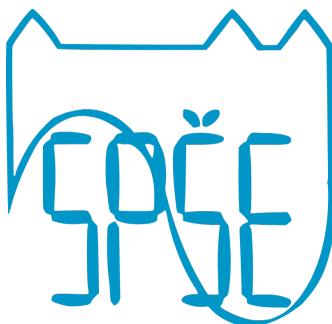


**STREDNÁ PRIEMYSELNÁ ŠKOLA ELEKTROTECHNICKÁ
ZOCHOVA 9, BRATISLAVA**



Golang Ray-Tracer

projekt z praktickej časti odbornej zložky maturitnej skúšky

Meno kandidáta: Matúš Benček

Trieda: 4.D

Vedúci práce: Mgr. Jakub Krcho

Šk. rok: 2024/2025

**STREDNÁ PRIEMYSELNÁ ŠKOLA ELEKTROTECHNICKÁ
ZOCHOVA 9, BRATISLAVA**

Golang Ray-Tracer

Meno kandidáta: Matúš Benček

Trieda: 4.D

Vedúci práce: Mgr. Jakub Krcho

Rozsah: 82 strán, 90 obrázkov

STREDNÁ PRIEMYSELNÁ ŠKOLA ELEKTROTECHNICKÁ ZOCHOVA 9, 811 03 BRATISLAVA



Študijný odbor: 2573M00 - programovanie digitálnych technológií
Meno: Matúš Benček
Trieda: IV.D
Školský rok: 2024/2025

Zadanie praktickej časti odbornej zložky maturitnej skúšky formou obhajoby vlastného projektu

Názov práce: Golang Ray-Tracer

Úloha:

Navrhnite program, ktorý je schopný renderovať 3D scénu pomocou techniky ray tracing v programovacom jazyku Go. Program bude sledovať svetelné lúče, ktoré prechádzajú cez 3D objekty, pričom bude počítať ich interakcie s povrchmi, ako sú odrazy, tieňa a lomy. Hlavným cieľom je dosiahnuť realistické zobrazenie scény na základe presných fyzikálnych výpočtov svetla a jeho správania. Program musí byť optimalizovaný tak, aby dokázal spracovať aj komplexnejšie scény v rozumnom čase.

Technické parametre:

- Golang

Písomná časť:

Odovzdať vo viazanej forme (hrebeňová, termoväzba, lepená...) formátu A4 s počtom strán min. 20, bude obsahovať:

- zadanie praktickej maturitnej práce - originál
- čestné prehlásenie
- podčiarkovanie (nie je povinné)
- zoznam obrázkov, tabuľiek
- obsah
- úvod
- popis použitých technológií (max. na 5 strán) - teoretická časť
- popis technického riešenia - praktická časť
- použitá literatúra podľa normy ISO 690
- záver
- prílohy

Stredná priemyselná škola
elektrotechnická
811 03 Bratislava, Zochova 9



Ing. Marián Beniak, PhD.
riaditeľ

V Bratislave, dňa 31.03.2025

Čestné prehlásenie

Čestne prehlasujem, že problematiku týkajúcu sa náplne zadaného projektu v rámci praktickej časti odbornej zložky maturitnej skúšky formou obhajoby som spracoval sám za pomocí uvedenej použitej literatúry. Inú, ako uvedenú literatúru, som nepoužil.

V Bratislave dňa: 27. 3. 2025

Matúš Benček

Pod'akovanie

Chcel by som sa pod'akovať svojmu vedúcemu práce nie len za pomoc a vedení pri tvorbe maturitného projektu ale aj za celé štyri roky, ktorý vo mňa veril a naučil ma mnoho do budúcnosti.

Obsah

Úvod.....	1
1.0 TEORETICKÁ ČASŤ.....	2
1.1 Ray Tracing.....	2
1.2 Voxel Rendering.....	2
1.3 Ray Marching.....	2
1.4 Podpora Shaderov.....	3
1.5 Použité technológie.....	3
2.0 PRAKTICKÁ ČASŤ.....	4
2.1 Architektúra Projektu GO-Draaw.....	4
2. 1.1 Backend.....	4
2.2 Backend Implementácia Web Servera.....	5
2.2.1 Koncové body.....	5
2.3 Dokumentácia Frontend Komponentov.....	6
2.3.1 Color Picker.....	7
2.3.3 Shader Menu.....	8
2.3.4 Render Options.....	15
2.3.5 Volume Picker.....	21
3.0 Princíp fungovania BVH.....	22
3.2 Surface Area Heuristic (SAH).....	23
3.3 Reprezentácia Trojuholníkov a Materiálové Vlastnosti.....	24
3.3.1 Geometrická Reprezentácia.....	24
3.3.2 Materiálové Vlastnosti.....	25
3.4 Nová Implementácia BVHLean.....	27
3.5 BVH a jej implementácia.....	28
3.5.1 Evolúcia BVH štruktúry.....	29
3.5.2 Optimalizovaná BVHLean.....	29
3.5.3 Optimalizácia BVHLean - porovnanie 2 bounding boxov naraz.....	31
3.5.4 Experimentálna array-based implementácia.....	32
4.0 Podpora Načítavania 3D Geometrie.....	33
4.1 Načítavanie .OBJ Súborov.....	33
5.0 RayTracing Vývoj Funkcionality.....	34
TraceRay - GUI Názov : V1.....	34
TraceRayV2 - GUI Názov : V2.....	36
TraceRayV3 - GUI Názov : V2M.....	37
TraceRayV3Advance - GUI Názov : V2Liner / V2Log.....	38
TraceRayV3AdvanceTexture - GUI Názov : V2LinearTexture / V2LogTexture.....	41
TraceRayV4AdvanceTexture - GUI Názov : V4Linear / V4Log.....	44
TraceRayV4AdvanceTextureLean - GUI Názov : V4LinOptim / V4LogOptim / V4Optim-V2.....	46

6.0 Systém Benchmarkovania a Výkonnostnej Analýzy.....	52
6.1 Zhrnutie Štatistik.....	52
6.2 Technologické Rozdiely Verzií.....	54
6.3 Záver Testov.....	55
6.3.1 Median Graph.....	56
6.3.2 Mean Graph.....	56
6.3.3 STD Graph.....	57
6.3.4 Min Frame Time.....	57
6.3.5 Max Frame Time.....	57
6.3.6 Bottom 10 % Frame Time.....	58
6.3.7 Top 10 % Frame Time.....	59
6.4 Úvod do Benchmarkingu.....	59
6.4.1 Testované Verzie Rendereru.....	59
7.0 Rendrovacie Pomocné Funkcie.....	62
7.1 FresnelSchlick Funkcia.....	62
7.2 GGX Distribučná Funkcia.....	63
8.0 Voxel Rendering.....	64
8.1 Vlastnosti voxelového renderingu.....	65
8.2 Interaktívne editačné možnosti.....	66
8.3 Optimalizácia Renderingu Voxelov.....	67
9.0 Implementácia Objemového Rendering-u.....	69
9.1 Kľúčové vlastnosti objemového renderingu:.....	71
9.2 Optimalizácie Výkonu.....	71
9.3 Fyzikálne Modely Objemu.....	72
10.0 Raymarching – Základný Popis.....	73
10.1 Výhody a Nevýhody Raymarchingu.....	73
10.2 Signed Distance Fields (SDF).....	74
10.3 Vývoj Raymarchingu – Verzie Implementácie.....	75
10.6 Možnosti Budúceho Vylepšenia.....	75
11.0 Podpora Post-Processing Shaderov.....	76
11.1 Úvod do Post-Processingu.....	76
11.2 Jazyk Shaderov Kage.....	76
11.3 Podporované Post-Processing Efekty.....	78
11.3.1 Základné Efekty.....	78
11.3.2 Komplexné Vizuálne Efekty.....	78
11.4 Výhody Implementácie.....	79
Záver.....	80

ZOZNAM OBRÁZKOV

- 1 - [Image - Frontend](#)
- 2 - [Image - Color Picker](#)
- 3 - [Image - Render Bez Shadrov](#)
- 4- [Image - Kontrast Shader](#)
- 5 - [Image - Tint Shader](#)
- 6 - [Image - Bloom Shader](#)
- 7 - [Image - Bloom V2 Shader](#)
- 8 - [Image - Sharpness Shader](#)
- 9 - [Image - Color Mapping Shader](#)
- 10 - [Image -Chromatin Aberration Shader](#)
- 11 - [Image - Edge Detection Shader](#)
- 12 -[Image - Lighten Shader](#)
- 13 - [Image - Vignette Shader](#)
- 14 - [Image - Horná Lišta](#)
- 15 -[Image - Pozícia Kamery](#)
- 16 - [Image - Ukázka Rendera](#)
- 17 - [Image](#)
- 18 - [Image - Vypriemerovaný Render](#)
- 19 - [Image - Ray Marching V1](#)
- 20 - [Image - Ray Marching V2](#)
- 21 - [Image - Ray Marching V2](#)
- 22 - [Image - Render Options GUI](#)
- 23 - [Image - Volume Options GUI](#)
- 24 - [Image - BVH](#)
- 25 - [Image - BVH](#)
- 26 - [Image - Triangle Simple Struct](#)
- 27 - [Image - TriangleBbox Struct](#)
- 28 - [Image - Texture Struct](#)
- 29 - [Image - BVH Porovnanie](#)
- 30 - [Image - BBoxPair Funkcia](#)
- 31 - [Image - BBox Porovanie](#)
- 32 - [Image - V1 Benchmark](#)

- 33 - [Image - V1 Render](#)
- 34 - [Image - V2 Benchmark Table](#)
- 35 - [Image - V2 Benchmark Flame Graph](#)
- 36 - [Image - V2M Benchmark](#)
- 37 - [Image - V2M Render](#)
- 38 - [Image - V2Lin Benchmark Table](#)
- 39 - [Image - V2Lin Flame Graph](#)
- 40 - [Image - V2Lin Render](#)
- 41 - [Image - V2Log Benchmark](#)
- 42 - [Image - V2Log Render](#)
- 43 - [Image - V2Lin Texture Table](#)
- 44 - [Image - V2Lin Texture Flame Graph](#)
- 45 - [Image - V2Lin Texture Render](#)
- 46 - [Image - V2Log Texture Flame Graph](#)
- 47 - [Image - V2Log Texture Table](#)
- 48 - [Image - V2Log Texture Render](#)
- 49 - [Image - V4Lin Table](#)
- 50 - [Image - V4Lin Flame Graph](#)
- 51 - [Image - V4Lin Render](#)
- 52 - [Image - V4Log Table](#)
- 53 - [Image - V4Log Flame Graph](#)
- 54 - [Image - V4Log Render](#)
- 55 - [Image - V4Lin Optim Flame Graph](#)
- 56 - [Image - V4Lin Optim Table](#)
- 57 - [Image - V4Lin Optim Render](#)
- 58 - [Image - V4Log Optim Table](#)
- 59 - [Image - V4Log Optim Flame Graph](#)
- 60 - [Image - V4Log Optim Render](#)
- 61 - [Image - V4V2 Table](#)
- 62 - [Image - V4V2 Flame Graph](#)
- 63 - [Image - V4V2 Render](#)
- 64 - [Image - Normals](#)
- 65 - [Image](#)

- 66 - [Image](#)
- 67 - [Image](#)
- 68 - [Image](#)
- 69 - [Image - Štatistiky](#)
- 70 - [Image - Median](#)
- 71 - [Image - Mean](#)
- 72 - [Image - Štandardná Diviácia](#)
- 73 - [Image - Minimálny čas](#)
- 74 - [Image - Maximálny čas](#)
- 75 - [Image](#)
- 76 - [Image](#)
- 77 - [Image - vypnutie GC počas testu](#)
- 78 - [Image - Implementácia Testu](#)
- 79 - [Image - Fresnel efekt](#)
- 80 - [Image - Voxel Rendering](#)
- 81 - [Image](#)
- 82 - [Image](#)
- 83 - [Image - Voxel reprezentácia](#)
- 84 - [Image - Optimalizácia Voxelov](#)
- 85 - [Image - Volume Rendering](#)
- 86 - [Image](#)
- 87 - [Image - SDF Funkcia](#)
- 88 - [Image - SDF Funkcie](#)
- 89 - [Image - SDF funkcie pre iné Tvary](#)
- 90 - [Image - Príklad Kage Shadera](#)

ZOZNAM SKRATIEK

Skratka	Popis
BVH	Bounding volume hierarchy
SDF	signed distance field
BBOX	Bounding Box
SPD	Speed
Go	Golang
RGBA	Červená / Zelená / Modrá / Alpa
PBR	Physically-Based Rendering

BVH	Bounding volume hierarchy
SDF	signed distance field
BBOX	Bounding Box
SPD	Speed
Go	Golang
RGBA	Červená / Zelená / Modrá / Alpa
PBR	Physically-Based Rendering

Úvod

V súčasnej dobe počítačová grafika zohráva kľúčovú úlohu v mnohých oblastiach, od herného priemyslu až po vedecké vizualizácie. Jednou z najvýznamnejších technológií v tejto oblasti je Ray-Tracing, ktorý umožňuje vytvárať fotorealistické zobrazenia 3D scén simuláciou fyzikálnych vlastností svetla. Táto maturitná práca sa zameriava na implementáciu vlastného 3D engine-u, ktorý využíva práve túto pokročilú technológiu renderovania.

Hlavným cieľom práce je vytvoriť flexibilný a výkonný 3D engine, ktorý bude schopný nielen základného renderovania 3D scén pomocou Ray-Tracingu, ale poskytne aj možnosť využívať rôzne shadre pre pokročilé vizuálne efekty. Významnou súčasťou projektu je implementácia podpory pre renderovanie volumetrických materiálov prostredníctvom technológie Voxel, čo ďalej rozširuje možnosti vizualizácie komplexných objektov a efektov.

Pre implementáciu bol zvolený programovací jazyk Golang, ktorý sa vyznačuje niekoľkými kľúčovými výhodami. Prvou je jeho efektívna podpora multiprocesingu prostredníctvom Go rutín, čo je esenciálne pre optimalizáciu výkonu pri ray-tracingu. Druhou výhodou je jeho výkonnosť, ktorá sa približuje tradičným systémovým jazykom ako C a C++. Pre implementáciu shaderových programov bude využitý jazyk Kage, ktorý bol vyvinutý pre Ebiten 2D engine. Kage poskytuje intuitívnu syntax inšpirovanú jazykom Go, čo umožňuje efektívny vývoj shaderov.

Aplikácia poskytne užívateľom možnosť interaktívne upravovať vlastnosti 3D geometrie, vrátane farieb a rôznych aspektov materiálov. Dôraz je kladený na optimalizáciu výkonu, aby bolo možné renderovať scény v realistickom čase.

1.0 TEORETICKÁ ČASŤ

1.1 Ray Tracing

Ray tracing je pokročilá technológia renderovania 3D scén, ktorá simuluje fyzikálne vlastnosti svetla pre dosiahnutie fotorealistických zobrazení. Využíva sa na vytváranie detailných a presných odrazov, lomov a tieňov, čo prispieva k celkovému realizmu renderovaného obrazu. Táto technológia je klúčová pre dosiahnutie vysokého stupňa vizuálnej kvality v 3D grafike. Ray tracing sleduje cestu jednotlivých lúčov svetla od zdroja svetla cez scénu až po oko pozorovateľa (alebo kamery). Pri každom strete lúča s objektom sa vypočítajú odrazy, lomy a tieňy na základe vlastností materiálu objektu a uhla dopadu lúča. Tento proces sa opakuje pre množstvo lúčov, čím sa vytvorí detailný a realistický obraz.

1.2 Voxel Rendering

Voxel rendering je technológia, ktorá umožňuje renderovanie volumetrických materiálov. Na rozdiel od tradičného renderovania, ktoré pracuje s povrchmi objektov, voxel rendering pracuje s objemovými dátami, čo umožňuje vizualizáciu komplexných efektov ako dym, hmla alebo oheň. Táto technológia rozširuje možnosti vizualizácie a pridáva do 3D scén ďalšiu úroveň detailu a realizmu. Voxely sú 3D pixely, ktoré reprezentujú objemové prvky v priestore. Voxel rendering rozdeľuje 3D priestor na mriežku voxelov a každému voxelu priradí informáciu o materiáli, farbe a hustote. Renderovanie sa potom vykonáva na základe týchto objemových dát, čo umožňuje vytváranie efektov, ktoré sú ťažko dosiahnuteľné tradičnými metódami.

1.3 Ray Marching

Ray marching je technika renderovania, ktorá sa používa na vizualizáciu implicitných povrchov definovaných matematickými funkciemi. Táto technika je efektívna pre renderovanie fraktálov, oblakov a iných organických tvarov, ktoré je ťažké modelovať tradičnými metódami. Ray marching umožňuje vytváranie zložitých a detailných geometrických štruktúr s relatívne nízkymi výpočtovými nárokmi. Namiesto sledovania lúčov svetla, ray marching postupuje pozdĺž lúča v

malých krokov a testuje, či sa lúč pretína s povrchom objektu. Ak sa pretne, vypočítá sa farba a osvetlenie daného bodu. Táto technika je vhodná pre renderovanie objektov, ktoré nemajú explicitne definované povrhy, ale sú popísané matematickými funkciami.

1.4 Podpora Shaderov

Shadery sú programy, ktoré sa spúšťajú na grafickom procesore a umožňujú pokročilé vizuálne efekty a úpravy renderovaného obrazu. V kontexte 3D engine-u umožňujú shadery implementáciu rôznych post-processing efektov, ako sú úpravy farieb, kontrastu, ostrosti a ďalšie, čo prispieva k finálnemu vizuálnemu štýlu a kvalite renderovaného obrazu. Shadery umožňujú programovateľné renderovanie, čo znamená, že vývojári môžu prispôsobiť grafický výstup podľa svojich potrieb. Používajú sa na implementáciu rôznych efektov, ako sú textúrovanie, osvetlenie, tiene, odrazy, lomy a ďalšie.

1.5 Použité technológie

Backend:

- Golang: Programovací jazyk použitý pre backendovú časť aplikácie. Golang, tiež známy ako Go, je kompilovaný, staticky typovaný programovací jazyk vyvinutý spoločnosťou Google. Je známy svojou jednoduchosťou, efektivitou a výkonom, čo ho robí vhodným pre vývoj backendových aplikácií, ktoré vyžadujú vysokú konkurentnosť a škálovateľnosť.
- Echo Framework: Webový framework v Golangu, ktorý uľahčuje vývoj webových aplikácií. Echo je ľahký a vysoko výkonný HTTP router a webový framework pre Golang. Poskytuje sadu nástrojov a funkcií, ktoré uľahčujú vývoj RESTful API a webových aplikácií, ako je smerovanie, spracovanie požiadaviek a odpovedí, middleware a ďalšie.

Frontend:

- Vue.js: JavaScriptový framework použitý pre vývoj frontendovej časti aplikácie. Vue.js je progresívny JavaScriptový framework pre budovanie používateľských rozhraní. Je navrhnutý tak, aby bol postupne adoptovateľný, čo znamená, že ho možno ľahko integrovať do existujúcich projektov. Vue.js sa zameriava na

vrstvu pohľadu a uľahčuje vývoj interaktívnych a dynamických webových aplikácií.

Shader programy:

- Kage: Jazyk vyvinutý pre Ebiten 2D engine, použitý na implementáciu shaderových programov. Kage je špecifický jazyk pre písanie shaderov pre Ebiten, čo je jednoduchý a prenosný 2D grafický engine v Golangu. Používa sa na vytváranie vlastných vizuálnych efektov a úpravu renderovaného obrazu v 2D grafike.

2.0 PRAKTIČKÁ ČASŤ

2.1 Architektúra Projektu GO-Draaw

Projekt je rozdelený na dve hlavné časti:

- Frontend: Vue.js
- Backend: Golang with Echo Framework a RayTracer

2. 1.1 Backend

- Vyvinutý v **Go (Golang)** s použitím **Echo frameworku**
- Pozostáva z dvoch hlavných komponentov:
 - **Ray-tracing engine:** Jadro výpočtového systému pre renderovanie
 - **Webový server:** Zabezpečuje komunikáciu s frontendovou časťou
- Backend beží asynchronne vo vlastných go-rutinách, čo minimalizuje potrebu zložitého manažmentu stavu a používania mutexov s pozitím unsefe, čím sa dosahuje vyšší výkon a lepšia odozva systému.
- Táto architektúra umožňuje efektívne oddelenie prezentačnej vrstvy od výpočtovej, pričom zachováva vysokú mieru interaktivity pre používateľa a zároveň poskytuje výkonný rendering komplexných 3D scén.

2.2 Backend Implementácia Web Servera

V tejto časti sa budeme zaoberať implementáciou webového servera určeného na komunikáciu s frontendom. Backend je postavený na asynchronnej architektúre, pričom beží vo vlastnej Go rutine, aby sme sa vyhli zložitému spravovaniu stavu. Na tento účel sme využili knižnicu Usefe, ktorá umožňuje priame pristupovanie k pamäti.

2.2.1 Koncové body

V tejto časti podrobnejšie opíšeme dostupné API endpointy a štruktúru dát, ktoré sa prenášajú medzi frontendom a backendom.

- **POST /submitColor** : Slúži na odoslanie farby a materiálových vlastností na backend.
- **POST /submitVoxel** : Slúži na úpravu vlastných voxelov.
- **POST /submitTextures** : Odosiela textúrové údaje na backend.
- **POST /submitRenderOptions** : Umožňuje odoslať konfiguráciu renderovania.
- **POST /submitShader** : Umožňuje odoslať nastavenia shaderu.
- **GET /getCameraPosition** : Získa aktuálnu pozíciu kamery.
- **POST /moveToPosition** : Presunúť kameru na určenú pozíciu
- **GET /getCurrentImage** : Získa aktuálne vyrenderovaný obrázok.
- **GET /getSpheres** : Získa aktuálne vlastnosti SDF objektov (napr. pozícia, farba). API odošle na frontend pole objektov.
- **GET /getTypes** : Odosiela mapu typov objektov s ich ID na frontend.
- **POST /updateSphere** : Slúži na odoslanie upraveného SDF objektu späť na backend.
- **POST /moveCamera** : Umožňuje vygenerovať trasu, po ktorej sa má kamera pohybovať v 3D scéne.

2.3 Dokumentácia Frontend Komponentov

V tejto časti sa budeme venovať dokumentácii frontend komponentov, ktoré slúžia na interakciu s 3D scénami vyrenderovanými ray tracerom. Prostredníctvom týchto komponentov môžeme meniť materiálové vlastnosti objektov, upravovať nastavenia renderovania alebo priamo ovládať kameru v scéne.

Frontend sme navrhli tak, aby umožňoval jednoduchú a intuitívnu manipuláciu so scénou bez potreby zásahov do zdrojového kódu ray tracera. Umožňuje nám napríklad upravovať farby a materiály objektov, nastavovať vlastnosti renderingu, meniť pozíciu a pohyb kamery či upravovať vlastnosti SDF objektov.

Dôvodom, prečo sme sa rozhodli implementovať používateľské rozhranie (GUI) práve za pomoci webových technológií, je ich aktuálna jednoduchosť, široká podpora a rýchly vývoj. Moderné webové technológie nám poskytujú možnosť rýchlo vytvárať interaktívne a flexibilné používateľské rozhranie bez potreby inštalácie akýchkoľvek externých aplikácií. Okrem toho ich využitie umožňuje jednoduché rozšírenie alebo úpravy do budúcnosti, napríklad o nové funkcie, ďalšie typy objektov alebo pokročilé nastavenia renderovania.

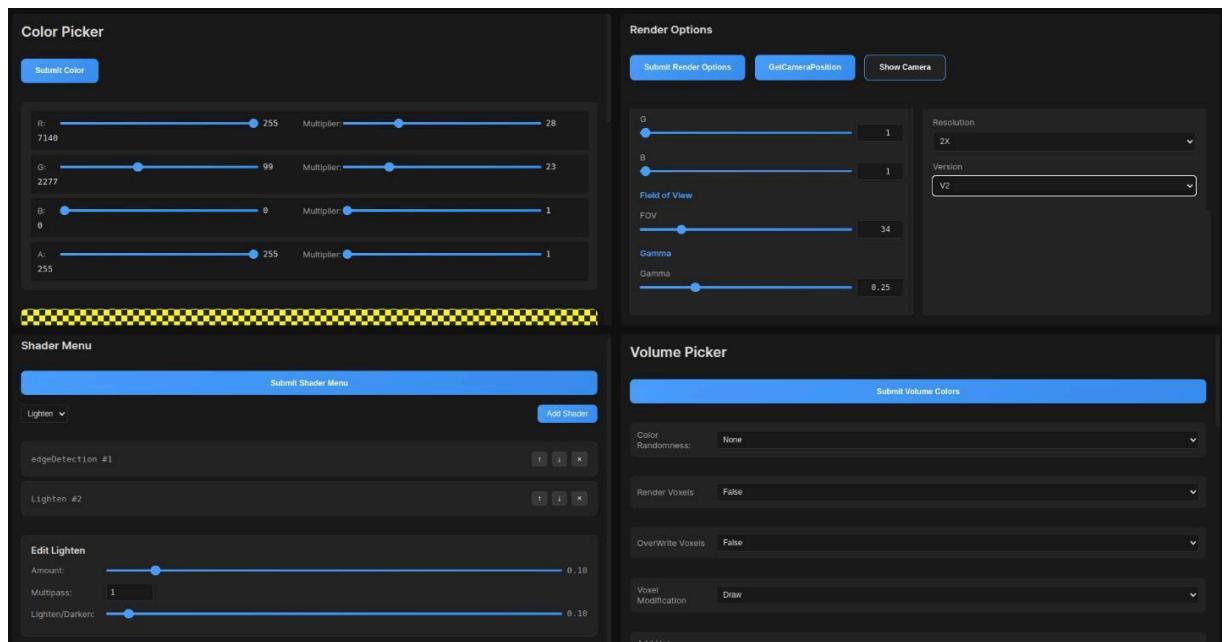


Image - 2 - |2

2.3.1 Color Picker

Pre efektívnu prácu so scénou sme vytvorili dvojicu kľúčových frontend komponentov — Color Picker a Texture Color Picker, ktoré nám umožňujú interaktívne upravovať vizuálne vlastnosti objektov v ray traceri.

Komponent Color Picker slúži na výber farby pomocou štyroch farebných kanálov (R, G, B, A) a voliteľného násobiteľa, ktorý upravuje výslednú intenzitu farby. Hodnoty jednotlivých kanálov môžeme intuitívne nastavovať a následne ich aplikovať na vybrané trojuholníky v scéne. Výsledná farba je následne odoslaná na backend, kde je spracovaná a použitá pri vykresľovaní. Color Picker taktiež umožňuje nahrávať normal mapy a upravovať základné materiálové vlastnosti ako je odrazivosť, drsnosť povrchu, kovový lesk či intenzita špeculárneho odrazu. Tieto vlastnosti môžeme nastavovať plynulo pomocou posuvníkov, čím získavame okamžitú spätnú väzbu priamo vo vizualizácii.

Pre pokročilejšiu prácu s materiálmi sme vytvorili aj komponent Texture Color Picker, ktorý umožňuje nahrávanie a editáciu vlastných textúr. Užívatelia môžu jednoducho nahrať textúru s rozlíšením 128×128 a prepojiť ju s objektmi v scéne. V kombinácii s Color Pickerom môžeme dosiahnuť detailnejšie a realistickejšie materiály. Okrem základnej textúry je možné pripojiť aj normal mapu, pričom máme možnosť voliť medzi rôznymi typmi normalizácie hodnôt. Pre jednoduché generovanie normal máp sme do GUI pridali aj rýchly odkaz na online nástroj, ktorý umožňuje ich okamžité vytvorenie.

Pomocou týchto komponentov môžeme jednoducho experimentovať s materiálmi a vzhľadom scény bez potreby zásahov do samotného kódu ray traceru. Upravovať vieme nielen farbu a textúru, ale aj vlastnosti materiálu ako sú odraz, drsnosť, kovový efekt či špeculárne zvýraznenie. Výsledkom je flexibilné a interaktívne prostredie, ktoré nám umožňuje vizuálne vylepšovať scénu v reálnom čase.

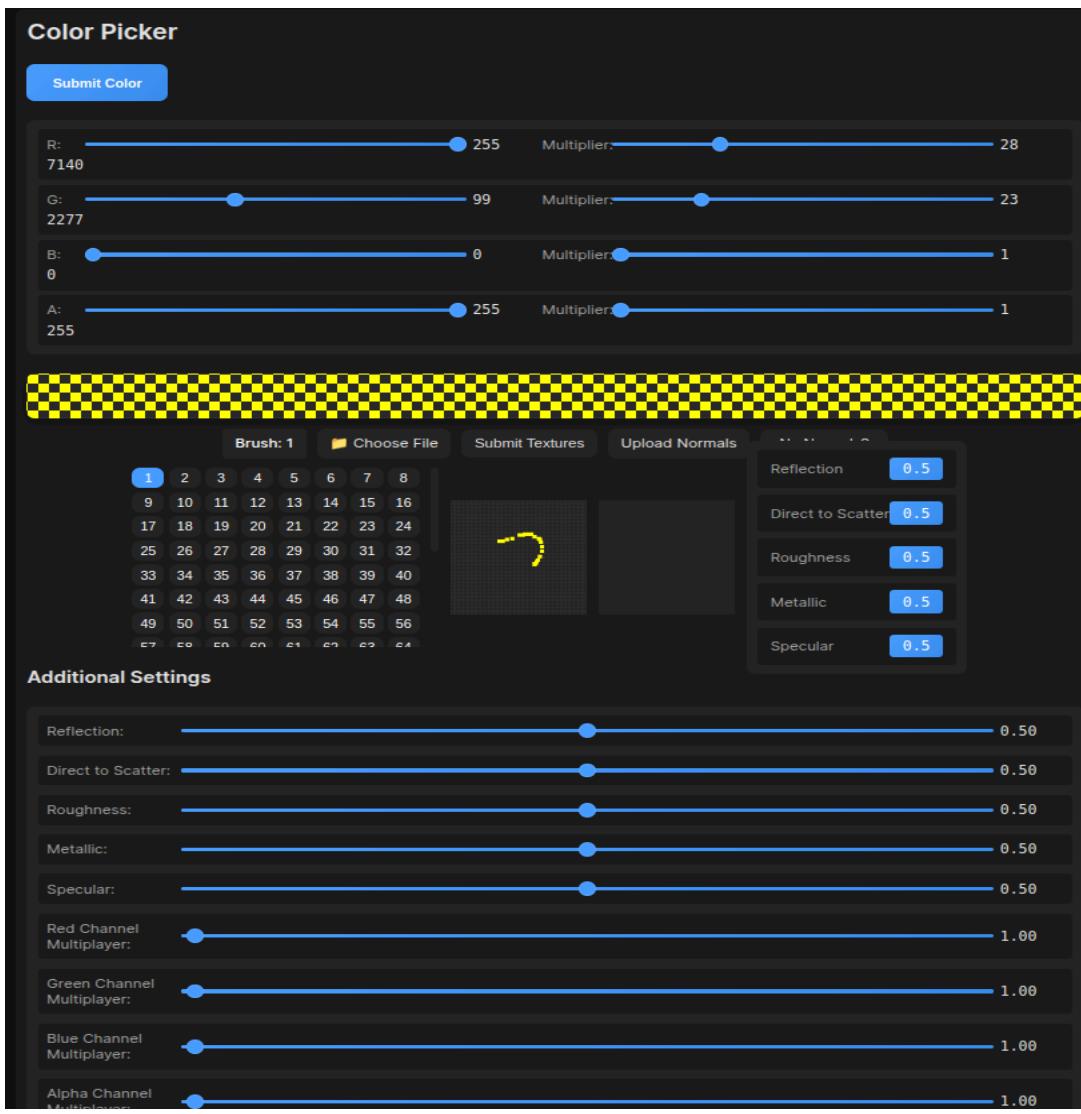


Image - 3 - \2

2.3.3 Shader Menu

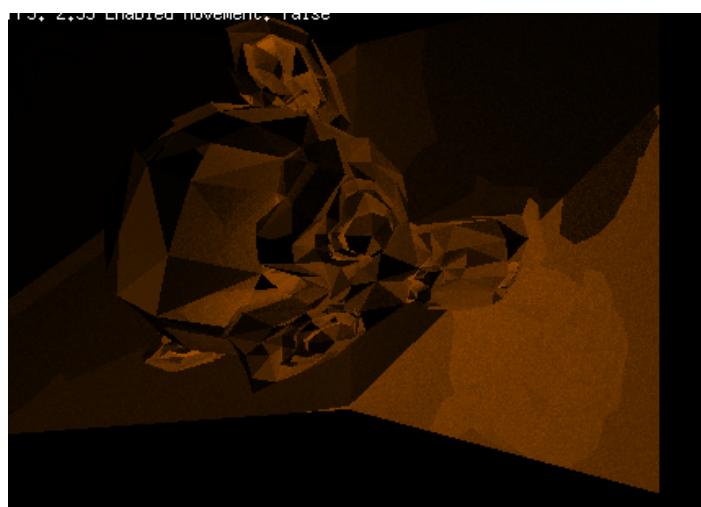
Komponent Shader Menu slúži na výber a konfiguráciu post-processing shaderov, ktoré sú použité na úpravu už vyrendrovaného obrázka. Tieto shadery sú napísané v Kage shader jazyku, ktorý je podobný syntaxe Go, čo zjednodušuje prácu pre vývojárov, ktorí sú oboznámení s týmto jazykom. Keďže shadery bežia na GPU, ich výpočty sú výrazne rýchlejšie v porovnaní s procesorom, pretože GPU je optimalizovaná na paralelné spracovanie dát. Tento prístup zaručuje vysoký výkon pri aplikovaní rôznych efektov na obrázky. Ďalšou výhodou aktuálnej implementácie je, že pridávanie nových shaderov je veľmi jednoduché a nevyžaduje veľké množstvo kódu.

Shader Menu umožňuje vytvárať reťazce post-processing shaderov, ktoré môžu upravovať obrázok krok po kroku. Napríklad, pôvodný obrázok môže byť najprv upravený kontrastom, potom môže nasledovať aplikácia tintu, a na konci sa zobrazí finálny obrázok. Tento proces sa môže prispôsobovať podľa potreby používateľa, pričom každý shader je aplikovaný postupne, čím sa dosahujú požadované vizuálne efekty.

Používateľské rozhranie pre správu shaderov umožňuje jednoduchý výber konkrétnych shaderov a ich konfiguráciu pomocou nasledujúcich parametrov. K dispozícii je tlačidlo Pridať Shader, ktoré umožňuje pridať nový shader do zoznamu, a tlačidlo Odoslať Shader Menu, ktoré zabezpečuje uloženie aktuálnych nastavení. Parametre shaderov zahŕňajú amount, čo je podiel upraveného obrázku, ktorý sa pridá do výsledného renderingu, a multipass, ktorý určuje počet po sebe nasledujúcich aplikácií shaderu, čím je možné dosiahnuť zložitejšie efekty.

Render Bez Shadrov

Tento obrázok slúži ako referenčný bod na porovnanie výsledného obrazu bez aplikácie akýchkoľvek post-processing shaderov. Zobrazuje pôvodný vyrendrovaný obrázok, ktorý neboli podrobene žiadnym dodatočným vizuálnym úpravám. Tento základný obrázok poskytuje jasný prehľad o tom, ako vyzerá scéna bez akýchkoľvek farebných či kontrastových zmien, a slúži ako východiskový bod pre hodnotenie účinnosti aplikovaných post-processing efektov. Takto je možné porovnávať rozdiely medzi pôvodným obrazom a tým, ktorý bol upravený pomocou shaderov, a analyzovať, ako jednotlivé efekty ovplyvňujú celkový vizuálny dojem.



Podporované Shadery

V tejto sekcii popisujeme implementované shadery, ktoré umožňujú aplikáciu rôznych post-processing efektov ako úprava farieb, kontrastu a jasu.

Kontrast

Kontrast shader slúži na úpravu kontrastu obrázka a umožňuje zlepšiť vizuálnu ostrosť a detailnosť. Tento shader upravuje rozdiel medzi svetlými a tmavými oblastami obrazu, čím zvyšuje dynamiku scén. Základná filozofia jeho fungovania spočíva v tom, že zvyšuje jas svetlých pixelov a znižuje jas tmavých, čím sa obraz stáva výraznejším a vizuálne intenzívnejším. Shader má niekoľko vstupných parametrov, ktoré umožňujú jemné doladenie efektu:

- **Amount:** Tento parameter určuje, aký podiel upraveného obrázka sa má aplikovať na aktuálny render. Vyššia hodnota znamená väčšiu intenzitu aplikovaného kontrastu.
- **Sila kontrastu:** Tento parameter určuje, ako agresívne bude kontrast aplikovaný. Vyššia hodnota vedie k výraznejšiemu zvýrazneniu rozdielov medzi svetlými a tmavými časťami obrazu.
- **Multipass:** Tento parameter určuje, kol'kokrát sa má shader aplikovať za sebou. Viacnásobná aplikácia (multipass) umožňuje postupné zvyšovanie kontrastu, čo môže viesť k ešte silnejšiemu efektu.

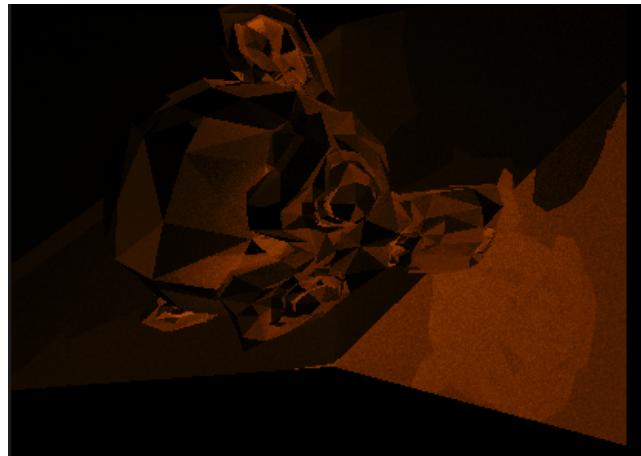


Image - 5 - \2

Tint

Tint shader funguje na báze primiešania malej dávky farby do pôvodného obrázka, čím sa mení jeho farebný nádych. Parametre shaderu umožňujú jemné doladenie efektu:

- **Množstvo:** Určuje pomer medzi pôvodným obrázkom a upraveným obrázkom s pridanou farbou.
- **Multipass:** Určuje, koľkokrát sa má shader aplikovať, čo umožňuje intenzívnejší efekt.
- **Tint farba:** Definuje farbu, ktorá sa primieša do obrázka.
- **Sila Tint shaderu:** Určuje, aký silný bude efekt pridaného farebného nádychu.

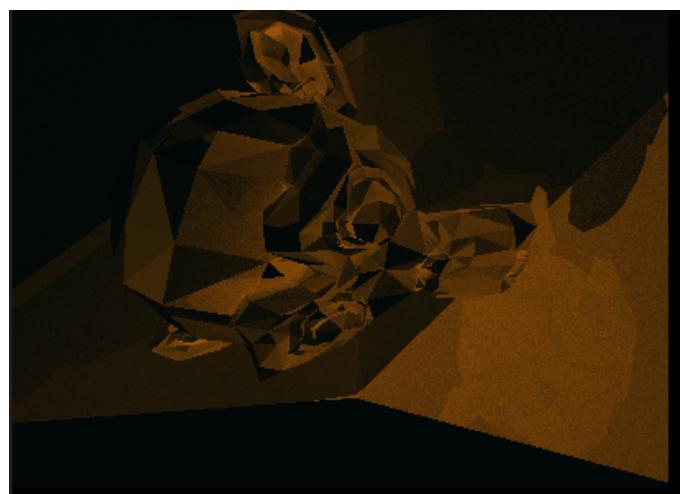


Image - 6 - \2

Bloom

Bloom shader pridáva efekt rozžhavených svetelných oblastí, čím zvyšuje vizuálnu jasnosť a zintenzívnuje svetelné efekty v obraze. Parametre shaderu umožňujú jemné nastavenie efektu:

- **Množstvo:** Určuje pomer medzi pôvodným obrázkom a aplikovaným bloom efektom.
- **Multipass:** Určuje, koľkokrát sa má bloom efekt aplikovať, čím sa zvyšuje jeho intenzita.
- **Prahová hodnota:** Nastavuje, ktoré oblasti obrazu sa budú považovať za dostatočne jasné na aplikáciu bloom efektu.
- **Intenzita:** Určuje, aký silný bude bloom efekt na svetelné oblasti obrázka.

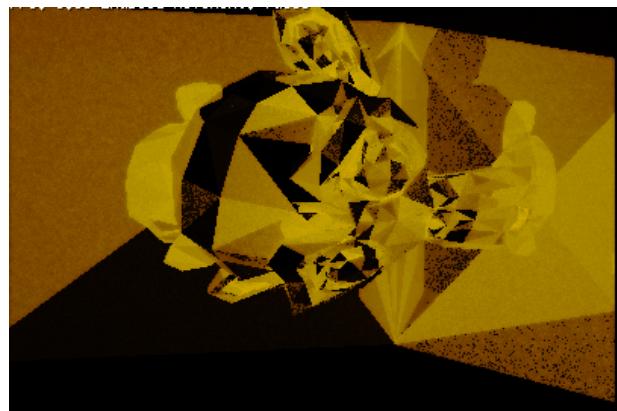
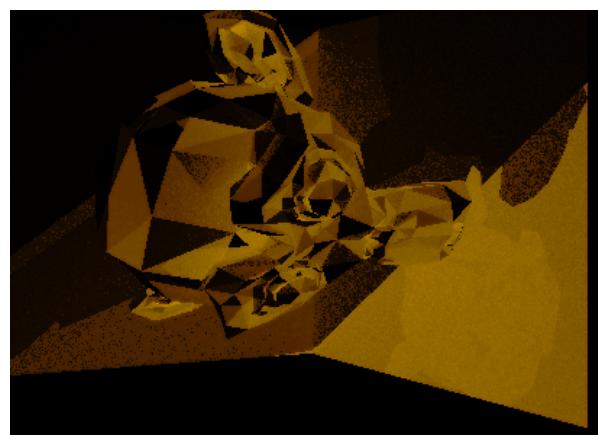


Image - 7 - \2

BloomV2

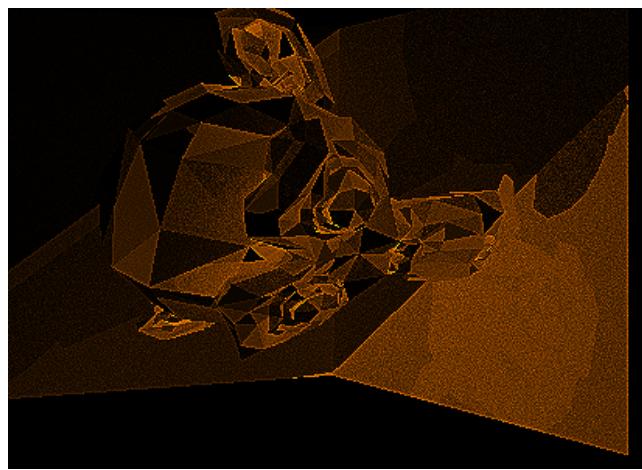
BloomV2 vylepšuje predchádzajúcu verziu a používa Gaussiánske rozmazanie na dosiahnutie prirodzenejšieho a hladšieho efektu rozžiarenia svetelných oblastí v obraze.



Ostrosť

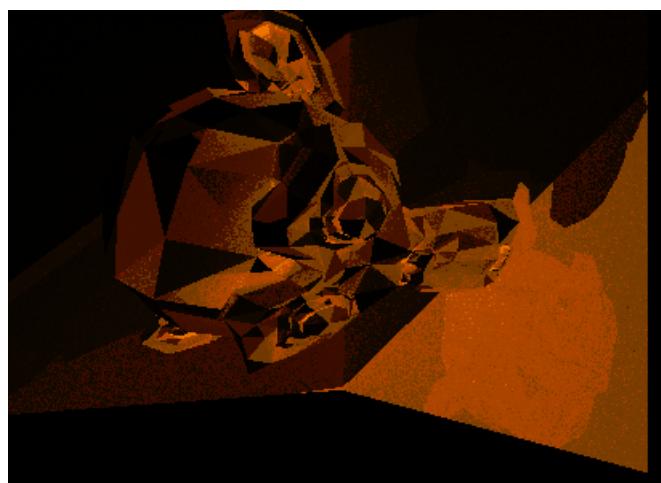
Tento shader zlepšuje viditeľnosť jemných detailov a ostrosť hraníc v obraze. Parametre shaderu umožňujú jemné nastavenie efektu:

- **Množstvo:** Určuje intenzitu aplikovaného efektu ostrosti.
- **Multipass:** Určuje, koľkokrát sa má filter aplikovať, čo umožňuje zvýšiť účinnosť efektu.
- **Sila filtra:** Nastavuje, ako silno bude efekt ovplyvňovať obraz a jeho detaily.



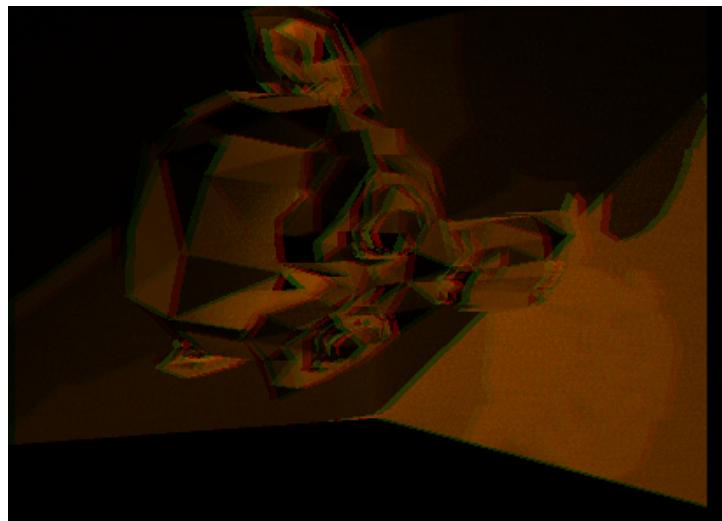
Mapovanie Farieb

Úlohou daného shaderu je obmedziť počet hodnôt pre jednotlivé farebné kanály, čím sa dosahuje zaujímavá textúra.



Chromatická Aberácia

Tento efekt simuluje rozptyl svetla cez objektív, kde sa farebné kanály (R, G, B) zobrazujú s malým posunom



Detekcia Hrán Shader

Tento shader využíva Sobelov filter na detekciu hran v obraze. Po identifikovaní hran sú tieto oblasti zafarbené definovanou farbou, čo umožňuje zvýrazniť kontúry a detaily v scé

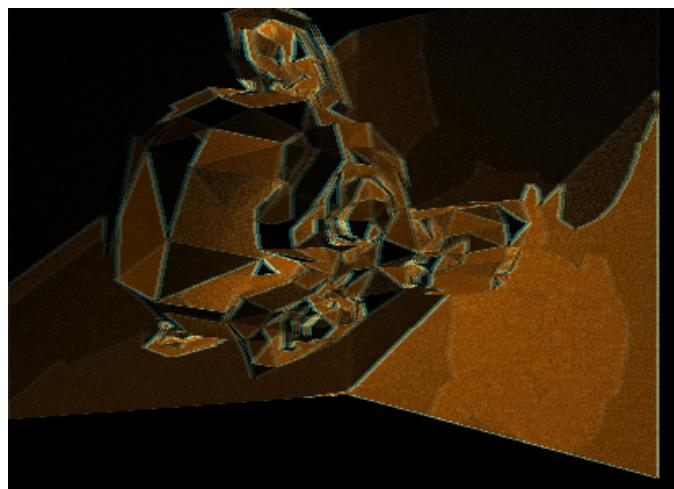


Image - 12 - \2

Zosvetlenie

Úlohou tohto shaderu je pridať možnosť zosvetliť alebo stmavit rendrovaný obraz. Týmto sa dá jednoducho upraviť celkový jas scény, čím sa dosiahne požadovaný vizuálny efekt.

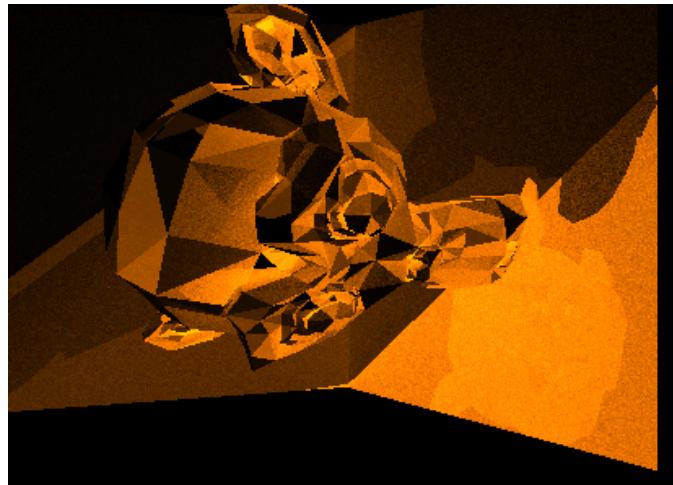


Image - 13 - \2

Vignette

Tento shader kombinuje aspekty bloom efektu, ktorý zvýrazňuje svetlejšie časti obrazu, a zároveň aplikuje Vignette efekt, ktorý tmavší okraje obrazu. Týmto spôsobom sa dosahuje výrazný kontrast medzi centrom a okrajmi scény, čo zvyšuje vizuálny dojem.

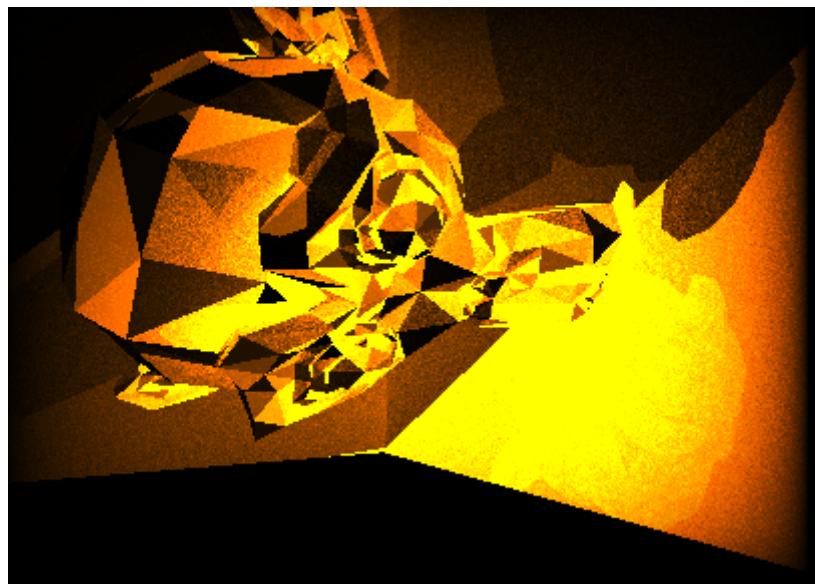


Image - 14 - \2

2.3.4 Render Options

V rámci možnosti interaktívneho nastavovania Ray Tracing enginu sme vytvorili frontend komponentu Render Options, ktorá umožňuje užívateľovi meniť rôzne parametre renderovania v reálnom čase. Tento komponent je navrhnutý tak, aby

poskytol flexibilitu a jednoduché ovládanie pri nastavovaní parametrov, ktoré ovplyvňujú výsledný obraz.

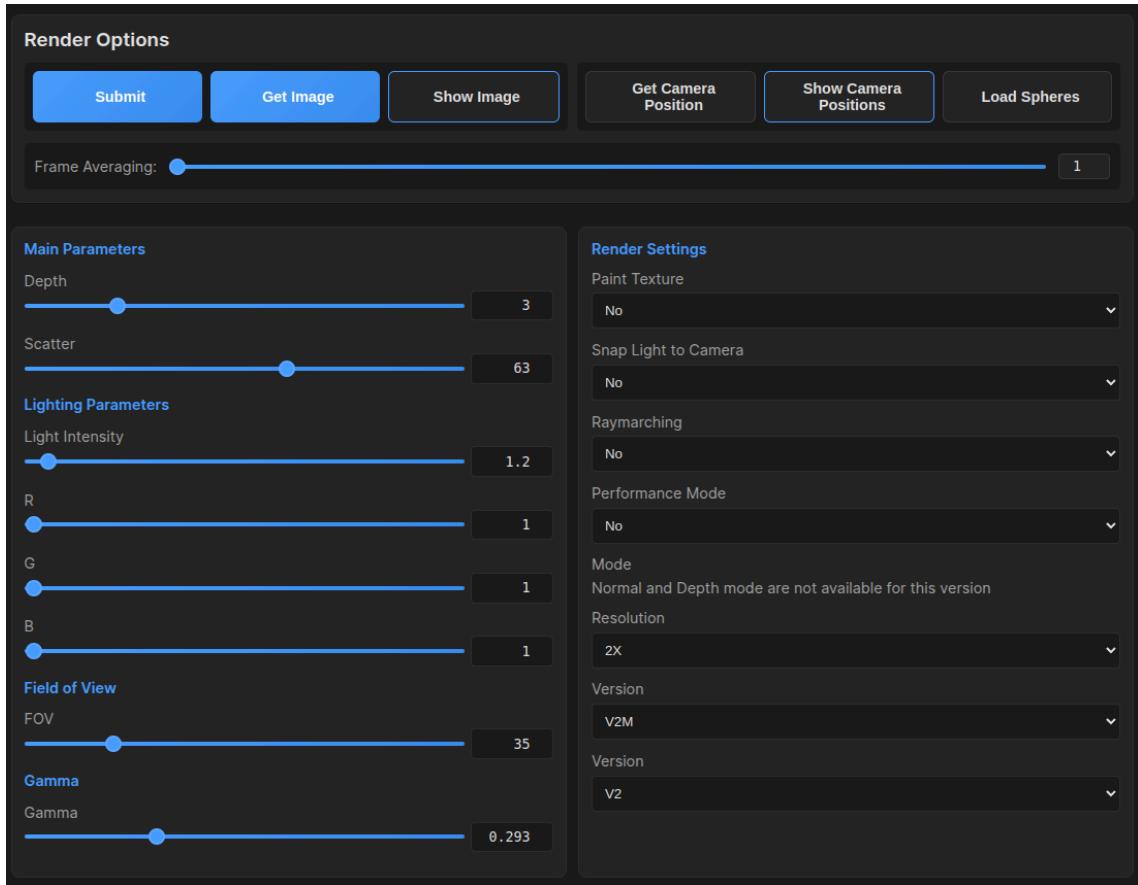


Image - 23 - |2

Tlačidlo „Odoslať Render Možnosti“ slúži na odoslanie aktuálnych nastavení renderovania do enginu. Ďalej, tlačidlo „Získať Pozíciu Kamery“ umožňuje získať aktuálnu pozíciu kamery, zatiaľ čo možnosť „Skryť/Zobrazit Pozíciu Kamery“ umožňuje skryť alebo zobraziť aktuálnu pozíciu kamery na obrazovke. Tlačidlo „Získať Vyrendrovaný Obrázok“ slúži na stiahnutie aktuálneho vyrendrovaného obrázka, a tlačidlo „Ukázať Vyrendrovaný Obrázok“ poskytuje možnosť zobraziť vyrendrovaný obrázok v rámci komponenty. Tlačidlo „Načítať SDF Objekty“ slúži na načítanie SDF (Signed Distance Field) objektov, ktoré sa používajú pri raymarching algoritme. Posuvník na určenie snímkov umožňuje nastaviť počet snímkov, z ktorých sa následne vytvorí výsledný obrázok.

V menu Pozície Kamery je možné zobraziť aktuálne získané pozície kamery. Užívateľ môže vybrať konkrétnu pozíciu a presunúť kameru na túto pozíciu,

alebo vytvoriť animáciu medzi viacerými pozíciami kamery. Tento komponent poskytuje flexibilitu pri práci s kamerou a uľahčuje ovládanie pohybu v 3D priestore.

Menu Ukážky Render-u slúži na zobrazenie vyrendrovaného obrázka, pričom v počiatočnej fáze je renderovaný obrázok šumnejší, pričom spriemerovanie viacerých snímkov poskytuje čistejší a detailnejší výsledok. Hlavné parametre renderingu zahŕňajú hĺbku, ktorá určuje počet odrazov svetla vykonávaných pri renderovaní, a rozptyl, ktorý určuje počet lúčov rozptylených z povrchu objektu, čím sa zvyšuje detailnosť a jemnosť výsledného obrazu.

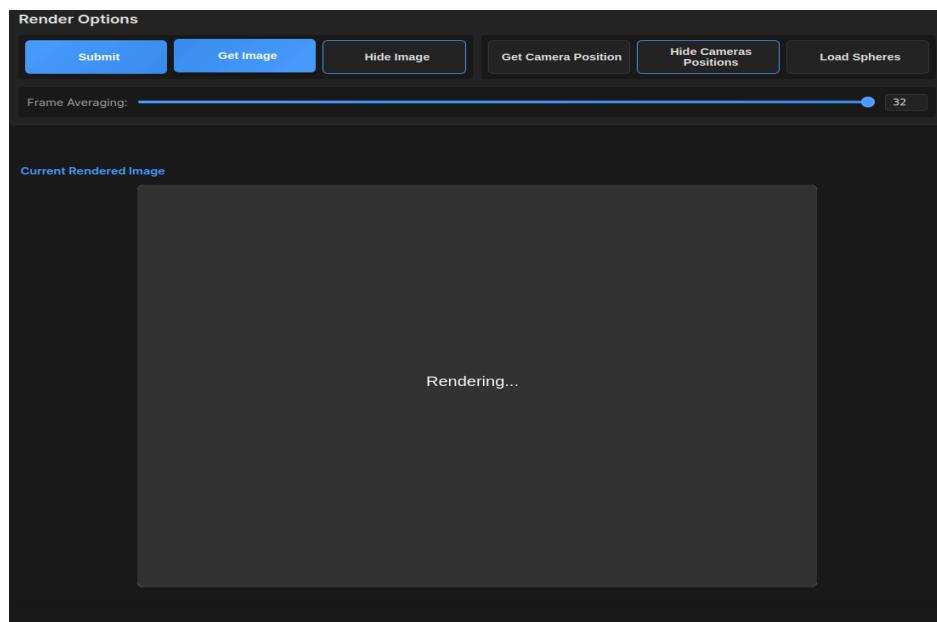
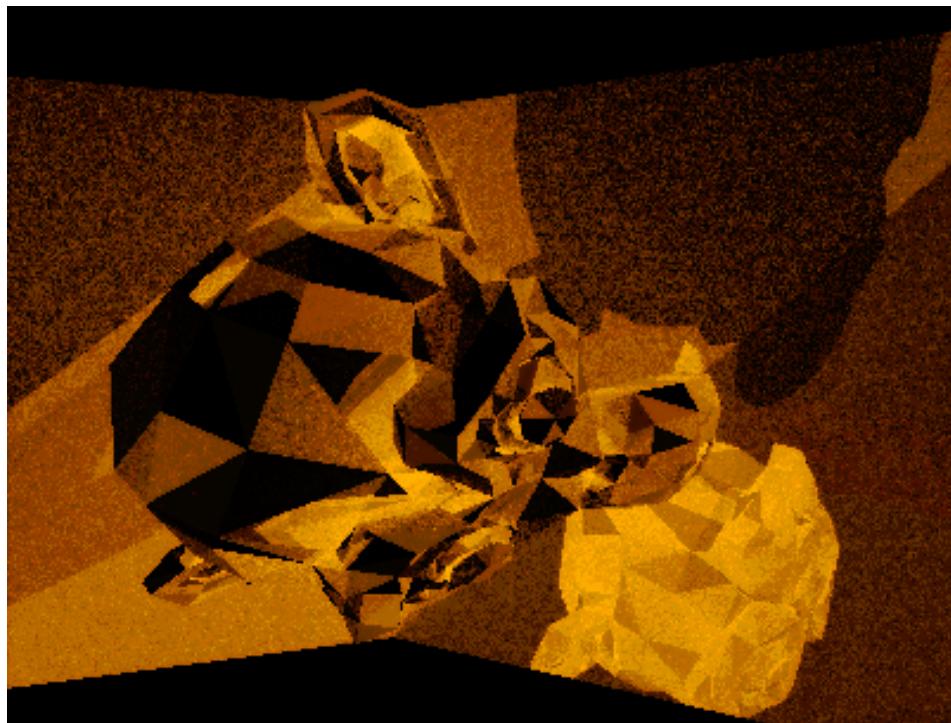
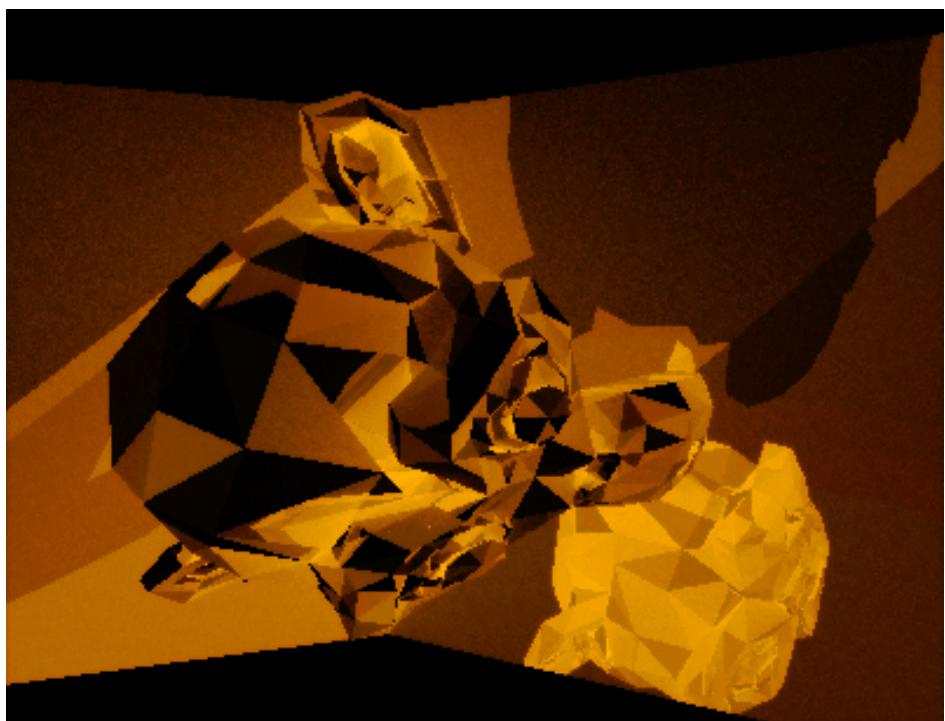


Image - 17 - \2



Normálny Obrázok

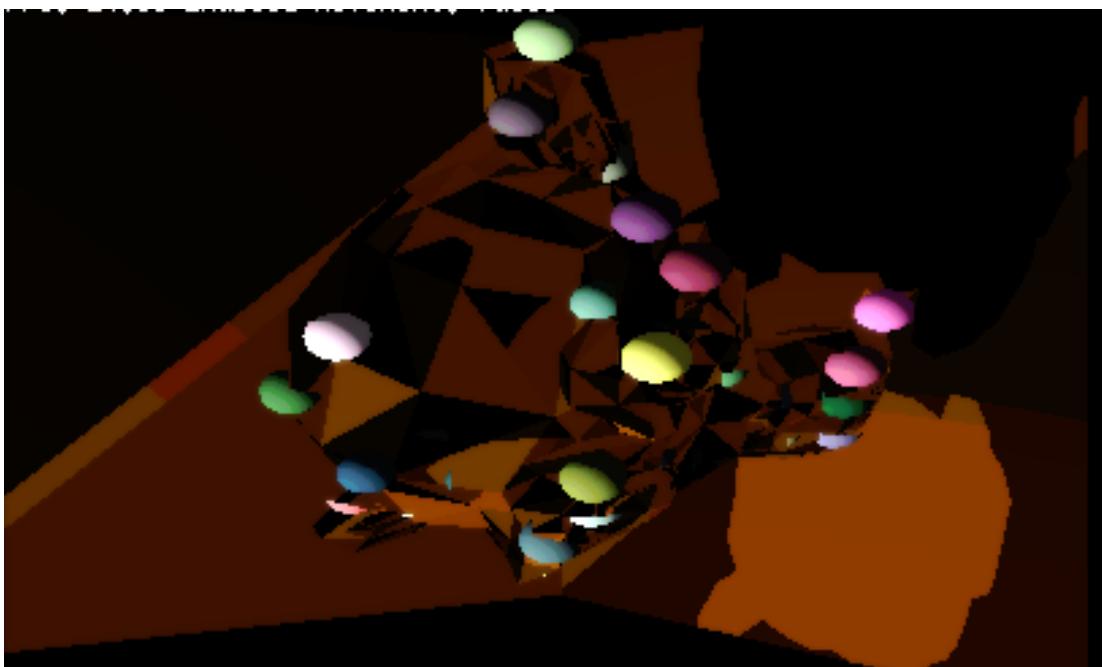


Spriemerovaný Obrázok

Parametre osvetlenia zahŕňajú intenzitu svetla, farbu svetla (R/G/B), zorné pole kamery a nastavenie gama, ktoré upravuje kontrast a jas medzi tmavými a svetlými tónmi. V sekcií nastavenia renderingu je možné zvoliť rôzne možnosti ako pripnúť svetlo ku kamere, vybrať verziu raymarching algoritmu, zapnúť performance

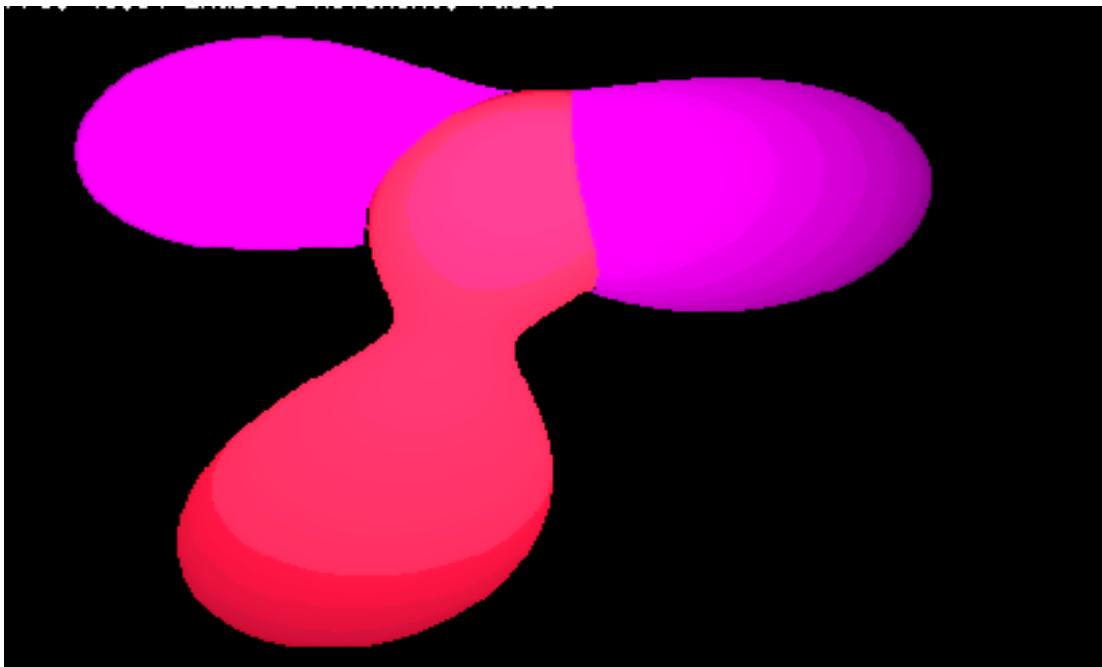
mód na optimalizáciu výkonu a nastaviť rozlíšenie, ktoré ovplyvňuje kvalitu obrázkov, pričom je možné zvoliť natívne rozlíšenie alebo 2X, 4X či 8X zväčšenie.

Pre verzie Raymarchingu sme implementovali V1, ktorá využíva BVH (Bounding Volume Hierarchy) pre efektívnejšie renderovanie, a V2, ktorá umožňuje meniť radius alebo SDF funkciu, čo zvyšuje flexibilitu pri práci s objektmi. Módy renderingu zahŕňajú klasické štandardné renderovanie, normálové renderovanie povrchov s rôznymi typmi textúr a vylepšení a vzdialenosné renderovanie, ktoré je momentálne nesprávne implementované, ale plánujeme ho vylepšiť pre presnejšie vykreslovanie na základe vzdialenosí medzi objektmi.

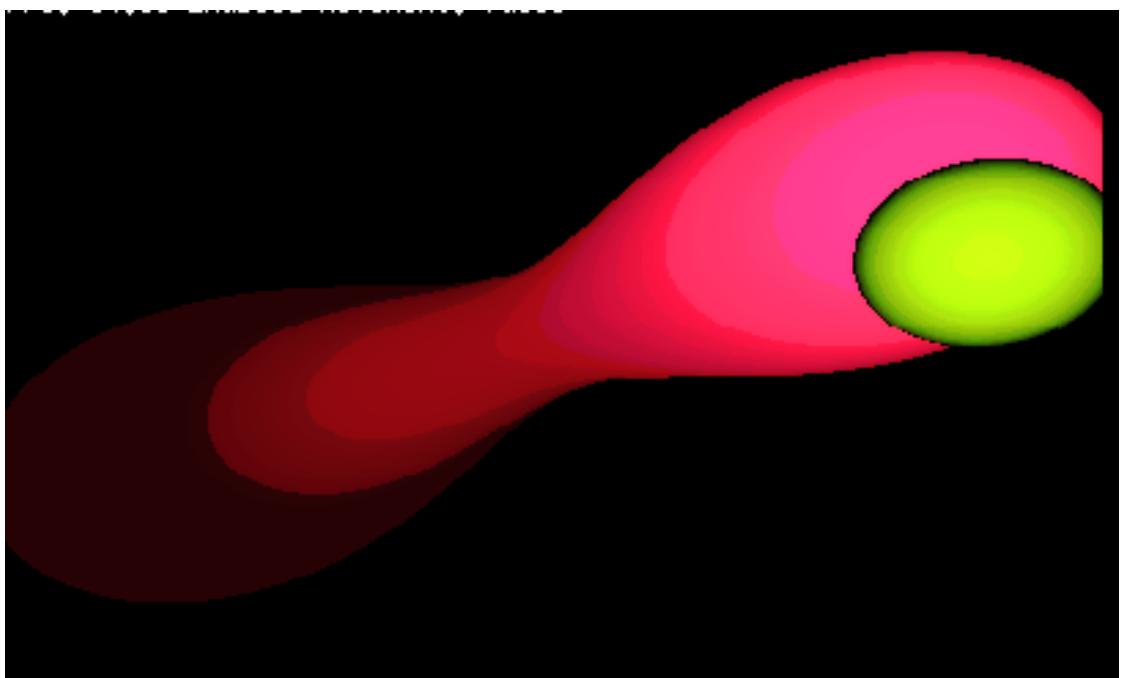


Ray Marching V1

Tento komponent sme implementovali s cieľom poskytnúť užívateľovi nástroje na detailné a flexibilné nastavenie všetkých dôležitých parametrov renderovania. Interaktívne ovládanie umožňuje priamu kontrolu nad výsledným obrazom, čo výrazne zlepšuje používateľský zážitok pri práci s Ray Tracing enginom.



Ray Marching V2



Ray Marching V2

2.3.5 Volume Picker

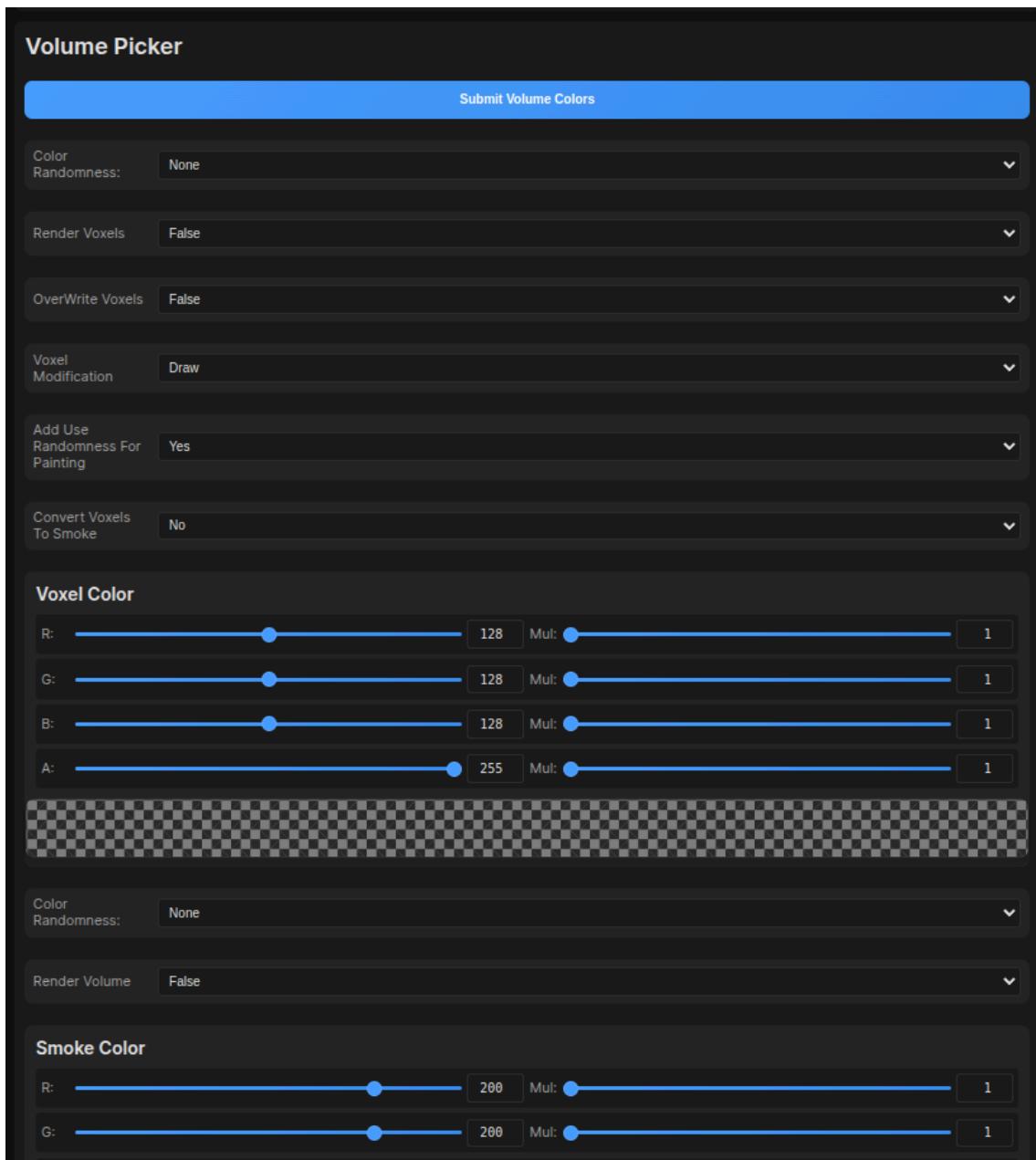


Image - 24 - 12

je navrhnutá na interaktívne spravovanie farby a vlastností objemov v prostredí renderovania. Tento komponent umožňuje užívateľovi jednoducho upravovať farby voxelov a objemových prvkov, čím sa zvyšuje flexibilita pri tvorbe a vizualizácii 3D objektov.

Funkcia Odoslať Farby Objemu zabezpečuje prenos vybraných farieb na objekty v scéne, zatiaľ čo možnosť Náhodnosť Farby pridáva variabilitu, ktorá zvyšuje realizmus a dynamiku zobrazovaných objemov. Prepínač Renderingu Voxellov

slúži na aktiváciu alebo deaktiváciu renderovania voxelov, čo ovplyvňuje spôsob, akým sa objemy zobrazujú na obrazovke. Taktiež je možné Prepísať Voxely, čo umožňuje zmeniť existujúce nastavenia voxelov pre novú konfiguráciu.

Pre ešte väčšiu kreativitu a flexibilitu je k dispozícii možnosť Pridať Náhodnosť do Maľovania, ktorá umožňuje generovať náhodné farby a vzory pri vytváraní objemov, čo môže byť užitočné pri simulácii prírodných alebo chaotických procesov. Možnosť Konvertovať Voxely na Dym umožňuje zobraziť objem ako dym alebo sklo, čím sa rozširujú možnosti renderovania objemov a zvyšuje sa ich vizuálna variabilita.

Súčasťou tohto komponentu je aj Výber Farby Voxelov s Náhľadom, ktorý umožňuje vizualizovať farbu objemu ešte pred jej aplikovaním, a Výber Farby Dymu, ktorý špecifikuje farbu dymových efektov pri renderovaní objemov. Parametre ako Hustota a Priehľadnosť ponúkajú možnosť detailného nastavenia objemových vlastností, pričom hustota ovplyvňuje tmavosť a hĺbku objemu, a priehľadnosť riadi, ako veľmi sú objemy priehľadné, čo má zásadný vplyv na konečný vzhľad renderovanej scény. Tento komponent bol implementovaný s cieľom umožniť užívateľovi flexibilne pracovať s objemovými efektmi a detailne kontrolovať vizualizáciu týchto objektov v 3D prostredí.

3.0 Princíp fungovania BVH

Pri ray-tracingu je klúčovou operáciou hľadanie priesecníkov medzi lúčom vyslaným z kamery a objektmi v scéne. Bez optimalizačnej štruktúry by bolo potrebné testovať každý lúč s každým objektom v scéne, čo by viedlo k časovej zložitosti $O(n)$ pre každý lúč, kde n je počet objektov v scéne. BVH rieši tento problém vytvorením hierarchickej štruktúry obaľujúcich objemov (najčastejšie osovo zarovnaných boxov - AABB), ktorá umožňuje rýchlo eliminovať veľké časti scény, ktoré lúč nemôže zasiahnuť.

Ked' lúč prechádza scénou, najprv sa testuje prienik s head Node BVH. Ak lúč nezasiahne obaľujúci objem uzla, môžeme okamžite preskočiť všetky objekty v tomto podstrome. Ak prienik existuje, algoritmus rekurzívne pokračuje do potomkov uzla, až kým nedosiahne listové uzly obsahujúce konkrétné objekty scény.

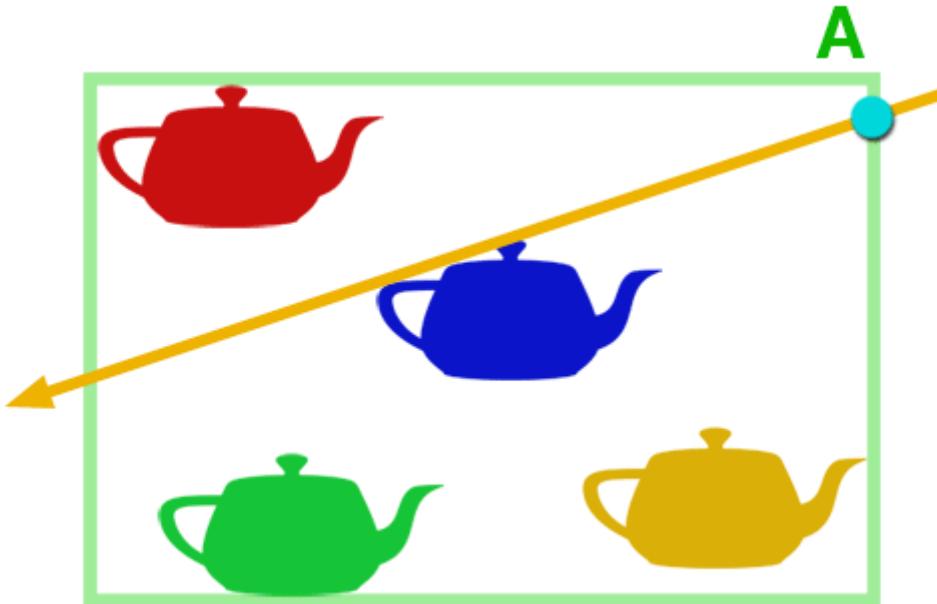


Image - 25 - 12

3.2 Surface Area Heuristic (SAH)

Pre optimálny výkon BVH je kľúčové, ako sa scéna rozdelí na podpriestory. Tu prichádza do hry Surface Area Heuristic (SAH). Táto heuristika optimalizuje rozdelenie objektov medzi children každej Node na základe plochy ich objemov. Cieľom je minimalizovať očakávaný čas potrebný na prechádzanie stromom a testovanie prienikov.

SAH pracuje na princípe, že pravdepodobnosť, že lúč zasiahne daný objem, je približne úmerná jeho povrchu. Pri delení uzla sa teda snažíme minimalizovať funkciu:

$$C = Ct + (SA(L)/SA(P)) * NL * Ci + (SA(R)/SA(P)) * NR * Ci$$

kde:

Ct je cena prechodu cez Node

Ci je cena testovania prieniku s objektom

SA(X) je plocha povrchu objemu

NL a **NR** sú počty objektov v ľavom a pravom potomkov

L , **R** , **P** označujú ľavého potomka, pravého potomka a parent Node

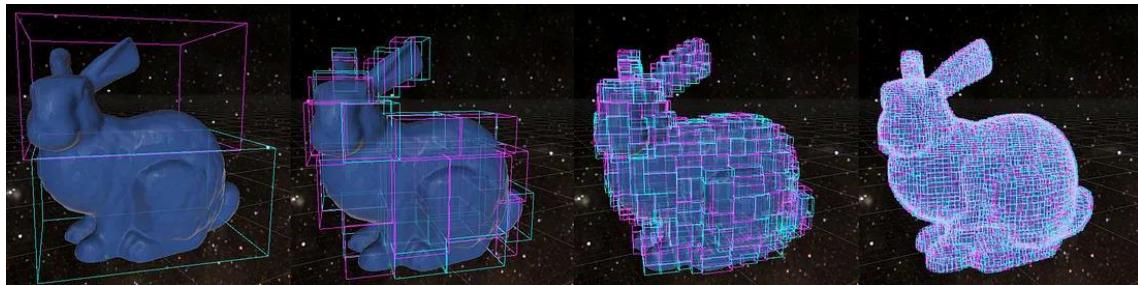


Image - 26 - 12

3.3 Reprezentácia Trojuholníkov a Materiálové Vlastnosti

Základným stavebným prvkom 3D scény v implementovanom ray-traceri je trojuholník, ktorý je reprezentovaný štruktúrou TriangleSimple. Táto štruktúra kombinuje geometrické vlastnosti trojuholníka s jeho materiálovými charakteristikami, čo umožňuje realistické zobrazenie rôznych povrchov a materiálov.

3.3.1 Geometrická Reprezentácia

```
type TriangleSimple struct {  
  
    v1, v2, v3           Vector      // Vrcholy trojuholníka  
  
    Normal              Vector      // Normálový vektor  
  
    // ... materiálové vlastnosti  
  
}
```

Geometria trojuholníka je definovaná troma 3D vektormi (v_1 , v_2 , v_3), ktoré predstavujú jeho vrcholy v priestore. Pre optimalizáciu výkonu je súčasťou štruktúry aj predpočítaný normálový vektor (Normal). Tento prístup významne urýchľuje proces renderovania, keďže normál Vector je klíčová pri výpočtoch osvetlenia a nie je potrebné ju opakovane počítať pri každom prieniku lúča s trojuholníkom.

3.3.2 Materiálové Vlastnosti

Materiálové vlastnosti trojuholníka sú reprezentované niekoľkými kľúčovými parametrami, ktoré určujú jeho vizuálne charakteristiky:

a) Farba (color ColorFloat32)

```
type ColorFloat32 struct {  
    R, G, B, A float32  
}
```

Farba povrchu je reprezentovaná pomocou vlastnej štruktúry ColorFloat32, ktorá využíva pre každý farebný kanál (červený, zelený, modrý) a alfa kanál hodnoty typu float32. Toto riešenie prináša niekoľko kľúčových výhod oproti tradičnej RGBA reprezentácii (uint8):

1. **Vysoký Dynamický Rozsah (HDR):**

- Na rozdiel od štandardnej RGBA reprezentácie, kde je každý kanál limitovaný rozsahom 0-255 (uint8), float32 umožňuje reprezentovať hodnoty výrazne presahujúce hodnotu 1.0. Toto je esenciálne pre realistické zobrazenie emisívnych materiálov, ktoré môžu vyžarovať svetlo s intenzitou mnohonásobne vyššou než 1.0.

2. **Emisívne Materiály:**

- ColorFloat32 umožňuje definovať materiály, ktoré aktívne emitujú svetlo do scény. Hodnoty vyššie ako 1.0 reprezentujú materiály, ktoré pridávajú energiu do scény.
- Toto je kľúčové pre implementáciu svetelných zdrojov priamo ako súčasti geometrie scény.

3. **Presnosť Výpočtov:**

- Float32 poskytuje vyššiu presnosť pri výpočtoch s farbami.
- Eliminuje sa problém kvantizácie, ktorý je typický pre uint8 reprezentáciu.
- Umožňuje jemnejšie prechody a gradienty v renderovanom obraze.

4. **Fyzikálna Korektnosť:**

- Reprezentácia pomocou float32 lepšie zodpovedá fyzikálnej realite, kde intenzita svetla nie je zhora obmedzená
- Umožňuje presnejšiu simuláciu svetelných interakcií v scéne

Táto implementácia je kľúčová pre dosiahnutie fotorealistického renderovania, keďže umožňuje pracovať s realistickými svetelnými podmienkami a materiálmi, ktoré by nebolo možné reprezentovať v štandardnom 8-bitovom farebnom priestore. Zároveň poskytuje základ pre implementáciu pokročilých renderovacích techník ako HDR rendering a tone mapping.

5. Direct-to-Scatter Ratio (directToScatter float32)

Tento parameter, definovaný v rozsahu [0, 1], určuje pomer medzi priamym odrazom svetla a difúznym rozptylom:

- Hodnota blízka 0: Väčšina svetla je rozptylená náhodným smerom (matný povrch)
- Hodnota blízka 1: Prevláda priamy odraz svetla (lesklý povrch) Tento parameter je kľúčový pre realistické zobrazenie

6. Reflection Coefficient (reflection float32)

Koeficient odrazu, definovaný v rozsahu [0, 1], určuje, ako silno povrch odráža

okolité prostredie:

- 0: Žiadne odrazy okolitého prostredia
- 1: Dokonalé zrkadlové odrazy Tento parameter ovplyvňuje pomer medzi vlastnou farbou objektu a farbou odrazenou z okolia, čo umožňuje simulať materiály od úplne matných až po zrkadlové povrhy.

7. Specular Intensity (specular float32)

Parameter v rozsahu [0, 1] určuje intenzitu spekulárneho odrazu:

- 0: Žiadny spekulárny odraz

- 1: Maximálny spekulárny odraz Tento

3.4 Nová Implementácia BVHLean

V novej implementácii BVHLean je štruktúra trojuholníka významne zjednodušená:

Pôvodná Štruktúra TriangleSimple

```
type TriangleSimple struct {
    // size=88 (0x58)
    v1, v2, v3 Vector
    // color color.RGBA
    color ColorFloat32
    Normal Vector
    reflection float32
    directToScatter float32
    specular float32
    Roughness float32
    Metallic float32
    id uint8
}
```

Image - 27 - \2

Nová Štruktúra TriangleBBOX

```
type TriangleBBOX struct {
    // size=52 (0x34)
    V1orBBoxMin, V2orBBoxMax, V3 Vector
    normal Vector
    id int32
}
```

Image - 28 - \2

Kľúčové zmeny:

- Veľkosť štruktúry sa zmenšila z 88 na 52 bajtov

- Veľkosť štruktúry sa zmenšila z 88 na 52 bajtov
- Vlastnosti trojuholníka sú teraz definované samostatne
- Zjednotenie bounding boxu a trojuholníka

Nová Štruktúra Textúry

```
type Texture struct {
    texture [128][128]ColorFloat32
    normals [128][128]Vector

    // Materiálové vlastnosti
    reflection      float32
    directToScatter float32
    specular        float32
    Roughness       float32
    Metallic        float32
}
```

Image - 29 - \2

Táto nová implementácia umožnila zrýchlenie BVH o:

- 18 % na procesore Ryzen 9 5950X
- Systém s 72 GB RAM

Táto optimalizácia zjednodušuje štruktúru dát a umožňuje efektívnejšiu prácu s pamäťou počas ray-tracingu.

3.5 BVH a jej implementácia

V procese optimalizácie ray-tracingu bola implementácia efektívnej akceleračnej štruktúry klúčovým faktorom pre zlepšenie výkonu. Evolúcia riešenia prešla niekoľkými fázami:

Vývojová cesta prístup - Pôvodná implementácia testovala prienik lúča s každým trojuholníkom v scéne, čo viedlo k lineárnej časovej zložitosti $O(n)$ a výrazne limitovalo výkon pri rastúcom počte trojuholníkov.

Bounding Box optimalizácia - Ako prvý krok optimalizácie boli implementované ohraničujúce boxy (Bounding Boxes) pre skupiny trojuholníkov, čo umožnilo rýchlejšie vylúčenie objektov mimo lúča. Toto zlepšenie však stále nebolo dostatočné pre komplexné scény.

BVH implementácia - Finálnym riešením bola implementácia Bounding Volume Hierarchy (BVH), ktorá hierarchicky organizuje priestor a umožňuje efektívne prechádzanie len relevantných častí scény, čím znižuje časovú zložitosť na približne $O(\log n)$

3.5.1 Evolúcia BVH štruktúry

V tejto sekcii sa venujeme tomu, ako sme postupovali počas vývoja BVH, ktorý slúži ako hlavná akceleračná štruktúra.

Pôvodná BVH implementácia

```
type BVHNode struct { // veľkosť=136  
    (0x88) bajtov Left, Right *BVHNode  
  
    BoundingBox [2]Vector  
  
    Triangles  
    TriangleSimple  
  
    active  
    bool  
}
```

Prvá verzia BVH používala štandardnú stromovú štruktúru s ukazovateľmi na ľavý a pravý podstrom. Táto implementácia však trpela rastúcou veľkosťou uzlov kvôli pridávaniu materiálových vlastností a normálových vektorov pre trojuholníky.

3.5.2 Optimalizovaná BVHLean

Táto verzia prináša signifikantné vylepšenia, najmä v oblasti redukcie hodnôt zo štruktúr, ktoré sú zbytočné.

```
type BVHLeanNode struct { // veľkosť=72
(0x48) bajtov Left, Right *BVHLeanNode
```

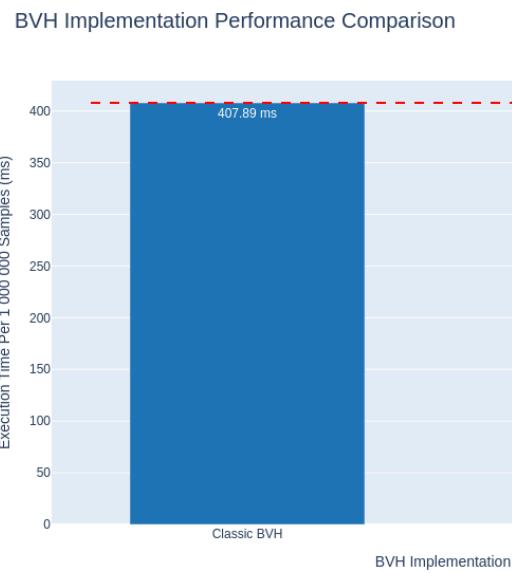
TriangleBBOX TriangleBBOX

active	bool
}	
type TriangleBBOX struct { // veľkosť=52	
(0x34) bajtov	
V1orBBoxMin, V2orBBoxMax, V3 Vector	
normal	Vector
id	int32
}	

Table 2.

Výsledky testovania preukazujú signifikantné zlepšenie výkonu novej optimalizovanej verzie BVH.

- Classic BVHNode : 407.888788ms
- BVHLean : 341.148485ms



Pre verziu V4 bola vytvorená optimalizovaná implementácia BVHLean, ktorá:

- Zmenšila veľkosť Nodu takmer na polovicu (zo 136 bajtov na 72 bajtov)
- Zlúčila BBox-ov a trojuholník do jednej štruktúry pre lepsiu lokalitu dát
- Odstránila priameho ukladania materiálových vlastností v BVHNode a nahradila ich systémom ID odkazov na textúry

3.5.3 Optimalizácia BVHLean - porovnanie 2 bounding boxov naraz

Kedže vždy musíme pozerať intersekciu s oboma dvoma bounding boxami, je omnoho efektívnejšie porovnať intersekciu v jednej funkcií, takže sa dá vyhnúť počiatočnej inverse direction

```
func BoundingBoxCollisionPair(box1Min, box1Max, box2Min, box2Max Vector, ray Ray) (bool, bool, float32, float32) {
    // Precompute the inverse direction (once for both boxes)
    invDirX := 1.0 / ray.direction.x
    invDirY := 1.0 / ray.direction.y
    invDirZ := 1.0 / ray.direction.z
    // Box 1 intersection
    tx1_1 := (box1Min.x - ray.origin.x) * invDirX
    tx2_1 := (box1Max.x - ray.origin.x) * invDirX
    tmin_1 := min(tx1_1, tx2_1)
    tmax_1 := max(tx1_1, tx2_1)
    ty1_1 := (box1Min.y - ray.origin.y) * invDirY
    ty2_1 := (box1Max.y - ray.origin.y) * invDirY
    tmin_1 = max(tmin_1, min(ty1_1, ty2_1))
    tmax_1 = min(tmax_1, max(ty1_1, ty2_1))
    tz1_1 := (box1Min.z - ray.origin.z) * invDirZ
    tz2_1 := (box1Max.z - ray.origin.z) * invDirZ
    tmin_1 = max(tmin_1, min(tz1_1, tz2_1))
    tmax_1 = min(tmax_1, max(tz1_1, tz2_1))
    // Box 2 intersection
    tx1_2 := (box2Min.x - ray.origin.x) * invDirX
    tx2_2 := (box2Max.x - ray.origin.x) * invDirX
    tmin_2 := min(tx1_2, tx2_2)
    tmax_2 := max(tx1_2, tx2_2)
    ty1_2 := (box2Min.y - ray.origin.y) * invDirY
    ty2_2 := (box2Max.y - ray.origin.y) * invDirY
    tmin_2 = max(tmin_2, min(ty1_2, ty2_2))
    tmax_2 = min(tmax_2, max(ty1_2, ty2_2))
    tz1_2 := (box2Min.z - ray.origin.z) * invDirZ
    tz2_2 := (box2Max.z - ray.origin.z) * invDirZ
    tmin_2 = max(tmin_2, min(tz1_2, tz2_2))
    tmax_2 = min(tmax_2, max(tz1_2, tz2_2))
    // Check intersections
    hit1 := tmax_1 >= max(0.0, tmin_1)
    hit2 := tmax_2 >= max(0.0, tmin_2)
    // Return hit status and distances
    return hit1, hit2, tmin_1, tmin_2
}
```

- Toto vylepšenie je výkonnejšie zhruba o 25.86%
 - BoundingBoxCollisionVector: 291.248548ms
 - BoundingBoxCollisionPair: 215.934921ms

Bounding Box Collision Functions Performance Comparison

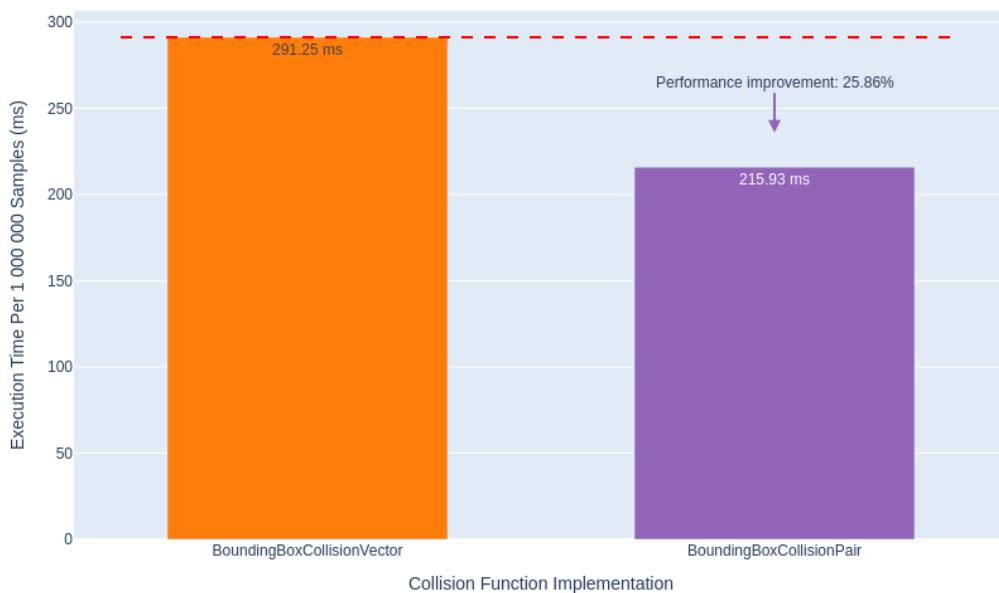


Image - 32 - \2

3.5.4 Experimentálna array-based implementácia

V rámci ďalšej optimalizácie bola experimentálne vytvorená array-based reprezentácia BVH, kde:

- Uzly sú uložené v súvislom poli miesto rozptýlených alokácií
- Vzťahy medzi uzlami sú implicitné (ľavý potomok má index $2n$, pravý $2n+1$)
- Zlepšuje sa lokalita referencií a efektivita cache pamäte procesora

```
type BVHArray struct { // veľkosť=65538508 (0x3e809cc) bajtov
    triangles [NumNodes]TriangleBBOX
    textures     [128]Texture
}
```

Výkonnostné výsledky

Testovanie preukázalo 21% zlepšenie výkonu oproti klasickej implementácii (218,831 ns/op vs. 278,146 ns/op) vďaka lepšiemu cache využitiu pri sekvenčnom prístupe k dátam.

Testovanie dostupné na: <https://github.com/DarkBenky/testBinaryTree>

Array-based implementácia zostala v experimentálnej fáze z dôvodu časových obmedzení projektu, ale predstavuje sľubný smer pre ďalší vývoj.

4.0 Podpora Načítavania 3D Geometrie

V tejto sekcii sa zameriame na funkcionality, ktorá umožňuje načítavanie .obj súborov a následné vytvorenie vnútorej reprezentácie objektu, bud' vo forme poľa trojuholníkov, alebo pomocou BVH reprezentácie objektu.

4.1 Načítavanie .OBJ Súborov

Implementovaný ray-tracer poskytuje robustnú podporu pre načítavanie 3D geometrie prostredníctvom štandardného .obj formátu, čo výrazne zvyšuje flexibilitu a použiteľnosť aplikácie.

Kľúčové vlastnosti implementácie

- Načítavanie priestorových vrcholov (vertices)
- Konverzia polygónov na trojuholníkovú siet'

Podpora Materiálov

- Parsing .mtl súborov
- Načítavanie základných materiálových vlastností:

Optimalizačné Techniky

- Predpočítavanie normálových vektorov
- Efektívna konverzia na interný formát TriangleSimple
- Podpora pre zložitejšie geometrické útvary

Proces Načítavania .OBJ Súborov

Proces načítavania .obj súborov zahŕňa niekoľko kľúčových krokov:

- 1. Parsovanie priestorových súradníc vertices**
- 2. Identifikácia a konverzia polygónov na trojuholníky**
- 3. Priradenie materiálových vlastností jednotlivým geometrickým prvkom**

5.0 RayTracing Vývoj Funkcionality

Počas vývoja raytracing enginu sme prešli viacerými implementáciami jednotlivých funkcií, pričom každá z nich priniesla určité výhody, ale aj nevýhody. V tejto časti sa podrobne zameriame na jednotlivé verzie a ich vlastnosti.

TraceRay - GUI Názov : V1

TraceRay, verzia V1, je pôvodná funkcia, ktorá poskytuje základnú funkcionalitu pre ray tracing. Táto verzia umožňuje prácu s komplexnejšími geometrickými útvarmi a využíva BVH (Bounding Volume Hierarchy) štruktúru na testovanie priesečníkov medzi lúčmi a objektmi v scéne. Funkcia vykonáva základný výpočet rozptyleného svetla pomocou metódy cosine-weighted hemisphere sampling, čo prispieva k realistickejšiemu osvetleniu objektov.

Okrem toho V1 počíta priame odrazy a zrkadlové body pomocou jednoduchého svetelného modelu, čo je dôležité pre simuláciu realistických svetelných efektov. Táto verzia tiež využíva rekurzívny prístup pre hĺbkové odrazy, čím sa zabezpečuje presnejšie vykreslovanie odrazov svetla na objektoch. Na záver, všetky svetelné efekty – priame svetlo, rozptylené svetlo a odrazy – sú kombinované lineárne, čím sa dosahuje základná úroveň realistického osvetlenia scény.

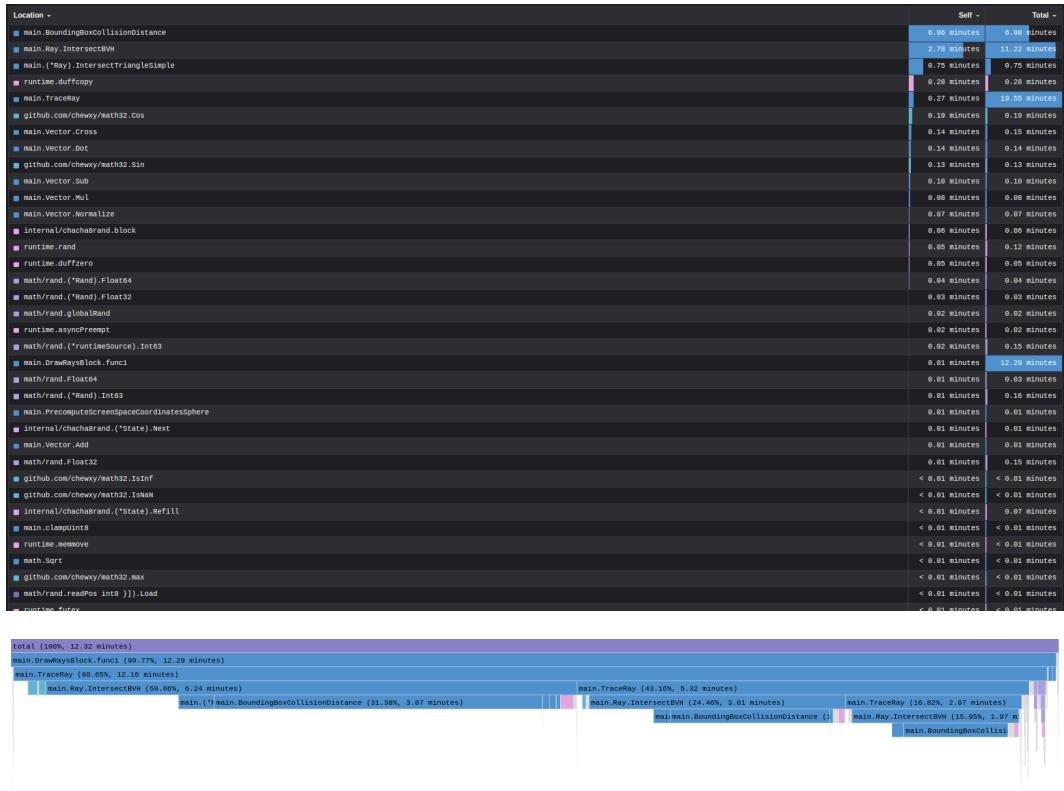


Image - 34 - 33 - 12

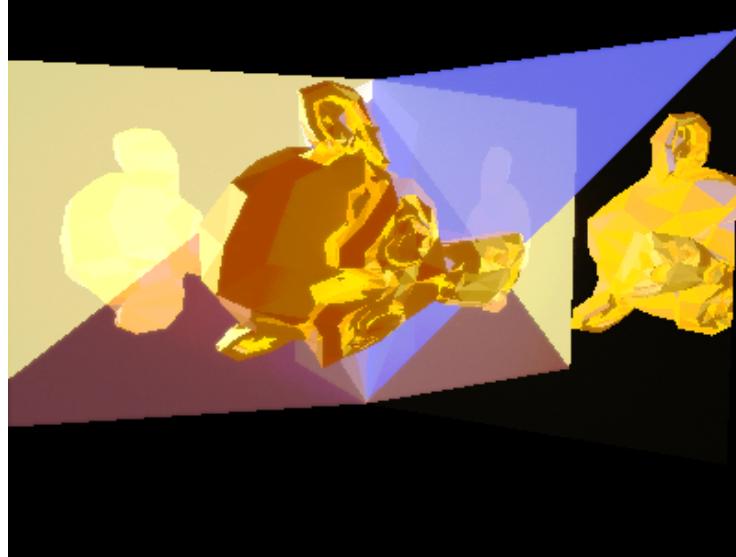


Image - 35 - 12

- V1 Profile

TraceRayV2 - GUI Názov : V2

logicky organizuje kód a oddeluje priame osvetlenie, nepriame osvetlenie a odrazy. Pridáva perturbáciu smerov odrazov na základe drsnosti povrchu a využíva vylepšenú logiku hemisphere sampling na lepší rozptyl svetla. Taktiež lepšie spracováva prechod medzi difúznym a priamym odrazením svetla, čím zvyšuje realistikosť renderovania.



Image - 36 - \2



Image - 37 - \2

- [V2 Profile](#)

TraceRayV3 - GUI Názov : V2M

TraceRayV3, verzia V2M, využíva prístup založený na PBR (Physically Based Rendering), ktorý implementuje Fresnel-Schlick aproximáciu na výpočet odrazov. Používa GGX distribúciu pre microfacet-based zrkadlové objekty, čo zlepšuje simuláciu materiálových vlastností, ako sú kovový lesk a drsnosť. Tento prístup tiež používa presnejšiu energetickú konzerváciu pri kombinovaní svetelných komponentov a vracia jednu farebnú hodnotu, čím poskytuje realistickejšie renderovanie.

V2M

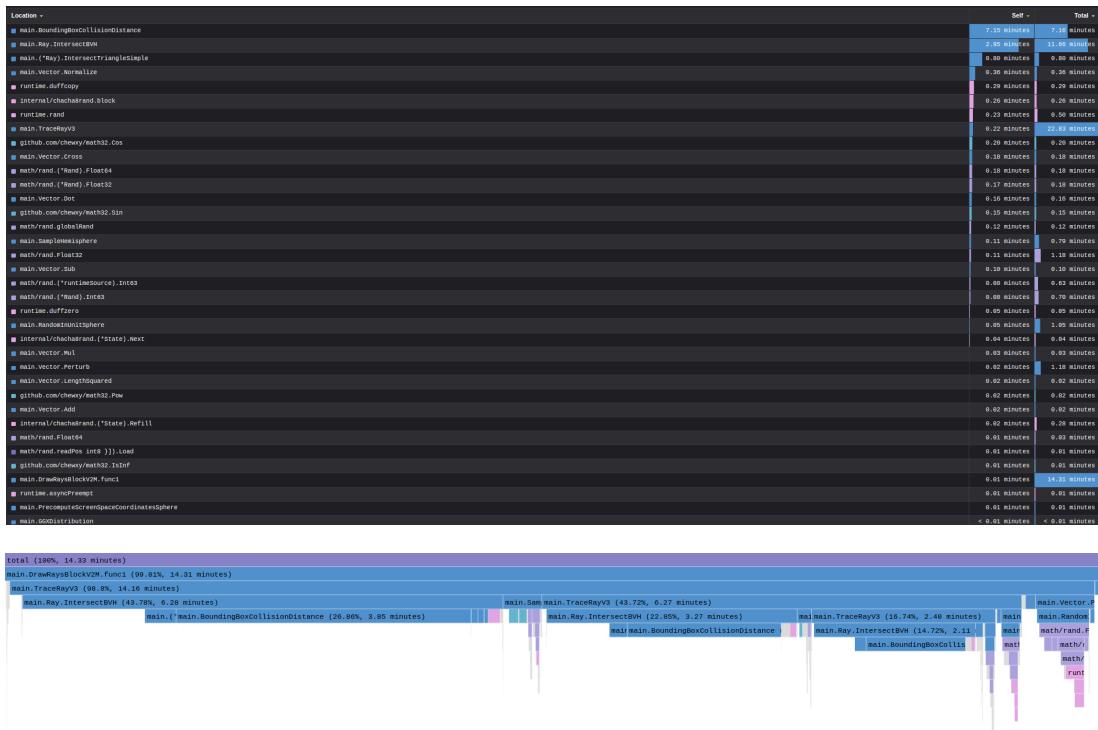


Image - 39 - 38 - 12

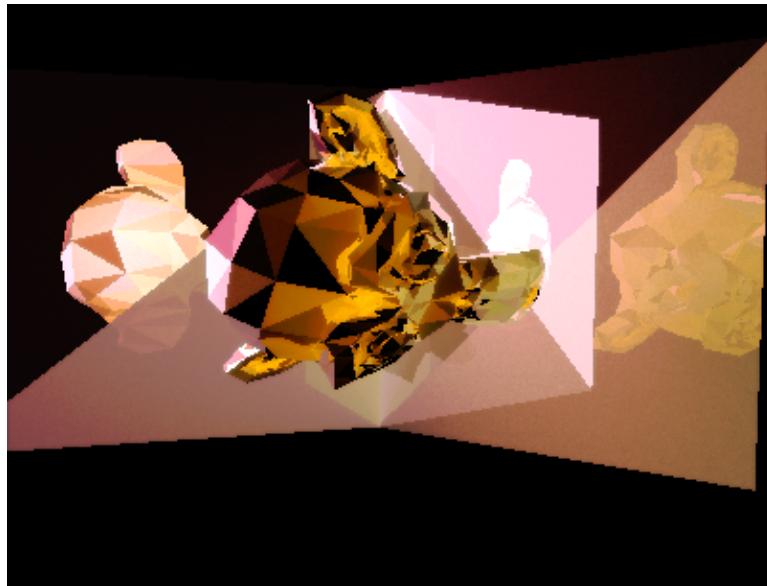


Image - 40 - \2

- [V2M Profile](#)

TraceRayV3Advance - GUI Názov : V2Liner / V2Log

TraceRayV3Advance, verzia V2Liner / V2Log, je rozšírením TraceRayV3, ktoré vracia dodatočné dátá, ako sú farba, vzdialenosť a normálový vektor. Tieto informácie umožňujú pokročilejšie post-processing techniky. Inak používa rovnaký PBR prístup ako TraceRayV3, čím si zachováva realistické simulácie materiálových vlastností a svetelných efektov.

V2Lin



Image - 41 - \2



Image - 42 - \2

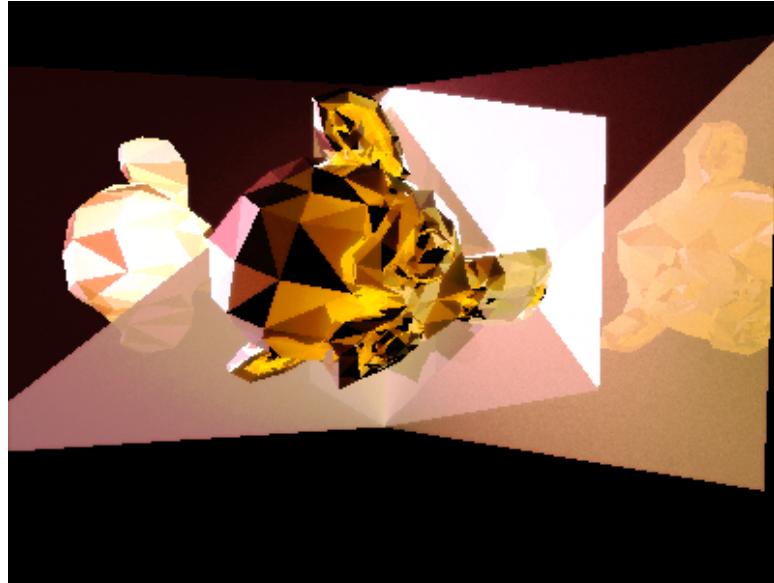


Image - 43 - \2

- V2Lin Profile

V2Log



Image - 45 - 44 - 12



- V2Log Profile

TraceRayV3AdvanceTexture - GUI Názov : V2LinearTexture / V2LogTexture

TraceRayV3AdvanceTexture, verzia V2LinearTexture /

V2LogTexture, pridáva podporu pre textúry a integruje materiálové vlastnosti priamo z textúr. Používa parameter textureMap na prístup k dátam textúr a vracia informácie o farbe a normále pre post-processing. Táto verzia využíva špecializovanú BVH traversal funkciu na podporu textúr a aplikuje dátá textúr na materiálové parametre, ako sú drsnosť a kovový lesk, čím zvyšuje realizmus renderovaných objektov.

V2LinTexture

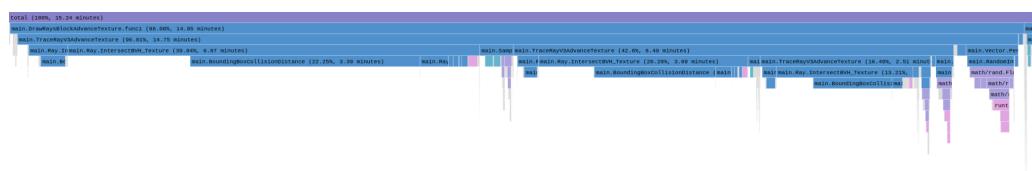
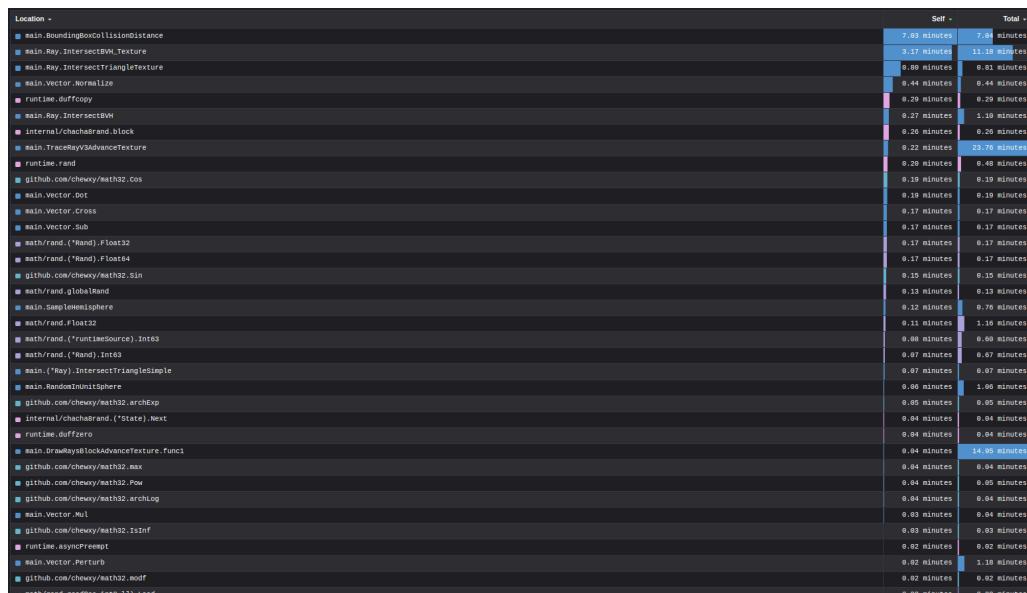


Image - 48 - \2

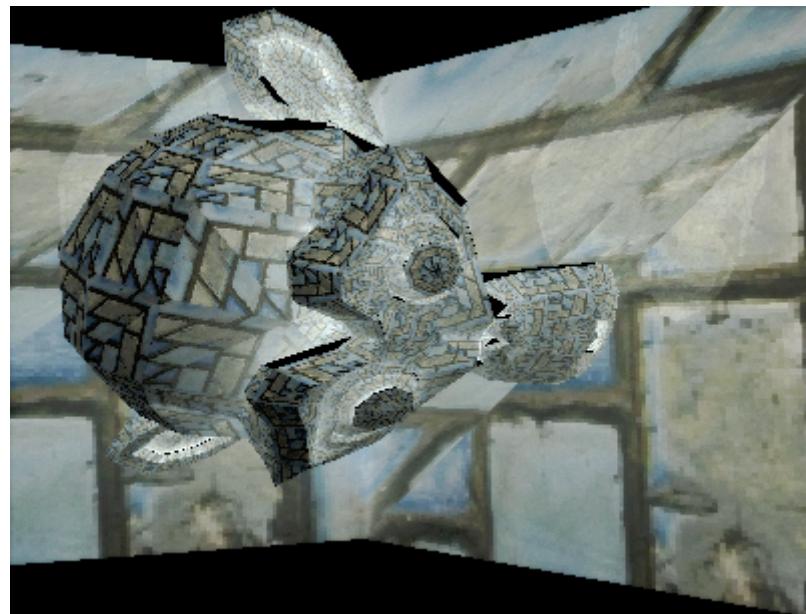


Image - 49 - \2

- V2LinTexture Profile

V2LogTexture

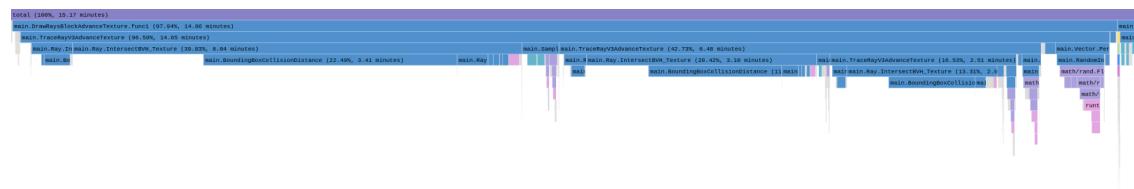


Image - 50 - \2

Location	Self	Total
main.BoundingBoxCollisionDistance	7.01 minutes	7.01 minutes
main.Ray_IntersectBVH_Texture	3.16 minutes	11.18 minutes
main.Ray_IntersectTriangleTexture	0.70 minutes	0.70 minutes
main.Vector_Normalize	0.47 minutes	0.47 minutes
runtime.dufcopy	0.28 minutes	0.28 minutes
main.Ray_IntersectBVH	0.26 minutes	1.05 minutes
internal/chacharand/block	0.24 minutes	0.24 minutes
main.TraceRayV3dAdvanceTexture	0.22 minutes	23.44 minutes
runtime.rnd	0.21 minutes	0.40 minutes
github.com/chevxy/math32.cos	0.20 minutes	0.20 minutes
main.Vector_Dot	0.19 minutes	0.19 minutes
main.Vector_Cross	0.18 minutes	0.18 minutes
math/rnd.(*)rand.Float64	0.17 minutes	0.17 minutes
main.Vector_Sub	0.16 minutes	0.16 minutes
math/rnd.(*)rand.Float32	0.16 minutes	0.16 minutes
github.com/chevxy/math32.Sin	0.15 minutes	0.15 minutes
math/rnd.globalrand	0.11 minutes	0.11 minutes
main.sampleInUnitSphere	0.11 minutes	0.77 minutes
math/rand.Float32	0.11 minutes	1.00 minutes
math/rand.(*Rand).Int63	0.08 minutes	0.08 minutes
main.(*Ray).IntersectTriangleSample	0.07 minutes	0.07 minutes
math/rand.(*Rand).Int63	0.07 minutes	0.07 minutes
main.RandomInUnitSphere	0.06 minutes	0.06 minutes
runtime.dufzero	0.05 minutes	0.05 minutes
github.com/chevxy/math32.Pow	0.05 minutes	0.05 minutes
github.com/chevxy/math32_archExp	0.05 minutes	0.05 minutes
github.com/chevxy/math32_max	0.05 minutes	0.05 minutes
github.com/chevxy/math32_archLog	0.05 minutes	0.05 minutes
internal/chacharand.(*State).Next	0.04 minutes	0.04 minutes
main.DrawByLockedAdvanceTexture.Func	0.04 minutes	0.04 minutes
main.Vector_Mul	0.03 minutes	0.03 minutes
github.com/chevxy/math32_isInf	0.03 minutes	0.03 minutes
main.Vector_Perturb	0.03 minutes	1.13 minutes
runtime.asyncPreempt	0.02 minutes	0.02 minutes
github.com/chevxy/math32_modf	0.02 minutes	0.02 minutes

Image - 51 - \2

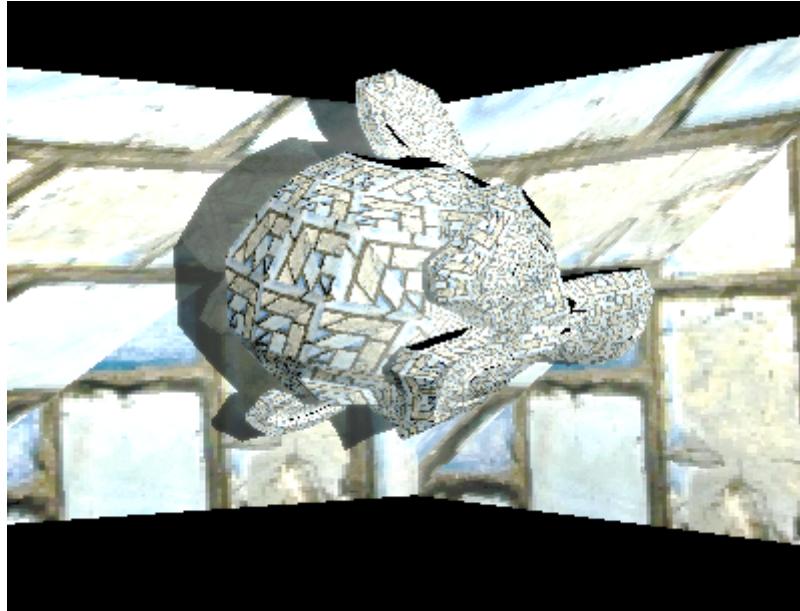


Image - 52 - \2

- V2LogTexture Profile

TraceRayV4AdvanceTexture - GUI Názov : V4Linear / V4Log

TraceRayV4AdvanceTexture, verzia V4Linear / V4Log, predstavuje optimalizovanú verziu s podporou textúr. Namiesto štandardnej BVH štruktúry využíva odľahčenú verziu BVHLeanNode, čo znižuje pamäťové nároky a zvyšuje výkon. Obsahuje aj optimalizovanú funkciu na výpočet priesčníkov (IntersectBVHLean_Texture). Funkcionalitou je podobná verzii TraceRayV3AdvanceTexture, pričom stále vracia informácie o farbe a normále pre ďalšie spracovanie.

V4Lin



Image - 53 - \2

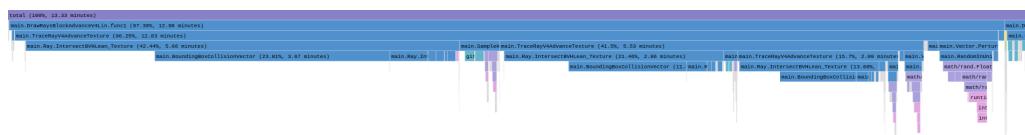


Image - 54 - \2

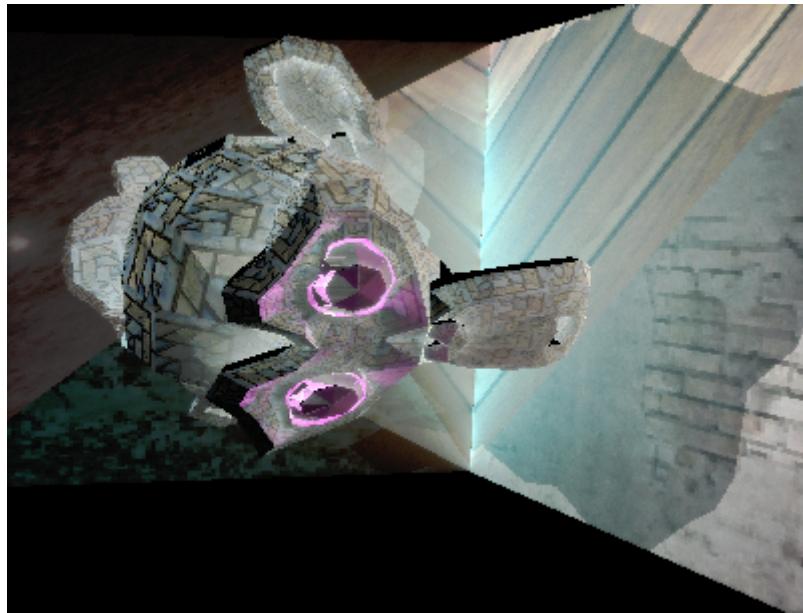


Image - 55 - \2

- **V4Lin Profile**

V4Log



Image - 56 - \2



Image - 57 - \2

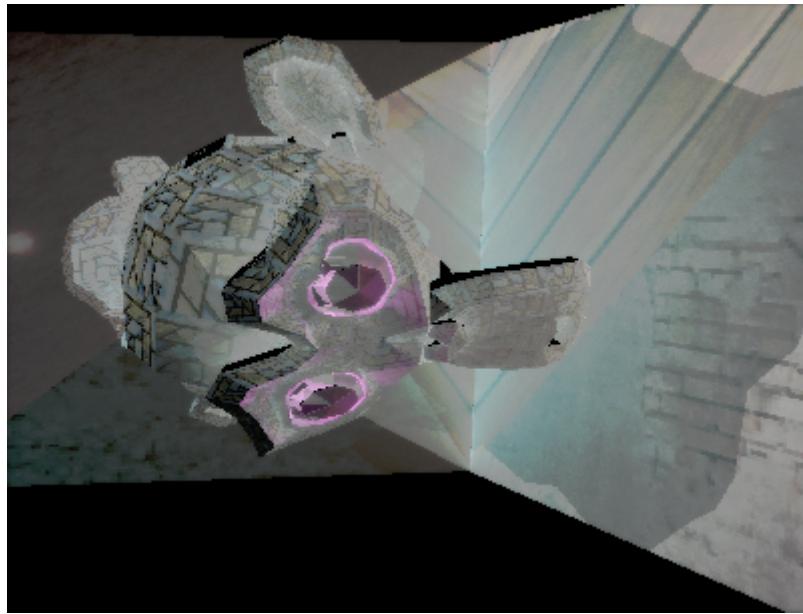


Image - 58 - \2

- [V4Log Profile](#)

TraceRayV4AdvanceTextureLean - GUI Názov : V4LinOptim / V4LogOptim / V4Optim-V2

TraceRayV4AdvanceTextureLean, verzie V4LinOptim / V4LogOptim / V4Optim-V2, je ešte optimalizovanejšia verzia, ktorá sa zameriava na zníženie pamäťovej náročnosti a zjednodušenie návratovej štruktúry. Vracia iba výslednú farebnú informáciu, bez normálových vektorov a vzdialenosť. Zachováva všetky PBR výpočty, takže kvalita osvetlenia a materiálových efektov ostáva nezmenená. Verzia V4Optim-V2 prináša dodatočné optimalizácie, vrátane zrýchlenej intersekcii s bounding boxami o 25,86 % a odstránenia gamma post-processingu, čím ďalej zvyšuje výkon.

V4LinOptim

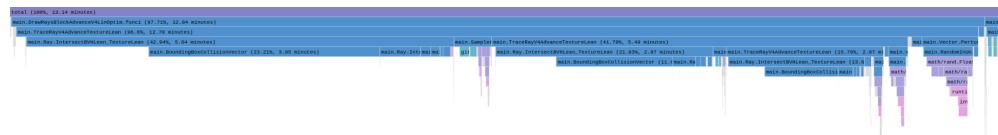
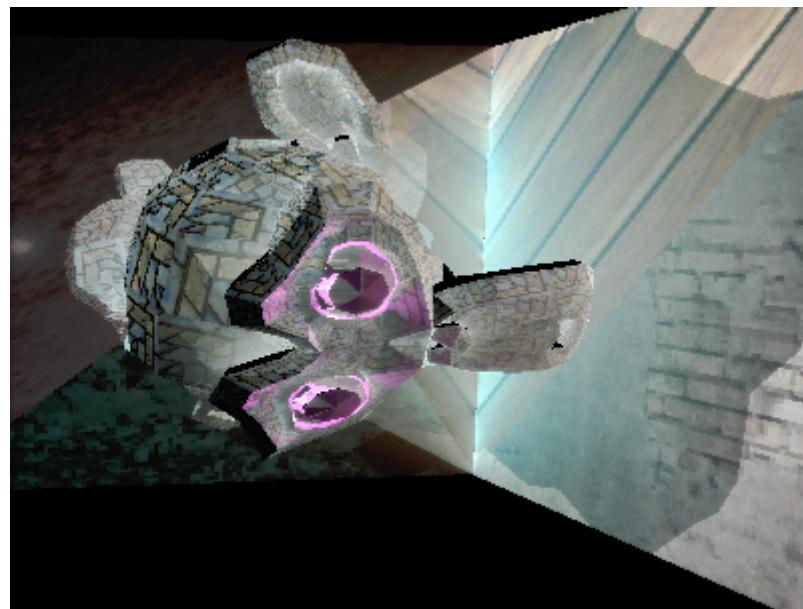


Image - 59 - \2

Location -	Self -	Total -
main.BoundingboxCollisionVector	5.95 minutes	5.95 minutes
main.Ray_IntersectBVLean_TextureLean	2.99 minutes	16.31 minutes
main.Ray_IntersectTriangleTextureGeneralLean	1.63 minutes	1.63 minutes
main.Vector_Normalize	0.45 minutes	0.45 minutes
main.Vector_Cross	0.27 minutes	0.27 minutes
main.Vector_Dot	0.25 minutes	0.25 minutes
internal/chachastrand.block	0.24 minutes	0.24 minutes
runtime.rand	0.21 minutes	0.47 minutes
main.Vector_Sub	0.20 minutes	0.20 minutes
github.com/chezy/math32.Cos	0.20 minutes	0.20 minutes
main.TraceRayAdvanceTextureLean	0.18 minutes	20.79 minutes
math/rnd.(~Rand).Float64	0.17 minutes	0.17 minutes
math/rnd.(~Rand).Float32	0.16 minutes	0.16 minutes
github.com/chezy/math32.Sin	0.14 minutes	0.14 minutes
math/rnd.(~Rand).Sin	0.12 minutes	0.12 minutes
main.SampleSphere	0.12 minutes	0.77 minutes
math/rnd.(~Rand).Float32	0.10 minutes	1.10 minutes
math/rnd.(~RuntimeSource).Int64	0.08 minutes	0.58 minutes
math/rnd.(~Rand).Int64	0.07 minutes	0.65 minutes
main.RandomUnitSphere	0.06 minutes	1.00 minutes
github.com/chezy/math32.ArchExp	0.05 minutes	0.05 minutes
runtime.Duffero	0.05 minutes	0.05 minutes
github.com/chezy/math32.ArchLog	0.05 minutes	0.05 minutes
github.com/chezy/math32.Pow	0.04 minutes	0.06 minutes
github.com/chezy/math32.Max	0.04 minutes	0.04 minutes
internal/chachastrand.(~State).Next	0.04 minutes	0.04 minutes
main.Vector_Mul	0.03 minutes	0.03 minutes
github.com/chezy/math32.IsInf	0.03 minutes	0.03 minutes
main.Vector_Perturb	0.03 minutes	0.03 minutes
main.BrakeSyncLocAdvanceValInOptim.Func1	0.03 minutes	1.13 minutes
github.com/chezy/math32.Mod	0.02 minutes	0.02 minutes
math/rnd.Float64	0.02 minutes	0.02 minutes
main.Vector_Lengthsquared	0.02 minutes	0.04 minutes
runtime.AsyncPreempt	0.02 minutes	0.02 minutes
internal/chachastrand.(~State).Refill	0.02 minutes	0.26 minutes

Image - 60 - \2



- V4LinOptim Profil

V4LogOptim

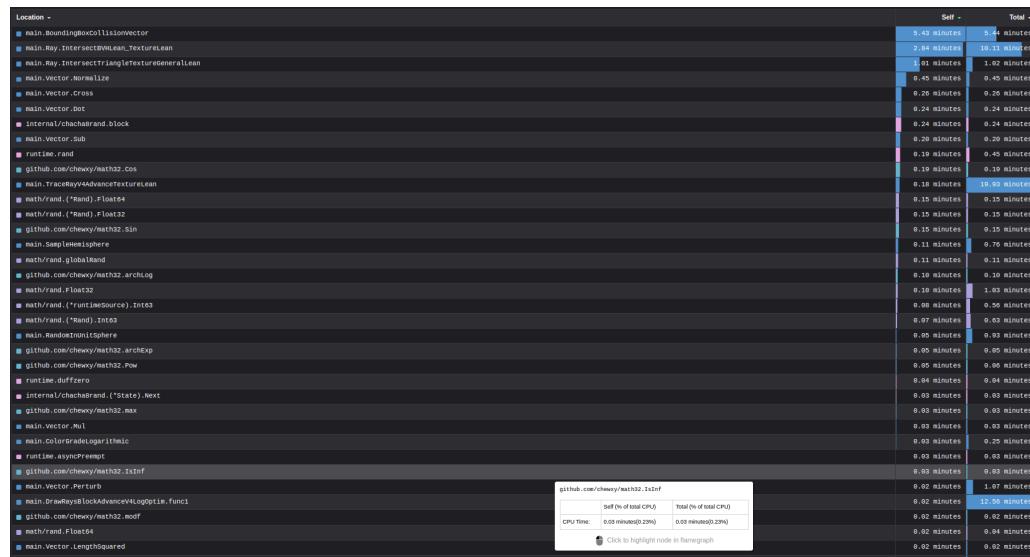


Image - 62 - 12

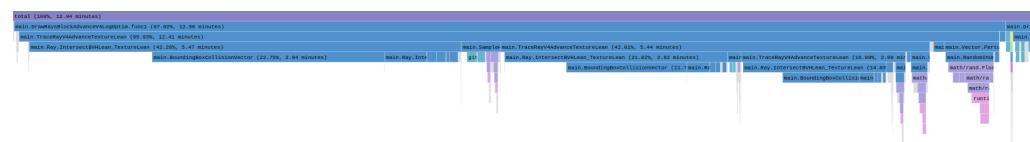
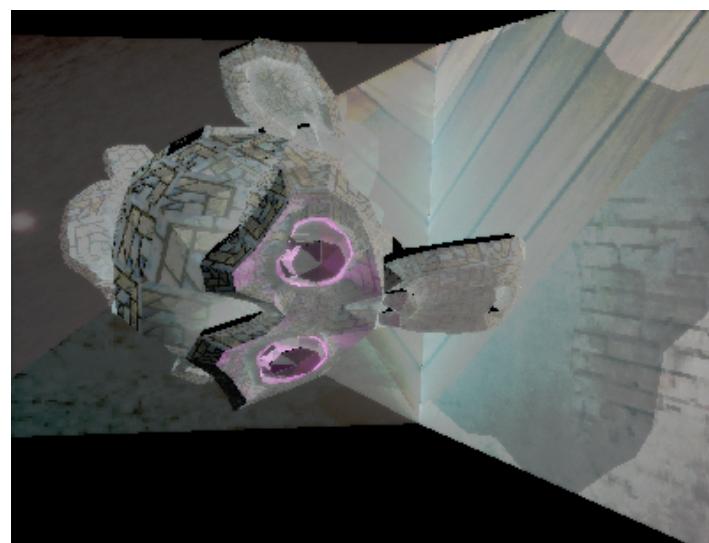


Image - 63 - 12



- V4LogOptim Profil

V4Optim-V2



Image - 65 - 12

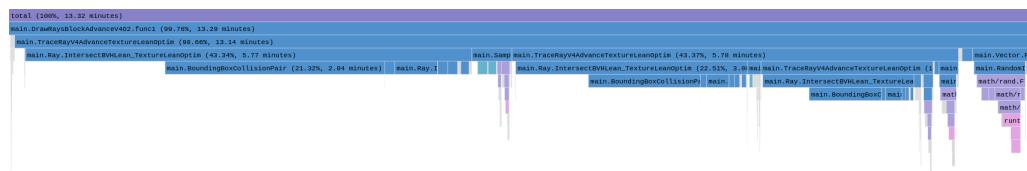
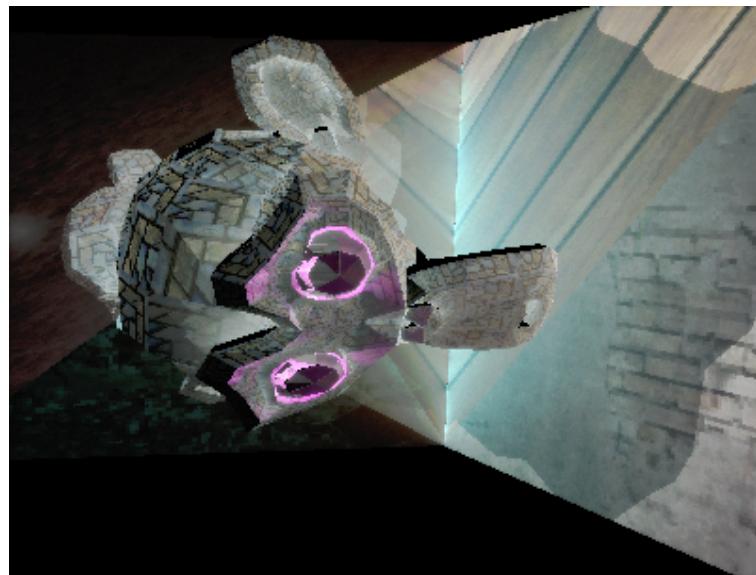
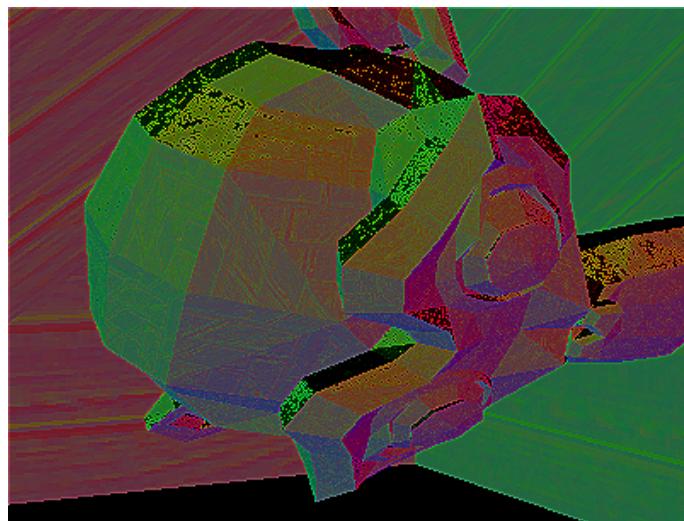


Image - 66 - 12



- [V4Optim-V2 Profil](#)

Normal Image



4.1 Klúčové body evolúcie:

Evolúcia ray tracingového systému prešla viacerými fázami, pričom každá priniesla zásadné vylepšenia v rôznych oblastiach. Počiatočný renderovací model (TraceRay) bol postupne rozšírený až po plnohodnotný PBR model vo verzích V3 a novších, čím sa dosiahla realistickejšia simulácia materiálov. Návratové dátá prešli vývojom od jednoduchého výstupu farby, cez pridanie informácií o vzdialosti a normálovom vektore, až po optimalizované verzie, kde sa vrátila len farba pre zvýšenie výkonu. Použitie BVH štruktúr sa neustále zdokonalovalo, od štandardného BVH až po odľahčené lean BVH varianty, ktoré znížujú pamäťovú náročnosť a urýchľujú výpočty.

Simulácia materiálových vlastností sa vyvíjala od jednoduchého odrazu svetla až po plný PBR model, ktorý zahŕňa kovový lesk, drsnosť a Fresnelovu reflektivitu, čím sa dosiahlo vyšší realizmus. Podpora textúr bola pridaná vo verzii V3AdvanceTexture a zachovaná aj v ďalších verzích V4, čím sa umožnilo presnejšie mapovanie materiálov. Postupné optimalizácie, najmä vo verzii V4Lean, výrazne znížili pamäťové nároky, pričom každá nová verzia prinášala kompromis medzi rozšírenými funkciami a výkonnostnou efektivitou.

Ilustračné obrázky



Image - 69 - \2

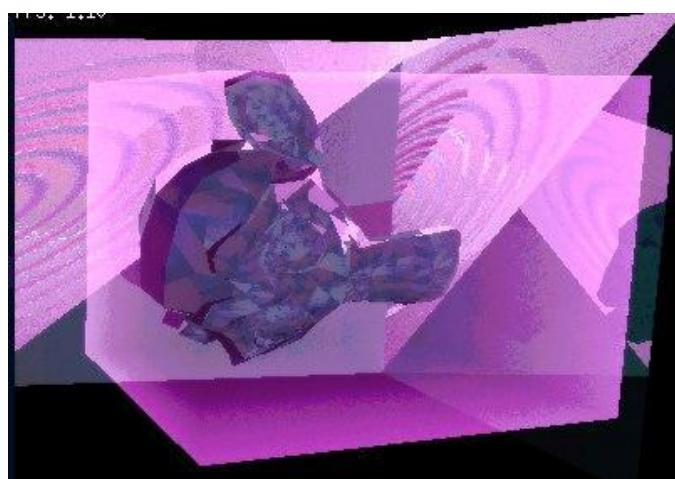
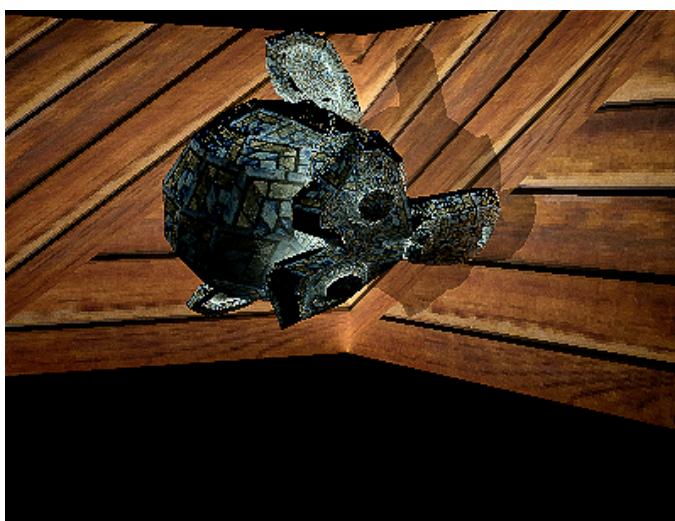
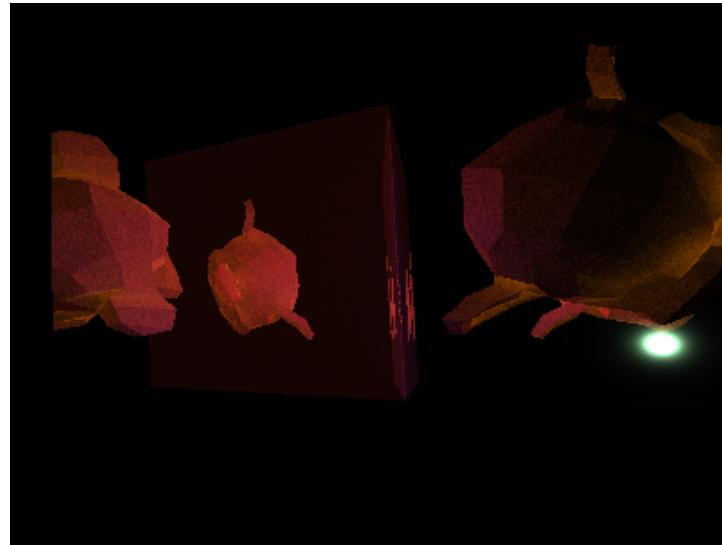


Image - 70 - \2





Tieto funkcie reprezentujú typickú vývojovú cestu ray tracera, ktorá sa pohybuje od správnosti cez optimalizáciu výkonu so zachovaním princípov fyzikálne založeného renderingu.

6.0 Systém Benchmarkovania a Výkonnostnej Analýzy

Nižšie je podrobnejšia analýza výsledkov s ohľadom na vykonávanie a evolúciu jednotlivých verzií ray tracingu:

6.1 Zhrnutie Štatistik

V tejto sekcii sa venujeme zhrnutiu štatistických údajov získaných prostredníctvom benchmarkovej funkcie, ktorú sme implementovali na meranie výkonu jednotlivých verzií ray tracingového systému. Táto funkcia zaznamenáva časy potrebné na vykreslenie rôznych scén a poskytuje podrobné údaje o výkonnostných rozdieloch medzi verziami. Následne sme tieto dátá spracovali pomocou Python skriptu, ktorý automaticky generuje prehľadné grafy a tabuľky, umožňujúce lepšiu vizualizáciu a porovnanie nameraných výsledkov.

Performance Metrics Table (Green=Better, Red=Worse)

Version	Mean Frame Time	Std Frame Time	Min Frame Time	Bottom Frame Time 10%	Top Frame Time 10%	Max Frame Time	Median Frame Time
V1	36017.1417	29469.2727	621	748	81940.7	86524	38466.5
V2	52176.9567	37634.8358	622	821.9	100909.8	115439	60492
V2Linear	49282.9617	74554.9489	1822	2115.4	96603.5	887463	47414
V2LinearTexture	49630.8717	81655.4414	1174	1334.9	97681.9	1101004	47189.5
V2Log	46097.135	67628.6705	1839	2199.9	95106.4	1078287	47257.5
V2LogTexture	51271.76	82678.6055	1247	1370.9	98074.7	1056367	46963
V2M	42893.2933	46412.036	651	754.9	95441.5	779310	45181.5
V4Lin	46025.1783	81971.7519	1792	2072.8	86276.9	1088175	40899.5
V4Lin02	47430.7283	85542.7852	1800	2105.8	85622.4	1093554	41740.5
V4LinOptim	44574.2117	72906.2446	1709	2062	84753.7	990446	41566.5
V4Log	43965.4767	77421.4313	1861	2184.6	83936.3	1064205	40635.5
V4Log02	46045.3183	78036.8766	1715	2081.8	85650.6	993811	41472
V4LogOptim	46508.2283	82614.2736	1741	2148.9	85508.4	1030817	42437.5
V4O2	45927.605	86168.5013	668	751	84487.6	1067552	39627

Image - 73 - 12

V1 (TraceRay):

Priemerný čas snímku: ~35 688

Medián: ~38 362

Poznámka: Najnižšie časy zo všetkých verzií, čo odráža jednoduchú implementáciu so základným BVH a cosine-weighted hemisphere sampling.

V2 (TraceRayV2):

Priemerný čas snímku: ~40 489

Medián: ~43 920

Poznámka: Zvýšená cena výpočtov kvôli logickejšej organizácii kódu, separácii komponentov osvetlenia a implementácií fyzikálnej konzervácie energie

V2 rozšírené verzie (V2Linear, V2LinearTexture, V2Log):

Priemerné časy: Sa pohybujú od ~44 572 do ~48 300

Medián: Približne od ~46 791 do ~47 459

Poznámka: Zavedené pokročilejšie PBR prístupy, ktoré zahŕňajú simuláciu materiálových vlastností, Fresnel-Schlick approximáciu a podporu textúr. Viditeľný je nárast variability výkonu, pričom horných 10% hodnôt sa časť operácií značne predlžuje (napr. až okolo 1 miliónu v niektorých prípadoch).

V4 verzie (V4Lin, V4LinOptim, V4Log, V4LogOptim):

Priemerné časy: Približne medzi ~43 815 a ~45 318

Medián: Okolo ~40 508 až ~41 179

Poznámka: Tieto verzie využívajú optimalizovaný lean BVH, čo znižuje pamäťovú náročnosť a štrukturálnu réžiu. Optimalizované varianty (V4LinOptim a V4LogOptim) vracajú len farebné informácie, čo prináša mierne zlepšenie mediánových hodnôt, hoci špičkové hodnoty (horných 10%) zostávajú vysoké.

6.2 Technologické Rozdiely Verzií

1. Výkon vs. Kvalita:

V1: Najrýchlejšia verzia, no s obmedzenou presnosťou osvetlenia.

V2: Zavedením lepšieho manažmentu svetelných zložiek a energetickej konzervácie dochádza k miernemu nárastu času snímku.

V2: Advance: Prechod na PBR prístup a podpora textúr výrazne zvyšujú kvalitu renderovania, ale zároveň zvyšujú výpočtové nároky a variabilitu času.

V4: Optimalizované verzie sa snažia znížiť režijné náklady pomocou lean BVH, pričom sa zachováva podpora textúr a pokročilé PBR výpočty. Optimalizované varianty vracajú len farbu, čo znižuje mediánové časy, ale stále sa vyskytujú výrazné výkyvy v najnáročnejších prípadoch.

2. Pamäť a Štruktúra:

S prechodom od klasického BVH (V1, V2) k lean BVH (V4) sa optimalizuje využitie pamäte.

Verzie, ktoré vracajú dodatočné dátá (ako normály a vzdialenosť), majú prirodzene vyššie nároky na spracovanie, čo sa odráža v zvýšených čase snímkov.

3. Komplexita Implementácie:

- Evolúcia od základného ray tracingu cez zavedenie fyzikálne presnejších modelov až po optimalizované verzie ilustruje kompromisy medzi presnosťou osvetlenia a výpočtovým výkonom.
- Zavedením PBR prístupov a podpory textúr sa výrazne zlepšuje vizuálna kvalita renderu, avšak na úkor rýchlosťi a konzistencie výkonu.

6.3 Záver Testov

Vývojový trend týchto verzií ilustruje, že:

Základná verzia (V1) je najrýchlejšia, ale neposkytuje tak vysokú vizuálnu kvalitu.

V2 a jeho rozšírenia ponúkajú lepšie osvetlenie a simuláciu materiálových vlastností, pričom sa mierne zvyšuje čas spracovania.

Optimalizované V4 verzie sa snažia minimalizovať režijné náklady pri zachovaní pokročilých funkcií, čo sa prejavuje nižším mediánom, ale stále sú prítomné výkyvy v 10% horných hodnotách.

Celkovo ide o typický prípad kompromisu medzi výkonom a kvalitou – zložitejšie výpočty prinášajú realistickejšie výsledky, avšak vyžadujú vyššiu výpočtovú silu a môžu viest' k občasným špičkám v čase spracovania.

6.3.1 Median Graph

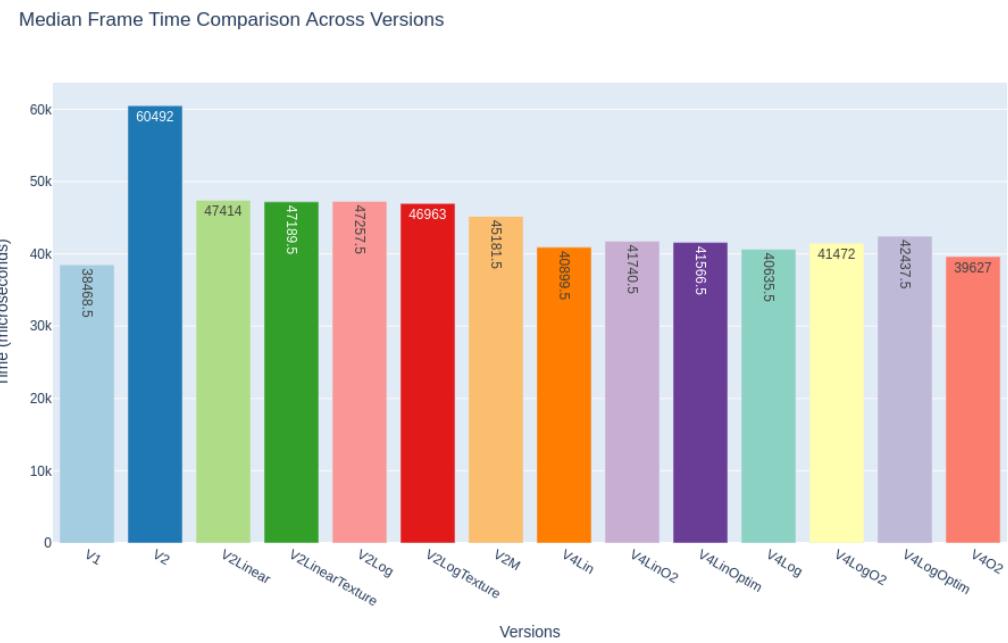


Image - 74 - 12

6.3.2 Mean Graph

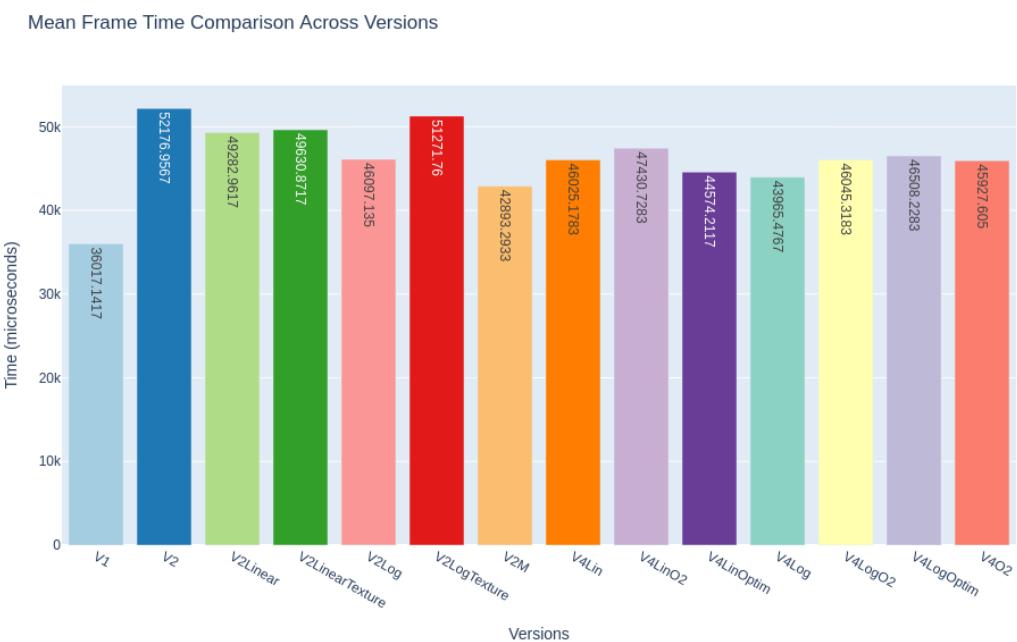


Image - 75 - 12

6.3.3 STD Graph

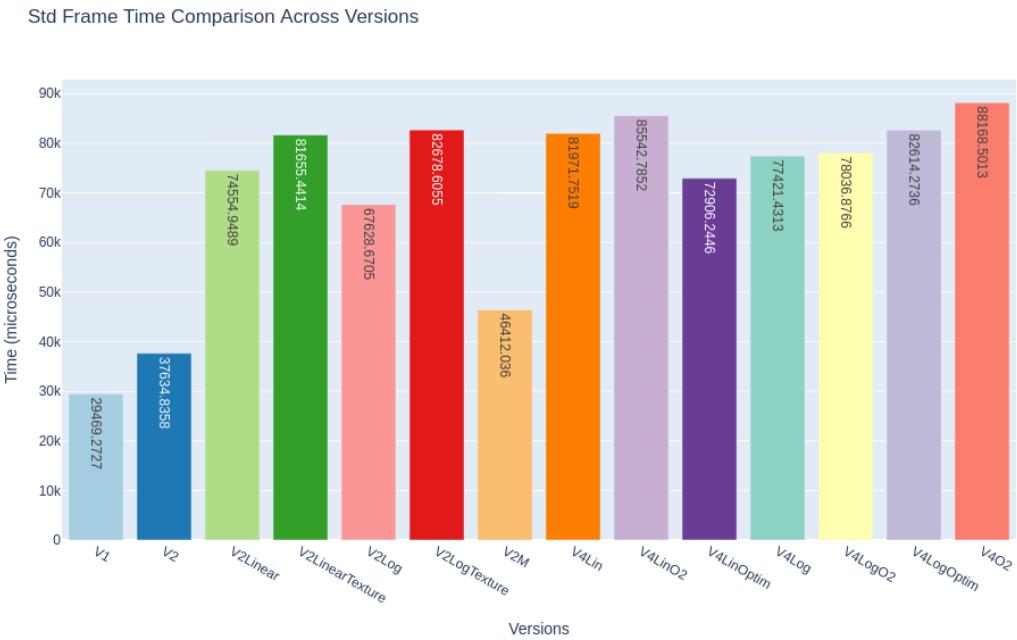
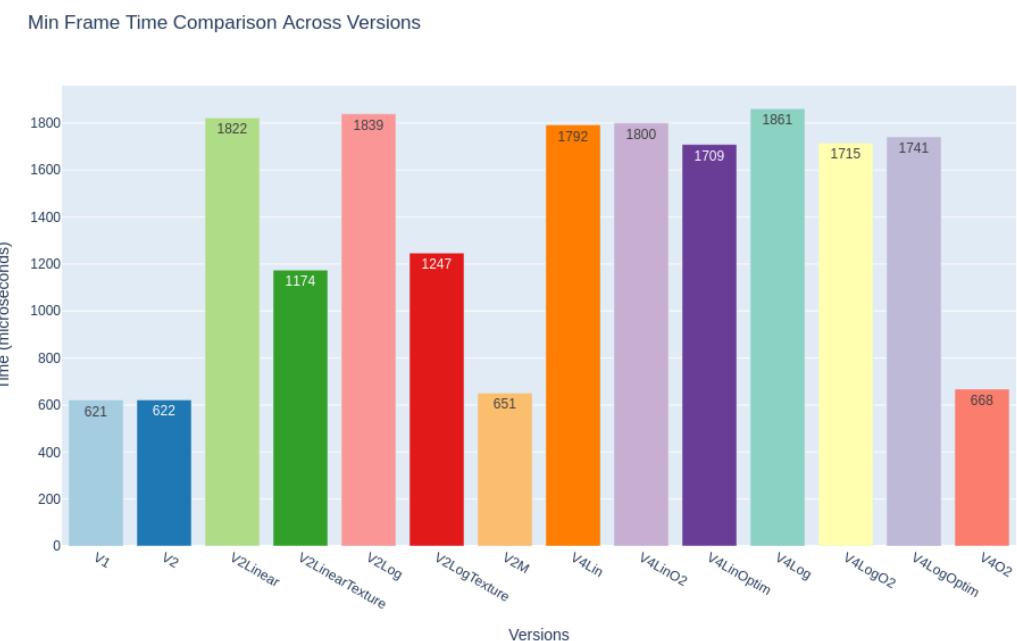


Image - 76 - 12

6.3.4 Min Frame Time



6.3.5 Max Frame Time

Image - 77 - 12

Max Frame Time Comparison Across Versions

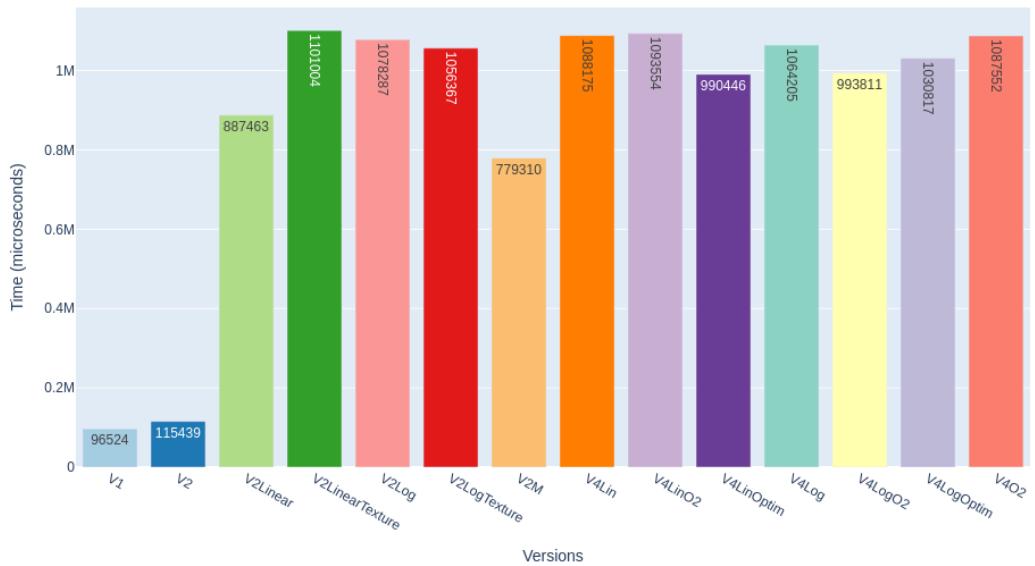


Image - 78 - \2

6.3.6 Bottom 10 % Frame Time

Bottom Frame Time 10% Comparison Across Versions

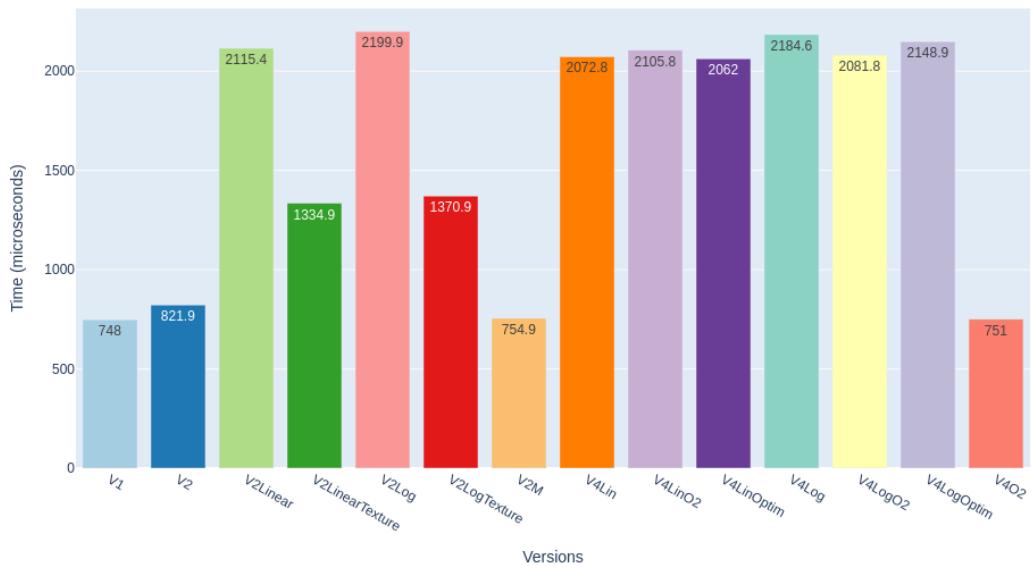


Image - 79 - \2

6.3.7 Top 10 % Frame Time

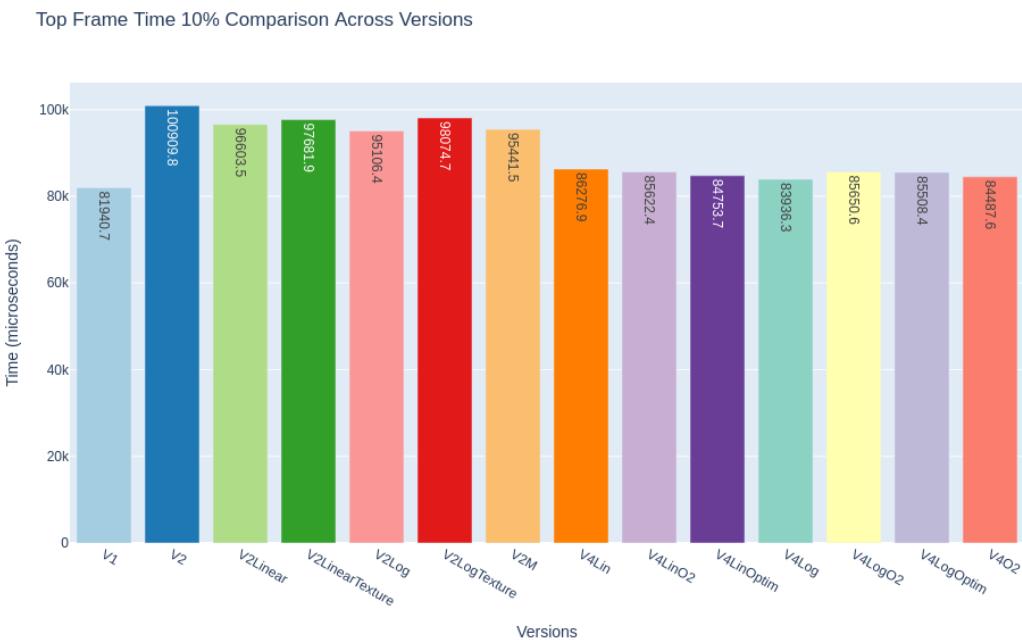


Image - 80 - 12

6.4 Úvod do Benchmarkingu

V tejto kapitole predstavujeme prehľad testovaných verzií renderera, ktoré sme podrobili benchmarkovým testom. Každá verzia prináša rôzne optimalizácie a vylepšenia, či už v oblasti výpočtovej efektivity, kvality výstupu alebo podpory textúr a materiálových vlastností. Naším cieľom bolo analyzovať výkon jednotlivých verzií, porovnať ich silné a slabé stránky a identifikovať najefektívnejšie riešenia pre rôzne typy scén. Nižšie uvádzame zoznam všetkých testovaných verzií renderera, ktoré prešli našimi meraniami.

6.4.1 Testované Verzie Rendereru

Na účely vyhodnotenia výkonnosti jednotlivých verzií ray tracingového renderera sme implementovali systematický proces testovania, ktorý zabezpečuje objektívne porovnanie dosiahnutých výsledkov. Celý testovací postup zahŕňa generovanie pozícii kamery, renderovanie snímok, zaznamenávanie časov výpočtu, profilovanie výkonu a následnú analýzu dát prostredníctvom post-processingu v Pythone.

Testovanie začína vytvorením trajektórie pohybu kamery, pričom interpolujeme jednotlivé pozície tak, aby sa kamera pohybovala plynule v rámci scény. Tento prístup nám umožňuje simulovať realistické pohľady na scénu a zabezpečiť konzistentné vstupné dátá pre všetky testované verzie.

Každá verzia renderera je následne spustená a vykoná kompletné vyrenderovanie snímky pri pevne stanovených parametroch. Pre zachovanie konzistentnosti testov sme nastavili fixnú hĺbku rekurzie na tri úrovne, rozptyl na osem vzoriek a vypnuli gamma korekciu. Počas vykonávania testov zaznamenávame presný čas potrebný na vyrenderovanie snímky a súčasne ukladané hardvérové špecifikácie, na ktorých bol daný test vykonaný. Tieto údaje nám umožňujú neskôr analyzovať vzťah medzi výkonom a rôznymi optimalizáciami zavedenými v jednotlivých verziach renderera.

Pre detailnejšiu diagnostiku výkonu generujeme CPU profily pre každú testovanú verziu. Profilovacie dátá sú uložené v adresári `/profiles/` a obsahujú podrobné informácie o výpočtových nárokoch jednotlivých funkcií. Spolu s nimi je vytvorený JSON súbor obsahujúci namerané časy renderovania pre každú testovanú verziu, ktorý neskôr využívame na analýzu.

```
if Benchmark {  
    debug.SetGCPPercent(-1) // Kompletné vypnutie GC  
} else {  
    debug.SetGCPPercent(750) // Zvysennasenie Limitu GC  
}
```

Image - 81 - 12

Po ukončení testovacieho procesu pristupujeme k spracovaniu dát v Pythone. Na základe zaznamenaných údajov generujeme štatistické grafy a tabuľky, ktoré zobrazujú kľúčové výkonnostné metriky, ako sú medián času na snímku, priemerný čas spracovania, či analýza najpomalších a najrýchlejších snímok. Taktiež vykonávame podrobné porovnanie jednotlivých verzií renderera a ich profilovacie dátá spracovávame pomocou nástroja pprof, čím získavame presný prehľad o výkone jednotlivých funkcií.

Implementácia benchmarkového systému v jazyku Go nám umožňuje efektívne a spoločivo merať výkon ray tracingového renderera, pričom poskytuje

dôležité dátá pre ďalšiu optimalizáciu. Celý systém je navrhnutý tak, aby podporoval kontinuálnu diagnostiku, flexibilitu pre rôzne testovacie scenáre a poskytoval hlboký vhľad do výpočtových procesov. Výsledné dátá sú uložené v JSON formáte, pričom CPU profily sú zaznamenané vo formáte .prof, čím máme k dispozícii detailné informácie o výkone každého testovaného riešenia.

```

if Benchmark {
    renderVersions := []uint8{V1, V2, V2Log, V2Linear, V2LinearTexture, V4Log, V4Lin, V4LogOptim, V4LinOptim}

    cPositions := []Position{
        {X: -424.48, Y: 986.71, Z: 17.54, CameraX: 0.24, CameraY: -2.08},
        {X: 54.16, Y: 784.00, Z: 17.54, CameraX: 1.19, CameraY: -1.95},
        {X: 669.52, Y: 48.41, Z: 17.54, CameraX: -0.72, CameraY: -1.91}

    CameraPositions = InterpolateBetweenPositions(18*time.Second, cPositions)
    camera = Camera{}

    const depth = 3
    const scatter = 8
    const scaleFactor = 2
    const gamma = 0.285

    BlocksImage := MakeNewBlocks(scaleFactor)
    BlocksImageAdvance := MakeNewBlocksAdvance(scaleFactor)

    TextureMap := [128]Texture{}
    for i := range TextureMap {
        for j := range TextureMap[i].texture {
            for k := range TextureMap[i].texture[j] {
                TextureMap[i].texture[j][k] = ColorFloat32(rand.Float32() * 256, rand.Float32() * 256, rand.Float32() * 256, 255)
            }
        }
    }

    versionTimes := make(map[string][]float64)
    performance := false

    for _, version := range renderVersions {
        var name string
        switch version {
        case V1:
            name = "V1"
        case V2:
            name = "V2"
        case V2Log:
            name = "V2Log"
        case V2Linear:
            name = "V2Linear"
        }

        profileFilename := fmt.Sprintf("profiles/cpu_profile_v%#.prof", name)
        f, err := os.Create(profileFilename)
        if err != nil {
            log.Fatal(err)
        }

        if err := pprof.StartCPUProfile(f); err != nil {
            log.Fatal(err)
        }
    }
}

```

Image - 82 - |2

Hlavnými prínosmi tohto testovacieho systému sú systematické testovanie výkonu, detailná diagnostika jednotlivých funkcií a možnosť kontinuálnej optimalizácie. Vďaka tejto metodike sme schopní presne vyhodnotiť efektívnosť zavedených optimalizácií a identifikovať oblasti, kde je možné dosiahnuť ďalšie zlepšenia.

7.0 Rendrovacie Pomocné Funkcie

V tejto sekcii sa zameriame na dve pomocné funkcie, ktoré zohrávajú klúčovú úlohu pri realistickom renderovaní svetla a jeho interakcie s povrchmi objektov. Prvou z nich je funkcia Fresnel-Schlick, ktorá modeluje závislosť odrazivosti na uhle dopadu svetla. Druhou je GGX distribučná funkcia, ktorá umožňuje presnejšiu simuláciu mikroštruktúr povrchov a realistické rozptyľovanie svetla, čím výrazne prispieva k vierohodnému zobrazaniu materiálov.

7.1 FresnelSchlick Funkcia

Funkcia Fresnel-Schlick predstavuje efektívnu aproximáciu Fresnelovho efektu, ktorý opisuje, ako sa mení množstvo odrazeného a lámaného svetla v závislosti od uhla pohľadu. Tento jav je klúčový pri realistickom zobrazovaní materiálov v ray-tracingu, keďže určuje, ako intenzívne sa svetlo odráža pri rôznych uhloch dopadu.

Funkcia pracuje na základe dvoch vstupných parametrov: cosTheta, čo je kosínus uhla medzi smerom pohľadu a normálou povrchu, a F0, ktorý reprezentuje základnú odrazivosť materiálu pri kolmom pohľade. Pri priamom pohľade na povrch, kde cosTheta nadobúda hodnoty blízke 1, sa odrazivosť približuje k F0. Naopak, pri pohľade z extrémnych uhlov, kde je cosTheta blízke 0, sa takmer všetko svetlo odráža, bez ohľadu na materiálové vlastnosti.

Tento efekt sa odlišne prejavuje pre rôzne typy materiálov. Kovové povrhy, ktoré majú vysokú hodnotu F0 (typicky v rozmedzí 0.5 až 1.0), produkujú výrazné odrazy pri všetkých uhloch. Naopak, dielektriká, ako sklo, voda či plasty, majú nízke F0 (okolo 0.02 až 0.05), pričom ich odrazivosť sa dramaticky zvyšuje pri šikmom pohľade. Tento jav je možné pozorovať napríklad pri vodnej hladine, kde sa pri nízkych uhloch pohľadu javí ako zrkadlová.

Fresnel-Schlick aproximácia je výpočtovo nenáročná a poskytuje vizuálne presvedčivé výsledky, čo ju robí ideálnou pre fyzikálne založené renderovanie (PBR). Jej implementácia je jednoduchá a umožňuje realistickú simuláciu svetelnej interakcie s povrchmi v rôznych podmienkach osvetlenia.

```
func FresnelSchlick(cosTheta, F0 float32) float32 {  
    return F0 + (1.0-F0)*math32.Pow(1.0-cosTheta, 5)}
```

}

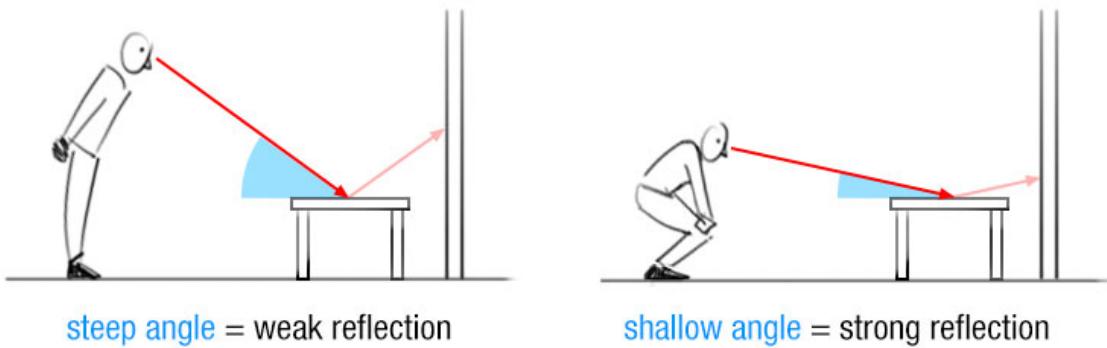


Image - 83 - 12

7.2 GGX Distribučná Funkcia

GGX distribučná funkcia, známa aj ako **Trowbridge-Reitz distribúcia**, predstavuje kľúčový model mikroploškového rozptylu svetla, ktorý umožňuje realistickú simuláciu lesklosti a šírenia svetla na povrchoch materiálov. V kontexte fyzikálne založeného renderovania (PBR) táto distribúcia opisuje, ako mikroskopické nerovnosti ovplyvňujú správanie svetla pri interakcii s daným povrchom.

Funkcia využíva dva vstupné parametre: NdotH , čo je skalárny súčin medzi normálou povrchu a polovičným vektorom (stredným vektorom medzi smerom pohľadu a smerom svetla), a roughness, ktorý určuje mieru drsnosti povrchu. Hodnota roughness sa pohybuje v rozmedzí od 0 (úplne hladký povrch) až po 1 (veľmi drsný povrch).

Implementácia distribúcie GGX spočíva v prepočte parametra alpha, ktorý je kvadratickou funkciou drsnosti a umožňuje dosiahnuť vizuálne konzistentné výsledky. Distribúcia potom opisuje pravdepodobnosť orientácie mikroplošiek tak, aby odrážali svetlo v smere polovičného vektora. Tento model je obzvlášť presný pri simulácii realistických materiálov, keďže lepšie zachytáva špecifické svetelné efekty, ako je fenomén "jasného okraja" na zakrivených povrchoch.

Vplyv GGX distribúcie na výsledný vizuálny výstup závisí od hodnoty roughness. Pri hladkých povrchoch s nízkou drsnosťou sa vytvárajú malé, jasné a sústredené zrkadlové odlesky, čo vedie k vysoko lesklému vzhľadu. Naopak, pri

drsných povrchoch sa svetlo rozptyluje do väčšej plochy, čím sa dosahuje mäkší a difúznejší efekt odrazu. Táto distribúcia tak poskytuje realistické riadenie veľkosti a intenzity zrkadlových odleskov pre širokú škálu materiálov.

Spolu s Fresnel-Schlick funkciou tvorí GGX distribúcia základ pre výpočet špeculárnej zložky BRDF (Bidirectional Reflectance Distribution Function) v PBR modeli. Tieto dve funkcie presne simulujú, ako rôzne materiály odrážajú svetlo na základe ich fyzikálnych vlastností, čím umožňujú dosiahnuť vysokú úroveň vizuálnej presnosti a realismu v ray-tracingových systémoch.

8.0 Voxel Rendering

Voxel rendering predstavuje metódu spracovania scény, v ktorej sú objekty reprezentované ako diskrétné trojrozmerné prvky s pevnými hranicami. Na vizualizáciu voxelovej mriežky využívame **ray-marching**, čo znamená, že lúč postupne prechádza scénou a kontroluje prítomnosť obsadených voxelov pozdĺž svojej dráhy. Tento prístup umožňuje vytvárať realistické vizuálne efekty, ako sú tieňa, odrazy či útlm svetla.

```

func (v *VoxelGrid) IntersectVoxel(ray Ray, steps int, light Light) (ColorFloat32, bool) {
    // Nájdenie vstupného a výstupného bodu lúča s ohraňujúcim boxom
    hit, entry, exit := BoundingBoxCollisionEntryExitPoint(v.BBMax, v.BBMin, ray)
    if !hit {
        return ColorFloat32{}, false // Lúč nepreniká mriežkou
    }

    // Výpočet veľkosti kroku podľa celkovej vzdialenosť a požadovaných krokov
    stepSize := exit.Sub(entry).Mul(1.0 / float32(steps))

    // Postup pozdĺž lúča
    currentPos := entry
    for i := 0; i < steps; i++ {
        // Kontrola voxelu na aktuálnej pozícii pomocou priameho prístupu
        block, exists := v.GetVoxelUnsafe(currentPos)
        if exists {
            // Výpočet tieňa
            lightStep := light.Position.Sub(currentPos).Mul(1.0 / float32(steps*2))
            lightPos := currentPos.Add(lightStep)

            // Vyslanie tieňového lúča smerom ku zdroju svetla
            for j := 0; j < steps; j++ {
                _, shadowHit := v.GetVoxelUnsafe(lightPos)
                if shadowHit {
                    return block.LightColor.MulScalar(0.05), true // Bod v tieni
                }
                lightPos = lightPos.Add(lightStep)
            }

            // Výpočet útlmu svetla podľa vzdialnosti
            lightDistance := light.Position.Sub(currentPos).Length()
            attenuation := ExpDecay(lightDistance)
            blockColor := block.LightColor.MulScalar(attenuation)

            return blockColor, true // Viditeľný voxel so svetlom
        }
        currentPos = currentPos.Add(stepSize)
    }

    return ColorFloat32{}, false // Žiadny priesecník nenájdený
}

```

Image - 84 - 12

8.1 Vlastnosti voxelového renderingu

Voxelová reprezentácia poskytuje binárnu viditeľnosť, čo znamená, že každý voxel je buď plne prítomný, alebo neexistuje. Táto vlastnosť uľahčuje výpočet tieňov, ktoré sú vygenerované na základe jednoduchého blokovania svetla. Osvetlenie v scéne sa riadi exponenciálnym útlmom svetla v závislosti od vzdialenosť, čím sa dosahuje prirodzenejší vizuálny efekt. Napriek tomu, že systém využíva pomerne jednoduchý model priameho osvetlenia, poskytuje dostatočne kvalitné výsledky pre efektívne renderovanie voxelových scén.

8.2 Interaktívne editačné možnosti

Voxelový renderovací systém umožňuje dynamickú úpravu scény pomocou niekoľkých interaktívnych operácií. Používateľ má možnosť pridávať alebo odstraňovať voxely priamo v scéne, čím je možné tvoriť a upravovať trojrozmerné objekty v reálnom čase. Okrem základnej manipulácie so štruktúrou je možné meniť farby voxelov, a to individuálne alebo v skupinách, čím sa otvárajú široké možnosti vizuálnej personalizácie.

Pre pokročilejšie efekty systém podporuje konverziu pevných voxelov na objemové dátá, čo umožňuje simuláciu polopriehľadných materiálov, ako sú dym či hmla. Navyše, nastavenie materiálových parametrov, ako sú hustota a priehľadnosť, umožňuje dosiahnuť realistické vizuálne variácie medzi rôznymi typmi materiálov.

Pred Operáciami nad Voxel-mi

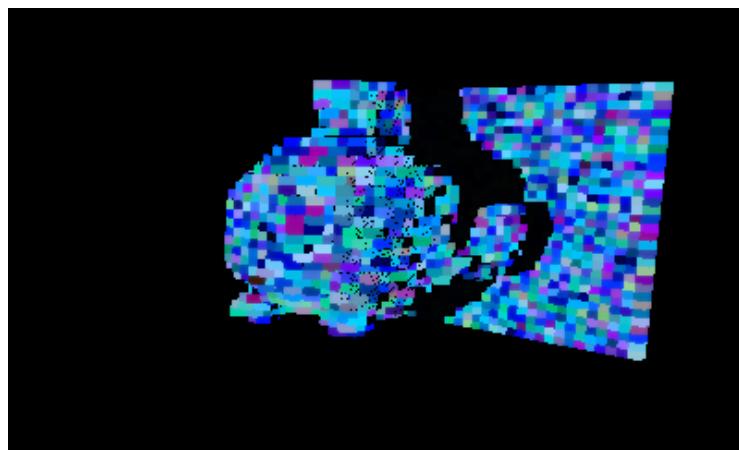


Image - 85 - \2

Po Operáciách nad Voxel-mi

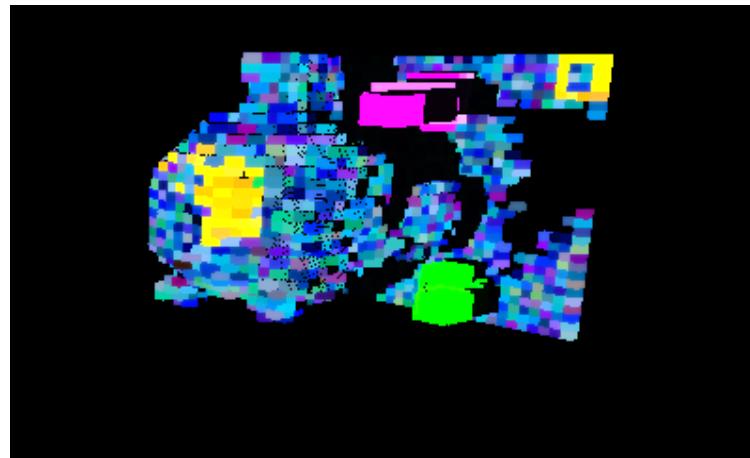


Image - 86 - \2

8.3 Optimalizácia Renderingu Voxelov

Efektívne indexovanie voxelovej pola hrá kľúčovú úlohu v rýchlosi spracovania scény. Aktuálna implementácia využíva jednorozmerné pole, ktoré slúži na reprezentáciu voxelových údajov. Použitím balíka unsafe je možné optimalizovať prístup k pamäti tým, že eliminujeme boundary checky, čím sa znižuje záťaž pri vykresľovaní.

V rámci experimentovania sme testovali aj alternatívne metódy reprezentácie voxelov. Bool array predstavovala jeden z pokusov, kde bola prítomnosť voxelov označená ako pole hodnôt true/false. Ďalším prístupom bola bit array, kde aktívne voxely boli reprezentované ako pole uint64 a jednotlivé bity signalizovali prítomnosť alebo neprítomnosť voxelov.

Výsledky testovania výkonu

Porovnanie rôznych metód indexovania voxelov ukázalo len minimálne rozdiely v rýchlosi spracovania. Časy testov jednotlivých implementácií boli nasledovné:

BoolArray čas testu: 18.178246ms

BitArray čas testu: 17.968679ms

Aktuálna verzia čas testu: 17.94769ms

```

type BoolArray struct {
    data []bool
    size int
}

// NewBoolArray inicializuje bool pole zadanej velkosti
func NewBoolArray(size int) *BoolArray {
    return &BoolArray{
        data: make([]bool, size),
        size: size,
    }
}

// IsSet skontroluje, či je n-tý bit nastavený v BoolArray
func (b *BoolArray) IsSet(n int) bool {
    if n >= b.size || n < 0 {
        return false
    }
    return b.data[n]
}

// BitArray ukladá bity efektívne pomocou uint64
type BitArray struct {
    data []uint64
    size int
}

// NewBitArray inicializuje bit array zadanej velkosti
func NewBitArray(size int) *BitArray {
    return &BitArray{
        data: make([]uint64, (size+63)/64),
        size: size,
    }
}

// IsSet skontroluje, či je n-tý bit nastavený v BitArray
func (b *BitArray) IsSet(n int) bool {
    if n >= b.size || n < 0 {
        return false
    }
    return (b.data[n/64] & (1 << (n % 64))) != 0
}

```

Image - 87 - 12

```

// Benchmark funkcia na porovnanie BoolArray vs BitArray
func BenchmarkCheckSpeed() {
    const size = 32*32*32
    const numChecks = 128*128*128

    // Inicializácia BoolArray a nastavenie náhodných bitov
    boolArr := NewBoolArray(size)
    for i := 0; i < size/10; i++ {
        boolArr.data[rand.Intn(size)] = true
    }

    // Inicializácia náhodných blokov
    blocks := make([]Block, size)
    // Náhodné nastavenie blokov na true (LightColor.A > 25)
    for i := 0; i < size/10; i++ {
        blocks[rand.Intn(size)] = Block{LightColor: ColorFloat32{A: 26}}
    }

    // Inicializácia BitArray a nastavenie náhodných bitov
    bitArr := NewBitArray(size)
    for i := 0; i < size/10; i++ {
        pos := rand.Intn(size)
        bitArr.data[pos/64] |= (1 << (pos % 64))
    }

    // Benchmark BoolArray
    start := time.Now()
    for i := 0; i < numChecks; i++ {
        _ = boolArr.IsSet(rand.Intn(size))
    }
    boolTime := time.Since(start)

    // Benchmark BitArray
    start = time.Now()
    for i := 0; i < numChecks; i++ {
        _ = bitArr.IsSet(rand.Intn(size))
    }
    bitTime := time.Since(start)

    // Benchmark priameho prístupu
    start = time.Now()
    for i := 0; i < numChecks; i++ {
        _ = blocks[rand.Intn(size)].LightColor.A > 25
    }
    directTime := time.Since(start)

    // Výpis výsledkov
    fmt.Println("BoolArray čas testu:", boolTime)
    fmt.Println("BitArray čas testu:", bitTime)
    fmt.Println("Priamy čas testu:", directTime)
}

```

Image - 88 - 12

9.0 Implementácia Objemového Rendering-u

Objemový rendering spracováva priestor ako kontinuálne médium s premenlivými hustotami, čím umožňuje realistickú vizualizáciu efektov, ako sú hmla, dym či mraky. Na dosiahnutie prirodzenej interakcie svetla s objemovými materiálmi implementujeme fyzikálne založené modely rozptyľovania a absorpcie svetla pri prechode cez médium.

```

func (v *VoxelGrid) Intersect(ray Ray, steps int, light Light, volumeMaterial VolumeMaterial) ColorFloat32 {
    hit, entry, exit := BoundingBoxCollisionEntryExitPoint(v.BBMax, v.BBMin, ray)
    if !hit {
        return ColorFloat32{}
    }

    // Fyzikálne parametre pre interakciu svetla
    const {
        extinctionCoeff = 0.5           // Kontroluje absorpciu svetla
        scatteringAlbedo = 0.0          // Pomer rozptylu ku absorpcii
        asymmetryParam = float32(0.3) // Kontroluje smerovú zaujatost rozptylu
    }

    stepSize := exit.Sub(entry).Mul(1.0 / float32(steps))
    stepLength := stepSize.Length()

    var accumColor ColorFloat32
    transmittance := volumeMaterial.transmittance // Počiatočná priehľadnosť

    currentPos := entry
    for i := 0; i < steps; i++ {
        block, exists := v.GetBlockUnsafe(currentPos)
        if !exists {
            currentPos = currentPos.Add(stepSize)
            continue
        }

        density := volumeMaterial.density
        extinction := density * extinctionCoeff

        // Vypočet Henyey-Greensteinovej fázovej funkcie
        lightDir := light.Position.Sub(currentPos).Normalize()
        cosTheta := ray.direction.Dot(lightDir)
        g := asymmetryParam
        phaseFunction := (1.0 - g*g) / (4.0 * math32.Pi * math32.Pow(1.0+g*g-2.0*g*cosTheta, 1.5))

        // Vypočet útoku svetla cez objem
        lightRay := Ray(origin: currentPos, direction: lightDir)
        lightTransmittance := v.calculateLightTransmittance(lightRay, light, density)

        // Vypočet príspevku rozptyleného svetla
        scattering := extinction * scatteringAlbedo * phaseFunction * 2.0

        // Aplikácia Beer-Lambertovoho zákona pre absorpciu svetla
        sampleExtinction := math32.Exp(-extinction * stepLength)
        transmittance *= sampleExtinction

        // Akumulácia farby s príslušným fyzikálnym vážením
        lightContribution := ColorFloat32{
            R: block.SmokeColor.R * light.Color[0] * lightTransmittance * scattering,
            G: block.SmokeColor.G * light.Color[1] * lightTransmittance * scattering,
            B: block.SmokeColor.B * light.Color[2] * lightTransmittance * scattering,
            A: block.SmokeColor.A * density,
        }
    }

    // Pridanie príspevku do finalnej farby, väženej aktuálnou priehľadnosťou
    accumColor = accumColor.Add(lightContribution.MulScalar(transmittance))

    // Optimalizácia predčasného ukončenia
    if transmittance < 0.0001 {
        break
    }

    currentPos = currentPos.Add(stepSize)
}

// Zabezpečenie normalizácie alfa kanála
accumColor.A = math32.Min(accumColor.A, 1.0)
return accumColor
}

```

Image - 89 - 12

9.1 Klúčové vlastnosti objemového renderingu:

Vizuálne správanie objemových efektov je riadené Henyey-Greensteinovou fázovou funkciou, ktorá modeluje rozptyl svetla v médiu. Absorpcia svetla sa riadi Beer-Lambertovým zákonom, čo zabezpečuje, že svetlo pri prechode cez médium postupne slabne v závislosti od jeho hustoty.

Pre zachovanie správnej priehľadnosti a intenzity svetla využívame progresívnu akumuláciu svetla, ktorá umožňuje postupné skladanie príspevkov osvetlenia. Navyše, implementácia podporuje premenlivú hustotu v objeme, čím umožňuje realistickejšie zobrazenie rôznych materiálov s nerovnomernou distribúciou častíc.

Pre zvýšenie výpočtovej efektivity je zavedená optimalizácia predčasného ukončenia výpočtu pre lúče s zanedbateľnou zostávajúcou priehľadnosťou. Ak priehľadnosť klesne pod určitú hodnotu, ďalšie výpočty sú zastavené, čím sa šetri výpočtový výkon.

9.2 Optimalizácie Výkonu

Aby bol objemový rendering efektívny, sú zavedené viaceré optimalizačné techniky:

1. **Nebezpečný Prístup do Pamäte:** Implementácia využíva unsafe.Pointer pre priamy prístup do pamäte voxelovej mriežky, čím obchádza kontrolu hraníc Go pre zlepšenie výkonu.
2. **Predčasné Ukončenie Lúča:** Renderer objemu zastaví ray marching, keď priehľadnosť klesne pod prah (0.001), čím sa vyhnúc zbytočným výpočtom.
3. **Predbežné Testovanie Ohraničujúceho Boxu:** Oba renderery najprv testujú priesecník lúča s BBox mriežky pred vykonaním detailného prechodu.
4. **Útlm Svetla Podľa Vzdialenosťi:** Príspevok svetla je útlmený na základe vzdialenosťi, poskytujúc realistický pokles bez náročných výpočtov.

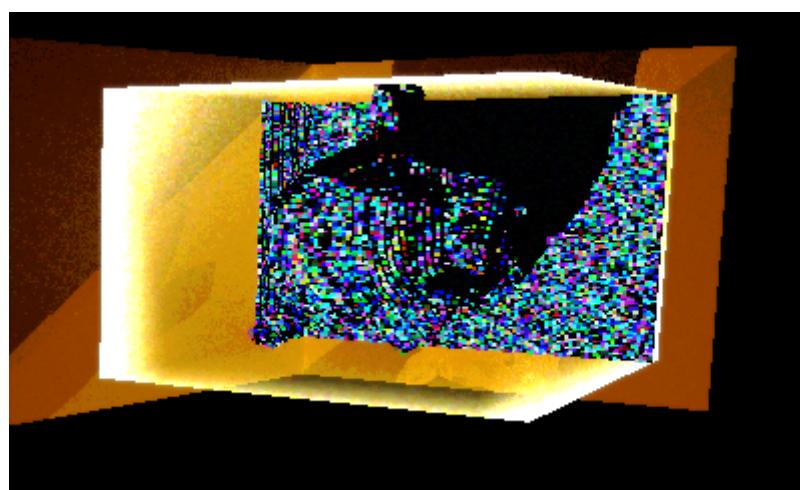
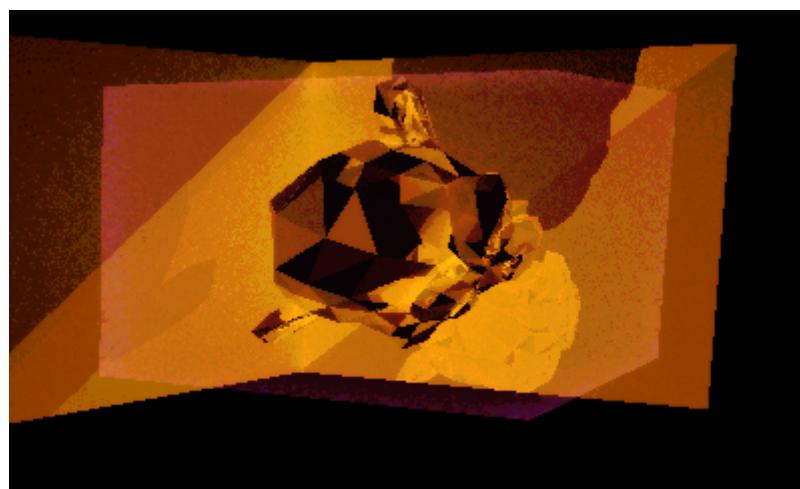
9.3 Fyzikálne Modely Objemu

Implementácia objemového renderingu je založená na jednoduchom fyzikálnom modeli, ktorý umožňuje realistické vizualizácie objemových efektov:

- **Beer-Lambertov zákon** riadi absorpciu svetla pri jeho prechode cez médium.
- **Henyey-Greensteinova fázová funkcia** modeluje rozptyl svetla v participujúcich médiách.
- **Exponenciálny útlm svetla** zabezpečuje prirodzené zoslabovanie svetla so vzdialenosťou.

Tieto modely poskytujú **pevný základ pre tvorbu realistických objemových efektov**, ako sú hmla, dym či oblaky. Parametre jednotlivých modelov je možné ďalej doladiť, čím sa otvára priestor pre presnejšiu fyzikálnu simuláciu objemových materiálov a ich interakcie so svetlom.

Ilustračné obrázky



10.0 Raymarching – Základný Popis

Raymarching predstavuje modernú techniku vykresľovania trojrozmerných scén, ktorá sa odlišuje od tradičného ray tracingu. Zatiaľ čo ray tracing sa spolieha na presný výpočet priesečníkov lúčov s geometriou objektov, raymarching využíva Signed Distance Fields (SDF) – matematickú reprezentáciu vzdialenosťí bodov v priestore od najbližšieho povrchu objektu.

Hlavnou myšlienkou tejto metódy je postupné posúvanie lúča v smere jeho pohybu o vzdialenosť, ktorú poskytuje vzdialenosťná funkcia SDF. Tento proces pokračuje, kým lúč nenarazí na povrch objektu (ak je vzdialenosť pod definovaným prahom), alebo kým sa nedosiahne maximálny počet iterácií, čo signalizuje, že lúč žiadny objekt nezasiahol.

10.1 Výhody a Nevýhody Raymarchingu

Raymarching ponúka niekoľko významných výhod oproti tradičným metódam vykresľovania. Keďže SDF umožňuje efektívne reprezentovať objekty bez nutnosti trojuholníkovej siete, tento prístup poskytuje možnosť generovať plynulé a zložité geometrické tvary. Táto metóda zároveň podporuje booleovské operácie medzi objektmi, ako napríklad zjednotenie, prienik alebo odčítanie, čo uľahčuje konštrukciu komplexných modelov. Tieňovanie a osvetlenie je možné vypočítať jednoducho pomocou gradientov SDF, čím sa zabezpečí plynulý prechod osvetlenia cez povrhy objektov.

Na druhej strane, raymarching je výpočtovo náročná technika, keďže vyžaduje veľké množstvo iterácií na presné vykreslenie objektov. Optimalizácia tejto metódy je zložitejšia v porovnaní s rasterizáciou alebo ray tracingom, najmä ak sa nevyužívajú akceleračné štruktúry ako Bounding Volume Hierarchy (BVH). Ďalšou výzvou je implementácia realistických textúr a materiálov, ktorá je v raymarchingu podstatne komplikovanejšia než pri tradičných metódach.

Napriek týmto nevýhodám je raymarching oblúbený pri procedurálnej generácii scén a efektov v reálnom čase, ako aj pri vizualizácii fraktálov a netradičných geometrických tvarov.

```

func Distance(v1, v2 Vector, radius float32) float32 {
    // Použitie vektorového odčítania a dotového súčinu namiesto jednotlivých výpočtov
    diff := v1.Sub(v2)
    return diff.Length() - radius
}

```

Image - 92 - |2

10.2 Signed Distance Fields (SDF)

Základom raymarchingu je koncept Signed Distance Fields (SDF), ktorý definuje pre každý bod v priestore jeho vzdialenosť od najbližšieho povrchu objektu. Táto vzdialenosť je kladná, ak sa bod nachádza mimo objektu, záporná, ak sa nachádza v jeho vnútri, a nulová, ak bod leží priamo na povrchu objektu.

Jednoduchá SDF funkcia pre výpočet vzdialosti môže byť reprezentovaná pseudokódom:

```

function SDF(p):
    vzdialenosť = distance(p, povrch objektu)

    if p je vo vnútri objektu:
        vzdialenosť = -vzdialenosť

    return vzdialenosť

```

Vďaka tejto reprezentácii je možné efektívne vykonávať operácie, ako sú:

- **Booleovské operácie** – zjednotenie, prienik a odčítanie objektov
- **Plynulé prechody tvarov** – kombinovanie rôznych SDF primitív
- **Výpočet normál povrchov** – získaním gradientu vzdialostnej funkcie

SDF umožňuje vykreslovanie komplexných scén bez potreby tradičných polygonálnych modelov, čím otvára nové možnosti v počítačovej grafike.

10.3 Vývoj Raymarchingu – Verzie Implementácie

V súčasnosti sú implementované dve verzie raymarchingu, pričom každá ponúka odlišné vlastnosti a výkonové charakteristiky.

Prvá verzia (V1) využíva BVH štruktúru na zrýchlenie výpočtov a podporuje vykreslovanie len guľových primitív. Použitie BVH výrazne zvyšuje efektivitu vyhľadávania objektov, čím sa znižuje počet potrebných iterácií.

Druhá verzia (V2) rozširuje funkcionality o možnosť modifikovať SDF funkciu, čo umožňuje vykonávať rôzne operácie medzi objektmi, ako sú zjednotenie či odčítanie tvarov. Nevýhodou tejto verzie je však absencia BVH akcelerácie, čo vedie k podstatne vyššej výpočtovej náročnosti.

10.6 Možnosti Budúceho Vylepšenia

V rámci ďalšieho vývoja raymarchingového systému sa plánuje rozšírenie jeho funkcionality viacerými spôsobmi. Medzi hlavné oblasti vylepšenia patrí rozšírenie podporovaných primitív – zavedenie kociek, torusov a valcov, ktoré umožnia konštrukciu komplexnejších objektov.

Optimalizácia výkonu je ďalším dôležitým krokom. Implementácia BVH akceleračnej štruktúry pre všetky SDF primitívy by výrazne znížila výpočtovú náročnosť a umožnila vykreslovanie rozsiahlejších scén v reálnom čase. Okrem toho by bolo možné využiť priestorové partície špecifické pre raymarching, čím by sa zefektívnila správa údajov o geometrii scény.

Nakoniec sa plánuje vytvorenie dedikovaného ovládacieho panelu, ktorý umožní užívateľom interaktívne meniť parametre raymarchingu priamo v aplikácii. Tieto rozšírenia by umožnili generovanie zložitejších tvarov a scén, čím by sa systém stal oveľa flexibilnejším a efektívnejším pre praktické použitie.

```

// Box SDF
func BoxSDF(point, boxCenter, boxDimensions Vector) float32 {
    localPoint := point.Sub(boxCenter)
    q := Vector{
        math32.Abs(localPoint.X) - boxDimensions.X/2,
        math32.Abs(localPoint.Y) - boxDimensions.Y/2,
        math32.Abs(localPoint.Z) - boxDimensions.Z/2,
    }

    return math32.Min(math32.Max(q.X, math32.Max(q.Y, q.Z)), 0.0) +
        Vector{math32.Max(q.X, 0), math32.Max(q.Y, 0), math32.Max(q.Z, 0)}.Length()
}

// Torus SDF
func TorusSDF(point, center Vector, majorRadius, minorRadius float32) float32 {
    localPoint := point.Sub(center)
    q := Vector{Vector{localPoint.X, 0, localPoint.Z}.Length() - majorRadius, localPoint.Y, 0}
    return q.Length() - minorRadius
}

// Cylinder SDF
func CylinderSDF(point, center Vector, height, radius float32) float32 {
    localPoint := point.Sub(center)
    d := Vector{Vector{localPoint.X, 0, localPoint.Z}.Length() - radius, math32.Abs(localPoint.Y) - he:
    return math32.Min(math32.Max(d.X, d.Y), 0) +
        Vector{math32.Max(d.X, 0), math32.Max(d.Y, 0)}.Length()
}

```

Image - 94 - 12

11.0 Podpora Post-Processing Shaderov

11.1 Úvod do Post-Processingu

Post-processing shadre sú kľúčovým nástrojom na vizuálne vylepšenie výstupného obrazu v ray-traceri. Umožňujú sofistikované úpravy renderovaného obrazu po jeho primárnom vygenerovaní.

11.2 Jazyk Shaderov Kage

Pôvod a Charakteristika: Kage je originálny shading jazyk vyvinutý pre herný engine Ebitengine. Má syntax podobnú jazyku Go, čo uľahčuje jeho použitie pre vývojárov oboznámených s týmto jazykom. Kage je navrhnutý s dôrazom na prenositeľnosť, pričom umožňuje komplikáciu shaderov na rôznych platformách bez potreby meniť ich kód.

Podporované Typy a Funkcie: Kage podporuje základné typy ako bool, int, float, vektory (vec2, vec3, vec4) a matice (mat2, mat3, mat4). Umožňuje využitie vstavaných funkcií na matematické operácie, manipuláciu s vektormi a textúrami.

Príklad Syntaxe Kage Shaderu:

```
package main

// Edge detection strength
var Strength float
var AlphaR float
var AlphaG float
var AlphaB float
var Alpha float

// Convert RGB to grayscale intensity
func luminance(c vec3) float {
    return (c.r + c.g + c.b) / 3.0
}

func Fragment(position vec4, texCoord vec2, color vec4) vec4 {
    // Define pixel offset based on texture size
    offset := vec2(Strength, Strength)

    // Sample neighboring pixels
    topLeft := imageSrc0At(texCoord + vec2(-offset.x, -offset.y)).rgb
    top := imageSrc0At(texCoord + vec2(0.0, -offset.y)).rgb
    topRight := imageSrc0At(texCoord + vec2(offset.x, -offset.y)).rgb
    left := imageSrc0At(texCoord + vec2(-offset.x, 0.0)).rgb
    right := imageSrc0At(texCoord + vec2(offset.x, 0.0)).rgb
    bottomLeft := imageSrc0At(texCoord + vec2(-offset.x, offset.y)).rgb
    bottom := imageSrc0At(texCoord + vec2(0.0, offset.y)).rgb
    bottomRight := imageSrc0At(texCoord + vec2(offset.x, offset.y)).rgb

    middle := imageSrc0At(texCoord) * Alpha

    tl := luminance(topLeft)
    t := luminance(top)
    tr := luminance(topRight)
    l := luminance(left)
    r := luminance(right)
    bl := luminance(bottomLeft)
    b := luminance(bottom)
    br := luminance(bottomRight)

    // Sobel kernel
    gx := (-1.0 * tl) + (-2.0 * l) + (-1.0 * bl) + (1.0 * tr) + (2.0 * r) + (1.0 * br)
    gy := (-1.0 * tl) + (-2.0 * t) + (-1.0 * tr) + (1.0 * bl) + (2.0 * b) + (1.0 * br)

    // Compute gradient magnitude
    edge := sqrt((gx * gx) + (gy * gy)) * Alpha

    // Output edge as grayscale
    return vec4(middle.r + edge*AlphaR, middle.g + edge*AlphaB, middle.b + edge*AlphaB, middle.a * Alpha)
}
```

Image - 95 - |2

11.3 Podporované Post-Processing Efekty

V tejto sekcii sa zameriam na post-process efekt, ktoré sú aktuálne implementované v systéme. Naša súčasná implementácia poskytuje flexibilitu pri pridávaní nových post-process shaderov s minimálnym zásahom do existujúceho kódu. Tento prístup zabezpečuje jednoduchú rozšíriteľnosť a umožňuje rýchle experimentovanie s rôznymi vizuálnymi efektmi bez nutnosti rozsiahlej prestavby systému.

11.3.1 Základné Efekty

- **Tint** (Farebný nádych):

Aplikácia farebného odtieňa na celý obraz. Tento efekt umožňuje upraviť celkový vizuálny tón renderovaného obrazu, čím sa dosahuje želaný **estetický vzhľad**.

- **Contrast** (Kontrast):

Zvyšovanie alebo znižovanie kontrastu medzi svetlými a tmavými časťami obrazu. Tento efekt umožňuje lepšie zvýraznenie detailov v scéne a zlepšuje vizuálny dojem.

- **Bloom** (Svetelný efekt):

Efekt, ktorý simuluje rozmazanie intenzívneho svetla na okrajoch svetelných zdrojov, čím sa vytvára dojem jasu a rozžiarenia. Tento efekt je často využívaný na zlepšenie realistikosti a vizuálnej dynamiky svetelných scén.

11.3.2 Komplexné Vizuálne Efekty

- **Bloom V2:**

Vylepšená verzia tradičného bloom efektu s lepším rozmazaním a prirodzenejšími svetelnými prechodmi. Pomáha dosiahnuť jemnejší a realistickejší vizuálny efekt svetelných zdrojov.

- **Sharpness** (Ostrosť):

Zvýraznenie okrajov objektov v obraze, čo vedie k ostrejšiemu zobrazeniu detaily. Tento efekt môže byť užitočný pri zvýraznení textúr alebo kontúr v scéne.

- **Color Mapping** (Farebné mapovanie):

Tento efekt limituje počet možných hodnôt farebných kanálov v obraze, čím

vytvára efekt zjednodušenia farebného spektra. Používa sa na dosiahnutie vizuálnych štýlov, kde je potrebné orezanie farebných hodnôt (napríklad v prípade retro alebo pixel art štýlov). Tento proces môže byť použitý aj na korekciu farebného gamutu obrazu pre špecifické farebné odtiene alebo estetické efekty.

- **Chromatic Aberration** (Chromatická aberácia):

Efekt, ktorý simuluje skreslenie svetelných lúčov pri prechode cez šošovku, spôsobujúce rozdelenie farieb na okrajoch objektov. Tento efekt pridáva vizuálny realizmus, často používaný v simulačných a filmových aplikáciách.

- **Edge Detection** (Detekcia hrán):

Tento efekt identifikuje hranice objektov v obraze, čím vytvára vizuálny efekt zvýraznenia kontúr. Môže byť užitočný na analyzovanie geometrie v obraze alebo na aplikovanie špecifických efektov na základné tvary.

- **Lighten** (Zosvetlenie):

Aplikácia svetelného efektu na zjasnenie tmavších oblastí obrazu. Tento efekt je vhodný na dosiahnutie jemného vylepšenia vizuálnej svetlosti v scénach s nízkym osvetlením.

11.4 Výhody Implementácie

Implementácia post-processing shaderov, najmä s využitím grafickej karty, ponúka zásadné výhody v oblasti výkonu. Grafické karty sú optimalizované na paralelný výpočtový proces, čo im umožňuje vykonávať komplexné výpočty oveľa rýchlejšie než CPU. Vďaka tejto schopnosti sú post-processing efekty schopné bežať efektívne aj v reálnom čase, čo výrazne zlepšuje vizuálny dojem bez výrazného spomalenia výkonu.

Záver

Vytvorením vlastného 3D ray-tracingového engine-u sme úspešne spojili teoretické poznatky z oblasti počítačovej grafiky s ich praktickou implementáciou. Hlavným cieľom bolo vybudovať vizualizačný systém, ktorý by nebol iba funkčný, ale zároveň aj dostatočne flexibilný pre ďalší rozvoj. Počas vývoja sme prešli od jednoduchého zobrazovania trojuholníkov, cez optimalizáciu výkonu až po integráciu pokročilých efektov a podpory objemového (voxel) renderingu.

Veľkým prínosom bolo prehĺbenie poznatkov o fungovaní ray-tracingu na nízkej úrovni — od pochopenia samotného princípu sledovania lúčov v priestore, výpočtu ich prienikov so scénou až po simuláciu fyzikálnych vlastností svetla. Implementácia efektívnej štruktúry BVH pre urýchlenie výpočtov nám umožnila zvládať aj zložitejšie scény bez výrazného dopadu na výkon.

Súčasťou projektu bolo aj vytvorenie podpory pre volume rendering, vďaka čomu sa engine stal schopným vizualizovať nielen povrchové modely, ale aj objemové dátá, čo môže byť využiteľné napríklad v simuláciach alebo technickej vizualizácii. K tomu sme pridali aj systém post-processingu, vrátane podpory pre vlastné shaderové efekty, ktoré ešte viac rozšírili vizuálne možnosti engine-u a otvorili dvere k ďalšiemu experimentovaniu.

Za zmienku stojí aj jednoduché frontendové rozhranie, ktoré umožňuje základnú úpravu scény bez nutnosti priamej manipulácie s kódom. Tento krok bol dôležitý najmä z pohľadu rozšíriteľnosti projektu a pohodlia pri testovaní rôznych nastavení.

Počas vývoja sme museli riešiť aj rôzne optimalizačné problémy — od optimalizácie dátových štruktúr až po správu pamäte a využívanie možností paralelizácie. Projekt bol teda výbornou príležitosťou na získanie praktických skúseností nielen s počítačovou grafikou, ale aj so samotným programovacím jazykom Go, s ktorým som pred začiatkom projektu nemal výraznejšie skúsenosti.

Výsledný engine predstavuje solídný základ pre budúce smerovanie. V budúcnosti by bolo možné integrovať pokročilejšie algoritmy na realistickejšie osvetlenie a globálnu ilumináciu, pridať podporu pre real-time ray-tracing alebo rozšíriť systém o animácii a dynamické objekty.

Projekt zároveň ukázal, že aj s relatívne obmedzeným časom a bez použitia hotových grafických knižníc je možné navrhnúť vlastný ray-tracingový systém, ktorý kombinuje výkon, flexibilitu a rozšíriteľnosť.

Počas práce som si nielen osvojil množstvo technických poznatkov, ale tiež získal cenné skúsenosti s navrhovaním komplexnejších systémov a hľadaním riešení na rôzne technické problémy. Tento projekt pre mňa predstavoval nielen výzvu, ale aj obohacujúcu skúsenosť, ktorá ma posunula bližšie k profesionálnej úrovni v oblasti počítačovej grafiky a optimalizácií.

ZOZNAM BIBLIOGRAFICKÝCH ODKAZOV

[1] Ray Tracing: in One Weekend

<https://raytracing.github.io/books/RayTracingInOneWeekend.html#overview>

[2] Ray Tracing: The Next Week

<https://raytracing.github.io/books/RayTracingTheNextWeek.html>

[3] Ray Tracing: The Rest of Your Life

[https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html#cleaninguppdfmanagement/diffuseversuspecular](https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html#cleaninguppdfmanagement/diffuseversusspecular)

[4] How Big Budget AAA Games Render Bloom

<https://www.youtube.com/watch?v=ml-5OGZC7vE>

[5] why is recursion bad?

https://www.youtube.com/watch?v=mMEmNX6aW_k

[6] Andrew Kelley Practical Data Oriented Design (DoD)

<https://www.youtube.com/watch?v=IroPQ150F6c>