Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Zobrazovanie viacrozmerných fraktálov

Bakalárska práca

2024
Adrián Kocifaj

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Zobrazovanie viacrozmerných fraktálov

Bakalárska práca

Študijný program:	Aplikovaná informatika
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra aplikovanej informatiky
Školiteľ:	Mgr. Andrej Mihálik, PhD.

Bratislava, 2024
Adrián Kocifaj

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Adrián Kocifaj
Študijný program: aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Zobrazovanie viacrozmerných fraktálov.
Hypercomlex fractal rendering.

Anotácia: Nie je jednoduché zobraziť objekty v priestore s dimenziou viac ako 3. Keď však uvážime pevný 3D objekt, tak okrem rozmerov v 3D priestore, vykazuje jeho vzhľad aj ďalšie atribúty ako je napríklad farba alebo priehľadnosť. S použitím volumetrického zobrazovania by sa takto dal priestor rozšíriť za hranicu 3D.

Cieľ: Vytvoriť rozhranie na vizualizáciu viacrozmerného fraktálu pomocou volumetrického zobrazovania.

Literatúra: Alan Norton, Generation and Display of Geometric Fractals in 3-D, ACM SIGGRAPH Computer Graphics, vol. 16, no. 3, pp. 61–67, 1982.

Vedúci: Mgr. Andrej Mihálik, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: doc. RNDr. Tatiana Jajcayová, PhD.

Dátum zadania: 15.10.2018

Dátum schválenia: 09.11.2023 doc. RNDr. Damas Gruska, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie: Rád by som touto cestou vyjadril úprimné poďakovanie svojmu školiteľovi Mgr. Andrejovi Mihálikovi, PhD., za jeho odborné vedenie, cenné rady a trpezlivosť počas celého procesu tvorby tejto práce.

Abstrakt

Táto práca sa zameriava na vizualizáciu štvorrozmerných fraktálov v trojrozmernom priestore, pričom štvrtú dimenziu reprezentuje farba. Tieto fraktály sú definované nad hyperkomplexnými číslami, kvaternionmi. V práci sme analyzovali teoretické základy fraktálov, kvaterniónov a rôzne zobrazovacie techniky. Navrhli a implementovali sme softvérový systém, ktorý využíva moderné technológie. Vyvinutý softvérový systém poskytuje používateľom intuitívne rozhranie na manipuláciu s fraktálmi, výber zobrazovacej techniky, ovládanie kamery a renderovanie obrázkov vo vysokom rozlíšení.

Kľúčové slová: fraktál, kvaternion, štvrtá dimenzia, farba

Abstract

This work focuses on the visualization of four-dimensional fractals in three-dimensional space, with the fourth dimension represented by color. These fractals are defined over hypercomplex numbers, quaternions. In this paper we have analyzed the theoretical foundations of fractals, quaternions and various imaging techniques. We have designed and implemented a software system that uses modern technology. The developed software system provides users with an intuitive interface for manipulating fractals, selecting imaging techniques, controlling the camera, and rendering high-resolution images.

Keywords: fractal, quaternion, fourth dimension, colour

Obsah

Zoznam obrázkov

Zoznam tabuliek

Úvod

Vo svete, kde technológia neustále prekračuje hranice možného, sa otvárajú nové obzory pre prieskum doteraz len málo preskúmaných oblastí. Jednou z takýchto oblastí sú fraktály, nekonečné a úchvatné štruktúry, ktoré stoja na rozmedzí medzi chaotickým a usporiadaným správaním. Tieto štruktúry, považované nielen za matematické záhady, ale aj za umelecké diela, majú obrovský potenciál v mnohých vedeckých oblastiach. Vďaka rýchlemu vývoju moderných techník vizualizácie a hardvérových riešení môžeme tieto objekty nielen presne analyzovať, ale aj vizuálne preskúmavať, čo otvára nové možnosti pre ich výskum a praktické využitie.

Cieľom tejto práce je vyvinúť sofistikovaný softvérový systém na vizualizáciu štvorrozmerných fraktálov. Vďaka reprezentácii štvrtej dimenzie ako farby, môžeme zobraziť štvorrozmerné fraktály v 3D prostredí. Systém používa rôzne techniky počítačovej grafiky a vizualizačných algoritmov na dosiahnutie interaktívnych zobrazení. Systém je navrhnutý tak, aby umožnil užívateľom nielen pasívne pozorovanie, ale aj aktívne experimentovanie s rôznymi parametrami fraktálov, čo podporuje hlbšie pochopenie ich štruktúry.

Táto práca je štrukturovaná do niekoľkých kľúčových kapitol. Prvá kapitola poskytuje prehľad teoretických základov fraktálov a hyperkomplexných čísel, kvaternionov, pomocou ktorých sú tieto fraktály definované v štvrtej dimenzii. Druhá kapitola sa podrobne venuje návrhu a architektúre vizualizačného softvéru. V tretej kapitole prechádzame implementáciu systému, techník zobrazovania a rôznych optimalizácií. Štvrtá kapitola sa zameriava na porovnanie zobrazovacích techník a diskusiu o výsledkoch, ku ktorým sme v priebehu práce dospeli.

Veríme, že táto práca prispeje k rozšíreniu poznatkov v oblasti vizualizácie viacrozmerných fraktálov a poskytne hodnotný nástroj pre výskumníkov a nadšencov, ktorí sa zaujímajú o túto fascinujúcu oblasť.

Kapitola 1

Východiská práce

1.1 Fraktály

Fraktály, označené počas rozvoja teórie chaosu ako "matematické monštrá", sú nekonečne detailné matematické štruktúry, ktoré majú detaily na ľubovoľne malých mierkach. Nedajú sa popísať pomocou klasických geometrických termínov, pretože ich tvary sú nepravidelné a zvyčajne vykazujú určitú mieru sebapodobnosti [5]. Je možné ich popísať pomocou iteratívnych metód, ktoré spočívajú v opakovanom aplikovaní rovnakých matematických pravidiel na výsledky predchádzajúcich krokov.

Napriek tomu, že boli pôvodne považované za "matematické monštrá", sú vlastnosti fraktálov v skutočnosti typické pre prírodu a stali sa nevyhnutnou súčasťou pri modelovaní a simulácii prírodných javov [16].

1.1.1 Mandelbrotova množina

Mandelbrotova množina je pomenovaná po matematikovi Benoîtovi Mandelbrotovi a je to jeden z najznámejších fraktálov, ktorý vzniká iterovaním komplexnej kvadratickej funkcie. Definícia Mandelbrotovej množiny je určená rovnicou

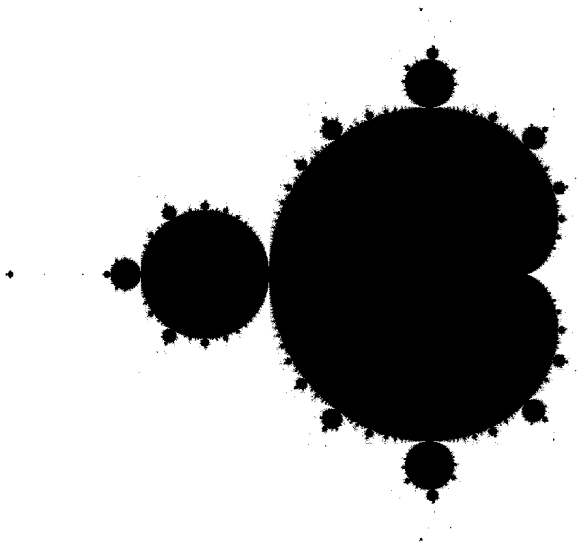
$$z_{n+1} = z_n^2 + c,$$

kde z a c sú komplexné čísla so začínajúcim členom postupnosti $z_0 = 0 + 0i$. Komplexné číslo c patrí do množiny, ak veľkosť postupnosti z_n konverguje [5]. Vizualizáciu Mandelbrotovej množiny môžeme vidieť na obrázku 1.1.

1.1.2 Juliina množina

Rovnako ako Mandelbrotova množina, definícia Juliinej množiny je definovaná iterovaním rovnakej komplexnej kvadratickej funkcie

$$z_{n+1} = z_n^2 + c,$$



Obr. 1.1: Mandelbrotova množina. Komplexné čísla c sú reprezentované bodmi v rovine, čierna farba znázorňuje body patriace do množiny.

kde z a c sú komplexné čísla s ľubovoľnou konštatou c. Komplexné číslo z patrí do množiny, ak veľkosť postupnosti z_n konverguje [5]. Vizualizáciu Juliinej množiny môžeme vidieť na obrázku 1.2.

1.2 Kvaterniony

Kvaterniony sú hyperkomplexné čísla skladajúce sa zo štyroch komponentov. Kvaternion q bol prvýkrát definovaný Williamom Rowanom Hamiltonom ako

$$q = a + bi + cj + dk,$$

kde a, b, c, d patria do množiny reálnych čísel, a zložky i, j, k sú základné jednotky kvaternionov s týmito vlastnosťami:

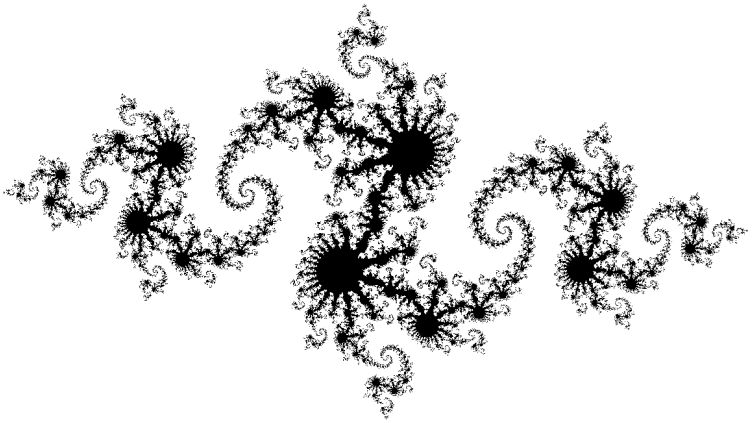
$$i^2 = j^2 = k^2 = ijk = -1,$$

$$ij = k, \quad ji = -k, \quad jk = i, \quad kj = -i, \quad ki = j, \quad ik = -j,$$

týmto Hamilton zaviedol nový matematický koncept rozširujúci tradičné komplexné čísla do štvorrozmerného priestoru [15].

1.2.1 Operácie s kvaternionmi

Nasledujúce operácie kvaternionov sú podrobne definované pomocou algebraických operáciií [15].



Obr. 1.2: Juliina množina. Komplexné čísla z sú reprezentované bodmi v rovine, čierna farba znázorňuje body patriace do množiny pre $c = -0.8 + 0.156i$.

Kvaterninoy sa sčítavajú a odčítavajú po zložkách, pre kvateriony q a q' je definované sčítanie a odčítanie ako

$$q + q' = (a + a') + (b + b')i + (c + c')j + (d + d')k,$$
$$q - q' = (a - a') + (b - b')i + (c - c')j + (d - d')k.$$

Násobenie sa prevádza distributívnym zákonom, pričom sa využívajú pravidlá násobenia zložiek i, j, k. Násobenie kvaterionov nie je komutatívne, pre kvateriony q a q' je definované ako

$$\begin{aligned} qq' = &(aa' - bb' - cc' - dd') \\ &+ (ab' + ba' + cd' - dc')i \\ &+ (ac' - bd' + ca' + db')j \\ &+ (ad' + bc' - cb' + da')k. \end{aligned}$$

Konjugát alebo doplnok kvaternionu $q = a+bi+cj+dk$ je definovaný ako kvaternion

$$q^* = a - bi - cj - dk.$$

Veľkosť alebo norma kvaternionu je definovaná ako

$$|q| = \sqrt{a^2 + b^2 + c^2 + d^2}.$$

Inverzný kvaternion q^{-1} pre nenulový kvaternion q je definovaný ako

$$q^{-1} = \frac{q^*}{|q|^2}.$$

1.2.2 Kvaterniony a fraktály

Fraktály vznikajúce iterovaním komplexných rovníc, ako sú Mandelbrotova množina či Juliina množina, môžeme rozšíriť nahradením komplexných čísel kvaternionmi. Tento postup rozširuje dvojrozmerné fraktály na štvorrozmerné. Výsledné fraktály nemôžu byť priamo graficky reprezentované, nakoľko ľudské vnímanie je obmedzené na tri dimenzie, a preto musí byť štvrtá dimenzia reprezentovaná iným spôsobom [7].

1.3 Osvetlenie

Pri vizualizácii trojrozmerných štruktúr, môžeme využiť osvetlenie na zvýraznenie a lepšie pochopenie tretej dimenzie (hĺbky) a priestorového usporiadania. Osvetlenie simuluje interakciu svetla s povrchom štruktúry a vytvára realistickejší vizuálny dojem. Útlm svetla popisuje jav, kedy intenzita svetla klesá so vzdialenosťou od zdroja, vďaka tomu pôsobia vzdialenejšie časti štruktúry tmavšie. Najbežnejšie sa používa kvadratický model útlmu, ktorý je matematicky vyjadrený rovnicou

$$I(d) = \frac{I_0}{1 + a \cdot d + b \cdot d^2},$$

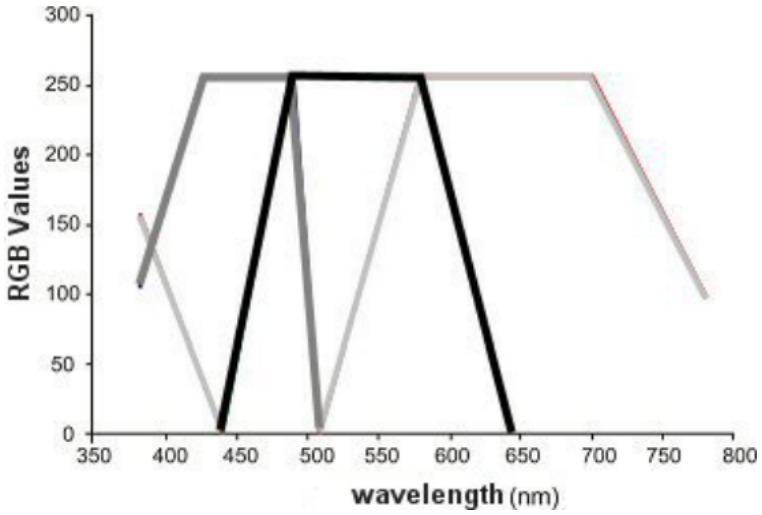
kde $I(d)$ označuje intenzitu svetla vo vzdialenosti d od zdroja, I_0 je počiatočná intenzita svetla a koeficienty a a b určujú mieru lineárneho a kvadratického útlmu [2].

1.4 Reprezentácia viacrozmerných dát

Viacrozmerné dáta môžeme reprezentovať v 3D priestore, pričom rozmery nad treťou dimenziou zobrazíme alternatívnymi spôsobmi. Jedným zo spôsobov je použitie farby, kde sú ďalšie dimenzie mapované na viditeľné farebné spektrum, teda na vlnové dĺžky, ktoré sú vnímané ľudským okom.

Transformácia vlnových dĺžok na RGB farebný model je kľúčová pre presnú vizualizáciu. Je nevyhnutné, aby táto premena bola dostatočne presná, aby sme čo najlepšie napodobnili realitu a vnímanie farieb ľudským okom.

Algoritmus tejto transformácie zahŕňa aditívne miešanie základných farebných zložiek, červenej, zelenej a modrej, na dosiahnutie rôznych odtieňov. Tento prístup využíva lineárne aproximácie CIE farebných funkcií zobrazených na obrázku 1.3, ktoré zabezpečujú efektívnu transformáciu farebných hodnôt [14]. Keďže sú tieto funkcie aproximované, výsledky nie sú úplne presné, ale sú dostatočne blízke reálnym farbám a použiteľné v reálnom čase. Táto vlastnosť je kľúčová pri spracovaní dát, kde je rýchlosť nevyhnutná.



Obr. 1.3: Lineárne aproximácie CIE farebných funkcií [14].

1.5 Kamera v 3D priestore

Manipulácia s kamerou v 3D priestore je kľúčovým prvkom pri tvorbe vizualizačných a interaktívnych aplikácií, pretože umožňuje používateľom efektívnejšie vnímať trojrozmerný priestor aj napriek jeho zobrazeniu na dvojrozmernom displeji. Kamera je v 3D priestore definovaná bodom, kde sa nachádza, a tromi vektormi, ktoré určujú jej orientáciu: *front* vektor určuje smer, kam sa kamera pozerá, *up* vektor definuje smer nahor, *right* vektor určuje smer vpravo.

Ovládanie rotácie kamery je zabezpečené pomocou Eulerových uhlov: *yaw*, *pitch* a *roll*. *Yaw* zabezpečuje rotáciu kamery okolo vertikálnej osi y, *pitch* riadi rotáciu okolo horizontálnej osi x a *roll* kontroluje rotáciu okolo osi z [6].

1.6 Volumetrické zobrazovanie

Volumetrické zobrazovanie je technika využívaná na vizualizáciu objemových dát, ktoré obsahujú informácie nielen o povrchu objektov, ale aj o ich vnútorných štruktúrach [8]. Táto metóda nám umožňuje pozerať sa na objekty iným spôsobom, študovať ich vnútornú charakteristiku a poskytuje pohľad na vnútorné detaily. Pri použití je dôležité mať reprezentáciu dát vo forme objemových prvkov, voxelov.

1.6.1 Voxel

Voxel je ekvivalentom pixelu v 2D obrazoch a predstavuje malý elementárny objem v priestore. Každý voxel je definovaný svojimi súradnicami x, y, z a môže mať priradené ďalšie hodnoty opisujúce jeho vlastnosti [8].

1.6.2 Fast Voxel Traversal Algorithm

Fast Voxel Traversal Algorithm je technika používaná na efektívne prechádzanie voxelového priestoru. Tento algoritmus je v princípe podobný DDA (*Digital Differential Analyzer*) algoritmu, ktorý sa používa na rasterizáciu úsečiek v 2D, ale je rozšírený do 3D priestoru. Oba algoritmy využívajú prírastkové výpočty na určenie, ktorý voxel alebo pixel je potrebné spracovať ďalej. Vďaka tomuto prístupu je možné rýchlo a efektívne prechádzať voxelovým priestorom a určiť, ktoré voxely ležia na dráhe lúča. Použitím tejto techniky sa znižuje počet potrebných operácií pri prechode z jedného voxelu do druhého, čo zvyšuje rýchlosť a efektivitu renderovania objemových dát [3].

1.6.3 Raycasting

Raycasting je technika vykresľovania, ktorá umožňuje priame premietanie trojrozmerných dát na dvojrozmerné zobrazenie. Pri tomto procese sa z pozície pozorovateľa vysielajú lúče smerom k zobrazovacej ploche, kde každý lúč zodpovedá jednému pixelu na konečnom obraze. Lúče sa pohybujú po malých krokoch a postupne získavajú dáta a vlastnosti, ktoré spracúvajú [17]. Po prejdení určitej vzdialenosti sú výsledné informácie reprezentované ako pixely, čím vzniká realistický obraz 3D priestoru.

1.6.4 Ray-Box Intersection

Ray-box intersection je metóda, pomocou ktorej určujeme, či sa lúč pretína s daným objektom v priestore, týmto objektom myslíme ohraničený objem typu AABB. AABB je jednoduchý geometrický útvar v tvare kvádra, ktorého strany sú rovnobežné s osami súradnicového systému. Táto metóda počíta priesečníky lúča so všetkými šiestimi rovinami, ktoré tvoria steny ohraničujúceho objemu [18]. Je nevyhnutným prvkom pri použití metód ako je *raycasting* alebo *raytracing*, kde je výpočtová zložitosť veľká a efektívne detekcie priesečníkov sú kľúčové na určovanie kolízií.

1.7 Zobrazovací kanál

Zobrazovací kanál, po anglicky známy ako *rendering pipeline*, je základný proces v počítačovej grafike, ktorý zabezpečuje transformáciu 3D scén na 2D obrazy. Tento proces zahŕňa niekoľko etáp, ktoré postupne spracúvajú geometrické dáta, materiálové vlastnosti a ďalšie informácie. Konkrétne etapy a ich poradie sa môžu líšiť v závislosti od použitého softvéru a zvoleného spôsobu zobrazovania.

1.7.1 OpenGL

OpenGL je štandardizované rozhranie API (*Application Programming Interface*) pre 2D a 3D grafiku, ktoré umožňuje vývojárom na rôznych platformách vytvárať graficky náročné aplikácie. Využíva sa v mnohých odvetviach, ako sú vývoj hier, virtuálna realita, CAD systémy a vizualizácia vedeckých dát.

Hlavnou výhodou je schopnosť priamej spolupráce s grafickými kartami a využívanie ich akceleračných funkcií, čo umožňuje dosiahnuť vyšší výkon [9]. OpenGL zobrazovací kanál a jeho jednotlivé etapy možno vidieť na obrázku 1.4.

Shader programy

V OpenGL sú shader programy neoddeliteľnou súčasťou zobrazovacieho procesu. Sú to programy napísané v jazyku GLSL (*OpenGL Shading Language*), ktoré umožňujú priamu manipuláciu s grafikou na hardvérovej úrovni [9]. Existujú rôzne typy shader programov, pričom každý je zameraný na špecifickú fázu vykresľovacieho procesu:

- **Vertex shader** sa zaoberá spracovaním jednotlivých vrcholov a vykonáva transformácie vrcholov do priestoru po projekcii, môže byť použitý aj pre iné operácie s vrcholmi, ktoré pokračujú do ďalších etáp.

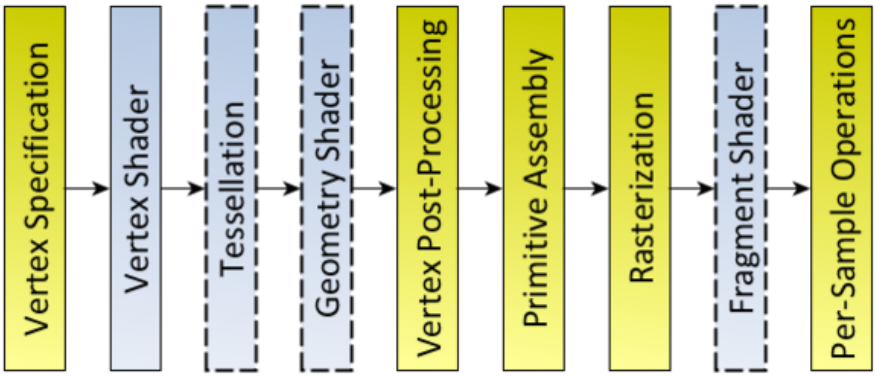
- **Tessellation shader** umožňuje vytvárať vyšší stupeň detailov na geometrických tvaroch bez potreby zvyšovania počtu vrcholov v pôvodnom modeli, a to pomocou rozdeľovania vrcholových dát na menšie primitíva.

- **Geometry shader** umožňuje spracovanie celej primitívy, ako sú trojuholníky alebo čiary, môže zmeniť počet a tvar primitív alebo generovať novú geometriu.

- **Fragment shader** je kľúčový pri vizualizácii finálneho obrazu, pretože na pixelovej úrovni manipuluje s jednotlivými fragmentami a ich vlastnosťami, ako sú farba a hĺbka. Každý fragment je spracovaný osobitne a nezávisle od ostatných, čo umožňuje vysokú úroveň paralelizácie.

- **Compute shader** je špeciálny typ shader programu v OpenGL, ktorý sa používa výhradne na výpočty nezávislé od tradičného renderovania grafiky. Tento shader program je ideálnym nástrojom pre komplexné vedecké výpočty, vďaka jeho efektívnemu spracovaniu dát a rýchlym výpočtom.

Uniformné premenné sú dôležitým prvkom shader programov, ktoré umožňujú prenos konštantných hodnôt z CPU do GPU. Tieto premenné sú definované v shader kóde a ich hodnoty zostávajú nemenné počas vykonávania shader programu pre všetky



Obr. 1.4: OpenGL zobrazovací kanál [10].

spracovávané vrcholy alebo fragmenty. Uniformné premenné sa často používajú na prenos maticových transformácií, svetelných parametrov alebo iných globálnych stavových informácií, ktoré sú potrebné na vykreslenie scény.

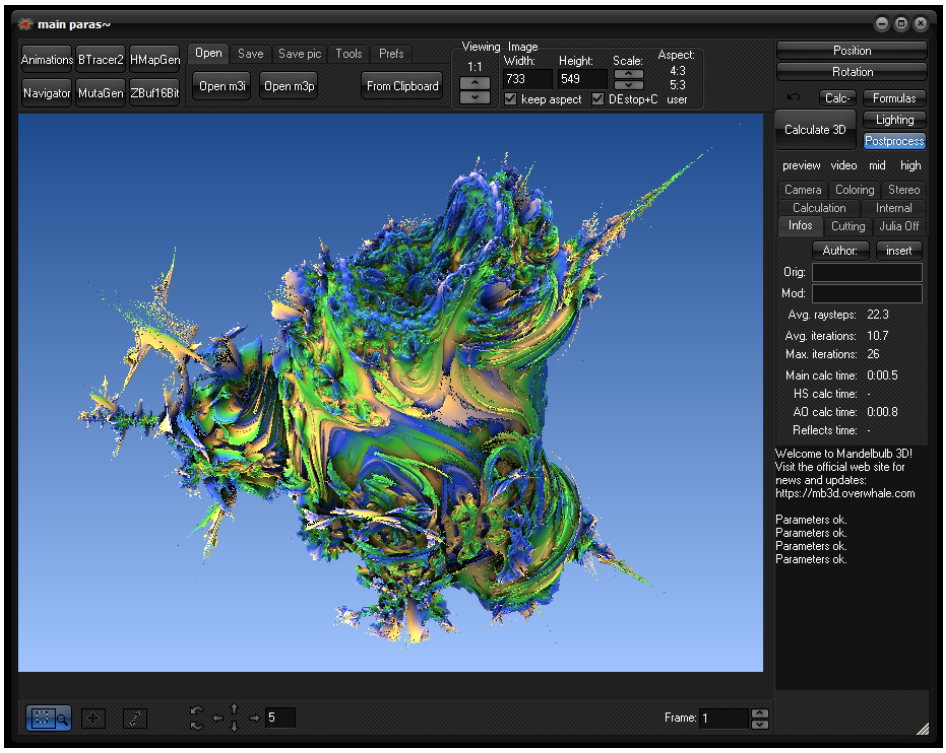
Proces presunu uniformných premenných z CPU do GPU zahŕňa niekoľko krokov, ako je získavanie lokácií uniformných premenných a nastavenie premenných. Tento proces umožňuje efektívne riadenie zobrazovacieho kanála a dynamické aktualizovanie parametrov renderovania bez potreby prerušovania práce GPU [9].

1.7.2 Vulkan

Vulkan je nízkoúrovňové API pre grafiku, ktoré poskytuje väčšiu kontrolu nad hardvérovými zdrojmi v porovnaní s OpenGL. Táto kontrola umožňuje vývojárom optimalizovať výkon a efektívnejšie spravovať pamäť, čo je kritické pre vysoko náročné grafické aplikácie. Vulkan podporuje širokú škálu platforiem, ako sú Windows, Linux a Android, čo z neho robí ideálne riešenie pre multiplatformový vývoj [11].

1.7.3 DirectX

DirectX je súbor API od Microsoftu, ktorý sa využíva na vývoj multimediálnych aplikácií, najmä hier, na platformách Windows. Obsahuje rôzne komponenty, z ktorých najznámejším je Direct3D pre 3D grafiku. Direct3D je špecificky navrhnutý pre optimalizáciu na systémoch Windows a poskytuje vývojárom kontrolu nad hardvérovými zdrojmi. Táto nízkoúrovňová kontrola umožňuje zlepšenie výkonu, ale zároveň zvyšuje jeho náročnosť [13].



Obr. 1.5: Aplikácia Mandelbulb3D.

1.8 Existujúce riešenia

Vizualizácia fraktálov spája matematiku s umeleckým vyjadrením a počítačovou grafikou. V oblasti 2D, 3D a 4D fraktálov existujú rôzne systémy, ktoré vizualizujú fraktály rôznymi technikami a optimalizáciami vzhľadom na komplexnosť výpočtov. Tieto systémy prezentujú estetiku fraktálov a zároveň poskytujú nástroje pre vedecké analýzy, ktoré umožňujú objavovanie nových vzorov a štruktúr.

1.8.1 Mandelbulb3D

Jeden z najpopúlarnejších systémov je Mandelbulb3D, umožňuje používateľom prieskum a renderovanie fraktálnych štruktúr v trojrozmernom priestore. Na vykresľovanie používa techniku *raymarching*, ktorá vysiela lúče po ktorých sa postupne pohybuje odhadom vzdialenosti od fraktálu [4]. Zároveň ponúka pokročilé možnosti osvetlenia a textúrovania, ktoré pridávajú realistickú hĺbku a detail. Ukážku tejto aplikácie môžeme vidieť na obrázku 1.5.

1.8.2 Mandelbulber2

Tento systém poskytuje známe variácie fraktálov, ako napríklad Mandelbox, Bulbbox a Juliabulb. Umožňuje prehliadanie 3D fraktálov v reálnom čase a podobne ako Mandelbulb3D využíva techniku *raymarching* [12].

Kapitola 2

Návrh práce

V tejto kapitole sa podrobne zaoberáme návrhom, výberom technológií, ktoré sú dostupné a vhodné pre náš systém, architektúrou aplikácie na vizualizáciu viacrozmerných fraktálov, konkrétne fraktálov definovaných nad kvaternionmi v štvorrozmernom priestore. Hlavným cieľom je poskytnúť detailný prehľad jednotlivých častí systému a ich procesov pre správne fungovanie aplikácie. Táto kapitola slúži nielen ako implementačný základ, ale aj ako základ pre budúci vývoj a inšpirácia pre ďalších vývojárov zaoberajúcich sa podobnými systémami.

2.1 Výber technológie

Na grafickú vizualizáciu viacrozmerných fraktálov potrebujeme technológiu, ktorá dokáže využívať GPU a umožní ich zobrazenie v reálnom čase. Pri výbere je kritické, aby sme mali určitú kontrolu nad hardvérom a mohli experimentovať s rôznymi zobrazovacími technikami.

Existujú systémy ako Unity alebo Unreal Engine 5, ktoré poskytujú množstvo nástrojov, ale často obmedzujú flexibilitu vývoja. Medzi technológie vyhovujúce našim požiadavkám patria OpenGL, Vulkan a Direct3D.

Po dôkladnom zvážení sme sa rozhodli vyvinúť našu aplikáciu s použitím OpenGL kvôli výbornej dokumentácii a širokej podpore na rôznych platformách. Z dostupných verzií OpenGL sme si vybrali najnovšiu, verziu 4.6, ktorá nám umožňuje používať compute shader. Použitie compute shader programu je kľúčovým bodom pre efektívne spracovanie fraktálov a zohráva podstatnú rolu v našej aplikácii.

Pri zvažovaní technológie Vulkan sme sa kvôli jeho vyššej vstupnej náročnosti a zložitosti rozhodli uprednostniť jednoduchší a flexibilnejší prístup OpenGL. DirectX je tiež výborným nástrojom, ale neponúka multiplatformový vývoj, nakoľko je kompatibilný len s platformami Windows.

2.1.1 Knižnice

OpenGL často vyžaduje pridanie dodatočných knižníc pre efektívnejší vývoj a správu základných operácií. Na poskytnutie dobrého základu a zjednodušenie týchto úloh sme sa rozhodli použiť nasledujúce knižnice:

- **GLAD** (*OpenGL Loading Library*) je nástroj, ktorý poskytuje dynamický mechanizmus na generovanie kódu potrebného na prístup k funkcionalitám OpenGL, keďže OpenGL je platformovo rozmanité API, ktoré samo osebe nespravuje načítanie svojich funkcií z hardvéru [9]. Tento prístup nám umožňuje implementovať funkcie OpenGL bez nutnosti zaoberania sa kompatibilitou na rôznych operačných systémoch a grafických kartách.

- **GLFW** poskytuje jednoduché rozhranie pre vytváranie okien, získavanie vstupov z klávesnice, myši, alebo iných vstupných zariadení a správu kontextov OpenGL [1].

- **Dear ImGui** je knižnica na tvorbu grafických používateľských rozhraní, ktorá nám umožňuje integrovať interaktívne a dynamické používateľské rozhranie do našej aplikácie. Táto knižnica je často využívaná v hernom vývoji, pretože poskytuje množstvo preddefinovaných komponentov.

2.2 Architektúra aplikácie

2.2.1 Vytvorenie a konfigurácia projektu

Pri návrhu a vývoji našej aplikácie sme sa rozhodli použiť Visual Studio, pretože toto integrované vývojové prostredie (IDE) poskytuje veľkú podporu pre programovanie v jazyku C++. Hlavným cieľom pri konfigurácii nášho projektu bolo zabezpečiť vysokú prenositeľnosť a minimalizovať potrebu externých inštalácií. Aby sme dosiahli tieto požiadavky, všetky potrebné knižnice sú priamo integrované do projektu, čo umožňuje jeho ľahký prenos a spustenie na inom systéme, ktorý obsahuje Visual Studio.

2.2.2 Triedny diagram

Na obrázkoch 2.1 a 2.2 je zobrazený triedny diagram, ktorý poskytuje vizuálny prehľad o štruktúre aplikácie a slúži na lepšie pochopenie vzťahov a funkcií jednotlivých častí, ktoré systém obsahuje.

Popis jednotlivých tried a ich funkcií:

- **Application** je jadro aplikácie, ktoré inicializuje OpenGL a spravuje hlavný cyklus vykonávania programu. Tento cyklus zabezpečuje neustále bežanie programu a koordinuje všetky hlavné procesy.

- **Camera** je zodpovedná za navigáciu a orientáciu v 3D priestore. Vektory, ktoré používa na určenie pozície a uhly na rotáciu dynamicky reagujú na používateľské vstupy.

- **Window** zabezpečuje inicializáciu a správu hlavného okna aplikácie, do ktorého sa vykresľuje grafický obsah. Spravuje vlastnosti okna ako sú rozmery a zmeny rozmerov používateľom.

- **Uniforms** je trieda obaľujúca všetky premenné a dáta, s ktorými aplikácia počas svojho behu pracuje. Umožňuje ľahkú prácu s premennými a ich manipuláciu. Zároveň slúži na prepojenie a aktualizáciu uniformných premenných používaných v shader programoch a pracuje s úložiskom potrebným pre výsledky vypočítané cez compute shader.

- **GUI** implementuje funkcionality knižnice Dear ImGui pre vytvorenie interaktívneho používateľského rozhrania. Táto trieda umožňuje používateľom interaktívnu manipuláciu s grafickou vizualizáciou.

- **VFShader** je trieda zodpovedná za správu vertex a fragment shader programu. Zabezpečuje kompiláciu a vytvorenie shader programu z načítaných súborov.

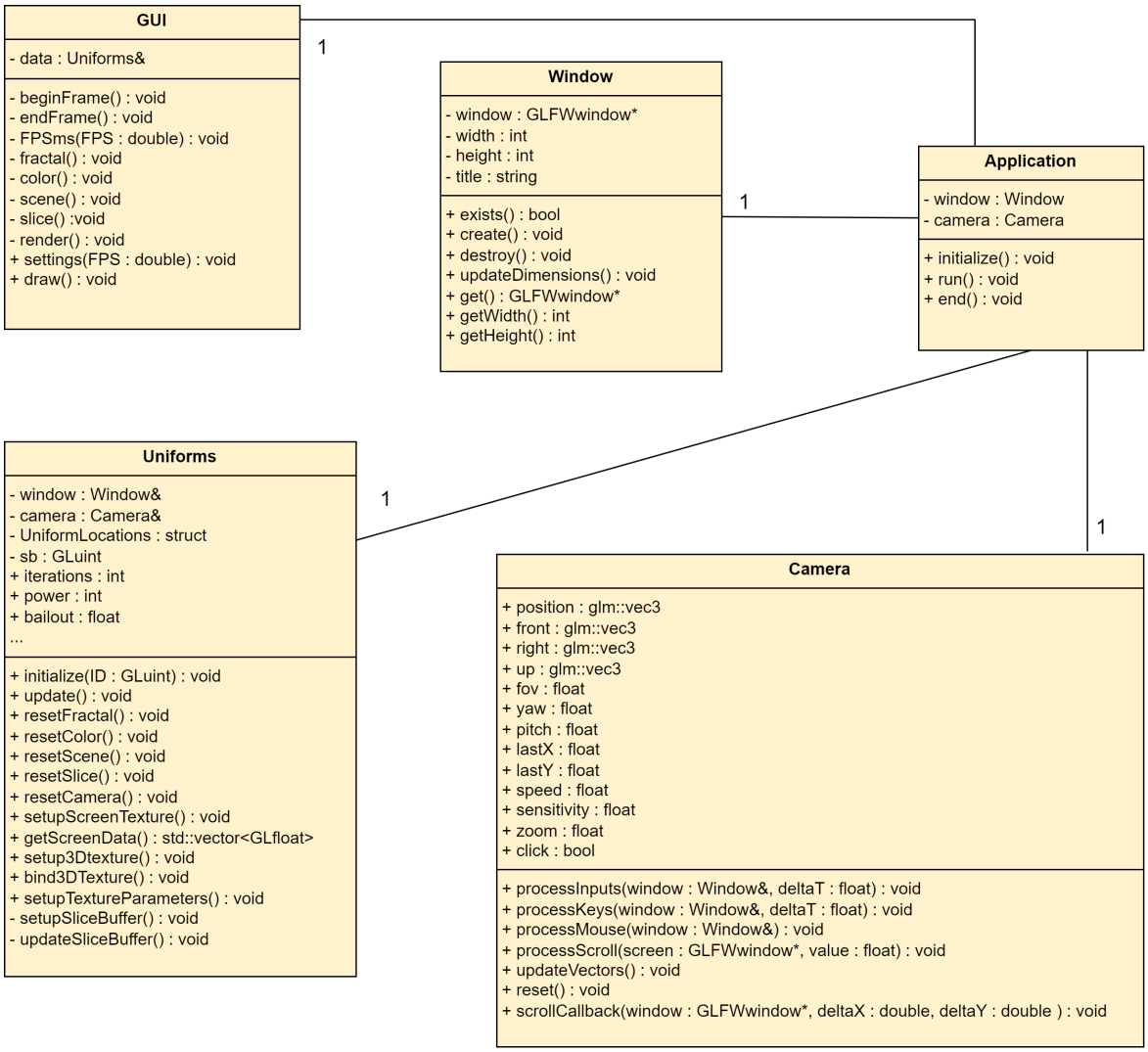
- **Compute** je trieda, ktorá spravuje compute shader program podobne ako trieda VFShader. Zabezpečuje kompiláciu a zodpovedá za jeho spustenie na GPU.

- **VBO** (*Vertex Buffer Object*) je zodpovedná za uchovávanie vrcholových dát v grafickej pamäti a umožňuje rýchlejšie vykresľovanie [9].

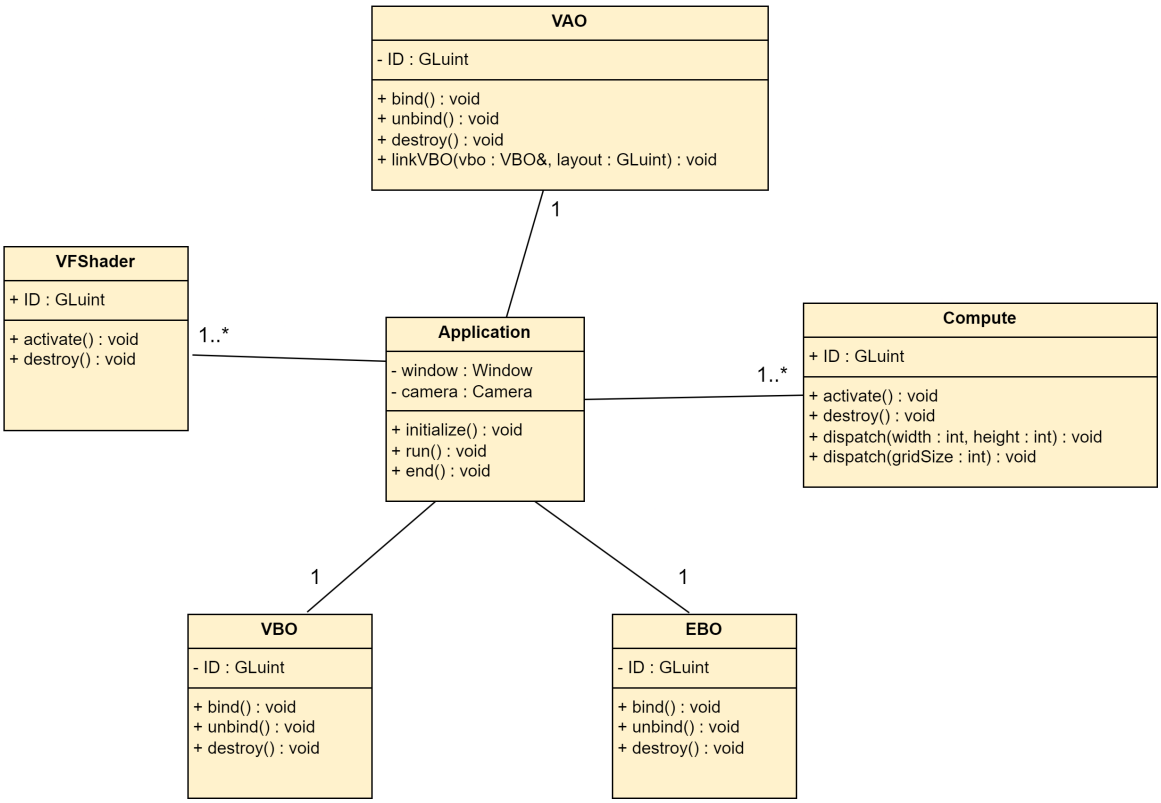
- **VAO** (*Vertex Array Object*) slúži ako kontajner pre mnoho VBO a definuje formát vrcholových dát uložených vo VBO [9].

- **EBO** (*Element Buffer Object*) sa používa spolu s VBO na efektívne ukladanie indexov, ktoré definujú, ako sú vrcholy spojené do primitív (trojuholníky alebo čiary) [9].

Časti VBO, VAO a EBO sú nevyhnutné aj v prípadoch, kedy aplikácia nepoužíva explicitné vrcholy alebo komplexné geometrie ako je to v našej aplikácií. OpenGL vyžaduje definované vrcholy a indexy pre renderovanie obrazu, je dôležité mať tieto základné komponenty pre správny výstup.



Obr. 2.1: Triedny diagram časť 1.



Obr. 2.2: Triedny diagram časť 2.

2.2.3 Používateľské rozhranie

Jedným z cieľov našej aplikácie je poskytnúť používateľom interaktívne používateľské rozhranie, ktoré im umožní pohybovať sa v priestore a prehliadať zobrazovanú štruktúru.

Návrh používateľského rozhrania

Po otvorení aplikácie sa používateľovi zobrazí interaktívne okno. Toto okno obsahuje jednotlivé sekcie, ktoré spravujú nastavenia fraktálu, farby, rezov štvrtej dimenzie, scény a renderovania obrázkov.

Popis jednotlivých sekcií:

- **Nastavenia fraktálu**: Táto sekcia umožňuje používateľovi vybrať z preddefinovaných fraktálov, ako sú Mandelbrotova a Juliina množina. Obsahuje nastavenia, ako sú počet iterácií, mocnina fraktálu a ďalšie možné nastavenia, ktoré sú špecifické pre vybraný fraktál.

- **Nastavenia farby**: Umožňuje zobraziť farbu štvrtej dimenzie a nastaviť osvetlenie fraktálov. Osvetlenie je dostupné len pri povrchovom zobrazení fraktálu.

- **Nastavenia rezov**: Umožňuje nastaviť rezy štvrtej dimenzie. Používatelia si môžu zvoliť interval rezov alebo nastaviť jednotlivé rezy manuálne, pričom je možné rezy pridávať alebo odoberať.

- **Nastavenia scény**: Poskytuje výber renderovacej techniky a s ňou spojené špecifické nastavania. Ďalej ponúka výber medzi povrchovým alebo volumetrickým zobrazovaním.

- **Nastavenia renderovania**: Umožňuje renderovať obrázky vo zvolenom rozlíšení a poskytuje možnosti na úpravu detailov, ktoré sa aplikujú len na výsledný obrázok.

Všetky nastavenia budú využívať komponenty, ktoré obsahuje knižnica Dear ImGui, a každá sekcia bude obsahovať tlačidlá pre resetovanie nastavení, čo umožňuje používateľom rýchlo vrátiť konfiguráciu do východiskového stavu.

2.2.4 Dátový tok

Pre správne fungovanie aplikácie je nevyhnutné zabezpečiť plynulý tok dát medzi rôznymi komponentami systému. Správna manipulácia a koordinácia dát sú kľúčové pre predchádzanie potenciálnym problémom.

Tok dát v našej aplikácii prechádza nasledujúcimi krokmi:

1. **Inicializácia dát:** Pri spustení aplikácie sa pred hlavným cyklom programu inicializujú všetky premenné konštatami.

2. **Vstup od používateľa:** Používateľ má k dispozícii grafické rozhranie na prácu s aplikáciou. Vstupy sú prijímané cez toto rozhranie a pomocou klávesnice a myši.

3. **Aktualizácia dát:** Pred každým príkazom na vykreslenie sa aktualizujú všetky dáta bez ohľadu na ich zmenu, tieto dáta sa potom posielajú do aktivovaného shader programu cez uniformné premenné.

4. **Spracovanie shader programom:** Po poslaní uniformných premenných môže shader program pracovať s aktualizovanými dátami. Shader program spracúvava tieto dáta a vytvára grafický výstup.

5. **Výstup:** Výsledné grafické dáta sú zobrazené na obrazovke.

2.2.5 Shader programy a ich riadenie

Aplikácia bude obsahovať tri hlavné shader programy, ktoré je v rámci architektúry našej aplikácie dôležité správne riadiť a koordinovať. Na začiatku aplikácie sa inicializujú

všetky tri shader programy. Tieto programy budú pripravené na spustenie v závislosti od ich potreby.

Jeden shader program je fragment shader, ktorý je aktívny väčšinu času počas behu aplikácie a stará sa o vizualizáciu scény v okne aplikácie. Ďalšie dva budú compute shader programy, jeden na vypočítavanie obrázkov a druhý na prepočítavanie voxelových dát. Compute shader programy sú spúšťané na požiadanie, keď je potrebné vykonať náročnejšie výpočty.

Prepínanie shader programov

OpenGL dovoľuje bežanie len jedného shader programu súčasne. Pri potrebe zmeny z fragment na compute shader systém najprv deaktivuje bežiaci fragment shader a spustí compute shader. Ideálnym spôsobom by bolo vykonávať čiastkové výpočty cez compute shader, pretože výpočet môže trvať dlhšiu dobu, a poskytovať tieto dáta fragment shader programu. Avšak tento prístup by mohol viesť k nekonzistencii dát v dôsledku častých zmien parametrov od používateľa. Riešením tohto problému je proces, ktorý umožňuje aplikácii zastaviť sa a počkať na dokončenie výpočtov compute shader programu predtým, ako aktivuje fragment shader program. Tento návrh zabezpečuje, že všetky spracované dáta sú konzistentné a kompletné. Každý shader program môže efektívne vykonávať svoju funkciu bez negatívneho ovplyvnenia a narušenia ostatných.

Kapitola 3

Implementácia

V tejto kapitole si podrobne rozoberieme implementačné detaily našej aplikácie a implementujeme rôzne techniky na vizualizáciu štvorrozmerných fraktálov. Popíšeme konkrétne kroky a techniky použité v procese vývoja, ktoré nás dovedú od začiatočných fáz až po finálnu verziu aplikácie. Okrem technických aspektov sa zameriame aj na optimalizácie, ktoré zvyšujú efektivitu a výkon našej aplikácie.

3.1 Využitie zobrazovacieho kanálu

Pri implementácii našej aplikácie sme sa zamerali na efektívne využitie zobrazovacieho kanálu. Zobrazovací kanál OpenGL ponúka viacero úrovní spracovania, ktorých úprava a implementácia nebola potrebná. Všetka logika pre zobrazenie je integrovaná v jedinom fragment shader programe, ktorý priamo manipuluje s obrazovými dátami na úrovni fragmentov.

3.2 Transformácia súradníc

Fragment shader dostáva na vstupe súradnice pixelov, ktoré sú definované rozmerom okna, v ktorom je aplikácia spustená. Tieto súradnice určujú, pre ktorý pixel sa vypočítavajú grafické vlastnosti.

Proces transformácie súradníc je kľúčovým krokom, ktorý zjednodušuje nasledujúce fázy zobrazovania a je štandardne implementovaný v mnohých grafických systémoch. Implementujeme konverziu súradníc z ich pôvodnej formy do normalizovanej formy. Pri tejto transformácii sa súradnice premapujú tak, aby pokrývali interval od -1 po 1 pre obe osi, x aj y. Aby sme predišli vizuálnej deformácii v dôsledku neštvorcových rozmerov okna, prispôsobíme os x pre zachovanie pomeru strán obrazovky. Po transformácii súradníc sa bod $[0, 0]$ nachádza presne v strede obrazovky.

3.3 Smer lúča

Pomocou normalizovaných súradníc pixelov a orientačných vektorov kamery vypočítavame smer lúča pre každý pixel. Tento proces nám umožňuje efektívne prechádzať scénu tretím rozmerom a vizualizovať priestor. Smer lúča začína na pozícii kamery a simuluje perspektívu ľudského oka, čo je kľúčové pre dosiahnutie realistického zobrazenia.

3.4 Výpočet fraktálu

Pre zobrazenie štvrorozmerného fraktálu do 3D priestoru používame výpočet fraktálu pomocou kvaternionových funkcií. Výpočet vykonávame rovnako ako v komplexnej rovine, používame iteratívnu funkciu, ktorú iterujeme zvoleným počtom iterácií.

Ukážka výpočtu je znázornená v algoritme 3.1, ktorý popisuje výpočet Mandelbrotovej množiny definovanej nad kvaternionmi. Algoritmus vracia príslušnosť vstupného kvaternionu c k Mandelbrotovej množine, kde prvé tri súradnice x, y, z predstavujú pozíciu v 3D priestore a štvrtá súradnica w predstavuje štvrtú dimenziu. Funkcia $qPower(q, n)$ umocňuje kvaternion q na mocninu n, $qNorm(q)$ vracia veľkosť kvaternionu q, *iterations* je počet iterácií a *bailout* je hodnota, ktorá určuje hranicu pre zastavenie iterácie, ak veľkosť kvaternionu presiahne túto hodnotu.

Algoritmus 3.1: Algoritmus na výpočet príslušnosti bodu k fraktálu.

```
bool fractal(vec4 c) {
    q = vec4(0.0);
    for (int i = 0; i < iterations; i++) {
        q = qPower(q, n) + c;
        if (qNorm(q) > bailout) {
            return false;
        }
    }
    return true;
}
```

3.5 Prechádzanie scény

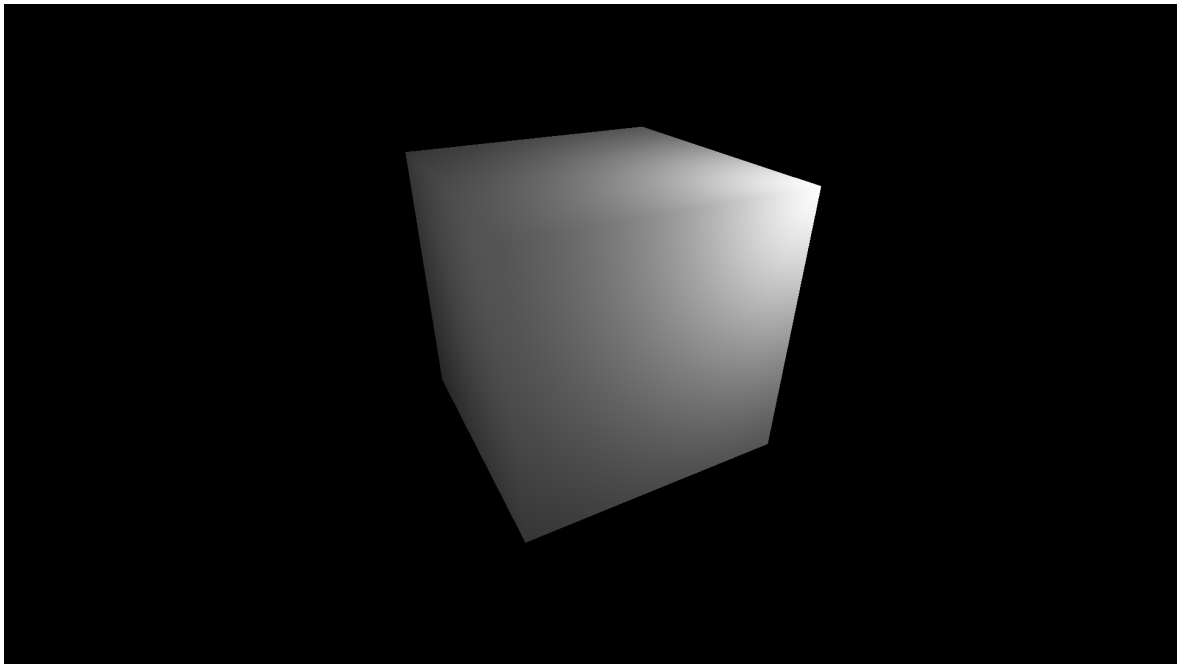
Pre každý pixel na obrazovke sme si pripravili smer lúča a máme pozíciu kamery, vďaka týmto parametrom sa môžeme malými krokmi pohybovať po lúči a zbierať informácie z 3D priestoru, ktoré vykazuje daný fraktál.

Hoci je táto technika široko používaná pre jej jednoduchosť, má svoje nevýhody. Jeden z problémov ktorý nastáva, je že pozícia kamery môže byť vzdialená od zau-

jímavých častí fraktálu a počet krokov musí byť výrazne vyšší, čo spôsobuje vysokú výpočtovú náročnosť.

3.5.1 Optimalizácia

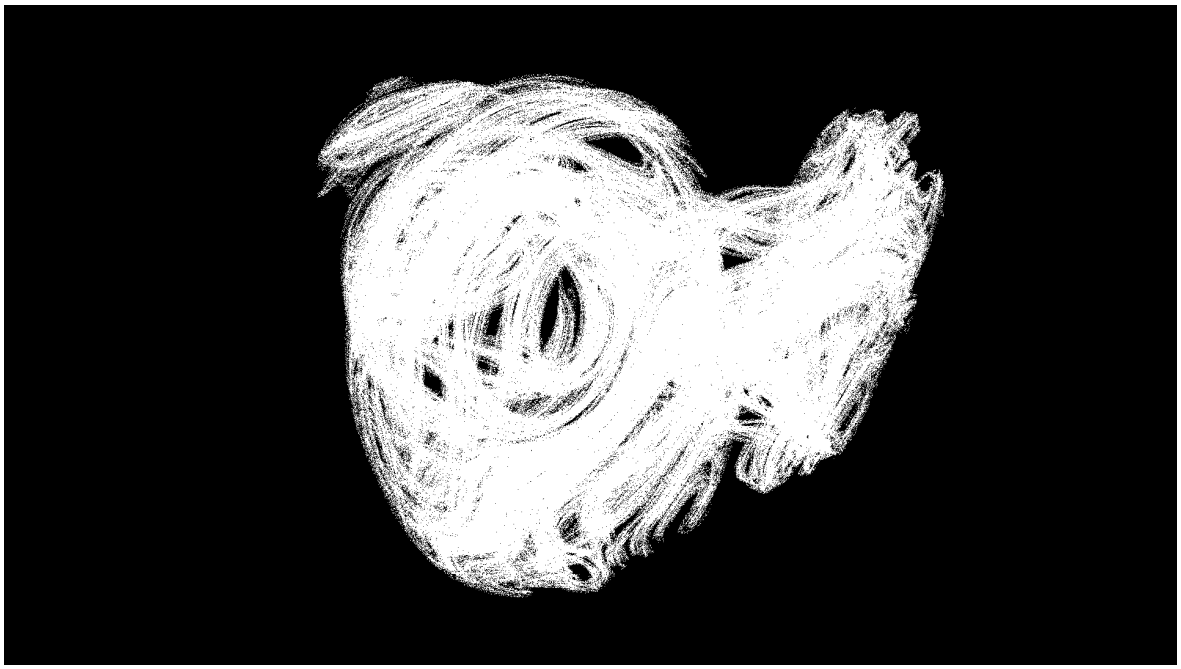
Aby sme minimalizovali túto náročnosť a zefektívnili vizualizáciu, definovali sme nemennú obálku priestoru $[-1, 1]^3$, v ktorej sa zobrazujú všetky výpočty. Všetko mimo tejto obálky ignorujeme. Vďaka použitiu algoritmu *Ray-Box Intersection* môžeme efektívne určiť, ktoré lúče prechádzajú obálkou a ktoré nie. Ak lúč neprechádza touto obálkou, pixelu priradíme farbu pozadia. Ak lúč prechádza obálkou, vstupujeme do obálky v mieste prvého priesečníka a pokračujeme v smere lúča až po jej koniec. Na obrázku 3.1 možno vidieť danú obálku, v ktorej zobrazujeme výpočty.



Obr. 3.1: Obálka v ktorej prebieha výpočet fraktálu.

Obálka je nemenná, táto vlastnosť zabezpečuje konzistentné zobrazenie fraktálu vždy na rovnakom mieste.

Obálka nedefinuje hodnoty fraktálu, ktoré počítame, pri nájdení priesečníku transformujeme body na požadovaný úsek ktorý chceme vypočítať. Tento postup umožňuje pozorovanie rôznych častí fraktálu na rôznych mierkach bez potreby prispôsobovať rozsah pozorovania alebo upravovať pohyb kamery.



Obr. 3.2: Fraktál bez farby a osvetlenia.

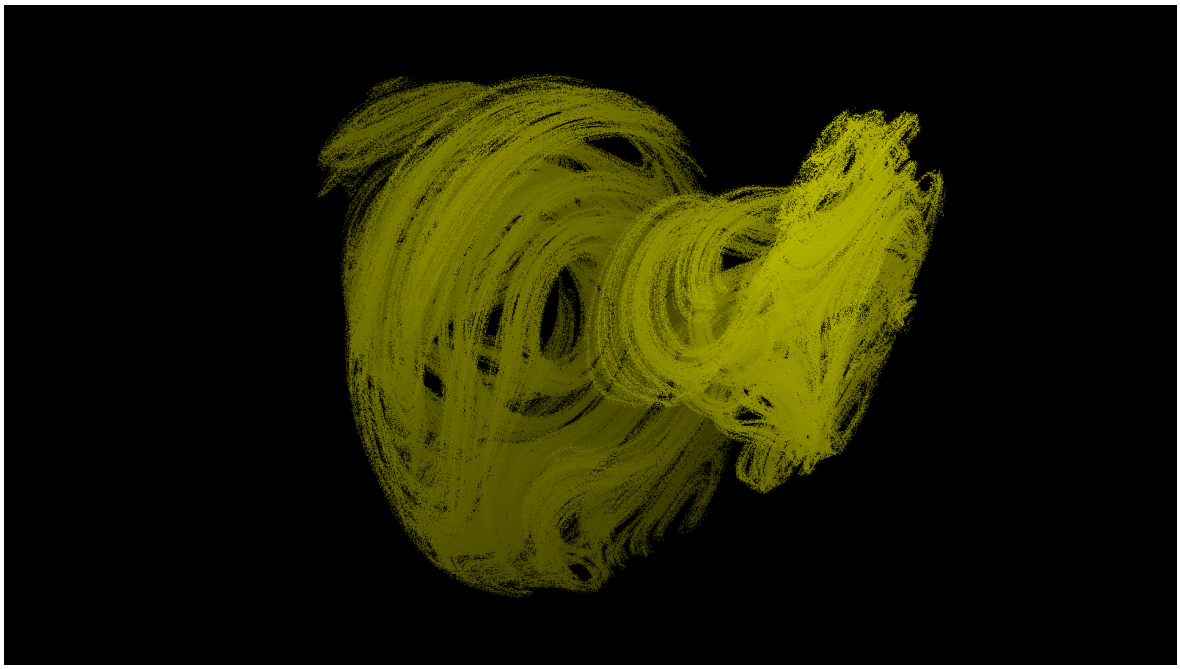
3.6 Farba a Osvetlenie Fraktálu

Farbu používame na reprezentáciu dodatočnej štvrtej dimenzie, ktorej detaily by boli inak nemožné rozlíšiť. Pri neaplikovaní farby nevieme o štvrtej dimenzii nič iba hodnotu pre ktorú sa vypočítava.

Pre transformáciu štvrtej dimenzie fraktálu na farbu, využívame metódu založenú na mapovaní intervalu $[-1, 1]$ na vlnové dĺžky v rozmedzí od 380 do 780 nanometrov, tento rozsah pokrýva celé viditeľné spektrum, od fialovej po červenú. Transformácia na konkrétne farby je realizovaná algoritmom od tvorcov Dragoş Mihai a Eugen Străjescu, ktorý je podrobne opísali v práci [14].

Použitie osvetlenia pridáva hĺbku do obrazu a zvyšuje realistickosť zobrazenia. Náš výpočet negeneruje geometriu povrchu, preto nie sú dostupné normálové vektory pre výpočet interakcie fraktálu a svetla. Osvetlenie, ktoré zlepšuje chápanie tretej dimenzie sme preto implementovali pomocou kvadratického modelu útlmu svetla. Tento model zohľadňuje vzdialenosť svetelného zdroja od bodov fraktálu, kde intenzita svetla klesá s rastúcou vzdialenosťou od zdroja.

Ukážku rozdielu medzi nepoužitím a použitím týchto techník na zlepšenie realistickosti je možné videiť na obrázkoch 3.2 a 3.3.



Obr. 3.3: Fraktál s farbou a osvetlením.

3.7 Povrchové a volumetrické zobrazenie

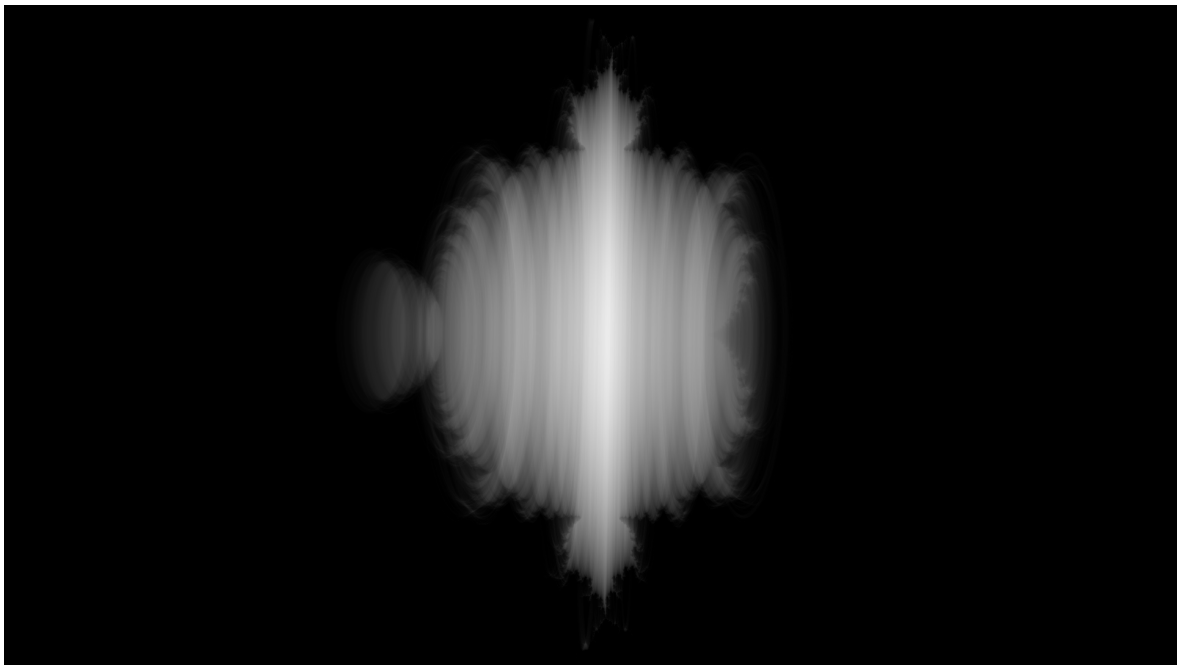
Fraktály môžeme zobraziť viacerými spôsobmi, pričom každá ponúka unikátne pohľady a informácie o štruktúre fraktálu.

Povrchové zobrazenie

Pri povrchovom zobrazení sa sústredíme na zobrazenie vonkajšej hranice fraktálu. Lúč vychádzajúci z kamery prechádza scénou a je postupne testovaný na prítomnosť fraktálu. Výpočet sa zastaví, keď lúč narazí na prvý bod, ktorý je súčasťou fraktálu. Tento prístup zjednodušuje výpočet tým, že sa zameriava len na najbližšie hraničné body fraktálu pozdĺž lúča. Každý pixel obrazovky teda reprezentuje najbližší povrchový bod fraktálu voči pozícii pozorovateľa. Ukážku povrchového zobrazenie môžeme vidieť na obrázku 3.3, na obrázku je znázornená aj farba štvrtej dimenzie, ktorá je fixná a zobrazuje jeden rez štvrtej dimenzie.

Volumetrické zobrazenie

Pri volumetrickom zobrazovaní prechádzame celým objemom, teda obálkou ktorú sme definovali pre zobrazenie výpočtov. Lúč vychádzajúci z kamery prechádza fraktálom a spočítava farby všetkých bodov, ktorými prejde. Spočítaná farba sa vydelí počtom krokov, ktorými prechádzame cez obálku, delením dosiahneme priemernú hodnotu farby . Ukážku volumetrického zobrazenia môžeme vidieť na obrázku 3.4.



Obr. 3.4: Volumetrické zobrazenie fraktálu.

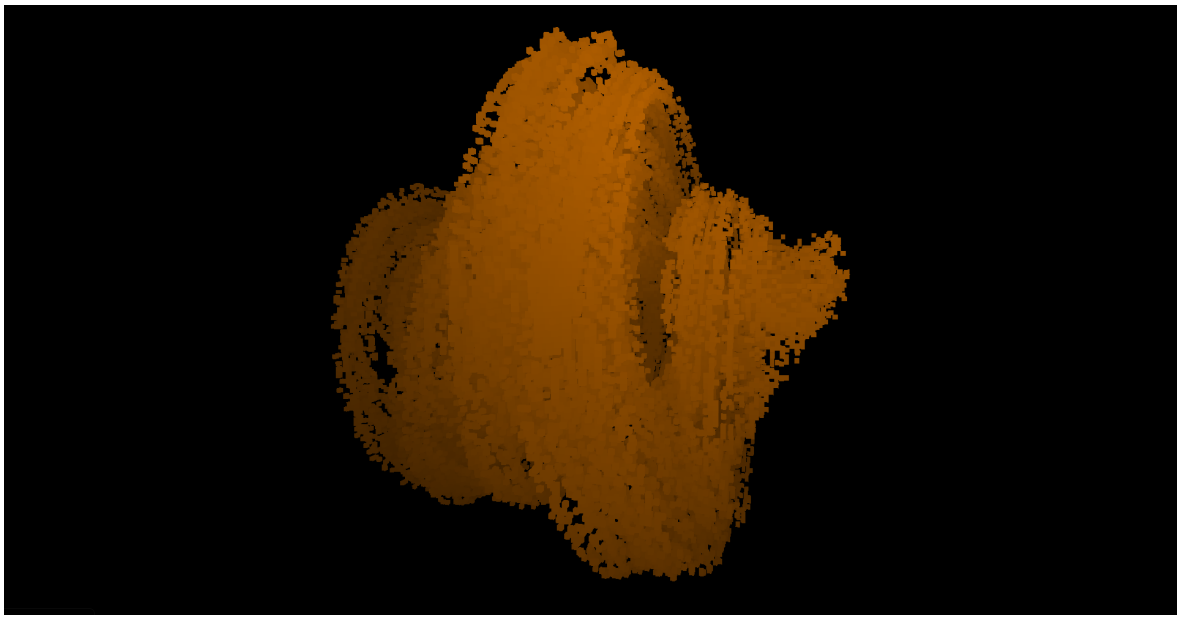
3.8 Implementácia voxelov

Doterajší postup zobrazovania fraktálov v našej aplikácií má významné nevýhody, ako je potreba opakovaného výpočtu fraktálu pre každú snímku. Tento proces je nepraktický a neefektívny, najmä keď technika postupného pohybu po lúči môže preskočiť oblasti, kde by sa mohol fraktál nachádzať. Tento prístup tiež vedie k nekonzistencii zobrazenia, keďže z rôznych uhlov môžeme zachytiť odlišné detaily.

Na riešenie týchto problémov sme zvolili predspracovanie dát fraktálu v compute shader programe. Tento program vypočíta fraktál pre každý voxel pre danú veľkosť voxelovej mriežky, na ukladanie dát používa *3D texture buffer*, ktoré po skončení výpočtov následne využije fragment shader pre vizualizáciu. Compute shader počíta prítomnosť fraktálu v danom bode, ak je fraktál prítomný, potom uložíme farbu štvrtej dimenzie, ktorá zaberá 24bitov pamäte pre každý voxel.

Na zefektívnenie procesu vizualizácie voxelových dát sme implementovali techniku, známu ako Fast Voxel Traversal Algorithm, ktorá sa neposúva po lúči ale po jednotlivých voxeloch vo voxelovom priestore. Ukážku reprezentácie fraktálu pomocou voxelov môžeme vidieť na obrázku 3.5.

Tento postup nám umožňuje vyhnúť sa opakovanému výpočtu a zároveň eliminuje nekonzistenciu v zobrazení fraktálu z rôznych perspektív.



Obr. 3.5: Zobrazenie fraktálu voxelmi.

3.9 Renderovanie obrázkov

Renderovanie obrázkov vo vysokom rozlíšení je považované za kľúčovú vlastnosť každého vizualizačného systému. Naša implementácia využíva compute shader, ktorý funguje podobne ako fragment shader. Hlavný rozdiel spočíva v tom, že compute shader vypočíta výstupné dáta len raz a uchováva ich v pamäti.

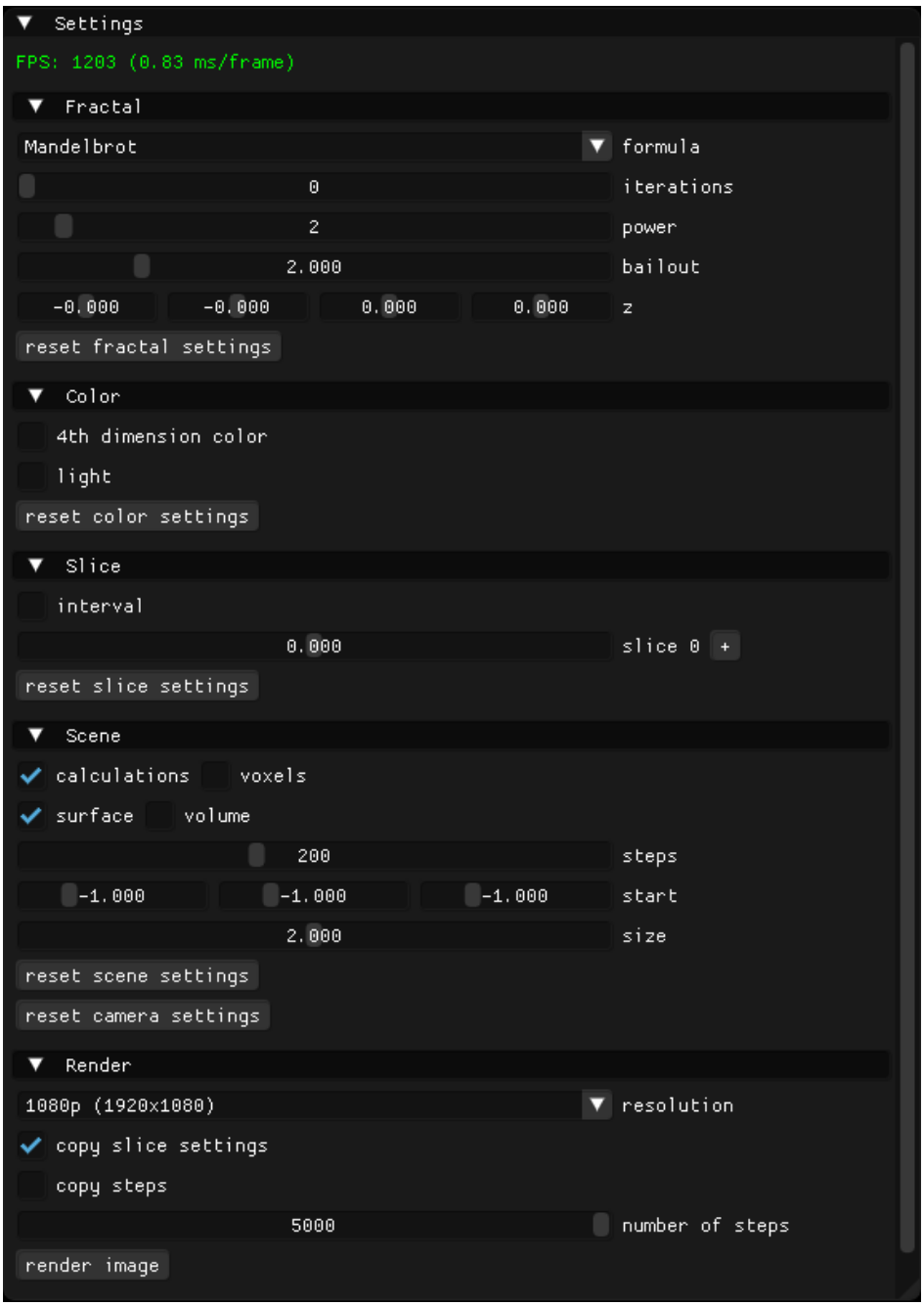
Na rozdiel od fragment shader programu, compute shader prijíma na vstupe súradnice pixelov, ktoré však korešpondujú s rozlíšením finálneho obrázku, nie okna aplikácie. Tento postup umožňuje výpočet obrázkov s použitím techniky postupného posúvania po lúči, kde sa obraz vykresľuje po malých krokoch. Ďalej sme implementovali ďalšie nastavenia, ako je určenie počtu krokov, čo umožňuje renderovanie obrázka s vysokým detailom.

Po dokončení všetkých výpočtov sa na základe vytvorí obrázok formátu .png. Ukladanie obrázkov prebieha automaticky a v hlavnom projekte sa vytvára priečinok *output*, z ktorého si môžu používatelia ľahko obrázky prezerať.

3.10 Používateľské rozhranie

Používateľské rozhranie bolo navrhnuté tak, aby umožnilo používateľom pohodlne prezeranie a experimentovanie s rôznymi typmi fraktálov. Hlavné rozhranie je rozdelené do viacerých sekcii, ktoré umožňujú detailné nakonfigurovanie fraktálu a využívanie rôznych zobrazovacích techník. Ukážku rozhrania môžeme vidieť na priloženom obrázku 3.6.

Implementácia tohto rozhrania bola vykonaná s využitím knižnice Dear ImGui,



Obr. 3.6: Používateľské rozhranie.

ktorej existujúce komponenty boli postačujúce na jeho vytvorenie. Rozhranie obsahuje intuitívne ovládacie prvky, ktoré sú zoskupené tematicky podľa ich funkcie a účelu.

Nastavenie fraktálu

Sekcia nastavenie fraktálu umožňuje používateľovi vybrať si z rôznych preddefinovaných fraktálov prostredníctvom rozbaľovacieho menu, implementovali sme nastavenia pre najznámešie množiny ako je Mandelbrotova množina a Juliina množina. Po výbere konkrétneho fraktálu sa zobrazia jeho špecifické nastavenia.

Medzi tieto nastavenia patrí počet iterácií, mocnina, ktorou je fraktál umocnený a bod, ktorý ovplyvňuje vizualizovaný fraktál. Užívateľ má možnosť upravovať tieto hodnoty pomocou posuvníkov. Posuvníky obsahujú vždy minimálnu a maximálnu hodnotu.

Nastavenie farby

Nastavenia farby umožňujú prispôsobiť vizuálnu reprezentáciu štvrtej dimenzie a zlepšiť vnímanie tretej dimenzie pomocou osvetlenia. Tieto voľby sa aktivujú pomocou zaškrtávacích políčok. Pri použití osvetlenia sa sprístupnia ďalšie nastavenia pre úpravu polohy svetelného zdroja v scéne.

Pri volumetrickom zobrazení sa sprístupní ďalšie nastavenie na úpravu intenzity farby každého zobrazovaného bodu. Táto funkcia bola pridaná z dôvodu nízkej viditeľnosti fraktálov, ktoré tvoria len malú časť zobrazovaného priestoru.

Nastavenia rezov

V nastaveniach rezov existujú dve možnosti. Prvou možnosťou, ktorá je predvolená pri spustení aplikácie, je pridávanie a odoberanie jednotlivých rezov pomocou tlačidiel $+$ a $-$. Jednotlivé rezy je možné odoberať a pridávať, nastavenie rezu je ovládané posuvníkom. Druhou možnosťou je výber intervalu, z ktorého sa rezy vyberú a počtu rezov, ktoré sa v tomto intervale zobrazia.

Nastavenia scény

Nastavenia scény ponúkajú najrozsiahlejšie možnosti. Používateľ si môže vybrať techniku zobrazenia fraktálu. Pri voľbe možnosti *calculations* sa fraktál zobrazuje postupným pohybom pozdĺž lúča, počet týchto krokov je možné upravovať pomocou posuvníka. Ďalšou technikou je zobrazenie fraktálu ako voxelov *voxels*, výber tejto techniky spúšťa funkciu, ktorá aktivuje compute shader na prepočítanie voxelov a nahradí možnosť výberu počtu krokov výberom veľkosti voxelovej mriežky v priestore, prostredníctvom rozbaľovacieho menu. V oboch technikách sú k dispozícii povrchové a

volumetrické zobrazenia.

Pri voľbe *voxels* sa zobrazí tlačidlo *recalculate*, pretože časté prepočítavanie fraktálu pri každej zmene nastavení je nepraktické. Po nakonfigurovaní požadovaného fraktálu je potrebné kliknúť na toto tlačidlo, aby sa voxely fraktálu znovu prepočítali.

Pomocou nastavení *start* a *size* je možné posúvať fraktálom v priestore a zobrazovať ľubovoľnú časť fraktálu, tieto možnosti sú implementované posuvníkmi.

Nastavenie renderovania

Nastavenia renderovania umožňujú používateľovi vytvoriť obrázok fraktálu s vysokým rozlíšením a detailom. K dispozícii je rozbaľovacie menu s preddefinovanými rozlíšeniami, z ktorých si používateľ môže vybrať.

Ďalej sú k dispozícii dve možnosti: kopírovanie nastavení rezov alebo kopírovanie počtu krokov. Tieto funkcie boli pridané z dôvodu vysokej náročnosti pri zobrazovaní veľkého počtu rezov a krokov v reálnom čase. Používateľ si tak môže nakonfigurovať vlastné nastavenia rezov a krokov, ktoré sa použijú len na vyrenderovaný obrázok.

3.10.1 Pohyb kamery

Pohyb kamery v priestore reaguje na vstupy klávesnice a myši. Pohyby hore, doľava, dole, doprava ovládajú klávesy *W*, *A*, *S*, *D*, pomocou kolieska myši sa posúvame dopredu alebo dozadu. Priblíženie sa ovláda stlačením klávesy *CTRL* a kolieskom myši, tento vstup aktualizuje zorné pole kamery a po pustení klávesy *CTRL* sa vrátime na pôvodné nastavenie zorného poľa. Stlačenie a držanie ľavého tlačidla myši, spolu s pohybom myši, umožňuje rotáciu kamery okolo aktuálnej pozície.

Kapitola 4

Výsledky

Táto kapitola je venovaná prezentácii a diskusii o výsledkoch dosiahnutých počas vývoja a implementácie nášho vizualizačného softvéru. Zvláštny dôraz kladieme na hodnotenie výpočtovej a pamäťovej zložitosti rôznych zobrazovacích techník, ktoré sme implementovali. Analyzujeme, ako jednotlivé prístupy ovplyvňujú výkon aplikácie a kvalitu zobrazenia fraktálov. Výsledky sú podložené kvantitatívnymi údajmi a vizuálnymi príkladmi.

4.1 Porovnanie zobrazovacích techník

Pre lepšie porozumenie údajov označíme metódu výpočtov pozdĺž lúča ako VPL a metódu zobrazovania voxelov ako MZV. Tieto dve metódy sa zásadne líšia vo svojich prístupoch a v detaile, ktorý poskytujú. Pre ich efektívne porovnanie sme navrhli testovacie prostredie, ktoré priblíži ich zložitosť a schopnosť zobrazovať detaily.

4.1.1 Testovacie prostredie

Využívame prostredie s konštantnými rozmermi zobrazovacieho okna 512x512 pixelov. Kamera je umiestnená tak, aby fraktálna štruktúra bola zobrazená cez celú obrazovku. Pre VPL stanovíme konštantný počet krokov na 512, zatiaľ čo pre MZV nastavíme voxelovú mriežku o rozmeroch 512x512x512. Týmto zaistíme, že každý pixel obrazovky bude reprezentovaný jedným voxelom, čo umožní rovnaké podmienky pre obidve metódy. Tento prístup zaručuje, že výsledky metód sú porovnateľné. Tabuľka 4.1 zobrazuje výsledky testovania, ktoré bolo vykonané na grafickej karte RTX 3060, pri použití volumetrického zobrazenia kvaternionovej formy Mandelbrotovej množiny s 32 iteráciami a umocnením na druhú.

Tabuľka 4.1: Doba a pamäť potrebná pre výpočet jedného snímku s daným počtom rezov štvrtej dimenzie.

Metóda	Počet rezov	Čas (ms)	Pamäť (MB)
VPL	1	17	0
	3	55	0
	10	180	0
MZV	1	7	384
	3	7	384
	10	7	384

4.1.2 Efektívnosť a pamäťová zložitosť

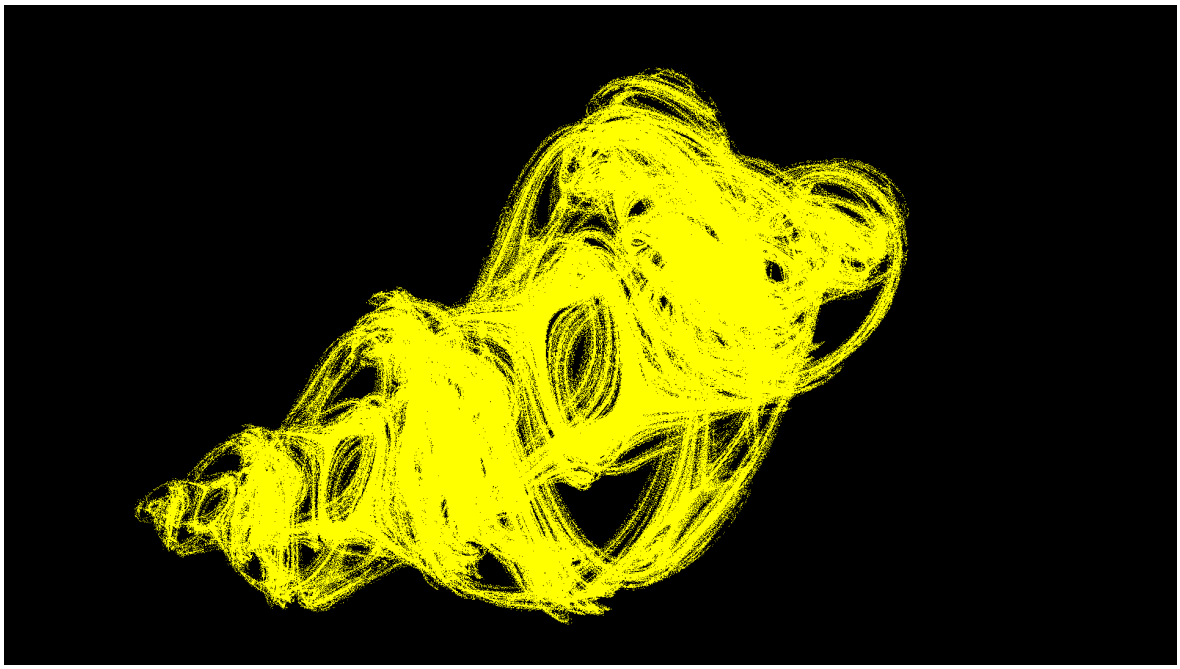
Pri porovnávaní Metódy VPL a metódy MZV je dôležité poznamenať, že každá z nich má svoje špecifické výhody a nevýhody. Ako je zrejmé z údajov v tabuľke 4.1, Metóda VPL je časovo náročnejšia, pričom doba potrebná na vykreslenie jedného snímku sa výrazne zvyšuje s počtom rezov štvrtej dimenzie. Napriek tomu, táto metóda nevyžaduje uchovávanie dodatočných dát v pamäti.

Metóda MZV si vyžaduje veľký objem pamäte na ukladanie predpočítaných voxelových dát. Pamäťové požiadavky sú konštantné bez ohľadu na počet rezov, keďže pamäť obsahuje len finálnu farbu voxelov. Táto metóda poskytuje výhodu v rýchlosti zobrazenia, pretože počas samotného renderovania nie sú potrebné ďalšie výpočty, čo značne skracuje čas potrebný na vykreslenie obrázka.

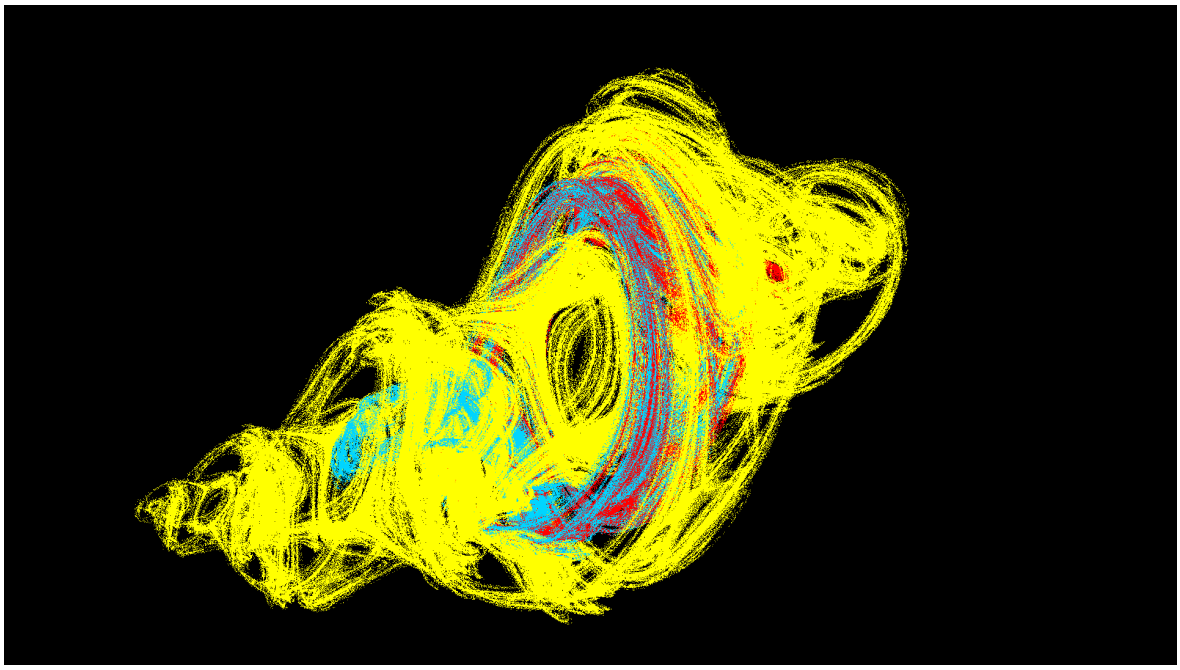
Táto rozdielnosť ukazuje, že metóda MZV je vhodnejšia pre aplikácie, kde je prioritou rýchlosť zobrazenia a je k dispozícii dostatočné množstvo pamäte. Metóda VPL by mohla byť preferovaná v prostrediach, kde nie je možné uchovávať veľké množstvá dát v pamäti.

4.2 Vizuálne ukážky

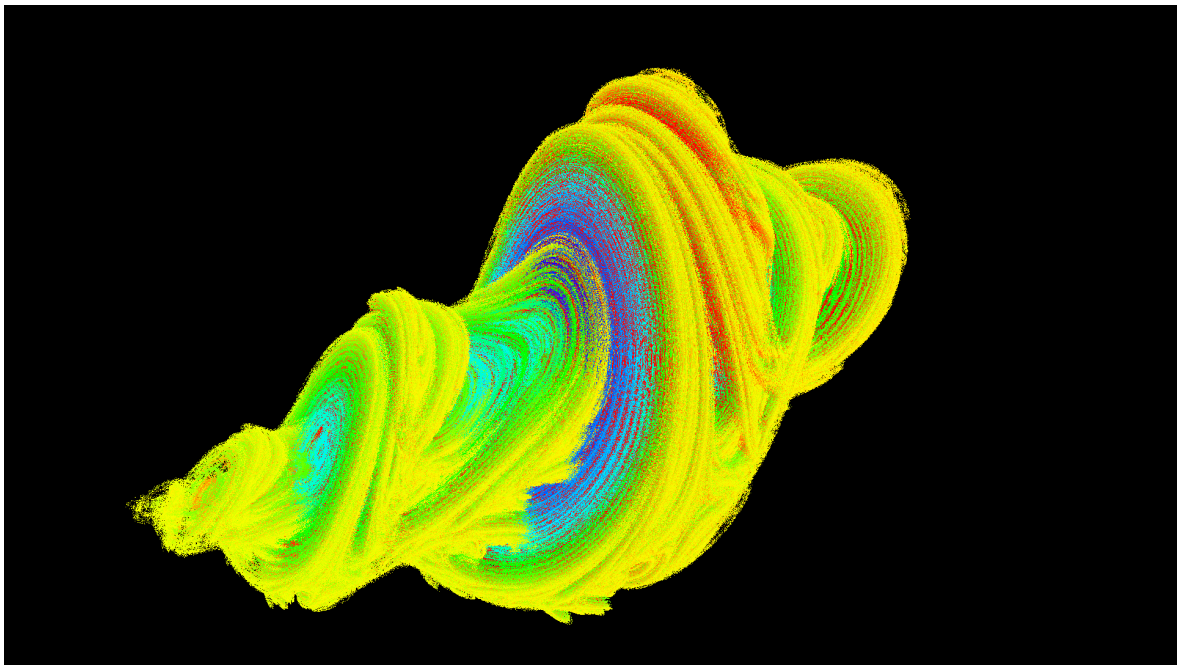
Pre vizualizáciu celého štvorrozmerného fraktálu nestačí zobraziť len jeho jednotlivé rezy, ale je potrebné ich spojiť. Ako príklad si vezmime Juliinu množinu a postupne navyšujme počet rezov vo štvrtej dimenzii (obrázky 4.1, 4.2, 4.3). Na obrázku 4.4 je použitý volumetrické zobrazenie, ktoré na rozdiel od povrchového zobrazenia odhaľuje aj vnútornú štruktúru fraktálu.



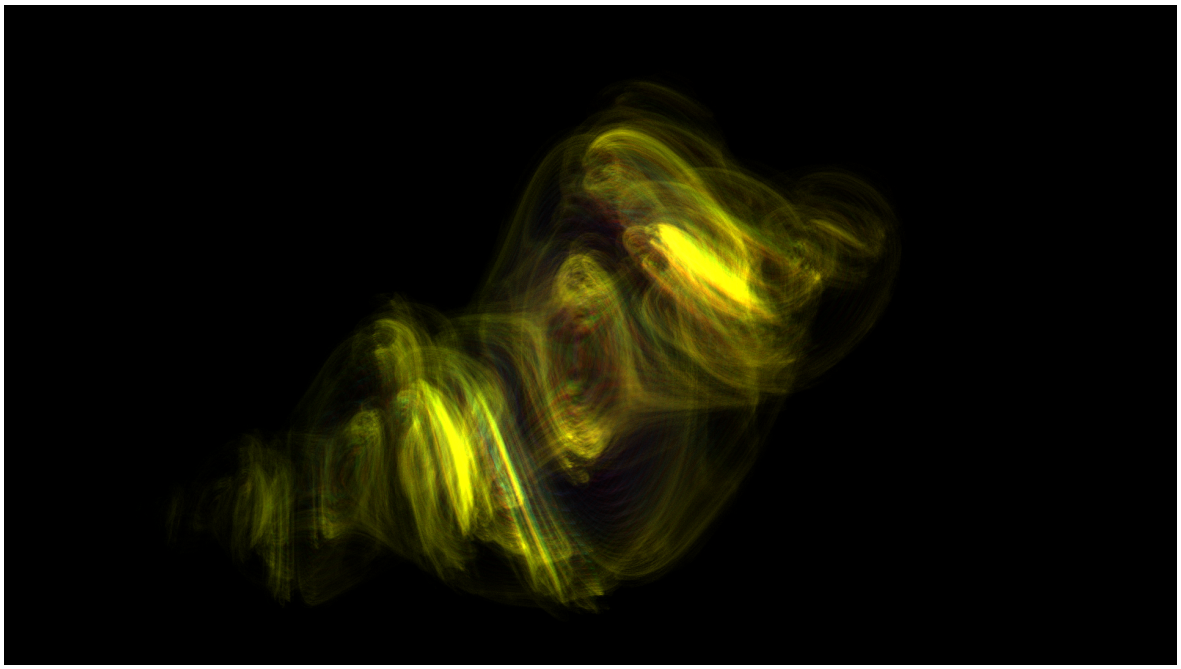
Obr. 4.1: Juliina množina 1 povrchový rez.



Obr. 4.2: Juliina množina 3 povrchové rezy.



Obr. 4.3: Juliina množina 50 povrchových rezov.



Obr. 4.4: Juliina množina 50 volumetrických rezov.

Záver

Hlavným cieľom tejto bakalárskej práce bolo zobraziť štvorrozmerné fraktály v trojrozmernom priestore, pričom štvrtú dimenziu fraktálu reprezentujeme pomocou farby. Tento prístup nám umožnil vizuálne preskúmať a analyzovať komplexné štruktúry týchto matematických objektov.

Na začiatku sme sa venovali teoretickým základom fraktálov a kvaternionov. Následne sme preskúmali rôzne techniky vizualizácie a optimalizácie, pričom sme čerpali inšpiráciu z existujúcich systémov.

Na základe tejto analýzy sme navrhli architektúru aplikácie, ktorú sme implementovali s využitím technológie OpenGL a shader programov.

V rámci implementácie sme sa sústredili na dve hlavné zobrazovacie techniky, výpočet pozdĺž lúča a zobrazovanie voxelov pomocou algoritmu Fast Voxel Traversal Algorithm. Obidve techniky sme optimalizovali pomocou algoritmu Ray-Box Intersection, ktorý výrazne zrýchlil výpočty priesečníkov lúčov s obálkou scény. Pre zobrazovanie voxelov sme museli najprv vykonať potrebné výpočty, preto sme implementovali compute shader program.

Porovnaním jednotlivých metód sme zistili, že zobrazovanie voxelov je efektívnejšia technika, ale vyžaduje si vysokú pamäťovú kapacitu. Do aplikácie sme integrovali kameru, ktorá zabezpečuje pohyb v 3D priestore, a vytvorili sme používateľské rozhranie, ktoré ponúka rozsiahle možnosti pre konfiguráciu scény.

Budúci vývoj by mohol zahŕňať implementáciu ďalších techník zobrazovania alebo štruktúr, ktoré by znižovali pamäťové nároky, napríklad dátová štruktúra octree. Ďalšou budúcou implementáciou by mohlo byť rozšírenie používateľského rozhrania, ktoré by umožnilo používateľovi priamo zadávať iteratívne funkcie fraktálov, ktoré by sa zobrazovali.

Táto práca nielenže položila pevné základy pre ďalší vývoj v oblasti vizualizácie viacrozmerných fraktálov, ale aj prispela k lepšiemu pochopeniu a skúmaniu týchto fascinujúcich matematických objektov.

Literatúra

[1] GLFW documentation. Dostupné na: `https://www.glfw.org/documentation.html`. Navštívené: 2024-05-20.

[2] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-time rendering*. AK Peters/crc Press, 2019.

[3] John Amanatides, Andrew Woo, et al. A fast voxel traversal algorithm for ray tracing. In *Eurographics*, volume 87, pages 3–10. Citeseer, 1987.

[4] Andreas Maschke. Mandelbulb3D. Dostupné na: `https://github.com/thargor6/mb3d`. Navštívené: 20-05-2024.

[5] Robert Delorto. Fractal dimension and julia sets. Master's thesis, Eastern Washington University, 2013.

[6] James Diebel et al. Representing attitude: Euler angles, unit quaternions, and rotation vectors. *Matrix*, 58(15-16):1–35, 2006.

[7] Andrzej Katunin. *A Concise Introduction to Hypercomplex Fractals*. CRC Press, 2017.

[8] Arie E Kaufman and Klaus Mueller. Overview of volume rendering. *The visualization handbook*, 7:127–174, 2005.

[9] Khronos Group. OpenGL documentation. Dostupné na: `https://www.khronos.org/opengl/`. Navštívené: 20-05-2024.

[10] Khronos Group. OpenGL Rendering Pipeline. Dostupné na: `https://www.khronos.org/opengl/wiki_opengl/images/RenderingPipeline.png`. Navštívené: 2024-05-20.

[11] Khronos Group. Vulkan documentation. Dostupné na: `https://registry.khronos.org/vulkan/`. Navštívené: 20-05-2024.

[12] Krzysztof Marczak. Mandelbulber2. Dostupné na: https://github.com/buddhi1980/mandelbulber2. Navštívené: 20-05-2024.

[13] Microsoft. DirectX documentation. Dostupné na: `https://learn.microsoft.com/en-us/windows/win32/directx`. Navštívené: 20-05-2024.

[14] Dragos Miha and Eugen Strajescu. From wavelength to r g b filter. *University"Politehnicaöf Bucharest Scientific Bulletin, Series D: Mechanical Engineering*, 69(2):77–84, 2007.

[15] Ramakrishnan Mukundan and Ramakrishnan Mukundan. Quaternions. *Advanced Methods in Computer Graphics: With examples in OpenGL*, pages 77–112, 2012.

[16] Heinz-Otto Peitgen, Hartmut Jürgens, Dietmar Saupe, and Mitchell J Feigenbaum. *Chaos and fractals: new frontiers of science*, volume 106. Springer, 2004.

[17] H. Ray, H. Pfister, D. Silver, and T.A. Cook. Ray casting architectures for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):210–223, 1999.

[18] Amy Williams, Steve Barrus, R Keith Morley, and Peter Shirley. An efficient and robust ray-box intersection algorithm. In *ACM SIGGRAPH 2005 Courses*, pages 9–es. 2005.

Príloha A: obsah elektronickej prílohy

V elektronickej prílohe priloženej k práci sa nachádza zdrojový kód programu a súbory s výsledkami experimentov.

Zdrojový kód je zverejnený aj na stránke `https://github.com/adrkoc/BachelorThesis`.