VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

GPU RAYTRACER PRO OSG

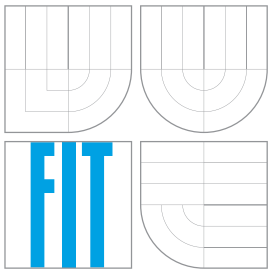
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JIŘÍ KANTOR

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

GPU RAYTRACER PRO OSG

GPU RAYTRACER FOR OSG

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JIŘÍ KANTOR

VEDOUCÍ PRÁCE
SUPERVISOR

ING. TOMÁŠ STARKA

BRNO 2013

Abstrakt

Tato práce popisuje tvorbu jednoduchého raytraceru pro OpenSceneGraph, který běží na grafické kartě. V práci jsou popsány věci, které bylo nutné provést v OpenSceneGraphu, aby bylo možno předávat data do GPU a také několik metod pro hledání průsečíků paprsku a trojúhelníku, což je klíčový algoritmus v raytracingu.

Abstract

This work describes creation of a simple raytracer for OpenSceneGraph, which performs its operations on the graphics card. Things, that needed to be done in OpenSceneGraph in order to pass data to the GPU and also several ray-triangle intersection methods, are described in this work.

Klíčová slova

raytracing, GPU, grafická karta, OpenSceneGraph, OpenGL, GLSL, texture buffer

Keywords

raytracing, GPU, graphics card, OpenSceneGraph, OpenGL, GLSL, texture buffer

Citace

Jiří Kantor: GPU raytracer pro OSG, bakalářská práce, Brno, FIT VUT v Brně, 2013

GPU raytracer pro OSG

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomáše Starky.

.....
Jiří Kantor
14. května 2013

Obsah

Kapitola 1

Úvod

Počítačová grafika je dnes velice dynamicky se rozvíjející odvětví informačních technologií. Snaha o co nejlepší a nejrealističtější zobrazení, a tím pádem i nutnost stále většího množství výpočtů v krátkém čase, žene tvůrce grafického hadrware k výrobě co nejrychlejších a nejflexibilnějších karet a procesorů. V současné době již téměř nelze najít kartu, která by neobsahovala programovatelný grafický procesor, který umožňuje provádět výpočty osvětlení tak, jak si přeje aplikační programátor a nikoliv jak je to definováno obvody karty. Tyto grafické procesory, a vlastně grafické karty celkově, jsou vysoce optimalizovány jednak pro výpočty v pohyblivé řádové čárce, a jednak pro nejpoužívanější operace v grafice: sčítání, odčítání a násobení vektorů, matic, skalární součin, vektorový součin atd. Je tedy nasnadě tuto výpočetní sílu využít pro raytracing, kde jsou tyto operace využívány v hojné míře a je třeba je provádět mnohokrát opakovaně.

V druhé kapitole se věnuji popisu zadání práce, použitým technologiím a návrhu aplikace. Ve třetí kapitole se věnuji popisu implementace. Závěrečná kapitola obsahuje zhodnocení dosažených výsledků a možná budoucí rozšíření.

Na přiloženém CD najdete zdrojové kódy, několik ukázkových modelů, přeloženou Win32 aplikaci i s knihovnami a elektronickou verzi tohoto dokumentu.

Obrázek 1.1: *Scéna vykreslená fixed pipeline*

Obrázek 1.2: *Scéna vykreslená pomocí raytracing a osvětlovacího shaderu*

Kapitola 2

Použité technologie

V této kapitole se budu věnovat popisu zadání, použitým technologiím a návrhu aplikace.

2.1 Popis zadání

Jelikož v samotném zadání není specifikace a cílová aplikace definována přesně, je nutné si říct, jak bude vlastně vypadat.

Cílem této práce je vytvořit knihovnu, která bude spolupracovat s OpenSceneGraphem a bude umožňovat raytracing výpočtem na grafické kartě. Dále je cílem vytvořit demonstrační aplikaci, která bude bude umožňovat vykreslování reflexivních objektů (jako zrcadlo, pochromované objekty…) Bude možno si zvolit, jak budou vykreslovány objekty: osvětlovací model může být počítán v OpenGL fixed pipeline nebo v shaderu a může se to aplikovat buď na všechny objekty nebo jen na odrazivé (samozřejmě, že pokud se aplikuje shader na nereflexivní objekty, tak tím nebudou reflexivní, ale jejich osvětlovací model může být počítán v shaderu). Tímto nám vznikají čtyři možnosti (tabulka 2.1)

Osvětlovací model – fixed pipeline Aplikace shaderů – jen na odrazivé objekty	Osvětlovací model – shader Aplikace shaderů – jen na odrazivé objekty
Osvětlovací model – fixed pipeline Aplikace shaderů – všechny objekty	Osvětlovací model – shader Aplikace shaderů – všechny objekty

Tabulka 2.1: *Shrnutí módů běhu programu*

2.2 OpenGL

OpenGL je multiplatformní API pro zobrazování 2D a 3D grafiky. Tento standard nespecifikuje jak se mají operace provádět, pouze které jsou k dispozici a co mají dělat. Je to jedno z nejrošířenějších API, které je používané v mnoha hrách a jiných grafických aplikacích.

OpenGL je stavový stroj, což znamená, že veškerá prováděná volání jsou ovlivněna právě nastaveným stavem. Tyto stavy kontrolují naprosto vše a jsou klíčovým prostředkem pro ovlivňování vykreslování a vůbec operace OpenGL.

Většina implementací využívá specializovaného hardware, který je optimalizován přímo pro operace používané při zobrazování grafiky (grafické karty). V tomto hardware jsou za

sebou napojené prvky, které tyto operace provádějí zřetězeně - těmto prvkům a způsobu, jak transformují 3D data do 2D rastrového obrazu, se dohromady říká rendering pipeline (jinak též graphics pipeline, zkráceně jen pipeline). Pipeline existují dva základní druhy – fixed–function pipeline a programmable pipeline. Fixed–function pipeline provádí transformační a další grafické operace stále stejným způsobem, který je hardwarově implementován v grafické kartě a nelze jej změnit. Programmable pipeline naopak umožňuje, aby si programátor aplikace část těchto operací sám definoval. Tyto operace jsou definovány v programu pro GPU, kterému se říká shader a tento program je pro OpenGL psán v jazyce GLSL (OpenGL Shading Language)

2.2.1 Pojmy

Nyní si řekneme několik pojmů, které budeme používat v textu v souvislosti s OpenGL

- vertex – bod v prostoru
- polygon – soustava několika vertexů, které tvoří jednu plochu (samozřejmě, že obecně pro n bodů nemůžeme očekávat, že budou všechny ležet v jedné rovině)
- trojúhelník – speciální případ polygonu o třech vertexech
- normála – vektor, který má jednotkovou velikost a v nejjednodušší verzi je kolmý k polygonu, v praxi se častěji používají vertex normály, které jsou definované pro každý vertex a nejčastěji bývají průměrem, nebo něčím podobným, normál okolních polygonů
- fragment – jsou to všechna data, která jsou potřebná pro výpočet konečné hodnoty pixelu (čili se jedná například o polohu vertexu, normálu, texturovací souřadnice, texturu...)
- osvětlovací shader – shader, který počítá osvětlovací model
- raytracing shader – shader, který počítá odrazivost (v podstatě je osvětlovací a raytracing shader spojený, ale pro vysvětlování různých módu běhu programu bude jednodušší rozdělit si je)
- depth buffer – oblast v paměti, kam se ukládají informace o hloubce jednotlivých fragmentů
- sampler – nástroj, který OpenGL a shader používá k získání správného texelu z textury
- texel – jednotka textury, typicky jeden pixel obrázku, který tvoří texturup
- texturovací jednotka – naráz lze mít navázaný jen určitý počet textur, tento počet je limitován počtem texturovacích jednotek (dle specifikace je minimum 8), což jsou hardwarové komponenty v GPU, které se starají o uchování a zpracování textur
- framebuffer – blok paměti, do kterého se ukládají vypočítané hodnoty pixelů (OpenGL nedefinuje jak se mají data zobrazovat, takže veškeré vykreslování je definováno do framebufferu)

http://www.opengl.org

2.3 OpenSceneGraph

OpenSceneGraph je 3D grafická knihovna psaná v C++ a vytvořená jako nadstavba nad OpenGL, která využívá graf scény. Přidává množství funkcionality, která v OpenGL není jako například organizace vertexů do primitiv a objektů, počítání času, automatická správa paměti a jiné. Je možné do ní přidávat podporu dalších typů souborů (ať už obrázků nebo 3D objektů) pomocí zásuvných modulů a tím ještě více rozšířit už tak poměrně velké množství souborových formátů, které podporuje.

Tato aplikace využívá OpenSceneGraph, a rozšiřuje jeho funkcionalitu, aby bylo možné uskutečnit výpočet raytracingu na GPU.

2.3.1 Pojmy

Nyní si řekneme několik pojmů, které budeme používat v textu v souvislosti s OpenSceneGraphem Další info viz [5].

- uzel (node) – základní stavební jednotka grafu scény, je analogická s uzlem v obecné teorii grafů
- state attribute – jedná se o stav OpenGL, například třída `BlendFunc` určuje chování při míchání a je ekvivalentní volání `glBlendFunc`.
- state set – množina state atributů, jeden uzel má jeden state set, ve kterém jsou združeny všechny stavy nastavované pro daný uzel
- drawable – vykreslitelný objekt
- geode (geometry node) – jedná se o uzel, jehož úkolem je seskupovat drawably a obvykle reprezentuje nějaký logický objekt (židle, lopata atd.)
- user data – menší množství dat, které je připojeno k uzlu a dá se vložit například do souboru scény abychom např. poskytli dodatečné informace o objektech

http://www.openscenegraph.org

2.4 Výpočet na GPU

Doba, kdy byla funkcionalita grafických karet dána jejich hardware je pryč. Dnes je grafická karta programovatelná a to nám otevírá možnosti pro výpočet nejen čistě k účelům zobrazování, ale i k počítání čehokoliv jiného. Navíc grafické procesory obvykle mívají více paralelních jader, takže na nich může probíhat více výpočtů současně, čímž se ještě zrychlí.

Pokud chceme provádět pouze výpočty, můžeme použít technologie jako OpenCL a CUDA, které jsou na toto přímo specializované - umožňují provádět na grafické kartě běžné výpočty, které nemusí vůbec souviset se zobrazováním.

Druhá možnost, jak počítat na grafické kartě, je použít shader, což je speciální program, který slouží k řízení jednotlivých částí grafické pipeline. Můžeme tedy pomocí něj řídit výpočet osvětlovacího modelu, dokonce i pozice vertexů. Shader se dělí na dva základní typy: fragment shader a vertex shader. Vertex shader pracuje s vertexy objektu. Jeho výstupem je poloha vertexu na obrazovce (respektive v souřadnicovém systému, jehož počátek je v levém spodním rohu obrazovky). Fragment shader pracuje s jednotlivými rasterizovanými pixely objektu – jeho výstupem je barva pixelu.

Pokud tedy do fragment shaderu budeme schopni předat celou 3D scénu, budeme moci v této scéně hledat průsečíky objektů s paprsky a tím pádem provádět výpočet raytracingu. Toto se v této práci děje.

2.5 Raytracing

Raytracing je metoda zobrazování 3D objektů, při níž sledujeme trasu paprsku vyslaného z kamery přes určitý pixel promítací plochy (tzv. primární paprsek) a zaznamenáváme, s jakými objekty se paprsek protne. Z těchto bodů poté můžeme vyslat další paprsky – například stínové paprsky ke světlům, abychom určili, zda je bod osvětlen nebo ne, nebo můžeme vypočítat úhel odrazu a vyslat paprsek znova do scény abychom určili, co se bude v daném bodě odrážet (tzv. sekundární paprsky). Poté všechny tyto faktory shrneme a vypočítáme, jakou bude mít pixel na promítací ploše barvu.

Tato metoda produkuje velmi kvalitní výsledky, protože je s její pomocí možné jednoduše vypočítat např. stíny nebo zrcadlové odrazy. Bohužel to ale je také jedna z nejpomalejších metod, protože pro každý pixel promítací plochy je potřeba projít celou scénu a spočítat, s kterým objektem se paprsek protne, což je časově dosti náročné (délka výpočtu samozřejmě závisí na složitosti objektu, který chceme vykreslit, efektech, kterých chceme dosáhnout, jako je např. lom nebo odraz, urychlovacích prostředcích pro vyhledávání objektu ve scéně atd.).

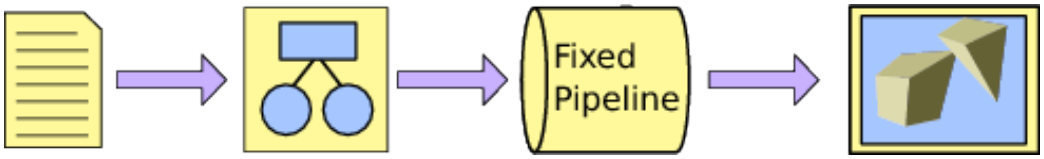
Raytracing je velmi výhodné počítat na grafické kartě a to z toho důvodu, že většina algoritmů pro výpočet průsečíku trojúhelníku a paprsku využívá skalární a vektorové součiny a násobení, vše v plovoucí řádové čárce, na což je grafický procesor optimalizován.

2.5.1 Hybridní raytracing

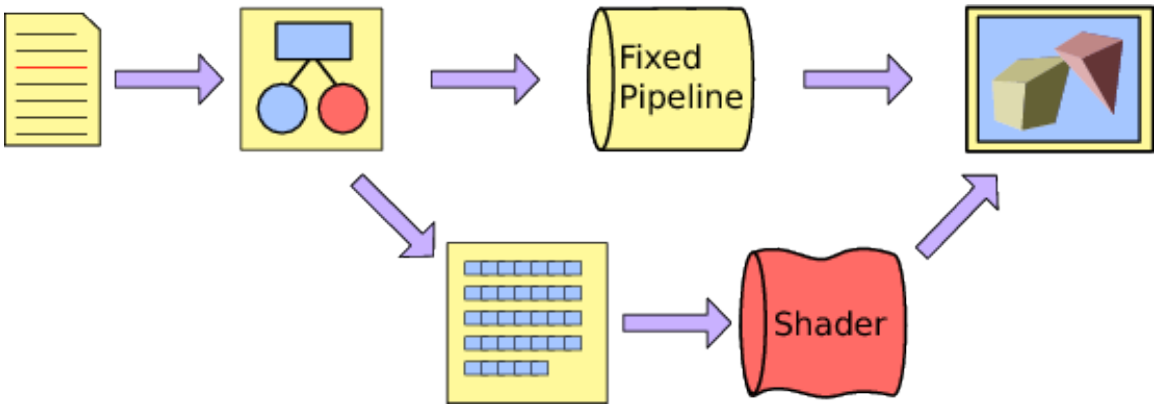
Protože raytracing je časově náročný, nelze si jím zobrazované objekty prohlížet v reálném čase, většinou dokonce ani interaktivně (tzn. že si nelze zobrazeným objektem otáčet apod. a přitom jej mít stále zobrazený pomocí raytracingu). Nabízí se tedy myšlenka, zda by se nedalo raytracingem zobrazit jen takovou část scény, která by bez něj zobrazit nešla a zbytek vykreslit nějakou méně náročnou metodou. A to je přesně to, co budeme v této práci chápat jako hybridní raytracing. Scéna je z většiny vykreslená pomocí rasterizace polygonů (standardní způsob v OpenGL, ale samozřejmě lze výpočet osvětlovacího modelu nechat na shaderu) a odrazivé objekty jsou poté raytracovány. Ale ani ony nejsou raytracovány úplně. Stále necháváme OpenGL rasterizovat i je, ale odrazy poté počítáme v shaderu právě vrháním paprsků do scény a zjišťováním, co se v daném pixelu odráží. Tímto způsobem nám odpádá vrhání primárního paprsku. Z formálního hlediska jej sice pořád bereme v úvahu, ale odpadá nám časově náročná operace hledání průsečíku, protože ten zjistíme právě rasterizací.

Kapitola 3

Analýza a návrh



Obrázek 3.1: *Schéma běžného vykreslování*



Obrázek 3.2: *Schéma vykreslování v této práci*

Je nutné si určit, co vlastně bude vstupem a výstupem aplikace.

Vstupem bude soubor grafu scény, ve kterém budou označené ty objekty, které chceme vykreslit jako odrazivé. Výstupem potom bude tato scéna vykreslená i s odrazivými objekty.

Mezi těmito dvěma body bude ležet několik kroků:

1. Načtení grafu scény
2. Identifikace odrazivých objektů
3. Posbírání dat scény pro raytracing
4. Úprava grafu scény pro dosažení správného vykreslení
5. Předání formátovaných dat scény do GPU

6. Vykreslení scény

Tyto kroky si nyní popíšeme detailněji.

3.1 Načtení scény a identifikace odrazivých objektů

Načtení grafu scény provede OpenSceneGraph. Graf scény poté projdeme a zjistíme, které objekty mají nastavenou user data `reflectiveCoefficient` na nenulovou hodnotu.

3.2 Úprava grafu scény

Ač je vykreslení odrazů pomocí hybridního raytracingu rychlejší než vykreslovat celou scénu pomocí klasického raytracingu, pořád to není tak rychlé jako prostá rasterizace. Navíc se velice jednoduše může stát, že pracně vyraytracujeme odrazivý objekt a potom nám jej kompletně překryje nějaký jiný, takže by vykreslování trvalo dlouho a všechny výpočty by byly k ničemu.

Jak tento problém vyřešit? OpenGL standardně řeší viditelnost objektů pomocí depth testu. Pokud je nově vykreslovaný fragment hlouběji ve scéně (=dále od kamery) než nějaký jiný na tom stejném pixelu, je jasné, že onen nový neuvidíme, protože bude překryt tím, co už tam je. OpenGL tedy zbytečně nepočítá osvětlovací model a další věci nutné pro zobrazení a rovnou tento nový fragment zahodí. Mimo jiné se také stane to, že pro tento fragment není zavolán fragment shader. To je pro naše potřeby ideální. Pokud bude odrazivý objekt překryt jiným, nespustí se fragment shader a tím pádem se nebude ani provádět náročný výpočet raytracingu odrazu.

Vykreslování tedy rozdělíme do dvou fází. V první fázi se vykreslí celá scéna i s odrazivými objekty, ale odrazy se nebudou vykreslovat. V druhé fázi se vykreslí odrazy a výsledek se smíchá s výsledkem první fáze. Správného smíchání se dosáhne nastavením OpenGL a správným nastavením alfa kanálu pixelů, ze kterých se skládá odraz.

OpenGL tedy provede rasterizaci a test viditelnosti a poté pro jednotlivé fragmenty spustí fragment shader. V závislosti na vybraném módu programu se fragment shader spustí pro různé objekty.

- Osvětlovací model počítán ve fixed pipeline
  - Raytracing shader aplikován jen na odrazivé objekty
    Fragment shader se spustí jen pro fragmenty odrazivých objektů a nebude se v něm počítat osvětlovací model, jen odraz.
  - Raytracing shader aplikován na všechny objekty
    Fragment shader se spustí pro fragmenty všech objektů ve scéně, ale protože je osvětlovací model počítán ve fixed pipeline, neprovede se u neodrazivých objektů žádný výpočet.
- Osvětlovací model počítán shaderem
  - Raytracing shader aplikován jen na odrazivé objekty
    Fragment shader se spustí jen pro fragmenty odrazivých objektů a počítá se odraz i osvětlovací model.

– Raytracing shader aplikován na všechny objekty

  Fragment shader se spustí pro fragmenty všech objektů ve scéně, takže u odrazivých objektů se bude počítat osvětlovací model i odraz a u ostatních se bude počítat osvětlovací model.

3.3 Sběr dat scény a předání do GPU

Abychom mohli počítat odrazy apod., potřebujeme vědět, jak vypadá scéna, kterou chceme vykreslit. Bohužel není žádný způsob jak se ze shaderu, který běží na grafické kartě, za běhu dostat do hlavní operační paměti počítače. Byla by to časově velice náročná operace, protože by se musela data přenáše přes sběrnici a tím pádem by se celé vykonávání shaderu velice zpomalilo. Naštěstí má grafická karta svou vlastní paměť, do které se ukládají textury, vertexy, normály, prostě vše, co je třeba pro vykreslení scény. Nicméně ze shaderu se obvykle můžeme dostat jen na informace o aktuálně kresleném vertexu nebo fragmentu.

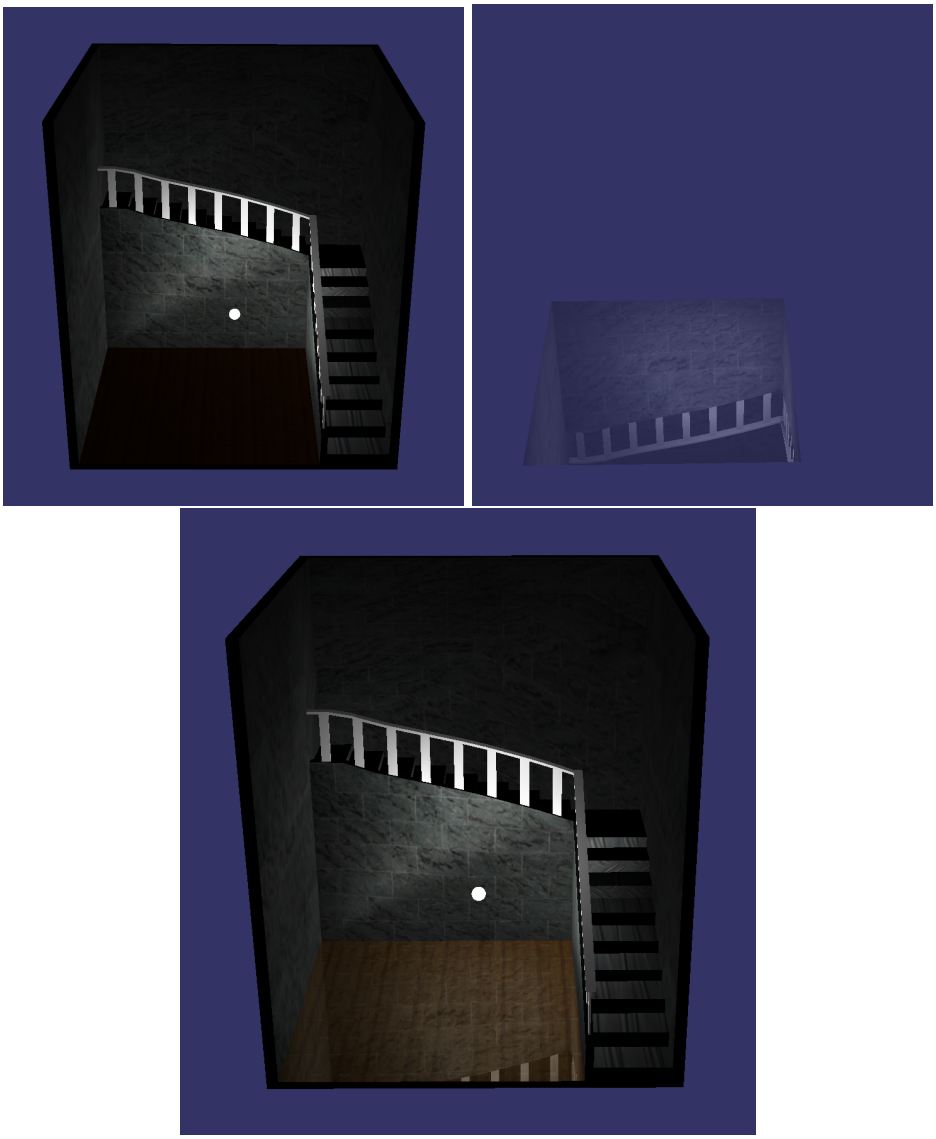
Předávání dat do GPU není tak jednoduché, jak by se mohlo na první pohled zdát. V shaderu sice můžeme z našeho programu nastavit hodnoty určitých proměnných (typu *uniform*), ale jejich maximální velikost je omezená (řádově desítky kB). Data, která mají značnou velikost a do shaderu se předávají naprosto běžně, jsou textury. Zdálo by se tedy, že naši scénu můžeme serializovat a předat do shaderu jako texturu. Bohužel, běžné textury v shaderu limitují svůj obsah na hodnoty $\in \langle 0.0, 1.0 \rangle$. Pro data o scéně, kde jsou např. pozice vertexů, které dozajista přesahují tyto limity, je to nevhodné.

Řešení představuje *texture buffer* (nebo buffer texture). Je to jednorozměrná textura, která svá data bere z nějakého buffer objectu, což je kus paměti, ve kterém OpenGL ukládá větší množství dat a u kterého se počítá, že se bude přenášet do paměti grafické karty. Texture buffer svá data nijak neupravuje, čili v shaderu je přečteme taková, jaká jsme je do něj vložili v OpenGL.

Avšak data, která se do shaderu předávají nejsou strukturovaná, tzn. že v nich nelze jednoduše vytvářet například hierarchické struktury. Musíme najít nějaký způsob, jak tuto hierarchičnost „zploštit“. Je nutné pro všechny objekty zjistit, jaké mají polygony, a převést je na trojúhelníky. Také je nutné zjistit, jaké mají objekty normály, texturovací souřadnice, texturu, materiál atd. Tyto informace je poté třeba zformátovat tak, aby se daly zapsat jako pole čísel s plovoucí řádovou čárkou. Toto pole poté předáme texture bufferu jako data. Pro OpenGL a shader to bude vypadat, jakoby se do shaderu předávala jednorozměrná textura. V shaderu poté použijeme speciální sampler, který data čtená z textury neomezuje do rozsahu $\in \langle 0.0, 1.0 \rangle$.

3.4 Raytracing na GPU

Raytracing na GPU dělá to, že počítá barvů pixelů odrazivého objektu. U daného pixelu se podle pozice kamery vypočítá, v jakém směru leží objekt, který se v něm bude odrážet, a poté hledá, který objekt to vlastně je. Pokud najde další odrazivý objekt, je možné, aby sledoval další odražený paprsek a hledal, co se odráží v tomto odraženém objektu - toto se dá opakovat mnohokrát. Pro každý bod je samozřejmě normálním způsobem vypočtená barva a s odrazem je poté smíchána podle koeficientu odrazivosti.



Obrázek 3.3: *Obrázek vlevo nahoře: Vykreslen osvětlovací model, obrázek vpravo nahoře: Raytracovaný odraz (modrá barva je jen pozadí a na výslednou barvu odrazu nemá vliv, protože výsledné pixely barvy odrazu se míchají rovnou s pixely osvětlovacího modelu) obrázek dole: Smíchané kroky*

Kapitola 4

Implementace

Práce je implementována částečně v jazyce C++ a částečně v jazyce GLSL (OpenGL Shading Language - jazyk pro psaní shaderů v OpenGL). Vývoj jsem prováděl pod Ubuntu Linuxem, ale jak OpenGL tak OpenSceneGraph jsou multiplatformní, takže není problém přeložit aplikaci jak na Windows tak na Linuxu. Jako vývojové prostředí jsem používal Eclipse.

4.1 Identifikace odrazivých objektů

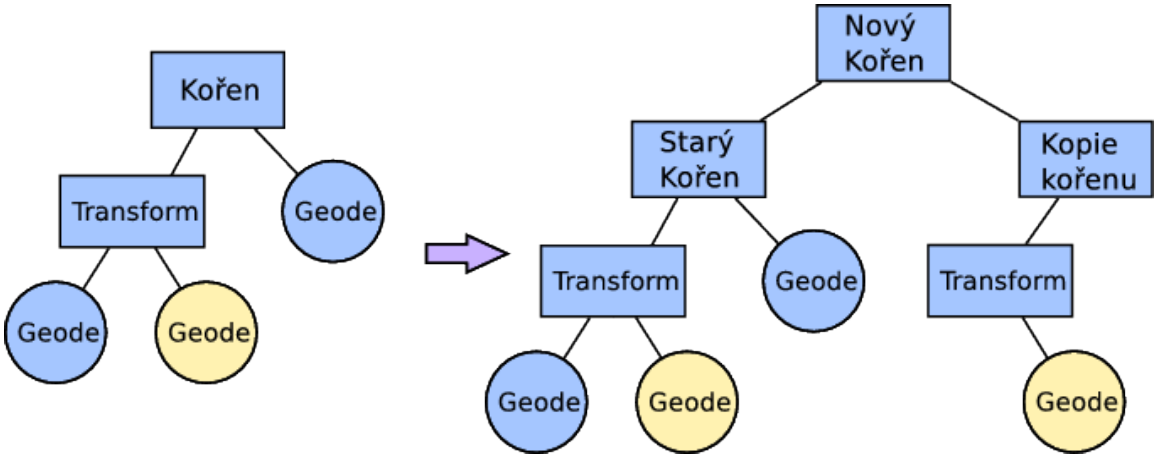
Odrazivé objekty jsou v soubou scény identifikovány pomocí tzv. UserData. Je to způsob, jak předat do grafu nějaké údaje, které můžeme poté v programu zpracovávat. Co se týče raytracingu v této práci, tak stačí do souboru typu `.osgt` (s `.osg` to nefunguje) k nějaké node přidat řádky

```
UserDataContainer TRUE {
  osg::DefaultUserDataContainer {
    UniqueID 25
    UDC_UserObjects 1 {
      osg::FloatValueObject {
        UniqueID 26
        Name "ReflectionCoefficient"
        Value 0.95
      }
    }
  }
}
```

kde Value je požadovaná hodnota koeficientu odrazivosti a UniqueID jsou unikátní identifikátory objektů pro OSG. Pokud máme vytvořený `.osgt` soubor a chceme některý soubor označit jako odrazivý, stačí se podívat, jaké UniqueID je nejvyšší a poté v UserData nastavit další (v ukázce bylo předtím nejvyšší 24).

4.2 Úprava grafu scény

Abychom dosáhli požadovaného vykreslení, je nutné poněkud změnit graf scény (pro znázornění viz obrázek 4.1). V prvé řadě musíme vytvořit nový kořen grafu a jako potomka



Obrázek 4.1: *Znázornění úpravy grafu scény (žlutý uzel reprezentuje odrazivý objekt)*

mu přidat původní graf scény. Jako druhého potomka mu přidáme zpracovaný graf scény – odrazivý graf. Ten vytvoříme tak, že začneme s prázdným grafem a postupně do něj přidáváme všechny odrazivé objekty.

OpenGL je stavový stroj a to znamená, že vykreslování je ovlivněno aktuálně nastaveným stavem. OpenSceneGraph tyto stavy ukládá pro každý uzel grafu v tzv. state atributech (třída `StateAttribute`, resp. její podtřídy) a při vykreslování prochází graf scény a nastavuje stavy podle toho, jak jsou nastaveny v uzlech. Standardně stavy platí pro daný uzel a celý podgraf, jehož je kořenem, ale vše záleží na nastavení konkrétního state atributu.

Je tedy nutno nastavit několik state atributů. Jedná se o:

- Render bin (třída `RenderBin`)

  OpenSceneGraph má podporu pro vykreslování v několika průbězích, což je přesně to, co potřebujeme. Render bin má nastavené číslo a toto číslo určuje, kolikátý se bude daný podgraf vykreslovat. Našemu původnímu grafu tedy nastavíme Render bin na 1 a odrazivému grafu na 2.

- Depth test (třída `Depth`)

  OpenGL používá ke správnému vykreslení překrývajících se objektů depth test neboli hloubkový test. Kromě informace o barvě si uchovává u každého pixelu i informaci o hloubce, tj. vzdálenosti od kamery. Pokud nově rasterizovaný fragment má hloubku menší než ten, který už uložený je, znamená to, že nový fragment je blíže kameře než uložený a tím pádem bude nový viditelný, protože uložený překrývá, a tudíž je tento nový fragment poslán k výpočtu osvětlení atd. V základu je depth test nastaven na LESS, čili že nový fragment se vykreslí, když je blíž než uložený. Avšak my vykreslujeme dvakrát a potřebujeme, aby se vykreslil i podruhé. Depth test tedy musíme nastavit na LESS_OR_EQUAL (`LEQUAL` v OpenSceneGraphu), což znamená, že fragment bude poslán k výpočtu osvětlení i v případě, že už uložený fragment má stejnou hloubku jako nový. Odrazivému grafu tedy nastavíme depth test na LESS_OR_EQUAL.

- Blending (třídy `BlendEquation` a `BlendFunc`)

  Samozřejmě, že u překrývajících se fragmentů není jediná možnost jeden z nich odstranit. Je možné jejich barvy také smíchat, čehož využijeme v této práci.

Pro dosažení takového míchání, jaké potřebujeme, je třeba nastavit dvě věci: jaký výpočet se vlastně bude provádět a odkud se berou váhové koeficienty. To, jaký výpočet se bude provádět, určíme nastavením blend equation na některou z hodnot v tabulce 4.1. Odrazivému grafu nastavíme BlendEquation na ADD.

Odkud se berou váhové koeficienty se nastavuje zvlášť pro source a destination (source je označení pro nový fragment, destination pro uložený) a možných nastavení je mnoho. My si vybereme, že váha source fragmentu bude SOURCE_ALPHA a destination fragmentu ONE_MINUS_SOURCE_ALPHA. Tím pádem budeme moci nastavením alfa hodnoty fragmentu ve fragment shaderu regulovat, v jakém poměru se budou míchat, takže čím vyšší hodnotu bude mít alfa nastavená ve fragment shaderu, tím odrazivější bude objekt.

Hodnota blend equation	Prováděný výpočet
`FUNC_ADD`	$C_{src}W_{src} + C_{dst}W_{dst}$
`FUNC_SUBTRACT`	$C_{src}W_{src} - C_{dst}W_{dst}$
`FUNC_REVERSE_SUBTRACT`	$C_{dst}W_{dst} - C_{src}W_{src}$
`MIN`	$min(C_{src}, C_{dst})$
`MAX`	$max(C_{src}, C_{dst})$

Tabulka 4.1: *Možné hodnoty blend equation a odpovídající prováděné výpočty. C_{src} je barva nového fragmentu, W_{src} je váhový koeficient nového fragmentu, C_{dst} je barva uloženého fragmentu a W_{dst} je váhový koeficient uloženého fragmentu (source je označení pro nový fragment, destination pro uložený). Tabulka převzata ve zjednodušené formě ze specifikace OpenGL.*

- Raytracing shader (třídy `Program` a `Shader`)

  OpenSceneGraph bere shader také jako state attribut. Je to logické, protože ho lze pro různé objekty zapnout nebo vypnout. Shader je program, takže je třeba jej přeložit a slinkovat jako každý jiný program. Pokud kompilace nebo linkování selže, OpenSceneGraph automaticky vypíše chybový záznam z OpenGL. Bližší popis shaderu je v sekci 4.5

- Texture buffer (třída `TextureBuffer`)

  Texture buffer se ukázal být poněkud problémem, protože OpenSceneGraph zatím podporu texture bufferu neobsahuje, takže bylo nutné ji doimplementovat. Nebylo to úplně triviální a bylo třeba provést určité změny v samotném OpenSceneGraphu, aby se do něj dalo texture buffer nějak hladce zakomponovat. Nepovedlo se to úplně, ale po nějakých úpravách by asi bylo možno přidat jej do oficiálních zdrojových kódů OpenSceneGraphu. Třída je potomkem třídy `Texture`, která je potomkem třídy `StateAttribute`. Bližší popis v sekci 4.4.

- Textury objektů scény

  Textury všech objektů scény jsou v této práci předány jednoduchým, i když ne ideálním způsobem: v odrazivém grafu jsou navázány textury všech otexturovaných objektů najednou. Větší množství textur se takto samozřejmě předat nedá, ale minimum

texturovacích jednotek je 8, dle specifikace OpenGL, což pro menší scény postačuje. Lepší by tedy bylo textury předat také v texture bufferu, nicméně tak by bylo třeba počítat ručně který texel vybereme podle texturovacích souřadnic. Podle specifikace rozšíření je maximální velikost texture bufferu aspoň 128MB. S tímto limitem by tedy bylo třeba počítat, ačkoliv prakticky je to mnohem víc (často to bývá omezeno pouze velikostí texturovací paměti).

4.3 Příprava dat pro GPU

Abychom mohli provádět na GPU raytracing, je nutné do GPU předat informace o tom, jak scéna vlastně vypadá.

4.3.1 Formát dat

Texture buffer je jednorozměrná textura a v shaderu jsou hodnoty z něj vraceny jako čtyřprvkové vektory s prvky v plovoucí řádové čárce. Celou scénu je tedy nutno uložit do jednoho velkého pole čísel s plovoucí řádovou čárkou a tato data poté předat do texture bufferu a do GPU. V shaderu je poté nutno k datům přistupovat po čtveřicích. Z toho důvodu jsou také veškeré datové struktury v tomto poli zarovnané na 4 čísla.

Samozřejmě by šlo toto zarovnání porušit a data zkomprimovat co nejvíce by to šlo. Například vertexy a normály by stačilo ukládat po 3 prvcích a ne po čtyřech (normála je vektor tak jako tak a s body v nekonečnu nepočítáme), struktury by také šlo poněkud zmenšit tím, že by se odstranily výplňové nuly. Nicméně v tomto projektu jsem i tak narazil na dost drobných chyb a nejasností, které jsem musel řešit a rozhodně jsem si nechtěl přidávat ještě další možné chyby v počítání toho, který vektor musím přečíst, abych se dostal na x-tý trojúhelník y-tého objektu.

Aby v takovýchto datech nebyl chaos, je třeba jim samozřejmě dát nějakou strukturu. Ta je ukázána v tabulce 4.2.

Světla	Materiály	Objekty	Vertexy	Normály	Texturovací souřadnice

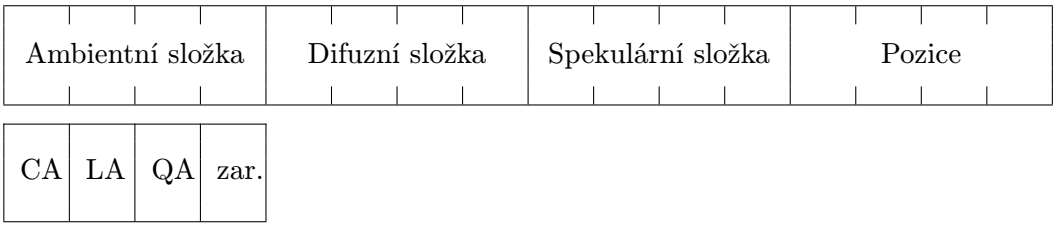
Tabulka 4.2: *Rozmístění dat v texture bufferu*

Každá ze sekcí reprezentuje část pole čísel v plovoucí řádové čárce. Jejich hranice jsou předány do shaderu jako `uniform` proměnné celočíselného typu, které určují na které čtveřici daný blok začíná a také kolik daný blok obsahuje prvků.

- `materialsStartOffset` udává start bloku materiálů
- `drawablesStartOffset` udává start bloku objektů
- `vertexStartOffset` udává start bloku vertexů
- `normalStartOffset` udává start bloku normál
- `texCoordsStartOffset` udává start bloku texturovacích souřadnic
- `numLights` udává počet světel (světla začínají na nule)
- `numMaterials` udává počet materiálů

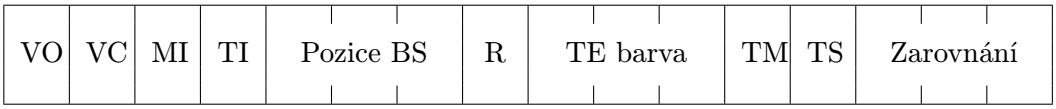
- `numDrawables` udává počet objektů

Bloky jsou vlastně pole struktur, které reprezentují daný prvek. Struktury vypadají následovně (jedna buňka je jedno číslo v plovoucí řádové čárce):



Obrázek 4.2: *Struktura pro světlo. CA je constant attenuation, LA linear attenuation, QA quadratic attenuation a zar. je zarovnání na násobek 4 čísel*

Obrázek 4.3: *Struktura pro materiál. S je shininess*



Obrázek 4.4: *Struktura pro objekt. VO je index prvního vertexu tohoto objektu v poli vertexů a normál, VC je počet vertexů a normál tohoto objektu, MI je index materiálu, TI je index textury, Pozice BS je pozice bounding sphere, R je její poloměr, TE barva je texture environment barva, která se využívá v některých módech k míchání materiálu a textury TM je mód míchání materiálu a textury a TS je index do prvního vertexu tohoto objektu v poli texturovacích souřadnic.*

Jelikož datové typy `float` a `integer` mají shodnou velikost (4 byty), je možné do datového pole zapsat jak jeden, tak i druhý. Má to tu výhodu, že informace, které mají větší smysl předávat jako celočíselné (indexy, počty), lze předat přímo jako integer. V C++ se zapsání celého čísla na místo čísla v plovoucí řádové čárce dá dosáhnout jednoduchým přetypováním ukazatele. V shaderu pak je k dispozici funkce `floatbitstoint`, která přijme jako argument číslo v plovoucí řádové čárce a vrátí celé číslo reprezentované stejnými byty. Vyhneme se tak nepříjemnostem pramenícím z nepřestnosti čísla v plovoucí řádové čárce. K tomuto opatření jsem přistoupil poté, co jsem nabyl silného dojmu, že to je to, co způsobuje problémy, které jsem měl. Posléze jsem zjistil, že jsou způsobeny něčím jiným, ale ze sémantického hlediska mi přišlo jako dobrý nápad to nechat takto.

Pole vertexů, normál a texturovacích souřadnic jsou uložena po trojúhelnících. To znamená, že za sebou jdoucí trojice vždy reprezentují jeden trojúhelník. Tento způsob je sice poněkud náročnější na paměť, ale zase je o něco rychlejší, protože zde odpadá nutnost dvojího přístupu do texturovací paměti pro zjištění jednoho vertexu (tak jak to je je nutno přistoupit jen jednou a rovnou získáme vertex, kdežto při uložení trojúhelníků přes indexy

by bylo nutno přistoupit jednou pro zjištění indexu a podruhé pro zjištění vertexu samotného). Vertexy a normály jsou uloženy jako čtyřprvkové vektory, texturovací souřadnice jsou uloženy jako dvouprvkové, čili v každém čtyřprvkovém vektoru (z textury jde v shaderu číst jenom po čtyřprvkových vektorech) jsou uloženy dvě texturovací souřadnice.

4.3.2 Konverze dat

Abychom si zjednodušili práci při serializaci scény, překonvertujeme si nejdříve data scény do vlastního tvaru. V OpenSceneGraphu jsou totiž data uložena za poměrně velkým množstvím struktur a navíc je nejlepší procházet grafem pomocí potomka třídy `NodeVisitor`, který postupně prochází všechny uzly a u každého zavolá metodu `apply` s odpovídajícím typem uzlu. Nás zajímají uzly typu `Geode` (to je uzel, který obsahuje uzly typu `Drawable`, které obsahují vertexy a polygony) a `LightSource`. Projdeme tedy všechny uzly typu `Drawable` nalezené ve všech uzlech typu `Geode` a data v nich uložená konvertujeme do objektu třídy `ProcessedDrawable`. Abychom si vysvětlili jak tato třída zapadá do sběru dat, je nejprve nutné si vysvětlit, jak jsou data objektu v OpenSceneGraphu typicky uložena.

OpenSceneGraph využívá klasický způsob uložení, kdy máme pole vertexů, normál, texturovacích souřadnic a polygony jsou uloženy jako soustava indexů do těchto polí. OpenSceneGraph navíc při načítání modelu tyto pole vytváří tak, že do pole vertexů, normál i texturovacíh souřadnic je stejný index. Chceme–li projít všechna tato data po trojúhelnících, je vhodné použít třídu `TriangleIndexFunctor`. Tato třída je šablonová a ve třídě, která je jí předána volá metodu `void operator()( const int i1 , const int i2 , const int i3 )` pro každý trojúhelník. Pokud se v objektu nachází nějaký polygon, který trojúhelníkem není, převede ho tato třída na několik trojúhelníků a pro každý zavolá výše uvedenou metodu. Tu je třeba implementovat ve třídě, kterou předáváme do šablony a `i1`, `i2` a `i3` jsou právě indexy do polí vertexů, normál a texturovacích souřadnic. Navíc `TriangleIndexFunctor` z předané třídy dědí, takže zároveň s ním je vytvořen objekt předané třídy. V této práci je předávána třída `ProcessedDrawable`, která ukládá data o objektu a poté je použita pro serializaci.

Každý objekt musí mít nějaké body a také, jelikož uvažujeme vyrobitelné (manifold) objekty, normály, protože je nutno počítat osvětlovací model a to je bez normál nemožné. Ne každý objekt ale musí mít texturu a texturovací souřadnice. Pokud na takový narazíme, nemá pole texturovacích souřadnic a v serializovaných datech tím pádem nezabírá žádné místo. Jeden z problémů, nad kterými jsem si lámal hlavu bylo, že jsem si toto neuvědomil a indexy do pole texturovacích souřadnic jsem počítal, jakoby každý objekt měl stejně texturovacích souřadnic jako vertexů a normál. To se projevovalo tak, že některé objekty měly na celém svém povrchu texturovací souřadnice $(0, 0)$ a tím pádem měl celý objekt barvu horního levého texelu. Potom, co jsem si to uvědomil, jsem změnil způsob počítání indexu a chybu opravil.

V průběhu sběru dat je samozřejmě třeba zjistit i různé vlastnosti objektů – materiály, jakou texturu používají apod. Tyto informace jsou uloženy ve `StateSet`u uzlů. Abychom ale získali plný state set, ve kterém jsou vzaty v úvahu i state attributy z vyšších uzlů, je nutné provést sloučení stavů. OpenSceneGraph k tomuto nabízí přímo metodu `merge`, která hierarchicky sloučí stavy dvou uzlů – jeden, na kterém se volá, a druhý, který se jí předá jako argument. Není jedno, který je který, protože se potom počítá s různými nastaveními state attributů (OVERRIDE, PROTECTED apod.).

4.4 Předání dat do GPU

K předání serializovaných dat o scéně do shaderu použijeme technologii zvanou texture buffer. Jedná se o jakýsi hybrid mezi texturou a buffer objectem v OpenGL. Z pohledu shaderu to je jednorozměrná textura, která může mít ve svých texelech uloženou jakoukoliv hodnotu. Z pohledu OpenGL to je jak buffer tak textura, protože pro jeho vytvoření je třeba alokovat jak buffer object tak texturu a pak je svázat a naplnit daty.

Nejdříve si popíšeme postup v čistém OpenGL a poté v OpenSceneGraphu. OpenSceneGraph totiž kolem buffer objectů a textur vytváří ještě poměrně silnou vrstvu, ve které provádí vlastní management těchto zdrojů.

4.4.1 Postup v OpenGL

- Pro buffer

    1. Zavolat metodu `glGenBuffers`, kterou knihovně řekneme, že chceme vytvořit unikátní identifikátor (popřípadě identifikátory, takto se dá generovat více buffer objectů najednou) a datové struktury pro buffer.
    2. Pomocí `glBindBuffer` buffer navázat, čímž řekneme, že daný typ bufferu (což je první argument, který se funkci předá), který bude v tomto případě `GL_TEXTURE_BUFFER`, je svázán s objektem, jehož identifikátor předáme jako druhý argument. V jeden okamžik může být pro jeden typ navázán jen jeden buffer.
    3. Do bufferu předat data voláním `glBufferData`, kterému předáme argumenty typ bufferu (`GL_TEXTURE_BUFFER`), velikost dat, ukazatel na data a použití, které umožňuje OpenGL optimalizovat přenosy dat apod.

- Pro texturu

    1. Zavolat metodu `glGenTextures`, kterou knihovně řekneme, že chceme vytvořit unikátní identifikátor (popřípadě identifikátory, takto se dá generovat více textur najednou) a datové struktury pro texturu.
    2. Pomocí `glBindTexture` texturunavázat, čímž řekneme, že daný typ textury (což je první argument, který se funkci předá), který bude v tomto případě `GL_TEXTURE_BUFFER`, je svázán s objektem, jehož identifikátor předáme jako druhý argument. V jeden okamžik může být pro jeden typ navázána jen jedna textura.
    3. Svázat buffer a texturu pomocí metody `glTexBuffer`, které předáme typ (`GL_TEXTURE_BUFFER`), způsob uložení dat (`GL_RGBA32F`, tím řekneme, že chceme čtyřkanálovou texturu – RGBA – a že každý kanál má být 32bitové číslo v plovoucí řádové čárce – 32F).

Nakonec zavoláme `glBindBuffer` a `glBindTexture` a u obou na cíl `GL_TEXTURE_BUFFER` navážeme 0, čili žádný objekt, abychom neovlivňovali další zobrazování dokud to nebudeme chtít.

Abychom poté dostali data do shaderu, stačí nám u vykreslování zavolat `glBindTexture` s identifikátorem vytvořené textury. Musíme si samozřejmě dát pozor, kterou máme aktivní texturovací jednotku.

Abychom v shaderu měli na sampler, přes který chceme k texture bufferu přistupovat, navázanou správnou texturovací jednotku, je třeba předat do shaderu `uniform` celočíselnou

proměnnou, která se bude jmenovat stejně jako proměnná sampleru v shaderu, a jako hodnotu ji nastavit texturovací jednotku, ve které texture buffer je.

Předání `uniform` proměnné do shaderu provedeme následujícím způsobem:

1. Zavoláme funkci `glGetUniformLocation`, které předáme id shaderu a jméno proměnné, jejíž umístění v programu chceme získat. Funkce nám vrátí identifikátor `uniform` proměnné.

2. Funkcí `glUniform1i`, které předáme identifikátor, který jsme získali v předchozím kroku, a hodnotu, kterou chceme do proměnné uložit, nastavíme sampleru správnou texturovací jednotku.

4.4.2 Postup v OpenSceneGraphu

Nejdříve si popíšeme, jak v OpenSceneGraphu vytvoříme a svážeme texturu a buffer object.

OpenSceneGraph si, podle všeho, textury a buffer objecty spravuje z velké části sám. Nasvědčuje tomu přítomnost tříd jako `TextureObjectManager`. Tato třída má statickou metodu, kterou je možné získat její instanci. Získáme tedy instanci `TextureObjectManageru` a požádáme jej o vygenerování nové textury (typu `GL_TEXTURE_BUFFER`).

S buffer objectem je to trochu složitější. Bufferů je stejně jako textur několik typů. Texture buffer je jeden z nich, takže si musíme vytvořit novou třídu `TextureBufferObject`, která nám bude zajišťovat, že se buffer object vytvoří se správným typem (nastaví se v konstruktoru) a také, správu dat (metoda `setData`) – v tomto případě nebudeme věci komplikovat a uděláme to tak, že buffer object bude moct mít jen jeden spojitý blok dat a pokud se pokusíme přidat nějaká další data, tak se prostě ukazatel na data přepíše a místo starých dat bude texture buffer obsahovat nová data.

Vytvoříme tedy nový `TextureBufferObject` a pomocí `setData` mu nastavíme data, která chceme, aby předával. Zatím se v OpenGL ještě nic neděje. Poté na takto vytvořeném objektu zavoláme funkci `getOrCreateGLBufferObject`, která vygeneruje nový OpenGL buffer object (někdy v průběhu volání této funkce může být zavolána funkce `glGenBuffers`) nebo použije nějaký starý, již nepoužívaný (tuto funkcionalitu přidává právě OpenSceneGraph, který si toto sám spravuje) – vznikne objekt třídy `GLBufferObject`.

Na tomto objektu poté zavoláme funkci `compileBuffer`, která jednak buffer naváže na cíl (`glBindBuffer`) a předá do něj data (`glBufferData`). Potom navážeme texture object zavoláním jeho funkce `bind`.

Nyní zavoláme přímo metodu `glTexBuffer` a předáme ji stejné argumenty jako v OpenGL verzi. Poté stačí odvázat buffer object (zavolat `glBindBuffer` s identifikátorem 0), protože ten navázaný mít nepotřebujeme.

Nyní si popíšeme, jak se tento postup do OpenSceneGraphu zakomponuje.

Jeden z problémů je, že funkce `glTexBuffer` není v žádné třídě zapouzdřena, protože OpenSceneGraph texture buffer nepodporuje. Potřebujeme–li funkci, která není v OpenGL základní specifikaci, ale je to rozšíření (naprosté minimum jen těch základních funkcí je v základní specifikaci, protože OpenGL se rozšiřuje tím, že některé rozšíření potvrdí a řekne, že jsou součástí OpenGL), je dobré to udělat standardním způsobem, jakým se to v OpenSceneGraphu dělá. To je tak, že ve třídě, která tyto funkce využívá, se vytvoří vnitřní třída (nested class) `Extensions`, která obsahuje ukazatele na funkce, které potřebujeme. Tyto ukazatele se získávají pomocí funkce `setGLExtensionFuncPtr`, která různými způsoby načte dynamickou knihovnu OpenGL a zjistí ukazatel podle toho, na kterém operačním systému je spuštěna. My potřebujeme funkci `glTexBuffer` a proto ji předáme jako

první jméno. Jako druhé jméno (tzv. fallback – pokud se nenajde funkce prvního jména, je proveden pokus o nalezení funkce druhého jména) předáme `glTexBufferEXT`, což je jméno té samé funkce, akorát ještě předtím, než byla uznána za součást OpenGl a bylo to pouze rozšíření.

Texture buffer jako celek budeme dědit od třídy Texture, protože je jí ve značné míře podobný. Navíc budeme schopni přidat TextureBuffer jako state attribut k jakémukoliv uzlu. State attributy mají několik metod, které se volají při průchodech grafem scény. Dvě, které budeme potřebovat, jsou `apply` a `compileGLObjects`.

Metoda `apply` se volá pokaždé, když je třeba daný state attribut aplikovat, tedy když je třeba nastavit stav, který tento state attribut reprezentuje. V našem případě to znamená, že je třeba navázat texturu, abychom k ní měli v shaderu přístup.

Metoda `compileGLObjects` se volá na začátku provádění programu a je určena k tomu, aby se v ní vytvořily OpenGL objekty, které budeme potřebovat. V této funkci tedy provedeme výše uvedený postup vytvoření textury, buffer objectu a jejich svázání.

S třídou `TextureBuffer` jsem narazil na několik problémů

- Jako svůj typ vrací `GL_TEXTURE_BUFFER`. Tento typ se někde uvnitř OpenSceneGraphu zjišťuje a předává do nějaké funkce OpenGL, ve které se ovšem jedná o neplatnou hodnotu. OpenSceneGraph totiž při každém vykreslení vypisuje OpenGL chybu 'invalid enumerant'. Tuto chybu se mi nepodařilo vystopovat a odstranit, ale zdá se, že na běh aplikace nemá žádný vliv.

- State attributy se v OpenSceneGraphu dělí na texturní a ostatní. Při přidávání state attributů do state setu se kontroluje, zda se nepředává texturní argument přes funkci, která přidává ostatní nebo naopak (funkce `setAttributeAndModes` a `setTextureAttributeAndModes`). Pokud ano, tak se vypíše varování a zavolá se druhá funkce. Zde nastávají dva problémy:

  1. Ve funkci `setTextureAttributeAndModes` se provádní trohu jiné úkony, než v `setAttributeAndModes`, takže pomocí `setAttributeAndModes` se nedá přidat textura.
  2. To, zda je zavolána správná funkce, se kontroluje a pokud je zavolána špatná, tak OpenSceneGraph vypíše varování a zavolá správnou funkci. U textur se kontroluje, zda jsou to skutečně textury a to pomocí jejich typu.

  To, zda daný typ je textura je definováno v OpenSceneGraphu ve třídě `TextureGLModeSet`. Samozřejmě, že hodnota `GL_TEXTURE_BUFFER` tam není, protože se s ní v OpenSceneGraphu vůbec nepočítá a bohužel naprosto není způsob, jak za běhu programu do těchto definic něco dodat. Potřebujeme zavolat metodu `setTextureAttributeAndModes`, takže jediné řešení je ve zdrojových kódech OpenSceneGraphu danou definici dopsat a OpenSceneGraph znova zkompilovat. Konkrétně je nutno do souboru `src/osg/StateSet.cpp` do konstruktoru třídy `TextureGLModeSet` nutno přidat řádek

  ```
  _textureModeSet.insert(GL_TEXTURE_BUFFER);
  ```

- OpenSceneGraph používá vlastní definice konstant z OpenGL, ve kterých, opět, není `GL_TEXTURE_BUFFER`, takže je nutno provést další zásah do zdrojových kódu a to do souboru `include/osg/Texture` přidat někde před `namespace` řádky

```
#ifndef GL_TEXTURE_BUFFER
  #define GL_TEXTURE_BUFFER                        0x8C2A
  #define GL_MAX_TEXTURE_BUFFER_SIZE               0x8C2B
  #define GL_TEXTURE_BINDING_BUFFER                0x8C2C
  #define GL_TEXTURE_BUFFER_DATA_STORE_BINDING     0x8C2D
  #define GL_TEXTURE_BUFFER_FORMAT                 0x8C2E
#endif
```

4.5 Výpočty na GPU

Shadery v této práci můžeme rozdělit na dva: osvětlovací shader a raytracing shader. Osvětlovací shader počítá Phongův osvětlovací model s Phongovým stínováním (pokud je tato možnost zapnuta) a raytracing shader může technicky vzato počítat cokoliv co se raytracingem počítat dá. V demo aplikaci počítá zrcadlové odrazy.

Shadery jsou v této práci rozděleny vždy do dvou souborů. Osvětlovací shader je v souborech `phong.vert` (vertex shader) a `phong.frag` (fragment shader) a raytracovací shader je v souborech `raytrace.vert` (vertex shader) a `raytrace.frag` (fragment shader).

Vertex shader je spouštěn pro každý vertex. Jako vstup má pozici vertexu v souřadnicích modelu (jsou mu vlastně přímo předány souřadnice, které jsou zadány v OpenGL) a jeho výstupem je pozice vertexu v souřadnicích framebufferu – pozice pixelu ve framebufferu + hloubka.

Ve fixed pipeline se tato transformace provede vynásobením modelovou, pohledovou a projekční maticí (OpenGL má modelovou a pohledovou spojenou do jedné.). Modelová matice transformuje souřadnice bodu z modelových souřadnic (počátek v počátku modelu) do světových souřadnic (počátek v absolutním počátku světa). Pohledová matice ze světových souřadnic do souřadnic oka (počátek v umístění kamery) a projekční aplikuje způsob promítání (perspektiva, volné rovnoběžné promítání...) a transformuje souřadnice ze souřadnic oka do souřadnic framebufferu.

V shaderu můžeme samozřejmě provést jakoukoliv transformaci chceme, ale pro správné zobrazení 3D objektů je dobré postupovat stejně jako v případě fixed pipeline.

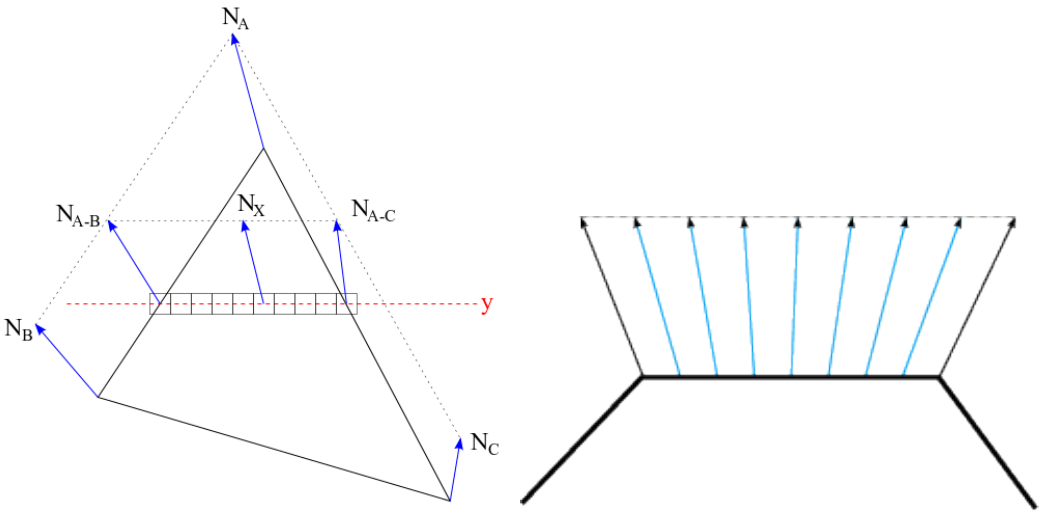
Vertex shader může také definovat `varying` proměnné. Tyto proměnné jsou automaticky interpolovány mezi jednotlivými vertexy a tyto interpolované hodnoty poté přichází do fragment shaderu. To je výhodné například pro interpolaci normál nebo texturovacích souřadnic.

4.5.1 Osvětlovací shader

Osvětlovací shader využívá k výpočtu osvětlení objektů Phongovo stínování [2] a Phongův osvětlovací model.

Stínování říká, jak se počítají normály objektu. Phongovo stínování je takový způsob, kdy se normály interpolují z vrcholů. Běžně se používá interpolace v trojúhelníku, na který se dají převést všechny složitější polygony. Tento způsob stínování vytváří dojem povrchu, který je oblý a nikoliv složený z mnoha plošek, jako vytváří například ploché stínování (pro celý polygon se bere jedna normála). Ilustraci phongova stínování můžeme vidět na obrázku 4.5.

Efektu zaoblení povrchu se dá dosáhnout i Gouraudovým stínováním (u něj se ve vrcholech vypočítá barva a ta se poté interpoluje přes polygon), avšak to má poměrně značný problém s velkými polygony. Jako příklad si vezměme situaci na obrázku 4.6.



Obrázek 4.5: *Ilustrace Phongova stínování. Vlevo interpolace na trojúhelníku, vpravo interpolace v průřezu. Černé šipky jsou vertex normály, modré jsou interpolované přes polygon.*

Barva se vypočítá ve vertexech A a B. Vzhledem k tomu, že situace v obou je téměř stejná (úhel mezi normálou a světlem a vzdálenost ke světlu), bude vypočítaná barva u obou velice podobná. Při interpolaci barvy poté dojde k tomu, že polygon bude mít vlastně skoro celý stejnou barvu. Což ovšem není dobře, protože správně by měla být nejjasněji osvětlená oblast kolem středu úsečky AB. Toho dosáhneme pomocí Phongova stínování, které počítá osvětlovací model pro každý fragment zvlášť, takže úhel mezi normálou a světlem a vzdálenost od světla vezme v úvahu při každém pixelu. Je to sice výpočetně náročnější metoda, ale produkuje přesnější a estetičtější výsledky. V raytracovacím shaderu je použita stejná metoda pro výpočet osvětlení odrážejících se objektů.

Pro interpolaci normál je velice výhodné použít `varying` proměnných. Stačí, aby vertex shader zapsal správné hodnoty normál a fragment shader je dostane již interpolované. Jen je třeba pamatovat na to, že při interpolaci není zaručeno, že normála bude mít správnou (jednotkovou) velikost. V každém fragmentu je tedy třeba normálu normalizovat.

4.5.2 Raytracovací shader

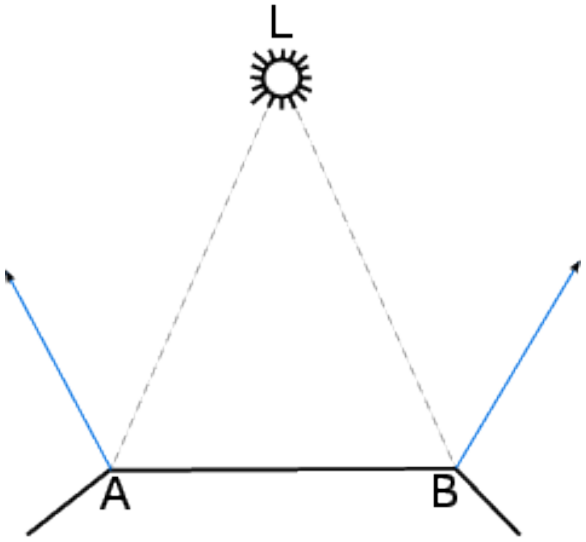
Při raytracingu je klíčová a nejvíce časově náročná jedna operace – nalezení průsečíku paprsku a trojúhelníku. Toto se totiž opakuje mnohokrát v průběhu výpočtu jednoho pixelu. Rychlost raytracingu proto ze značné části záleží právě na rychlosti tohoto algoritmu. V této práci jsou použity dva algoritmy – algoritmus pro výpočet barycentrických souřadnic [1] a geometrický algoritmus.

Základem pro oba algoritmy je vypočítat průsečík paprsku a roviny, ve které trojúhelník leží. Máme tedy parametrickou rovnici paprsku (jedná se o rovnici přímky, ale v případě paprsku nabývá l jen nezáporných hodnot),

$$\vec{x} = \vec{p} + l \cdot \vec{s}$$

kde x je bod na paprsku, $\vec{p}$ je počátek, $\vec{s}$ je směr a l je parametr, a rovnici roviny,

$$\vec{n} \cdot \vec{x} + d = 0$$



Obrázek 4.6: *Ilustrace chyby ve výpočtu osvětlení Gouraudovým stínováním.*

kde $\vec{n}$ je normála, $\vec{x}$ je bod ležící na rovině a d je číslo reprezentující, jak daleko je rovina od počátku souřadnic. Naším cílem je vypočítat l tak, abychom po dosazení do rovnice paprsku dostali bod ležící na rovině. $\vec{x}$ je tedy v obou rovnicích stejné takže můžeme dosadit

$$\begin{aligned} \vec{n} \cdot (\vec{p} + l \cdot \vec{s}) + d &= 0 \\ \vec{n} \cdot \vec{p} + l \cdot \vec{s} \cdot \vec{n} + d &= 0 \\ l \cdot \vec{s} \cdot \vec{n} &= -d - \vec{n} \cdot \vec{p} \\ l &= \frac{-d - \vec{n} \cdot \vec{p}}{\vec{s} \cdot \vec{n}} \\ l &= -\frac{d + \vec{n} \cdot \vec{p}}{\vec{s} \cdot \vec{n}} \end{aligned}$$

Jedinou věcí, kterou nám zbyvá dopočítat v této rovnici je d, které jednoduše odvodíme z rovnice roviny

$$d = -\vec{n} \cdot \vec{b}$$

kde b je libovolný bod roviny a $\vec{n}$ je normála. Už při výpočtu l můžeme vyloučit některé výsledky. Pokud vyjde $\vec{n} \cdot \vec{s} \geq 0$, znamená to, že paprsek je s trojúhelníkem rovnoběžný (při 0) nebo prochází jeho zadní stranou (což v případě, kdy bereme v úvahu pouze vyrobitelné objekty, které nemohou mít nulovou tloušťku, nebudeme vykreslovat) a tedy se s trojúhelníkem neprotíná. Po výpočtu l můžeme eliminovat další situace a to sice pokud $l < 0$, tak trojúhelník leží na opačnou stranu od počátku, než je směr paprsku, takže jej paprsek neprotíná. Pokud nepočítáme s průhlednými objekty (což v této práci nebereme v úvahu), je nutno v tomto bodě zavést kontrolu hloubky, abychom počítali pouze s průsečíkem, jež je nejblíže počátku paprsku. Toho dosáhneme tím, že při hledání průsečíků si pamatujeme nejmenší l průsečíku a pokud je nalezené l větší než uložené, tak to znamená, že by nový průsečík ležel dále od počátku než nějaký již nalezený. Tím pádem pro tento vzdálenější

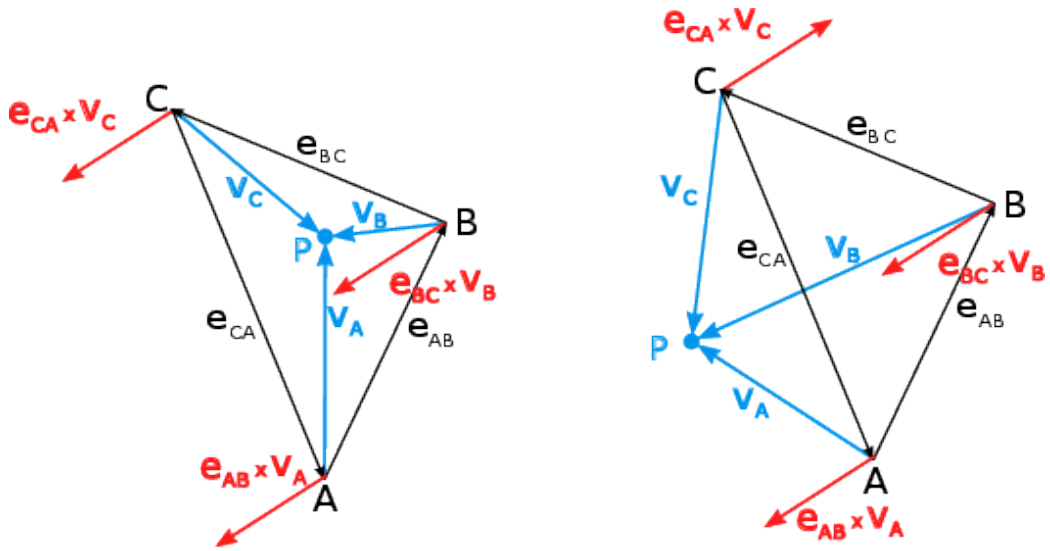
průsečík nemá ani cenu kontrolovat, jestli leží uvnitř trojúhelníku, protože by stejně byl zahozen, takže tímto způsobem můžeme urychlit výpočet.

Dalším krokem pro zjištění, zda paprsek protíná trojúhelník, je tedy zjistit, zda průsečík, který leží ve stejné rovině jako trojúhelník, skutečně leží uprostřed trojúhelníku. Trojrozměrný problém jsme si zjednodušili na dvojrozměrný. Právě k řešení tohoto problému jsou použity výše uvedené algoritmy.

Geometrický algoritmus

U tohoto algoritmu bohužel naprosto nemám tušení, kdo je jeho autorem a jak se vlastně doopravdy jmenuje. Než jsem totiž začal hledat algoritmy pro průsečík paprsku a trojúhelníku na Google, vzal jsem si tužku a papír a přišel jsem na tento způsob. Až později, při hledání rychlejších řešení, jsem našel tento algoritmus popsán na [4].

Jeho princip je velice podobný Pinedovu algoritmu pro rasterizaci polygonu [3]. Pro kandidátní bod zjišťujeme, zda leží ve „správné“ polorovině určené postupně každou stranou trojúhelníka. To, na které straně leží, zjišťujeme pomocí vektorového součinu. Vektory při něm totiž tvoří pravotočivou bázi, takže když máme vektor $\vec{e}_{AB}$, který směřuje od bodu A do bodu B trojúhelníka, a $\vec{v}_A$, který směřuje z bodu a do kandidátního bodu, jejich vektorový součin bude směřovat nahoru (pokud se díváme z bodu A směrem k bodu B), pokud bude kandidátní bod vlevo a dolů, pokud bude vpravo. Pokud po výpočtu pro všechny tři strany trojúhelníka dostaneme tři vektorové součiny, které míří stejným směrem, je bod uvnitř tohoto trojúhelníka. Pokud se některý z nich liší, je bod vně. Algoritmus je ilustrován na obrázku 4.7.



Obrázek 4.7: *Ilustrace geometrického algoritmu pro zjištění, zda bod leží v trojúhelníku nebo ne*

Nevýhodou tohoto algoritmu je, že je poté třeba ještě dopočítat barycentrické souřadnice, aby bylo možno interpolovat z vrcholů. Pokud bereme v úvahu jen jednostranné trojúhelníky, tak je možné algoritmus zrychlit tím, že po vypočítání každého vektorového součinu zkontrolujeme, zda má stejný směr jako normála trojúhelníku. Pokud je totiž opačný, znamená to, že bod leží na vnější straně poloroviny a tím pádem můžeme výpočet ukončit.

Výpočet barycentrických souřadnic

Tento algoritmus není příliš efektivní provádět na CPU, protože je v něm počítáno mnoho skalárních součinů a navíc v plovoucí řádové čárce, se kterou CPU počítá jen pomalu. Na druhou stranu GPU je přesně na tyto operace optimalizováno, takže je výhodné tento algoritmus použít, protože rovnou dostaneme barycentrické souřadnice.

V tomto algoritmu máme 3 vektory: $\vec{u}$, což je vektor z bodu A do bodu B, $\vec{v}$, což je vektor z bodu A do bodu C a $\vec{w}$, což je vektor z bodu A do kandidátního bodu.

Jako první vypočteme s souřadnici, která udává, nakolik je bod posunut podle vektoru u.

$$s = \frac{(\vec{u} \cdot \vec{v})(\vec{w} \cdot \vec{v}) - (\vec{v} \cdot \vec{v})(\vec{w} \cdot \vec{u})}{(\vec{u} \cdot \vec{v})^2 - (\vec{u} \cdot \vec{u})(\vec{v} \cdot \vec{v})}$$

Jelikož pro hodnoty s a t platí určitá omezení, můžeme nyní zrychlit výpočet ukončením, pokud nebude s tyto podmínky splňovat (místo toho, abychom to kontrolovali až na konci výpočtu). V tuto chvíli je podmínka, že $s \in \langle 0, 1 \rangle$. Pokud je s mimo tyto meze, znamená to, že je mimo trojúhelník. Pokud je v mezích, můžeme přistoupit k výpočtu t.

$$s = \frac{(\vec{u} \cdot \vec{v})(\vec{w} \cdot \vec{u}) - (\vec{u} \cdot \vec{u})(\vec{w} \cdot \vec{v})}{(\vec{u} \cdot \vec{v})^2 - (\vec{u} \cdot \vec{u})(\vec{v} \cdot \vec{v})}$$

Pro t platí stejná podmínka jako pro s a sice, že $t \in \langle 0, 1 \rangle$. Navíc platí, že $s + t \in \langle 0, 1 \rangle$, jinak bod leží mimo trojúhelník. Je možno si povšimnout, že oba zlomky mají stejný jmenovatel, což skýtá další možnost ke zrychlení výpočtu – je možno si ho předpočítat a tím si ušetřit několik operací. Každý skalární součin se také ve výpočtu vyskytuje nejméně dvakrát, takže tyto si také můžeme předpočítat.

Tímto algoritmem vypočteme souřadnice s a t, pomocí kterých můžeme potom dosazením do rovnice

$$\vec{w} = s\vec{u} + t\vec{v}$$

vypočítat souřadnice, normálu, texturovací souřadnici nebo třeba barvu pro kterýkoliv bod v trojúhelníku. Nicméně rovnice, tak jak je, nám moc nevyhovuje, protože se v ní počítá s vektory, tedy s rozdíly mezi vrcholy, a my přitom máme přímo hodnoty ve vrcholech. Tuto rovnici si tedy můžeme upravit, abychom počítali skutečně barycentrické souřadnice.

$$\begin{aligned} \vec{w} &= s\vec{u} + t\vec{v} \\ P - A &= s(B - A) + t(C - A) \\ P &= A + sB - sA + tC - tA \\ P &= (1 - s - t)A + sB + tC \end{aligned}$$

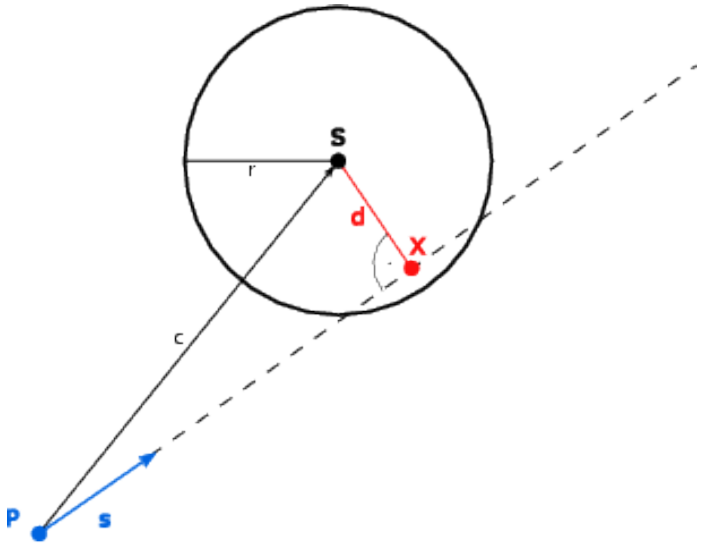
A, B a C jsou vrcholy trojúhelníku, P průsečík. Číslo $1 - s - t$ nazveme třeba r a dostaneme tím pádem známou rovnici pro barycentrické souřadnice.

$$P = rA + sB + tC$$

Obálková koule

Samozřejmě, že mít rychlý algoritmus pro průsečík paprsku a trojúhelníku je základ, ale pokud je nutno pokaždé kontrolovat všechny trojúhelníky ve scéně, trvá vykreslení moc dlouho i s tím nejrychlejším algoritmem. Rozhodl jsem se proto implementovat jednoduchou heuristiku, která je však při scéně sestávající z navzájem nepropletených objektů poměrně účinná: obálková koule (bounding sphere). Pro každý objekt (Drawable v OpenSceneGraphu) je vypočítána obálková koule a je předána do shaderu spolu s dalšími informacemi o objektech.

Algoritmus pro výpočet, zda paprsek protíná kouli je jednoduchý. Nepotřebujeme totiž znát průsečíky, takže nám stačí zjistit kolmou vzdálenost paprsku od středu koule a porovnat ji s poloměrem koule. Algoritmus je ilustrován na obrázku 4.8.



Obrázek 4.8: *Ilustrace algoritmu pro zjištění protnutí paprsku a koule*

cs Aby algoritmus fungoval, musí být vektor $\vec{s}$ normalizovaný. Pro zjištění vzdálenosti paprsku od středu musíme nejdříve spočítat polohu bodu X. To provedeme tak, že nejdříve vypočítáme parametr l parametrické rovnice paprsku a poté s jeho pomocí vypočítáme bod X. Parametr zjistíme vztahem

$$l = \vec{c} \cdot \vec{s}$$

Pro urychlení výpočtu můžeme už v tuto chvíli vyloučit protnutí, pokud je $l < 0$, protože to znamená, že paprsek míří směrem od koule. Je zde však jedna situace, kdy paprsek kouli protíná i v tomto případě: když začíná uvnitř koule a míří směrem od středu. Potom je nutné provést ještě kontrolu, zda vzdálenost PS je menší než poloměr. Pokud je, tak paprsek vychází zevnitř koule a tím pádem může obalovaný objekt samozřejmě protnout. Další možnost pro urychlení je neporovnávat délky, ale druhé mocniny délek. Vyhneme se tím výpočetně náročnějšímu odmocňování.

Výpočet raytracingu

Nyní si řekneme, jak vlastně probíhá výpočet v raytracovacím shaderu.

V shaderu není možno využívat rekurzi. Ačkoliv GLSL je jazyk velice podobný jazyku C, hardware GPU je přecijen trochu jiný, takže tuto možnost nemá. My však potřebujeme počítat několik sekundárních paprsků, na které je nejjednodušší postup právě rekurze. Tento přístup tedy musíme změnit na iterativní za pomocí zásobníku. Nicméně v shaderu není možno ani vytvářet pole, jehož velikost je daná nějakou `uniform` proměnnou (alespoň v GLSL ne), natož nějaké dynamické struktury. Nejjednodušší je tedy vytvořit pole statické velikosti a to použít jako zásobník. Velikost pole jsem zvolil 30, což je sice magická konstanta, ale řekl bych, že 30 odrazů je poměrně dost. V poli jsou uloženy struktury typu `rayStackElement`, které ukládají informace o nalezeném průsečíku.

Výpočet tedy probíhá tak, že nejdříve se pomocí funkce `getRayIntersection`spočítají všechny sekundární paprsky (samozřejmě, že pokud paprsek narazí na neodrazivý objekt tak už výpočet dále neprobíhá), pro které se uloží do pole `rayStack` do kterého objektu se trefily, jaký bod, jaká je jeho normála, texturovací souřadnice a z jakého směru do objektu paprsek přišel. Poté se pole prochází od posledního protnutí a počítá se barva. Pro každé protnutí se spočítá osvětlovací model a potom se podle jeho koeficientu odrazivosti smíchá s bodem který je dále v poli a tím pádem byl už spočítán (protože výpočet probíhá od konce pole, respektive od posledního nalezeného průsečíku). Nakonec dostaneme barvu paprsku, který dopadá na naše právě počítané odrazivé těleso, takže nakonec uložíme do `gl_FragColor` barvu a jako alfa kanál jeho koeficient odrazivosti.

Kapitola 5

Závěr

Obsahem zadání bylo vytvořit jednoduchý raytracer pro OpenSceneGraph, který bude výpočty provádět na GPU. Tento cíl se podařilo naplnit.

Při řešení jsem nejdříve nastudoval jak se programují shadery a jak funguje texture buffer v OpenGL. Tímto směrem jsem postupoval z toho důvodu, že čistě v OpenGL jsem zjistil, co je třeba udělat v OpenGL, abychom mohli s texture bufferem spolupracovat. Potom jsem v OpenSceneGraphu identifikoval, co v něm potřebuji udělat, aby se dostal do stejného stavu jako v OpenGL. Výstupem tohoto je třída `TextureBuffer` a `TextureBufferObject`. Dále jsem průběžně vytvářel formát pro předání dat scény do GPU a psal raytracovací shader.

Povedlo se mi vytvořit řešení, jak předávat do GPU značný objem libovolných dat, což samo o sobě není úplně jednoduché, a také formát dat pro předání scény do GPU. Dále jsem vytvořil shadery v GLSL, které jsou schopné raytracovat a nejen to. Funkce `getRayIntersection` prostě zjistí, se kterým objektem se paprsek protíná, takže si dovedu představit například využití pro výpočet globální iluminace. Výstupem je i demo aplikace, která počítá zrcadlové odrazy pro zadané objekty.

Jako možný vývoj do budoucna bych viděl například doladit zakomponování texture bufferu do OpenSceneGraphu a možná návrh na zahrnutí do oficiálních zdrojových kódů. Dále bych vylepšil systém předávání textur všech objektů scény do shaderu. Tak jak to je implementováno momentálně nelze předat více než 6 textur, protože OpenGL specifikace udává jako počet texturovacích jednotek aspoň 8. Ačkoliv to v praxi bývá více, je nutno počítat s tímto omezením (jedna se musí vzít na texturu aktuálního objektu a jedna na texture buffer). Nicméně k texturám scény se v shaderu přistupuje pomocí funkce `getTexel`, která přijímá index textury a texturovací souřadnici, takže změna způsobu předávání by vyžadovala v podstatě přeimplementovat pouze tuto metodu.

Literatura

[1] Ericson, C.: *Real-time collision detection*. Morgan Kaufmann, 2004.

[2] Phong, B. T.: Illumination for computer generated pictures. *Communications of the ACM*, ročník 18, č. 6, 1975: s. 311–317.

[3] Pineda, J.: A parallel algorithm for polygon rasterization. In *ACM SIGGRAPH Computer Graphics*, ročník 22, ACM, 1988, s. 17–20.

[4] Scratchapixel: Ray-Triangle Intersection: Geometry solution [online] (cit. 8.5.2013). 2012.
URL http://www.scratchapixel.com/lessons/3d-basic-lessons/lesson-9-ray-triangle-intersection/ray-triangle-intersection-geometric-solution/

[5] Wang, R.: *OpenSceneGraph 3 Cookbook*. Packt Publishing, Limited, 2012.