Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Vizualizace 3D scény pomocí metody Ray marching

Bakalářská práce

Autor: Daniel Paleček
Studijní obor: Aplikovaná Informatika

Vedoucí práce: Ing. Jakub Beneš

Hradec Králové Duben 2024

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 22.4.2024 Daniel Paleček

Poděkování:

Děkuji vedoucímu bakalářské práce Ing. Jakubovi Benešovi za metodické vedení práce, ochotu, odbornou pomoc a věcné připomínky.

Abstrakt

Bakalářská práce se zabývá problematikou ray marchingu a jeho využití k vykreslení 3D scény. V práci jsou popisovány a testovány algoritmy a způsoby implementace ray marchingu a algoritmy s ray marchingem související. Kromě vykreslení scény se bakalářská práce zabývá i osvětlením, optimalizací a dalšími metodami vylepšení výsledné scény. V teoretické části jsou vysvětleny principy a pojmy související s oblastí ray marchingu. Jednotlivé algoritmy jsou zde také porovnávány s tradičními metodami vykreslení 3D scény. Praktická část bakalářské práce se následně zabývá samotnou implementací zmíněných algoritmů a vytvořením a otestováním funkční aplikace. Účelem vytvořené aplikace je demonstrovat výhody a vlastnosti ray marchingu jako vykreslovací metody. K vytvoření aplikace bylo využito grafické API OpenGL.

Klíčová slova: Ray marching, OpenGL , 3D grafika, Osvětlení 3D scény

Abstract

Title: 3D scene visualization using the Ray marching method

The bachelor thesis explores the ray marching algorithm and its use for rendering 3D scenes. The thesis describes and tests algorithms and methods of ray marching implementation and algorithms related to ray marching. In addition to rendering the scene, the thesis also deals with shading, optimization and other methods of visually improving the resulting scene. The theoretical part explains the principles and concepts related to the field of ray marching. The individual algorithms are also compared with traditional methods of rendering a 3D scene. The practical part of this bachelor thesis deals with the actual implementation of the mentioned algorithms and the creation and testing of a functional application. The purpose of the developed application is to demonstrate the advantages and features of ray marching as a rendering method. The application was created using the OpenGL graphics API.

Key words: Ray marching, OpenGL, 3D graphics, 3D scene shading

Obsah

1 Úvod ... 1

2 Cíl a metodika práce ... 2

3 Teoretická část ... 3

3.1 Princip ... 3

3.1.1 Vrhání/sledování paprsku ... 3

3.1.2 Implicitní povrchy ... 5

3.1.3 Signed distance function (SDF) ... 6

3.1.4 Kombinování SDF ... 8

3.1.5 Sphere tracing ... 9

3.1.6 Transformace ... 9

3.1.7 Odraz a lom ... 11

3.1.8 Fraktály ... 13

3.2 Kamera ... 13

3.3 Osvětlení a stínování ... 14

3.3.1 Zdroj světla ... 15

3.3.2 BRDF ... 15

3.3.3 Odraz světla ... 16

3.3.4 Metody stínování ... 16

3.3.5 Složky světla Phongova modelu ... 17

3.3.6 Ambient occlusion ... 18

3.3.7 Soft shadows ... 19

3.4 Optimalizace ... 20

3.4.1 Bounding sphere ... 20

3.4.2 Reintroducing Analytic Roots ... 20

3.4.3 Over-relaxation ... 21

3.5 OpenGL ... 21

3.5.1 Reprezentace objektů ... 21

3.5.2 Transformace 22
3.5.3 Pipeline 23
3.5.4 Shadery 24
4 Praktická část 27
4.1 Příprava 27
4.1.1 Renderování pomocí dvou trojúhelníků 27
4.2 Implementace ray marchingu 27
4.2.1 UV souřadnice 27
4.2.2 Počátek a směr paprsku 28
4.2.3 Sphere tracing algoritmus 28
4.2.4 Reprezentace scény 29
4.2.5 Transformace 30
4.2.6 Kombinace SDF 31
4.2.7 Textury 31
4.3 Osvětlení scény 32
4.3.1 Výpočet normály 32
4.3.2 Implementace osvětlení 32
4.3.3 Implementace vržených stínů 34
4.3.4 Více zdrojů světla 35
4.3.5 Implementace ambient occlusion 35
4.4 Implementace odrazu a lomu 35
4.4.1 Odraz 36
4.4.2 Lom 37
4.5 Fraktály 39
4.6 Finální Aplikace 40
4.6.1 Přehled scén 40
4.7 Testování 41
5 Shrnutí a diskuse výsledků 44

6 Závěry a doporučení 45
7 Seznam použité literatury 46
8 Přílohy 49

1 Úvod

Počítačová grafika je velice dynamicky vyvíjejícím se podoborem počítačové vědy. Rychlý vývoj technologií a velká poptávka z množství různých odvětví a oborů umožnily mnoho nových způsobů, jak vytvářet různé obrazové výstupy, které by byly dříve příliš náročné na výpočetní výkon. 3D počítačová grafika je podoblastí počítačové grafiky, která se za posledních pár let až dekád změnila asi nejvíce. Fotorealistické scény renderované v reálném čase na běžných počítačích by byly ještě před pár lety nedosažitelné. K tomuto přispěl především vývoj grafických karet s podporou ray tracingu. Aby se touto metodou vykreslila scéna, je nutné pro každý pixel obrazovky vytvořit paprsek, který protíná scénu. Ray tracing algoritmus pak vypočítává trajektorii a odrazy paprsku ve scéně. Tento přístup umožňuje snadno vypočítat odrazy světel, zrcadlových povrchů, ale i dalších efektů, které výslednou scénu vizuálně vylepší.

Podobně jako ray tracing, tak i ray marching je takzvanou ray-based rendering metodou. Ray marching najde své uplatnění například při vykreslování objektů s nejednotvárným objemem (mraky, kouř), fraktálů nebo procedurálně generovaných objektů. Ray marching může být implementován více způsoby, tato bakalářská práce se však primárně zaměří na implementaci pomocí algoritmu sphere tracing.

2 Cíl a metodika práce

Cílem bakalářské práce je prozkoumat a popsat metodu ray marching jako techniku vizualizace 3D scén. Práce se zaměří na návrh, implementaci a optimalizaci ray marching algoritmů pro generování vizuálních reprezentací 3D scén s důrazem na geometrické struktury a osvětlení. Dalším cílem je porovnat uvedenou metodu s tradičními technikami renderování 3D scén a identifikovat její výhody a omezení. Výstupem práce bude implementace vizualizace 3D scény pomocí techniky ray marching.

Teoretická část bakalářské práce se bude zabývat algoritmy a metodami implementace ray marchingu a porovnávat je s implementací tradičními metodami rasterizace. Teoretická část, kromě samotného ray marching algoritmu, dále rozebírá i implementaci kamery, osvětlení, optimalizace a využití OpenGL. Praktická část následně otestuje algoritmy a metody zmíněné v teoretické části a také zdokumentuje tvorbu výsledné aplikace.

3 Teoretická část

Teoretická část je zaměřená na shrnutí teoretických poznatků v oblasti ray marchingu a souvisejících témat. Dále zde bude vysvětlen princip fungování ray marchingu a dalších algoritmů, které se dají při renderování scény s ray marchingem použít.

3.1 Princip

Na rozdíl od ostatních renderovacích metod ray marching, ale i jiné ray-based algoritmy, využívá k vykreslení 3D scény vrhání nebo sledování paprsků. V případě ray marchingu následně algoritmus „pochoduje“ v krocích k implicitně zadanému povrchu tělesa [1].

3.1.1 Vrhání/sledování paprsku

Metoda vrhání/sledování paprsku má své kořeny v optice. Umožňuje simulovat různé optické jevy vzniklé odrazem a lomem světelných paprsků ve scéně [2]. Na rozdíl od reálného světa emitorem paprsků většinou není světelný zdroj, ale paprsky jsou vrhány z kamery. Pro každý pixel obrazovky je vržen jeden nebo více paprsků [1, 2].

Jelikož je algoritmus nutné provést pro každý pixel, je renderování při vyšších rozlišeních výpočetně náročné. Dalším faktorem, který ovlivňuje výpočetní náročnost, je počet odvozených paprsků neboli paprsků vytvořených v bodech, kde se předešlý paprsek protnul s tělesem. Tyto paprsky reprezentují odraz nebo lom světla na povrchu tělesa [1, 2]. Algoritmy se následně dělí na dva typy.

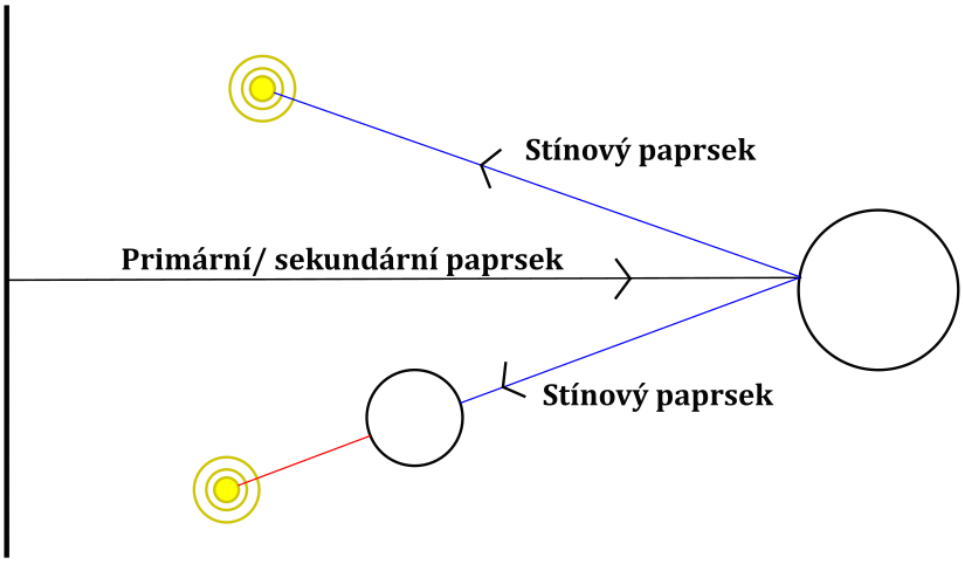
1. **Vrhání paprsku neboli sledování paprsku prvního řádu**

Jedná se o jednodušší přístup, kdy se paprsek sleduje pouze do prvního protnutí se scénou. V bodě dotyku se vypočítá barva tělesa, popřípadě je možné využít některý z jednoduchých osvětlovacích modelů. [2]

2. **Sledování paprsku vyššího řádu**

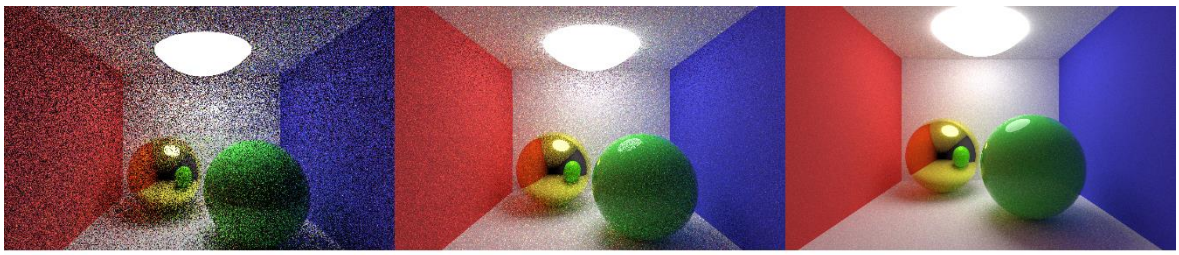
Sledování paprsku nekončí po nalezení prvního bodu dotyku se scénou (primary ray), ale pokračuje dále formou odvozených paprsků (secondary ray). Tyto paprsky jsou vytvořeny na základě materiálu tělesa. Materiál může paprsky odrážet, lámat či pohlcovat. Z výpočetních důvodů je vhodné algoritmus omezit maximální hloubkou. [2]

Osvětlení scény je vypočítáno pomocí vrhání takzvaných stínových paprsků [2, 3 s. 11]. Stínové paprsky jsou vytvořeny z místa dopadu primárních a sekundárních paprsků a míří směrem ke každému světelnému zdroji [2, 3 s. 11, 4]. Intenzita světla je následně vypočtena ze vzdálenosti od zdroje, ale i z přičtení dalších složek, jako jsou složky ambientní a difuzní. Pokud dráha stínového paprsku prochází tělesem viz. Obrázek 1, je přínos tohoto světelného zdroje ignorován, čímž vznikají vržené stíny [2].



Obrázek 1 Vrhání stínových paprsků. Zdroj autor

Zmíněná metoda vykreslování scény umožňuje vytvořit realistické osvětlení scény s vrženými stíny, zrcadlovými povrchy, průhlednými či částečně průhlednými tělesy a řeší další problémy, které by byly při běžné rasterizaci náročné nebo i nemožné. Sledování paprsku má svá omezení, mezi něž patří například měkké stíny nebo lesklé a difúzní povrchy [1]. Pro tyto případy je vhodné využít metody jako path tracing (sledování cesty). Path tracing sleduje paprsek a jeho odrazy ve scéně, dokud nedorazí ke zdroji světla[2]. Jedná se o kombinaci ray tracingu a Monte Carlo simulací [1, 2]. Jelikož je algoritmus velice výpočetně náročný z důvodu nutnosti vrhnout velké množství paprsků, je nutné omezit počet vržených paprsků, což vede k šumu [1, 2], viz. Obrázek 2.



1 paprsek na pixel 10 paprsků na pixel 100 paprsků na pixel

Obrázek 2 Path tracing. Zdroj: Autor s použitím Shadertoy smallerpt by ockiller

3.1.2 Implicitní povrchy

Implicitní povrchy se od explicitních nebo parametrických liší tím, že nejsou zadány pomocí na sobě závislých proměnných ani pomocí parametrů jako bodů v prostoru, ale jsou zadány pomocí funkce více proměnných (1) [1, 2].

$$f(x, y, z) = 0 \qquad (1)$$

Takovýto povrch pak tvoří body, pro které je daná rovnice rovna nule. Bod, jenž leží na povrchu a zároveň na paprsku, který algoritmus sleduje, je průsečíkem. Ray marching algoritmus označí za průsečík bod, který se nachází v okolí ε. Čím menší ε, tím větší kvalita výstupu, ale i vyšší výpočetní náročnost.

Pro výpočet osvětlení, odrazů a dalších informací o scéně je nutné vypočítat normálu. Pro popis normály je možné použít funkci $\nabla f(x, y, z) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}\right)$. Pro výpočet ovšem musí být funkce diferencovatelná, tudíž spojitá [1]. Mnohem častěji se tedy přistupuje k aproximaci normály.

$$\begin{aligned} \nabla fx &\approx f(p + \varepsilon e_x) - f(p - \varepsilon e_x) \\ \nabla fy &\approx f(p + \varepsilon e_y) - f(p - \varepsilon e_y) \\ \nabla fz &\approx f(p + \varepsilon e_z) - f(p - \varepsilon e_z) \end{aligned} \qquad (2)$$

Vzorce (2) jsou použity k výpočtu aproximace normály. Výpočet každé souřadnice je popsán samostatně. Průsečík neboli místo, kde byl povrch zasažen, je ve vzorci značen písmenem p a písmeno ε značí malou hodnotu, o kterou se bod posune. Posun je v jednom ze tří směrů značených e_x pro vektor (1,0,0), e_y pro vektor (0,1,0) a e_z pro vektor (0,0,1).

Na implicitně zadané povrchy je možné aplikovat Booleovské operátory (AND, OR, NOT). Použitím těchto operátorů je možné udělat sjednocení, rozdíl a průnik. Dále je možné objekty deformovat nebo smíchat. [1, 2]

Pomocí implicitních funkcí je možné reprezentovat různé tvary. Nejjednodušší funkcí je funkce pro kouli $f(p, r) = \|p\| - r$. Jedná se o vzdálenost bodu p, od něhož je odečten poloměr koule [1].

3.1.3 Signed distance function (SDF)

Signed distance function je funkce, která vrací vzdálenost bodu od daného tělesa. Jak již bylo zmíněno, pokud funkce vrátí 0, bod se nachází na povrchu. Pokud se bod nachází mimo těleso, vrátí funkce pozitivní hodnotu. V případě, že je bod v tělesu, funkce vrátí negativní hodnotu [1]. Signed distance functions zkráceně SDFs jsou hlavním a nejpoužívanějším způsobem, jak reprezentovat objekty scény vykreslované pomocí ray marchingu.

V konferenčním příspěvku *PRIF: Primary Ray-Based Implicit Function* byl představen nový způsob reprezentace implicitních povrchů nazvaný PRIF. Na rozdíl od SDF, která vrací vzdálenost bodu od povrchu, PRIF vrací přímo průsečík vstupního paprsku s tělesem [5]. Tato technika ale není v této bakalářské práci implementována a je pouze zmíněna jako možná alternativa.

Signed distance functions mohou mít různé vstupní parametry (viz. podkapitola 3.1.3.1), avšak vždy vrací reálné číslo reprezentující vzdálenost daného bodu od povrchu jimi definovaného tělesa.

Na SDF se dají aplikovat operátory, jako je například modulo, které zapříčiní nekonečné opakování objektu ve scéně [6]. Dále lze použít boolean operátory, ty umožňují mísit různé SDF dohromady.

3.1.3.1 Příklady SDF

Koule:

```
float sdSphere(vec3 point, float size, vec3 pos)
{
    return length(point + pos) - size;
}
```

Výpis z kódu 1 Koule. Zdroj: [6] upraveno autorem

Ve Výpis z kódu 1 je vidět funkce určující vzdálenost bodu od povrchu koule. Funkce má jako parametry vstupní bod, velikost a pozici koule.

Kvádr:

```
float sdBox(vec3 point, vec3 size, vec3 pos)
{
    vec3 d = abs(point + pos) - size;
    return min(max(d.x, max(d.y, d.z)), 0.0)
    + length(max(d, 0.0));
}
```

Výpis z kódu 2 Kvádr. Zdroj: [6] upraveno autorem

Další funkce slouží k určení vzdálenosti bodu od povrchu kvádru. Funkce má jako parametry vstupní bod, velikost a pozici kvádru.

Rovina:

```
float sdPlane(vec3 point, vec4 n)
{
    //n musí být normalizované
    return dot(point, n.xyz) + n.w;
}
```

Výpis z kódu 3 Rovina. Zdroj: [6] upraveno autorem

V případě funkce pro rovinu je nutné do funkce kromě vstupního bodu dodat i normalizovaný vektor normály roviny.

Torus:

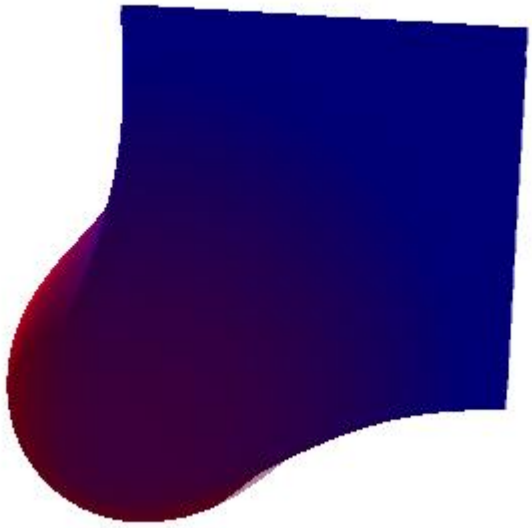
```
float sdTorus(vec3 point, vec2 t, vec3 pos)
{
    vec3 p = point + pos;
    vec2 q = vec2(length(p.xy)-t.x,p.z);
    return length(q)-t.y;
}
```

Výpis z kódu 4 Torus. Zdroj: [6] upraveno autorem

Poslední ukázka kódu je funkce vracející vzdálenost vstupního bodu od povrchu torusu. Kromě vstupního bodu do funkce vstupuje i poloměr (vnější a vnitřní, proto vec2) a pozice torusu.

3.1.4 Kombinování SDF

Kombinování SDF je možné docílit pomocí boolean operátorů. V kombinaci s blending function, neboli funkcí mísení nebo smíchání, je možné prolnout objekty do sebe, viz. Obrázek 3. V případě SDF se operace aplikují na vzdálenosti vrácené z SDF [1, 6].



Obrázek 3 Smíchání kvádru a koule. Zdroj: autor

Příklady operátorů [6]:

Sjednocení: Funkce (3) vrací vzdálenost bližšího ze dvou objektů, takže dochází k vykreslení obou.

$$min(d1, d2) \quad (3)$$

Průnik: Funkce (4) vrací vzdálenost toho objektu, který se nachází dále od bodu, pro který je vzdálenost počítána, takže se vykreslí pouze části objektů, které se protínají.

$$max(d1, d2) \quad (4)$$

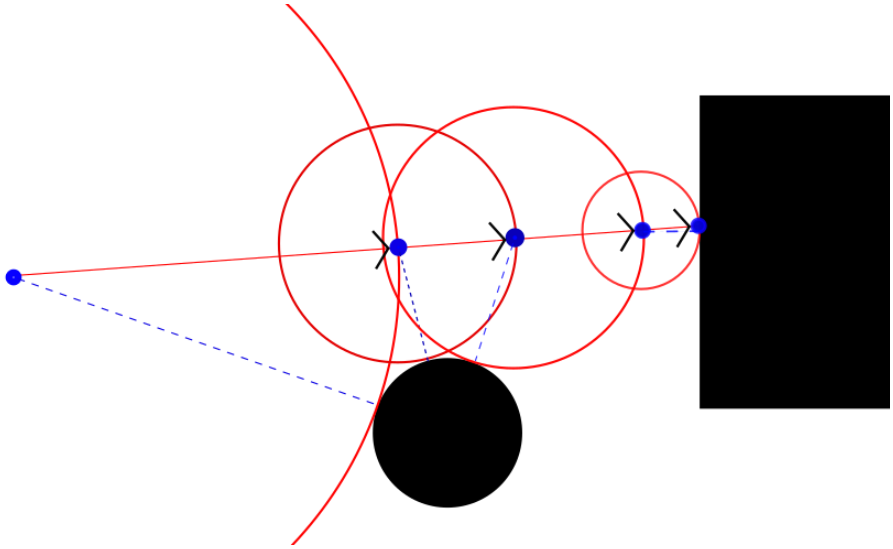
Rozdíl: Funkce (5) funguje stejně jako průnik, ale jeden objekt je invertovaný, čímž dojde k oříznutí jednoho objektu druhým.

$$max(-d1, d2) \quad (5)$$

Do těchto funkcí je možné vložit parametr hladkosti, který v upravené formě těchto funkcí určuje hladkost přechodu mezi tělesy [6]. S využitím parametru hladkosti byl vytvořen obrázek výše, viz. Obrázek 3.

3.1.5 Sphere tracing

Sphere tracing je jeden z ray marching algoritmů. Sphere tracing využívá funkcí, jako jsou již dříve zmíněné signed distance functions, k nalezení průsečíku [7]. Algoritmus postupuje po paprsku (v obrázku 4 znázorněno plnou čarou) po krocích, kdy velikost kroku je určena vzdáleností současné pozice od nejbližšího tělesa ve scéně (v obrázku 4 znázorněno čárkovanou čarou), viz. Obrázek 4.



Obrázek 4 Sphere Tracing. Zdroj: autor

Absolutní hodnota signed distance function tvoří poloměr koule (v obrázku 4 znázorněno kružnicí), která neprotíná žádný implicitní povrch. Vzniklá koule se nazývá unbounding sphere. Název pochází z bounding volume, což je objem, který obsahuje dané těleso. Tudíž unbounding volume (v tomto případě sphere) je objem, který zaručeně neobsahuje těleso. [7]

Vzhledem k výpočetní náročnosti je vhodné algoritmus omezit maximálním počtem krků a zavést minimální vzdálenost, která bude vyhodnocena jako dotyk. Toto omezení vyřeší problém s paprsky, které jsou těsně u tělesa a jsou paralelní s jeho povrchem. Bez omezení by algoritmus krokoval paprsek podél tělesa v příliš malých krocích, čímž by výpočet výrazně zpomalil. Optimalizací tohoto a jiných problémů se bude zabývat kapitola 3.4.

3.1.6 Transformace

Aby bylo možné upravovat tělesa a scénu, je nutné definovat transformace. Při používání transformací záleží na pořadí. Mezi základní transformace patří posunutí (translation), otočení (rotation) a škálování (scale). Jedním ze způsobů, jak tyto

transformace aplikovat, je pomocí násobení bodu p (současná pozice na cestě paprsku) maticí transformace. [6]

Translace
$$\begin{bmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (6)$$

Rotace podle X
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(t) & sin(t) & 0 \\ 0 & -sin(t) & cos(t) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (7)$$

Rotace podle Y
$$\begin{bmatrix} cos\,(t) & 0 & -sin\,(t) & 0 \\ 0 & 1 & 0 & 0 \\ sin\,(t) & 0 & cos\,(t) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (8)$$

Rotace podle Z
$$\begin{bmatrix} cos\,(t) & sin\,(t) & 0 & 0 \\ -sin\,(t) & cos\,(t) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (9)$$

Škálování
$$\begin{bmatrix} 1/x & 0 & 0 & 0 \\ 0 & 1/y & 0 & 0 \\ 0 & 0 & 1/z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (10)$$

Maticový přístup k transformacím umožňuje mnohem více než jen výše uvedené transformace. Bod p je vektor, který může být transformován jakoukoli funkcí, která na jeho místo vrátí nový vektor. Příkladem užitečných transformací je například symetrie [6]:

$$p.x = abs(p.x) \quad (11)$$

V tomto případě funkce (11) se jedná o symetrii v ose x. Část objektu, která se nachází v pozitivní části osy x, se symetricky převrátí do negativní části osy x. Další transformace, která stojí za to zmínit, je doménové opakování. Základní myšlenkou je vytvořit funkci, která se bude periodicky opakovat [6].

$$p.x = p.x - round(p.x) \quad (12)$$

Je to jednoduchý způsob, jak docílit nekonečného opakování tělesa na ose x. Obrovskou výhodu těchto transformací je jejich nízká výpočetní náročnost. To umožňuje renderovat scény s nekonečně se opakujícími objekty do nekonečna v reálném čase, což je užitečné například při renderování fraktálů, viz. kapitola 3.1.8.

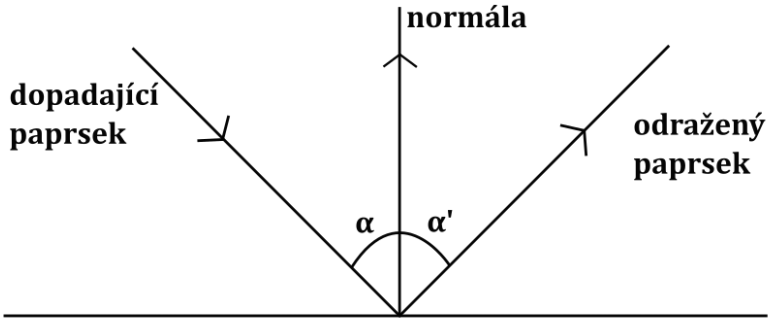
3.1.7 Odraz a lom

Jelikož reálné materiály mají mnoho různých vlastností, následující podkapitola se bude zabývat simulováním reflexních a refrakčních materiálů. Ray marching nabízí poměrně jednoduchý způsob, jak tohoto efektu dosáhnout. V principu dochází k výpočtu nového paprsku, který má počátek v bodě, kde se předchozí paprsek protnul s objektem ve scéně a směřuje ve směru odrazu či lomu.

3.1.7.1 Odraz

Odraz je jednodušší a výpočetně méně náročný případ. Pro výpočet odrazu existuje zákon odrazu [8]. Podle zákonu odrazu platí, že úhel odrazu je roven úhlu dopadu paprsku. Pro výpočet vektoru paprsku lze použít následující vzorec (13), kde vektorOP je vektor odraženého paprsku, vektorDP je vektor dopadajícího paprsku a normála je vektor normály povrchu, na který paprsek dopadá.

$$vektorOP = vektorDP - 2 * (vektorDP \cdot normála) * normála \qquad (13)$$



Obrázek 5 Odraz paprsku. Zdroj: autor

Po výpočtu vektoru odraženého paprsku dochází k zahájení nového cyklu ray marching algoritmu pro získaný paprsek s počátkem v bodě dotyku [9]. Po nalezení dalšího průsečíku je nutné rozhodnout, zda se bude proces opakovat. To závisí jak na materiálu nově zasaženého povrchu, tak na limitaci samotné implementace (maximální počet odrazů). Při implementaci je počet odrazů nutné omezit, aby se předešlo do nekonečna se odrážejícím paprskům a nedošlo k příliš velké výpočetní náročnosti.

3.1.7.2 Lom

K lomu světla dochází při přechodu z jednoho prostředí do druhého s odlišnými optickými vlastnostmi. K výpočtu lomu slouží Snellův zákon [8].

$$n_1 sin\alpha = n_2 sin\beta \qquad (14)$$

Pro výpočet Snellova zákonu (14) je potřeba znát index lomu v daném prostředí značený písmenem n. Index se bude v každém prostředí lišit.

Výpočet lomu je výrazně složitější než výpočet odrazu. Vzorec (15) je jeden z možných vzorců pro výpočet vektoru lomu.

$$vektorLP = n * vektorDP + (n * cosI - cosT) * normála \qquad (15)$$
$$cosI = -(normála \cdot vektorDP)$$
$$cosT = \sqrt{1 - (n * n * (1 - cosI * cosI)}$$
$$n = n_1/n_2$$

VektorLP a vektorDP ve vzorci (15) jsou vektory lomeného a dopadajícího paprsku. Normála je vektorem normály zasaženého povrchu a $n_{1|2}$ jsou indexy lomu prostředí, mezi kterými paprsek prochází.

Pro získaný směrový vektor je vytvořen nový paprsek z bodu dotyku a je na něj znovu aplikován ray marching. Tentokrát je však nutné, aby paprsek procházel vnitřkem tělesa. Toho je možné docílit vynásobením vzdáleností získaných z SDF mínus jedničkou, čímž se těleso invertuje. Po nalezení nového průsečíku je nutné vytvořit nový lomený paprsek pro přechod z optického prostředí tělesa ven. Pro nový paprsek je následně znovu zahájen ray marching. [9 s. 45]

S lomem světla je spojeno množství dalších jevů. Jedním z nich je total internal reflection (TIR). K TIR dochází, když je paprsek na rozhraní dvou prostředí, přičemž druhé prostředí má nižší hodnotu indexu lomu a paprsek dopadá pod šikmým úhlem. Paprsek je následně odražen namísto toho, aby byl lámán [10 s. 47]. Při implementaci stačí ověřit, zda je velikost směrového vektoru paprsku nulová. V takovém případě se místo lomu použije odraz.

Další jev, který je možné do simulace zahrnout, je situace, kdy jsou paprsky odraženy i lomeny zároveň. Jev je popsán Fresnelovými rovnicemi [10]. Pro zjednodušení výpočtu se používá Schlickova aproximace vypočítaná vzorcem (16) [11].

$$R(\theta) = R_0 + (1 - R_0)(1 - cos\theta)^5 \qquad (16)$$
$$R_0 = (\frac{n_1 - n_2}{n_1 + n_2})^5$$

Θ - úhel mezi paprskem dopadu a normálou povrchu

n - indexy lomu

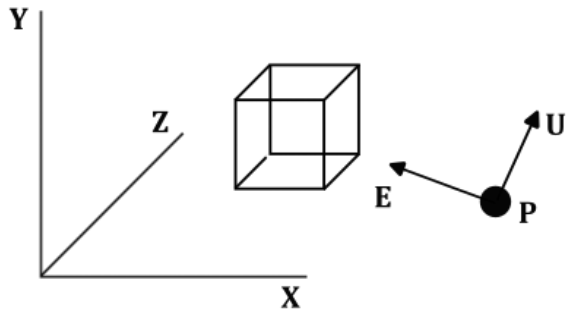
Poté se výsledek lomu a odrazu interpoluje. Nutné podotknout, že tyto výpočty jsou výpočetně náročnější, protože s každým lomem a odrazem je nutné spustit ray marching algoritmus i vícekrát.

3.1.8 Fraktály

Způsob, jakým ray marching funguje, z něj dělá ideální renderovací metodu pro vykreslování 3D fraktálů. Jedinou podmínkou je, že fraktál musí být vyjádřitelný funkcí. Fraktály jsou nekonečně komplexní geometrické tvary, které opakují nějaký vzor do nekonečna. Jako koncept je představil Benoit Mandelbrot [12]. Od té doby vzniklo mnoho řešení jejich vizualizace z hlediska počítačové grafiky. Jedno z řešení je právě použití techniky ray marchingu.

3.2 Kamera

Jednou z nejdůležitějších věcí, kterou je třeba při renderování 3D scény definovat, je kamera. Kamera slouží ke stanovení pozice, orientace a směru pohledu. Pozice určuje, kde ve scéně se pozorovatel nachází, v této práci bude značena jako P. Orientaci kamery reprezentuje upvektor, to je vektor, který určuje směr, který je pro pozorovatele vždy nahoru. V této bakalářské práci bude značen jako U. Nakonec směr pohledu je určený vektorem, který je zde označen jako E.

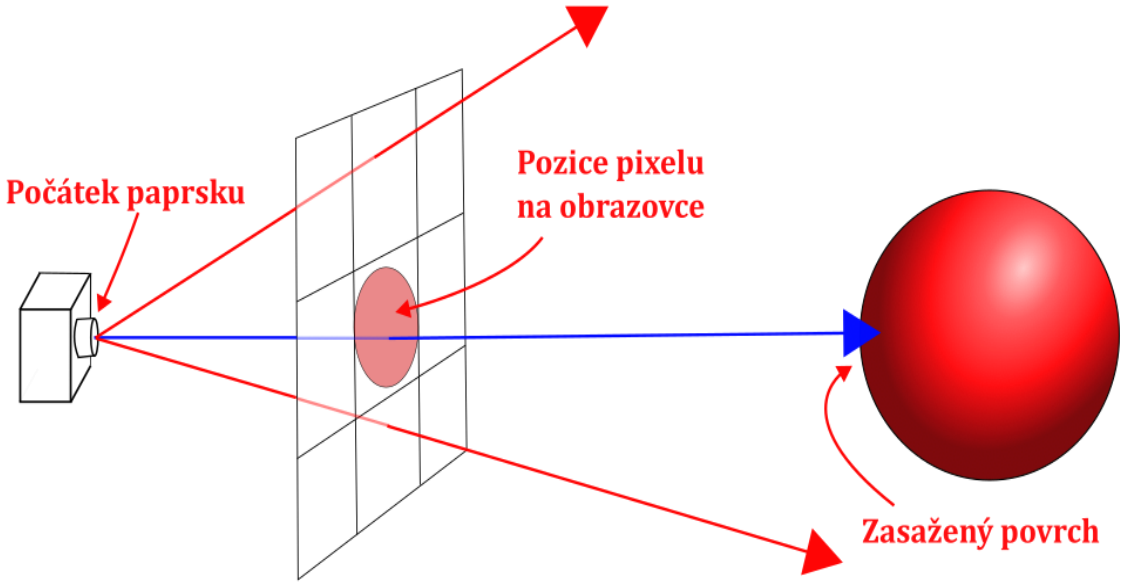


Obrázek 6 Kamera. Zdroj: autor

V tradičních renderovacích technikách kamera často určovala způsob projekce. Projekce je nejčastěji definována projekční maticí. Existuje více projekcí, nejvíce se však používají rovnoběžné (ortogonální) a středové (perspektivní). Rovnoběžné promítání je nejjednodušším typem promítání [2 s. 309]. Výsledné vyobrazení scény, ale špatně vykresluje hloubku scény, jelikož vzdálenost objektu od kamery nemá vliv na velikost vykresleného objektu na obrazovce. Je vhodné na tvorbu bokorysů, půdorysů a dalších specifických příkladů pohledu na scénu [2 s. 309]. Na druhé straně

středové promítání odpovídá lidskému vidění. Objekty se se vzdáleností od kamery proporcionálně zmenšují, takže perspektivní promítání nezachovává rovnoběžnost úseček [2 s. 312].

V případě ray marchingu a dalších podobných metod je však přístup jiný. Tyto metody vykreslují scénu tím, že pro každý pixel obrazovky vytvoří paprsek, pomocí kterého vypočítají barvu daného pixelu [3 s. 4]. Tyto paprsky nejsou různoběžné s vektorem pohledu pozorovatele (Všechny paprsky vychází z jednoho bodu, ale každý prochází jinými pixelem obrazovky.) a tím vytváří stejný jev jako perspektivní promítání. Na tyto výpočty je vhodné použít fragment shader, jehož kód se spouští pro každý pixel a běží na GPU.



Obrázek 7 Ray marching kamera. Zdroj: autor

3.3 Osvětlení a stínování

Osvětlení je důležitou součástí každé 3D scény. Osvětlení a stíny pomáhají pozorovateli vnímat hloubku scény a zároveň dělají scénu více realistickou. Existuje mnoho algoritmů, které vypočítávají osvětlení scény. Některé jsou jednodušší a méně náročné. A jiné počítají složitější optické jevy a jejich výsledek má vyšší kvalitu, ale mají větší výpočetní náročnost.

Aby se výpočet simulace zjednodušil a zrychlil, počítají osvětlovací algoritmy s předpoklady [2]:

1. Dráhou světla je přímka.
2. Světlo se šíří neomezenou rychlostí. Scéna je osvětlená okamžitě.
3. Světlo není ovlivněno vnějšími silami.

3.3.1 Zdroj světla

Zdroj světla je možné reprezentovat různými způsoby. Zdroje mohou mít svou intenzitu, tvar, barvu a velikost [1]. Podle jejich vlastností se světelné zdroje dělí na různé typy [2, 13]:

- Bodové světelné zdroje:
    - Světlo se z jednoho bodu šíří do všech směrů se stejnou intenzitou.
- Směrové světelné zdroje:
    - Světlo pokrývá scénu rovnoměrně a paprsky vychází z jednoho směru a jsou navzájem rovnoběžné.
    - Často využívané jako světlo ze slunce
- Reflektorové světelné zdroje
    - Světlo vychází z jednoho bodu, ale směr je omezen do tvaru kuželu.
- Plošné zdroje světla
    - Světlo vychází z konečné plochy.
    - Vytváří měkké stíny.

3.3.2 BRDF

BRDF, neboli dvousměnová odrazová distribuční funkce, slouží k určení odrazivosti povrchu a jeho schopností absorbovat světlo [2 s. 325; 13 s. 21]. BRDF funkce je zjednodušením BSSRDF (dvousměnová rozptylová distribuční funkce povrchové odrazivosti), ovšem bakalářská práce se bude zabývat pouze BRDF.

Aby funkce správně reprezentovala chování světla, musí splňovat určité předpoklady [13 s. 22]:

- Helmholtzův princip reciprocity – role energie odražené a dopadající jsou zaměnitelné.

- Zákon zachování energie – množství energie, která dopadla, musí být rovna množství energie, která se odrazí plus energie, která se vstřebá. Energie nemůže vzniknout ani zaniknout.

3.3.3 Odraz světla

BRDF může být reprezentována různě. Často se však používá jednoduchá empirická funkce [2 s. 328]. Výsledkem je aproximace, která je pro většinu modelů dostačující. [13 s. 22].

3.3.3.1 Difuzní odraz

Difuzní odraz je odraz, který nastává, pokud se paprsek odráží od difuzního neboli Lambertovského povrchu. Při tomto typu odrazu dochází k rozptylu do všech směrů rovnoměrně. [2 s. 329]

3.3.3.2 Zrcadlový odraz

Tento typ odrazu vzniká na zrcadlovém povrchu. Paprsky světla se odráží pod souměrným úhlem, pod kterým dopadají [2 s. 330].

3.3.4 Metody stínování

Výpočet osvětlení je možné řešit různými způsoby, jejich výsledky se však mohou lišit. Sekce se bude některými z těchto metod v krátkosti zabývat.

3.3.4.1 Konstantní stínování

Konstantní stínování je velice jednoduchá a rychlá metoda [2], ale její výsledky jsou v porovnání s ostatními méně kvalitní. Tento algoritmus vypočítává osvětlení pro celou plochu podle normály [2]. Popisovaná metoda není vhodná pro složitější tělesa složená z mnoha polygonů, protože tyto plochy zvýrazňuje, ale u jednoduchých těles je dostačující. Pokud není realističnost cílem, mohou některé styly tvorby 3D objektů jako například „low poly“, tuto techniku využívat úmyslně.

3.3.4.2 Gouraudovo stínování

Na rozdíl od předchozí metody je Gouraudova metoda vhodná pro stínování povrchů tvořených velkým množstvím ploch. K určení barvy při výpočtu osvětlení je použita bilineární interpolace barev jednotlivých vrcholů. Tato metoda počítá pouze s difuzní a ambientní složkou, ty jsou více rozebrané v podkapitole 3.3.5. [2 s. 340]

3.3.4.3 Phongovo stínování

Metoda navržená Bui-Tuong Phongem vypočítává osvětlení pomocí normálových vektorů uložených ve vrcholech polygonů. Následně dochází k bilineární interpolaci nejen barvy, ale i samotných normálových vektorů. Pomocí normálového vektoru je možné dopočítat zrcadlovou složku osvětlení. Oproti Gouraudovu stínování nabízí Phongova metoda lepší výsledky, avšak za cenu vyšší výpočetní náročnosti. [2 s. 341]

3.3.4.4 Blinn-Phongovo stínování

Blinn-Phongova metoda je modifikací Phongova stínování, kterou vytvořil James F. Blinn. Blinnova úprava používá místo výpočtu odraženého paprsku normalizovaný půl úhel mezi vektorem pohledu a vektorem ke zdroji světla. [14 s. 550]

3.3.5 Složky světla Phongova modelu

Světlo je možné rozložit na složky a následně spočítat jejich příspěvky k finálnímu osvětlení. Jedná se o model zjednodušující realitu, avšak dosahuje dostačujících výsledků.

3.3.5.1 Difuzní složka

Jedná se o odraz světla na povrchu nerovného materiálu. Paprsek se odráží do všech stran se stejnou pravděpodobností. Z hlediska kamery není závislý na směru pohledu a je závislý pouze na úhlu dopadu.

$$I_D = I_i * k_D * cos\,\alpha \qquad (17)$$

Difuzní složka se vypočítá jako intenzita zdroje světla krát koeficient difúzního odrazu světla krát kosinus úhlu dopadu. Výpočet (17) má smysl pouze pro úhly větší než nula, v opačném případě je povrch odvrácen od zdroje světla [2].

3.3.5.2 Zrcadlová složka

Jedná se o odraz světla na povrchu reflektivního materiálu. Při výpočtu záleží na úhlu dopadu i směru k pozorovateli. Pro výpočet je vztah (18) zjednodušen, zjednodušení navrhl americký vědec James F. Blinn [2].

$$I_S = I_i * k_S cos^{h_B}\beta \qquad (18)$$

Zrcadlová složka se vypočítá jako intenzita zdroje světla krát koeficient zrcadlového odrazu světla krát kosinus úhlu mezi vektorem dopadu a pohledu umocněný ostrostí zrcadlového odrazu [14 s. 550].

3.3.5.3 Ambientní složka

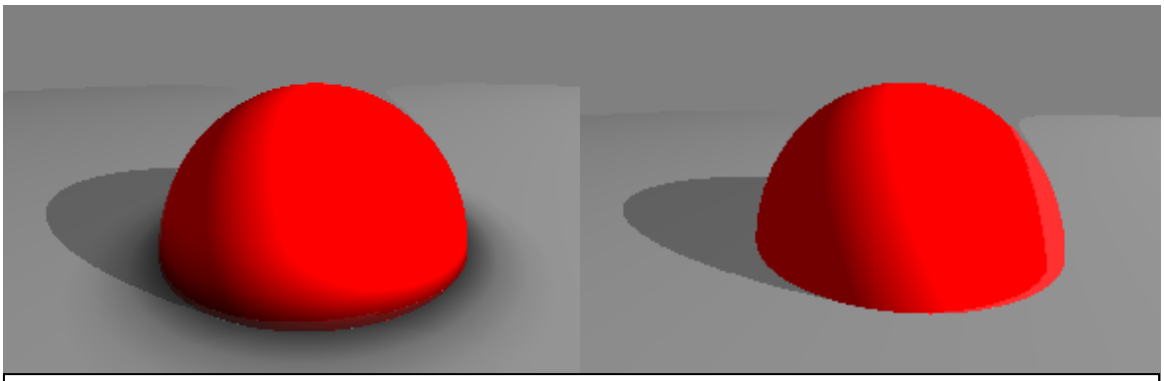
Jedná se o světlo rozptýlené v prostoru ve všech směrech. Je způsobené mnohonásobnými odrazy o tělesa a odrazy světla ve vzduchu [2]. Jedná se o jednoduchou složku, která napomáhá zamezit příliš tmavým stínům.

$$I_A = I * k_A \qquad (19)$$

Ambientní složka se vypočítá jako intenzita okolního světla krát koeficient schopnosti tělesa odrážet okolní světlo. Koeficient se může lišit pro různé kanály RGB.

3.3.6 Ambient occlusion

Vzhledem k tomu, že fyzikálně správná simulace světla v reálném čase je na většině současných počítačů výpočetně příliš náročná, existují techniky, které se soustředí pouze na určité jevy a jejich výpočet zjednodušují. Jednou z těchto technik je ambient occlusion, jedná se o zjednodušení fyzikálních jevů, které působí k zvýraznění tvaru objektu [1 s. 446; 15 s. 1]. Tuto techniku vytvořili Hayden Landis, Ken McGaugh a Hilmar Koch.



Scéna s ambient occlusion (vlevo)	Scéna bez ambient occlusion (vpravo)

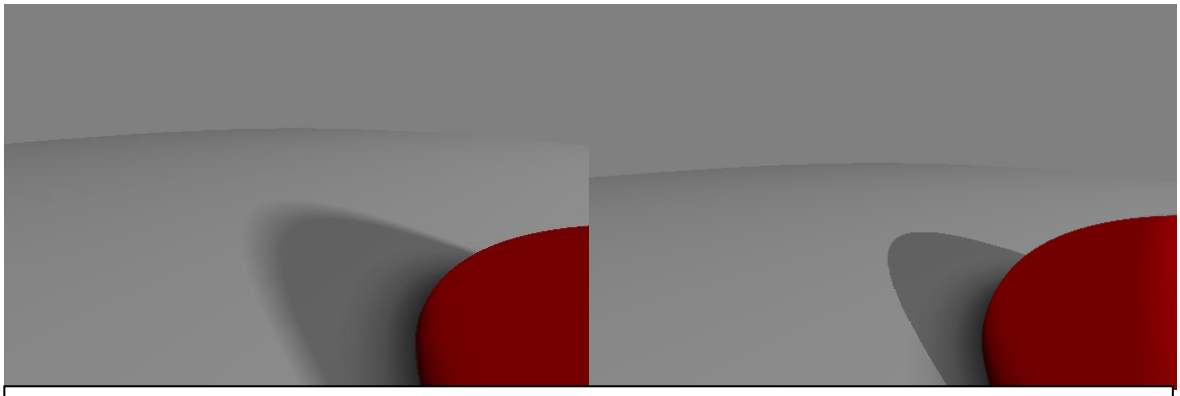
Obrázek 8 Ambient occlusion. Zdroj: autor

Ambient occlusion pomáhá vytvářet měkké stíny v místech, která nejsou plně vystavena ambientnímu světlu v okolním prostředí [15 s. 94]. Stíny se vypočítávají pomocí příspěvků zastínění od okolních objektů. Příspěvkům je přidána váha, kdy největší váhu mají objekty paralelní s normálou stínovaného povrchu. Objekt, který je

stínovaný, se nazývá receiver a objekty, které ke stínování přispívají, se nazývají emittery [16 s. 225].

3.3.7 Soft shadows

Opravdové stíny mají jen zřídka ostré hrany (hard shadows) a většinou přechází plynule do ztracena. Těmto stínům se říká měkké stíny (soft shadows). Podobně jako u ambient occlusion je i u vržených stínů možné využít různých efektů, které umožní vylepšit vizuální stránku scény za zlomek ceny kompletně fyzikálně správné simulace světla.



Scéna s měkkým stínem (vlevo)	Scéna s tvrdým stínem (vpravo)

Obrázek 9 Měkký a tvrdý stín. Zdroj: autor

Stejně jako při výpočtu tvrdých stínů se využívá ray marching algoritmus k nalezení světelného zdroje. Pokud po cestě paprsek protne jiný objekt ve scéně, je stínované místo vyhodnoceno jako plný stín. Po tento krok se od algoritmu výpočtu tvrdých stínů nic neliší. Při řešení měkkých stínů je následně každému pixelu, jehož stínový paprsek není přerušen jiným objektem, přiřazen koeficient, který je určen nejmenší vzdáleností stínového paprsku ke scéně. [17]

$$koeficient_i = min\left(koeficient_{i-1}, k * \frac{h}{t}\right) \qquad (20)$$

Vzorec (20) slouží k výpočtu koeficientu v dané iteraci. Jedná se o minimum mezi koeficientem předchozí iterace a nově vypočtené hodnoty. Ta se vypočítá vynásobením konstanty měkkosti stínu (k) podílem uražené vzdálenosti po paprsku (t) a vzdálenosti od nejbližšího povrchu z nového bodu (h).

Pomocí tohoto koeficientu je následně dopočítána intenzita stínu pro daný pixel. A pixel je poté adekvátně obarven.

3.4 Optimalizace

Renderování rozsáhlých scén čítajících velké množství objektů může být výpočetně velice náročné. Pokud je cílem vytvořit obrázek nebo animaci, nemusí být nároky na rychlost výpočtů tak velké, nicméně v případě renderování scény v reálném čase to může představovat problém. Naštěstí stejně jako ostatní renderovací metody i ray marching se dá optimalizovat.

Nejjednodušší způsob, jak navýšit výkon, je zmenšení maximálního kroku (pokud je jeho hodnota překročena dochází k ukončení ray marching cyklu, protože nic nebylo zasaženo), zmenšení maximálního počtu iterací, zvětšení minimální přijatelné vzdálenosti nebo snížením rozlišení. Tyto řešení ale vedou ke snížení kvality výsledného obrazu. [18]

3.4.1 Bounding sphere

Bounding sphere je optimalizační metoda, při které je pro každé těleso scény vytvořena ohraničující koule. Podmínkou je, že koule musí obsahovat těleso celé. Algoritmus následně zkontroluje, zda paprsek tyto ohraničující koule protne, a ray marching se provádí pouze v rámci hranic ohraničujícího objemu. Uvedená metoda vede k poměrně velkému zvýšení výkonu, protože eliminuje nutnost drahých výpočtů, které jsou mimo ohraničující objemy. Při malém množství objektů ve scéně nebo když jsou objekty jednoduché, nemusí dojít k navýšení výkonu. V některých případech může dokonce dojít ke snížení výkonu z důvodu vyšší režie optimalizace. [7, 17, 18]

3.4.2 Reintroducing Analytic Roots

Optimalizační metoda zvaná Reintroducing Analytic Roots řeší velké geometrické roviny, které často zabírají velkou část obrazovky. Ray marching algoritmus se pohybuje po dráze paprsku v krocích, které se rovnají vzdálenosti od těles. V případě těchto rovin je krok roven výšce paprsku nad rovinou. To může znamenat, že při sklonu paprsku, který se blíží paralelnímu s rovinou a je k ní v blízké vzdálenosti, budou kroky malé v porovnání se vzdáleností od průsečíku. Proto je vhodné pro tyto plochy vypočítat průsečíky analyticky a ray marching použít pro ostatní objekty ve scéně. [18]

3.4.3 Over-relaxation

Over-relaxation je optimalizační metoda, kdy velikost kroku není definována jako vzdálenost od nejbližšího tělesa, ale je vypočítána jako (21):

$$\delta_i = f(p_i) * \omega \qquad (21)$$

Kde f(p_i) je vzdálenost od nejbližšího tělesa i-té iterace a ω ∈ [1; 2) je relaxační parametr. Aby se předešlo „překročení" některého průsečíku, je nutné zavést následující opatření. Kdykoli se dvě unbounding spheres (koule, jejíž poloměr je vzdálenost kroku) překrývají, je nutné zajistit, že paprsek mezi nimi nemůže protínat žádné těleso. To je možné vyřešit tak, že pokud platí $|f(p_{i-1})| + |f(p_i)| < \delta_{i-1}$, může over-relaxation průsečík přejít a je nutné over-relaxation nepoužít. Tyto výpočty mohou být ale v některých případech příliš složité, a využití této metody se nemusí vždy vyplatit. [19]

3.5 OpenGL

OpenGL je API neboli application programming interface pro tvorbu 3D grafiky. OpenGL je využitelné jak na počítačích, tak i na dalších zařízeních ve verzi OpenGL ES [20]. Jedná se o multiplatformní open standard a royaltyfree API. Hlavním důvodem, proč použít OpenGL nebo jiné API, jako například Vulkan nebo Direct3D je, že programátorům poskytuje fixní a programovatelné grafické pipeliny. Díky tomu je možné grafické výpočty provádět na grafických kartách, které jsou pro tuto činnost určené a provádějí grafické výpočty rychleji. [20 s. 1, 21]

OpenGL je stavový stroj, což znamená, že definice proměnných určuje, jak se bude chovat. Rendering tedy probíhá na základě stavu neboli kontextu, který může program upravovat. K tomuto nastavování slouží state-changing funkce. Na druhé straně tu jsou state-using funkce, které provádějí operace na základě současného stavu, ve kterém se OpenGL nachází. [22 č. OpenGL]

3.5.1 Reprezentace objektů

OpenGL nenabízí žádné pokročilé příkazy na definování složitých objektů. API podporuje pouze 2D a 3D primitiva jako jsou body, úsečky, trojúhelníky, čtyřúhelníky a konvexní polygony. Všechny složitější objekty musí být vytvořeny z těchto základních primitiv. Jelikož je OpenGL optimalizované primárně na práci s trojúhelníky, tak se složitější povrchy rozkládají na množinu trojúhelníků, které povrch aproximují. Tomuto procesu se říká teselace. [21 s. 3, 23]

Primitiva se skládají z vertexů (vrcholů). OpenGL dovoluje definovat vetexy více způsoby. Pro každý vertex je možné definovat jeho pozici, barvu, normálu a jednu nebo více souřadnic do textury i určit reprezentaci těchto atributů. Tyto vertexy je v OpenGL (od verze 1.5) možné uložit do vertex buffer objektu. Vertex buffer objekty umožňují alokovat paměť pro uložení vertexů, kterou spravuje OpenGL. K uložení vertexů do bufferu slouží příkazy glBufferData a glMapBuffer. Využití vertex buffer objektů umožňuje dosáhnout většího výkonu díky přímému přístupu k datům. [21 s. 14]

Další užitečnou strukturou je triangle list, který definuje list trojúhelníků, které mezi sebou mohou sdílet vertexy. Sdílení vertexů nastává u dvou a více spojených trojúhelníků a usnadňuje výpočty, protože není nutné operace se sdílenými vertexy provádět pro každý trojúhelník zvlášť. [21 s. 16]

3.5.2 Transformace

Transformace jsou v OpenGL prováděny pomocí 4 × 4 matic a data jsou reprezentována pomocí tzv. tuplů (tuples), což jsou čtyř složkové vektory [21 s. 19]. Transformace tedy jsou sekvence násobení souřadnic vrcholů maticemi. Transformační pipeline obsahuje několik souřadnicových prostorů [21 s. 20; 22 č. Coordinate Systems]:

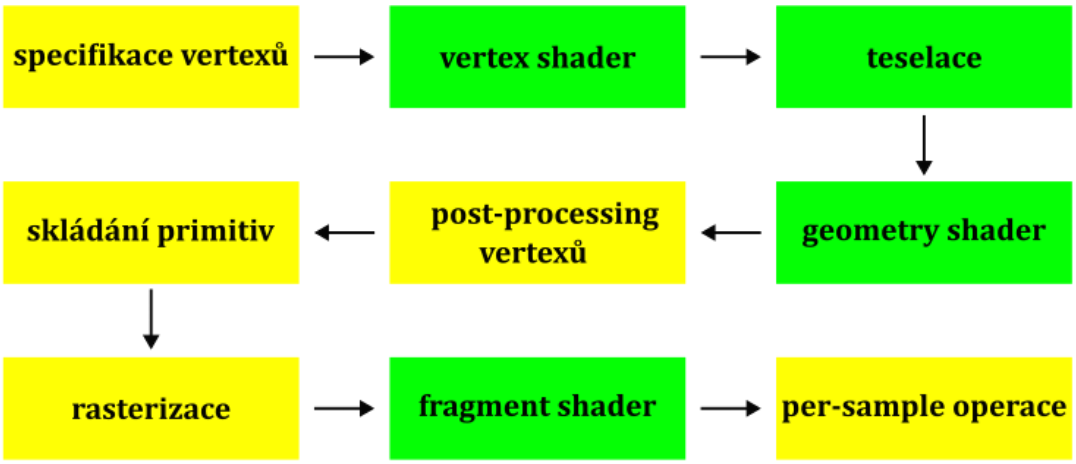
1. Souřadnicový prostor objektu: Jedná se o výchozí souřadnicový prostor objektů. Jsou zde souřadnice jednotlivých vertexů vzhledem k počátečnímu bodu objektu (0,0,0).
2. Souřadnicový prostor scény: Určuje pozice jednotlivých objektů ve scéně. Transformace jako je posunutí, natočení nebo škálování se provádějí zde.
3. Souřadnicový prostor pozorovatele: Jedná se o prostor pohledu kamery. Střed této soustavy je posunut na pozici kamery a celá scéna je natočena podle směru pohledu. Toho je dosaženo aplikací translačních a rotačních matic.
4. Souřadnicový prostor ořezání: Určuje, které prvky budou viditelné. Scéna je po přechodu do těchto souřadnic ořezána ořezávacím objemem.
5. NDC: Normalized Device Coordinates je prostor, který je výsledkem perspektivního vydělení. Souřadnice x, y, a z jsou vyděleny čtvrtou souřadnicí w (22).

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \rightarrow \begin{pmatrix} x/w \\ y/w \\ z/w \\ w/w \end{pmatrix} \quad (22)$$

6. Souřadnicový prostor obrazovky: 2D prostor, který scénu mapuje na pixely obrazovky

3.5.3 Pipeline

K renderování v OpenGL je definována renderovací pipeline, to je posloupnost kroků, které se při renderování objektů provedou. Pipeline vzniká před začátkem renderování a provádí kroky, které jsou uvedené na Obrázek 10. Specifikace vertexů je proces, při němž dochází k předání seřazeného listu vertexů, ze kterých jsou tvořena primitiva, do pipeline. Vertex shader, tesselace, geometry shader a fragment shader jsou programátorem vytvořené programy, které jsou zmíněné v kapitole 3.5.4. Při post-processingu vertexů dochází k provedení operací s vertexy fixně definovaných OpenGL, jako face culling, ořezání do okna obrazovky a další. Rasterizace je proces, kdy jsou primitivům přiřazovány pixely na obrazovce a vzniká tak množina fragmentů. Nakonec proběhnou per-sample operace, což je série testů, jako je pixel ownership test, scissor test, stencil test a depth test. Po dokončení per-sample operací jsou data fragmentů zapsána do framebufferu. [24]



Zeleně vyznačené položky jsou programovatelné

Obrázek 10 OpenGL pipeline. Zdroj: autor

3.5.4 Shadery

Shadery jsou programovatelnou částí dříve zmíněné pipeline. Jedná se o speciální programy, které běží přímo na GPU. Podle svého typu jsou spouštěny ve specifické sekci pipeline, viz. Obrázek 10. Shader program běží paralelně pro každý prvek, jehož se týká. Tedy pro každý vertex v případě vertex shaderu, pro každý fragment, v případě fragment shaderu, apod. Paralelní provedení na grafické kartě znamená výrazně vyšší výkon, ale zároveň to znamená, že jsou programy izolované a nemohou spolu komunikovat. Celá funkce tohoto programu tedy spočívá v transformaci vstupů na výstupy. [22 č. Shaders]

3.5.4.1 Vertex shader

První shader v pipeline je vertex shader a ten je spouštěn pro každý vertex. Hlavním úkolem vertex shaderu je provádět transformace pozic vertexů a také provádět další transformace, jako je například projekční transformace, nebo transformace normál [25]. Programovací jazyk musí být v OpenGL částečně omezen, aby byla hardwarová implementace na grafické kartě efektivní [21 s. 33].

3.5.4.2 Tessellation shader

Tessellation shader umožňuje rozkládat velké plochy na mnoho menších, a tím lépe aproximovat povrch objektu. Díky tomuto shaderu je také možné dynamicky měnit level of detail povrchů. Největší uplatnění tessellation shaderu je při renderování velkých ploch jako je například terén. [26 s. 91]

3.5.4.3 Geometry shader

Geometry shader přijímá jako vstupy výstupy z teselace (pokud proběhla) nebo přímo z vertex shaderu. Tento druh shaderu se spouští pro každé vstupní primitivum. Jeho využití je různorodé, může řešit například backface culling, procedural geometry a další. Jelikož je geometry shader poslední shader před ořezáním a rasterizací, jsou jeho výstupní proměnné v souřadnicovém prostoru ořezání. [22 č. Geometry Shader; 27 s. 73]

3.5.4.4 Fragment shader

Fragment shader je pro tuto bakalářskou práci nejdůležitější, protože v něm bude celý ray marching probíhat. Fragment shader se spouští pro každý pixel okna aplikace. Finálním výstupem z tohoto shaderu je tedy vektor nebo více vektorů

reprezentujících barvu daného pixelu. Využití tohoto shaderu je různorodé, a to od texturování až právě třeba k ray marchingu a dalším technikám. [1 s. 23]

3.5.4.5 GLSL

Každý shader musí být napsán v programovacím jazyce. V případě OpenGL jsou shadery psány v jazyce GLSL (OpenGL Shading Language). GLSL je C-like jazyk se zaměřením na grafické výpočty, jako je manipulace s vektory a maticemi [22 č. Shaders].

Na začátku každého shader programu psaného v GLSL musí být definovaná verze. Ta informuje kompilátor o tom, jaká syntaxe a jaké konstrukty budou v shaderu použity. Verze se mohou lišit nejen syntaxí, ale i funkcemi (features), které nabízí. Některé starší verze mohou obsahovat menší množství datových typů nebo omezovat maximální délku instrukcí shader programu. [20 s. 98]

GLSL obsahuje klíčová slova, z těch nejdůležitějších in a out pro vstupní a výstupní proměnné, uniform pro vstupní konstantní data, samplers pro textury [20 s. 188]. Vzhledem k tomu, že GLSL je určeno ke grafickým výpočtům, nabízí datové typy týkající se vektorů a matic, viz. Tabulka 1.

Třída proměnné	**Typ**	**Popis**
Skaláry	float, int, uint, bool	Klasické C-like typy
Vektory s plovoucí desetinou čárkou	float, vec2, vec3, vec4	1 až 4dimenzionální vektory float hodnot
Celočíselné vektory	int, ivec2, ivec3, ivec4	1 až 4dimenzionální vektory celočíselných hodnot
Celočíselné vektory bez znaménka (unsigned)	uint, uvec2, uvec3, uvec4	1 až 4dimenzionální vektory celočíselných hodnot, které nemají znaménko a jsou pouze pozitivní hodnoty
Booleovské vektory	Bool, bvec2, bvec3, bvec4	1 až 4dimenzionální vektory hodnot true/false
Matice	mat2, mat2x2, mat2x3, mat2x4, mat3x2, mat3, mat3x3, mat3x4, mat4x2, mat4x3, mat4, mat4x4	Matice 2x2 až 4x4, v případě čtvercové matice lze napsat jako matY nebo matYxY

Tabulka 1 Typy proměnných v GLSL. Zdroj: [20 s. 99]

4 Praktická část

Praktická část bakalářské práce se bude zabývat vytvořením spustitelné aplikace zobrazující scénu pomocí ray marching metody. Cílem je ověřit a vyzkoušet algoritmy a metody zmíněné v teoretické části a také otestovat vytvořenou aplikaci.

4.1 Příprava

Pro programování aplikace bylo zvoleno OpenGL API a jako programovací jazyk aplikace Java. Z tohoto důvodu byla pro přístup k OpenGL API použita knihovna Lightweight Java Game Library (LWJGL). V kódu aplikace byly také využity balíčky lwjglutils a transforms, které byly vytvořeny na fakultě informatiky a managementu UHK.

4.1.1 Renderování pomocí dvou trojúhelníků

Vzhledem k tomu, že celý ray marching algoritmus probíhá ve fragment shaderu, je nutné celé okno aplikace překrýt trojúhelníky. Toho bylo docíleno využitím vertex bufferu obsahujícího dva trojúhelníky. Každý trojúhelník překrývá půlku okna a tyto trojúhelníky jsou vzájemně symetricky převrácené podél diagonální osy.

Dalším krokem byla kompilace shader programu a jeho aplikace na zmíněné trojúhelníky. Na začátek je nutné shader zkompilovat. Zkompilovaný shader je následně možné aplikovat na výše zmíněné trojúhelníky ve vertex bufferu pomocí:

```
buffers.draw(GL_TRIANGLES, shader);
```

Výpis z kódu 5 Vertex buffer. Zdroj: autor

Pro použití v real time aplikaci je uvedené volání nutné provádět v nekonečném cyklu rendereru.

4.2 Implementace ray marchingu

Od této kapitoly dále, pokud není specifikováno jinak, je všechen kód tvořen ve fragment shaderu. Implementace samotného raymarching algoritmu je stěžejní součástí této bakalářské práce, proto bude implementace popsána co nejpodrobněji.

4.2.1 UV souřadnice

Před samotnou implementací ray marching algoritmu je nutné vyřešit problém s poměrem stran okna aplikace a získat souřadnice pixelů na obrazovce. Získání souřadnic fragmentu je možné pomocí gl_FragCoord. Tyto souřadnice jsou dále

převedeny na souřadnice od -1 do 1. K tomu je nutné od původních souřadnic odečíst 0,5 protože v openGL *gl_FragCoord* jsou posunuty do středu fragmentu (místo souřadnice 0; 0 je zde souřadnice 0,5; 0,5). Následně jsou souřadnice vyděleny šířkou okna aplikace v případě x a výškou okna aplikace v případě y. Nakonec se už jen každá souřadnice vynásobí dvěma a odečte se od ní jedna.

```
uv.x = ((gl_FragCoord.x - 0,5) / width)*2-1
uv.y = ((gl_FragCoord.y - 0,5) / height)*2-1
```

Výpis z kódu 6 Výpočet uv souřadnic. Zdroj: autor

Pro vyřešení deformace obrazu způsobené poměrem stran bylo implementováno následující řešení. Cílem je vynásobení jedné ze souřadnic poměrem stran, aby došlo ke zhuštění souřadnic v dané ose.

```
float aspectRatio = width/height;
if (aspectRatio < 1) {
    uv.y *= 1/aspectRatio;
} else {
    uv.x *= aspectRatio;
}
```

Výpis z kódu 7 Výpočet poměru stran. Zdroj: autor

4.2.2 Počátek a směr paprsku

Počátek paprsku je souřadnice určující, ze kterého místa paprsek vzniká. Pro účely bakalářské práce budou jako počátek paprsku dosazeny souřadnice pozice kamery. V případě směru paprsku je jeho vektor získán normalizací vektoru pohledu kamery vynásobeného vektorem uv rozšířeného na čtyřrozměrný vektor.

4.2.3 Sphere tracing algoritmus

Sphere tracing je implementovaný jako funkce, která přijímá počátek paprsku a směr paprsku, popřípadě další parametry, které ale teď nejsou důležité. Funkce vrací informace o zasaženém povrchu. Vzhledem k tomu, že v současné fázi ještě není způsob, jak obarvit tělesa, je k vizualizaci použita délka cesty uražené paprskem, která je následně převedena na černo bílou škálu, viz. Obrázek 11.

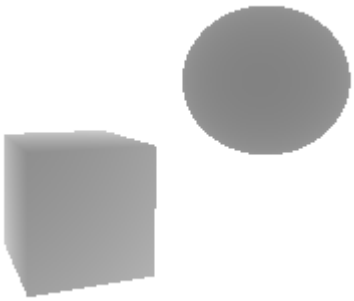
Sphere tracing algoritmus obsahuje hlavní cyklus, ve kterém se postupně pohybuje po paprsku, až narazí na nejbližší povrch, nebo uražená vzdálenost překoná maximální povolenou vzdálenost. Z důvodu nekonečného zacyklení je cyklus omezen

maximálním počtem kroků (steps). Obsah cyklu probýhá následovně. Otestuje se, zda nebyla překročena maximální vzdálenost. K současné pozici se přičte směr paprsku vynásobený uraženou cestou (při první iteraci 0). Získá se vzdálenost nejbližšho povrchu ze současné pozice. Pokud je vzdálenost menší než minimální vzdálenost, je povrch považovaný za zasažený, pokud ne, přičte se vzdálenost od nejbližšího povrchu k celkové uražené vzdálenosti a cyklus se opakuje.

Implementace v kódu:

```
for (int i = 0; i < steps; i++) {
    if (distTraveled > maxDist) {
        break;
    }
    currentPos = rayOri + rayDir * distTraveled;
    float distToClosest = mapScene(currentPos);
    if (distToClosest < minDist) {
        return distTraveled;
    }
    distTraveled += distToClosest;
}
```

Výpis z kódu 8 Implementace sphere tracing algoritmu. Zdroj: autor



Obrázek 11 Sphere tracing vracející délku uražené cesty paprsku. Zdroj: autor

4.2.4 Reprezentace scény

Scéna je reprezentována pomocí SDF. Výpočet nejbližšího povrchu má na starost funkce mapScene. V této funkci se volají funkce jednotlivých SDF těles, aplikují se zde transformace a kombinační funkce, viz. kapitola 3.1.4.

Jelikož je nutné o povrchu uchovat více informací, než je jen jeho vzdálenost, je definována struktura, která obsahuje veškeré informace o zasaženém povrchu potřebné pro další výpočty.

```
struct Hit {
    float distance;
    vec3 col;
    float reflectivity;
    float refractivity;
};
```

Výpis z kódu 9 Struktura Hit. Zdroj: autor

Po získání zásahu se volá modifikovaná verze funkce mapScene, která už vrací strukturu Hit. Díky tomu je možné pixely obrazovky vybarvit příslušnou barvou povrchu.

4.2.5 Transformace

Transformace jsou implementovány jako funkce, které mají jako parametr bod 3D prostoru, který následně transformují a vrátí. Základní transformace (translace, škálování, rotace) byly implementovány následovně.

```
vec3 translate(vec3 p, vec3 t) {
    return p - t;
}
```

Výpis z kódu 10 Posunutí. Zdroj: autor

```
vec3 scale(vec3 p, vec3 s) {
    mat4 S = mat4(
    vec4(1/s.x, 0, 0, 0),
    vec4(0, 1/s.y, 0, 0),
    vec4(0, 0, 1/s.z, 0),
    vec4(0, 0, 0, 1));
    return (vec4(p, 1)*S).xyz;
}
```

Výpis z kódu 11 Škálování. Zdroj: autor

```
vec3 rot3D(vec3 p, vec3 axis, float angle) {
    angle = radians(angle);
    return mix(dot(axis, p) * axis, p, cos(angle))
    + cross(axis, p) * sin(angle);
}
```

Výpis z kódu 12 Rotace. Zdroj: autor

Mezi další implementované transformace patří symetrie, ohnutí, zkroucení, a opakování.

4.2.6 Kombinace SDF

Na rozdíl od transformací ke kombinování objektů scény dochází až po návratu z jejich SDF. Kombinační funkce byly vytvořeny tak, aby přijímaly jako vstup dva objekty (jejich vzdálenost nebo strukturu Hit) a parametr hladkosti k. Pro použití jsou implementovány funkce sjednocení, rozdíl a průnik.

4.2.7 Textury

Barvu zasaženého místa povrchu je možné zadat mnoha způsoby. Barva může být konstantní pro celý povrch nebo může být vypočtená nějakou funkcí. Tyto funkce mohou barvu vypočítat pomocí matematických formulí nebo ji získat z textury.

V GLSL je 2D textura reprezentovaná typem proměnné s názvem sampler2D. Je možné ji do shaderu předat jako uniform. Pro čtení jejích kanálů slouží metoda texture2D.

Pro správné namapování textury na povrch tělesa je nutné kromě pozice bodu, kde byl povrch zasažen, znát i normálu povrchu v tomto bodě. O výpočtu normály více v kapitole 4.3.1. Způsobů, jak texturu namapovat, je více. V rámci této práce výpočet postupuje následovně. Nejdříve se získá z textury barva v daném bodě pro každou rovinu 3D prostoru (XY, XZ, YZ). Je zde použita funkce modulo, aby se textura opakovala do nekonečna. Následně je vypočítána třetí mocnina absolutní hodnoty normály. Výsledná barva je získána sečtením barev získaných v jednotlivých rovinách, které jsou vynásobeny jednou ze souřadnic umocněné normály. V případě roviny XY se násobí souřadnicí Z, dále pak XZ souřadnicí Y a YZ souřadnicí X. Vzhledem k tomu, jak jsou implementované transformace, je možné je aplikovat i na textury.

4.3 Osvětlení scény

Poté, co je nalezen průsečík sledovaného paprsku a povrchu, je výsledek předán na výpočet osvětlení. Osvětlení scény přispívá ke zlepšení vizuální kvality renderované scény. Kapitola se nebude zabývat pouze stínováním ale i vrženými stíny a ambient occlusion.

4.3.1 Výpočet normály

Předtím než je možné vypočítat osvětlení, je nutné získat normálu povrchu v místě zásahu. K výpočtu normály slouží metoda konečných diferencí, jedná se tak o aproximaci. Nejdříve je nutné definovat hodnotu, o kterou se bod posune (dále jen epsilon). Hodnota epsilon by měla být tak malá, aby výpočet normály nenarušili detaily povrchu, ale ne příliš malá, aby se při výpočtech ztratila, kvůli float precision. Následně se vytvoří nové body z předchozího bodu na zasaženém povrchu, které se posunou o epsilon, každý na jedné z os. Pro tyto body se znovu mapuje scéna, aby se získala jejich vzdálenost od povrchu. Výsledná normála je pak tedy vektor tvořený získanými vzdálenostmi bodů v korespondujících osách, od kterých je odečtena vzdálenost bodu posunutého v opačném směru, a to celé je vyděleno epsilonem.

4.3.2 Implementace osvětlení

Pro výpočet osvětlení byl implementován Blinn-Phongův osvětlovací model. Model pracuje se třemi složkami světla ambientní, difuzní a spektrální. Ambientní složku není nutné počítat, jedná se pouze o konstantu mezi 0 a 1. Výpočet difuzní složky je velmi jednoduchý, jedná se o skalární součin normály a vektoru směru ke zdroji světla.

```
float diffuseStrength = max(0.0, dot(dirToLight, normal));
```

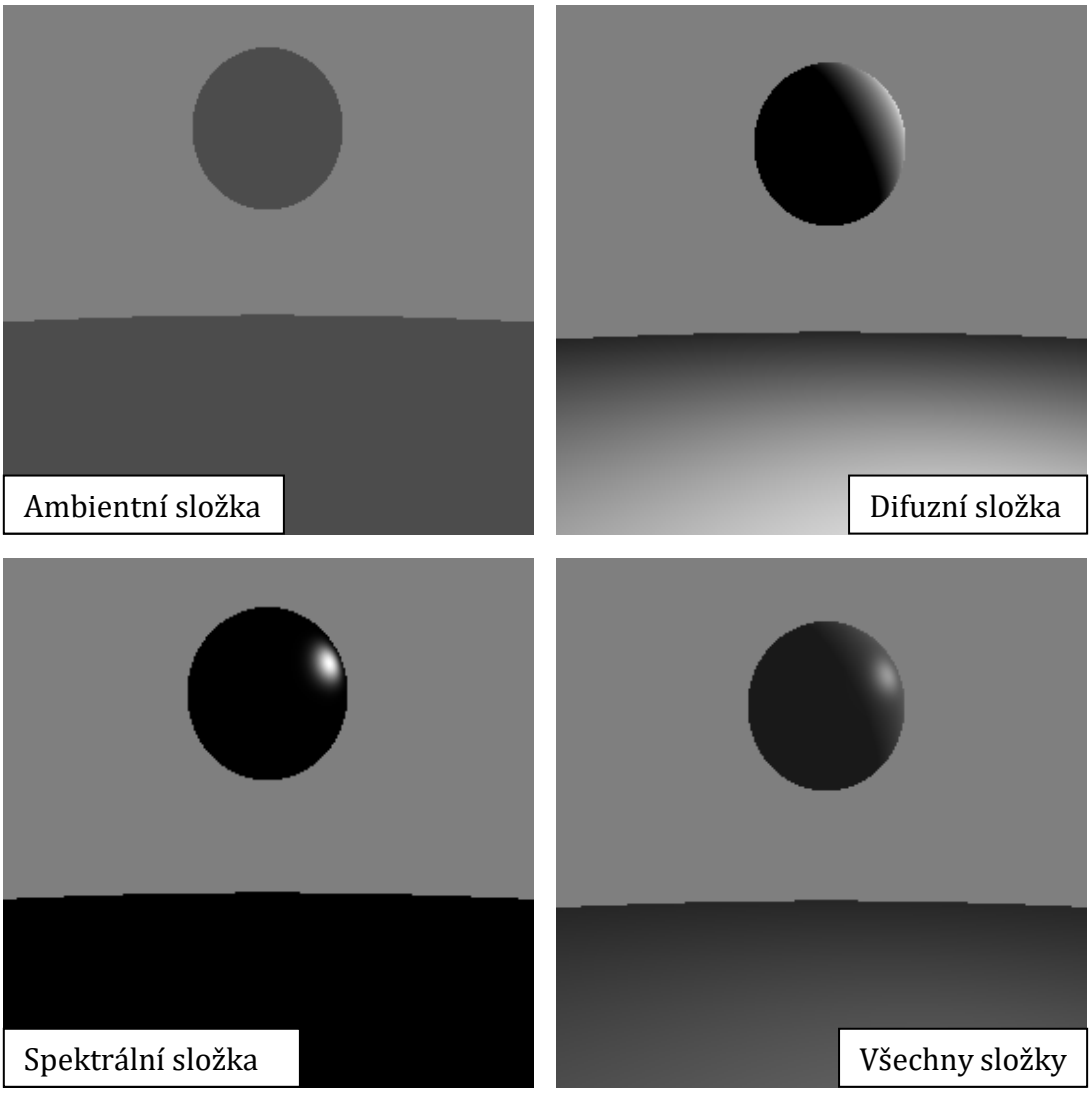
Výpis z kódu 13 Výpočet difuzní složky. Zdroj: autor

Spektrální složka se v Phongově modelu vypočítává pomocí odraženého paprsku světla. V případě Blinn-Phongova modelu stačí vypočítat halfway vektor. Halfway vektor je normalizovaný vektor tvořený rozdílem vektoru směru ke světlu a vektoru normály. Spektrální složka se následně vypočítá skalárním součinem normály a halfway vektoru. Výslednou hodnotu je vhodné umocnit, aby se odlesky utlumily.

```
vec3 halfway = normalize(dirToLight - cameraDir);
float specularStrength = max(0.0, dot(normal, halfway));
specularStrength = pow(specularStrength, 32);
```

Výpis z kódu 14 Výpočet spektrální složky. Zdroj: autor

Výsledné hodnoty osvětlení je nutné vynásobit barvou světla, popřípadě intenzitou světla. Intenzita světla je konstanta a její hodnota záleží čistě na implementaci. Nakonec se barva zasaženého povrchu vynásobí získanou hodnotou osvětlení (vektor).



Obrázek 12 Osvětlení scény. Zdroj: autor

4.3.3 Implementace vržených stínů

Dalším krokem je výpočet vržených stínů. Vržené stíny vznikají, když je dráha paprsku vrženého ze zasaženého povrchu směrem ke světlu přerušena jiným objektem scény. Vzhledem k tomu, že vržené stíny z bodového zdroje světla jsou vždy tvrdé, byla aplikována metoda pro vytvoření falešných měkkých stínů, viz. Obrázek 9.

Principem výpočtu vržených stínů je vytvoření nového paprsku z místa, pro které je stín vypočítáván, směrem ke zdroji světla. Po tomto paprsku se následně postupuje stejně jako v implementaci samotného sphere tracing algoritmu. Pokud je při tomto procesu zasažen nějaký povrch, vrátí funkce 0, pokud ne, vrátí 1. V případě algoritmu pro výpočet měkkých stínů je místo nuly a jedničky vrácena hodnota vypočtená následovně (23):

$$výsledná\ hodnota = k * h/t \qquad (23)$$

Kdy k je koeficient určující ostrost stínu (čím větší číslo, tím více se podobá tvrdému stínu). Dále h je vzdálenost bodu na paprsku od nejbližšího povrchu a t je uražená vzdálenost po paprsku. Výsledkem je nejmenší hodnota získaná tímto výpočtem ze všech iterací.

```
float result = 1.0;
for (float t = mint; t < maxt;)
{
    float h = mapScene(rayOri+rayDir*t);
    if (h < 0.001)
    {
        return 0.0;
    }
    result = min(result, k*h/t);
    t += h;
}
return result;
```

Výpis z kódu 15 Implementace algoritmu na výpočet měkkých vržených stínů.
Zdroj: autor

Ve Výpis z kódu 15 je zmíněn mint a maxt, to jsou parametry určující vzdálenost, pro kterou se bude vržený stín zjišťovat, proto příliš malá hodnota maxt způsobí, že vržené stíny od objektů příliš daleko od povrchu nebudou započítány.

4.3.4 Více zdrojů světla

Pokud je ve scéně více zdrojů světla je nutné, aby se difuzní složka, spektrální složka a vržené stíny vypočítaly pro každý zdroj světla zvlášť. Následně se hodnoty osvětlení z každého zdroje sečtou. Zároveň je nutné vypočítat odstín barvy ambientního osvětlení. V případě této aplikace je odstín vypočítán jako průměrná hodnota barev všech světelných zdrojů.

4.3.5 Implementace ambient occlusion

Poslední algoritmus spojený s výpočtem osvětlení ve vytvářené aplikaci je algoritmus řešící ambient occlusion. Vzhledem k tomu, že vysvětlení, co je ambient occlusion, je v části 3.3.6, bude zde uvedeno pouze implementační řešení zvolené v rámci aplikace. Implementace tohoto algoritmu je jednoduchý způsob, jak vylepšit vizuální stránku renderování scény, viz. Obrázek 8.

Principem implementovaného algoritmu je posunout bod, ve kterém paprsek zasáhl povrch o určitou vzdálenost ve směru normály povrchu. Následně se pro posunutý bod získá vzdálenost od nejbližšího povrchu. Pokud je vzdálenost menší než vzdálenost, o kterou se bod posunul, je rozdíl vzdáleností vydělen vzdáleností, o kterou se bod posunul, a zaznamenán do proměnné. Pro zvýšení kvality je vhodné bod posunout vícekrát o různé vzdálenosti a výsledky sečíst. Implementace výpočtu ambient occlusion cyklu je zde:

```
for (int i = 1; i <= aoIterations; i++)
{
dist = step * i;
ao += max(0.0, (dist - mapScene(p + n * dist)) / dist);
}
return (1.0 - ao * aoIntensity);
```

Výpis z kódu 16 Ambient occlusion cyklus. Zdroj: autor

Výstupem z funkce ve Výpis z kódu 16 je hodnota, která je rozdílem 1.0 a proměnné ambient occlusion (ao) získanou výpočtem. K regulaci intenzity efektu je vhodné získanou proměnou vynásobit volitelnou konstantou.

4.4 Implementace odrazu a lomu

Odraz a lom jsou výpočetně náročné, ale umožňují vytvářet zrcadlové a refraktivní povrchy. Tyto vlastnosti povrchu jsou uloženy ve struktuře Hit, která byla

zmíněna dříve. Jak hodnota reflektivity, tak i refractivity, jsou čísla mezi 0 a 1 a určují, jak moc velký příspěvek k výsledné barvě bude mít odražený/lomený paprsek. Vzhledem k tomu, že GLSL nedovoluje rekurzi, je nutné algoritmus provádět v cyklu. Cyklus ale zároveň dovoluje jednoduchou implementaci maximálního počtu odrazů, díky které se předejde nekonečnému odrazu paprsku ve scéně.

4.4.1 Odraz

Pro vytvoření paprsku odrazu je nejdříve potřeba posunout pozici zásahu o kousek ve směru normály povrchu, aby se předešlo zasažení stejného povrchu ray marching algoritmem. Následně je možné vypočítat pomocí směru dopadu paprsku a normály povrchu směr odraženého paprsku:

$$odraženýPaprsek = směrDopadu - 2.0 * dot(směrDopadu, normála) * normála;$$

V získaném směru je znovu spuštěn sphere tracing algoritmus, který vrací nový zásah (Hit). Struktura hit obsahuje informace o materiálu, takže je možné k původní barvě přičíst barvu odraženého povrchu vynásobenou odrazivosti původního povrchu. Hit také obsahuje informace o reflexivnosti a refraktivnosti nově zasaženého povrchu, z těchto vlastností je možné dále určit, zda je nutné pokračovat s výpočtem lomu a odrazu na novém povrchu.

```
currentPos += normal*0.02;
vec3 reflected = reflect(rayDir, normal);
Hit o = rayMarching(currentPos, reflected, currentPos, normal);
color += o.col*reflective;
```

Výpis z kódu 17 Implementace odrazu světla. Zdroj: autor

Parametry currentPos a normal zde vstupují jako in out parametry a jsou vráceny jako pozice nově zasaženého povrchu a jeho normála.

Obrázek 13 Vícenásobný odraz paprsku. Zdroj: autor

4.4.2 Lom

Implementace lomu paprsku je oproti odrazu složitější a výpočetně náročnější. Je totiž nutné paprsek lomit, jak při vstupu do tělesa, tak i při výstupu z tělesa. Zároveň je nutné, aby sphere tracing algoritmus fungoval vně těles (je potřeba invertovat scénu), a proto je nutné vynásobit vzdálenosti, které vrací SDF mínus jedničkou. K tomuto účelu byla v rámci této aplikace vytvořena odlehčená verze ray marching algoritmu, která neřeší vlastnosti povrchu ani osvětlení, ale funguje v invertované scéně. Implementace pak vypadá následovně:

```
vec3 refractedIn = refract(rayDir, normal, 1.0, 1.5);
vec3 rayOriIn = currentPos - normal*0.02;
float dIn = rayMarchingSimple(rayOriIn, refractedIn, currentPos,
normal, -1.0);
vec3 refractedOut = refract(refractedIn, normal, 1.5, 1.0);
```

Výpis z kódu 18 Implementace lomu světla. Zdroj: autor

Výpočet směru lomeného paprsku je poměrně složitý a je vysvětlený v teoretické části, viz. kapitola 3.1.7.2. Výsledný kód je uveden ve Výpis z kódu 19 Výpočet vektoru odraženého paprsku. Zdroj: autorVýpis z kódu 19, kdy n1 a n2 jsou indexy lomu prostředí, mezi kterými paprsek prochází.

```
float n = n1/n2;
float cosI = -dot(normal, rayDir);
float sinT2 = n*n*(1.0-cosI*cosI);
if (sinT2 > 1.0) {
return vec3(0.0);
}
float cosT = sqrt(1.0-sinT2);
return n*rayDir + (n*cosI-cosT)*normal;
```

Výpis z kódu 19 Výpočet vektoru odraženého paprsku. Zdroj: autor

Následně je nutné ošetřit situaci, kdy skalární součin paprsku opouštějícího těleso (refractedOut) sama se sebou je roven nule. V takovém případě je nutné nahradit paprsek lomu paprskem odrazu. Následně je možné pokračovat s novým paprskem dalším ray marchingem mimo těleso. Proces je pak velice podobný odrazu.

```
refractedOut = reflect(refractedIn, normal);
vec3 rayOriOut = rayOri - normal*0.02;
Hit dOut =rayMarching(rayOriOut, refractedOut, currentPos,
normal);
```

Výpis z kódu 20 Řešení v případě TIR. Zdroj: autor

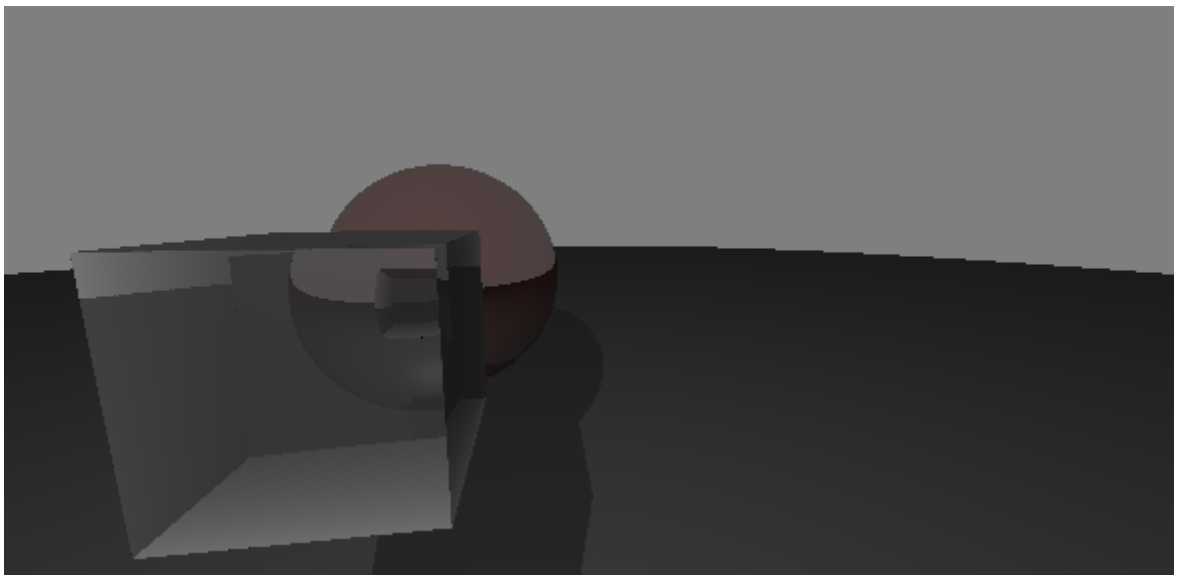
V této fázi se také vypočítává optická vzdálenost (opticalDist), která simuluje útlum paprsku uvnitř tělesa. To způsobí, že čím delší trasu musí paprsek uvnitř tělesa urazit, tím menší bude jeho přínos k výsledné barvě.

Nakonec zbývá implementace Fresnelovy rovnice, která je implementovaná jako $(1 + (směr\ paprsku \cdot normála))^{Fresnelův\ koeficient\ odrazu}$. Pomocí této hodnoty se interpoluje mezi barvou získanou lomem a barvou získanou odrazem od povrchu tělesa.

```
float fresnel = pow(1.0+dot(rayDir, normal), 5.0);
vec3 reflected = reflect(dir, normal);
Hit o = rayMarching(currentPos, reflected, currentPos, normal);
color = mix(color, o.col, fresnel);
```

Výpis z kódu 21 Implementace Fresnelovy rovnice. Zdroj: autor

Z důvodu zjednodušení je zde Fresnelův koeficient odrazu pouze konstanta (ve Výpis z kódu 21 se jedná o hodnotu 5.0).



Obrázek 14 Lom paprsku. Zdroj: autor

4.5 Fraktály

Implementace fraktálů pomocí ray marchingu je poměrně snadná, ale s narůstajícím detailem narůstá i výpočetní náročnost. Existuje mnoho různých druhů 3D fraktálů jako například Sierpiński tetrahedron, Apollonian sphere packing, Menger sponge a mnoho dalších.

V rámci implementace byl zvolen fraktál s názvem Menger sponge, který je 3D verzí 2D fraktálu s názvem Sierpiński carpet. Jedná se o krychli rozdělenou na 27 podkrychlí třetinové velikosti, z jejíhož středu a středu každé její strany jsou odstraněny krychle. Celý proces se pak opakuje pro zbývající menší krychle při každé iteraci, a tím je tvořen fraktál.

Samotná implementace je inspirovaná článkem od Iniga Quileze [6]. Aby bylo možné odstranit středy krychle, je nutné vytvořit kříž. Kříž je tvořen krychlemi roztaženými do nekonečna, každá v jednom směru. Tímto křížem je následně krychle ořezána pomocí operace rozdílu, zmíněné v kapitole 4.2.6. Tento proces se opakuje pro všechny „podkrychle“ s využitím operátoru modulo.

Pro ukázku možností této renderovací metody je fraktál do nekonečna opakován pomocí transformace opakování.

```
vec3 repeated( vec3 p, float s )
{
    vec3 r = p - s*round(p/s);
    return r;
}
```

Výpis z kódu 22 Implementace opakování. Zdroj: autor

Obrázek 15 Fraktál. Zdroj: autor

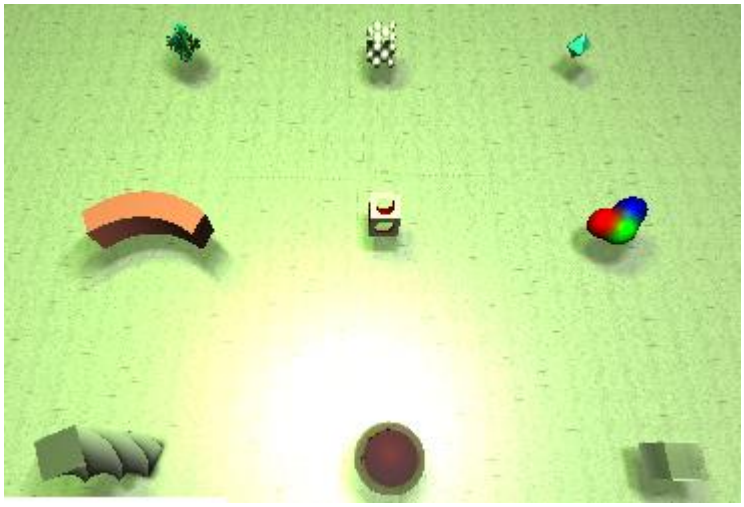
4.6 Finální Aplikace

Vzhledem k tomu, že hlavním požadavkem na aplikaci je demonstrace použití metody ray marchingu k renderování scény, obsahuje aplikace několik scén. Tyto scény mají za úkol ukázat různé výhody, vlastnosti a využití této renderovací metody. Scény jsou navigovatelné pomocí kláves WASD a rozhlížení je umožněno podržením pravého tlačítka myši. Přechod mezi scénami je zařízený pomocí hlavní nabídky, do které je možné kdykoli vstoupit stisknutím klávesy escape.

4.6.1 Přehled scén

Fraktál: První scéna je ukázkou implementace předchozí kapitoly. Jak už bylo zmíněno, jedná se o fraktál s názvem Menger sponge, viz. Obrázek 15. Scéna obsahuje nekonečně se opakující fraktál ve všech směrech a zdroj osvětlení.

Ukázka 1: Smyslem této scény je ukázat různé transformace a SDF. Scéna obsahuje 9 objektů, které jsou vytvořené pomocí technik zmíněných v této bakalářské práci. Scéna také obsahuje více zdrojů světla a osvětlení periodicky mění barvy.



Obrázek 16 Ukázka 1. Zdroj: autor

Ukázka 2: Poslední scéna je vytvořená jako ukázka, že ray marching a SDF je možné použít k vytvoření komplexnějších objektů. Tato scéna obsahuje brýle, jejichž sklo je ve tvaru spojné čočky, díky čemuž funguje jako lupa. Scéna také obsahuje telefon, na kterém je demonstrováno mapování textur v 3D prostoru, a zrcadlo.

Obrázek 17 Ukázka 2. Zdroj: autor

4.7 Testování

Poslední kapitola praktické části se bude zabývat testováním vytvořené aplikace. Vzhledem k tomu, že je každá z dříve uvedených scén jinak výpočetně náročná, byly scény testovány zvlášť. Testování bylo provedeno na více počítačích, aby byly výsledky více vypovídající.

Jako metrika měření bylo vybráno fps (snímky za sekundu). Vzhledem k tomu, že fps se mohu lišit nejen v závislosti na hardwaru a složitosti scény, ale i na směru, kterým je kamera natočená, tak byla vždy zaznamenána nejnižší naměřená hodnota fps (vyjma anomálii). Poklesy fps v závislosti na směru pohledu kamery mohou být způsobeny například různými vlastnostmi povrchů. Reflektivní a refraktivní povrchy vytvářejí sekundární paprsky, a proto je vykreslování těchto povrchů výpočetně náročnější.

K účelu měření fps bylo do aplikace implementováno jednoduché počítadlo fps. Všechny výsledky byly naměřeny pomocí tohoto počítadla.

V dalších odstavcích budou zmíněny testované počítače s hardwarovými parametry a naměřenými hodnotami fps. Při testování aplikace byly vypnuty všechny ostatní programy, aby pro testované počítače vznikly rovné podmínky. Kategorie fullscreen je vázaná na parametry monitoru testovaného počítače, a proto se zde podmínky testovaných počítačů liší.

Testy:

1. **Testovaný počítač:**

OS: Windows 10
Procesor: Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
GPU: NVIDIA GeForce RTX 3060, 12GB VRAM
Verze GPU ovladače: 551.76
Monitor: 1680 x 1050

Scéna	***600 x 400***	***1680 x 1050***
Hlavní menu	720 fps	720 fps
Fraktál	76 fps	20 fps
Scéna 1	315 fps	160 fps
Scéna 2	120 fps	35 fps

Tabulka 2 Tabulka naměřených hodnot 1. Zdroj: autor

2. **Testovaný počítač:**

OS: Windows 10

Procesor: Intel(R) Core(TM) i3-5005U CPU @ 2.00GHz

GPU: Intel(R) HD Graphics 5500

Monitor: 1366 x 768

Scéna	***600 x 400***	***1366 x 768***
Hlavní menu	60 fps	60 fps
Fraktál	4 fps	<1 fps
Scéna 1	30 fps	10 fps
Scéna 2	15 fps	4 fps

Tabulka 3 Tabulka naměřených hodnot 2. Zdroj: autor

Nejslabší výsledky testování byly naměřeny ve scéně s fraktálem, což je zapříčiněno vyšší výpočetní náročností této scény. Další rozdíl v rychlosti aplikace tvořilo rozlišení, to však bylo očekávané. Ray marching provádí veškeré výpočty na úrovni pixelů, a proto velké množství pixelů sníží výkon aplikace.

5 Shrnutí a diskuse výsledků

Bakalářská práce prozkoumává oblast ray marchingu a témat s ní spojených. Teoretická část práce v kapitolách popisuje a vysvětluje algoritmy, metody a postupy s ray marching metodou spojené a porovnává je s tradičními metodami vykreslování 3D scény. Praktická část se dále zabývá samotnou implementací zmíněných algoritmů a tvorbou ukázkové aplikace.

Vytvořená aplikace je funkční implementací ray marching algoritmu. Obsahuje více scén, jejichž účelem je ukázat různé vlastnosti a možnosti ray marchingu. Z testování se však ukázalo, že na slabších zařízeních aplikace běží pomalu. Z tohoto důvodu by bylo vhodné implementovat alespoň nějaké optimalizační metody. Příkladem mohou být optimalizační metody popsané v teoretické části 3.4.

6 Závěry a doporučení

Primárním cílem bakalářské práce bylo prozkoumat a popsat metodu ray marching jako techniku vizualizace 3D scén. Dalším cílem bylo implementování techniky a vytvoření funkční aplikace. Oba tyto cíle byly dosaženy. Cílem bakalářské práce bylo také porovnat tuto metodu s tradičními metodami zobrazování 3D scény. Cíl byl naplněn v rámci teoretické části, kde byly vytyčeny výhody a nevýhody této metody oproti ostatním metodám.

Ray marchng je díky jeho vlastnostem použitelný v různorodých typech úloh. Hojně se využívá například v mnoha filmech a počítačových hrách v kombinaci s jinými technikami vizualizace. S vývojem grafického hardware tak lze očekávat i větší využití ray marchingu.

Vzhledem k tomu, že se tato bakalářská práce zabývá primárně základy vykreslování 3D scény, je vhodné v případě zájmu o tuto techniku doporučit i pokročilejší využití ray marchingu, jako je rendering mraků, volumetrického světla, vody a dalších jevů, implementace kolizí, nebo například využití této techniky na vizualizaci dat. Dalším způsobem, jak tuto práci rozšířit, by bylo například implementování optimalizačních metod.

7 Seznam použité literatury

[1] MÖLLER, Tomas, Eric HAINES a Naty HOFFMAN. *Real-time rendering*. Fourth edition. Boca Raton: CRC Press, Taylor and Francis Group, 2018. ISBN 978-1-351-81615-1.

[2] ŽÁRA, Jiří, Bedřich BENEŠ, Jiří SOCHOR a Petr FELKEL. *Moderní počítačová grafika*. B.m.: Computer Press, 2004. ISBN 978-80-251-0454-5.

[3] GLASSNER, Andrew S. *An Introduction to Ray Tracing*. B.m.: Morgan Kaufmann, 1989. ISBN 978-0-12-286160-4.

[4] SHIRLEY, Peter. *Physically based lighting calculations for computer graphics*. B.m., 1991. b.n.

[5] FENG, Brandon Y., Yinda ZHANG, Danhang TANG, Ruofei DU a Amitabh VARSHNEY. PRIF: Primary Ray-Based Implicit Function. In: Shai AVIDAN, Gabriel BROSTOW, Moustapha CISSÉ, Giovanni Maria FARINELLA a Tal HASSNER, ed. *Computer Vision – ECCV 2022* [online]. Cham: Springer Nature Switzerland, 2022, s. 138–155. Lecture Notes in Computer Science. ISBN 978-3-031-20062-5. Dostupné z: doi:10.1007/978-3-031-20062-5_9

[6] QUILEZ, Inigo. *Inigo Quilez* [online]. [vid. 2023-11-25]. Dostupné z: https://iquilezles.org

[7] HART, John C. Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer* [online]. 1996, **12**(10), 527–545. ISSN 01782789. Dostupné z: doi:10.1007/s003710050084

[8] DE GREVE, Bram. Reflections and Refractions in Ray Tracing. In: [online]. 2004. Dostupné z: https://www.semanticscholar.org/paper/Reflections-and-Refractions-in-Ray-Tracing-Greve/b38816d735e4c763ee7027b5365ebbc13028e4fd

[9] HUANG, Benjamin, Jake WAKSBAUM a Anvay GROVER. COS 426 Final Project Writeup Raymarching with Signed Distance Fields [online]. 2020 [vid. 2024-01-03]. Dostupné z: https://jbaum98.github.io/cos426_final_project/writeup.pdf

[10] BORN, Max a Emil WOLF. *Principles of Optics: Electromagnetic Theory of Propagation, Interference and Diffraction of Light*. 7. vyd. Cambridge: Cambridge University Press, 1999. ISBN 978-0-521-64222-4.

[11] SCHLICK, Christophe. An Inexpensive BRDF Model for Physically-based Rendering. *Computer Graphics Forum* [online]. 1994, **13**(3), 233–246. ISSN 1467-8659. Dostupné z: doi:10.1111/1467-8659.1330233

[12] FEDER, Jens. *Fractals* [online]. Boston, MA: Springer US, 1988 [vid. 2024-01-08]. ISBN 978-1-4899-2126-0. Dostupné z: doi:10.1007/978-1-4899-2124-6

[13] GOESELE, Michael. New Acquisition Techniques for Real Objects and Light Sources in Computer Graphics [online]. 2004 [vid. 2023-12-03]. Dostupné z: doi:10.2312/8166

[14] ELIAS, Rimon. Illumination, Lighting and Shading. In: Rimon ELIAS, ed. *Digital Media: A Problem-solving Approach for Computer Graphics* [online]. Cham: Springer International Publishing, 2014 [vid. 2023-12-08], s. 525–579. ISBN 978-3-319-05137-6. Dostupné z: doi:10.1007/978-3-319-05137-6_11

[15] LANDIS, H. Production-Ready Global Illumination. In: [online]. 2004 [vid. 2023-12-10]. Dostupné z: https://www.semanticscholar.org/paper/Production-Ready-Global-Illumination-Landis/4a9de79235445fdf346b274603dfa5447321aab6

[16] BUNNELL, Michael. Dynamic Ambient Occlusion and Indirect Lighting. In: [online]. 2005 [vid. 2023-12-10]. Dostupné z: https://www.semanticscholar.org/paper/Dynamic-Ambient-Occlusion-and-Indirect-Lighting-Bunnell/b09e285522294aab80062b607bfce546aae3a290

[17] JIARATHANAKUL, Prutsdom. Ray Marching Distance Fields in Real-time on WebGL. In: [online]. 2012 [vid. 2023-12-17]. Dostupné z: https://www.semanticscholar.org/paper/Ray-Marching-Distance-Fields-in-Real-time-on-WebGL-Jiarathanakul/a964750aa212bd490d258221bc9756e7e58c5317

[18] MCGUIRE, Morgan. *Graphics Codex* [online]. 2021 [vid. 2023-12-17]. Dostupné z: https://graphicscodex.courses.nvidia.com/app.html?page=_rn_rayMrch

[19] KEINERT, Benjamin, Henry SCHÄFER, Johann KORNDÖRFER, Urs GANSE a Marc STAMMINGER. *Enhanced Sphere Tracing* [online]. B.m.: The Eurographics Association, 2014 [vid. 2023-12-17]. ISBN 978-3-905674-72-9. Dostupné z: doi:10.2312/stag.20141233

[20] GINSBURG, Dan a Budirijanto PURNOMO. *OpenGL ES 3.0 programming guide*. 2nd ed. Upper Saddle River, NJ: Addison-Wesley, 2014. ISBN 978-0-13-344013-3.

[21] MCREYNOLDS, Tom a David BLYTHE. *Advanced Graphics Programming Using OpenGL*. 1st edition. San Francisco, CA: Morgan Kaufmann, 2005. ISBN 978-1-55860-659-3.

[22] VRIES, Joey de. *Learn OpenGL: Learn modern OpenGL graphics programming in a step-by-step fashion.* Erscheinungsort nicht ermittelbar: Kendall & Welling, 2020. ISBN 978-90-90-33256-7.

[23] SHREINER, Dave a The Khronos OpenGL ARB WORKING. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. B.m.: Pearson Education, 2009. ISBN 978-0-321-66927-8.

[24] *Rendering Pipeline Overview - OpenGL Wiki* [online]. [vid. 2023-12-14]. Dostupné z: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

[25] MCCOOL, Michael D, Zheng QIN a Tiberiu S POPA. Shader Metaprogramming. *SIGGRAPH/Eurographics Graphics Hardware Workshop*. 2002, 57–68.

[26] MUKUNDAN, Ramakrishnan. Mesh Tessellation. In: Ramakrishnan MUKUNDAN, ed. *3D Mesh Processing and Character Animation: With Examples Using OpenGL, OpenMesh and Assimp* [online]. Cham: Springer International Publishing, 2022 [vid. 2023-12-15], s. 91–128. ISBN 978-3-030-81354-3. Dostupné z: doi:10.1007/978-3-030-81354-3_5

[27] MUKUNDAN, Ramakrishnan. The Geometry Shader. In: Ramakrishnan MUKUNDAN, ed. *3D Mesh Processing and Character Animation: With Examples Using OpenGL, OpenMesh and Assimp* [online]. Cham: Springer International Publishing, 2022 [vid. 2023-12-15], s. 73–89. ISBN 978-3-030-81354-3. Dostupné z: doi:10.1007/978-3-030-81354-3_4

8 Přílohy

1) Zdrojový kód ray marching aplikace - https://gitlab.com/danpalecek/pgrf2

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání bakalářské práce

Autor:	**Daniel Paleček**
Studium:	I2100258
Studijní program:	B1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název bakalářské práce:	**Vizualizace 3D scény pomocí metody Ray marching**
Název bakalářské práce AJ:	3D scene visualization using the Ray marching method

Cíl, metody, literatura, předpoklady:

Cílem bakalářské práce je prozkoumat a popsat metodu Ray marching jako techniku vizualizace 3D scén. Práce se zaměří na návrh, implementaci a optimalizaci Ray marching algoritmů pro generování vizuálních reprezentací 3D scén s důrazem na geometrické struktury a osvětlení. Dalším cílem je porovnat tuto metodu s tradičními technikami renderování 3D scén a identifikovat její výhody a omezení. Výstupem práce bude implementace vizualizace 3D scény pomocí techniky Ray marching.

Osnova

- Průzkum tématu - principů a metod
- Teoretický přehled
- Analýza a návrh implementace
- Implementace
- Testování a porovnání
- Hodnocení výsledků

- Žára, J., Beneš, B., Sochor, J., Felkel, P. (2004). Moderní počítačová grafika. Česko: Computer Press. ISBN: 9788025104545, 8025104540
- Ginsburg, D., Purnomo, B., Shreiner, D., Munshi, A. (2014). OpenGL ES 3.0 Programming Guide. Velká Británie: Pearson Education. ISBN: 9780133440126, 0133440125
- Akenine-Möller, T., Haines, E., Hoffman, N. (2018). Real-Time Rendering, Fourth Edition. Spojené státy americké: A K Peters/CRC Press. ISBN: 9781351816151, 1351816152

Zadávající pracoviště:	Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu
Vedoucí práce:	Ing. Jakub Beneš
Datum zadání závěrečné práce:	26.1.2021