsledky Hilbertova křivka a Mortonova křivka [4].

Po řádcích

Jelikož volumetrická data lze chápat jako sekvenci dvourozměrných snímků, které jsou nejčastěji serializovány po řádcích, mohou být volumetrická data serializována jako sekvence po řádcích serializovaných snímků. Tato metoda má mnoho výhod, jako je snadné zobrazení dat po vrstvách (sekvence snímků), nebo jednoduchost ukládání dat při jejich generování, jelikož část zdrojů volumetrických dat data získává právě po dvourozměrných snímcích. Z těchto důvodů se jedná o nejběžnější podobu volumetrických dat u většiny jejich běžných formátů. Co se však týče prostorové lokality, dva prostorově sousední vzorky budou v nejhorším možném případě vzdálené o celou vrstvu, tudíž tato metoda nároky na prostorovou lokalitu nesplňuje.

Mapování trojrozměrného pole o rozměrech l_x, l_y, l_z s indexy x, y, z na index jednorozměrného pole při serializaci po řádcích ve směru osy x, je-li dále postupováno ve směru osy y, a poté ve směru osy z, je dáno vztahem 2.1,

$$i = x + y \cdot l_x + z \cdot l_x \cdot l_y \tag{2.1}$$

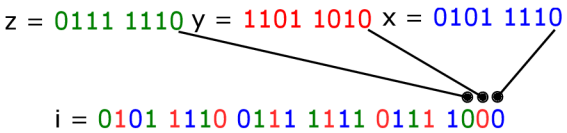
Opačný převod pak lze provést dle vztahů 2.2,

$$\begin{aligned} z &= i \; div \; (l_x \cdot l_y) \\ y &= i \; mod \; (l_x \cdot l_y) \; div \; l_x \\ x &= i \; mod \; (L_x \cdot l_y) \; mod \; l_x \end{aligned} \tag{2.2}$$

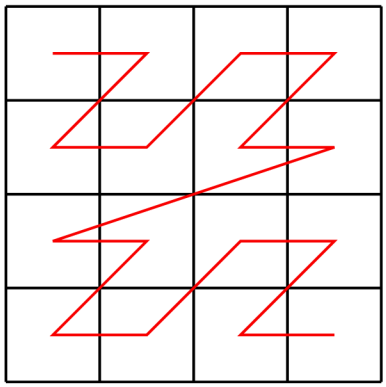
kde *div* je operátor celočíselného dělení, a kde *mod* je operátor zbytku po celočíselném dělení.

Mortonova Z-křivka

Mortonova Z-křivka je jednou z hojně používaných křivek vyplňujících prostor, které disponují vysokou mírou prostorové lokality. Její prostorová lokalita je však nižší než u křivky Hiblertovy [7]. Oproti té však disponuje jednodušším způsobem mapování, který spočívá v prokládání bitů jednotlivých souřadnic, a lze jej tedy implementovat poměrně efektivně. Tento princip je znázorněn na obrázku 2.7. Dvourozměrnou variantu Mortonovy křivky znázorňuje obrázek 2.8.



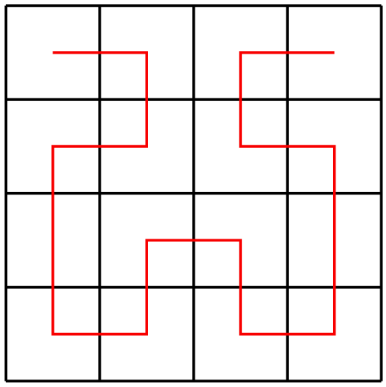
Obrázek 2.7: Mapování indexů trojrozměrného pole na index jednorozměrného pole pomocí Mortonovy křivky. (Binárně)



Obrázek 2.8: Dvourozměrná varianta Mortonovy křivky.

Hilbertova křivka

Samotná Hilbertova křivka byla navržena pouze pro dvourozměrný prostor, avšak existuje několik rozšíření pro prostor vícerozměrný, které se liší mírou prostorové lokality [7]. Jak již bylo zmíněno, tato křivka poskytuje vyšší míru prostorové lokality, avšak její mapování je komplikovanější. I přes tento nedostatek je Hilbertovy křivky v informatice hojně využíváno. Tvar dvourozměrné Hilbertovy křivky znázorňuje obrázek 2.9.



Obrázek 2.9: Dvourozměrná varianta Hilbertovy křivky.

Kapitola 3

Koncept systému pro zobrazování rozsáhlých volumetrických dat

V předešlé kapitole 2 byly zmíněny problémy, které vznikají při potřebě zobrazit rozsáhlá volumetrická data. Z hlediska řešení problematiky úložiště byly nastíněny možnosti použití externích diskových jednotek, nebo využití klient-server architektury. Druhá možnost je sice implementačně složitější, ale oproti první nabízí mnoho výhod. Bude tedy vhodné systém pro zobrazování rozsáhlých volumetrických dat vytvořit právě takto.

Výhodou tohoto návrhu je možnost přístupu více uživatelů k jednomu datasetu současně, prakticky odkudkoli. Při použití tlustého klienta však za podmínky propojení uživatele a serveru pomocí vysokorychlostní internetové sítě. Tuto podmínku lze většinou dobře splnit v rámci lokálních sítí, kde často bývají přenosové rychlosti poměrně vysoké, což umožňuje kvalitní přístup k úložišti alespoň v rámci jednoho pracoviště.

3.1 Příklady existujících klient-server přístupů

Mwalongo et al. [17] využívá klient-server architektury pro zobrazování rozsáhlých volumetrických dat ve webovém prohlížeči. Data jsou zde zobrazována ve webovém prohlížeči klientského zařízení pomocí technologie WebGL 2.0 GPU metodou založenou na 3D texturách. Server, implementovaný pomocí technologie *Node.js*, zde slouží jako úložiště volumetrických dat, která server rozděluje na bloky v různých kvalitách. Typicky jsou zde využívány bloky o velikosti 32^3, nebo 64^3. Server jednotlivé bloky neukládá na disk, ale udržuje je pouze v operační paměti. Výsledný systém vykazuje dobrých výsledků u menších datasetů, u větších již systém nedosahuje příliš vysoké snímkové frekvence (autor uvádí $\sim 2\ hz$ u datasetu 1024^3).

Gutenko et al. [8] představuje vzdálený zobrazovací řetězec pro vizualizaci medicínských dat na mobilních zařízeních, které samotné nedisponují dostatečným zobrazovacím výkonem a u kterých může být problém i se samotným datovým přenosem. Pro řešení těchto problému navrhuje architekturu tenkého klienta, kdy server provádí zobrazování samotných dat a následně výsledný obraz streamuje do klientského zařízení. Práce klade důraz na výběr vhodného hardwarově akcelerovaného kodéru s nízkou latencí, aby bylo možné data zobrazovat interaktivně. Výsledný systém umožňuje interaktivně zobrazovat datasety o velikosti až 2048^3. Umožňuje taktéž vykonávat kvalitnější typy zobrazení, které by na klientském zařízení bylo jen velmi obtížné implementovat.

Campoalegre et al. [3] diskutuje a porovnává různé existující klient-server architektury pro zobrazování volumetrických dat. Zabývá se jak typy realizace pomocí tenkého klienta, kdy samotné zobrazování probíhá přímo na serveru, tak i realizacemi, kde server slouží pouze jako úložiště volumetrických dat. Většina zde zmíněných implementací využívá zobrazování založené na texturách, ale najdou se zde i implementace využívající vrhání paprsků, nebo zobrazování založené na iso-plochách.

Lalgudi et al. [15] se zabývá problematikou komprese obrázků přenášených po síti při zobrazování volumetrických dat založených na klient-server architektuře, kde dochází ke zobrazování dat na serveru pomocí hardwarově akcelerovaných zobrazovacích metod.

Callahan et al. [2] popisuje progresivní metodu zobrazování velmi rozsáhlých neuspořádaných mřížek za pomocí klient-server architektury. Server zde slouží jako úložiště dat a streamuje tato data na klientská zařízením, kde jsou tato data zobrazována.

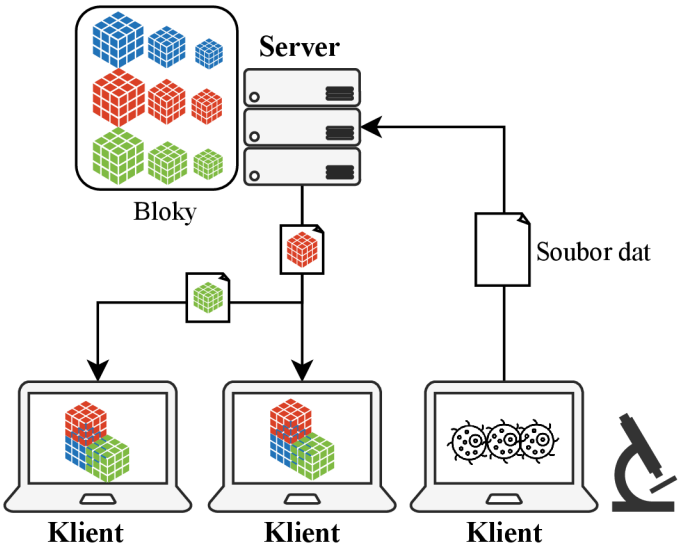
Wald et al. [23] představuje framework *OSPRay* pro sledování paprsků, který umožňuje zobrazovat rozsáhlá polygonální, nepolygonální a volumetrická data na CPU. Tento framework cílí na vysoký výkon a na snadnou použitelnost v rámci jiných projektů. Mohl by být tedy využit v i rámci klient-server řešení pro zobrazování rozsáhlých volumetrických dat.

3.2 Koncept vlastního řešení

Jelikož řešení cílí na zařízení s dostatečným výkonem, byla zvolena architektura tlustého klienta, kdy server slouží jako vzdálené úložiště volumetrických dat, která jsou zobrazována až klientským zařízením. Tato architektura sice zvyšuje nároky na rychlost přenosu dat mezi klientským a serverovým zařízení, avšak oproti zobrazování na serveru vyžaduje výrazně menší nároky na výpočetní výkon serverového stroje.

Při zobrazování rozsáhlých volumetrických dat není vždy nutné zobrazovat všechny části scény, nebo je zobrazovat v plné kvalitě, bude proto vhodné, aby server poskytoval klientovi data po určitých částech a v různých rozlišeních. Poskytování těchto částí dynamicky by bylo však pro server značně výpočetně náročné, proto bude vhodné, aby server nejdříve data rozdělil do menších bloků v různých rozlišeních, a ty pak poskytoval klientské aplikaci. Výhodou tohoto přístupu je, že předzpracování dat může být provedeno pouze jednou pro daný dataset, a poté může server data poskytovat bez jakýchkoli dodatečných operací. Jak již bylo zmíněno v předchozí sekci 3.1, rozdělení dat do bloků o různých kvalitách využívá i například Mwalongo et al. (viz [17]), ale jelikož necílí na tolik rozsáhlá data, udržuje bloky pouze v operační paměti. Tento přístup však pro rozsáhlejší datasety nepřipadá v úvahu a bloky bude nutné uhovávat v rámci úložiště serveru, což však zvýší nároky na jeho kapacitu.

Klientská aplikace bude implementovat potřebný mechanismus pro přístup k tomuto datovému serveru, ze kterého bude potřebná data získávat a zobrazovat. Taktéž zde bude vhodné zavést mechanismus pro vzdálenou správu jednotlivých datasetů. Data budou zobrazována na CPU metodou vrhání paprsků. Tato metoda přináší pro rozsáhlá data několik výhod, které jsou popsány v kaptiole 5.3. Zobrazování bude prováděno na CPU, jelikož běžné grafické karty často nedisponují příliš velkou grafickou pamětí v porovnání s běžně dostupnými velikostmi pamětí RAM. Princip celého systému je znázorněn schématem 3.1.



Obrázek 3.1: Schéma vlastního klient-server systému pro zobrazování rozsáhlých volumetrických dat.

Kapitola 4

Návrh datového serveru

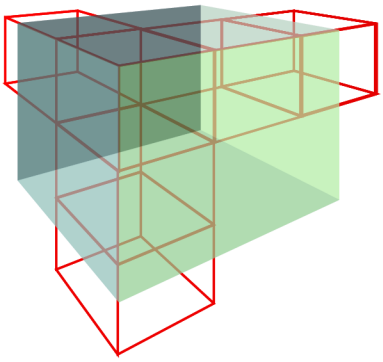
V předešlé kapitole 3 bylo vysvětleno, z jakého důvodu je výhodné implementovat server, který bude sloužit jako vzdálené úložiště volumetrických dat. Tato kapitola se zabývá návrhem tohoto datového serveru, vhodnou reprezentací volumetrických dat na straně serveru, možnostmi jejich předzpracování a taktéž návrhem vhodného aplikačního programového rozhraní (API) pro jejich poskytování a správu nad protokolem *HTTP*.

4.1 Příprava a dekompozice dat

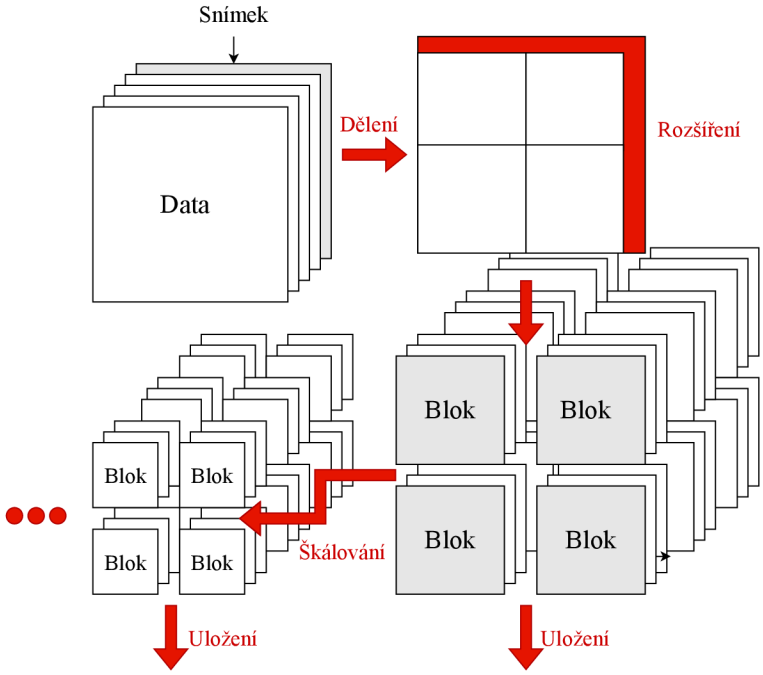
Jak již bylo zmíněno v kapitole 3, data budou na serveru uchovávána po malých blocích v různých kvalitách. Jelikož mapování na prostor vyplňující křivky, zmíněné v kapitole 2.1, funguje nejlépe pro prostor o rozměrech s hodnotou mocniny dvou, bude žádoucí, aby se jednalo o bloky ve tvaru krychle o hraně 2^N, kde N je libovolné přirozené číslo. Samotnou velikost bloku však bude vhodné zavést dynamickou, aby bylo možné experimentálně najít nejoptimálnější řešení. Tyto bloky budou postupně škálovány na bloky s nižším rozlišením. V tomto ohledu se jeví nejvhodnější škálování vždy na polovinu rozlišení hrany škálovaného bloku.

Samotná volumetrická data nemusí mít velikosti soudělné s velikostí bloků, bude proto nutné s touto situací počítat při dělení těchto dat. Tuto problematiku znázorňuje obrázek 4.1. Situace se dá řešit několika způsoby, kdy nejjednodušší je doplnění chybějících dat konkrétní hodnotou (například nejnižší možná hodnota). Taktéž by bylo možné prostor rozdělit na bloky, které se částečně překrývají, neboli tak, aby se v blocích některá data opakovala. Tato metoda však komplikuje některé další postupy, a tato práce se jí dále nezabývá.

Dále je třeba určit, jaký typ serializace bloků bude na serveru prováděn, a jakým způsobem budou tato data ukládána. Jelikož klient bude data číst ze síťového rozhraní a následně si je bude ukládat do vnitřní paměti, může být změněn typ serializace při tomto procesu. Nejvhodnější bude, aby na serveru byla data uložena klasicky po řádcích, což umožní v případě potřeby využít některých kompresních formátu, které tuto serializaci vyžadují, zjednoduší případné zobrazení po vrstvách, což je přínosné i pro účely ladění, a taktéž zjednoduší zpracování dat serverem. Bloky budou vytvářeny dělením jednotlivých snímků vstupních dat na podsnímky, které poté budou skládány do samostatných bloků. Tyto bloky budou dále rekurzivně škálovány na bloky s nižší kvalitou. Princip vytváření bloků je znázorněn na obrázku 4.2. Způsobem ukládání těchto dat a jejich metadat se zabývá sekce 4.2.



Obrázek 4.1: Dělení volumetrických dat na bloky pevné velikosti může způsobit situaci, kdy některé bloky nejsou zcela zaplněny.



Obrázek 4.2: Princip dělení volumetrických dat na bloky o různých kvalitách.

Sever může být provozován obecně na stroji s různou endianitou, stejně tak může mít různou endianitu soubor vstupních dat. Při použití externích knihoven není třeba při načítání endianitu řešit, avšak při uložení jednotlivých bloků je již situace jiná. Rozhodl jsem se pro zachovávání endianity stroje při ukládání bloků, což zjednoduší zápis dat, avšak klientu bude muset být informace o endianitě poskytnuta. Na výkon klienta toto řešení nebude mít žádný vliv, jelikož i ten může být provozován na stroji s různou endianitou, a musel by tedy převod implementovat i při předem definované endianitě bloku. Nejlepší řešení je definovat tento parametr v rámci metadat vytvořených bloků, jelikož již předpřipravené datasety budou přenosné mezi architekturami.

4.2 Ukládání dat a metadat

V sekci 4.1 bylo popsáno, jakým způsobem bude server zpracovávat vstupní data a rozdělovat je na jednotlivé bloky. Musí být však vyřešen způsob ukládání těchto bloků do úložiště počítače spolu s jejich metadaty. Pro ukládání bloků lze vybírat z mnoha možností, ale dobře zde poslouží i jejich ukládání do samostatných souborů v běžném souborovém systému. Bloky by však neměly být příliš malé v porovnání s bloky paměťovými, aby nedocházelo k nadměrnému zvyšování paměťových nároků. Jelikož jednotlivé bloky se liší pouze svým umístěním v prostoru a kvalitou, nemusí být u každého uchovávána metadata zvlášť a postačí jediný soubor metadat pro celou scénu. Tento soubor bude kromě samotných metadat obsahovat také stavové informace projektu, které bude potřebovat klientská aplikace. Pro reprezentaci tohoto souboru byl zvolen formát *JSON*, který je podporován většinou dnešních programovacích jazyků a je poměrně přívětivý. V rámci metadat budou uchovávány tyto informace:

1. Datový typ voxelu
2. Rozměry voxelové mřížky výsledných dat (Násobky velikosti bloku)
3. Rozměry voxelové mřížky původních dat (Slouží k omezení zobrazovaných dat.)
4. Rozměry voxelu
5. Velikost bloku
6. Stav projektu (Blíže popsáno v 4.3.)
7. Endianita dat

Samotné bloky by mohly být uloženy i v surové podobě, jelikož však budou přenášeny po internetové síti, bude vhodné provést alespoň základní kompresi. Komprese taktéž umožní snížit nároky na úložiště serveru. Je také nutné myslet na náročnost dekomprese těchto dat na straně klientské aplikace. Pro účely vytvářené aplikace jsem se rozhodl pro bezztrátový kompresní formát *GZIP*, jelikož jak bylo zmíněno v kapitole 2.1, ztrátová komprese nemusí být pro některá využití vždy tolerována. Výhodou tohoto formátu je schopnost provádět kompresi a dekompresi přímo na datových streamech, za použití pouze malého množství dodatečné paměti [5]. Budou-li uchovávána v jednom adresáři pouze data jediného datasetu, mohou být jednotlivé komprimované bloky uloženy do souborů s názvy skládající se z x, y, z indexů těchto bloků a indexu značícího úroveň jejich podškálování, tedy například *data_{X}_{Y}_{Z}_{S}*. Podle tohoto názvu bude docházet k vyhledávání požadovaného bloku. Název souboru pro metadata může být libovolný, například *info.json*.

4.3 Webové API

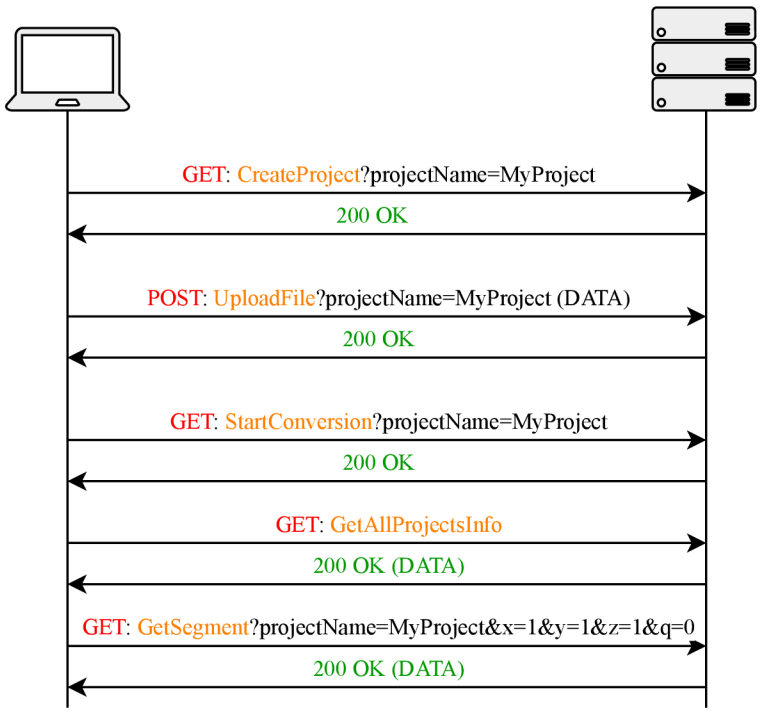
Aby klientská aplikace mohla ze serveru získávat data a mohla provádět správu datasetů, musí být navrhnuto vhodné aplikační programové rozhraní (API). Server bude s klientskou aplikací komunikovat nad protokolem *HTTP*, který nabízí velkou míru flexibility, jak rozhraní implementovat.

Nejdříve je nutné určit operace, které po serveru budou požadovány. Samotný server bude schopen pojmout více datasetů, které budou organizovány do projektů. Jeden projekt

bude vždy obsahovat pouze jeden dataset. Server tedy nejdříve bude muset být schopen vytvořit nový projekt, poté dojde k nahrání souboru zdrojových dat (samotný dataset), a následně bude možné provést převod těchto dat na data, nad kterými pracuje aplikace (bloky a metadata). Klientská aplikace bude dále potřebovat možnost získání informací o všech dostupných projektech a možnost dotazovat se na jednotlivé bloky. Poslední nezbytnou funkcí je možnost odstranění projektu. Bude tedy nutné implementovat tyto operace:

- Vytvoření nového projektu
- Odstranění projektu
- Nahrání vstupních dat
- Spuštění konverze dat (Vytvoření bloků ze vstupních dat)
- Získání informací o všech projektech
- Získání daného bloku v dané kvalitě

Z výše uvedených operací je patrné, že bezstavové API bude dostačující. Toto rozhraní bude založeno na návrhovém vzoru RPC (vzdálené volání procedur). Jednotlivé akce a jejich parametry budou výhradně identifikovány prostřednictvím *URI*, a budou využívat *HTTP* metod *GET* a *POST*, kdy jednotlivé operace budou identifikovány jejich názvem. Při chybovém stavu bude standardním způsobem navrácen odpovídající chybový *HTTP* status kód, v opačném případě dojde provedení požadavku a zaslání odpovědi s kódem *200 OK*. Ukázka komunikace je znázorněna sekvenčním diagramem 4.3. Jednotlivé funkce API jsou pak podrobně popsány v rámci kapitoly 6.1.



Obrázek 4.3: Sekvenční diagram komunikace mezi serverem a klientem popisuje vytvoření nového projektu, nahrání vstupních dat, konverzi dat a dotaz na datový bok. (Jednotlivé dotazy jsou oproti reálné implementaci zjednodušené.)

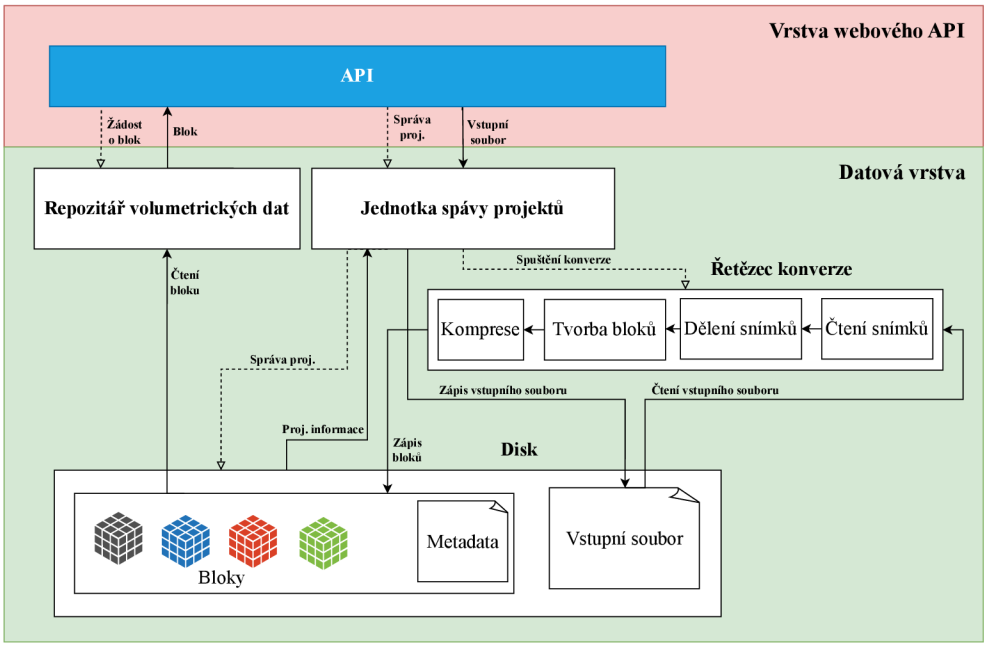
Jelikož vytvoření a příprava projektu před prvním dotazem na data sestává z několika nezávislých operací (vytvoření projektu, nahrání souboru, spuštění konverze), kdy konverze bude kvůli svému možnému trvání vykonána nezávisle na klientském požadavku (Klient pouze iniciuje její spuštění.), bude potřebné uchovávat informaci o stavu, ve kterém se daný projekt nachází. Stavy projektu budou následující:

- Projekt vytvořen
- Probíhá nahrávání vstupního souboru
- Vstupní soubor nahrán
- Probíhá předzpracování
- Předzpracování dokončeno

Informace o stavu bude uchovávána v souboru společně s metadaty.

4.4 Výsledný návrh

V předešlých sekcích byly popsány jednotlivé části, které budou muset být v rámci serverové aplikace implementovány. Na jejich základě tedy může být představen finální návrh serverové části. Jelikož aplikace je poměrně jednoduchá, byla zvolena dvouvrstvá architektura, která rozděluje aplikaci na datovou vrstvu a vrstvu webového API. Vrstva webového API zajišťuje komunikaci s klientskou aplikaci přes *HTTP* protokol a řídí datovou vrstvu. Datová vrstva řeší správu projektů, konverzi původního souboru na soubory vnitřní reprezentace a poskytování potřebných dat nadřazené vrstvě. Výsledné schéma je znázorněno na obrázku 4.4.



Obrázek 4.4: Schéma datového serveru. Plné šipky reprezentují směr toku dat, čárkované šipky reprezentují řízení.

Kapitola 5

Návrh klientské aplikace

Cílem této kapitoly je navrhnout klientskou aplikaci, která bude schopná zobrazovat rozsáhlá volumetrická data umístěná na datovém serveru, přes jehož rozhraní bude tato data získávat. Aplikace by měla být schopna data zobrazovat v reálném čase pouze s využitím běžně dostupného CPU a s běžnou velikostí operační paměti. Bude tedy nutné vytvořit strategii pro načítání a ukládání dat, najít vhodný typ zobrazení a navrhnout vhodnou paralelizaci jednotlivých procesů. Neméně důležitou součástí bude vytvoření uživatelského rozhraní poskytujícího důležité ovládací prvky, které umožní pohodlné prohlížení těchto dat a jejich správu na datovém serveru.

5.1 Získávání dat prostřednictvím webového API

Volumetrická data jsou na serveru uložena po malých blocích v různých kvalitách (viz kapitola 4.1). Klientská aplikace tedy bude muset implementovat mechanismus, kterým pomocí webového API tato data bude získávat. Data budou čtena ze síťového rozhraní bajt po bajtu, následně budou dekomprimována a převedena na pole hodnot daného typu. Jelikož server i klient mohou pracovat na různých architekturách, klientská aplikace bude muset být schopna přijímat data v obojí endianitě.

Jak již bylo zmíněno v kapitole 4.1, na serveru budou data serializována po řádcích, což pro účely zobrazování metodou vrhání paprsků není příliš vhodné. Data tedy budou budou při jejich přijímání přeserializována metodou s vyšší prostorovou lokalitou. Vhodné se jeví využití Mortonovy Z-křivky, která, jak již bylo zmíněno v kapitole 2.3, nemá nejvyšší míru prostorové lokality, avšak převod prostorových souřadnic na souřadnice z křivky lze provést poměrně rychle a efektivně (viz kapitola 2.3), například zde lze využít specializovaných instrukcí sady bitové manipulace (BMI).

5.2 Paměť dat

Paměť dat bude klíčovou částí klientské aplikace, jelikož bude pro souřadnice x, y, z poskytovat konkrétní hodnoty volumetrických dat. Aplikace bude muset rozhodovat o tom, jaká data bude v aktuální chvíli uchovávat v paměti, o jaká data bude žádat server a která data bude aktuálně zobrazovat. Cílem je navrhnout co možná nejlepší strategii tak, aby uživatelský dojem byl co možná nejlepší, s ohledem na odezvu sítě při omezené velikosti operační paměti.

Strategie paměti dat

Jelikož načítání jednotlivých bloků přes síť není zdaleka okamžité a nějaký čas trvá, bude vhodné, aby před prvním zobrazení byla scéna dostupná alespoň ve velmi nízké kvalitě celá. Surová data v nejnižší kvalitě budou mít velikost danou vztahem $S_N = \frac{S}{8^N}$, kde N je hloubka zanoření postupného podškálování a kde S je velikost surových dat po zarovnání do bloků. Při velikosti datasetu jeden terabajt a hloubce 3 se jedná přibližně o velikost dvou gigabajtů, což při dnes běžných velikostech operační pamětí není nikterak vysoká hodnota. Pokud by byly nároky na paměť vyšší, lze zvážit zvýšení maximální úrovně škálování. Tato data budou po celou dobu zobrazování k dispozici, aby je bylo možné kdykoli využít, a tudíž budou nezávislá na datech ve vyšších kvalitách.

Paměť dat tedy bude sestávat ze dvou polí, kdy jedno bude obsahovat ukazatele na bloky v nejnižší kvalitě, a kdy druhé bude obsahovat ukazatele na bloky ve vyšší kvalitě. V druhém poli mohou mít data bloků hodnotu null. Indexace těchto polí bude odpovídat indexaci bloků po řádcích, aby nebylo nutné blok vyhledávat, ale aby bylo možné přímo zjistit jeho adresu. Dalo by se zde využít i různých vyhledávacích struktur, kdy by nebylo potřebné mít vytvořené pole o délce všech prvků, avšak první způsob zajistí rychlejší získání bloku.

Strategie načítání dat

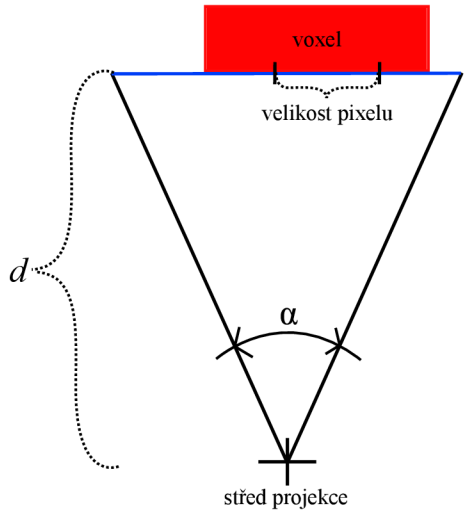
Aby bylo možné zobrazovat data v přijatelné kvalitě, musí být vytvořena strategie, která zajistí dostupnost potřebných dat v závislosti na aktuálním pohledu. Po vytvoření každého snímku zobrazovacími vlákny dojde ke kontrole kvality bloků, které byly v tomto zobrazení alespoň jednou použity. Pokud by bylo vhodné zobrazit tato data ve vyšší kvalitě, aplikace požádá server o zaslání těchto dat. Jelikož tato akce trvá nějaký čas, navíc takovýchto bloků může být velké množství, bude načítání bloků vykonáváno nezávisle.

Při potřebě vyšší kvality bude blok společně s informací o budoucí kvalitě zařazen do obslužné datové struktury, kterou budou obsluhovat jiná vlákna. Pokud se v této struktuře blok již nachází, dojde pouze ke změně hodnoty jeho budoucí kvality. Pro zobrazování bude nejvhodnější pro tyto účely využít zásobník, jelikož LIFO přístup zde zařídí, že po změně pohledu budou mít vyšší přednost aktuálně zobrazovaná data. Bude však vhodné před spuštěním načítání daného bloku znovu ověřit, zdali byl blok v minulých zobrazeních použit, nebo se již nepoužívá, aby nedocházelo k aktualizování nepoužívaných bloků.

Při kontrole kvality bude proveden výpočet potřebné kvality dle polohy kamery, zorného úhlu, a velikosti voxelu. Tento výpočet je dán vztahem 5.1,

$$q = \frac{w_x \cdot v_{max}}{2\tan(\frac{\alpha}{2}) \cdot d} \tag{5.1}$$

kde w_x je horizontální rozlišení obrazovky v pixelech, α je zorný úhel, d je vzdálenost bloku od kamery a kde v_{max} je maximální rozměr voxelu. Vztah byl odvozen jako podíl maximálního rozměru voxelu a velikosti pixelu obrazové roviny ve vzdálenosti d. Princip znázorňuje obrázek 5.1. Výsledný koeficient q říká, kolika násobně vyšší kvalita dat je potřebná (vztaženo na jeden rozměr). V této aplikaci bude referenčním rozlišením originální rozlišení dat.



Obrázek 5.1: Princip výpočtu potřebné kvality dat je založen na maximálním rozměru voxelu, velikosti pixelu a vzdálenosti od kamery.

Strategie uvolňování paměti

Jelikož aplikace musí pracovat s omezeným množstvím operační paměti, musí být vytvořena strategie, podle které budou při nedostatku paměti bloky odstraňovány. Tato strategie bude založena na odstraňování bloků, které nejsou aktuálně využívány a bloků, které jsou načteny ve vyšší kvalitě, než je aktuálně potřebné.

Pokud po vygenerování snímku bude zjištěno, že paměť se blíží ke svému naplnění, dojde nejdříve ke snížení kvality několika bloků, které mají vyšší kvalitu, než byla potřebná pro dokončené zobrazení. V případě, kdy takové bloky již nejsou k dispozici a paměti je stále nedostatek, bude vybráno k odstranění několik bloků, které nebyly v právě dokončeném pohledu využity. Bloky nebudou odstraněny okamžitě, ale budou zařazeny do fronty bloků k odstranění (důvod je popsán v následující podsekci). Bude-li v aplikaci po změně pohledu potřeba zobrazit odstraněný blok, jedná se o obdobnou situaci jako při prvním zobrazení. V tuto situaci se tedy uživatel bude muset spokojit se zobrazením bloku v nejnižší kvalitě a v případě volné paměti bude kvalitnější blok načten ze serveru.

Nebude-li dostatek místa pro načtení bloku v požadované kvalitě, bude nejvhodnější blok ponechat beze změny. Navíc bude zakázáno přidávání dalších bloků do zásobníku bloků k přenačtení.

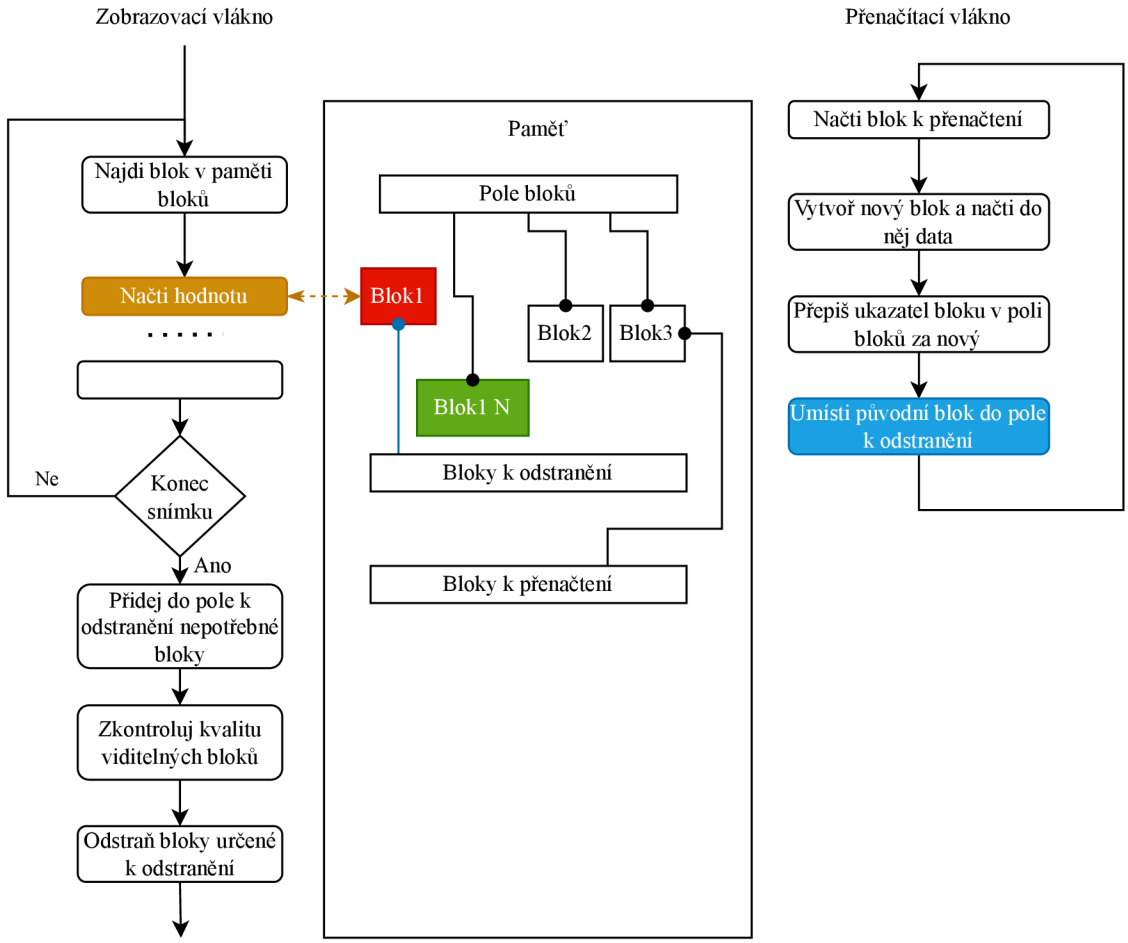
Správa paměti dat

V předešlých podsekcích byly popsány strategie, které se v paměti budou uplatňovat. Jelikož některé operace mohou trvat poměrně dlouhou dobu, bude vhodné zajistit jejich nezávislost a paralelizaci. Bude tedy nutné řešit komunikaci a řízení vláken, kdy takováto realizace bude muset být v některých výkonnostně kritických místech dobře promyšlena.

V aplikaci se bude vyskytovat několik zobrazovacích vláken, které budou číst informace z jednotlivých bloků, a několik vláken, která budou provádět načítání bloků ze serveru. Nejdříve bude muset být vyřešeno, jakým způsobem bude provedeno nahrazení bloku s nižší kvalitou za blok s vyšší kvalitou, když k tomuto bloku může zároveň přistupovat jedno nebo více zobrazovacích vláken. Nabízí se zde možnost využití zamykacích mechanismů, které jsou však pro toto využití příliš pomalé, primárně z důvodu nutnosti častého zápisu do sdílené

paměti. Je důležité poznamenat, že čtení informace z bloku je výkonnostně nejkritičtějším místem celého programu. Nejlepší možností bude řešit tuto situaci bez nutnosti zamykání.

Uvažujme situaci, kdy si zobrazovací vlákno zkopíruje ukazatel na daný blok z pole bloků. Jelikož jak kopírování ukazatele, tak jeho zápis lze provést atomicky, bude možno nahrazení bloku provést jako zápis ukazatele nově vytvořeného bloku. Načítací vlákno tedy vytvoří nový blok v požadované kvalitě, uschová si ukazatel na blok původní a provede atomický přepis původního ukazatele v poli bloků na nový. Zobrazovacího vlákna se tato akce nijak nedotkne, jelikož díky kopii ukazatele provádí celou posloupnost operací výhradně nad jedním z těchto dvou bloků. Původní blok však z tohoto důvodu nemůže být prozatím odstraněn a musí čekat na dokončení práce zobrazovacích vláken. Pro tento účel bude zavedena datová struktura, do které budou tyto bloky přidány a odstraněny budou teprve po dokončení zobrazení aktuálního snímku. Tato problematika je znázorněna na schématu 5.2, které znázorňuje jednotlivá vlákna a stav společné paměti daný jejich barevně znázorněnou pozicí. Pro zabezpečení přístupu k přenačítací datové struktuře může být využito již klasického zámku.



Obrázek 5.2: Diagram přístupu zobrazovacího a přenačítacího vlákna do paměti bloků. Červený blok značí čtený blok, který je připraven k odstranění. Zelený blok červený nahradil, avšak některá část aplikace stále může přistupovat k původnímu bloku. Odstranění bude bezpečné provést až po ukončení zobrazování daného snímku.

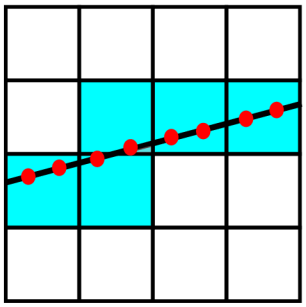
Stejný výkonnostní problém, jako při využití zamykacích struktur, se projeví i při jakýchkoli častých zápisech do sdílené paměti. Tento problém je způsoben nutností udržovat data vyrovnávacích pamětí v koherentním stavu. Při implementaci je tedy nutné vyvarovat se zbytečným zápisům do sdílené paměti.

5.3 Zobrazování metodou vrhání paprsků

Jak již bylo řečeno v kapitole 2.2, metoda vrhání paprsků pracuje na principu vzorkování paprsku vrženého ze středu kamery skrze pixel obrazovky. Jelikož cílem aplikace je zobrazování dat, která jsou velmi rozsáhlá v reálném čase, algoritmus pro vrhání paprsku bude zjednodušen i za cenu horších vizuálních výsledků do podoby, která umožní data zobrazovat rychle a efektivně. V kapitole 2.2 byly zmíněny metody, které globálně poskytují vyšší výkon, avšak aby byly efektivní, vyžadují data uložená v několika variantách, což je v případě takto rozsáhlých dat problematické. Tato duplicita by mohla být řešena na straně serveru, avšak to by drasticky zvýšilo síťovou komunikaci a docházelo by k problémům s latencí. Pokud by v těchto metodách byla uvažována data neseřazená, metody by zde přišly o svou hlavní výhodu oproti metodě vrhání paprsků.

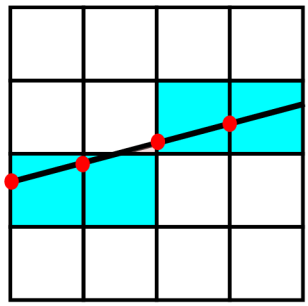
Vrhání paprsků v jednotkové voxelové mřížce

Uvažujeme situaci na obrázku 5.3, kde je znázorněn paprsek procházející mřížkou s velikostmi buňky 1. Při provedení vzorkování s pevně daným krokem může být v každém kroku aktuální hodnota získána buď čtením hodnoty nejbližší buňky, nebo lze provést trilineární interpolaci hodnot okolních buněk v závislosti na dané pozici. Druhá metoda nabízí opticky hezčí výsledky díky plynulým přechodům, avšak výpočet hodnoty pro daný vzorek je výpočetně mnohem náročnější. Z důvodu požadavků na vysoký výkon bude zvolena první metoda.



Obrázek 5.3: Paprsek procházející mřížkou vzorkující s konstantním krokem.

Z obrázku 5.3 si můžeme povšimnout, že celá situace je podobná rasterizaci přímky. Pokud by byl krok zvolen takový, aby vzorkování v ose nejrychlejšího postupu postupovalo vždy o jedničku, byla by situace podobná rasterizaci přímky algoritmem DDA. Tuto situaci nastiňuje obrázek 5.4. Opačně lze využít algoritmy pro průchod mřížkou při rasterizaci pro průchod volumetrickými daty při jejich zobrazování. Tyto algoritmy jsou efektivní z hlediska kroku, kdy nedochází k opakovanému čtení stejných voxelů v rámci jednoho paprsku, při zachování alespoň diagonální sousednosti po sobě čtených voxelů. Z tohoto důvodu bude využito procházení prostoru pomocí DDA. Tohoto algoritmu pro průchod voxelovou mřížkou využívá ve své práci také například Hilton [10].



Obrázek 5.4: Paprsek procházející mřížkou vzorkující s krokem podle DDA.

Uvažujme perspektivní kameru o středu $S[S_1, S_2, S_3]$, rozlišení w_x, w_y se zorným úhlem α, kdy kamera se dívá ve směru osy z. Směrový vektor $\vec{r} = (r_1, r_2, r_3)$ paprsku procházejícího pixelem P o indexech P_x, P_y lze získat pomocí vztahu 5.2.

$$\vec{r} = \left(P_x - \frac{1}{2}w_x, P_y - \frac{1}{2}w_y, \frac{w_x}{2\tan\frac{\alpha}{2}} \right) \tag{5.2}$$

Parametrickou rovnice přímky paprsku pak lze zapsat jako 5.3.

$$[x, y, z] = S + t \cdot \vec{r}, \quad t \in \mathbb{R} \tag{5.3}$$

Tuto rovnici lze následně upravit tak, aby sloužila jako generátor vzorků pro DDA. Toho lze dosáhnout vydělením všech složek směrového vektoru jeho největší složkou a vhodným posunutím počátku. Úprava je znázorněna rovnicemi 5.4,

$$\begin{aligned} i &= index\_of\_max(\{r_1, r_2, r_3\}) \\ \vec{r}_n &= \left(\frac{r_1}{r_i}, \frac{r_2}{r_i}, \frac{r_3}{r_i} \right) \\ [x, y, z] &= S + t \cdot \vec{r}_n + (S_i \bmod 1)\, \vec{r}_n, \quad t \in \mathbb{N} \end{aligned} \tag{5.4}$$

kde $index\_of\_max$ vrací index největšího prvku.

Jelikož kamera bude moci kromě své pozice měnit i své natočení, bude nutné měnit směr paprsku v závislosti na rotaci kamery. Toho lze dosáhnout aplikováním rotační matice R na směrový vektor $\vec{r}$. Pro účely této aplikace bude dostačující popis rotace Eulerovými úhly v posloupnosti xyz.

Jednotlivé vzorky budou dle mapovací tabulky (tabulka, která mapuje vstupní hodnotu na výstupní hodnotu) převáděny na $RGBA$ barevnou hodnotu a postupně budou skládány do výsledné barvy. Toto skládání bude prováděno postupně, kdy výsledná hodnota bude počítána dle vztahů 5.5,

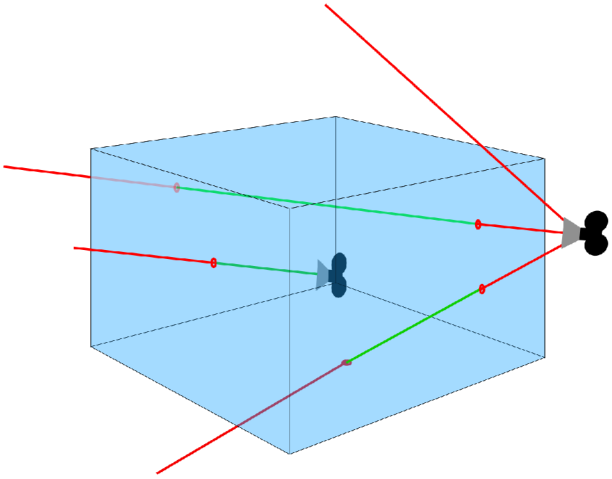
$$\begin{aligned} R_i &= R_{i-1} + r \cdot \alpha \cdot (1 - A_{i-1}) \\ G_i &= G_{i-1} + g \cdot \alpha \cdot (1 - A_{i-1}) \\ B_i &= B_{i-1} + b \cdot \alpha \cdot (1 - A_{i-1}) \\ A_i &= A_{i-1} + \alpha \cdot (1 - A_{i-1}) \end{aligned} \tag{5.5}$$

kde R_i, G_i, B_i, A_i jsou výsledkem barvy i-tého kroku při aktuálních barevných hodnotách voxelu r, g, b, α v rozsahu hodnot $\langle 0, 1 \rangle$. A (α) značí hodnotu neprůhlednosti. Bude-li se

hodnota A blížit jedné, bude paprsek ukončen, protože další vzorky by již na výslednou barvu neměly žádný vliv. Jedná se tedy o optimalizaci předčasným ukončením paprsku.

Na hodnoty barev jednotlivých vzorků by mohl být aplikován osvětlovací model, pro který je však potřebné vyčíslit gradienty jednotlivých voxelů, kterými paprsek prochází. Tento výpočet může být předpočítán, avšak ukládání těchto hodnot pro každý voxel by mohlo mohlo zabírat stejný paměťový prostor, jako data samotná. Metody kvalitnějšího zobrazování mohou být námětem k dalšímu vylepšení, avšak při požadavku na zobrazování velmi rozsáhlých dat v reálném čase je výkon prioritou.

Jelikož samotná volumetrická data zaobírají předem daný objem, bylo by zbytečné mimo tento objem paprsek počítat. Bude tedy nutné získat úseky paprsku, které daným objemem prochází. Jelikož volumetrická data mají v uvažovaném případě vždy tvar kvádru, který je konvexním tělesem, bude existovat maximálně jeden takovýto úsek. Nejdříve musí být získány průsečíky přímky paprsku se stěnami tohoto kvádru, kdy jsou uvažovány pouze ty, které leží před kamerou. Pokud se kamera nachází uvnitř prostoru dat, bude vzorkování vykonáváno od středu kamery k nejvzdálenějšímu průsečíku. V případě, že kamera leži vně, bude vzorkována část paprsku mezi nejbližším a nejvzdálenějším průsečíkem. V případě, že kamera je vně a počet průsečíků je menší nebo roven jedné, paprsek nebude počítán vůbec. Různé situace při průchodu paprsku daty jsou znázorněny na obrázku 5.5.



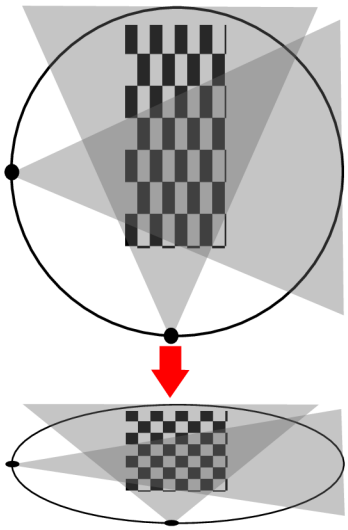
Obrázek 5.5: Různé situace při průchodu paprsku scénou. Pouze zelené části mají význam pro výsledné zobrazení.

Výpočet průsečíků přímky s jednotlivými rovinami je velmi jednoduchý, jelikož každá z rovin je kolmá na jednu z os. Stačí tedy z parametrických rovnic přímky vyjádřit parametr t pro danou hodnotu x, y, nebo z a zpětně dopočítat hodnoty ostatních souřadnic. Pokud uvažujeme parametrickou rovnici popsanou v 5.3, pak kladná hodnota t značí, že bod leží před kamerou. Následně stačí porovnat souřadnice těchto průsečíků s hranicemi stěn, aby bylo možné určit, zdali bod leží uvnitř stěny, nebo vně.

V případě implementace metody vrhání paprsků na více vláknech bude vhodné zajistit, aby byly počítány pospolu paprsky, které procházejí blízkými pixely. Při tomto výběru je nejvyšší šance, že pro zobrazování budou používána blízká data, a bude tedy docházet k optimálnějšímu využití vyrovnávacích pamětí procesoru.

Transformace obecné scény na scénu s jednotkovou voxelovou mřížkou

V minulé sekci byla popsána zjednodušená metoda vrhání paprsku pro jednotkovou voxelovou mřížku. U běžných volumetrických dat však jednotlivé voxely nemají povahu krychle s hranou o délce jedna, ale jedná se o kvádry. Pro implementaci bude tedy nutné zařídit, aby bylo možné zobrazit tato data se správnými rozměry. Problém lze řešit buď úpravou vzorkování tak, aby správně fungovalo pro danou velikost voxelu, nebo by mohla být provedena transformace scény tak, aby samotné vzorkování zůstalo nezměněno. V praxi se nejčastěji využívá první metody, avšak druhá umožňuje provést vzorkování v jednotkové mřížce. Princip této metody je znázorněn na obrázku 5.6. Hlavní výhodou metody je, že souřadnice ve scéně, po převedení na celočíselnou hodnotu, přímo odpovídají prostorovým indexům jednotlivých voxelů, a není tedy nutné provádět jejich přepočet. Jelikož jsem chtěl zmapovat možnosti a úskalí této metody, rozhodl jsem se pro její implementaci.



Obrázek 5.6: Transformace scény na jednotkovou mřížku.

Princip metody bude spočívat v aplikaci matice změny měřítka na pozici kamery a na směrový vektor paprsku. Uvažujme scénu, kde mají voxely tvar kvádru o rozměrech v_x, v_y, v_z. Matice transformace změny měřítka H bude dána vztahem 5.6.

$$H = \begin{pmatrix} \frac{1}{v_x} & 0 & 0 & 0 \\ 0 & \frac{1}{v_y} & 0 & 0 \\ 0 & 0 & \frac{1}{v_z} & 0 \\ a & 0 & 0 & 1 \end{pmatrix} \tag{5.6}$$

Ve scéně se nachází kamera s pozicí danou jejím středem S, kdy její rotace kolem tohoto středu je popsána rotační maticí R. Směrový vektor paprsku vycházejícího z kamery bez aplikace rotační matice má hodnotu $\vec{r}$. Transformace pozice kamery a hodnoty směrového

vektoru jsou pak dány vztahy 5.7.

$$\begin{pmatrix} S'_1 \\ S'_2 \\ S'_3 \\ q' \end{pmatrix} = H \cdot \begin{pmatrix} S_1 \\ S_2 \\ S_3 \\ q \end{pmatrix}$$
$$\begin{pmatrix} r'_1 \\ r'_2 \\ r'_3 \\ q' \end{pmatrix} = H \cdot R \cdot \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ q \end{pmatrix} \tag{5.7}$$

Jelikož absolutní délka kroku při uvažovaném DDA vzorkování není ve všech směrech stejná, což je navíc umocněno zmíněnou transformací, docházelo by ke změnám průhlednosti prostoru v závislosti na směru paprsku. Uvažujme například situaci, kdy je zobrazována krychle, která je složena z voxelů ve tvaru kvádru s délkami hran 1,2,3. Pokud rovnoběžně s těmito hranami vyšleme skrze krychli paprsky, každý z nich provede jiný počet vzorků, což se projeví na výsledné průhlednosti. Obdobný problém by vznikal také v případě, kdy by byla absolutní délka kroku konstantní, avšak by se projevil pouze při změně velikosti kroku (Při menším kroku by se scéna jevila méně průhledná a obráceně.).

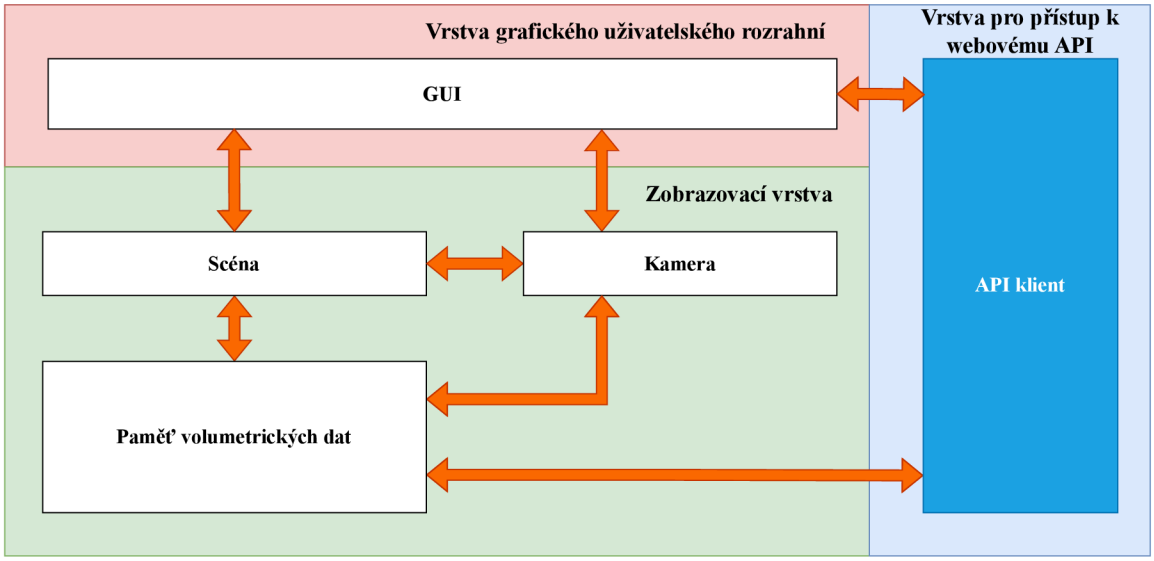
Tento problém lze snadno vyřešit přepočtem hodnoty α pro danou délku kroku (bez transformace změny měřítka). Pokud je hodnota α vztažena na jednotkovou délku, pak pro délku kroku l lze výslednou hodnotu neprůhlednosti α' získat pomocí vztahu 5.8.

$$\alpha' = 1 - (1 - \alpha)^l \tag{5.8}$$

Díky tohoto výpočtu lze navíc snadno implementovat změnu délky kroku v závislosti na potřebné kvalitě dat. Pro zachování výhod DDA bude tato délka vždy 2^N-tým násobkem původní délky.

5.4 Výsledný návrh

Na základě předešlých sekcí může být představen finální návrh výsledné klientské aplikace. Aplikace bude rozdělena celkem na tři vrstvy, a to na zobrazovací vrstvu, vrstvu grafického uživatelského rozhraní a vrstvu pro přístup k webovému API. Zobrazovací vrstva bude implementovat paměť volumetrických dat, objekt scény a objekt kamery, které společně budou realizovat samotné zobrazování volumetrických dat. Data budou získávána prostřednictvím vrstvy pro přístup k webovému API, která bude zajišťovat komunikaci se serverem přes *HTTP* protokol. S těmito vrstvami bude pracovat vrstva grafického uživatelského rozhraní. Blokové schéma je znázorněno na obrázku 5.7.



Obrázek 5.7: Schéma klientské aplikace. Šipky značí interakci mezi jednotlivými částmi aplikace.

Kapitola 6

Implementace

Tato kapitola se zabývá implementací výsledného systému pro zobrazování rozsáhlých volumetrických dat. Implementace je založena na kapitolách 5.4 a 4.4, kde byly popsány jednotlivé principy a části, které bylo nutné v rámci realizovaného systému implementovat.

6.1 Implementace datového serveru

Tato sekce se zabývá implementací výsledného datového serveru, který byl vytvořen s využitím technologie *ASP.NET Core* v jazyce *C#*. Pro účely datového serveru byl zvolen právě tento webový rámec, jelikož nabízí poměrně vysoký výkon a umožňuje plného využití všech vlastností jazyka *C#*. Jelikož server pracuje s různými datovými typy, nad kterými musí provádět výpočty, byl vybrán jazyk *C#* v nejnovější verzi 11, jelikož ta přinesla podporu generické matematiky.

Při implementaci byl kladen důraz na architektonický návrh s využitím vhodných návrhových vzorů tak, aby byla aplikace snadno rozšiřitelná a modifikovatelná. Konfigurace je řešena pomocí konfiguračního souboru *AppSettings.json*. Dle typického návrhu pro danou platformu je zde využito injektáže závislostí, která při použití vhodných rozhraní umožňuje lepší modularitu a snadnější testovatelnost.

Implementace aplikace odpovídá návrhu představenému v sekci 4.4, kdy jednotlivé vrstvy jsou realizovány projekty *Data* a *API*. Kromě těchto projektů se v aplikaci nachází projekt *Core*, který obsahuje soubory potřebné pro obě vrstvy, kdy bylo jejich umístění do nejnižší vrstvy nevhodné. Jedná se zejména o soubory konfigurace.

Datová vrstva

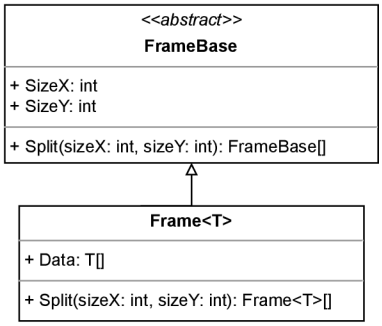
Tato vrstva zprostředkovává veškeré operace prováděné nad daty, kdy komunikaci s nadřazenou vrstvou obstarávají třídy `ProjectManager` a `VolumeDataRepository` prostřednictvím jim nadřazených rozhraní. První z těchto tříd řeší vytváření, odstraňování a editaci projektů, druhá zodpovídá za načítání jednotlivých bloků. Jelikož `VolumeDataRepository` je velmi jednoduchá, bylo by možné tyto třídy sloučit, avšak pro budoucnost projektu, kdy by bylo požadováno poskytovat data například po snímcích, bude lepší ponechat třídy oddělené.

Čtení vstupních dat

Čtení vstupních souborů zajišťují třídy implementující rozhraní `ISourceFileDataReader`, které poskytuje informace o volumetrických datech a generickou funkci `ReadFrame<T>` pro

načtení snímku v daném datovém typu. Serverová aplikace aktuálně obsahuje pouze jedinou takovouto třídu, která slouží pro čtení souboru ve formátu *Nifti* a byla vytvořena úpravou již existující knihovny v souladu její licencí. Pro podporu dalších formátu však stačí do aplikace přidat další třídy implementující toto rozhraní, jelikož celá aplikace, s výjimkou místa, kde dochází k rozpoznávání formátu souboru, pracuje již pouze s rozhraním.

Aby bylo možné snadno pracovat s daty v různých datových typech, byly třídy pro jejich reprezentaci vytvořeny tak, aby bylo možné v místech, kde datový typ není podstatný, pracovat bez jeho znalosti se zachováním možnosti zahrnout všechny potřebné datové typy. Dvojici tříd reprezentující jeden snímek znázorňuje diagram tříd 6.1. Abstraktní třída `FrameBase` umožňuje pracovat se snímkem bez znalosti typu jeho dat. Od této třídy dědí třída `Frame`, která je již generická a slouží v místech, kde dochází k přístupu k datům. Třídy obsahují jedinou funkci `Split`, která slouží k rozdělení snímku na podsnímky.



Obrázek 6.1: Diagram tříd, které společně reprezentují jeden snímek. Tyto třídy jsou navrženy tak, aby umožňovaly v určitých částech aplikace pracovat se s snímkem bez znalosti typu jeho dat.

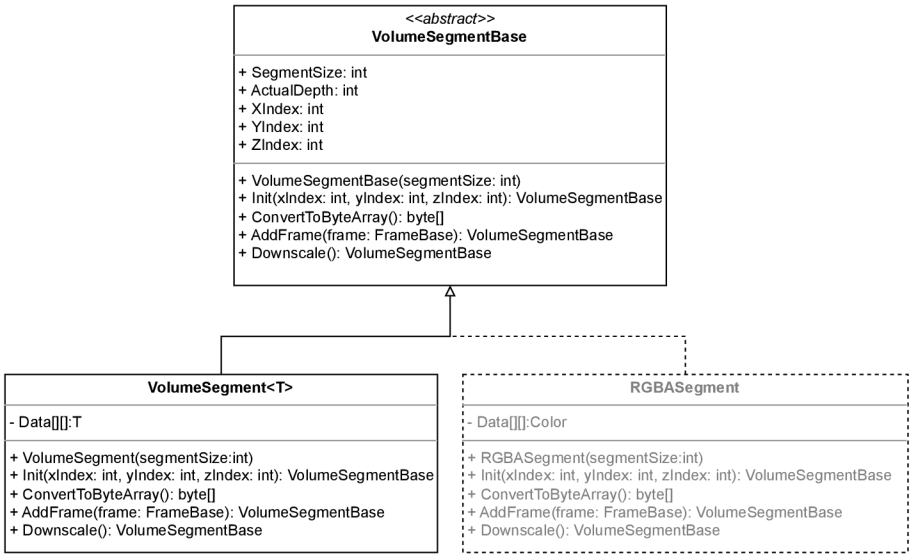
Konverze vstupních dat

Konverzi vstupního souboru na vnitřní reprezentaci zajišťuje třída `ConvertingPipeline`, která je odvozená od rozhraní `IConvertingPipeline`. Toto rozhraní obsahuje jedinou funkci `Apply`, která přijímá objekt třídy implementující rozhraní `ISourceFileDataReader` a cestu k adresáři, kam budou konvertovaná data uložena.

Při konverzi dochází k úkonům popsaným v kapitole 4.1. Pro reprezentaci bloků bylo nutné taktéž zavést třídy, které umožní práci s různými datovými typy, avšak situace zde je o něco složitější, jelikož nad daty jsou prováděny matematické operace. Jedná se o třídy `VolumeSegment` a `VolumeSegmentBase`, které jsou znázorněny v diagramu 6.2. Každá funkce třídy `VolumeSegmentBase`, která nevrací výsledek, vrací odkaz sebe sama, čímž je docíleno možného řetězení operací za sebou (tzv. Fluent API). Při práci s objektem bloku je objekt nejdříve inicializován pomocí funkce `Init`, poté dochází k vkládání jednotlivých snímků funkcí `AddFrame`, dále jsou data cyklicky převáděna na pole bajtů funkcí `ConvertToByteArray` a podškálovávána funkcí `Downscale`. Bajtová pole jsou v každém cyklu serializována, komprimována a poté ukládána do samostatných souborů. Tento postup se opakuje dokud nejsou všechna data zpracována.

Při práci s volumetrickými daty se výjimečně můžeme setkat i s daty barevnými, proto bylo při návrhu serverové aplikace myšleno na budoucí možné rozšíření podpory tohoto typu dat. Serverová aplikace však aktuálně tento typ dat nepodporuje, jelikož klientská část se zaměřuje pouze na data jednokanálová. Pro rozšíření o barevná data však stačí implementovat funkce v již předpřipravené třídě `RGBAVolumeSegment`. Tato třída je taktéž

znázorněna v diagramu 6.2. Obdobným způsobem by bylo možné implementovat i podporu libovolných vícekanálových dat.



Obrázek 6.2: Diagram tříd, které společně reprezentují jeden blok volumetrických dat. Třída `RGBAVolumeSegment` není prozatím implementována, ale slouží pro možné budoucí rozšíření podpory barevných dat.

Metadata jsou ukládána do souboru *info.json*, kdy každý projekt obsahuje právě jeden tento soubor. Pro budoucí práci by však bylo vhodné zvážit nahrazení tohoto principu uchovávání metadat relační databází, jelikož zde může docházet k souběžnému přístupu, což při požadavcích na konzistenci těchto dat komplikuje implementaci. Souběžný přístup při zápisu zde byl vyřešen opakujícími se pokusy získání výhradního přístupu k tomuto souboru, což však není optimální řešení.

Vrstva webového API

Webové API je velmi důležitou částí vytvořeného systému, jelikož umožňuje klientské aplikaci získávat data a spravovat projekty. Systém byl primárně zamýšlen pro provoz uvnitř lokální sítě, proto zde nebylo implementováno žádné zabezpečení. Pro práci mimo lokální síť nebo v síti, ke které mají přístup uživatelé neoprávnění manipulovat s těmito daty, by však bylo nutné implementovat autentizaci a autorizaci a taktéž případně využít namísto běžného *HTTP* protokolu zabezpečený protokol *HTTPS*.

Veškeré metody webového API byly implementovány do dvou kontrolerů. Kontrolér `ProjectManagementController` zajišťuje správu projektů, tedy jejich vytváření, mazání, nahrávání vstupních souborů a získávání informací o projektech. `DataController` pouze poskytuje jednotlivé bloky. Rozdělení akcí do dvou kontrolerů by mohlo být nápomocno i pro budoucí možné zabezpečení, jelikož pro editaci projektů většinou bývá potřebné vyšší oprávnění než pro pouhé získávání dat.

Díky zvolené technologii bylo možné všechny akce kontrolerů, které přistupují k datům, implementovat asynchronně, což může zvýšit výkon serveru, jelikož vlákna, která čekají na daný zdroj mohou být uvolněná, aby vykonávala jinou práci, než je zdroj dostupný.

Dále bylo nutné ošetřit chybové stavy a určit odpovídající *HTTP* status kód, který bude v případě této chyby klientovi navrácen. Pokud dojde k neošetřené chybě, aplikace defaultně taktéž odpoví chybovým kódem, je ale vhodné snažit se chybové stavy ošetřit, aby mohla být klientská aplikace informována o typu konkrétní chyby, například *NOT_FOUND* při nenalezení projektu.

Jednotlivé metody podporované serverem, včetně tvaru *URI* adresy a dalších informací, jsou následující.

- Vytvoření nového projektu

  URI: http://{server}/ProjectManagement/CreateProject?projectName=„“

  HTTP metoda: GET

  Chybové kódy: CONFLICT – projekt již existuje

- Odstranění projektu

  URI: http://{server}/ProjectManagement/DeleteProject?projectName=„“

  HTTP metoda: GET

  Chybové kódy: NOT_FOUND – projekt neexistuje

- Nahrání souboru s daty

  URI: http://{server}/ProjectManagement/UploadFile?projectName=„“ &fileName=„“

  HTTP metoda: POST

  Chybové kódy: NOT_FOUND – projekt neexistuje

- Spuštění konverze dat

  URI: http://{server}/ProjectManagement/ConvertProject?projectName=„“

  HTTP metoda: GET

  Chybové kódy: NOT_FOUND – projekt neexistuje

- Získání informací o všech projektech

  URI: http://{server}/ProjectManagement/GetAllProjectsInfo

  HTTP metoda: GET

- Získání daného bloku v dané kvalitě

  URI: http://{server}/Data/GetBlock?projectName=„“&xIndex=„“&yIndex=„“ &zIndex=„“&downsacale=„“

  HTTP metoda: GET

  Chybové kódy: NOT_FOUND – projekt, nebo blok neexistuje

6.2 Implementace klientské aplikace

V této sekci je popsána implementace klientské aplikace, která umožňuje zobrazování rozsáhlých volumetrických dat na CPU s využitím implementovaného datového serveru. Samotná aplikace je z důvodu výkonnostních požadavků vytvořena v jazyce *C++* ve verzi 20. Pro tvorbu uživatelského rozhraní byla zvolena knihovna *Dear ImGui*, která poskytuje možnost velmi jednoduché tvorby i komplexních grafických rozhraní pomocí návrhového vzoru *IMGUI*. Tento typ grafických rozhraní pracuje na principu nekonečné smyčky, díky čemuž je velmi jednoduchý na tvorbu, avšak pro některé typy aplikací, zvláště tam, kde je potřeba úspory energie, může být nevhodný. Pro účely implementace výpočtů z oboru lineární algebry, zejména pro práci s vektory a maticemi, zde byla využita knihovna *Eigen*.

Aplikace je opět rozdělena do několika vrstev, které jsou implementovány do samostatných projektů, tak jak bylo popsáno v kapitole 5. Projekt *Core* slouží primárně k získávání dat prostřednictvím webového rozhraní, zobrazovací vrstva je implementována v projektu *VolumeRender*. Tyto dva projekty spolu tvoří knihovny pro zobrazování rozsáhlých volumetrických dat, nad kterými se nachází nezávislé grafické rozhraní, které tyto knihovny používá. V případě potřeby by tedy bylo možné implementovat jiné grafické rozhraní, nebo knihovny integrovat do jiné aplikace.

Vrstva pro přístup k webovému API

Tato vrstva slouží pro komunikaci s vytvořeným serverem a umožňuje aplikaci přístup ke všem jeho funkcím. Pro implementaci *HTTP* komunikace byla zvolena uživatelsky přívětivá knihovna *cpp-httplib*, která již obsahuje většinu potřebných funkcionalit. Jak již bylo zmíněno, aplikace komunikuje nad nezabezpečeným protokolem *HTTP*, avšak implementace zabezpečení by díky této knihovně neměla být příliš náročná.

Vrstva je implementována v rámci projektu *Core* jedinou třídou `ProjectManager`, která obsahuje funkce odpovídající funkcím serveru. Na rozdíl od serveru tyto funkce nejsou tedy rozděleny do dvou tříd, což snižuje již tak velké množství parametrů konstruktorů tříd, které k webovému API přistupují. Většina funkcí je poměrně jednoduchá, kdy výjimkou je pouze nahrání vstupního souboru, kdy z důvodu velikosti není možné celý soubor přednačíst do paměti a musí být tedy streamován po částech. V případě navrácení chybového kódu, nebo při chybě připojení dojde k vyvolání výjimky s vhodným textem, která je dále odchycena v rámci uživatelského rozhraní a její text je vypsán uživateli.

Zobrazovací vrstva

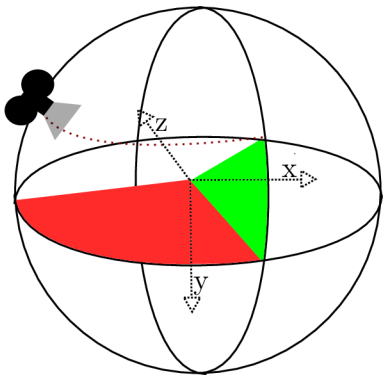
Zobrazovací vrstva je nejsložitější částí celé aplikace, která implementuje všechny důležité prvky pro zobrazování rozsáhlých volumetrických dat uložených na serveru metodou CPU ray casting. Je složena z několika dílčích částí, které společně umožňují zobrazovat data, měnit pohledy nad daty a taktéž měnit parametry daného zobrazení. Jednotlivé části jsou popsány v následujících podsekcích.

Kamera

Jedná se o abstrakci pozorovatele volumetrické scény, nebo také kamery, která je definována svou polohou, zorným úhlem, rozlišením a natočením. Kamera je implementována třídou `RayCastingCamera`. Krom výše zmíněných parametrů obsahuje i některé další pomocné parametry a obslužné metody, které slouží k jednoduššímu ovládání. Kamera taktéž

implementuje metody, které slouží k získávání hodnot v prostoru upraveném transformací změny měřítka popsané v kapitole 5.3, kdy tato transformace je implementována právě v rámci této třídy. Nejdůležitější je metoda `GetShrinkedRayDirection`, která slouží pro výpočet směrového vektoru paprsku pro daný pixel v perspektivní projekci v transformací upraveném prostoru, díky čemuž je možné provádět zobrazování metodou vrhání paprsků. Samotné zobrazování kamera však již neobstarává.

Jelikož bylo třeba realizovat vhodný způsob prohlížení dat, byl nakonec vybrán hojně využívaný koncept, kdy kamera rotuje kolem středu rotace v určité vzdálenosti, kdy střed rotace a vzdálenost je možné měnit. Rotace kamery je prováděna kolem os x, y v tomto pořadí, jak je znázorněno na obrázku 6.3, díky čemuž je docíleno intuitivního ovládání. Rotace kolem osy x je umožněna pouze v rozmezí $\pm 90°$, jelikož v případě rotace o větší úhel by bylo nutné provádět změny směru rotace kolem osy y. I tento přístup byl v rámci implementace testován, avšak ve výsledku působil spíše rušivě.



Obrázek 6.3: Kamera je nejdříve rotována kolem osy x a poté y, což umožňuje prohlížení scény obvyklým způsobem.

Zvolený způsob ovládání je v rámci aplikaci nazýván *orbiter*. Kamera implementuje funkci `Observe`, která přijímá delty dílčích pohybů, které jsou odvozeny z pohybů myši v rámci grafického rozhraní. Na základě vstupních parametrů dojde k relativnímu přenastavení kamery. Kameru lze také ovládat přímým nastavením pozice kamery a její rotace, což do budoucna nabízí možnost implementace i jiných typů ovládání.

Paměť volumetrických dat

Paměť volumetrických dat je velice důležitou částí klientské aplikace, jelikož poskytuje data ke zobrazení a zajišťuje jejich dostupnost. Paměť jako taková tedy nemá jen na starost uchovávání volumetrických dat, ale zajišťuje jejich načítání ze serveru, spravuje jejich velikost a poskytuje tato data jiným částem aplikace.

Paměť volumetrických dat je implementována třídou `VolumeObjectMemory`. V počátcích vývoje jsem považoval za vhodné, aby bylo nad třídou vytvořeno rozhraní, které by umožnilo vytvořit paměť dat, která by byla vázána například na lokální úložiště, avšak tento návrh se neukázal jako vhodný, jelikož by se zde používaly virtuální metody, jejichž volání je oproti běžným metodám pomalejší a nelze u nich provádět inlining (vkládání samotného těla metody na místo jejího volání). Třída je implementována pomocí šablony, aby bylo možné pracovat s daty různých datových typů.

Jednotlivé bloky jsou reprezentovány objekty třídy `VolumeSegment`, která má následující rozhraní.

```
template <typename T>
class VolumeSegment
{
public:
    bool isInReloadStack, used, downscaleChecked;
    short x, y, z;
    short actualDownscale, lastRequiredDownscale futureDownscale;
    int unusedCount = 0;
    T* data = nullptr;

    VolumeSegment(short x, short y, short z);
    ~VolumeSegment();
};
```

Tato třída je taktéž implementována pomocí šablony. Data jsou uložena ve veřejné třídní proměnné `data`, další proměnné slouží především k řízení procesů pro přenačítání a odstraňování bloků. K uchovávání bloků slouží dvě instance datového typu `std::vector`, které obsahují jejich ukazatele. První z nich slouží k uchovávání bloků v nízkém rozlišeni, druhá obsahuje již bloky ve vyšší kvalitě. Bloky v nejnižší kvalitě jsou načteny již v konstruktoru paměti, bloky ve vyšší kvalitě jsou načítány dle potřeby.

Data jsou čtena prostřednictvím metody `GetValue`, která pro dané souřadnice voxelu vrací výslednou hodnotu. Pro výpočet indexu ze souřadnic je využita instrukce `_pdep_u32`, čímž je možnost nasazení této aplikace omezená na procesory *x86/64*. Výsledná funkce vypadá takto:

```
int getZCurveIndex(int_8 x, int_8 y, int_8 z){
    unsigned int mask = 0b10010010010010010010010010010001;
    return _pdep_u32(xpos, mask)
    | _pdep_u32(ypos, mask << 1)
    | _pdep_u32(zpos, mask << 2);
}
```

Instrukce *_pdep_u32* realizuje vložení jednotlivých bitů levého parametru na pozice jedniček pravého parametru. V případě potřeby nasazení na jiných typech procesorů by bylo nutné tuto funkci implementovat bez této instrukce. Jelikož dostupná kvalita dat je v plné režii paměti, metoda taktéž vrací i úroveň podškálování dat. Díky tomu může zobrazovací algoritmus přizpůsobit svůj krok.

Funkce `Revalidate` slouží k vyvolání obsluhy paměti po ukončení zobrazování daného snímku. Dochází zde ke kontrole kvality bloků, kdy v případě potřeby vyšší kvality jsou bloky přidány do zásobníku bloků k přenačtení `_volumesForReload`. Taktéž se zde v případě nedostatku paměti vybírají bloky, které nejsou potřebné, nebo mají příliš vysokou kvalitu. Konečným krokem je odstranění všech bloků, které byly k odstranění vybrány v rámci uvolňování paměti, nebo při nahrazení novějším blokem. Bloky k odstranění jsou uchovávány v poli `_volumesToDelete`, ke kterému je řízen přístup pomocí zámku `_volumesToDeleteMutex`.

Načítání bloků je zajištěno nezávislými vlákny, jejichž počet je možné uživatelsky specifikovat. Tato vlákna postupně odebírají bloky určené k přenačtení ze zásobníku, načítají potřebné bloky ze serveru a nahrazují těmito bloky bloky původní. Aby tato vlákna zbytečně neubírala výkon, nejsou-li potřebná, dochází při prázdném zásobníku k jejich uspání

na dobu sta milisekund, po kterých se provede opětovná kontrola zásobníku. Přístup k zásobníku je řízen zámkem `_reloadStackMutex`.

Data bloku jsou načítána prostřednictvím vrstvy pro přístup k webovému API. Data jsou nejdříve dekomprimována prostřednictvím knihovny *Zlib*, a poté dochází k jejich převodu z podoby pole bajtů na pole patřičného datového typu za použití korektní endianity.

Scéna

Scéna je ústřední částí zobrazování, jelikož sjednocuje všechny objekty, které do scény patří a implementuje nad nimi zobrazování metodou vrhání paprsků. Každá scéna obsahuje kameru, která poskytuje informace o daném zobrazení, a paměť volumetrických dat. Kamera vygeneruje směrový vektor paprsku pro daný pixel v perspektivní projekci, a ve scéně poté dochází k provádění vzorkování tímto paprskem. Jednotlivé vzorky jsou čteny z paměti volumetrických dat, která nezávisle zajišťuje dostupnost potřebných dat.

Scéna je implementována v rámci třídy `VolumeScene`, nad kterou je vytvořeno rozhraní `IVolumeScene`, které vypadá následovně.

```
class IVolumeScene
{
public:
    virtual void ComputeFrame(int width, int height,
        const ColorMappingTable& mappingTable) = 0;
    virtual int GetFrameWidth() = 0;
    virtual int GetFrameHeight() = 0;
    virtual bool SceneChanged() = 0;
    virtual unsigned char* GetFrame() = 0;
    virtual ~IVolumeScene() {};
};
```

Díky tomuto rozhraní lze v budoucnu snadněji integrovat do aplikace i jinou implementaci scény, například s odlišnou metodou zobrazování. Scéna povinně obsahuje metodu `ComputeFrame` pro výpočet snímku, která přijímá rozměry tohoto snímku a tabulku, která mapuje hodnoty volumetrických dat na *RGBA* barevnou hodnotu. Pro získání snímku slouží metoda `GetFrame`, která vrací ukazatel na vnitřní framebuffer (pole hodnot reprezentující jednotlivé pixely). Scéna obsahuje vždy dva framebuffery, kdy jeden může být zobrazován a s druhým třída pracuje. Jelikož uvnitř scény a v podřízených objektech dochází k některým operacím nezávisle na hlavním vláknu, scéna obsahuje metodu `SceneChanged`, pomocí které je možné zjistit, zdali došlo ke změně v rámci scény a je ji tedy nutné překreslit.

Zobrazování metodou vrhání paprsků je paralelizováno použitím vláken, kdy jejich počet je opět možné specifikovat. Každé vlákno zajišťuje zobrazení určité části snímku. Snímek však není rozdělen na blokové úseky, jelikož v při takovém postupu by vlákna nesdílely téměř žádná data, což by mělo negativní důsledek na využití vyrovnávacích pamětí. Každé vlákno namísto toho počítá pixely mezi sebou vzdálené o počet rovný celkovému počtu zobrazovacích vláken, kdy je počátek vždy posunut o index samotného vlákna. Délka výpočtu jednoho pixelu však může být odlišná (v závislosti na datech, přepínání kontextu a dalších vlivech), proto u vláken může docházet k rozcházení. Pokus o jejich synchronizaci však nevedl ke zvýšení celkového výkonu, a proto se v aplikaci již neprovádí. Pro budoucí optimalizaci by však bylo možné vytvořit systém přidělování pixelů dynamicky, což by mohlo vést k rychlejším výsledkům.

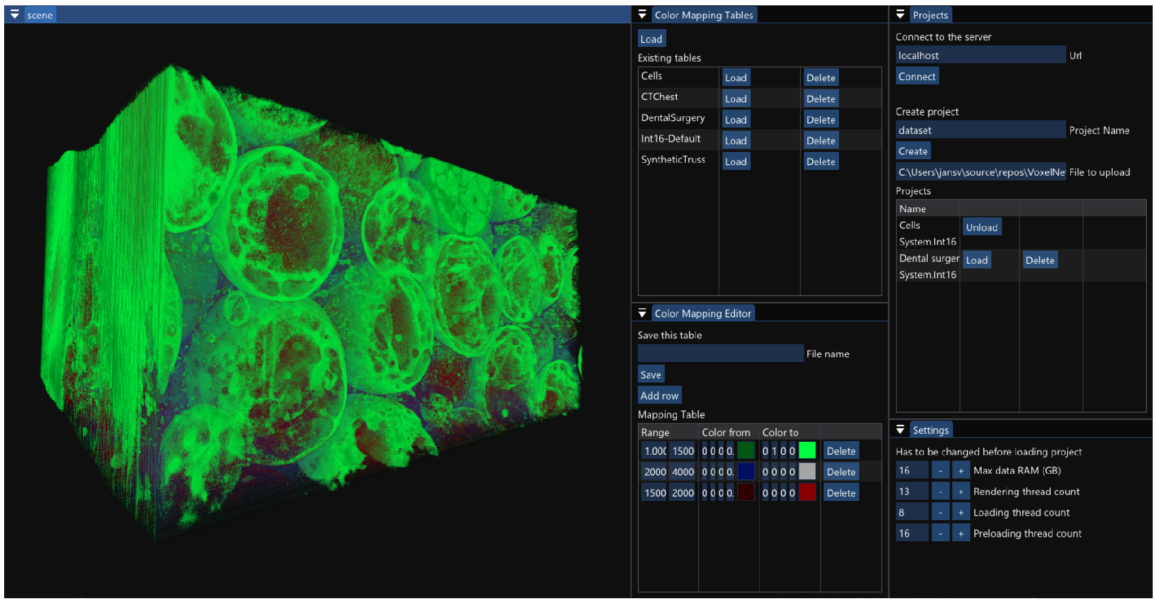
Při vrhání paprsků probíhá převod získaných vzorků na *RGBA* hodnoty, k čemuž je využita mapovací tabulka. Ta je tvořena polem objektů, kdy každý objekt definuje interval hodnot, pro který daný objekt mapuje vstupní hodnoty, a počáteční a konečnou *RGBA* hodnotu. Výstupní hodnota je pak dopočítána lineární interpolací těchto dvou *RGBA* hodnot. Mapovací tabulka je reprezentována třídou `ColorMappingTable`.

Vrstva grafického uživatelského rozhraní

Tato vrstva je založena na knihovně *Dear ImGui* ve verzi pracující nad technologií *DirectX 11*. Knihovna *Dear ImGui* obsahuje zabudovanou podporu dokovatelných oken, což umožňuje obrovskou flexibilitu při práci s grafickým uživatelským rozhraním (*GUI*). Konfigurační část, nacházející se v souboru *main.cpp*, byla vytvořena rozšířením ukázkové aplikace této knihovny, jednotlivá okna pak byla implementována do samostatných tříd obsahujících jedinou veřejnou funkci `Update`, která slouží k vyvolání sekvence pro vykreslování okna. Pro sdílení informací mezi okny byly vytvořeny třídy s příponou `Context`, které jsou sdílené mezi okny a které tyto informace uchovávají.

Jak již bylo zmíněno v úvodu kapitoly 6.2, návrhový vzor *IMGUI* pracuje na principu nekonečné smyčky, ve které dochází k neustálému překreslování jednotlivých komponent. Jelikož výpočet snímku však může trvat nějaký čas, při prostém volání metody pro generování snímku by docházelo k výraznému Zpomalování smyčky GUI. Aby uživatelské rozhraní nemuselo čekat na vytvoření snímku volumetrických dat, je jeho vytvoření vždy puštěno v nezávislém vlákně. Díky tomu zůstává hlavní zobrazovací smyčka neovlivněna, a pouze v případě dokončení práce na novém snímku dojde k jeho zobrazení. Výpočet nového snímku taktéž není spouštěn bezdůvodně, ale pouze v případě, že došlo ke změně pohledu, nebo byla změněna samotná data (například byl načten blok ve vyšší kvalitě).

Samotné zobrazování probíhá v rámci okna `VolumeViewWindow`, které navíc přijímá pohyby myši, které promítá do parametrů kamery. Otáčení kolem pevného středu probíhá pohybem myši při stisknutém levém tlačítku. Stisknutí pravého tlačítka v kombinaci s pohyby nahoru a dolů posunují střed této rotace v ose kamery. Při stisknutém kolečku myši lze pak měnit pozici středu rotace v rovině kolmé na osu kamery. Kolečko myši taktéž slouží pro přibližování a oddalování. Okno `ProjectManagementWindow` implementuje ovládání projektů na serveru a umožňuje načíst projekt. Další dvě okna slouží pro výběr a vytváření tabulek pohledů. Výběr libovolné tabulky je nutný pro zobrazení dat. Posledním oknem je okno nastavení `SettingsWindow`, které umožňuje nastavit vlastnosti aplikace, jako je maximální množství paměti dat, nebo počty vláken pro různé úkony. Obrázek 6.4 je ukázkou výsledného *GUI*.



Obrázek 6.4: Ukázka výsledného grafického uživatelského rozhraní.

Kapitola 7

Testování a vyhodnocení

Tato kapitola se zabývá testováním vytvořeného systému pro zobrazování rozsáhlých volumetrických dat. Cílem testování je nalezení nejvhodnější kombinace jednotlivých parametrů systému, a taktéž objektivní zhodnocení dosaženého výkonu.

Sekce 7.2 se zabývá experimenty, jejichž cílem je zjistit, jaký vliv má velikost bloku na výsledný systém a navhrnou nejvhodnější velikost pro další používání. Sekce 7.3 vyhodnocuje efektivitu paralelizace zobrazovacích algoritmů, jelikož na ni bylo při návrhu a implementaci vynaloženo značného úsilí. Nakonec je v sekci 7.4 vyhodnocena účinnost strategií pro načítání dat a uvolnňování paměti.

Obě části systému (server, klient) byly testovány společně na jediném zařízení, jehož parametry jsou uvedeny v tabulce 7.1. Zařízení bylo nakonfigurováno na maximální možný výkon. Experimenty probíhaly při spuštěných vývojových prostředích *Visual Studio 2022* pro obě aplikace, což mohlo mít mírně negativní dopad na výsledný výkon. Ostatní aplikace na popředí byly během měření vypnuty.

Tabulka 7.1: Vlastnosti testovacího stroje.

Operační systém	Microsoft Windows 11 Education
Procesor	AMD Ryzen 7 5800H
Disk	SAMSUNG MZVLB512HBJQ-000L2
RAM	40 GB DDR4

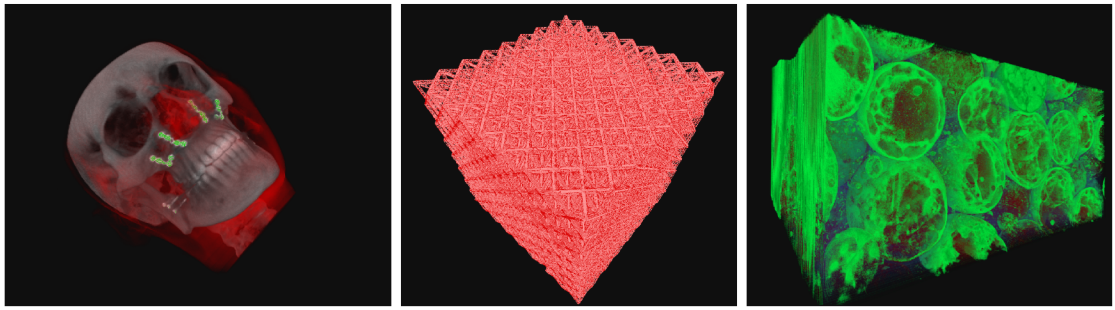
7.1 Testovací datasety

Pro účely testování byly vybrány tři různé datasety, jejichž vlastnosti jsou uvedeny v tabulce 7.2. Pro Každý z těchto datasetů byla vytvořena mapovací tabulka, která se snaží přibližně simulovat běžné využití těchto dat. Výsledná data po aplikaci mapovací tabulky jsou znázorněna v obrázcích 7.1. Tyto datasety byly čerpány z několika zdrojů, a to z knihovny nástroje 3D Slicer (viz [1, 6]), archivu Empiar (viz [11]) a webu Klacansky[1]. Některé z těchto datasetů jsou však součástí nezávislé práce a proto se zdroje v tabulce 7.2 odkazují právě na ně.

[1] https://klacansky.com/open-scivis-datasets/

Tabulka 7.2: Testovací datasety

Pracovní název	Rozsah	Datový typ	Zdroj
Dental surgery	360x360x330 (85 MB)	Int16	[1, 6]
Cells	3720x2082x464 (7,2 GB)	Int16	[21]
Synthetic truss	12000x12000x12000 (6,9 GB)	Float32	[13]



(a) Dental surgery (b) Synthetic truss (c) Cells

Obrázek 7.1: Ukázky zobrazení použitých datasetů.

7.2 Vyhodnocení vlivu velikosti bloku

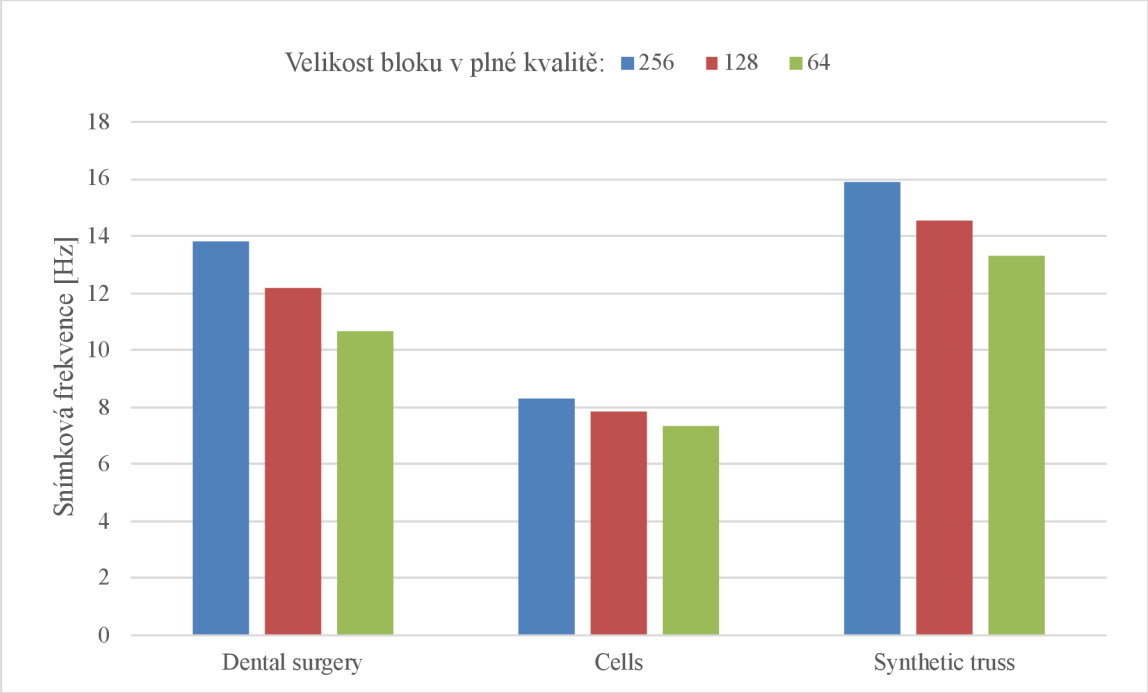
Lze předpokládat, že větší bloky budou efektivnější z hlediska zobrazování a načítání, a naopak menší bloky by měly poskytovat nižší paměťovou náročnost. Cílem těchto experimentů je tento předpoklad potvrdit, nebo vyvrátit, zjístit skutečný vliv velikosti bloku na jednotlivé aspekty systému a navrhnout nejvhodnější variantu pro další použití.

Vliv velikosti bloku na rychlost zobrazování

Experiment zkoumá, jaký vliv má velikost bloků (udávána pro blok v nejvyšší kvalitě) na rychlost samotného zobrazování. Při experimentu byly zobrazovány snímky v rozlišení 792px × 693px, kdy kamera plynule rotovala kolem scény v počáteční vzdálenosti. Při tomto experimentu byla měřena snímková frekvence (FPS) ve 45 sekundových intervalech. Výsledné hodnoty jsou zaneseny do grafu 7.2.

Experiment byl proveden až po plném načtení všech potřebných bloků v požadované kvalitě do paměti RAM. Jelikož testovací stroj disponuje poměrně vysokou kapacitou RAM paměti, bylo možné experiment takto provést i u rozsáhlejších datasetů. Aplikace při zobrazování využívala 16 zobrazovacích vláken a 8 načítacích. Po plném načtení však načítací vlákna nespotřebovávají téměř žádný výpočetní výkon. Přednačítací vlákna jsou pro výsledek nepodstatná, jelikož jsou ukončena před začátkem zobrazování.

Tento experiment by bylo možné provést i při omezené paměti dat, avšak výsledky experimentu by nebyly příliš užitečné, jelikož data, která se do paměti v potřebné kvalitě nevejdou, jsou zobrazována v nižší kvalitě, tedy celé zobrazování probíhá rychleji. Z tohoto pohledu je však zajímavý jeden z dalších experimentl v této sekci, který zkoumá vliv velikosti bloku na paměťovou náročnost.

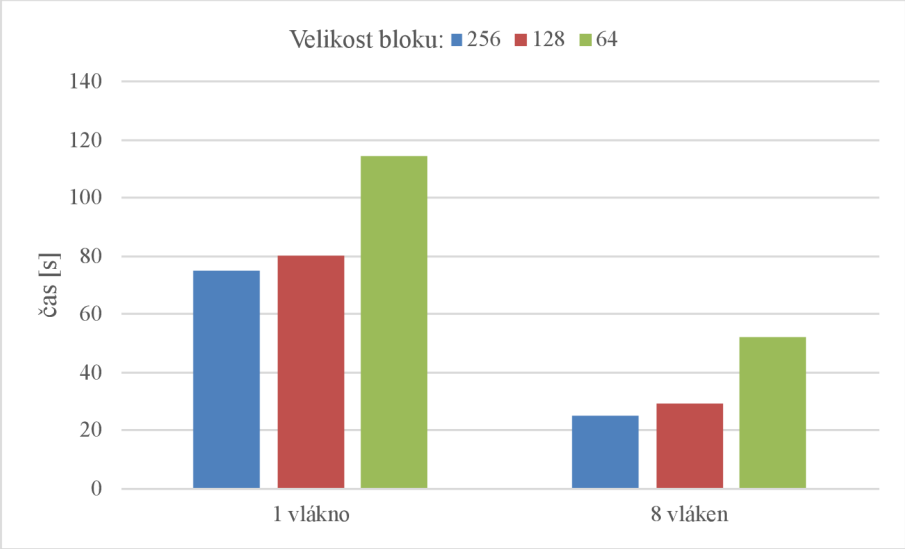


Obrázek 7.2: Snímková frekvence pro jednotlivé datasety při různých velikostech bloku.

Z grafu 7.2 je patrné, že s většími bloky je zobrazování rychlejší. Tento výsledek byl předpokládaný. Mohly by být takto testovány i větší bloky, avšak ty již neposkytují dostatečnou granualitu. Generování větších bloků je navíc poměrně náročné na operační paměť serveru.

Vliv velikosti bloku na rychlost načtení scény

Jelikož jednotlivé bloky jsou čteny ze samostatných souboru a následně jsou přenášeny po síti, jejich velikost by měla ovlivňovat rychlost načítání scény. Testování proběhlo načtením celého datasetu *Cells*, v plné kvalitě, při různých velikostech bloků. Nejdříve byl experiment proveden pouze s jedním vláknem, poté byl experiment opakován s osmi vlákny. Byl zaznamenáván čas začátku a konce načítání. Výsledné časy potřebné pro načtení scény jsou uvedeny v grafu 7.3



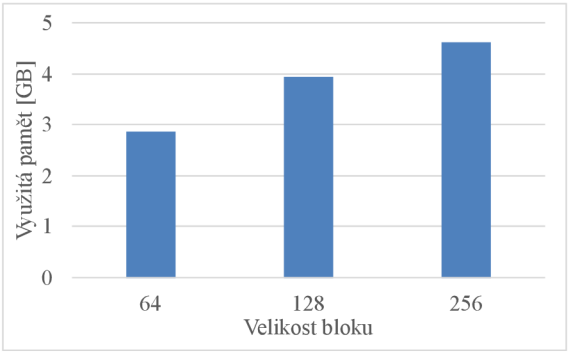
Obrázek 7.3: Rychlost načtení datasetu *Cells* v plné kvalitě při různé velikosti bloku.

Z grafu 7.3 je patrné, že dataset uložený do větších bloků je načten rychleji. Rychjelší načítání větších bloků bylo předpokládáno. U velikosti bloku 64^3 je však rozdíl velmi výrazný. Tato skutečnost je pravděpodobně způsobena větší režijí čtení a přenosu u malých souborů. Svůj vliv může mít i komprese, která je efektivnější u větších bloků. Je důležité zmínit, že bloky v nižších kvalitách jsou mnohem menší, tedy tyto jevy se na nich projeví ještě znatelněji.

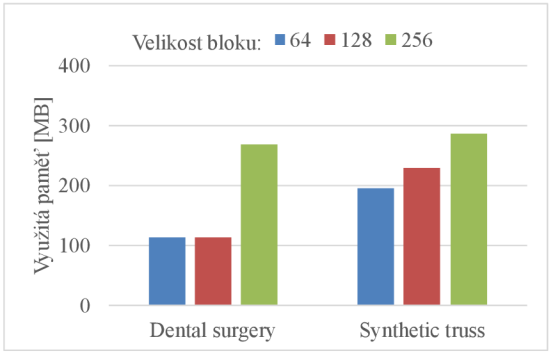
Při načítání velkého množství malých bloků po sobě (například u datasetu Cells se při velikosti bloku 64^3 jedná o 21 tisíc) server v některých případech začal dočasně reagovat blokováním požadavků. Tento problém se neprojevil při testování serveru na *WSL (Windows Subsystem for Linux)*, ale tam bylo načítání velmi pomalé. Je pravděpodobné, že tento problém je způsoben vnitřními limity serveru, které by však mělo být možné v případě potřeby navýšit. Toto množství bloků je však příliš vysoké a takto malé bloky tedy nesjou pro načítání příliš vhodné.

Vliv velikosti bloku na paměťovou náročnost

Menší bloky by měly poskytovat vyšší granualitu a tedy i vyšší efektivitu strategií pro načítání dat. U malých datasetů by se navíc mělo výrazně projevovat doplnění dat v každé ose na násobek velikosti bloku. Experiment byl proveden načtením scény v požadované kvalitě ve výchozím pohledu. Jeho výsledky jsou znázorněny v grafech 7.4.



(a) Využití paměti při zobrazení datasetu *Cells* při různých velikostech bloku.



(b) Využití paměti při zobrazení ostatních datasetů při různých velikostech bloku.

Obrázek 7.4: Využití paměti při různých velikostech bloku.

Z grafů je patrné, že předpoklad byl správný a menší bloky jsou opravdu z hlediska paměťové náročnosti efektivnější. Nejvýraznější rozdíl je u datasetu *Dental surgery*, avšak zde je způsoben zarovnáním dat do jednotlivých bloků, jelikož dataset je velmi malý. Pro velikosti bloku 64^3 a 128^3 má hodnoty shodné, jelikož po zarovnání jsou v obou případech data stejně velká a při daném zobrazení je tento dataset načten vždy celý.

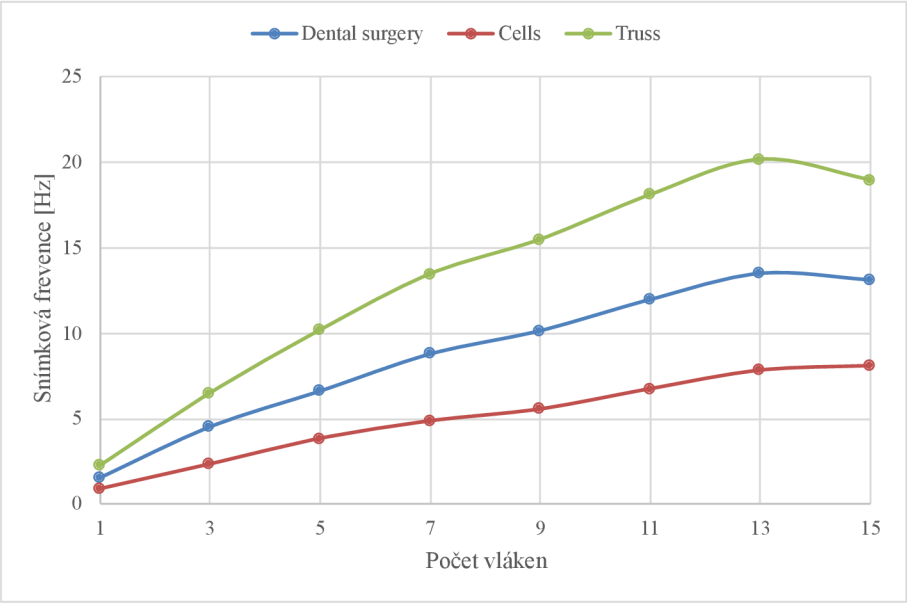
Závěr

Bylo potvrzeno, že větší bloky jsou efektivnější z hlediska načítání a zobrazování, avšak z hlediska paměťové náročnosti jsou menší bloky výhodnější. Je třeba také vzít v potaz velikost bloků v nižších kvalitách. Nelze tedy jednoznačně určit nejvhodnější velikost bloku, avšak vhodně se jeví velikosti 256^3 a 128^3. Větší bloky jsou vhodnější při požadavcích na vyšší zobrazovací výkon, menší bloky jsou naopak efektivnější z hlediska paměťové náročnosti. V dalších experimentech bude využito velikosti bloku 256^3.

7.3 Vyhodnocení vlivu počtu zobrazovacích vláken

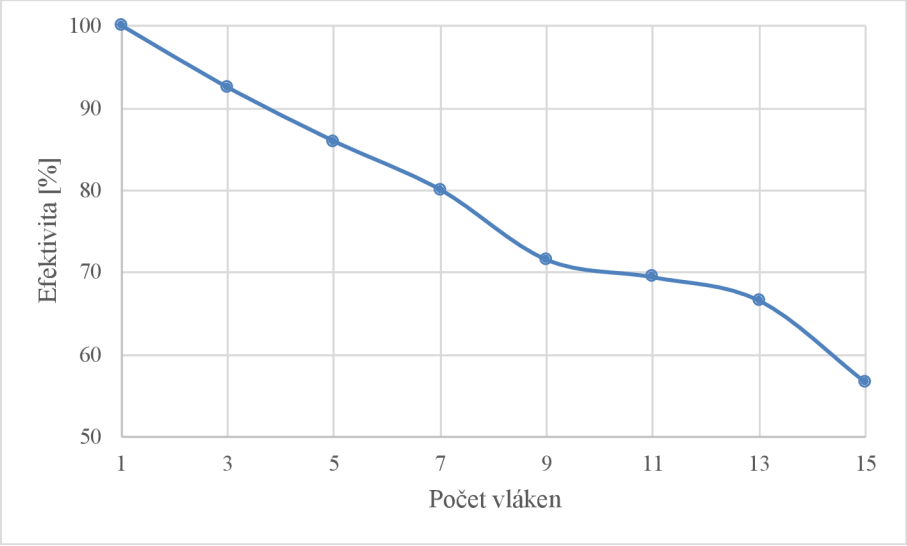
Aby byl výkon aplikace co nejvyšší, probíhá zobrazování na několika vláknech. Při implementaci byl kladen důraz na efektivní přístup do sdílené paměti ve výkonnostně kritických částech, je tedy vhodné zjistit, jak byla tato snaha efektivní.

Experiment probíhal při plynulé rotaci kamery kolem scény v počáteční vzdálenosti, při různých počtech zobrazovacích vláken, s konstantní velikostí bloku 256^3 (pro blok v nejvyšší kvalitě) a při rozlišení generovaného pohledu 792px × 693px. Výsledky jsou zobrazeny v grafu 7.5.



Obrázek 7.5: Snímková frekvence při velikosti bloku 256^3 (pro blok v nejvyšší kvalitě) pro různé počty zobrazovacích vláken.

Z výsledků je patrné, že nejvyššího výkonu je na testovacím stroji dosaženo při třinácti zobrazovacích vláknech a při vyšším počtu již více nestoupá. V ideálním případě by byla snímková frekvence při počtu N vláken N-násobná oproti zobrazování jedním vláknem. Pro každý počet vláken a každý dataset byl tedy vypočten podíl skutečné a ideální snímkové frekvence, a následně byly tyto podíly zprůměrovány do výsledných hodnot, které jsou v procentech zobrazeny v grafu 7.6.



Obrázek 7.6: Efektivita zobrazování více vlákny, vypočtena jako podíl skutečné a ideální snímkové frekvence pro daný počet vláken vyjádřený v procentech.

Při zobrazování třemi vlákny dosahuje efektivita přes 90% a pro všechny kombinace nikdy neklesá pod 55%, což značí, že snaha byla úspěšná. Důležitým výsledkem jsou i samotné

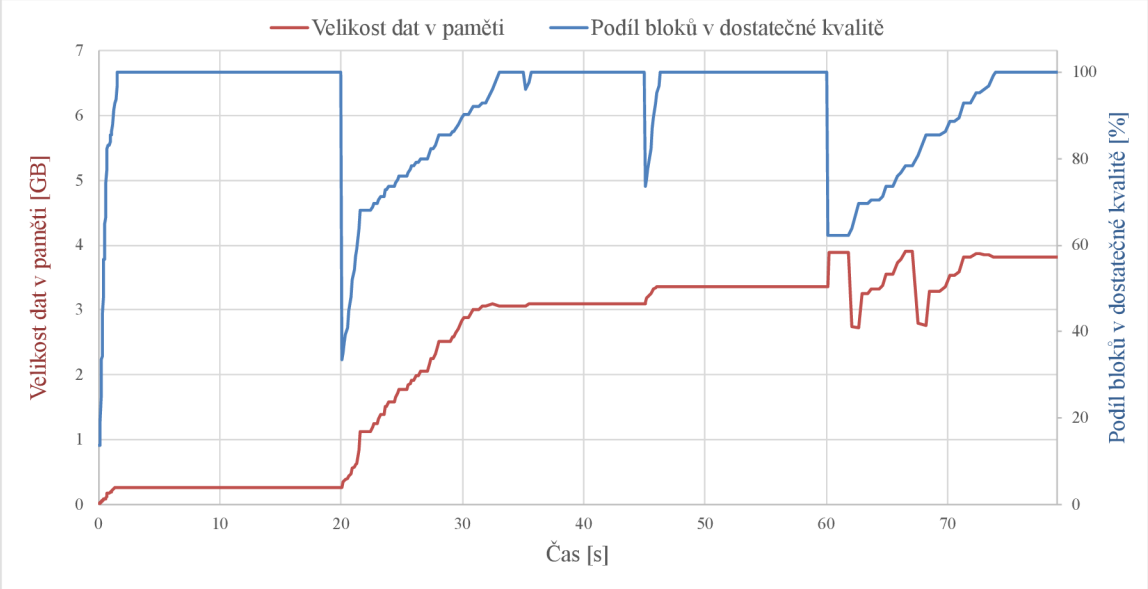
hodnoty snímkové frekvence, které jsou dostatečně vysoké pro zobrazování dat v reálném čase.

7.4 Účinnost strategií pro načítání dat a uvolňování paměti

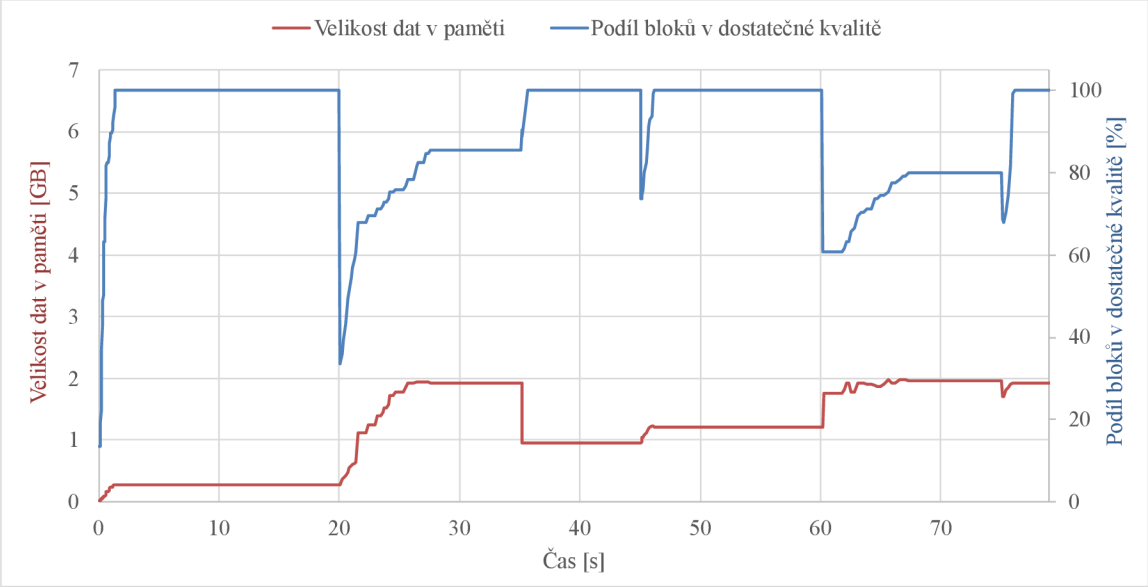
Jelikož v rámci aplikace byly vytvořeny strategie pro uvolňování paměti a načítání dat, je vhodné ověřit jejich účinnost. Experiment byl proveden zobrazováním datasetu *Synthetic Truss* při nízkých maximálních velikostech paměti dat, kdy byly prováděny následující úkony:

- Přiblížení scény (20. sekunda)
- Oddálení scény (35. sekunda)
- Otočení scény o 90 stupňů kolem osy y (45. sekunda)
- Přiblížení scény (60. sekunda)
- Oddálení scény (75. sekunda)

Experiment byl opakován při maximální velikosti dat 4 GB a 2 GB. Byla zaznamenávána jak velikost v paměti obsažených dat, tak i podíl bloků, které mají pro aktuální zobrazení dostatečnou kvalitu. Výsledky jsou zaznamenány do grafů 7.7 a 7.8. Velikost bloku byla 256^3.



Obrázek 7.7: Velikost paměti dat a podíl bloků v dostatečné kvalitě při maximální velikosti paměti dat 4GB.



Obrázek 7.8: Velikost paměti dat a podíl bloků v dostatečné kvalitě při maximální velikosti paměti dat 2GB.

Z experimentu je patrné, že v případě maximální velikosti dat 4 GB bylo vždy možné scénu donačíst do potřebné kvality. U velikosti bloku 2 GB již nastávaly situace, kdy některé části scény nebylo možné vždy zobrazit v potřebné kvalitě. Z výsledných křivek je zřejmé, že strategie pro uvolňování paměti a načítání dat fungují korektně. Dle subjektivního zhodnocení zobrazení by bylo přínosné do systému implementovat přednostní načítání důležitých bloků, což by mohlo poskytnout lepší výsledné zobrazení při stejné maximální velikosti paměti dat.

Kapitola 8

Závěr

Cílem práce byl návrh a implementace systému, který umožní zobrazování rozsáhlých volumetrických dat na běžném počítači v reálném čase. Tohoto cíle bylo dosaženo realizací řešení pomocí klient-server architektury, kdy server, realizovaný na technologii *ASP.NET Core* slouží jako úložiště volumetrických dat rozdělených do menších bloků v různých kvalitách. Klient, realizovaný v jazyce *C++* s grafickou knihovou Dear ImGui, tato data zobrazuje metodou vrhání paprsků na CPU.

Server realizuje konverzi vstupního souboru na jednotlivé bloky, které jsou uchovávány jako samostatné soubory v rámci jeho úložiště a disponuje webovým API, pomocí kterého je možné k datům přistupovat a spravovat je. Klientská aplikace tato data ve formě bloků načítá dle vytvořených strategií do lokální paměti tak, aby bylo možné provádět zobrazení scény v přijatelné kvalitě. Taktéž zde byly implementovány strategie pro uvolňování paměti při jejím nedostatku. Data jsou v klientské aplikaci serializována pomocí Mortonovy křivky, aby byl optimalizován přístup do rychlých vyrovnávacích pamětí procesoru. Zobrazování metodou vrhání paprsků bylo optimalizováno pomoci algoritmu DDA, předčasného ukončení paprsku a přizpůsobení délky kroku tak, aby byl výsledný výkon aplikace co možná nejvyšší, i za cenu horších vizuálních výsledků. Dále byl navržen a implementován postup, kdy je při zobrazování celá scéna transformována do jednotkové voxelové mřížky. Pro účely vysokého výkonu byl taktéž kladen značný důraz na paralelizaci.

Při experimentech bylo zjištěno, že zobrazování i načítání dat je optimálnější pro větší bloky, menší bloky mají zase nižší paměťovou náročnost. Toto zjištění bylo v souladu s předpokladem. Výběr vhodné velikosti bloku tedy záleží na prioritách při výsledném použití. Dále bylo experimentálně potvrzeno, že paralelizace zobrazování byla úspěšná, jelikož při třech zobrazovacích vláknech na daném stroji její efektivita přesahovala 90% a i při větších počtech vláken neklesla pod 55%. Účinné se taktéž ukázaly i strategie pro načítání dat a uvolňování paměti.

Při dalším vývoji by bylo vhodné rozšířit podporu dalších vstupních formátů volumetrických dat, aby bylo možné systém používat snadněji. Pro účely ukládání metadat by bylo vhodné zvážit využití relační databáze, jelikož ta lépe řeší souběžný přístup více uživatelů. Taktéž by bylo možné implementovat další optimalizace, jako je přeskakování prázdných prostorů, nebo by mohl být navrhnuty strategie načítání dat, které pracují s prioritou. Dále se zde nabízí implementace množství dalších rozšíření, jako je možnost provádět řezy, případně umožnit i různé kvality zobrazení.

Literatura

[1] *3D Slicer* [online]. Slicer Community, ©2022. 2022-11-22 [cit. 2023-04-09]. Dostupné z: https://www.slicer.org/.

[2] Callahan, S., Bavoil, L., Pascucci, V. a Silva, C. Progressive Volume Rendering of Large Unstructured Grids. *IEEE transactions on visualization and computer graphics*. New York: IEEE. Září–Říjen 2006, sv. 12, č. 5, s. 1307–1314. DOI: 10.1109/TVCG.2006.171. ISSN 1077-2626.

[3] Campoalegre, L. a Dingliana, J. *A Survey of Client-Server Volume Visualization Techniques* [online]. School of Computer Science and Statistics, Trinity College Dublin, leden 2018 [cit. 2023-04-09]. Dostupné z: https://www.scss.tcd.ie/publications/tech-reports/2018/TCD-SCSS-TR-2018-01.pdf.

[4] Dai, H. K. a Su, H. C. On the Locality Properties of Space-Filling Curves. In: Ibaraki, T., Katoh, N. a Ono, H., ed. *Algorithms and Computation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, s. 385–394. DOI: 10.1007/978-3-540-24587-2_40. ISBN 978-3-540-20695-8.

[5] Deutsch, L. P. *GZIP file format specification version 4.3* [RFC 1952]. RFC Editor, květen 1996 [cit. 2023-04-04]. DOI: 10.17487/RFC1952. Dostupné z: https://doi.org/10.17487/RFC1952.

[6] Fedorov, A., Beichel, R., Kalpathy Cramer, J., Finet, J., Fillion Robin, J.-C. et al. 3D Slicer as an image computing platform for the Quantitative Imaging Network. *Magnetic Resonance Imaging*. Elsevier Inc. 2012, sv. 30, č. 9, s. 1323–1341, [cit. 2023-04-09]. DOI: 10.1016/j.mri.2012.05.001. ISSN 0730-725X. Quantitative Imaging in Cancer.

[7] Franco, P., Nguyen, G., Mullot, R. a Ogier, J.-M. Alternative patterns of the multidimensional Hilbert curve: Application in image retrieval. *Multimedia tools and applications*. New York: Springer US. 2018, sv. 77, č. 7, s. 8419–8440. DOI: 10.1007/s11042-017-4744-4. ISSN 1380-7501.

[8] Gutenko, I., Petkov, K., Papadopoulos, C., Zhao, X., Park, J. H. et al. Remote volume rendering pipeline for mHealth applications. In: *Progress in Biomedical Optics and Imaging – Proceedings of SPIE*. SPIE, 2014, sv. 9039, s. 903904–903904–7. DOI: 10.1117/12.2043946. ISBN 9780819498328.

[9] Hadwiger, M., Al Awami, A. K., Beyer, J., Agus, M. a Pfister, H. SparseLeap: Efficient Empty Space Skipping for Large-Scale Volume Rendering. *IEEE transactions on visualization and computer graphics*. LOS ALAMITOS: IEEE. Leden 2018, sv. 24, č. 1, s. 974–983. DOI: 10.1109/TVCG.2017.2744238. ISSN 1077-2626.

[10] Hilton, J. Dynamic modelling of radiant heat from wildfires. In: *Proceedings – 22nd International Congress on Modelling and Simulation* [online]. Modelling and Simulation Society of Australia and New Zealand Inc., Prosinec 2017, s. 1104–1110 [cit. 2023-04-09]. Dostupné z: https://www.researchgate.net/publication/329503247_Dynamic_modelling_of_radiant_heat_from_wildfires.

[11] Iudin, A., Korir, P. K., Somasundharam, S., Weyand, S., Cattavitello, C. et al. EMPIAR: the Electron Microscopy Public Image Archive. *Nucleic Acids Research*. England: Oxford University Press. Leden 2023, sv. 51, D1, s. D1503–D1511. DOI: 10.1093/nar/gkac1062. ISSN 0305-1048.

[12] Jönsson, D., Sundén, E., Ynnerman, A. a Ropinski, T. A Survey of Volumetric Illumination Techniques for Interactive Volume Rendering. *Computer graphics forum*. Oxford: Blackwell Publishing Ltd. Únor 2014, sv. 33, č. 1, s. 27–51. DOI: 10.1111/cgf.12252. ISSN 0167-7055.

[13] Klacansky, P., Miao, H., Gyulassy, A., Townsend, A., Champley, K. et al. Virtual Inspection of Additively Manufactured Parts. [online]. Los Alamitos, CA, USA: IEEE Computer Society. Duben 2022, s. 81–90, [cit. 2023-04-09]. DOI: 10.1109/PacificVis53943.2022.00017. Dostupné z: https://doi.ieeecomputersociety.org/10.1109/PacificVis53943.2022.00017.

[14] Lacroute, P. G. *Fast volume rendering using a shear-warp factorisation of the viewing transformation*. 1995. Disertační práce. Computer Systems Laboratory Departments of Electrical Engineering and Computer Science Stanford University Stanford.

[15] Lalgudi, H. G., Marcellin, M. W., Bilgin, A. a Nadar, M. S. Scalable low complexity image coder for remote volume visualization. In: *Proceedings of SPIE*. Bellingham, Wash: SPIE, 2008, sv. 7073, č. 1, s. 707317–707318. DOI: 10.1117/12.794430. ISBN 9780819472939.

[16] Mueller, K. a Kaufman, A. E. Volume Visualization in Medicine. In: *Handbook of Medical Image Processing and Analysis*. 2nd Edition. Elsevier, 2009, s. 785–815. DOI: 10.1016/B978-012373904-9.50057-X. ISBN 9780123739049.

[17] Mwalongo, F., Krone, M., Reina, G. a Ertl, T. Web-based Volume Rendering using Progressive Importance-based Data Transfer. In: Beck, F., Dachsbacher, C. a Sadlo, F., ed. *Vision, Modeling and Visualization* [online]. The Eurographics Association, 2018 [cit. 2023-04-09]. DOI: 10.2312/vmv.20181264. ISBN 978-3-03868-072-7. Dostupné z: https://doi.org/10.2312/vmv.20181264.

[18] Nguyen, K. G. a Saupe, D. Rapid High Quality Compression of Volume Data for Visualization. *Computer graphics forum*. Oxford, UK and Boston, USA: Blackwell Publishers Ltd. 2001, sv. 20, č. 3, s. 49–57. DOI: 10.1111/1467-8659.00497. ISSN 0167-7055.

[19] Nguyen, K. G. a Saupe, D. Rapid High Quality Compression of Volume Data for Visualization. *Computer graphics forum*. Oxford, UK and Boston, USA: Blackwell Publishers Ltd. 2001, sv. 20, č. 3, s. 49–57. DOI: 10.1111/1467-8659.00497. ISSN 0167-7055.

[20] Schlegel, P. a Pajarola, R. Layered Volume Splatting. In: *Advances in Visual Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, sv. 5876, č. 2, s. 1–12. Lecture Notes in Computer Science. DOI: 10.1007/978-3-642-10520-3_1. ISBN 9783642105197.

[21] Seeger, C., Dyrhage, K., Mahajan, M., Odelgard, A., Lind, S. B. et al. The Subcellular Proteome of a Planctomycetes Bacterium Shows That Newly Evolved Proteins Have Distinct Fractionation Patterns. *Frontiers in Microbiology* [online]. Květen 2021, sv. 12, [cit. 2023-04-09]. DOI: 10.3389/fmicb.2021.643045. ISSN 1664-302X. Dostupné z: https://www.frontiersin.org/articles/10.3389/fmicb.2021.643045.

[22] Sugimoto, Y., Ino, F. a Hagihara, K. Improving cache locality for GPU-based volume rendering. *Parallel computing*. AMSTERDAM: Elsevier B.V. 2014, sv. 40, 5-6, s. 59–69. DOI: 10.1016/j.parco.2014.03.013. ISSN 0167-8191.

[23] Wald, I., Johnson, G., Amstutz, J., Brownlee, C., Knoll, A. et al. OSPRay – A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics*. LOS ALAMITOS: IEEE. Leden 2017, sv. 23, č. 1, s. 931–940. DOI: 10.1109/TVCG.2016.2599041.

Příloha A

Obsah paměťového média

- **datasets/** – soubory s několika malými datasety pro testování předzpracování
- **doc/** – zdrojové soubory technické zprávy
- **bin/client/** – přeložená klientská aplikace
- **bin/server/** – přeložená serverová aplikace
- **src/** – zdrojové soubory systému
- **storage/** – úložiště serveru obsahující již předzpracované ukázkové datasety
- **poster.pdf** – vytvořený plakát
- **thesis.pdf** – technická zpráva
- **thesis-print.pdf** – technická zpráva ve verzi pro tisk
- **README.md** – návod na spuštění, ovládání a překlad ze zdrojových souborů