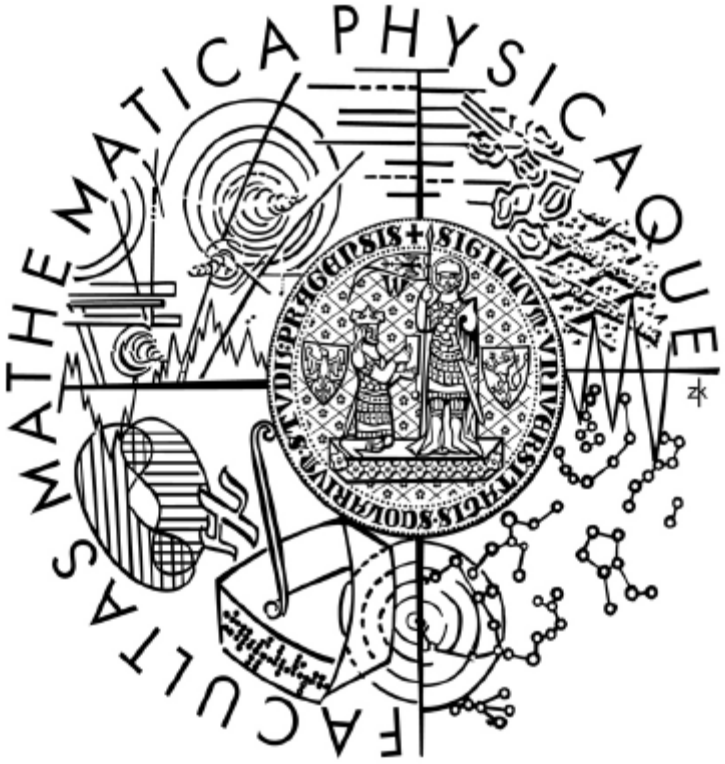
Universita Karlova v Praze

Matematicko-fyzikální fakulta

Diplomová práce



Lukáš Maršálek

Osvětlení na GPU

Kabinet software a výuky informatiky
Vedoucí diplomové práce: RNDr. Josef Pelikán
Studijní program: Informatika, Softwarové systémy, Počítačová grafika

Na tomto místě bych rád poděkoval vedoucímu práce RNDr. Josefu Pelikánovi za připomínky a podněty, které umožnily vznik této práce.

Dále děkuji svým rodičům a své sestře za bezvýhradnou podporu během celého studia a při tvorbě této práce.

Děkuji také Marku Benešovi a Martinu Ducháčkovi za podporu v zaměstnání, která učinila tvorbu této práce snažší.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 9.8.2005 Lukáš Maršálek

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů

Název práce: Osvětlení na GPU
Autor: Lukáš Maršálek
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí diplomové práce: RNDr. Josef Pelikán
e-mail vedoucího: Josef.Pelikán@mff.cuni.cz

Abstrakt:

Cílem práce je prozkoumat možnost implementace zobrazování objemu na GPU. Zobrazováním objemu se rozumí metody, jejichž vstupními daty nejsou přímo povrchové reprezentace těles, ale data zadaná trojrozměrnou mřížkou skalárních nebo vektorových veličin. Jejich tradičními vlastnostmi jsou na jedné straně schopnost vykreslovat komplexní procedurálně definovaná tělesa a na straně druhé obtížné a pomalé zobrazování. Proto neměly tyto metody v oblasti zobrazování v reálném čase velké uplatnění.

Výkon a flexibilita grafických systémů v běžně dostupných stolních počítačích narostly v současné době natolik, že umožňují alespoň experimentální implementaci těchto metod.

V úvodu je nejprve vytvořen kontext a vytyčen cíl práce. Poté jsou zevrubně prezentovány dosavadní přístupy k problematice zobrazování objemu a je diskutována jejich vhodnost pro implementaci na GPU. Zahrnut je také teoretický základ optických modelů.

Další části jsou věnovány podrobně navržené zobrazovací metodě a způsobu a úskalím její implementace na GPU. Práci uzavírá otestování implementované metody, zhodnocení dosažených výsledků a nakonec je uveden nástin dalších směrů vývoje.

Klíčová slova: GPU, zobrazování objemu, shader, vrhání paprsku, optické modely

Title: Lighting using GPU
Author: Lukáš Maršálek
Department: Department of Software and Computer Science Education
Supervisor: RNDr. Josef Pelikán
Supervisor's e-mail address: Josef.Pelikan@mff.cuni.cz

Abstract:

The goal of this work is to examine the possibility of implementing a volume rendering method entirely on GPU.

By the term volume rendering I mean methods which take as an input tree-dimensional scalar or vector grid data, rather than a classical surface representation. Traditional characteristics of these methods is the capability to render highly complex, possibly procedurally defined objects on one side and high computational cost on the other side. It was that computational cost which in the past prevented their widespread use in real-time rendering.

However sheer computational power and flexibility of current widely available graphical systems has reached levels which allow for at least experimental implementation of these methods.

First in the introduction the background and goals of this work are presented. Next current and past approaches to volume rendering are presented in detail and also their suitability for GPU implementation is discussed. A theoretical foundation for the optical models is also given.

Further chapters are devoted in detail to the suggested method and to the means of its implementation on GPU. The work is closed by the testing of the method, analysis of the achieved results and finally further research directions are suggested.

Keywords: GPU, volume rendering, shaders, ray-casting, optical models

Kapitola 1

Úvod

Cílem této kapitoly je podat úvod do problematiky, kterou se práce zabývá, nebo která má vliv na prezentované výsledky a postupy.

Nejprve se v §1.1 seznámíme s oblastí zobrazování objemových dat, k čemu se metody z této oblasti používají a jak je lze obecně klasifikovat. §1.2 navazuje přehledem základní posloupnosti kroků, kterou musí výpočetní systém při zobrazování objemu provést. Principům zobrazování v reálném čase a způsobům měření výkonu a kvality zobrazovacích algoritmů je věnován §1.3. V §1.4 je shrnuta platforma současných běžně dostupných grafických řešení pro stolní počítače, která tvoří implementační prostředí pro tuto práci. Kapitolu uzavírá vytyčení cílů práce v §1.5 a rozvrh členění textu v §1.6

1.1 Zobrazování objemu

1.1.1 Principy a použití

Zobrazováním objemu nazýváme souhrnně metody převádějící data z vícerozměrných skalárních či vektorových mřížek na dvourozměrnou průmětnu [Owen 99]. Od tradičního zobrazování objektů uložených v povrchové reprezentaci se liší v mnoha aspektech. Způsob, jakým jsou data reprezentována, jejich charakter a velikost a také cíle jejich vizualizace, jsou všechno příčiny návrhu a implementace odlišných zobrazovacích technik.

Nejběžněji se setkáváme s třírozměrnými mřížkami, které reprezentují hodnoty měřené nebo vypočítané veličiny v prostoru. Časté jsou také čtyřrozměrné mřížky reprezentující danou veličinu v prostoru a čase. Podle zdroje dat se také velmi liší druh mřížek, ve kterých jsou data definována [Elvins 92]. Lze se setkat s kartézkými mřížkami, kde buňky jsou identické, osově orientované krychle. Obecnější jsou pravidelné a rovnoběžné mřížky, kde buňky tvoří identické osově orientované kvádry, respektive osově orientované kvádry ne nutně stejných rozměrů. Nejobecnější jsou pak strukturované, blokově strukturované a nestrukturované mřížky.

Zdroje dat pocházejí ze dvou základních oblastí. Jedná se o vědeckou či průmyslovou vizualizaci a fotorealistické zobrazování.

Příkladem z první oblasti mohou být medicínská data získaná z MRI (Magnetic Resonance Imaging), CT (Computed Tomography) či PET (Positron Emission Tomography). Tato data jsou skalární a jsou zaznamenávána zpravidla do třírozměrné rovnoběžné mřížky. Vícerozměrná data jsou produkována například v meteorologii, kde rozměr bývá i v řádu desítek. Často je však k těmto datům přistupováno jako ke skalárním a je zobrazována najednou vždy jen jedna veličina. Odvětvím, kde jsou produkována skutečně vektorová data, typicky na čtyřrozměrné mřížce, jsou průmyslové počítačové simulace jako CFD (Computational Fluid Dynamics). Ve vědecké a průmyslové vizualizaci se zobrazování objemu používá zejména pro získání lepšího náhledu na sledovaná data. Jde například o jejich prostorové rozložení, pozice extrémů a způsoby přechodu mezi nimi či množství a velikost oblastí se stejnými hodnotami. Při zobrazování dat v čase bývá navíc snahou zachytit trend změny dat, jako například směr či druh proudění.

Ve fotorealistickém zobrazování jsou pomocí tří nebo čtyř rozměrných skalárních dat modelovány přírodní fenomény jako mlha, mraky, oheň a kouř nebo také objekty s extrémně složitým povrchem. Cílem zde bývá dosažení maximální věrohodnosti napodobovaného fenoménu. Velkým přínosem některých metod zobrazování objemu je možnost věrně modelovat fyzikální vlastnosti průhledných a průsvitných objektů jako je útlum, rozptyl a lom světla.

V obou uvedených oblastech je velkou výhodou, pokud lze dosáhnout interaktivních rychlostí zobrazování. Při zobrazování prostorových dat ve vědecké a průmyslové vizualizaci interaktivní rychlost umožňuje například efektivně nastavovat a měnit parametry mapování barev či prohlížet data z různých pozic a vzdáleností. V oblasti fotorealistického zobrazování se pak otevírají možnosti využít zobrazovaných objektů ve virtuálních prostředích či ve hrách a nejen ke generování statických snímků a neinteraktivních videí.

Tato práce se zaměřuje na oblast zobrazování skalárních dat definovaných na třírozměrné rovnoběžné mřížce.

Jak uvidíme v následujících dvou kapitolách, bylo vyvinuto mnoho různých přístupů k zobrazování objemu. Lze však i přesto nalézt několik kritérií, podle kterých se tyto metody klasifikují.

1.1.2 Klasifikace podle způsobu promítání

Tento způsob klasifikace metod pro zobrazování objemu je velmi častý. Dělí algoritmy podle toho, jak promítají objemová data na průmětnu, na metody hledající povrch a na přímé zobrazovací metody.

Metody hledající povrch
Tyto metody jsou shodně založeny na principu hledání míst v objemu, kde hodnoty nabývají zadaného prahu (isoploch) a vizualizaci těchto míst pomocí standardních polygonálních ploch či povrchových bodů a již známých algoritmů pro jejich zobrazování. Obecně jsou tyto metody velmi rychlé, jakmile je jednou hledaný povrch zkonstruován. Nevýhodou jsou některé artefakty, jako například falešné rysy povrchu nebo nepřesnosti při zachycování drobných detailů a větvení povrchu. Tyto algoritmy mají také omezené použití na objekty, u nichž je přirozené isoplochy hledat a zobrazovat. Pro „amorfní“ objekty jako mraky a oheň jsou tyto metody obecně nevhodné.

Přímé zobrazovací metody
Jiný přístup používají takzvané přímé zobrazovací metody. Jak název napovídá, snaží se tyto metody přímo promítnout objemová data na průmětnu bez použití polygonálních objektů. Jejich obecnými vlastnostmi jsou schopnost generovat vysoce kvalitní výstup, ovšem za cenu značné výpočetní náročnosti. Těmto metodám byla a je věnována velká pozornost, neboť na rozdíl od metod hledajících povrch, umožňují zobrazit objem jako celek a s pomocí simulace průchodu světla objemem zobrazovat jeho důležité či požadované vlastnosti.

1.1.3 Klasifikace podle dominantního průchodu

Jiným kritériem pro klasifikaci metod zobrazování objemu je dominantní průchod. Metody lze podle tohoto kritéria opět rozdělit na dvě třídy.

Objemově orientované metody
Primární smyčkou pro tyto algoritmy je průchod přes objemová data. Algoritmus projde každou buňku objemu a provede s ní nějakou operaci. V této třídě lze nalézt jak algoritmy hledající povrch, tak přímé zobrazovací metody.

Obrazově orientované metody
U těchto metod naopak převládá průchod přes pixely výsledného obrazu. Tyto jsou procházeny jeden za druhým a pro každý je určena jeho barva. Pro určení barvy je tedy nutné nalézt buňky objemu, které mají na daný pixel vliv. Obrazově orientované metody najdeme pouze mezi přímými zobrazovacími metodami.

1.2 Struktura zobrazovacího systému

Jak bylo řečeno v minulém odstavci, metody pro zobrazování objemu jsou velmi rozmanité a lze se setkat se zcela odlišnými přístupy. Ovšem stejně jako lze, i přes tuto rozmanitost, metody klasifikovat, lze také obecně určit posloupnost kroků, kterou je nutné při zobrazování objemu provést. Většina dále popsaných metod, včetně metody prezentované v této práci, implementuje podmnožinu níže uvedených kroků. Ostatní buďto zjednodušuje pomocí dodatečných předpokladů nebo předpokládá, že jsou provedeny mimo vlastní metodu.

Obdobnou klasifikaci, jako je uvedena níže, lze nalézt např. v [Elvins 92]. Práci s objemovými daty, od jejich získání až po vytvoření konečného obrazu, lze shrnout do následujících kroků:

- Pořízení dat
- Úprava dat do podoby požadované pro zobrazování
- Klasifikace dat
- Promítnutí dat na průmětnu
- Zobrazení pixelů průmětny

Pořízení dat (s1)
V tomto kroku jsou naměřena nebo spočítána vlastní objemová data. Přitom již dochází k prvnímu vzorkování. Veličina či funkce je měřícím zařízením nebo

výpočtem vzorkována. Všechny metody pro zobrazování objemu předpokládají, že tento krok byl proveden korektně. Tedy zejména, že data byla navzorkována s frekvencí nad Nyquistovým limitem a lze je proto korektně rekonstruovat. Je samozřejmé, že tento požadavek nelze vždy dodržet, neboť nelze obecně zaručit frekvenční omezenost měřených dat.

Úprava dat do podoby vhodné pro zobrazování (s2)
Spočítaná či naměřená data nemusejí být přímo vhodná pro zobrazování nebo je žádoucí na nich provést před zobrazením nějaké korekce. Typické prováděné kroky bývají ekvalizace histogramu, úprava kontrastu a světlosti či odstranění šumu. Pokud jsou naměřená data v nestrukturovaných mřížkách či mají nesymetrické rozměry, bývají často přepočítána do symetrických kartézkých nebo rovnoběžných mřížek. To zahrnuje interpolaci dat v nových uzlových bodech či tvorbu dodatečných řezů. Nejčastěji se používají pomalé, ale kvalitní interpolace, jako například trikubická nebo radiální. Zejména v medicínských aplikacích se v tomto kroku mohou také kombinovat různé datové množiny. Tento proces zahrnuje jejich registraci (tedy proces zarovnání odpovídajících si bodů v obou množinách) a kombinaci hodnot do výsledné množiny.

Klasifikace dat (s3)
Předchozí dva kroky bývají stejné pro všechny metody zobrazování objemu. Klasifikace dat je první krok, kde se metody začínají odlišovat. Klasifikací dat se rozumí mapování vstupních hodnot na veličiny používané při zobrazování.

V metodách hledajících povrch se tento krok skládá z volby prahu, pro který je hledán povrch a někdy také z volby barvy, kterou je tento povrch zobrazován. V některých aplikacích lze zobrazovat v jednom obrázku více povrchů pro více prahových hodnot a barevně je odlišovat.

V metodách přímého zobrazování objemu, klasifikace dat typicky sestává z přiřazení průhlednosti a barvy pro každou z možných vstupních hodnot. V některých aplikací je také do klasifikace zahrnuta pozice vzorku. Hustota a barva jsou tak modifikovány nejen podle hodnoty vzorku, ale i podle jeho prostorových souřadnic.

V aplikacích ve fotorealistickém zobrazování bývá také prováděn speciální krok, který simuluje šíření světla ze světelných zdrojů objemem. Například pokud objem napodobuje mraky, je nutné pro každý světelný zdroj vyřešit rovnice pro šíření záření v prostoru (viz §3.2) a pro každou buňku objemových dat zaznamenat množství, popřípadě barvu dopadajícího světla. Někdy bývá tento krok prováděn současně s promítáním dat na průmětnu. Je to zejména v případě, kdy neřešíme sekundární rozptyl světla objemu a počítáme pouze přímé příspěvky ze světelných zdrojů. I v takovém případě ale bývá efektivnější provést tyto výpočty v klasifikačním kroku a opakovat je pouze tehdy, pokud se světelné podmínky změní.

Promítnutí dat na průmětnu (s4)
Tento krok je zásadním momentem celé metody. Co přesně je provedeno za operace, se značně liší od metody k metodě a bude to podrobně probráno u každé prezentované metody zvlášť.

Obecně lze říci, že kromě vlastního promítnutí objemových dat na průmětnu, tedy určení jaké buňky mají vliv na jaké pixely, se v tomto kroku provádí také kompozice a určení viditelnosti a bývají aplikovány prostředky antialiasingu.

Zobrazení pixelů průmětny (s5)
Závěrečný krok je opět typicky stejný u většiny metod. Liší se spíše podle implementačního prostředí. Pokud je metoda implementována přímo v software, je potřeba uložená barevná či šedotónová data přenést z hlavní paměti do zobrazovacího zařízení a vydat potřebné instrukce k jejich zobrazení. Pokud pracujeme na hardwarové architektuře, bývá o tento krok často již postaráno, neboť obrazová data jsou zapisována přímo do obrazové paměti, odkud jsou automaticky zobrazována na výstupním zařízení.

1.3 Zobrazování v reálném čase

Zobrazování v reálném čase se uplatňuje samozřejmě nejenom v oblasti zobrazování objemu. Je to velmi široké a aktuální téma. Hlavní snahou je prezentovat grafické informace uživateli takovou rychlostí, aby měl pocit okamžité odezvy na své akce, či byl schopen vnímat plynulý pohyb virtuálních objektů.

V oblasti teorie algoritmů bylo formalizováno několik postupů pro měření rychlosti algoritmů a jejich srovnávání na teoretické bázi. Pojem „v reálném čase“ je z toho pohledu velmi vágní a jeho přesný význam se typicky odvozuje až od příslušné aplikační oblasti. Je to zejména proto, že se snažíme měřit spokojenost uživatele s nějakým rysem a je závislé na druhu aplikace, co přesně tyto pocity určuje. Nicméně během času se vyvinulo několik měřítek, která jsou obecně přijímána jako ukazatel rychlosti zobrazování. Nejběžněji používanými měřítky, kterých se budeme držet i v této práci, je hodnota snímkovací frekvence a její konzistence.

Snímkovací frekvencí (dále fps z anglického frames per second) rozumíme počet snímků, který je systém schopen prezentovat uživateli za jednu vteřinu. Tato hodnota může být značně ovlivněna způsobem, jakým se pracuje se zobrazovacím zařízením. Lze ji chápat jako počet snímků, které je systém nebo algoritmus skutečně schopen dodat za vteřinu nebo jako počet, kolikrát se obrázek na monitoru skutečně objeví. K rozdílu mezi první a druhou interpretací dojde, pokud systém používá metodu double-bufferingu (nebo i více bufferů) a čeká na synchronizaci se zobrazovacím zařízením. V takovém případě nemůže hodnota fps přesáhnout vertikální obnovovací frekvenci zobrazovacího zařízení (v anglické terminologii někdy uváděné jako refresh rate). V této práci je používán první způsob.

Konzistencí snímkovací frekvence rozumíme schopnost algoritmu či systému udržet hodnotu fps konstantní v čase. V mnoha studiích (např. [Yuan et al. 97]) se ukazuje, že kromě vlastní maximální hodnoty fps je důležitá právě její konzistence. Typické problémy, kterými uživatelé trpí, pokud není hodnota fps konzistentní, jsou neschopnost provádět zadané úkoly ve virtuálním prostředí a v některých případech i např. nevolnost či ztráta zrakově-pohybové koordinace.

Je zřejmé, že dvě výše uvedená měřítka určitým způsobem vypovídají o rychlosti zobrazovacích algoritmů, ale jejich vztah ke skutečné odezvě systému, tedy době mezi akcí uživatele a jejím projevením se na grafické informaci, je ovlivněn několika dalšími faktory. Jde zejména o zpoždění, které nastává mezi akcí uživatele a předáním změněných parametrů zobrazovacímu systému. Toto zpoždění je často způsobeno zařízeními sledujícími akce uživatele (složitější polohovací systémy,

head-mounted nebo head-coupled displays apod.) a také interpretací této akce a jejím převedením na parametry zobrazovacího systému.

Nyní máme tedy rozumné obecné měřítko rychlosti zobrazovacího algoritmu. Zbývá určit, jaká hodnota fps je přijatelná pro konkrétní účely. Typicky je tato hodnota zjišťována experimentálně. Uživatelům je předložen nějaký úkol a poté je informace zobrazována s různými hodnotami fps a je sledována reakce uživatelů.

Nejběžněji je cílová hodnota fps určena podle vnímání plynulosti pohybu. Přesto, že je opět závislá na uživateli, typická minimální hodnota je 72-75 fps pro zařízení a algoritmy, které neaplikují některou z metod pro zvýšení plynulosti pohybu, jako např. rozmazání pohybem (motion blur). V oblasti počítačových her či simulací se tyto metody aplikují méně často, neboť zhoršují vnímání detailů a přesné pozice rychle se pohybujících objektů.

Často je hodnota fps také studována v souvislosti s prostředími pro virtuální realitu. Konkrétně se schopností uživatelů provádět nějaké úkony, jako například přenos či zaměření těles ve virtuálním světě. Typické měřené hodnoty se pohybují v rozmezí 10 až 30 fps. Studie, jako [Meehan et al. 02] ukazují, že zvyšující se hodnota fps dává uživateli lepší pocit při pobytu ve virtuálním prostředí. Nízké hodnoty okolo 10 fps dávají často nejasné výsledky, které mají tendenci se lišit podle uživatelů. Hodnoty jsou značně nižší, než u vnímaní plynulosti pohybu, neboť typické úkoly dávané v uvedených studiích jsou jednodušší, než např. v počítačových hrách a nezahrnují rychle se pohybující objekty.

V oblasti zpracování videa se studie zaměřují na to, jak uživatelé hodnotí kvalitu přenosu. Typickou požadovanou hodnotu je 24 až 30 fps. Některé studie [McCarthy et al. 04] ovšem naznačují, že postačují i hodnoty 6 fps, tedy daleko nižší, než se běžně předpokládá. Zde je ovšem kladen důraz na celkový dojem z pasivního pozorování a není vyžadována žádná aktivita uživatelů.

Jiní autoři [Lastra et al. 95], zabývající se programovatelným stínováním, považují za interaktivní hodnotu 30 až 60 fps.

1.4 Grafický hardware

V této kapitole se zaměříme na principy, jak se programuje současný konzumní grafický hardware. Cílem této kapitoly není podat vyčerpávající popis stávajících architektur, ale poukázat na základní principy a rozdíly od programování pro obecné procesory.

Historie

Po velmi dlouhou dobu byl hardware specializovaný na grafické operace doménou velmi drahých pracovních stanic, simulátorů či super-počítačů, převážně pod kontrolou firmy SGI.

Postupem času se ovšem začíná objevovat grafický hardware v běžně dostupných stolních počítačích. Tento trend začal v roce 1996 příchodem karet Voodoo 1 od firmy 3Dfx a v letech 1997 a 1998 uvedením prvních karet od firmy Nvidia Riva 128 a Riva TNT. Tyto ranné grafické akcelerátory byly zaměřeny zejména na urychlení práce s texturami v multimédiích a v počítačových hrách.

V dalších letech se rozsah činnosti, kterou vykonává hardware, začíná rozšiřovat. Objevují se první hardwarově urychlované transformace a osvětlovací výpočty (souhrnně nazývané T&L: Transform&Lighting) v řadě karet Nvidia

GeForce 256, které byly uvedeny na trh v roce 1999. Vývoj pokračoval v roce 2000 řadami karet Nvidia GeForce 2 a ATI Radeon. Tyto karty přinášejí další zvýšení výkonu a flexibility v oblasti texturování a objevují se první programovatelné vertex shadery.

Skutečný zlom nastává ovšem v roce 2001 představením standardu Microsoft DirectX 8.0 a řadou karet Nvidia Geforce3 a ATI Radeon 8500. Ve specifikaci DirectX 8.0 jsou standardizovány první verze vertex a fragment shaderů a začíná tak éra programovatelného stínování na konzumních počítačích. Tyto karty lze také označit jako první GPU (Graphics Processing Unit). Obsahují totiž skutečné procesory, které jsou schopné vykonávat i když zatím jen velmi omezený kód psaný programátorem ve speciálním jazyku, který je podobný assembleru. Je to právě tento jazyk, který DirectX 8.0 standardizoval.

Poté nastává velký rozvoj programovatelného stínování. V roce 2002 přichází nová specifikace Microsoft DirectX 9.0 s ní další rozšíření možností, zejména v oblasti flexibility programovacího jazyka. Začínají se také objevovat první vysokoúrovňové programovací jazyky pro práci s GPU, jako jsou Cg a HLSL. V roce 2003 se stávají karty podporující DirectX 9.0 skutečně dostupnými pro každého.

Ke konci roku 2004 spatřila světlo světa specifikace Microsoft DirectX 9.0c a v ní definovaný Shader Model 3.0. Zároveň se objevuje první řada karet, která tuto specifikaci plně podporuje a to Nvidia GeForce 6. Shader Model 3.0 přinesl nové možnosti, zejména z programátorského hlediska. Umožňuje totiž používat ve fragment shaderech skutečné podmíněně ukončené smyčky a tak rozšiřuje výrazové možnosti, stejně jako škálu implementovatelných algoritmů. Shader Model 3.0 přinesl samozřejmě ještě daleko více novinek, ovšem výše uvedené dynamické smyčky jsou zásadním přínosem pro tuto práci. Je to totiž právě platforma karet Nvidia GeForce 6 a specifikace Microsoft DirectX 9.0c, pro niž je cílena metoda navrhovaná v této práci.

V polovině roku 2005 se objevuje ještě nová řada karet Nvidia GeForce 7, která přináší extrémní rychlost pro třídu programů implementovaných podle standardu DirectX 9.0c.

Fixní a programovatelný zobrazovací řetězec

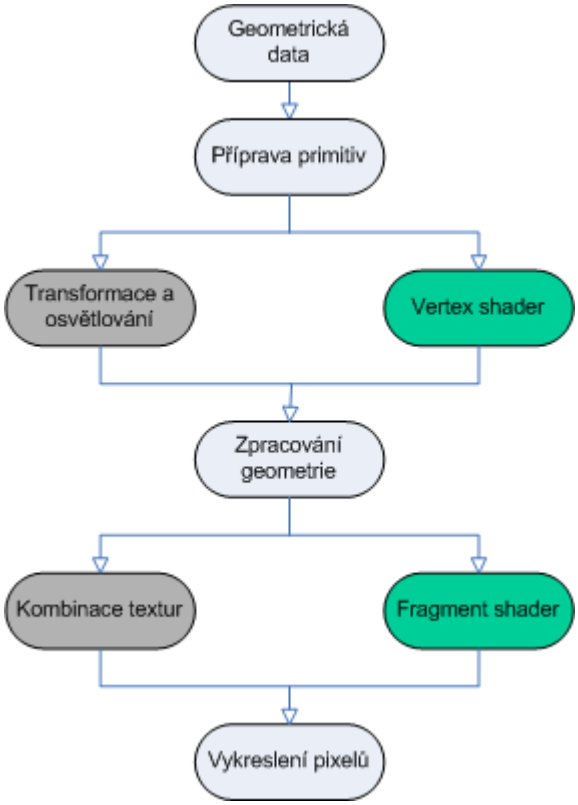
S trochou nadsázky lze říci, že než se objevil specializovaný grafický hardware, byl celý zobrazovací řetězec programovatelný. Veškerý kód byl tvořen v obecných programovacích jazycích a spouštěn na běžných procesorech. Výsledkem těchto výpočtů byla matice hodnot jednotlivých pixelů výsledného obrazu.

S postupem času, jak bylo shrnuto v předcházející sekci, začal být vyráběn specializovaný hardware, který vykonával některé funkce zobrazovacího procesu. Tento hardware byl stavěn z důvodů urychlení grafických výpočtů. Z konstrukčních důvodů byl ovšem velmi omezený v rozsahu funkcí, které zastával. Až do příchodu GPU nebylo možné tento hardware programovat v klasickém slova smyslu pomocí nějakého programovacího jazyka. Bylo pouze možné přepínači nastavovat chování některých jeho částí. Například ve fázi osvětlovaní byl v hardware implementován pouze jeden algoritmus pro aproximaci osvětlení ve vrcholech. Bylo možné nastavit parametry tohoto algoritmu, ale nikoliv jej změnit. Takto pracující zobrazovací řetězec začal být označován jako *fixní řetězec*.(Fixed-Function pipeline).

Jak se hardware vyvíjel dále, začal být systém práce s přepínači příliš nepřehledný a nemotorný. Neboť již padla i technická omezení, byly některé části

fixního řetězce nahrazeny programovatelnými jednotkami. Pro takto zkonstruovaný zobrazovací řetěz se vžil název *programovatelný řetězec* (Programmable pipeline).

Obrázek 1.1 ukazuje schéma zobrazovacího řetězce. Modré části jsou společné pro fixní i programovatelný řetězec. Šedé části patří do fixního řetězce a v programovatelném řetězci jsou nahrazeny zelenými částmi.



Obrázek 1.1: Schéma zobrazovacího řetezce

Nejprve jsou do zobrazovacího řetězce dodána geometrická data.Ta mohou být ve formě polygonální sítě nebo jako různé druhy plátů či povrchy vyšších řádů.

V následujícím kroku jsou tato primitiva zpracována, přesněji jsou vždy převedena na trojúhelníkovou síť. Výsledem tohoto kroku je seznam unikátních vrcholů s atributy (např. pozice, normála, texturovací souřadnice atd.) a informace o tom, které vrcholy tvoří které trojúhelníky.

V následující fázi jsou na každý vrchol aplikovány různé transformace a může být proveden výpočet osvětlení. Ve fixním řetězci byl sled transformací a osvětlovacích výpočtů přesně určen a bylo pouze možné nastavit příslušné transformační matice a parametry osvětlovacího modelu. V programovatelném řetězci je tento krok nahrazen zpracováním, programátorem definovaného, programu. Tento program se nazývá *vertex shader*. Jeho vstupem je **právě jeden vrchol** s atributy a výstupem minimálně pozice v souřadném systému promítací roviny (clip-space position). Vertex shader může produkovat ještě řadu dalších dat. Zásadním předpokladem je, že vertex shader zpracovává vždy právě jeden vrchol a nemá k dispozici informace o ostatních vrcholech, ani o topologii trojúhelníkové sítě. Nelze tedy například přímo získat informaci o sousedních vrcholech nebo vytvořit nový vrchol.

Dále jsou transformovaná data ořezána podle pohledového objemu a potencionálně i podle dalších, uživatelsky definovaných rovin. Je také provedena

transformace do okna (viewport transformation) a výsledné trojúhelníky jsou v souřadném systému okna rasterizovány.

Výsledkem předchozí fáze jsou tzv. *fragmenty*. V některých kontextech bývá zaměňován pojem fragment a pixel. Fragment je výsledem rasterizace a jedná se o pozici v souřadném systému okna spolu s množstvím dalších atributů. Důležitým rozdílem je, že tato pozice nemusí být celočíselná a často je se sub-pixelovou přesností. Do výsledné barvy jednoho pixelu tak může přispět i několik fragmentů (např. při aplikaci technik anti-aliasingu). Atributy fragmentu jsou interpolovány bilineární interpolací z hodnot ve vrcholech trojúhelníku, odkud byl fragment vygenerován. Ve fixním řetězci je nyní ve fázi Kombinace textur určena barva fragmentu (ještě ne ale finální barva) jako kombinace primární barvy a barev z textur, které jsou nalezeny podle interpolovaných texturovacích souřadnic. Počet textur, které je možné v jednom průchodu aplikovat na jeden fragment rostl s vývojem grafického hardware. Stejně tak se zdokonalovaly způsoby, jak lze tyto textury kombinovat do výsledné barvy. Mechanismy práce byly stejné jako všude jinde ve fixním řetězci, tedy existovala sada předdefinovaných operací pro kombinaci textur a přepínači bylo možné je měnit a nastavovat jejich parametry. V programovatelném řetězci je celá tato fáze nahrazena zpracováním programátorem definovaného kódu, tzv. *fragment shaderu*.(lze se setkat také s označením pixel shader, např. v [DirectX9.0c 05]). Tento kód má na vstupu opět právě jeden fragment, spolu s jeho atributy a výstupem je barva fragmentu a v některých architekturách také hloubka fragmentu. Opět nelze fragmenty vytvářet, ani přímo odstraňovat. Nelze také **měnit pozici** fragmentu.

Závěrečným stádiem je určení barvy jednotlivých pixelů. V tomto stádiu dochází ještě k řadě dalších operací jako určení barvy mlhy, aplikaci alfa a stencil testů a v neposlední řadě také test hloubky a aplikace technik anti-aliasingu.

Principy programovatelného stínování

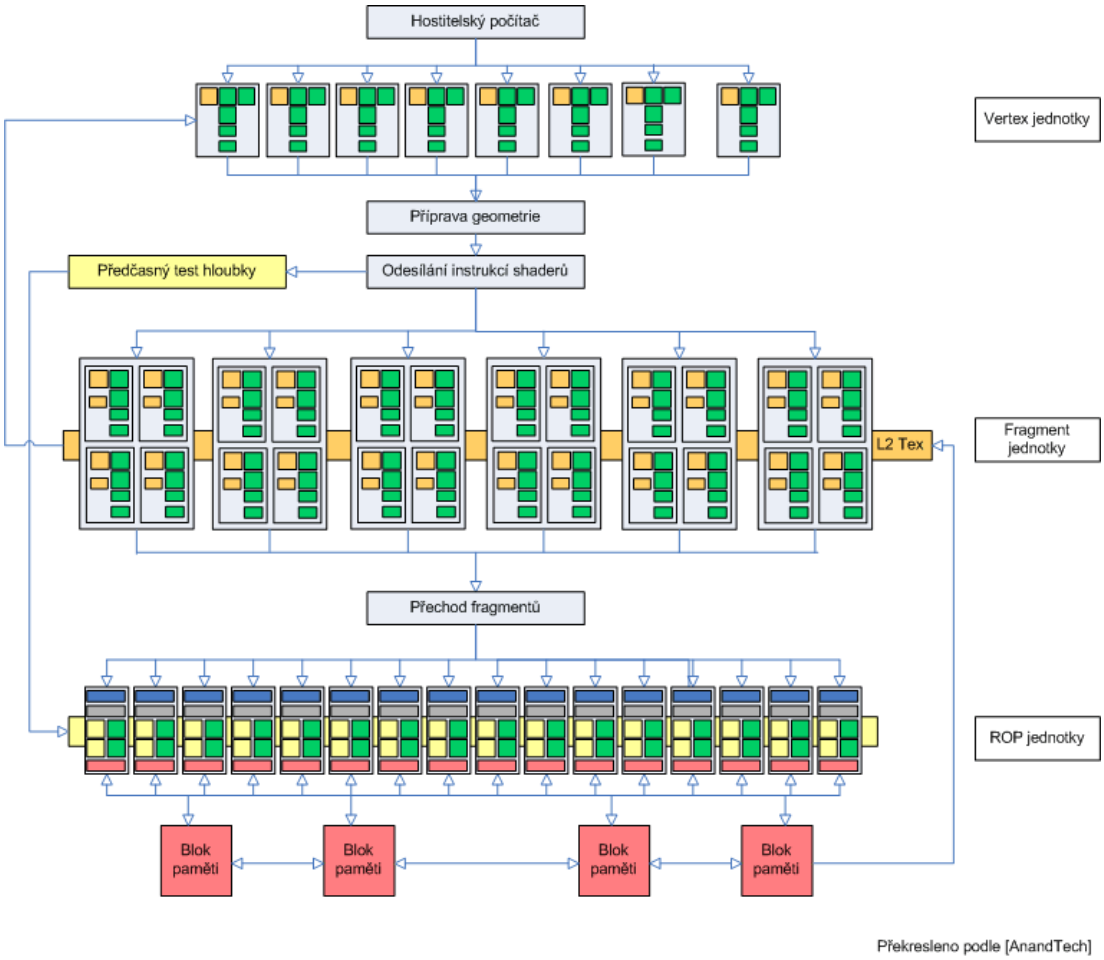
Programovatelným stínováním nazýváme práci s programovatelným řetězcem. Práce v tomto výpočetním prostředí se velmi liší od programování pro obecné procesory.

Zásadním rozdílem je, že nepíšeme program, který na vstupu dostává objekty ve scéně a na výstupu má matici pixelů výsledného obrazu, ale tvoříme dva oddělené programy, vertex shader a pixel shader.

Pro správné a efektivní používání vertex a pixel shaderů je nutné porozumět blíže způsobu, jak je programovatelný řetězec uvedený v předchozí sekci implementován.

Obrázek 1.2 ukazuje schéma nejnovější řady karet firmy Nvidia, přesněji architekturu G70 používanou v kartách řady GeForce 7800 GTX. Na schématu je vidět, že programovatelných jednotek, starající se o zpracování vertex a pixel shaderů, je velké množství. Konkrétně v této architektuře se jedná o 8 vertex jednotek a 24 fragment jednotek. Ve spodní části schématu vidíme také 16 tzv. ROP (Render Output Pipeline), což jsou jednotky, které se starají o operace v Obrázek 1.1 označené jako Vykreslení pixelů.

Při programování vertex a pixel shaderů tedy pracujeme s masivně paralelní SIMD architekturou. Takováto masivní paralelizace je umožněna omezeními na vstupy a výstupy vertex a fragment shaderů, jak byly uvedeny v předchozí sekci. Jedná se hlavně o předpoklad, že vertex shader zpracovává pouze jeden vrchol bez závislosti na ostatních. To samé platí o fragment shaderu a fragmentu, který zpracovává.



Obrázek 1.2: Schéma architektury G70

Aniž by bylo nutné zabíhat do úplných detailů, lze formulovat několik základních pravidel a odlišností pro programovatelné stínování:

- Výstupy vertex shaderu jsou interpolovány do vstupů fragment shaderu, aniž by grafický hardware vždy znal jejich sémantiku. Vertex a fragment shadery tedy na sebe „navazují“ a programátor si může předávat mezi nimi víceméně libovolná data.
- Vertex shader je spuštěn na každý vrchol zcela samostatně s vlastním kontextem.
- Fragment shader je spuštěn na každý fragment zcela samostatně s vlastním kontextem. Počet zpracovávaných fragmentů může být obrovský v řádu stovek miliónů na jeden snímek.
- Stále existují části zobrazovacího řetězce, které nelze přímo programovat a tak je třeba zajistit jejich správné nastavení pro spolupráci s vertex a fragment shadery.
- Vertex a fragment jednotky jsou specializované vektorové procesory, které mají speciální instrukční sadu přizpůsobenou pro grafické operace. To znamená, že operace až na 4 složkových vektorech trvají stejně dlouho jako operace na skalárech a existují speciální instrukce, jako například instrukce *dp4* provádějící skalární součin dvou 4 složkových vektorů nebo instrukce *lit*, která přímo provádí výpočet jednoduchého osvětlovacího modelu.

- Vertex a fragment shadery nejsou samostatné jednotky. Jinými slovy napsáním vertex a fragment shaderů práce nekončí. Pro jejich zavedení a spuštění v nějakém grafickém engine je potřeba ještě nastavit neprogramovatelné části grafického řetězce, implementovat přenos potřebných parametrů, zajistit kontrolu potřebné verze hardwarového profilu a v neposlední řadě určit pro jakou geometrii se mají tyto vertex a fragment shadery používat.
- Nelze přímo iniciovat spuštění vertex nebo fragment shaderu. Vertex a fragment shadery jsou spouštěny automaticky grafickým hardware jako reakce na existenci vrcholu nebo fragmentu. Pokud neexistuje žádný relevantní vrchol nebo fragment, vertex a fragment shadery se vůbec neprovedou. Celý zobrazovací řetězec tedy spustíme zasláním geometrie z aplikace.

Je ještě několik subsystému grafického hardware, o kterých jsem se v této kapitole nezmínil. Jedná se především o paměťový subsystém, který má také svá specifika a je opět optimalizován pro grafické operace, zejména výběr hodnot z textur.

Další podrobné informace o hardwarové architektuře lze nalézt například v [AnandTech]. Detailní popis zobrazovacího řetězce je uveden v [DirectX9.0c 05] a popisy interface lze nalézt v [DirectX9.0c 05, OpenGL 2.0 04].

1.5 Cíle práce

Cílem této práce bylo navrhnout a implementovat metodu pro zobrazování objemu, která by splňovala následující kritéria:

Rychlost zobrazovaní (c1)
Metoda by měla být použitelná pro zobrazování v reálném čase. Snahou bylo dosáhnout maximální snímkovací frekvence měřené ve snímcích za vteřinu. Při tvorbě práce byl brán ohled i na konzistenci snímkovací frekvence. Její přímé měření jsem ovšem neprováděl, neboť použité experimentální prostředí to přímo neumožňuje a neexistovala přiměřeně schůdná cesta, jak toto prostředí rozšířit.

Kombinace s jinými zobrazovacími metodami (c2)
Ve většině reálných aplikací vyvstává problém, jak zobrazovat najednou objemová data a ostatní, typicky polygonálně reprezentované objekty [Levoy 90]. Je proto výhodné, pokud metoda umí buď zobrazovat jak objemová, tak polygonální data nebo je schopná v jednom snímku zobrazit různé skupiny objektů různými metodami.

Implementace výhradně uvnitř shaderů (c3)
Výhodou používání shaderů kromě jejich rychlosti je fakt, že zatěžují hlavní procesor počítače jen minimálně, a tak jej uvolňují pro jiné výpočty. To je velmi důležité pro zobrazovací metody, neboť typicky v reálných aplikací tvoří zobrazování jen část výpočtů, které je třeba provádět. Zároveň je to velmi výhodné vzhledem k požadavku (c2), neboť současná běhová prostředí pro práci se shadery umožnují relativně snadno nastavovat aktivní shadery a měnit tak způsob zobrazování od objektu k objektu.

Samostatnost a přenositelnost(c4)
Jak bylo nastíněno v části o grafickém hardware, spuštění kódu pro programovatelný řetězec je relativně komplikovanou záležitostí zahrnující daleko více činností než jen napsání vlastního kódu. Z toho důvodu je přenositelnost vertex a fragment shaderů omezená, zejména u komplikovaných programů. V nedávné době se začaly objevovat první pokusy o standardizaci formátu, který by byl schopen zachytit všechny potřebné informace na jednom místě a jedním způsobem tak, aby množina souborů v tomto formátu byla samostatnou jednotkou, kterou lze zavést v každém enginu, který tento formát podporuje. Cílem pro kód napsaný v této práci je implementovat ho v tomto formátu a dosáhnout tak samostatnosti a přenositelnosti.

Hlavní přínosy práce by měly být zejména:

Vysoké snímkovací frekvence a kvalita zobrazení
Ambicí implementace bylo dosáhnout větších snímkovacích frekvencí než u předchozích prací [Krüger a Westermann 03] a srovnatelných výsledků, co se týče rychlosti i kvality zobrazení, se současnými prácemi [Green 05, Stegmaier et al. 05].

Implementace pomocí Shader Modelu 3.0
Při implementaci jsem se zaměřil pouze na nejnovější třídu grafického hardware pro stolní počítače a Shader Model 3.0, který podporují. To umožnilo oproti předchozím pracím výrazně snížit počet průchodů uvnitř shaderu a také poukázalo na možnosti implementace urychlovacích metod. Zároveň jsem ověřil výpočetní rychlost této generace karet a našel limity, kdy je třeba pro další zvýšení rychlosti a kvality použít urychlovací techniky a nelze se spoléhat pouze na výpočetní výkon.

Zhodnocení vhodnosti použité architektury
Současné architektury GPU jsou navrženy zejména pro zpracování polygonálních modelů a nejsou na první pohled příliš vhodné pro implementaci metod pro zobrazování objemu. Nicméně úspěšná implementace zobrazování objemu na GPU by byla velmi vítaná, jak v oblasti fotorealistického zobrazování, tak vědecké a průmyslové vizualizaci. Pokud by se ukázalo, že implementace není z nějakého důvodů možná, bylo cílem tyto potíže objektivně zhodnotit a navrhnout jejich možné řešení.

1.6 Rozvržení textu

V této úvodní kapitole jsem uvedl základní principy a pojmy z oblasti zobrazování objemu. Probral jsem také problematiku zobrazování v reálném čase a jeden možný způsob měření rychlosti zobrazování. Shrnut byl také vývoj a principy grafického hardware pro stolní počítače. Kapitolu uzavírá vytyčení cílů práce v podobě čtyř požadavků, formulace předpokládaných přínosů a rozvržení obsahu práce.

V kapitole 2 se podrobněji věnuji kategorii metod hledajících povrch. Je prezentována historie těchto metod, jejich obecné výhody a nevýhody. Na metodách Napojování isočar a Marching Cubes je prezentován princip přístupů z této oblasti. Nakonec je zhodnocena vhodnost použití pro implementaci na GPU.

V kapitole 3 jsou prezentovány základní metody z oblasti přímého zobrazování objemu. Kapitola je rozdělena na dvě části. První část se věnuje optickým modelům a druhá zobrazovacím metodám. Obdobně jako v kapitole 2 je shrnuta historie zobrazovacích postupů a jejich obecné výhody a nevýhody.

Kapitoly 2 a 3 jsou souhrnně koncipovány tak, aby poskytly kompletní přehled o oblasti zobrazování objemu. Přehled historie v obou kapitolách mapuje vývoj zobrazování objemu od počátku až po současnost a věnuje se i metodám, které nejsou přímo vysvětlovány.

Kapitola 4 prezentuje návrh metody pro implementaci na GPU. Na úvod je metoda uvedena do kontextu zobrazovacího systému tak, jak je uveden v §1.2. Poté je rozebrán návrh jednotlivých částí a prezentováno použití pokročilých optických modelů. Kapitolu uzavírá část o přípravě testovacích a vývojových dat.

Kapitola 5 je věnována podrobně implementaci postupu navrženého v Kapitole 4. Nejprve je detailně rozebrán princip práce s programovatelným řetězcem a je vysvětlena obecná struktura kódu. Následuje představení použitého vývojového a experimentálního prostředí, jeho výhod a omezení. V dalších podkapitolách upozorňuji na některé zajímavé části, které se týkají optických modelů, geometrických a jiných výpočtů. Nakonec je vysvětlen způsob, jak jsou v implementaci ukládána objemová data a celá kapitola je uzavřena výsledky testování metody a jejich diskuzí.

Závěrečná Kapitola 6 podává shrnutí celé práce. Jedná se o zhodnocení dosažených výsledků a zdůraznění vlastního přínosu. Poté jsou nastíněny možnosti dalšího rozvoje a práce je ukončena přehledem citované literatury.

V textu nejsou překládány názvy obecně známých metod (*Marching Cubes*, *Dividing Cubes*) a pojmů, pro které zatím neexistuje uspokojivý český ekvivalent (*vertex*, *fragment*, *shadery*, *GPU*). V případě, že neexistuje ustanovený překlad cizojazyčného pojmu, ale lze jej uspokojivě se zachováním významu přeložit, je u jeho prvního výskytu uveden ekvivalent v původním jazyce v závorkách.

Kapitola 2

Metody hledající povrch

Cílem v této kapitole není dopodrobna vysvětlit každou z metod založených na principu hledání povrchu. Cílem je ukázat, jak tato skupina metod pracuje a v čem se liší od metod přímého zobrazování.

V úvodu kapitoly (§2.1) rozeberu základní kroky těchto metod a uvedu výhody a nevýhody toho přístupu jako celku. Dále jsou probrány dva algoritmy zajišťující stěžejní část, tedy konstrukci povrchu z objemových dat. Různých algoritmů pro konstrukci povrchu lze samozřejmě v literatuře nalézt více. Uvedené dva postupy představují základní metody a ostatní další algoritmy na nich stavějí či je rozšiřují nebo jsou myšlenkově podobné. V částech o historii jsou uvedeny i základní principy a reference na tyto rozšiřující práce a to takovým způsobem, aby obě kapitoly dohromady tvořily úplný přehled vývoje v oblasti metod hledajících povrch. Na konci kapitoly v §2.3 se věnuji možnostem implementace na GPU.

2.1 Úvod

Společnou myšlenkou metod hledajících povrch je zobrazovat pouze části objemu odpovídající zadané hodnotě. Tento požadavek vychází z reálných, zejména medicínských aplikací. Lidské tělo si lze totiž velmi dobře představit jako skupinu orgánů a tkání, z nichž každá má dobře definovanou a homogenní hustotu. Zároveň důležitou informací je právě tvar rozhraní těchto tkání.

Tento myšlený povrch se pak metody snaží aproximovat polygonální sítí, která je následně zobrazena běžnými zobrazovacími technikami, včetně řešení viditelnosti.

Výhody a nevýhody

Již při úvodní formulaci problému naráží tyto metody na několik obtíží. Jde zejména o způsob, jak formulovat, pro které buňky objemu se má generovat polygonální povrch. Nejjednodušším přístupem by bylo definovat prahovou hodnotu f_p a

zobrazit povrch pro každou buňku, která má hodnotu $f \geq f_p$. To ovšem znamená, že lze najednou zobrazit pouze jeden povrch pro jednu prahovou hodnotu. Navíc spousta geometrie, která je vygenerována, nebude díky zakrytí vidět.

Jinou možností by bylo specifikovat okno pomocí dvou hodnot f_{bot} a f_{top} a pracovat s buňkami, pro jejichž hodnotu f platí $f_{bot} \leq f \leq f_{top}$. Pokud ovšem okno nastavíme příliš úzké, budou se v povrchu objevovat mezery, naopak bude-li příliš silné, dostáváme se do stejného problému, jako v případě jednoho prahu.

Oba výše zmíněné přístupy navíc trpí defekty, které plynou z nucené binární klasifikace, tedy nutnosti vždy rozhodnout, zda má pro danou buňku být geometrie vygenerována či nikoliv. Důsledkem jsou falešné „vypouklé“ části, kde byla nesprávně zahrnuta dodatečná buňka nebo falešné mezery, kde byla nesprávně buňka opominuta. Tyto defekty mají obzvláště velký dopad právě v medicínských aplikací, kde by mohly být nesprávně považovány za rys dat.

Společnou výhodou těchto metod je rychlost. Jakmile je jednou povrch zkonstruován, lze jej rychle zobrazovat, bez ohledu na pozici pozorovatele a světelné podmínky.

Stínování a optické modely objemu

Metody hledající povrch se nesnaží modelovat objem jako celek a nesnaží se napodobovat jeho reálné optické vlastnosti. Nicméně pro zlepšení vnímání vlastností zobrazovaného povrchu se často aplikují empirické stínovací modely, například Phongův model, který je schopný simulovat difusní i lesklý odraz světla na povrchu.

Pro aplikaci těchto modelů je nutné znát normálu povrchu v příslušném bodě. Bylo by možné ji spočítat přímo jako normálu polygonální sítě. Tento přístup se ovšem nepoužívá, neboť dává příliš hrubě vypadající výsledky. Častěji se normála aproximuje pomocí centrálních diferencí přímo z objemových dat. Tento postup je vysvětlen v §4.4.

2.2 Konstrukce povrchu

V této podkapitole uvedu dva algoritmy generování polygonální sítě z objemových dat. Již se nebudu zabývat způsoby, jak takto vygenerovaný povrch zobrazit.

2.2.1 Napojování isočar

Metoda napojování isočar je založena na myšlence nalézt pro každý dvoudimensionální průřez objemu množinu uzavřených isočar, které odpovídají zadané prahové hodnotě a poté, řez po řezu, tyto isočáry napojovat a vytvářet tak isoplochu.

Ukazuje se, že tento přístup obsahuje několik obtížných míst. V době publikování prvních prací byl problém již s první krokem, tedy hledáním isočáry ve dvourozměrném průřezu. Zejména v hodně zašuměných datech nebo v datech s nízkým kontrastem bylo obtížné nalézt isočáry tak, aby tvořily množinu uzavřených křivek. Často bylo zapotřebí zásahu operátora, aby sporná místa opravil [Elvins 92].

Druhý problém vzniká při napojování isočar v sousedních řezech. V některých případech nelze jednoznačně rozhodnout, které části mají být k sobě

napojeny a vzniká tak chybný povrch, což je zejména v medicínských aplikacích nepřijatelné. Tomuto problému lze zabránit dostatečně hustým vzorkováním tak, aby nevznikala situace napojení M:N (tedy případ, kdy v jednom řezu je $M > 1$ a v sousedním $N > 1$ uzavřených isočar). Nevýhodou je také poměrně komplikovaná implementace, která obsahuje řadu speciálních případů.

Historie

První pokusy se zobrazováním objemových dat vedly na použití tehdy známých algoritmů pro generování kontur na dvourozměrných průřezech a jejich napojovaní tak, aby vznikl polygonální povrch ve třírozměrném prostoru. Poprvé byl tento přístup publikován v [Keppel 75], později zdokonalen ve [Fuchs et al. 77] a dále rozvíjen např. v [Ekoule et al. 91]. V dalších letech nenajdeme mnoho prací publikovaných na stěžejních konferencích. Je to zejména proto, že problém hledání povrchu byl později vhodněji formulován a byly vyvinuty nové přístupy, které umožňovaly hledat a zobrazovat isoplochy v objemu efektivněji a v lepší kvalitě při použitích jednodušších algoritmů.

Princip

Následující popis vychází z práce [Ekoule et al. 91]. Tato práce patří mezi pokročilejší metody v oblasti napojování isočar a umí řešit i velmi komplikované situace. Algoritmus lze rozdělit na tři základní kroky:

- Výpočet isočar v jednotlivých řezech. Isočáru reprezentujeme jako lomenou čáru, tedy posloupností bodů. Jeden řez může obsahovat více uzavřených isočar.
- Výpočet isoplochy ze sousedních řezů s isočárami. Isoplocha se vytváří triangulací mezi sousedními isočárami. Základním požadavkem je, aby vytvářené trojúhelníky měly rozumný tvar, tedy nebyly příliš tenké a protáhlé.
- Zobrazení vytvořené trojúhelníkové sítě

Největší komplexita algoritmu se skrývá v druhém kroku. K problému vytváření isoplochy lze přistoupit dvěma způsoby. Buď hledáním optimální triangulace nebo použitím heuristické metody. Jak je v takovýchto případech obvyklé, optimální triangulace dává dobré výsledky, ale její výpočet je extrémně pomalý. Naopak heuristický proces je rychlejší, ale v některých případech generuje nesprávné výsledky. Následující algoritmus používá heuristickou metodu, která je speciálně upravena tak, aby generovala korektní výsledky i v obtížných situacích. Dalším problémem, který je nutné vyřešit, jsou topologicky obtížné konfigurace isočar ve dvou sousedních řezech.

V druhém kroku tedy postupujeme následujícím způsobem:

1. Pracujeme vždy na dvou sousedních řezech, které si připravíme.
2. Rozhodneme, o který druh napojení se jedná:
    a. **1:1, obě kontury konvexní**. V tomto případě je postup nejjednodušší. Určíme konturu, která má méně bodů. Pro každý bod této kontury pak v sousedním řezu najdeme bod, který je mu nejblíže. Poté již jen doplníme hrany od nespárovaných vrcholů tak, aby vznikla triangulace.
    b. **1:1, jedna nebo obě kontury jsou nekonvexní**. Výše uvedený heuristický způsob napojovaní dává dobré výsledky pouze pro podobné konvexní isočáry. Proto v tomto případě promítneme nekonvexní isočáru na její konvexní obal. Poté provedeme triangulaci

podle bodu a. Transformace isočáry na její konvexní obal dává jednoznačnou korespondenci mezi body na isočáře a odpovídajícími body na konvexním obalu. V momentě, kdy máme hotovou triangulaci mezi konvexními obaly, lze tuto korespondenci přímo použit k triangulaci původních isočar.

c. **1:N**. V tomto případě je postup o něco složitější. V jednom řezu máme tedy isočáru Q a v druhém sadu isočar P_j, $j \in [1, N]$. Nejprve z isočar P_j vytvoříme společnou uzavřenou isočáru P, která je jejich obalem. Poté z isočar Q a P vytvoříme pomocnou isočáru S, která leží v polovině jejich vzdálenosti. Nyní již pouze provedeme napojení isočar Q a S podle bodu a. nebo b. Na druhé straně provedeme n napojení podle bodu a. nebo b. Vždy spojujeme isočáru P_j s isočárou S.

d. **M:N**. Tento případ je jednoznačně nejsložitější. Problémem zde je, které z isočar v jednom řezu napojit na které isočáry v druhém řezu. Budeme předpokládat, že pokud mají být dvě isočáry v sousedních řezech napojeny, jsou přibližně na stejném místě. To lze zajistit dostatečně hustým vzorkováním. Na Obrázek 2.1 vlevo je případ, kdy tento předpoklad neplatí a kdy nelze jednoznačně rozhodnout o správném napojení. Tento případ lze ovšem vyřešit přidáním dodatečného řezu, jako je ukázáno na Obrázek 2.1 vpravo. Jestliže se tedy spolehneme na uvedený předpoklad, lze isočáry mezi sebou přiřadit na základě jejich vzájemného zakrytí. Určíme tedy zakrytí všech dvojic isočar v různých řezech a vybereme množiny, které mají být sobě přiřazeny. Tyto množiny nemusí korespondovat 1:1, ale může také nastat případ 1:N. Oba dva ale již umíme řešit z předchozích kroků.

Obrázek 2.1: Nejednoznačné napojení isočar. Převzato z [Ekoule et al. 91]

2.2.2 Marching Cubes

Algoritmus Marching Cubes opět slouží k vytvoření polygonální aproximace isoplochy v objemu. Tentokrát je povrch ovšem přímo generován jako trojúhelníková síť ve třech dimenzích.

Výhodou algoritmu je snadnost implementace a to i přímo v hardware. Příjemnou vlastností také je, že generuje trojúhelníky vhodných tvarů, tedy trojúhelníky maximálně podobné rovnostranným, bez dlouhých a protáhlých hran.

Jednou z mála nepříjemností je, že počet generovaných trojúhelníků je velmi vysoký a trojúhelníky jsou často velmi malé, někdy i sub-pixelových velikostí.

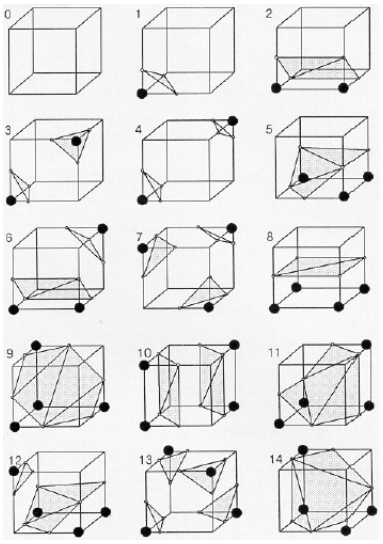
Co se týče rychlosti zobrazovaní, chová se tento algoritmus stejně jako ostatní metody hledající povrch. Tedy, jakmile je jednou povrch zkonstruován, lze jej rychle zobrazovat bez ohledu na pozici pozorovatele a světelné podmínky.

Historie

První přístup, podobný algoritmu Marching Cubes, je z doby jen o málo pozdější než metoda napojování isočar. Jedná se o algoritmus neprůhledných kostek, neboli Cuberille [Herman a Liu 79]. V této práci jsou hledány buňky objemu, jejichž vrcholové hodnoty leží na obou stranách explicitně zadaného prahu. Pro každou takovou buňku je vygenerován polygonální kvádr, který je následně zobrazen. Tento směr byl dále rozvíjen a zdokonalován a v roce 1987 byla publikována právě metoda Marching Cubes [Lorensen a Cline 87]. Její princip je stejný jako v práci Hermanna a Liu. Povrch je ovšem souvislý a je tvořen trojúhelníky, které jsou generovány podle způsobu, jak isoplocha buňkou prochází. Tento postup byl dále zkoumán a vznikly další, na něm založené, algoritmy jako Marching Tetrahedra [Shirley a Tuchman 90], který každou buňku rozdělí dále na pět, šest či dvacet čtyři čtyřstěnů a generuje trojúhelníky podle polohy isoplochy v těchto čtyřstěnech. Známý algoritmus této kategorie Dividing Cubes byl prezentován v práci [Cline et al 88]. Tato metoda si všímá skutečnosti, že mnoho generovaných polygonů je menších než pixel a spolu s hierarchickým dělením buněk používá tento fakt k vykreslování elementů objemu jako povrchových bodů.

Princip

Následující popis vychází z práce [Lorensen a Cline 87]. Úkolem algoritmu je vygenerovat trojúhelníkovou síť, která aproximuje isoplochu zadanou uživatelsky definovaným prahem. Uvažme krychli takovou, že ve všech osmi rozích známe vstupní hodnoty a chceme zjistit, jak povrch touto buňkou prochází. Pro každý z osmi vrcholů jsou možné dva stavy: hodnota v něm je větší nebo rovna zadanému prahu (1) a vrchol tedy leží „uvnitř“ nebo na isoploše. Nebo je hodnota ostře menší (0) než zadaný práh a pak vrchol leží „vně“ isoplochy. To nám při osmi vrcholech dává celkem $2^8 = 256$ možných konfigurací. Autoři algoritmu ukázali, že symetriemi lze tento počet zredukovat na 15 konfigurací. Tyto konfigurace jsou zobrazeny na Obrázek 2.2.



Obrázek 2.2: Konfigurace v algoritmu Marching Cubes. Převzato z [Lorensen a Cline 87].

Před vlastním započetím práce algoritmu je potřeba připravit tabulku těchto konfigurací. Vytvoříme strukturu, do které se budeme odkazovat osmi bitovým indexem, kde každý bit reprezentuje stav jednoho vrcholu. Tabulka obsahuje pro každou ze 14 konfigurací (konfigurace 0 negeneruje žádné trojúhelníky) seznam hran, které jsou povrchem protnuty.

Vlastní algoritmus Marching Cubes lze pak popsat následujícími kroky:

1. Čtyři sousední řezy jsou načteny do paměti.
2. Procházíme prostřední dva řezy a vždy vytvoříme krychli tak, že čtyři její vrcholy leží v jednom řezu a čtyři v druhém řezu.
3. Spočteme index této krychle tak, že porovnáme hodnoty objemových dat ve vrcholech krychle se zadaným prahem.
4. Tento index použijeme do tabulky konfigurací a načteme seznam hran, které jsou povrchem protnuty.
5. Z hodnot ve vrcholech hran určených předchozím krokem lineárně interpolujeme přesnou pozici průsečíku s povrchem na každé z těchto hran.
6. Spočteme normálu ve vrcholech krychle pomocí centrálních diferencí (z tohoto důvodu potřebujeme pracovat se čtyřmi řezy, ne jen se dvěma). Tyto normály pak interpolujeme do vrcholů vygenerovaných trojúhelníků.
7. Na výstup pošleme trojúhelníky a normály ve vrcholech.

2.3 Komentář

Na závěr kapitoly o metodách hledajících povrch zhodnotím jejich vhodnost pro implementaci na GPU.

Cílem této práce bylo implementovat metodu zobrazování objemu zcela pomocí GPU, tedy tak, aby co nejmenší objem prací byl prováděn na hlavním procesoru počítače. Zobrazování povrchů, které vygenerují metody hledající povrch lze velmi dobře hardwarově urychlovat, a také toho je často využíváno. Takto je ovšem urychlováno pouze zobrazování vygenerovaného povrchu a vlastní průchod objemovými daty a extrakce povrchu je typicky prováděna na hlavním procesoru.

Pokud bych chtěl některou z metod hledající povrch kompletně implementovat uvnitř shaderů, znamenalo by to procházet přímo na GPU objemová data a konstruovat z nich polygonální povrch.

Tento typ průchodu daty je ovšem pro výpočetní model GPU nevhodný. Jak bylo naznačeno v úvodní kapitole, současné GPU dosahují vysokého výkonu zejména kvůli proudové architektuře a vysokému stupni paralelizace. Tato paralelizace ovšem probíhá na úrovni zpracování vrcholů a fragmentů. Princip spočívá v tom, že zpracování jednotlivých vrcholů a fragmentů lze efektivně oddělit a tak výpočty nad nimi masivně paralelizovat.

Proto přímo implementovat průchod objemem na současných GPU architekturách nelze. Programátor má možnost pouze ovlivňovat zpracování existujících jednotlivých vrcholů a fragmentů. Současný výpočetní model také neumožňuje uvnitř vertex shaderů vytvářet nové vrcholy nebo existující rušit. Aby bylo ale možné implementovat průchod objemem, je nutné umět vytvářet nové polygony a tedy i nové vrcholy. Toto omezení lze částečně obejít tím, že se na GPU ke zpracování odešle sada degenerovaných polygonů a uvnitř vertex shaderů se mění pozice jejich vrcholů tak, aby vzniklo požadované těleso. Stále ovšem zůstává problém, že předem nevíme, kolik polygonů bude potřeba a také uvnitř vertex

shaderů nemáme k dispozici informaci o topologii polygonů, tedy informaci, které polygony jsou tvořeny kterými vrcholy.

Při implementaci během této práce jsem měl také na mysli zobrazování „amorfních“ objektů, jako jsou mraky a oheň, pro které je použití metod hledajících povrch nevhodné.

Proto, když jsem zkombinoval uvedené technické a estetické potíže, ukázaly se metody hledající povrch jako nevhodný základ pro tuto práci.

Všimněme si ještě jednoho faktu, který plyne z diskuse v předchozích odstavcích. Díky konstrukci současných GPU lze obecně říci, že jakákoliv metoda, která používá jako primární průchod objemovými daty a při tomto průchodu vytváří obraz na průmětně, je nevhodná pro implementaci uvnitř shaderů. Tento fakt bude hrát důležitou roli u některých přímých zobrazovacích metod, jako například u metody otisků.

Kapitola 3

Přímé zobrazovací metody

V této kapitole se budeme zabývat druhou velkou skupinou přístupů k zobrazování objemu, a to jsou přímé zobrazovací metody. Přitom se budu volně držet rešerše uvedené v pracích [Kajiya 84 a Von Herzen, Krüger a Westermann 03].

Škála vyvinutých přístupů se zde liší ještě více než u metod hledajících povrch. Protože jako základ pro tuto práci byl vybrána právě metoda z oblasti přímých zobrazovacích metod, je této kapitole věnován větší prostor než kapitole 2.

Po úvodní části (§3.1), kde opět proberu základní principy a výhody a nevýhody této třídy metod, následuje kapitola o optických modelech (§3.2). Zde se budeme zabývat způsoby, jak simulovat průchod světla prostředím. Tento krok je u přímých zobrazovacích metod daleko důležitější, než u metod hledajících povrch.

V §3.3 o zobrazovacích postupech jsou představeny čtyři základní přístupy. U každého je shrnuta jeho historie a obdobně jako v kapitole 2 takovým způsobem, aby souhrn částí o historii dával celkový přehled vývoje oblasti přímých zobrazovacích metod. V §3.3.1 se zabývám metodami průřezů. Jde o myšlenkově velmi jednoduchý přístup navržený přímo pro implementaci v hardware. Druhými v pořadí v §3.3.2 jsou metody otisků. Tento princip a práce z něj vycházející jsou velmi zajímavé a ukazují širokou škálu možností, jak k problému zobrazování objemových dat přistoupit. V předposledním §3.3.3 je představena skupina metod založených na faktorizaci. Tyto metody reprezentují směr vývoje, který byl udáván snahou o využití prostorové koherence dat u existujících metod. V §3.3.4 jsou představeny přístupy založené na vrhání paprsku, které tvoří také základ metody navrhované v této práci. Část o zobrazovacích postupech uzavírá §3.3.5, kde stručně zmíním ještě další přístupy, které se během vývoje objevily.

Na závěr kapitoly v §3.4 je opět uveden komentář ke vhodnosti implementace na GPU.

3.1 Úvod

Přímé metody zobrazování objemu jsou historicky o málo starší, než přístupy založené na hledání povrchu. Snahou při návrhu přímých zobrazovacích metod bylo odstranit některé defekty hledání povrchu a také umožnit práci s objemovými daty i v jiných oblastech.

Metody přímého zobrazování objemu odstraňují použití polygonálních primitiv a snaží se objem zobrazit přímou projekcí na průmětnu. Tímto přístupem je možné se zbavit velkého množství problémů, zejména nutnosti klasifikace buněk na přítomnost či nepřítomnost povrchu a s tím spojených artefaktů. Zároveň je možné zobrazovat „amorfní“ objemy, kde pojem povrchu je nejasný či nevhodný pro vizualizaci, např. přírodní fenomény jako mraky a oheň.

Je zde ovšem také mnoho problémů nových, z nichž některé se ukazují jako velmi složité.

Nyní již nemáme k dispozici povrch, který bychom mohli přímo stínovat a je nutné vyvinout nějaké postupy pro způsob přenosu objemových dat na zobrazitelné hodnoty, jako je průhlednost a barva. Tento bod se ukazuje jako klíčový, a proto je mu věnován celý §3.2.

Dalším problémem je zřejmá velká výpočetní náročnost těchto metod. Při konstrukci povrchu stačilo jednou projít celý objem, vytvořit povrch a pak ho opakovaně zobrazovat. Zde ovšem podobnou „mezilehlou“ reprezentaci nemáme a je tedy nutné objem opakovaně procházet při změně pozice pozorovatele a/nebo světelných podmínek. Je to proto, že díky přímému promítání je každý vygenerovaný obraz závislý právě na pozici pozorovatele. Zajímavý způsob, jak tento problém řešit a přitom stále ještě pracovat s přímým zobrazování nabízí třída metod založených na faktorizaci (§3.3.3).

Lze tedy shrnout, že přímé zobrazovací metody jsou schopny dosahovat velmi kvalitních výsledků, jsou flexibilní díky možnosti využití široké škály optických modelů, ale jsou velmi výpočetně náročné.

3.2 Optické modely

Při použití přímých zobrazovacích metod je důležitým krokem způsob, jakým mapujeme vstupní hodnoty na zobrazitelné veličiny, jako barva a průhlednost.

K tomuto problému lze přistoupit dvěma způsoby. Buď se budeme snažit simulovat skutečné fyzikální vlastnosti nějaké látky nebo budeme mapování tvořit empiricky tak, abychom zobrazili či zdůraznili požadované rysy objemu.

V oblasti fotorealistického zobrazování se budeme držet fyzikálních vlastností simulovaného fenoménu. I když i zde lze nalézt empirické přístupy. Bývá to z důvodu výpočetní složitosti fyzikálně založených metod. V oblasti vědecké vizualizace lze nalézt argumenty pro ospravedlnění obou přístupů.

Výhoda simulace fyzikálních vlastností určité látky spočívá v tom, že lidé pracující s výslednými obrázky či animacemi, mají s touto látkou reálné zkušenosti a bývají schopni odvodit dodatečné vlastnosti, které nejsou přímo zřejmé. Tato vlastnost se může ovšem ukázat jako fatální, pokud by naše simulace vlastností látky nebyla ve shodě s fyzikální realitou.

Empirické mapování má naopak výhodu, že lze přímo zobrazit některé vlastnosti, které se ve fyzikální realitě vizuálně neprojevují. Nevýhodou naopak je, že uživatelé nejsou na použité mapování zvyklí a je potřeba přípravy a tréninku, aby byli schopni jeho vlastností využít. Velmi dobrým příkladem je práce [Sabella 88].

3.2.1 Simulace vlastností reálných látek

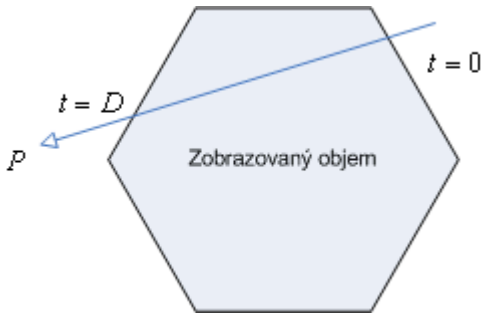
Pro napodobení vizuálních vlastností libovolné reálné látky musíme umět simulovat způsob interakce této látky se světlem. Budeme se tedy zajímat o to, jak se světlo chová v médiu, jak toto chování matematicky formalizovat a jak získané vzorce využít.

Nejprve proberu, od nejjednodušších k nejsložitějším, fyzikálně reálné modely šíření záření v prostoru a nakonec zmíním jeden empirický model, který se snaží napodobovat násobný rozptyl, ovšem nikoliv pomocí fyzikálně reálných výpočtů, ale pomocí empirického vzorce.

Velmi dobrý přehled optických modelů a jejich odvození je uveden v [Max 95] a této práce se v následujícím výkladu držím. Podrobná odvození obdobných rovnic lze nalézt také například v pracích [Kajiya a Von Herzen 84, Blinn 82, Sabella 88]. Obecně tyto práce vycházejí z fyzikálních teorií o šíření záření, např.[Chandrasekhar 50].

Chování záření v prostoru lze popsat jeho pohlcováním, vyzařováním a rozptylem od malých částic média, jako jsou například kapičky vody, prachové částice nebo i jednotlivé molekuly látky. V následujících odstavcích popíši geometricky optické chování takovýchto částic a z nich odvozené rovnice pro popis průchodu světla látkou.

Pokud nebude řečeno dále jinak, budeme studovat průchod záření látkou v situaci zobrazenou na Obrázek 3.1. Záření sledujeme podél nějakého paprsku *P*, který je parametrizovaný parametrem *t*. V bodě odpovídajícím $t = 0$ záření do objemu vstupuje a v bodě odpovídajícím $t = D$ objem opouští.



Obrázek 3.1: Situace při průchodu světla objemem

Dále v následujících odvozeních předpokládám, že médium je spojité, a že částice jsou nekonečně malé, tedy přesněji, že události pohlcování, vyzařování a rozptylování se odehrávají v každém nekonečně malém segmentu paprsku *P*.

Pohlcující model

Nejjednodušším médiem je látka složená z částic s vlastnostmi perfektního černého tělesa, které veškeré incidentní světlo pohltí, nic nevyzařuje ani nerozptyluje. Nyní odvodíme model šíření světla v takovéto látce.

Předpokládejme, že částice v látce jsou dokonalé identické koule o poloměru *r* a promítnuté na dvourozměrnou průmětnu mají tedy plochu $A = \pi \cdot r^2$. Uvažme nyní válcový pás se základnou *B* a výškou Δs. Plochu základny *B* označme *E*. Záření skrze tento pás prochází ve směru válce. Označme ρ počet částic na jednotkový objem látky. Potom, neboť objem toho pásu je $V = E \cdot \Delta s$, lze v něm

nalézt $N = \rho E \Delta s$ částic. Pokud bude Δs malé, je také malá pravděpodobnost, že se průměty částic na základnu *B* budou protínat. Pak plocha, kterou průměty částic na základnu *B* zabírají, je přibližně $N \cdot A = \rho E \Delta s A$. Tedy část záření, která projde základnou je úměrná $\rho E A \Delta s / E = \rho A \Delta s$.

Nyní uvažme tento výraz v limitě pro $\Delta s \to 0$. Dostáváme tak diferenciální rovnici

$$\frac{dI}{ds} = -\rho(s) A I(s) = -\tau(s) I(s) \tag{3.1}$$

kde:

- $I(s)$ je intensita záření v bodě, který odpovídá parametru *s*
- $\rho(s)$ je počet částic na jednotkový objem v bodě, který odpovídá parametru *s*

Tato rovnice říká, jaký je přírůstek (změna) intensity záření v diferenciálně malém válcovém pásu. Člen $\tau(s)$ je označován jako koeficient útlumu. Řešením této diferenciální rovnice je výraz

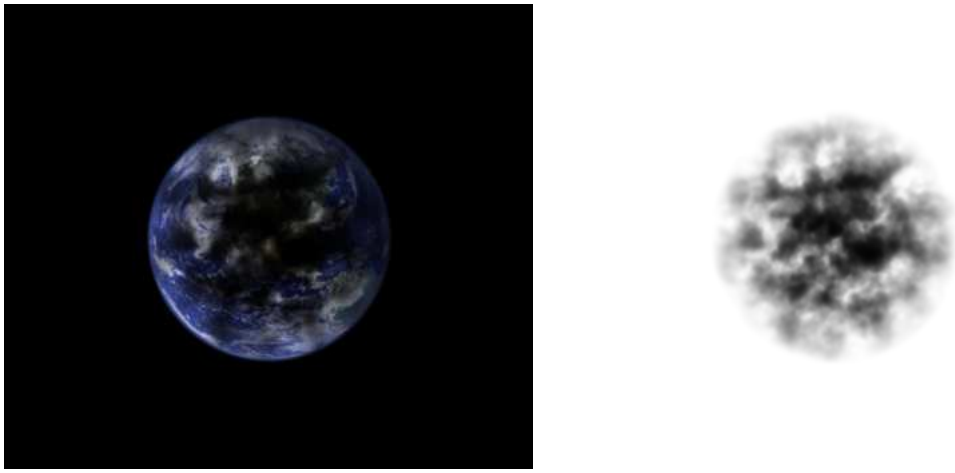
$$I(s) = I_0 \cdot \exp\left(-\int_0^s \tau(t)\, dt \right) \tag{3.2}$$

kde

- I_0 je intesita záření v bodě $s = 0$, tedy v bodě, kdy záření do objemu vstoupilo.

Při řešení bylo využito faktu, že pravděpodobnost $P(0,V)$, že v objemu *V* nebude žádná částice, lze vyjádřit Poissonových rozdělením $P(0,V) = e^{-N}$, kde stejně jako výše *N* je předpokládaný počet částic v objemu *V*.

Často bývá v podobných výrazech otočen smysl sledovaného paprsku. V rovnici (3.2) je bod pro $t = 0$ nejdále od pozorovatele a bod pro $t = s$ nejblíže k pozorovateli.



Obrázek 3.2: Použití pohlcujícího modelu šíření světla

Obrázek 3.2 byl vytvořen zobrazovací metodou implementovanou v této práci, za použití výše uvedeného modelu šíření světla. Obrázek nalevo ukazuje, jak procedurálně definovaný mrak pohlcuje světlo přicházející zpoza objemu z obrázku

Země. Obrázek napravo ukazuje, jak lze pomocí pohlcujícího modelu vytvářet obrázky napodobující rentgenové snímky.

Vyzařující model

Dalším druhem velmi jednoduchého média je látka, která pouze vyzařuje světlo a ani jej neodráží a ani nepohlcuje. Fyzikálně je takováto látka nereálná, ale nejvíce se jí blíží například plamen ohně.

Opět předpokládejme, že látka je složena z částic, které jsou identické koule. Dále předpokládejme, že jsou zcela průhledné a tedy ani neodrážejí ani nepohlcují světlo. Každá taková částice nechť vyzařuje s intesitou C na jednotku promítnuté plochy. Potom podle vzorce z minulé sekce dostáváme, že základna válcového pásu vyzařuje s intensitou $C\rho AE\Delta s$ neboli $C\rho A\Delta s$ na jednotkovou plochu. Opět pro $\Delta s \to 0$ dostáváme diferenciální rovnici:

$$\frac{dI}{ds} = C(s)\rho(s)A = C(s)\tau(s) = g(s) \tag{3.3}$$

kde:

- $C(s)$ je intensita vyzařování v bodě, který odpovídá parametru s na paprsku.od pozorovatele.

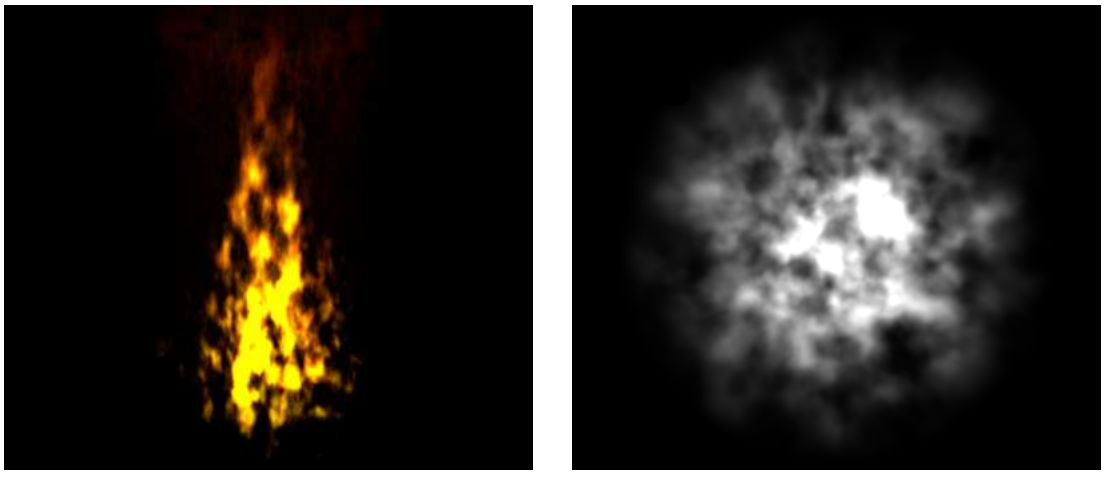
Člen $g(s)$ je nazýván zdrojový člen a obecně se používá k vyjádření množství vyzařovaného světla. V dalších složitějších modelech do tohoto členu zahrneme i vyzařování způsobené odrazem od vnějších zdrojů světla. Řešení této diferenciální rovnice je jednoduché, ve tvaru:

$$I(s) = I_0 + \int_0^s g(t)\,dt \tag{3.4}$$

kde

- $g(t)$ je zdrojový člen.

Obrázek 3.3 ukazuje použití vyzařujícího modelu světla. Obrázek napravo pracuje pouze s vyzařovanou intenzitou. Obrázek nalevo ukazuje použití pro modelování plamene. Látka vyzařuje různou barvu podle polohy. Uvnitř je barva žlutá a směrem k okrajům plamene přechází do oranžové až červené.



Obrázek 3.3: Použití vyzařujícího modelu šíření světla

Všimněme si jednoho zásadního rozdílu od pohlcujícího modelu světla. V pohlcujícím modelu mohla intesita dosáhnout maximálně hodnoty, která přichází zpoza objemu. Naopak u vyzařujícího modelu toto omezení není a v místech, kde je materiál dostatečně hustý, snadno dojde k překročení zobrazitelného rozsahu hodnot. Tento efekt lze na Obrázek 3.3 pozorovat uvnitř látky. Tento efekt je ovšem u silně vyzařujících materiálů obvyklý, neboť i lidské oko je schopné se adaptovat jen na omezený rozsah hodnot jasu (i když tento rozsah je řádově vyšší než použitých 256 hodnot).

Vyzařující a pohlcující model

Logickým krokem při vytváření dalších, složitějších modelů šíření světla, je sloučit oba předchozí modely do jednoho. V této sekci se tedy budeme zabývat látkou, která pohlcuje i vyzařuje světlo.

Do diferenciální rovnice, která nám popisuje přírůstek intensity světla v diferenciálně malém válcovém pásu, tedy zahrneme jak člen pro útlum, tak člen pro vyzařování. Rovnice má tedy tvar:

$$\frac{dI}{ds} = g(s) - \tau(s) I(s) \tag{3.5}$$

kde

- $g(s)$ je zdrojový člen. Nyní budeme, na rozdíl od předchozí sekce, předpokládat, že tento člen má obecný tvar a nemusí nutně být ve formě $g(s) = C(s)\tau(s)$

Tuto rovnici lze několika aritmetickými úpravami převést do tvaru:

$$\frac{d}{ds}\left[I(s) \exp\left(\int_0^s \tau(t)\, dt \right) \right] = g(s) \exp\left(\int_0^s \tau(t)\, dt \right)$$

Pokud budeme nyní integrovat od $s = 0$ na vzdáleném konci objemu až po $s = D$ u pozorovatele, dostáváme rovnici,

$$I(D) \exp\left(\int_0^D \tau(t)\, dt \right) - I_0 = \int_0^D g(s) \exp\left(\int_0^s \tau(t)\, dt \right) ds$$

kterou lze opět několika aritmetickými úpravami dovést do konečného tvaru:

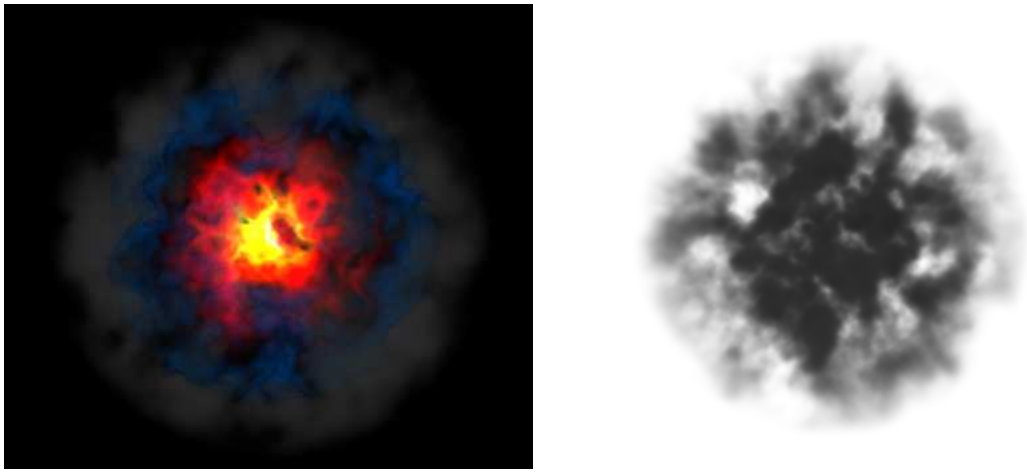
$$I(D) = I_0 \exp\left(-\int_0^D \tau(t)\, dt \right) + \int_0^D g(s) \exp\left(-\int_s^D \tau(t)\, dt \right) ds \tag{3.6}$$

Podrobné odvození, včetně vynechaných aritmetických úprav, lze nalézt například v [Max 95].

Na Obrázek 3.4 je opět ukázka použití právě odvozeného modelu šíření světla. Obrázek napravo ukazuje model aplikovaný, podobně jako výše, na jednoduchý objekt připomínající mrak. V tomto obrázku se pracuje pouze s vyzařovanou intensitou záření. Obrázek vlevo je o poznání složitejší. Je na něm zobrazena ohnivá koule, která by mohla například vznikout po výbuchu. Vnitřek

látky září žlutým a oranžovým světlem, které je dále pohlcováno vnějšími vrstvami kouře, který sám vyzařuje málo, ale má vysoký koeficient útlumu. Koeficient útlumu navíc roste směrem od středu exploze. Centrální „žhavé“ části, představující oheň, mají koeficient útlumu minimální, naopak vnější kouřové části velmi vysoký.

Na první pohled je patrné, že obrázek je příliš tmavý. To je způsobeno tím, že použití pohlcujícího a vyzařujícího modelu nebere v úvahu rozptyl záření uvnitř látky. Tento rozptyl by způsobil, že vnější kouřové vrstvy by byly daleko více osvětleny, než jsou nyní.



Obrázek 3.4: Použití pohlcujícího a vyzařujícího modelu šíření záření

Stínování a rozptyl

Dalším krokem k většímu realismu a věrohodnosti optického modelu je zahrnutí osvětlení látky z vnějších zdrojů. Doposud jsem uvažoval pouze isotropní vnitřní vyzařování látky.

Je-li látka osvětlována z externího zdroje, záření dopadá na částice látky, kde je částečně odraženo a částečně pohlceno. Modelování chování záření při odrazu, nazýváme stínování. V závislosti na druhu látky, dochází ještě k násobným odrazům uvnitř média, než je záření zcela pohlceno nebo než objem opustí. Odrážení záření uvnitř látky se nazývá rozptyl a způsob, jak jej napodobujeme model rozptylu.

V této části se zaměřím na nejjednoduší model osvětlení z vnějších zdrojů a hlavně na princip stínování. Budu uvažovat osvětlení jedním vnějším zdrojem. Dalším předpokladem bude, že záření není na své dráze nijak utlumeno, a to ani mezilehlými částicemi objemu a je odraženo právě jednou. Zajímat nás bude pouze změna intensity rozptýleného záření. V obecném případě není pouze popis intensity dostačující a je třeba sledovat několik dalších faktorů, jako například změnu polarizace či vlnové délky.

Nejdříve se věnujme stínování z fyzikálního hlediska. Pro výše uvedený zjednodušený případ lze událost odrazu záření od částice v objemu popsat následující rovnicí:

$$S(X,\omega,\omega') = r(X,\omega,\omega')\,i(X,\omega') \tag{3.7}$$

kde

- $S(X,\omega,\omega')$ je intensita záření přicházejícího ze směru ω' odraženého v bodě X do směru ω

- $r(X,\omega,\omega')$ je takzvaná BRDF (Bidirectional Reflectance Distribution Function) neboli Obousměrná funkce odrazivosti
- $i(X,\omega')$ je intensita záření, které dopadá do bodu *X* ze směru ω'

Rovnice (3.7) je platná právě pro jeden zdroj osvětlení, který leží ve směru ω'. Tento směr může být, v případě rovnoběžného zdroje osvětlení, stejný pro všechny body *X* nebo je nutné jej pro každý bod určit zvlášť. Nejdůležitějším členem rovnice (3.7) je BRDF. Tato funkce je závislá na konkrétním bodu *X*, směru ω', odkud záření přichází a směru ω, ve kterém intensitu záření sledujeme. Hodnota funkce BRDF je z intervalu $[0,1]$ a udává vztah, jaká část záření dopadajícího ve směru ω' je odražena do směru ω.

Důležitý je také předpoklad jednoduchého rozptylu. V rovnici (3.7) je totiž intensita záření odraženého do směru ω závislá pouze na intensitě záření přicházejícího ze směru ω', kde leží světelný zdroj. V případě násobného rozptylu bychom museli integrovat přes všechny směry na jednotkové kouli okolo bodu *X*. V případě více světelných zdrojů stačí vyhodnotit rovnici (3.7) pro každý z nich a výsledky sečíst.

Pro případ látky, kde pracujeme nikoliv s jednotlivými částicemi látky, ale s jejich hustotou na jednotkový objem, lze blíže specifikovat tvar BRDF:

$$r(X,\omega,\omega') = a(X)\,\tau(X)\,p(\omega,\omega') \tag{3.8}$$

kde

- $a(X)$ se nazývá albedo látky a udává, jaká část z útlumu připadá na rozptyl spíše, než na pohlcování záření. Vysoké albedo má například látka složená z kapiček vody, nízké například kouř složený převážně se sazí.
- $\tau(X)$ je opět koeficient útlumu, tak jak jej známe z předchozích modelů. Pouze není parametrizovaný parametrem na paprsku, ale bodem *X* v prostoru.
- $p(\omega,\omega')$ se nazývá fázová funkce a udává směrovost rozptylu látky, tedy kolik ze záření přicházejícího ze směru ω' bude rozptýleno do směru ω

Jak lze předpokládat, zásadním prvkem je v rovnici (3.8) fázová funkce. Její podoba se liší od látky k látce a pro přesnou simulaci reálných látek by bylo nutné její hodnoty naměřit. Jak ve fyzice, tak v počítačové grafice byla vyvinuta řada přístupů, jak aproximovat hodnotu fázové funkce pro určitou třídu látek. Jejich podrobný popis lze nalézt v literatuře a detailně zmínit byť jen několik z nich je mimo rozsah této práce. Uvedu proto pouze informativně pár nejběžněji používaných aproximací.

Látka složená z kulových částic nebo z náhodně orientovaných částic libovolného tvaru, bude vykazovat isotropický rozptyl. V takovém případě hodnota fázové funkce bude záviset pouze na úhlu *x* mezi směry ω a ω', tedy:

$$p(\omega,\omega') = \cos(x) = \omega\cdot\omega' \tag{3.9}$$

Na způsob rozptylu světla má zásadní vliv poměr velikosti částic látky k vlnové délce záření. Pro různé hodnoty tohoto poměru byly sestaveny modely, které popisují rozptyl za příslušných podmínek.

Rozptyl záření částicemi, jejichž velikost je řádově menší, než vlnová délka záření se popisuje takzvaným Rayleigh rozptylem (Rayleigh scattering). Zevrubný popis lze nalézt například v [Chandrasekhar 50].

Rozptyl v látce, jejíž částice jsou srovnatelné s vlnovou délkou světla a mají kulový tvar nebo je jejich orientace náhodná, je popsán takzvaným Mie rozptylem (Mie scattering). Teorie zabývající se tímto druhem rozptylu je relativně složitá a odvozené vztahy příliš komplikované pro přímé použití (narozdíl např. od Rayleigh rozptylu). Proto je v praxi tento druh rozptylu aproximován empirickými funkcemi, klasicky například pomocí Henyey-Greensteinovy funkce, jejíž detailní popis lze nalézt v [Henyey a Greenstein 40].

Abychom měli výčet úplný, je třeba dodat, že zvláštní anomálie nastává u rozptylu částicemi, které jsou řádově větší než vlnová délka záření. Zde, oproti očekávání, je útlum světla dvojnásobný, než je geometrický průřez částice kolmý na směr šíření vlny. Ovšem, aby bylo možno tento poznatek aplikovat, je třeba jev pozorovat ze vzdálenosti opět řádově větší, než je velikost částice. Proto se tento jev uplatňuje spíše v astronomických rozměrech a v počítačové grafice není často vůbec uvažován.

V některých aplikacích není potřeba, nebo to ani není vhodné, provádět stínování podle fyzikální principů. Jde zejména o případy, kdy chceme z objemových dat zobrazit pouze ty části, které odpovídají nějaké předem stanovené hodnotě přechodové funkce. Tyto části objemu bychom nyní chtěli zobrazovat jako pseudopovrch s odpovídajícím stínováním. Pro tyto účely se často používají, místo fyzikálně založené BRDF, empiricky odvozené modely, nejčastěji Phongův model (viz [Phong 75]). Proto, abychom mohli tyto empirické modely používat, potřebujeme typicky znát normálový vektor k myšlenému povrchu. Nejběžněji používaným způsobem, jak tento normálový vektor z objemových dat získat, je aproximovat ho pomocí metody centrální diferencí. Normálový vektor v bodě $X = \left(x_i, y_j, z_k\right)$ získáme jako

$$N\left(X\right) = \frac{\nabla f\left(X\right)}{\left|\nabla f\left(X\right)\right|} \tag{3.10}$$

kde

- $N\left(X\right)$ je normálový vektor v bodě X
- $f\left(X\right)$ je hodnota funkce vracející objemová data v bodě X
- $\nabla f\left(X\right)$ je gradient funkce f v bodě X

a vektor derivací ∇f aproximujeme diferencemi jako:

$$\nabla f\left(X\right) = \nabla f\left(x_i, y_j, z_k\right) = \begin{pmatrix} \frac{1}{2}\left(f\left(x_{i+1}, y_j, z_k\right) - f\left(x_{i-1}, y_j, z_k\right)\right), \\ \frac{1}{2}\left(f\left(x_i, y_{j+1}, z_k\right) - f\left(x_i, y_{j-1}, z_k\right)\right), \\ \frac{1}{2}\left(f\left(x_i, y_j, z_{k+1}\right) - f\left(x_i, y_j, z_{k-1}\right)\right) \end{pmatrix} \tag{3.11}$$

V sekci o vyzařujícím modelu jsem napsal, že do zdrojového členu $g(s)$ lze zahrnout, kromě vlastního vyzařování, také přírůstek intensity způsobený rozptylem. Zdrojový člen lze tedy nyní vyjádřit jako:

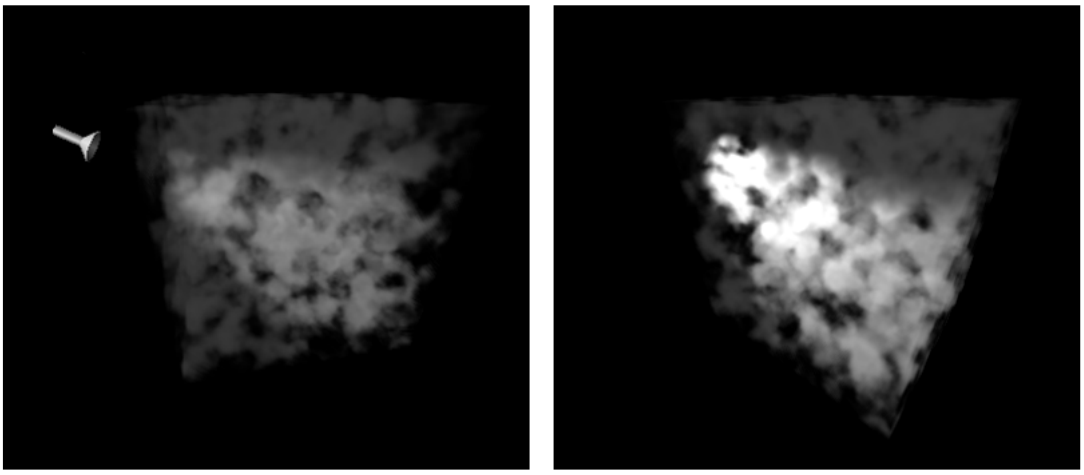
$$g(X,\omega,\omega') = E(X) + S(X,\omega,\omega') \tag{3.12}$$

kde

- $E(X)$ je vlastní vyzařování látky v bodě X

Všimněme si, že nyní již zdrojový člen není závislý pouze na pozici uvnitř objemu, ale také na směru odkud přichází světlo a na směru, kterým je světlo vyzařováno. Z toho důvodu již není zdrojový člen parametrizován parametrem na paprsku.

Pro vlastní práci s odraženým světlem tak, jak je popsáno v této sekci, stačí do rovnice (3.6) substituovat upravený zdrojový člen $g(X,\omega,\omega')$.

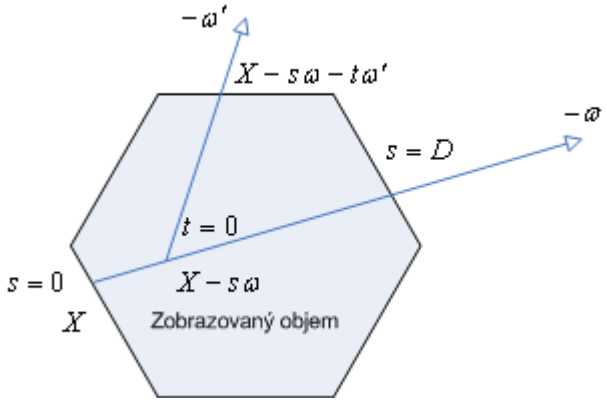


Obrázek 3.5: Ukázka směrovosti HG aproximace pro c = 0,5

Vržené stíny

Dříve než se pustíme do odvozování vztahů pro útlum záření na jeho cestě od vnějšího světelného zdroje, provedeme úpravu základní situace, ve které budeme záření sledovat.

Doposud jsme záření sledovali podél paprsku tak, jak je to zachyceno na Obrázek 3.1. Nyní je potřeba udělat dvě úpravy. Novou situaci zachycuje Obrázek 3.6.



Obrázek 3.6: Situace pro modelování vržených stínů

První změnou je, že již nebudeme parametrizovat pouze jednorozměrným parametrem na paprsku, ale přímo bodem v třírozměrném prostoru. Tento způsob parametrizace byl použit již v minulé části o stínování. Důvodem je, že v následujících rovnicích potřebujeme pracovat s body v prostoru, které neleží pouze na jednom paprsku jako doposud.

Druhou změnou je otočení smyslu parametru na paprsku. Tu provedeme zejména z důvodu názornosti a přehlednosti zápisu vztahů, které budou následovat.

Máme tedy sledovaný bod *X*, který leží na paprsku, jehož jednotkový směrový vektor je $-\omega$. Body na tomto hlavním paprsku jsou parametrizovány parametrem *s* tak, že bod ve vzdálenosti *s* od *X* ve směru $-\omega$ vyjádříme jako $X - s\omega$. Pro každý tento bod máme směr $-\omega'$, odkud přichází osvětlení. Libovolný bod na paprsku určeném směrem $-\omega'$ a bodem $X - s\omega$ lze pak vyjádřit jako $X - s\omega - t\omega'$. Nyní se můžeme pustit do odvození potřebných vztahů.

V minulé sekci jsem se zabýval chováním záření při dopadu na částici látky a způsobem, jak lze modelovat způsob jeho odrazu. Přitom jsem předpokládal, že světlo není na své cestě nijak utlumeno. Přirozeným rozšířením tohoto modelu, při zachování jednoduchého rozptylu, je uvažovat útlum záření na cestě od zdroje.

Vezmeme tedy rovnici (3.7) a upravíme člen vyjadřující intensitu záření dopadajícího ze směru ω':

$$i(X, \omega') = L \exp\left(-\int_0^{\infty} \tau(X - t\omega') dt \right) \tag{3.13}$$

kde

- *L* je intensita zdroje záření

Vidíme tedy, že intensita záření *L* je utlumena o výraz, který je nám znám již z rovnice (3.2) uvedené u úvodního pohlcujícího modelu. Jediným rozdílem je, že v rovnici (3.13) je obrácen smysl parametrizace paprsku. Tedy bod pro $t = 0$ je bod *X*, ve kterém nastala událost rozptylu. V praxi samozřejmě integrace neprobíhá až po $t = \infty$, ale pouze k hranici objemu.

Obrácený smysl parametrizace paprsku, tedy pro $t = 0$ v místě pozorovatele *X* je nyní i přes složitost zápisu výhodnější. Proto přepíšeme rovnici (3.6) s touto obrácenou parametrizací. Dostáváme tedy:

$$I(X) = I_0 \exp\left(-\int_0^{D} \tau(X - t\omega)\, dt \right) + \int_0^{D} g(X - s\omega) \exp\left(-\int_0^{s} \tau(X - t\omega)\, dt \right) ds \tag{3.14}$$

kde

- ω je směr paprsku od pozorovatele v bodě *X*.

Pro přehlednost označme:

$$\mathrm{T}(t) = \exp\left(-\int_0^{D} \tau(X - t\omega)\, dt \right), \tag{3.15}$$

$$\mathrm{T}'(t) = \exp\left(-\int_0^{s} \tau(X - t\omega)\, dt \right), \tag{3.16}$$

$$\mathrm{T}''(t,s) = \exp\left(-\int_0^{\infty} \tau\left(X - s\omega - t\omega'\right) dt\right) \tag{3.17}$$

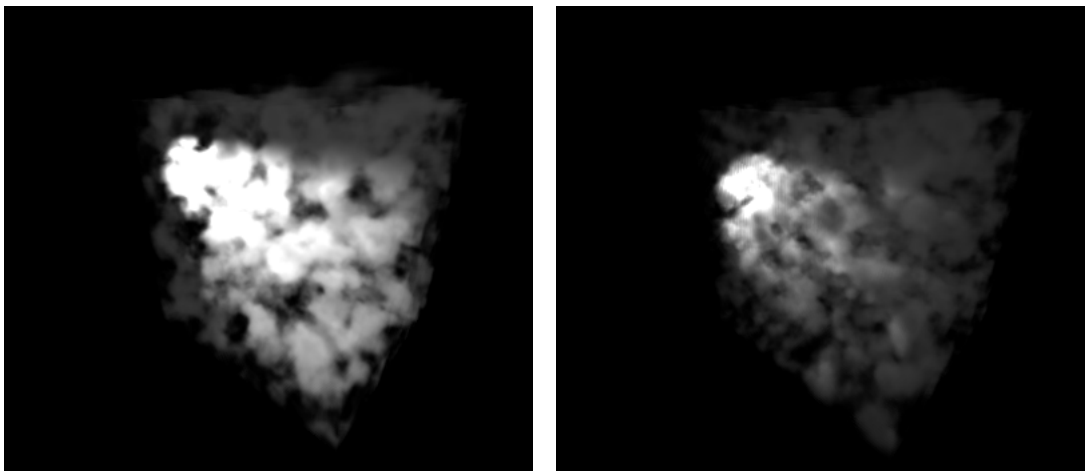
Nyní nám zbývá pouze do rovnice (3.14) substituovat výrazy (3.7), (3.12) a (3.13) a dostáváme:

$$I(X) = I_0 \mathrm{T}(t) + \int_0^{D} \left(E\left(X - s\omega\right) + r\left(X - s\omega, \omega, \omega'\right) L\mathrm{T}''(t,s)\right) \mathrm{T}'(t) ds \tag{3.18}$$

kde

- ω je opět směr paprsku od pozorovatele v bodě X
- ω' je směr paprsku od bodu $X - s\omega$ ke zdroji světla. Tento směr je v případě rovnoběžného osvětlení stejný pro všechny takové body, v případě bodového osvětlení je nutné jej určit pro každý takový bod zvlášť.

V modelu, který vyjadřuje rovnice (3.18) tedy máme zahrnuty všechny doposud zmíněné faktory. Člen $E\left(X - s\omega\right)$ vyjadřuje vlastní vyzařování látky. Osvětlení o intensitě L z externího zdroje je utlumeno jednak průchodem objemem o faktor $\exp\left(-\int_0^{\infty} \tau\left(X - s\omega - t\omega'\right) dt\right)$ a také dvousměrovou funkcí odrazivosti reprezentovanou členem $r\left(X - s\omega, \omega, \omega'\right)$. Celý tento přírůstek intensity je dále utlumen na dráze k pozorovateli o faktor $\exp\left(-\int_0^{s} \tau\left(X - t\omega\right) dt\right)$.



Obrázek 3.7: Srovnání efektu útlumu světla

Model pro násobný rozptyl

Logicky dalším krokem v rozvoji optických modelů je zahrnutí násobného rozptylu. Tento krok je ovšem daleko složitější, než byly přechody v minulých sekcích. Řešení tohoto problému je ekvivalentní například problému globálního osvětlení a výpočtu

zobrazovací rovnice (viz [Kajiya 86]). Při jednoduchém rozptylu jsme pracovali s jedním směrem světla a zdrojový člen zahrnoval vlastní vyzařování plus příspěvek z toho jednoho směru utlumený pomocí BRDF. Nyní bychom potřebovali příspěvek ze všech možných směrů. Nyní již nelze ovšem použít přímo utlumenou hodnotu intensity světelného zdroje, ale je potřeba ta samá hodnota, jako se snažíme spočítat. Člen vyjadřující rozptyl by tedy bylo nutné z tvaru (3.7) upravit na:

$$S(X,\omega,\omega') = \int_{4\pi} r(X,\omega,\omega') I(X,\omega') d\omega' \qquad (3.19)$$

kde

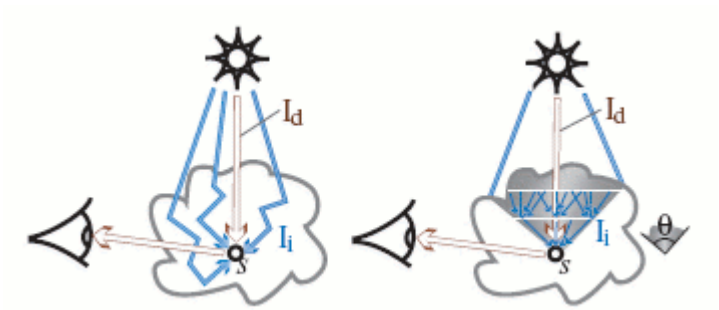
- $r(X,\omega,\omega')$ je opět BRDF pro směry ω a ω'
- $I(X,\omega')$ je intensita záření dopadajícího do bodu X ze směru ω'

Uvědomme si, že výsledek celého výpočtu je hodnota $I(X,\omega)$. Implementace tohoto modelu osvětlení je mimo možnosti této práce. Jak je vidět v §5.6, již předchozí model s jednoduchým rozptylem je velkým zatížením. Problém násobného rozptylu je ovšem ještě o řád složitější.

Empirický model podle Kniss et al.

V mnoha pracích a na mnoha ukázkách bylo prokázáno, že vliv násobného rozptylu a tedy nepřímé propagace světla v objemu je velmi důležitý pro simulaci mnoha fenoménů a i běžný uživatel je schopen zachytit jeho nepřítomnost v poloprůhledných mediích jako například mraky, kouř či vosk. Proto bylo vyvinuto spousta zjednodušení problému obecného rozptylu tak, aby tento jev mohl být alespoň částečně simulován. V této části si představíme alespoň jeden takový model, jako zástupce za řadu ostatních.

Tato zjednodušení lze rozdělit do dvou kategorií. V prvním případě se jedná o zjednodušení vlastností média či okolních vlivů tak, aby bylo možné zobrazovací rovnici analyticky spočítat. Tato omezení bývají ovšem často příliš velká. Jedná se například o požadavek homogenní hustoty média či omezení konfigurace světelných zdrojů.



Obrázek 3.8: Princip modelu podle Kniss et al. Převzato z [Kniss et al. 02a]

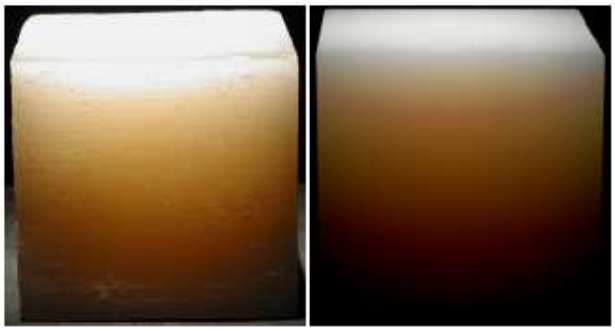
Druhou skupinou jsou empirické aproximace. Tyto aproximace záměrně vypouštějí některou z částí zobrazovací rovnice tak, aby bylo visuálně dosaženo požadovaného efektu i za cenu, že výpočet je evidentně nekorektní.

Model, který je prezentovaný v práci [Kniss et al 02a] spadá do druhé kategorie. Autoři si všímají faktu, že nejdůležitějším efektem, který násobný rozptyl

přináší, je propagace světla objemem dále, než by bylo možné jen za přímého osvětlení. Jejich snahou je tedy napodobit tento efekt.

Obrázek 3.8 ukazuje princip tohoto empirického modelu. Na obrázku vlevo vidíme situaci při reálném násobném rozptylu, kdy se záření do bodu *S* dostává ze všech směrů. Na obrázku vpravo je situace zjednodušena tak, že uvažujeme příspěvek nikoliv ze všech směrů, ale pouze ze směrů pod určitým úhlem θ. Důležité je, že vzhledem ke směru zdroje osvětlení, leží všechny potřebné body **před** bodem *S*. Tento fakt umožňuje počítat příspěvek nepřímého osvětlení po úrovních ve směru od zdroje světla (jak je na obrázku naznačeno horizontálními bílými čarami).

Na výsledcích prezentovaných autory se ukazuje, že tato přibližná aproximace dává rozumně vypadající výsledky a že hlavní cíl, tedy simulovat propagaci světla objemem, je dosažen. Na Obrázek 3.9 jsou ukázány dosažené výsledky.



Obrázek 3.9: Výsledky empirického modelu. Převzato z [Kniss et al. 02a]

Obrázek vlevo ukazuje fotografii voskové svíčky a napravo je ta samá scéna zobrazená pomocí prezentovaného empirického modelu. Vidíme, že světlo je opravdu propagováno ve směru od světelného zdroje. Díky zanedbání některých příspěvků ovšem znatelně méně. Pokud bychom ovšem neměli k dispozici fotografii, jen velmi zkušený pozorovatel by poznal rozdíl.

3.2.2 Empirické mapování

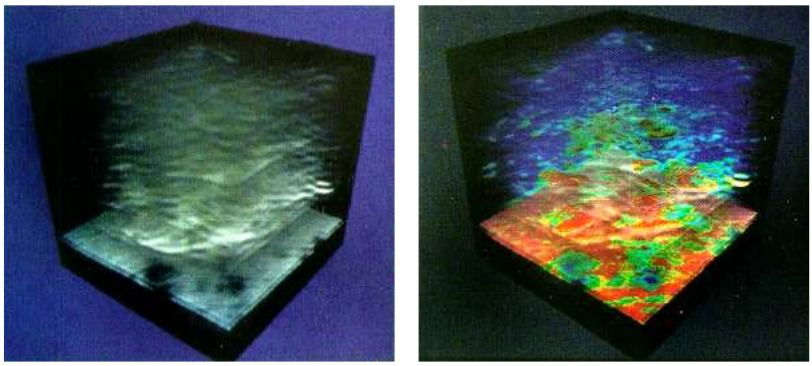
V předchozí sekci jsem podrobně prezentoval fyzikálně založené optické modely a na konci jeden empirický model, jehož účelem je ovšem simulovat reálné vlastnosti látky a empirický postup je použit pouze k zjednodušení fyzikální reality.

V oblasti vědecké vizualizace ovšem nemusí být reálně vypadající obrázek vždy cílem. Někdy to ani není možné, neboť například při zobrazování map elektronové hustoty neexistuje pozorovatelná fyzikální předloha.

Cílem ve vědecké vizualizaci je náhled na data, a proto je někdy žádoucí barvou či stínováním zdůraznit některé rysy v datech. Jiným případem, kdy je lepší sáhnout po empirickém mapování je moment, kdy potřebujeme zobrazit nějakou vlastnost dat, která by při fotorealistickém zobrazení zůstala skryta.

Tyto modely bývají často konstruovány přímo pro konkrétní množinu dat, na rozdíl od fyzikálně založených optických modelů. Nejčastěji tyto modely počítají nějakou z verzí optického modelu prezentovaného v §3.2.1. Nepoužívají ovšem tento výpočet pro všechny barevné kanály, ale s výsledkem pracují jako se skalárem, který modifikuje empiricky určené mapování do barevného prostoru. Jedním takovým modelem je model prezentovaný v práci [Sabella 88].

Model podle Sabelly
V této práci je prezentována metoda pro zobrazování objemu založená na vrhání paprsku. Podél každého z paprsků je počítána utlumovaná intensita vyzařující látky analogicky, jako v §3.2.1 v pohlcujícím a vyzařujícím modelu. Výsledek toho výpočtu je skalár určují intensitu záření podél daného paprsku. Zajímavý je způsob, jak jsou další vlastnosti objemu mapovány do barevného prostoru. V následujících výrazech používám identické značení jako v §3.2.1.



Obrázek 3.10: Výsledky mapování podle Sabelly. Převzato ze [Sabella 88]

Podél paprsku jsou sledovány následující vlastnosti:

- Maximální hodnota *M* podél paprsku: $M = \max_0^D \left(\tau(t)\right)$
- Vzdálenost *M* od bodu pro $t = D$
- Utlumená intensita *I*, která je výsledkem výpočtu pomocí optického modelu
- Těžiště *C*: $C = \frac{\int_0^D t\tau(t)\,dt}{\int_0^D \tau(t)\,dt}$

Pro práci s barvou pak není použit běžnější RGB model, ale HSV model. Mapování je dvou druhů:

1. *M* je mapováno na odstín *H*, *D* je mapováno na saturaci *S*, *I* je mapováno na světlost *V*
2. *M* je mapováno na odstín *H*, *C* je mapováno na saturaci *S*, *I* je mapováno na světlost *V*

Výsledkem toho mapování jsou tedy obrázky, kde barva určuje velikost maximální hodnoty (červená je největší). Čím blíže je tato maximální hodnota k pozorovateli, tím je barva jasnější a čím hustší je látka v daném místě, tím světlejší toto místo je. Výsledky jsou ukázány na Obrázek 3.10. Vlevo jsou seismická data zobrazená pouze pomocí intensity *I*. Vpravo ta samá data s mapováním barev podle klíče 1.

3.3 Zobrazovací postupy

V následujících podkapitolách jsou prezentovány čtyři třídy přístupů k zobrazování objemu. Z hlediska principu zobrazovacího systému tak, jak je uveden v §1.2 budu

hovořit o krocích (s3, s4), tedy klasifikaci dat a jejich promítnutí na průmětnu. Přitom se budu hojně odkazovat na §3.2 o optických modelech. Kroky (s1, s2) se nezabývám z důvodů uvedených v §1.2. Tento přístup je ve shodě s praxí autorů původních prací, kteří se typicky odkazují na stejné předpoklady. Krok (s5) je implicitně daný použitou architekturou.

Dále budu předpokládat, že existuje skalární spojitá funkce $f'(X):\mathbb{R}^3 \to \mathbb{R}$, která vrací hodnoty v libovolném bodě objemu *X*. Typicky je tato funkce implementována pomocí interpolace z diskrétních vstupních hodnot. V praxi se používají dva způsoby interpolace.

První z nich je hodnotou nejbližšího souseda, což znamená, že pro daný bod *X* nalezneme nejbližší bod mřížky a jeho hodnotu vrátíme jako hodnotu v bodě *X*. V tomto případě se na objem konceptuálně nahlíží jako na sadu voxelů, kde každý voxel má definovanou hodnotu, která je konstantní přes celý jeho objem.

Druhým konceptuálním náhledem je, dívat se na objem jako na sadu hodnot v uzlových bodech a při dotazu na hodnotu v bodě *X* ji interpolovat z hodnot uzlů v okolí tohoto bodu. Nejčastěji je používána trilineární interpolace, kdy je hodnota určena z osmi nejbližších uzlů. Lze se setkat i s interpolacemi vyšších řádů, ale pouze poměrně zřídka. Je to zejména z důvodu, že uváděná interpolace je jednou z nejčastějších operací při zobrazování objemu a tedy cena, například trikubické interpolace, kdy používáme 64 sousedních uzlových hodnot, je příliš vysoká.

Někdy budu také tuto funkci z důvodu přehlednosti psát jako $f(t):\mathbb{R} \to \mathbb{R}$, kde *t* bude značit parametr na paprsku.

3.3.1 Metody průřezů

Metoda průřezů vznikla jako snaha využít specializovaných schopností hardware k urychlení zobrazování objemu.

Nejvíce perspektivní z nich bylo hardwarově urychlované mapování textur, přesněji trilineární interpolace implementovaná při čtení hodnot z 3D textur.

Typicky je vytvořena sada polygonů, vzniklá ořezáním rovin rovnoběžných s průmětnou pomocí obálky objemu. Na tyto polygony je pak namapována 3D textura (tedy vlastní objemová data) a je ponecháno na hardware, aby provedl interpolaci z hodnot uložených v textuře.

Výhodou toho přístupu je snadnost implementace a vysoká rychlost zobrazování. Množství kódu, které musí programátor fyzicky napsat je překvapivě nízké. Často stačí pomocí standardizovaných API pouze vhodně nastavit zobrazovací řetězec a množinu rovin pro ořezávání a hardware a runtime příslušného API provede většinu zbývajících operací.

Nevýhodou je, že množství paměti pro uložení 3D textury je jen omezené a tak i na nejmodernějších konzumních grafických kartách lze zobrazovat pouze středně velké množiny dat.

Historie

Poprvé byl tento přístup použit v práci [Cullip a Neumann 93]. Využit byl přitom hardware v tehdejších stanicích firmy SGI s názvem RealityEngine. Rychlost této metody byla na tehdejší dobu skutečně revoluční. Autoři dosahují na datech velikosti 128x128x64 při výstupu do okna 512x512 rychlostí okolo 10 fps. Postup byl dále zkoumán a rozšířen na aplikace v medicíně v práci [Cabral et al. 94]. V následujících letech byly tyto techniky zdokonalovány a postupně byl přístup založený na

mapování textur přijat jako metoda přímého zobrazování objemu schopná dosáhnout interaktivních rychlostí. Byl k tomu využíván speciálně zkonstruovaný hardware v pracovních stanicích [Van Geldern a Kwansik 96, Westermann a Ertl 98, Meissner et al. 99] a později i hardware konzumních počítačů, jako například v pracích [Rezsk-Salama et al. 00, Engel et al. 01, Guthe et al. 02, Kniss et al. 02a]. V nedávné době byl tento přístup, spolu s čistě vyzařujícím modelem plynů, použit pro vizualizaci planetárních mlhovin [Magnor et al. 04]

Princip

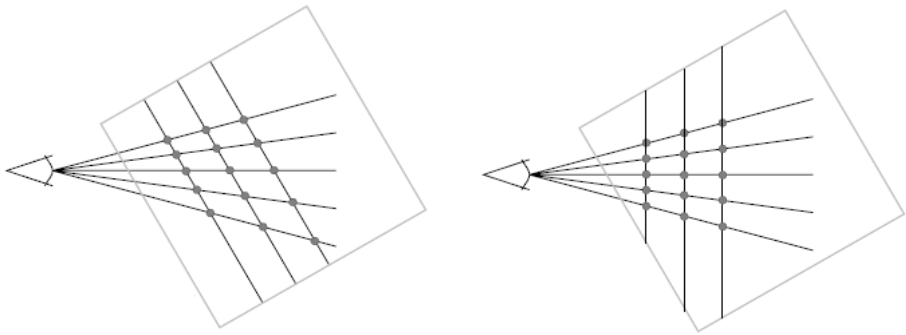
Následující výklad sleduje práci [Cullip a Neumann 93].

Abychom mohli zobrazovat objemová data, potřebujeme umět vzorkovat tato data podél paprsků, které směřují od pozorovatele skrze průmětnu do objemu. Uvážíme případ, kdy tyto vzorky leží vždy v jedné rovině. Potom můžeme tuto rovinu nahradit jedním polygonem a na něm vzorkování provést. Pokud navíc uložíme vstupní objemová data jako 3D texturu, je postup velmi přímočarý. Stačí vygenerovat sadu polygonů v příslušných pozicích uvnitř objemu a namapovat na ně tuto 3D texturu. Hardware pak již zajistí správné navzorkování.

Jsou dva možné přístupy, jak umístit vzorkovací polygony do objemu. Buď je možné je orientovat v souřadnicovém prostoru objemu nebo v souřadnicovém prostoru průmětny. Obrázek 3.11 nalevo ukazuje situaci při práci v souřadnicovém prostoru objemu, obrázek napravo při práci v souřadnicovém systému průmětny.

Při práci v souřadnicovém systému objemu je nutné nejprve určit osu, která je nejvíce rovnoběžná se směrem pohledu. To lze udělat například analýzou pohledové a modelové matice. Polygony poté vygenerujeme přímo tak, aby byly kolmé na tuto osu a ležely přesně uvnitř objemu.

Při práci v souřadnicovém systému průmětny můžeme polygony vygenerovat přímo. Je ovšem nutné, je pak převést do souřadného systému objemu a tam je, podle ohraničující obálky objemu, oříznout. Nicméně i tuto operaci může velmi rychle provádět grafický hardware.



Obrázek 3.11: Orientace vzorkovacích polygonů. Převzato z [Cullip a Neumann 93].

Poslední věcí, kterou zbývá rozhodnout, je jakým způsobem budeme hodnoty navzorkované na polygonech skládat. Lze to provést pomocí nějaké obecné operace skládání, pokud nám nejde o fyzikální věrnost. Pokud bychom chtěli simulovat například pohlcující model tak, jak je prezentován v §3.2 a generovat tak obrázky podobné negativu rentgenovým snímkům, je potřeba do kompozice zahrnout vzdálenost mezi jednotlivými vzorkovacími polygony. Tato vzdálenost je jiná pro každý paprsek, ale stejná pro všechny vzorky na paprsku. V takovém případě stačí za objem umístit dodatečný polygon, který bude mít namapovanou texturu

s akumulovanými vzdálenostmi mezi rovinami. Tuto texturu je pak potřeba přepočítávat při každé změně pozice pozorovatele.

3.3.2 Metody otisků

Historie

Ke konci devadesátých let byly vyvinuty nové přístupy, které lze souhrnně označit jako metody otisků (splatting). První takovou metodou je V-buffer prezentovaný v [Upson a Keeler 88], následovaný prací Westovera [Westover 90]. Společnou myšlenkou těchto metod je promítnout každou buňku objemu na průmětnu a pomocí rekonstrukčního, typicky Gaussovského jádra (nazývaného otisk buňky), určit její příspěvek do pixelů výsledného obrazu.

Princip

Následující postup sleduje práci [Westover 90]. Základní kroky „splatting“ algoritmu lze shrnout následujícím způsobem:

1. Zpracováváme jednu buňku objemu za druhou. Podle druhu mřížky a velikosti otisku lze určit, které buňky se nemohou ovlivňovat a zpracovávat více buněk objemu najednou.
2. Určíme střed otisku buňky na průmětně.
3. Určíme tvar (rozsah) otisku buňky na průmětně.
4. Pro každý pixel, který je otiskem pokryt určíme integrací rekonstrukčního jádra příspěvek otisku do pixelu
5. Příspěvek z předchozího kroku přičteme k již akumulované hodnotě v pixelu. Pokud tato hodnota přesáhla například maximální mez průhlednosti, lze pixel označit a v dalších buňkách jej již nezpracovávat.

Tento na první pohled jednoduchý algoritmus má řadu úskalí. Prvním z nich je určení tvaru otisku na průmětně. Jako tvar, který promítáním deformujeme, je použita koule. Koule se nám může promítnout do kruhu nebo do elipsoidu. Tvar otisku závisí na zvolené projekci a druhu mřížky, ve které jsou objemová data uložena. Pokud máme stejnou vzdálenost mezi buňkami mřížky ve všech směrech (pracujeme tedy s kartézkou mřížkou), tak při rovnoběžném promítání je otisk vždy kruh a navíc pro daný směr pohledu vždy stejný pro všechny buňky objemu. Pokud je mřížka jiná, je potřeba otisk počítat, v nejhorším případě pro každou buňku zvlášť. Obdobný nepříjemný případ nastává u perspektivního promítání, kde se koule také promítne na elipsoid a je nutné jej počítat pro každou buňku zvlášť.

Tím ovšem těžkosti nekončí. V bodu 4. je uvedeno, že příspěvek získáme integrací rekonstrukčního jádra. Typicky složitost jádra odpovídá nepřímo úměrně kvalitě zobrazení. Proto jádra často nelze integrovat analyticky nebo to ani z důvodu výpočetní složitosti není možné (uvažme, že tato integrace jádra je jednou z nejčastějších operací algoritmu). Mezi použitá jádra patří například kužel, Gaussovské jádro nebo několik prvních vln funkce sync. Proto bývají hodnoty jádra předintegrovány pro obecný tvar otisku a uloženy v tabulce. V tomto případě je nutné transformovat pohledově závislý otisk na obecný otisk tak, aby bylo možné vyzvednout správnou hodnotu z předpřipravené tabulky. Jedná se tedy o transformaci z elipsoidu na kruh.

Je také jasné, že velikost tabulky velmi ovlivňuje kvalitu zobrazení. Velikosti tabulek jsou řádku desítek na každý rozměr.

3.3.3 Faktorizace

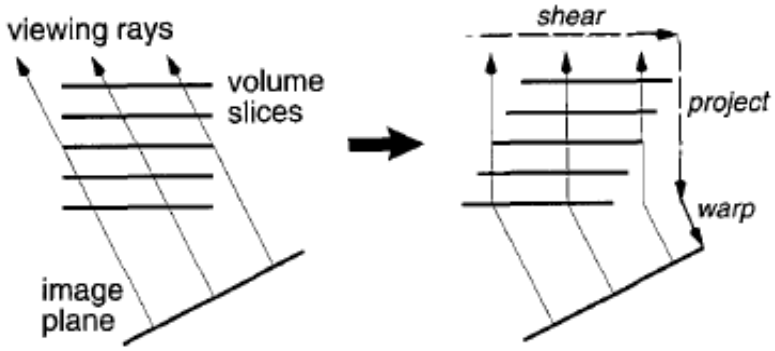
Historie metody

V pracích [Cameron a Undrill 92, Lacroute a Levoy 94] byla prezentována ještě další metoda. Již dříve byly snahy využít koherence buněk objemu či pixelů průmětny pro urychlení existujících metod. Problémy způsobovala především perspektivní projekce, dlouhá doba předzpracování objemu nutná pro vytvoření prostorově koherentních datových struktur a jejich pohledová závislost. Metoda faktorizace (shear-warp factorization) spočívá v rozdělení (faktorizaci) pohledové matice na dvě (u některých autorů i více) části. Nejprve jsou objemová data promítnuta na dočasnou průmětnu a poté je obraz na této průmětně zdeformován jednoduchou 2D transformací na skutečnou průmětnu. Princip metod spočívá v tom, že dočasná průmětna je rovnoběžná s jednou z primárních os a pixely v ní jsou zarovnány s buňkami objemu, což umožňuje jeho efektivní promítnutí. Výhodou těchto metod je využití prostorové koherence objemových dat při uchování krátké doby předzpracování a pohledové nezávislosti.

Princip

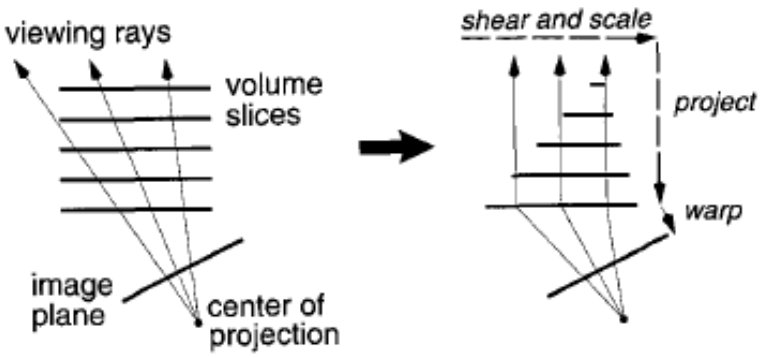
Následující výklad sleduje práci [Lacroute a Levoy 94]. Princip celé metody spočívá v transformaci objemových dat do pomocného souřadného systému, který je zvolen tak, aby se v něm objemová data snadno a rychle promítala na průmětnu. Zavedeme pomocný „zkosený“ souřadnicový systém následujícím způsobem: Zkosený souřadnicový systém je takový, v němž jsou všechny promítací paprsky (viewing rays) rovnoběžné se třetí souřadnou osou. Dále budeme předpokládat, že objemová data jsou vzorkována na rovnoběžné mřížce.

Transformaci dat do zkoseného souřadného systému provedeme tak, že určíme, který z hlavních směrů je nejvíce rovnoběžný se směrem pohledu a přeuspořádáme souřadnice tak, aby tento směr byl v pořadí třetí. Následně zkosíme objemová data v osách kolmých na tento směr a provedeme promítnutí na dočasnou průmětnu. Pokud používáme perspektivní promítání, musíme kromě zkosení objemovým datům také příslušně změnit měřítko.



Obrázek 3.12: Zkosení při rovnoběžném promítání. Převzato z [Lacroute a Levoy 94].

Výše uvedený postup pro rovnoběžné promítaní ukazuje Obrázek 3.12 a perspektivní případ ilustruje Obrázek 3.13.



Obrázek 3.13: Zkosení při perspektivním promítání. Převzato z [Lacroute a Levoy 94].

Definici zkoseného souřadného systému můžeme formalizovat jako faktorizaci pohledové matice na tři části:

$$M_{view} = P \cdot S \cdot M_{warp} \tag{3.20}$$

kde:

- M_{view} je pohledová matice
- P je permutační matice, která prohazuje souřadnice tak, jak je popsáno výše
- S je matice zkosení, případně matice zkosení a změny měřítka

Matice zkosení je velmi jednoduchá:

$$S_{paral} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ s_x & s_y & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.21}$$

kde:

- s_x je míra zkosení v první souřadné ose
- s_y je míra zkosení v druhé souřadné ose

Pro perspektivní případ není o mnoho složitější:

$$S_{persp} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ s_x & s_y & 1 & s_w \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.22}$$

kde:

- s_x je míra zkosení v první souřadné ose
- s_y je míra zkosení v druhé souřadné ose
- s_w je míra změny měřítka

Poslední člen, tedy matici M_{warp} získáme jako doplněk výše uvedeného součinu, tedy: $M_{warp} = S^{-1} \cdot P^{-1} \cdot M_{view}$.

Nyní můžeme celý algoritmus shrnout do tří kroků:

1. Objemová data jsou transformována do zkoseného souřadného systému pomocí matice *S*. Matice *P* určuje kolmo k jakému směru budeme posouvat.
2. Transformovaná a převzorkovaná data promítneme na dočasnou průmětnu, která je také ve zkoseném souřadném systému a je kolmá na hlavní směr.
3. Obraz na dočasné průmětně zdeformujeme pomocí matice M_{warp} do konečné podoby.

Výše uvedený postup prezentuje základní verzi algoritmu. Důležitým krokem, kvůli kterému je celý algoritmus navržen, je projekce na dočasnou průmětnu. Díky tomu, že máme možnost data libovolně vůči pixelům průmětny naškálovat a zarovnat, otevírá se prostor pro mnoho urychlovacích technik, které jsou v obecném případě nepoužitelné či příliš „drahé“. Jejich podrobný popis lze nalézt například v práci [Lacroute a Levoy 94].

3.3.4 Vrhání paprsku

Metody založené na vrhání paprsku se objevují velmi brzo po prvních metodách hledajících povrch.

Jsou velmi přirozeným způsobem, jak řešit rovnice uvedené v §3.2. Ve skutečnosti jsou přímým rozšířením způsobu, jak jsou tyto rovnice odvozovány. Není proto náhodou, že tyto metody se objevují spolu s prvními prácemi o použití teorie šíření záření v počítačové grafice.

Jejich velkou výhodou je schopnost generovat vysoce kvalitní výstup a velmi snadná implementace základních variant těchto algoritmů.

Nevýhodou je vysoká výpočetní náročnost, která velmi dlouho vyřazovala metody toho druhu z použití pro interaktivní aplikace.

Historie

Od počátku osmdesátých let byly zkoumány nové přístupy k zobrazování objemu. Ukazovalo se, že metody hledající povrch jsou vhodné pouze pro některé aplikace a v případě, že bychom se snažili zobrazit objem jako celek, jsou zcela nedostatečné.

Jednou z motivací za touto snahou byl určitě fakt, že mnoho přírodních fenoménů lze pohodlně definovat právě ve formě objemových dat, ale k jejich fotorealistickému zobrazení nestačí vykreslovat povrch určený jedním nebo více prahy.

Objevují se proto nové přístupy založené na technice vrhání paprsku a výpočtu chování světla uvnitř objemu podél této lineární dráhy. Pro fotorealistické zobrazení přírodních fenoménů, jako mlha či mraky, bylo potřeba realisticky modelovat přenos záření uvnitř objemu a zobrazovat jej jako poloprůhledný materiál. Poprvé byly integrální rovnice pro simulaci přenosu záření uvnitř objemu použity Blinnem [Blinn 82], který s jejich pomocí zobrazoval rovinnou atmosféru. V následujících letech byla jeho technika rozšířena na obecnější objekty [Max 83], [Voss 83], byly formulovány rovnice i pro simulaci sekundárního rozptylu [Kajiya a Von Herzen 84] a byla prezentována nutnost jejich použití v objemech s vysokou

mírou rozptylu (high-albedo volumes). Jak je vidět z chronologických údajů, vývoj těchto metod probíhal ruku v ruce s rozvojem ostatních oblastí počítačové grafiky a není náhodou, že stěžejní práce globálních osvětlovacích metod [Kajiya 86], kde je formulována integrální globální zobrazovací rovnice, byla publikována nedlouho po práci [Kajiya a Von Herzen 84], kde byly prezentovány rovnice pro sekundární rozptyl světla, které jsou založeny na totožném fyzikálním principu, tedy přenosu záření v prostředí.

V dalších letech byla zkoumána možnost zobrazovat s objemovými daty také standardní polygonálně reprezentované objekty, například [Levoy 90]. Tato snaha byla motivována jednak aplikacemi ve fotorealistickém zobrazování, kde objemová data tvořila jen menší část zobrazovaných objektů, ale také medicínskými a vědeckými aplikacemi, kde bylo potřeba spolu s objemem zobrazovat pomocné objekty, které umožňují lepší náhled na data. Jedná se například o směry léčebných radioaktivních paprsků v onkologii modelované jako komolé kužele, umělé kloubní a jiné náhrady v ortopedii či dvourozměrné mřížky umožnující lepší prostorovou orientaci a svázání objemového modelu s jeho prostředím

V souvislosti se vzrůstem výkonu běžně dostupných grafických karet byly zkoumány implementace i mimo pracovní stanice na konzumním grafickém hardware. Byla publikována implementace rekurzivního sledování paprsku [Purcell et al. 02] a o rok později i zobrazování objemu pomocí vrhání paprsku, které bylo rozšířeno o standardní urychlovací techniky, jako je předčasné ukončení zpracování paprsku (early ray-termination) a přeskakování prázdného prostoru (empty-space skipping) [Krüger 03]. Při použití obou těchto urychlovacích technik autoři dosahují, v závislosti na druhu zobrazované scény, rychlostí mezi 13.6 a 23.4 snímků za vteřinu při vykreslování do čtvercového okna o straně 512 pixelů. Nicméně v určitých případech použití těchto urychlovacích technik výpočet naopak zpomaluje. Jedná se zejména o poloprůhledné objemy vyplňující celou obálku, neboť zde se ani jedno z kritérií neuplatní.

Princip

Princip této metody je dobře známý a je podrobně rozebírán dále, zmíním se proto o něm na tomto místě jen velmi stručně.

Algoritmus vychází z faktu, že k určení barvy pixelu v průmětně je potřeba zjistit množství dopadajícího záření na plochu tohoto pixelu. Záření budeme sledovat podél paprsku, který vrháme v opačném směru než proudí záření, tedy od pozorovatele skrze průmětnu ven do scény.

V oblasti zobrazování objemových dat je určen průsečík tohoto paprsku s obálkou objemu. Část paprsku, která leží uvnitř objemu je navzorkována a příspěvky z jednotlivých vzorků jsou složeny do výsledné hodnoty, kterou tento paprsek do pixelu přispěje.

Způsobů, jak jsou navzorkované hodnoty na paprsku skládány, je velké množství a některé z nich byly představeny v §3.2.

Důležité je také poznamenat, že nemusí existovat korespondence 1:1 mezi pixely průmětny a paprsky. Je možné vrhat více paprsků jedním pixelem a stejně tak symetricky některé pixely opomíjet a hodnoty v nich dopočítávat z hodnot sousedů.

Je samozřejmé, že do výše zmíněné analogie s dopadajícím záření má tento základní algoritmus daleko. Je ovšem výhodou této metody, že ji lze snadno rozšiřovat tak, aby se maximálně blížila fyzikální realitě. Většinou ovšem za cenu zvýšení výpočetní náročnosti.

3.3.5 Jiné přístupy

Zcela odlišný a ojedinělý přístup byl prezentován v pracích [Dunne et al. 90, Levoy 92, Malzbender 93, Totsuka a Levoy 93]. Jeho význam je spíše v ukázce rozmanitosti přístupů k danému problému, než v praktické použitelnosti. Je zde použito zobrazování objemu pomocí převodu do frekvenční oblasti. Objemová data jsou nejprve převedena Rychlou Fourierovou Transformací do frekvenční oblasti. Poté je v této třídimenzionální projekci vybrán průřez, který je obsažený v rovině rovnoběžné s průmětnou a procházející počátkem. Tento průřez je pak Inverzní Fourierovou Transformací převeden zpět do prostorové oblasti a zobrazen. Velkou nevýhodou těchto metod je neschopnost zachytit zakrytí částí objemu místy, která jsou blíže k pozorovateli.

3.4 Komentář

Nyní máme přehled o přímých metodách zobrazování objemu a je tedy možné opět zhodnotit jejich vhodnost pro implementaci na GPU.

V §2.3 jsem zdůvodnil, proč objektově orientované metody nejsou vhodné pro implementaci na GPU a tak metody otisků a faktorizace z následující diskuse rovnou vypadávají.

Zůstávají dvě metody a to metoda průřezů a metoda vrhání paprsku. Je na první pohled zřejmé, že tyto dvě metody jsou velmi podobné. Metoda průřezů je analogická metodě vrhání paprsků. Pouze vzorkování neprovádí po jednotlivých paprscích, ale po rovinách.

Jako základ pro implementovanou metodu jsem zvolil vrhání paprsku z následujících tří důvodů:

- Implementace této metody pomocí grafického hardware je daleko méně prozkoumaná. Většina prací o hardwarově urychlovaném zobrazování objemu se donedávna věnovala právě metodám průřezů. Proto je tato oblast dobře prozkoumána a metoda sama nenabízí příliš mnoho možností k softwarové optimalizaci. Je to právě proto, že její hlavní výhoda je v masovém provádění jednoduchých operací přímo pomocí hardware.
- Metoda má větší potenciál, co se týče rozšiřitelnosti. Pokročilé efekty jako například lom světla jsou pomocí vrhání paprsku, přesněji jeho rozšíření rekurzivního sledování paprsku, lépe implementovatelné
- Metoda průřezů je evidentně limitovaná rasterizační rychlostí grafického hardware. Naproti tomu metoda vrhání paprsku nabízí určité možnosti, jak část zátěže přenést do stadia zpracování vrcholů. (viz §5.4.1).

Kapitola 4

Návrh metody pro implementaci na GPU

Na základě poznatků shrnutých do předchozích kapitol této práce, jsem navrhl a implementoval metodu pro zobrazování objemu na GPU. Při návrhu jsem se řídil cíli uvedenými v §1.5.

Na úvod v §4.1 představím princip implementované metody a uvedu ji do kontextu s klasifikacemi a strukturou zobrazovacího systému uvedenou v kapitole 1. Následuje detailní popis zobrazovacího algoritmu a jeho variant, které byly vyzkoušeny (§4.2). Poté jsou v §4.3 shrnuty použité optické modely a jejich vztah k modelům odvozeným v §3.2. Kapitolu uzavírá §4.4 o přípravě vývojových a testovacích dat, včetně dat použitých k vytvoření ilustrací v této práci.

4.1 Princip řešení

Navržená metoda je obrazově orientovaná metoda pro přímé zobrazování objemu určená pro implementaci na platformě programovatelného grafického hardware, který podporuje Shader Model 3.0 [DirectX9.0c 05].

Pořízení dat (s1) a jejich úprava do podoby vhodné pro zobrazování (s2)
Metoda předpokládá, že tyto kroky byly provedeny v předstihu a jejich návrhem se nezabývá. Předpokládám, že výsledkem těchto kroků a tedy vstupem do navrhované metody je třídimenzionální vektorová kartézká mřížka dat. V každém bodě této mřížky jsou uloženy čtyři hodnoty *x*, *y*, *z*, *w*.

Poslední hodnota *w* je z rozsahu $[0,1]$ a představuje naměřenou nebo spočítanou hodnotu, kterou se metoda snaží vizualizovat. Jak je tato hodnota mapována na barvu a průhlednos, je závislé na použitém optickém modelu a je to uvedeno u každého z použitých modelů v §4.3. V dalším textu budu $f(s)$ značit funkci, která pro libovolný parametr *s* na nějakém paprsku vrací hodnotu *w*. Tato hodnota je trilineárně interpolována z hodnot vstupní mřížky.

První tři hodnoty *x*, *y*, *z* jsou používány jen ve speciálních případech a představují gradient aproximovaný pomocí metody centrálních diferencí, viz §3.2. Každá z těchto tří hodnot leží v intervalu $[-2, 2]$.

Klasifikace dat (s3)
Klasifikační funkce se liší podle implementovaného optického modelu a obsahuje vždy sadu parametrů, které je možné v reálném čase nastavovat a měnit tak do určité míry způsob interpretace vstupních dat. Podrobně jsou tyto úpravy rozebrány v §4.3

Promítnutí dat na průmětnu (s4)
Vlastní promítání na průmětnu je prováděno pomocí vrhání paprsku. Metoda modifikuje klasický obrazově orientovaný přístup v tom smyslu, že paprsky nejsou vrhány pro každý pixel průmětny, ale jenom pro pixely, do kterých zasahuje průmět obálky objemu (viz následující podkapitola).

U složitějších optických modelů je prováděno stínování a výpočet vržených stínů. Oba tyto kroky jsou implementovány přímo při procesu vzorkování paprsku a jsou opakovány při každém vzorkování vstupních dat. To umožňuje metodě interaktivně reagovat, jak na změnu pozice pozorovatele, tak na změnu osvětlovacích podmínek. Metoda neklade omezení na počet a konfiguraci světelných zdrojů. Lze tedy používat světelné zdroje v libovolné pozici, včetně pozice uvnitř zobrazovaného objemu. Lze používat i libovolné bodové světelné zdroje. Pozice pozorovatele je omezena na prostor vně obálky objemu. Jakmile se pozorovatel ocitne uvnitř, přestane metoda objemová data zcela zobrazovat.

Zobrazení pixelů průmětny (s5)
Hodnoty spočítané pro dané pixely průmětny jsou následně automaticky zapisovány do obrazové paměti, odkud jsou přímo zobrazovány. Metoda je schopná pracovat ve dvou odlišných režimech.

První režim je pro samostatné zobrazování objemových dat. V tomto režimu je na začátku zpracování každého snímku barevný buffer vyčištěn a paměť hloubky vypnuta. Hodnoty získané pomocí vrhání paprsků jsou pak přímo zapisovány do barevného bufferu.

V druhém režimu metoda nic nepředpokládá o obsahu barevného bufferu a paměť hloubky je zapnuta. Získané hodnoty jsou pak do barevného bufferu míchány pomocí alfa kanálu. Tak je možné spolu s objemovými daty zobrazovat také polygonální geometrii nebo i jiná objemová data. Omezením je, že tato data se nesmí protínat s obálkou objemu. Při tomto režimu jsou hodnoty již uložené v barevném bufferu správně objemem utlumeny. Příklad takové kombinace je na Obrázek 3.2 vlevo.

4.2 Zobrazovací algoritmus

V této podkapitole se budu detailně zabývat popisem způsobu promítání dat na průmětnu. Zaměřím se na způsob vrhání paprsku a kompozice navzorkovaných dat podél tohoto paprsku.

Nejprve je třeba pro objemová data definovat takzvanou proxy geometrii. Proxy geometrie je polygonální objekt reprezentovaný trojúhelníkovou sítí a splňuje následující kritéria:

Obálka
Proxy geometrie musí být obálkou zobrazovaného objemu. Jakákoliv část objemu, která se nachází mimo proxy geometrii nebude zobrazena.

Konvexita
Proxy geometrie musí být konvexní těleso. Pokud by tomu tak nebylo, nebude metoda poskytovat korektní výsledky. Při použití generování paprsků pomocí dvou průchodů (viz níže) je nutný předpoklad, že na libovolném paprsku leží právě jeden fragment přivrácených ploch a právě jeden fragment odvrácených ploch proxy geometrie. Při použití geometrického přístupu by bylo zřejmě možné metodu modifikovat tak, aby připouštěla, že průnik paprsku s proxy geometrií je tvořen více úseky. Tato možnost ovšem nebyla zkoumána, zejména z důvodu, že omezení na konvexní proxy geometrii nepřináší žádné výrazné problémy.

V implementaci byl použit jako proxy geometrie osově orientovaný kvádr. Transformace proxy geometrie v zobrazované scéně také určuje transformaci zobrazovaného objemu. Objemová data jsou vždy transformována tak, že vyplňují právě vnitřek proxy geometrie.

Zobrazování objemu zahájíme posláním dat proxy geometrie do zobrazovacího řetězce. Proxy geometrie je promítnuta na průmětnu a rasterizována na jednotlivé fragmenty. Pro každý takový fragment vygenerujeme paprsek od pozorovatele a určíme jeho průsečíky s proxy geometrií objemu. Vznikne tedy orientovaná úsečka, kterou v pravidelných krocích navzorkujeme a určíme v těchto bodech hodnotu klasifikační funkce f. Takto získané hodnoty převedeme pomocí nějaké klasifikační funkce na smysluplné hodnoty, které poté průchodem zepředu-dozadu složíme. Jakým způsobem se takto získaná hodnota vztahuje k výsledné barvě fragmentu je odvislé od použitého optického modelu. Tento fragment je pak odeslán k dalšímu zpracování v zobrazovacím řetězci, jak bylo popsáno výše.

Následující dvě podkapitoly se věnují způsobům generování paprsku, jeho vzorkování a kompozici získaných dat.

4.2.1 Generování paprsku

Nyní jsme v algoritmu v situaci, kdy máme k dispozici prostorové souřadnice bodu *F* na povrchu přivrácené plochy proxy geometrie, který přispěje do právě zpracovávaného fragmentu. Nejprve potřebujeme paprsek, který je určen tímto bodem a pozicí kamery. Za druhé potřebujeme určit dva parametry na tomto paprsku, které určují jeho průnik s proxy geometrií.

Paprsek hledáme v parametrickém tvaru určený jedním jeho libovolným bodem a směrových vektorem. Paprsek je pak množina bodů, pro které platí:

$$P = S + t \cdot \vec{v} \tag{4.1}$$

kde:

- P je bod paprsku
- S je jeho počáteční bod
- t je parametr paprsku

- $\vec{v}$ je směrový vektor.

Potřebujeme tedy nalézt čtyři hodnoty: Souřadnice počátečního bodu *S*, směrový vektor $\vec{v}$ a dvě hodnoty t_0 a t_1 parametru paprsku *t*, které ohraničují množinu bodů paprsku ležících uvnitř proxy geometrie.

Následující dvě podkapitoly prezentují použité metody pro výpočet těchto hodnot.

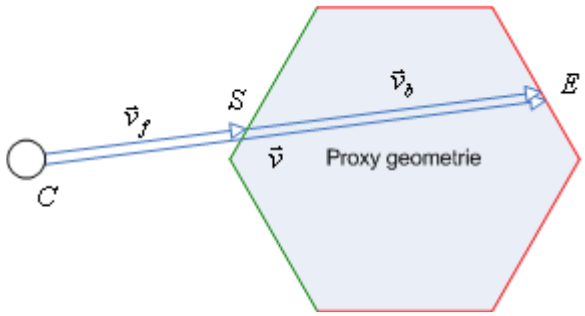
Použití dvou průchodů

Prvním přístupem, který jsem implementoval byl výpočet pomocí dvou průchodů přes proxy geometrii.

Nejdříve se v prvním průchodu uloží pomocné informace. Tyto informace se v druhém průchodu použijí k určení směrového vektoru paprsku a k určení parametrů t_0 a t_1.

Pro první průchod nastavíme orientaci vykreslovaných primitiv tak, aby se vykreslovaly odvrácené plochy. Pro každý fragment těchto odvrácených ploch se tedy provede kód připraveného fragment shaderu. V něm zaznamenáme souřadnice vektoru $\vec{v}_b$ určeného souřadnicemi bodu *F* a pozicí kamery *C*.

Pro druhý průchod otočíme nastavení orientace primitiv tak, aby se tentokrát vykreslovaly přivrácené plochy. Obdobně jako vektor $\vec{v}_b$, spočítáme vektor $\vec{v}_f$ daný opět jako rozdíl souřadnic bodu *F* a pozice kamery *C*. (zde dochází k kolizi ve značení, ovšem z důvodu přehlednosti tento nedostatek neopravuji. V obou průchodech značím bod na povrchu proxy geometrie, který reprezentuje aktuálně zpracovávaný fragment, *F*. Ovšem pokaždé se jedná o bod s jinými souřadnicemi)



Obrázek 4.1: Generování paprsku ve dvou průchodech.

Z uvedených dvou vektorů jsme již schopni určit všechny požadované informace. Jako počáteční bod *S* paprsku použijeme souřadnice bodu *F*. Směrový vektor $\vec{v}$ paprsku spočteme jako rozdíl vektorů $\vec{v}_b$ a $\vec{v}_f$ (Obrázek 4.1) Zbývá nám určit hodnoty parametrů t_0 a t_1. Vzhledem ke způsobu výpočtu vektoru $\vec{v}$ a určení počátečního bodu *S* víme, že hledané hodnoty jsou $t_0 = 0$ a $t_1 = 1$, neboť:

$$S + 0 \cdot \vec{v} = S$$

a bod *S* je shodný s bodem *F* a je tedy zároveň prvním bodem paprsku $P = S + t \cdot \vec{v}$, který protíná proxy geometrii. Zároveň víme, že poslední bod *E* paprsku, který leží uvnitř proxy geometrie, lze vyjádřit jako

$$E = C + \vec{v}_b$$

a zároveň platí:

$$S + 1 \cdot \vec{v} = S + \left(\vec{v}_b - \vec{v}_f\right) = S - \vec{v}_f + \vec{v}_b = C + \vec{v}_b = E$$

a tedy $t_1 = 1$.

Geometrický přístup

Geometrický přístup byl navržen a implementován podle [Green 05]. Základní myšlenkou toho přístupu je fakt, že v naprosté většině případů postačují jako proxy geometrie jednoduchá primitiva, například koule či kvádr. Výpočet průsečíků s těmito tělesy je dobře prozkoumán a jsou pro něj vyvinuty efektivní metody. Navíc lze výpočet ještě urychlit pomocí vektorových výpočtů na GPU.

Geometrický přístup spotřebuje na vygenerování počátečního bodu *S*, vektoru $\vec{v}$ a dvou parametrů t_0 a t_1 pouze jeden průchod. V tomto průchodu je orientace primitiv nastavena tak, jak je běžné, tedy na zobrazování přivrácených ploch. Pro každý fragment těchto ploch je opět spuštěn fragment shader. V tomto shaderu nejprve určíme paprsek procházející pozicí kamery a bodem *F* (který obdobně jako v minulém odstavci označuje bod na povrchu proxy geometrie, který přispěje do právě zpracovávaného fragmentu) a poté nalezneme analyticky hodnoty parametrů t_0 a t_1, které určují vstupní a výstupní bod paprsku vůči proxy geometrii.

Za počáteční bod $S = \left(S_0, S_1, S_2\right)$ paprsku P použijeme pozici kamery C. Opět předpokládáme, že pozice kamery je uvedena v souřadném systému proxy geometrie. Směrový vektor $\vec{v} = \left(\vec{v}_0, \vec{v}_1, \vec{v}_2\right)$ získáme jako rozdíl pozice fragmentu a pozice kamery.

Jako obalové těleso byl v metodě použit osově orientovaný kvádr. Neboť pracujeme v lokálních souřadnicích proxy geometrie, nemusí brát ohled na její posunutí a otočení. Takovýto kvádr lze určit pomocí tří skalárních hodnot délek jednotlivých stran. Proto pokud je použit geometrický přístup, metoda musí na vstupu obdržet vektor hodnot $d = \left(d_0, d_1, d_2\right)$ určující po řadě délky stran v osách *x*, *y* a *z*.

Dolním levým rohem kvádru je tedy bod $K_{bl} = \left(-d_0/2, -d_1/2, -d_2/2\right)$ a horním pravým rohem je bod $K_{tr} = \left(d_0/2, d_1/2, d_2/2\right)$.

Osově orientovaný kvádr se skládá ze tří dvojic rovnoběžných rovin. Nejprve vypočteme průsečík paprsku *P* s každou z těchto dvojic podle následujícího postupu:

$$t_0 = \frac{\frac{d_x}{2} - S_x}{\vec{v}_x}, t_1 = t_0 + \frac{d_x}{\vec{v}_x} \qquad (4.2)$$

kde:

- t_0 je jeden hraniční parametr na paprsku
- t_1 je druhý hraniční parametr na paprsku
- d_x zastupuje hodnotu rozměru kvádru pro každou z os
- S_x zastupuje souřadnici počátečního bodu paprsku pro každou z os
- $\vec{v}_x$ zastupuje souřadnici směrového vektoru paprsku pro každou z os

Nyní máme šest hodnot parametrů na paprsku, které určují průsečíky se šesti rovinami určujícími osově orientovaný kvádr. Z výše uvedeného výpočtu ovšem nemáme zaručeno, že $t_0 \leq t_1$, jak bychom očekávali. Proto dalším krokem je

porovnání parametrů odpovídajících průsečíkům dvou rovnoběžných rovin a jejich případné prohození tak, aby vždy $t_0 \leq t_1$

Posledním krokem je určení správné dvojice parametrů na paprsku, která určuje skutečné průsečíky s osově orientovaným kvádrem. Díky způsobu, jakým generujeme paprsky se nemůže stát, že by paprsek kvádr zcela minul. Je to proto, že paprsek je vygenerován z pozice bodu F a tedy již tato hodnota určuje první průsečík, který odpovídá parametru t_0. Proto je hledání správných dvou parametrů zjednodušené a stačí pouze najít nejvnitřnější dvojici, tedy největší z parametrů t_0 a nejmenší z parametrů t_1.

Srovnání obou přístupů

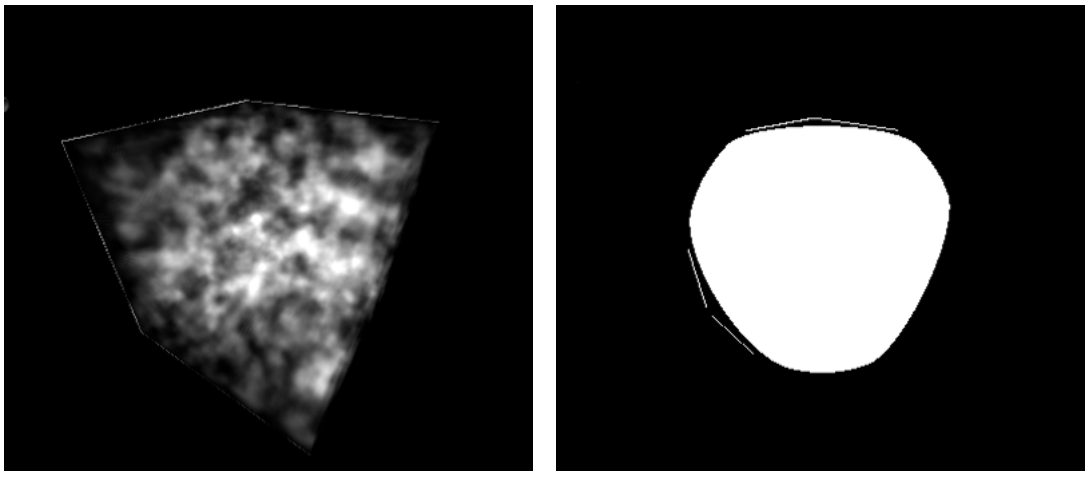
Nyní se nabízí otázka, který z přístupů je lepší a bylo by výhodnější jej použít ve výsledném programu. Jako obvykle, není odpověd na tuto otázku jednoznačná a zaleží na kritériích, podle kterých oba přístupy posuzujeme.

Efektivita

Měření hodnot fps ukazuje, že za identických nastavení je shader používající geometrický přístup přibližně o dvě třetiny rychlejší než s pomocí dvou průchodů. To je způsobeno nižším počtem průchodů u geometrického přístupu a také efektivitou výpočtu parametrů na paprsku, u kterých se ukázalo, že je lze velmi pěkně na GPU implementovat s pomocí vektorových operací.

Kvalita zobrazení

Metoda dvou průchodů trpí nežádoucími artefakty na obrysových hranách proxy geometrie. Je to způsobeno tím, že v těchto místech jsou paprsky v metodě dvou průchodů nekorektně generovány.



Obrázek 4.2: Artefakty metody dvou průchodů

Přesněji se při implementaci ukázalo, že v textuře, kde jsou uloženy vektory $\vec{v}_b$, není hodnota vůbec zaznamenána, což značí, že příslušný fragment nebyl pro odvrácenou plochu vůbec vygenerován. Je tedy načteno pozadí textury a vektor tak získá nekorektní směr. Přesný důvod, proč se tak děje, zatím není znám.

Obecnost

Velkou výhodou metody dvou průchodů je skutečnost, že jako proxy objekt je schopná zpracovat libovolnou geometrii. Naproti tomu geometrický přístup je

omezen na explicitně zadaná jednoduchá tělesa. Na první pohled by se tato skutečnost mohla jevit jako velké omezení. Ovšem v praktickém použití je její vliv zanedbatelný. To proto, že většinu běžných dat lze bez velkých neefektivit (jako např. příliš mnoho volného prostoru mezi proxy geometrií a zpracovávanými daty) obalit kvádrem. Stejně tak většina procedurálně definovaných objemů pracuje s definičním oborem $[0,1]^3$, který je pro tyto účely vhodný.

Závěrem lze tedy říci, že kromě speciálních případů, kdy potřebujeme jedinečnou proxy geometrii, vychází daleko lépe generovaní paprsků pomocí čistě geometrického přístupu.

4.2.2 Vzorkování paprsku a kompozice

Na paprsek vržený od pozorovatele lze koncepčně nahlížet jako na směr šíření světla, podél kterého se snažím aproximovat matematické vztahy uvedené v §3.2. Základním bodem je správná aproximace integrálu tvaru:

$$\int_a^b h(t)\,dt \tag{4.3}$$

kde

- $h(t)$ je nějaká funkce a *t* je parametr na paprsku od pozorovatele.

Interval $[a,b]$ rozdělme na n stejných úseků délky Δs. I-tý úsek je ohraničen hodnotami parametrů x_i a x_{i+1}, přičemž $x_i = a$ a $x_{n+1} = b$.
Potom lze vztah (4.3) aproximovat Riemannovskou sumou jako:

$$\sum_{i=1}^{n} h(x_i)\,\Delta s \tag{4.4}$$

neboť hodnota Δs je stejná pro všechny intervaly, lze ji vytknout před sumu jako

$$\Delta s \sum_{i=1}^{n} h(x_i) \tag{4.5}$$

To nám dává základ, jak vzorkovat objemová data podél dráhy paprsku. Z fáze generování paprsku, popsané v §4.2.1, máme dvě hodnoty parametru t_0 a t_1, které ohraničují množinu bodů ležících uvnitř proxy geometrie. Položíme $x_1 = t_0$ a $x_{n+1} = t_1$, n-krát určíme požadované hodnoty v bodech $S + x_i\vec{v}$ a složíme je podle vzorce (4.5).

Hodnoty, které budeme skládat, se liší podle použitého optického modelu. Někdy se jedná pouze o klasifikační funkci pro útlum, jindy o sadu hodnot určujících útlum, vyzařování a přírůstek z externích zdrojů světla.

Otázkou zcela na místě je, jak určit vzdálenost vzorků v objemu. Předpokládejme, že chceme, aby na každém paprsku, který objemem projde bylo umístěno maximálně *n* vzorků. Jak bylo řečeno výše, jako proxy geometrie byl

v implementaci použit osově orientovaný kvádr o stranách (a,b,c) se středem v počátku. Nejdelší paprsek, který může skrze tento kvádr projít je ve směru jeho tělesové úhlopříčky, jejíž velikost je $d=\sqrt{a^2+b^2+c^2}$. Pak pro vzdálenost vzorků dostáváme vztah

$$\Delta s=\frac{d}{n} \tag{4.6}$$

4.3 Použité optické modely

V experimentální implementaci jsem použil čtyři z optických modelů popsaných v §3.2. Jedná se o:

- Pohlcující model
- Vyzařující model
- Vyzařující a pohlcující model
- Rozptylující model a vržené stíny

Všechny tyto modely jsou fyzikálně založené pro zcela korektní simulaci látek, pomocí nich by bylo potřeba znát materiálové konstanty, jako například koeficient útlumu či hodnotu fázové funkce. Nicméně cílem implementované metody je vytvořit věrně vypadající obrázky. Proto jsem do vzorců přidal, podobně jako například v práci [Sabella 88], sadu interaktivně nastavitelných konstant, které ovlivňují hodnoty neznámých fyzikálních veličin.

Pohlcující model
V pohlcujícím modelu je zobrazován pouze útlum způsobený částicemi látky. Do modelu byly přidány dvě konstanty podle [Sabella 88]. Konstanta γ řídí způsob mapování hodnot funkce *f* na hustotu částic. Konstanta κ řídí velikost koeficientu útlumu. Původní rovnice (3.2), jejíž tvar je

$$I(s)=I_0\cdot\exp\left(-\int_0^s\tau(t)\,dt\right)$$

má po přidání těchto konstant tvar:

$$I(s)=I_0\cdot\exp\left(-\kappa\int_0^s f^{\gamma}(t)\,dt\right) \tag{4.7}$$

Vyzařující model
Ve vyzařujícím modelu je naopak modelováno pouze vnitřní vyzařování látky. Toto vnitřní vyzařování je kombinováno se zářením přicházejícím zpoza objemu, tedy konkrétně s barvou již uloženou v barevném bufferu. Kombinace je provedena pouhým sečtením, neboť žádný útlum není modelován.

Do modelu tak, jak je popsán rovnicí (3.4), je přidán člen γ, který opět řídí mapování hodnot funkce *f* na hustotu částic. Po přidání této konstanty má tedy vztah (3.4) tvar:

$$I(s) = I_0 + \int_0^s C(s)\, f^{\gamma}(t)\, dt \tag{4.8}$$

kde

- $I(s)$ je výsledná intensita
- $C(s)$ je míra vyzařování částice podle vztahu (3.3)
- I_0 je intensita pozadí

Hodnota $C(s)$ nemusí být konstantní pro celou látku, a je proto také v určitém smyslu parametrem.

Vyzařující a pohlcující model
Tento model je pouze kombinací předchozích dvou modelů, a proto jsou použity identické konstanty. Původní vztah (3.6) má po přidání konstant tvar:

$$I(D) = I_0 \exp\left(-\kappa \int_0^D f^{\gamma}(t)\, dt\right) + \int_0^D C(s)\, f^{\gamma}(s) \exp\left(-\kappa \int_0^s f^{\gamma}(t)\, dt\right) ds \tag{4.9}$$

Rozptylující model a vržené stíny
Tento model je nejkomplikovanější z implementovaných modelů. Modeluje nejen vyzařování a pohlcování látky, ale také její osvětlení z vnějších zdrojů světla. Do vztahu (3.18) byla přidána řada konstant, z nichž některé jsou shodné z konstantami předchozích modelů. Členy (3.15), (3.16), (3.17) reprezentující útlum jsou upraveny stejně jako v případě předchozích modelů, tedy konstantami κ a γ:

$$\mathrm{T}(t) = \exp\left(-\kappa \int_0^D f^{\gamma}(X - t\omega)\, dt\right), \tag{4.10}$$

$$\mathrm{T}'(t) = \exp\left(-\kappa \int_0^s f^{\gamma}(X - t\omega)\, dt\right), \tag{4.11}$$

$$\mathrm{T}''(t,s) = \exp\left(-\kappa \int_0^{\infty} f^{\gamma}(X - s\omega - t\omega')\, dt\right) \tag{4.12}$$

Úprava zdrojového členu (3.12) je o trochu složitější. Člen $E(X)$ reprezentující vlastní vyzařování látky je upraven stejně jako u vyzařujícího modelu, tedy:

$$E(X) = C(X)\, f^{\gamma}(X) \tag{4.13}$$

Zcela novým prvkem je BRDF. V experimentální implementaci je BRDF rozvinuta podle vztahu (3.8) a jako fázová funkce je použita Henyey-Greensteinova aproximace. Hodnota albeda $a(X)$ je nastavitelný parametr, který je ovšem stejný

pro celou látku. Stejně tak lze nastavit konstantu *c* určující směrovost rozptylu. Výsledný tvar BRDF je tedy:

$$r\left(X - s\omega, \omega, \omega'\right) = af^{\gamma}\left(X - s\omega\right)\frac{1}{4\pi}\frac{1-c^2}{\left(1+c^2-2c\left(\omega\cdot\omega'\right)\right)^{3/2}} \qquad (4.14)$$

4.4 Příprava experimentálních dat

V § 4.1 je přesně specifikováno, jak mají vypadat vstupní data pro implementovanou metodu. Abych mohl metodu otestovat, bylo nutné taková data připravit.

Pro testování jsem připravil dvě syntetické množiny dat. Každou z nich jak v mřížce 64x64x64 bodů, tak v mřížce 128x128x128 bodů. Data byla vytvořena jako turbulence upraveného Perlinova šumu. Síla turbulence a její přírůstek (gain) je libovolně nastavitelný, základní hodnoty jsou čtyři oktávy a přírůstek 2. První množina, uložená v souborech s názvy *BoxTurb.dds* a *BoxTurb128.dds* je přesně turbulence, jak je popsána výše. Druhá, uložená v souborech s názvy *SphereTurb.dds* a *SphereTurb128.dds* je stejná turbulence, pouze omezená na kouli vepsanou původní mřížce. Směrem od centra této koule je také snižována „hustota“ množiny tak, aby se „kulový mrak“, který představuje, pozvolna rozplýval.

Perlinův šum byl použit v moderní variantě publikované v práci [Perlin 02]. Oproti původnímu šumu, který byl publikován v práci [Perlin 84], tato varianta odstraňuje některé nedostatky původní implementace, zejména občasné nespojitosti ve výsledných hodnotách. Je také o něco málo výpočetně efektivnější. Perlinův šum obecně generuje hodnoty v rozsahu $[-1,1]$. Ve vygenerovaných datech je tento rozsah oříznut na $[0,1]$, což způsobuje větší nespojitosti v generovaných hodnotách a spolu s aplikovanou turbulencí dodává výsledné množině více „rozvlněný“ dojem.

Hodnoty gradientu tak, jak jsou požadovány v §4.1, jsou vygenerovány pomocí metody centrálních diferencí.

Kapitola 5

Implementace na GPU

V minulé kapitole jsem uvedl navrženou metodu z teoretického hlediska. V této kapitole se jí budu věnovat po implementační stránce.

Nejprve detailně proberu model práce s programovatelným řetězcem (§5.1). Uvedu princip použitých technologií, způsob jak se s nimi pracuje a jak je s jejich pomocí kód navržen.

Dále představím použité vývojové prostředí a budu se zabývat jeho přednostmi a nevýhodami (§5.2). §5.4 obsahuje popis programátorsky zajímavých míst kódu nebo jeho částí, kde jsem při implementaci narazil na vážnější problémy. Následuje §5.5, který představuje způsob přípravy experimentálních dat, jejich uložení na grafickou kartu a poukazuje na některé zvláštnosti v práci s nimi. Kapitolu uzavírají §5.6 a §5.7. První z nich obsahuje popis testování implementace, naměřené výsledky a jejich diskusi. Druhý se zabývá zhodnocením implementace jako celku, zejména vzhledem k cílů vytyčeným v §1.5.

5.1 Model práce a struktura kódu

Jak bylo nastíněno v úvodní kapitole, v části o grafickém hardware, práce s programy pro programovatelný řetězec zahrnuje daleko více operací, než jen napsání kódu vertex a fragment shaderů.

Při zvažování způsobu implementace jsem vycházel ze zadaných cílů, hlavně z (c2), (c3) a (c4). Při návrhu metody se ukázalo, že kromě vlastního kódu uvnitř shaderů bude potřeba provádět zejména následující operace:

- Nastavovat způsob konečného míchání barev (color blending) tak, aby výsledná barva byla správnou kombinací barvy pozadí a barvy fragmentu.
- Provádět více průchodů přes geometrii s tím, že tyto průchody si musejí předávat data skrze pomocné textury (off-screen buffers).
- Čistit barevné a pomocné buffery mezi průchody.
- Předávat hodnoty z aplikace do relativně velkého počtu parametrů pro shadery.

Tyto operace není možné provádět uvnitř vertex a fragment shaderů. Spadají totiž do oblasti nastavování neprogramovatelné části zobrazovacího řetězce, a to se typicky

provádí voláním funkcí běhového prostředí. Stejně tak předávání hodnot parametrům shaderů je aplikačně závislou činností.

Řešením se ukázalo být použití takzvaných Efektů spolu s DXSAS (DirectX Standard Annotations and Semantics), které jsou oba součástí specifikace Microsoft DirectX (viz [DirectX9.0c 05]).

Efekty jsou systém, který umožňuje jednotným způsobem a na jednom místě definovat nejenom vertex a fragment shadery, ale také potřebný stav zobrazovacího řetězce a pomocí metadat přiřazovat význam parametrům shaderů, což lze následně použít pro automatické předávání hodnot těchto parametrů z aplikace.

DXSAS je standard, který definuje způsob zadávání metadat a jejich význam tak, že aplikace a efekt, které se oba tohoto standardu drží, mohou vzájemně zcela automaticky komunikovat. DXSAS také definuje jednoduchý skriptovací jazyk, který lze zapsat v metadatech a který umožňuje automatizovat práci s více průchody, efektům komunikovat mezi sebou a řídit stav zobrazovacího řetězce.

Jak je tedy vidět, kombinace těchto dvou standardů je schopná vyřešit všechny problémy uvedené v úvodu tohoto paragrafu.

5.1.1 Efekty

Systém efektů se skládá ze dvou částí. První je definice formátu textových souborů, do kterých se efekty zapisují. Druhou je běhové prostředí a sada funkcí, které slouží k práci s tímto formátem. Běhové prostředí obsahuje funkce, které umožňují načtení efektu ze souboru, jeho kompilaci a aktivaci. Jejich výčet lze nalézt v [DirectX9.0c 05]. Daleko zajímavější je pro nás v tuto chvíli formát souborů s efekty.

Každý efekt je uložen v samostatném souboru, typicky s příponou .fx. Každý takový soubor se skládá ze tří bloků:

- Parametry
- Funkce
- Techniky a průchody

Parametry
Parametry jsou proměnné efektu. Deklarují se podobně jako proměnné v jiných programovacích jazycích a jejich účel je stejný. Výpis 5.1 ukazuje syntaxi parametru efektu. Většina prvků v deklaraci je stejná jako v jiných programovacích jazycích, až na dva zvláštní, kterými jsou anotace a sémantika.

```
usage  type  id  [: semantic]  [< annotation(s) >]  [= expression];
```

Výpis 5.1: Syntaxe parametru efektu

Sémantika, jak název napovídá, jsou metadata, která napovídají jednak kompilátoru, ale hlavně externí aplikaci, o smyslu příslušného parametru.

```
float4x4 worldViewProj : WORLDVIEWPROJECTION
<string UIWidget="none";>;
texture flameOutline
<
	string ResourceType = "2D";
	string ResourceName = "..\\Images\\FlameOutline2.dds";
>;
```

Výpis 5.2: Použití anotací a sémantiky

Anotace je další druh metadat, do kterých lze napsat libovolný text. Tento text není zpracováván kompilátorem, ale programem, který s efekty pracuje. Do anotací lze zapisovat dodatečné informace, jako například jména souborů s texturami, způsob nastavení grafického řetězce při práci s parametrem, ale také například vyžádat grafický manipulační prvek pro hodnotu toho parametru apod.

Výpis 5.2 ukazuje použití sémantiky a anotací. Nejdříve je deklarován parametr `worldViewProj` jako čtyřrozměrná matice a poté je sémantikou řečeno, že do tohoto parametru má být dosazována transformační matice složená z modelové, pohledové a projekční matice. Zároveň je u toho parametru anotací řečeno, že nemá být vůbec zobrazován v uživatelském rozhranní. Druhým deklarovaným parametrem je textura. Tento parametr nepoužívá sémantiku, pouze je anotací řečeno, že se jedná o dvou-dimensionální texturu a že se její obsah má načíst z uvedeného souboru.

Ještě jednou je třeba zdůraznit, že s anotacemi a sémantikou nepracuje kompilátor ani ovladač grafické karty, ale musí je podporovat aplikace, která s efekty pracuje (i když některé kompilátory používají sémantiku k optimalizaci uložení parametrů v registrech). Je dobré také poznamenat, že sémantika je pouze vodítko a není pro programátora závazná. Nic tedy programátorovi nebrání v tom, výše zmíněný parametr použít jakýmkoliv jiným způsobem, než jako transformační matici.

Funkce

Funkce jsou výkonnou částí efektu. Vertex a fragment shadery se zapisují právě jako funkce. Jejich syntaxe je podobná jazyku C. Funkce nemohou být vnořené a všechny tedy musejí být definované na globální úrovni. K parametrům efektu se chovají jako ke globálním proměnných. Uvnitř funkce lze tedy přistupovat k dříve deklarovaným parametrům. Kromě toho má samozřejmě funkce také svoje předávané parametry. Existuje pouze jeden způsob předávání a to hodnotou.

Techniky a průchody

Techniky jsou hybnou silou vykreslování (slovy specifikace [DirectX9.0c 05]). Technika sdružuje definici stavu zobrazovacího řetězce s voláním vertex a fragment shaderů. Každá technika obsahuje jeden nebo více průchodů. Každý průchod, přesně podle svého názvu, reprezentuje jeden průchod geometrie skrze zobrazovací řetězec. Má tedy vlastní sadu nastavení a také vlastní vertex a fragment shadery.

Dobrou otázkou je proč by efekt měl obsahovat více technik ? V odpovědi se skrývá jedna z velmi silných stránek celého systému efektů. U každé techniky lze nadefinovat typ hardwarové architektury, pro kterou jsou vertex a fragment shadery v této technice určeny. Běhové prostředí pro efekty pak obsahuje funkce, které jsou schopné tuto definici porovnat se schopnostmi hardware, kde efekt skutečně běží a vydat verdikt o tom, zda je technika spustitelná na tom kterém systému. Je tedy možné v jednom efektu definovat více technik, každou pro jinou skupinu hardware a bude automaticky zajištěno, že se nakonec spustí technika, která nejvíce odpovídá schopnostem hardware. To umožňuje s minimem práce psát programy tak, že jsou spustitelné na velmi široké škále hardware a přitom nemusí obětovat kvalitu zobrazení šíři podporovaného hardware. Jinými slovy, každý dostane přesně to, jaký má v systému hardware.

5.1.2 DXSAS

Výše popsaný systém anotací a sémantik sám o sobě ovšem nestačí. Aby byl skutečně k užitku, je potřeba obsah sémantik a anotací standardizovat. To přesně dělá DXSAS. Je to tedy standard, který definuje druhy sémantik a jejich význam a obsah anotací a jejich význam. DXSAS obsahuje jednak pravidla pro tvůrce efektů, tedy syntaktická pravidla, jak se mají sémantiky a anotace psát a jednak pravidla pro tvůrce aplikací, které s efekty pracují, kde se říká, co která sémantika znamená a jaká hodnota má být kam doplněna.

Příkladem sémantiky podle DXSAS je ve Výpis 5.2 sémantika `WORDVIEWPROJECTION`. Anotace v tomtéž výpisu se také drží DXSAS.

DXSAS definuje také jednoduchý skriptovací jazyk, který se zapisuje do anotací a umožňuje detailně kontrolovat průchod geometrie zobrazovacím řetězcem a také umožňuje interakci mezi dvěma či více efekty. Detailní popis tohoto skriptovacího jazyka je mimo rozsah této práce. Podrobný popis lze nalézt ve specifikaci [DirectX9.0c 05].

```
technique RayCast
<
       string Script =
              "ClearSetColor=clearColor;"
              "Pass = MainPass;";
>
{
       pass MainPass
    <
       string Script =
              "RenderColorTarget=;"
              "Clear=color;"
              "Draw=Geometry";
    >
    {
      …
    }
}
```

Výpis 5.3: Skript DXSAS v0.8

Na Výpis 5.3 můžeme vidět ukázku skriptu DXSAS. Vidíme také, že anotace lze přidávat k technikám a průchodům, stejně jako k parametrům.

5.1.3 Efekty a HLSL

Až dosud jsem se nezmínil, kromě pár slov v úvodní kapitole, o vysokoúrovňových programovacích jazycích a jejich vztahu k systému efektů.

```
{
  …
  VertexShader = compile vs_2_0 Transform();
  PixelShader  = compile ps_2_0 PassThrough(dTSampler);
}

{
  …
  VertexShader = asm{…};
  PixelShader  = asm{…};
}
```

Výpis 5.4: Použití HLSL a shader assembleru

HLSL (High-Level Shader Language) je jazyk, který byl vyvinutý pro psaní vertex a fragment shaderů. Před příchodem HLSL se vertex a fragment shadery psaly v jazyce podobném assembleru, který byl specifický pro každou verzi shaderů.

Formát efektů je z návrhu na HLSL nezávislý. Potud teorie. Syntaxe parametrů efektů odpovídá syntaxi HLSL a jako typ parametru lze zadat libovolný typ jazyka HLSL. Funkce mají také syntaxi HLSL a o návratových typech platí to samé, co o typech parametrů. Těla funkcí **musí** být napsána v jazyce HLSL. Syntaxe technik a průchodů není na HLSL přímo závislá. Je sice HLSL podobná, ale jazykové konstrukce jsou specifické pro formát efektů.

Pokud bychom chtěli používat systém efektů, ale vyhnout se použití HLSL, je to možné. V takovém případě nebudeme vůbec používat parametry a funkce, pouze deklarujeme příslušné techniky a průchody. V průchodu poté při deklaraci stavu vertex a fragment shaderů nepoužijeme klíčové slovo `compile` následované jménem funkce, ale klíčové slovo `asm` uvozující blok kódu v shader assembleru. Tento rozdíl ukazuje Výpis 5.4. Horní část kódu ukazuje použití HLSL funkcí jako vertex a fragment shaderů, dolní část pak použití shader assembleru.

5.1.4 Efekty a shadery mimo DirectX API

Z výkladu v předchozích kapitolách by se mohlo zdát, že pokud chceme používat systém efektů a vysokoúrovňový programovací jazyk, nemáme jinou šanci, než se přiklonit k DirectX API.

Naštěstí tomu tak není. Firma Microsoft spolu s firmou Nvidia původně vyvíjely vysokoúrovňový jazyk pro programovatelné stínování. Microsoft ho do svého DirectX API zařadil pod názvem HLSL a Nvidia jej představila jako Cg. Tyto jazyky se pak dále začaly trochu lišit, hlavně v podporovaných hardwarových profilech. Stále však platí, že kód v HLSL lze jen s minimálními úpravami přeložit kompilátorem pro Cg a naopak.

To samé platí i o formátu efektů. Firma Nvidia připravila vlastní formát efektů nazývaný CgFX. Tento formát je ovšem zcela totožný s efekt formátem uvedeným ve specifikaci DirectX. Liší se pouze běhové prostředí. Běhové prostředí CgFX umí pracovat jak nad OpenGL, tak nad DirectX.

Závěrem lze tedy říci, že všechno řečené v předchozích odstavcích lze aplikovat i na vývoj nad OpenGL API, s jazykem Cg a formátem CgFX.

5.2 Vývojové prostředí

V době psaní této práce je odvětví programovatelného stínování mladá oblast, a proto zatím neexistuje integrované programátorské prostředí, jako například Microsoft Visual Studio .NET. Je tomu tak z několika důvodů:

- Vertex a fragment shadery jsou relativně malé kusy kódu a jejich správa a udržování není zdaleka tak komplexním úkolem jako u jiných programátorských projektů. Proto po integrovaném prostředí není mezi programátory zatím taková poptávka.

- Tvorba nástrojů pro ladění je velmi obtížná. Vychází to z faktu, že grafické karty jsou masivně paralelní architektury a počet jednotlivých spuštění vertex nebo fragment shaderu je obrovský.
- Vertex a fragment shadery jsou malým kouskem skládanky v oblasti, kde se používají. Plnohodnotný grafický engine obsahuje spoustu dalších komponent, které by v integrovaném vývojovém prostředí musely být přítomny.

Nicméně i přes tyto obtíže existovalo v době zahájení této práce rozumné vývojové prostředí od firmy Nvidia s názvem FXComposer a to ve verzi 1.6. Zvažovanou alternativou použití toho prostředí bylo použití vlastního grafického enginu, který jsem vytvořil spolu s týmem dalších lidí při práci na projektu Almorea (viz. [Almorea]). Prostředí Almorea je plnohodnotný grafický engine, který je schopen pracovat právě s vertex a fragment shadery. FXComposer naproti tomu je prostředí vyvinuté jedním z průkopníků programovatelného stínování, firmou Nvidia.

Nakonec byl pro tuto práci zvolen FXComposer, zejména z následujících důvodů:

- FXComposer podporuje efektový formát podle specifikace DirectX.
- FXComposer poskytuje integrovaný textový editor přizpůsobený pro práci s vertex a fragment shadery v jazyce HLSL.
- Prostředí má poměrně rozumně vyřešenou práci s geometrií a navazování geometrických vlastností těles na parametry shaderů a je tak možné se zaměřit hlavně na vývoj vertex a fragment shaderů.
- FXComposer lze použít také jako experimentální prostředí. Je v něm integrovaný interaktivní náhled a cyklus vývoje je tak velmi urychlen. Zároveň poskytuje možnost měřit rychlost zobrazování ve snímcích za vteřinu.
- Podporuje práci 3D texturami.

Práce s FXComposer přinesla ovšem také některé nevýhody. Mezi ně patří zejména:

- FXComposer podporuje pouze DirectX a HLSL. Není možné v něm pracovat se Cg a OpenGL.
- Měření rychlosti zobrazování není úplně srovnatelné s herním enginem. Lze očekávat, že hodnoty naměřené uvnitř FXComposeru budou nižší.

Požadavkem, který v době vytváření této práce nemohlo uspokojit žádné vývojové prostředí, je možnost ladění vertex a fragment programů. Možnosti ladění jsou i v FXComposeru velmi omezené. K dispozici je pouze běžný grafický výstup. Ladění je tedy nutné provádět tak, že hodnoty, které chceme znát, nějakým způsobem převedeme na barevnou reprezentaci a následně přímo zobrazíme. V současné době jsou k dispozici pouze nástroje na analýzu výkonu. Tyto nástroje jsou schopny zobrazovat zatížení jednotlivých částí grafického systému a dovolují tak analyzovat a nalézat úzká místa.

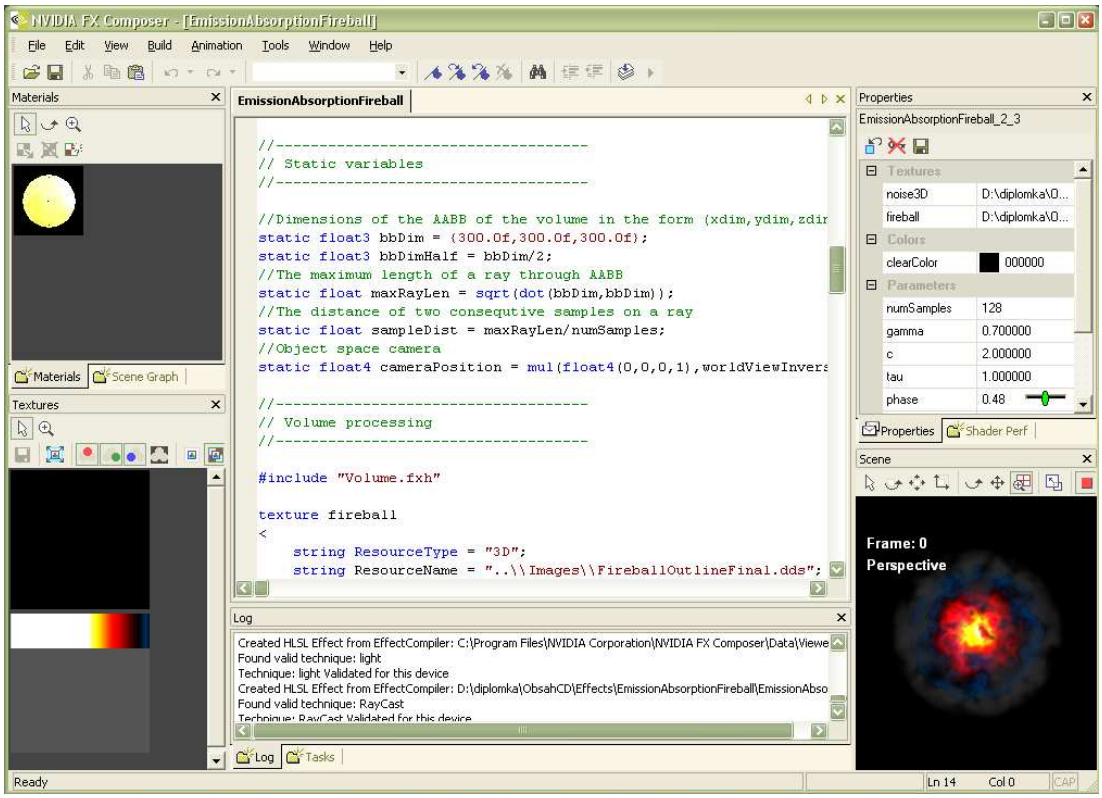
Kromě FXComposeru bylo pro vývoj programu, který připravuje experimentální data, použito prostředí Microsoft Visual Studio .NET. Pro vytvoření scén s proxy geometrií jsem použil modelovací nástroj Blender.

Obě prostředí, jak FXComposer, tak Blender jsou volně šiřitelné programy, Blender dokonce pod licencí LGPL. Pro práci s Microsoft Visual Studiem .NET byla použita instalace v universitních laboratořích.

5.2.1 FXComposer 1.6

Nyní stručně představím prostředí FXComposeru. Obrázek 5.1 ukazuje pracovní plochu FXComposeru. Centrální okno je textový editor se záložkami, který má vestavěný syntax-highlighting pro jazyk HLSL i pro efekt formát. Uvnitř FXComposeru se pracuje s pojmem materiál. Každé těleso ve scéně má přiřazeno nějaký materiál, který určuje vzhled tohoto tělesa. Takovýto materiál se skládá z právě jednoho efektového souboru.

Dole pod oknem pro práci s materiály je okno nazvané Textures. Jak název napovídá, jsou v tomto okně zobrazeny všechny textury, kterou jsou použité v právě aktivním materiálu.



Obrázek 5.1:FXComposer

Na pravé straně pracovní plochy nahoře je okno, kde jsou zobrazeny nastavitelné parametry aktivního materiálu. Zde se zobrazují právě prvky uživatelského rozhraní, které jsou specifikovány pomocí DXSAS anotací. Takto jsou například zpřístupněny parametry osvětlovacích modelů, jak jsou uvedeny v §4.3.

Napravo dole je vidět okno náhledu na scénu. Veškeré změny provedené v textovém editoru se po uložení souboru nebo jeho kompilaci ihned promítnou v okně náhledu. To umožňuje velmi rychlou reflexi na právě vykonané změny a značně tak zrychluje práci, zejména v konečných fázích tvorby efektu, kdy se jedná převážně o správné nastavení různých parametrů. V okně náhledu lze také interaktivně scénou procházet či otáčet a pohybovat tělesy v ní.

Dole uprostřed nalezneme i nezbytné okno s hlášeními. Zde se zobrazují zprávy o kompilaci a jejích případných chybách.

Nakonec se volba FXComposeru za vývojové prostředí ukázala jako zdařilá. Vývoj v něm probíhal, i přes jeho drobné nedostatky, pohodlně a relativně rychle. Ve verzi

1.6 je jeho asi největší nevýhodou stabilita, se kterou má občas problémy a také způsob práce s geometrií a se scénou není ve všech případech úplně ideální. Ovšem odvětví vývojových prostředí pro programovatelné stínování je teprve v začátcích a velmi rychle se rozvíjí. Již během tvorby této práce vyšla nová verze 1.7.

5.3 Struktura kódu

Nyní jsme již vybaveni potřebnými znalostmi principů práce s programovatelným řetězcem a znalostí vývojového prostředí, takže je možné prezentovat strukturu kódu pro implementovanou metodu.

V rámci této práce bylo vytvořena sada efektů, tak jak jsou popsány výše. Pro vývoj bylo použit systém Efektů podle specifikace [DirectX9.0c 05], k programování shaderů byl použit jazyk HLSL a DXSAS ve verzi 0.8. Podrobněji jsou důvody pro tuto volbu popsány v následujícím paragrafu o vývojovém prostředí.

Pro každý implementovaný optický model, tak jak jsou popsány v §4.3 byl vytvořen samostatný efekt, který je uložen v jednom souboru s příponou .fx. Dále byla vytvořena sada „hlavičkových souborů“, které neobsahují přímo efekty, ale pouze pomocné HLSL funkce a deklarace efektových parametrů, které jsou společné pro většinu efektů. Jedná se o následující soubory:

- *Geometry.fxh*. Tento soubor obsahuje funkce, které počítají průsečík paprsku s proxy geometrií
- *Volume.fxh*. Zde jsou funkce, které se starají o práci se souřadnicemi a vyzvedávání vstupních objemových dat z 3D textur.
- *Lighting.fxh*. Tento soubor obsahuje definice světel a funkce pro práci s nimi, včetně všech použitých BRDF.

Nakonec, kromě efektů pro optické modely, byl vytvořen ještě soubor *SimpleTexture.fx*, který obsahuje kód pro jednoduché mapování textury na objekt. Tento efekt je použitý v některých ukázkových scénách, například při tvorbě Obrázek 3.2.

Popis všech souborů spolu s jejich adresářovou strukturou je v Dodatku A.

Pro testování efektů byly použity tři experimentální scény. První z nich, s názvem *Box300.x* obsahuje krychli o straně 300 jednotek s středem v počátku. Druhá s názvem *Box500a300.x* obsahuje, jak název napovídá, centrovanou krychli o straně 300 jednotek a pod ní umístěný kvádr o rozměrech 500x500x2. Poslední scéna *Box2-2-4.x* byla připravena pro scénu s ohněm (viz Obrázek 3.3) a obsahuje centrovaný kvádr o stranách 200x200x400. Tyto krychle a kvádry slouží jako proxy geometrie pro zobrazovaný objem.

Celková struktura efektového souboru je stejná pro všechny druhy efektů. Její schématické znázornění je na Výpis 5.5.

Každý efekt začíná hlavičkou, kde jsou uvedeny informace o daném souboru a optickém modelu, který implementuje.

Další část *Adjustable parameters* obsahuje všechny parametry efektu, které jsou přístupné uživateli. Všechny tyto parametry mají nadefinován uživatelský prvek uvnitř FXComposeru. Typicky tu najdeme parametry optického modelu, počet vzorků na paprsku a barvu pozadí scény.

```
//-----------------------------------------------------------------------
// GPU Volume Rendering
// Lukas Marsalek, 2005
// ...
//-----------------------------------------------------------------------

//-------------------------------------
// Adjustable parameters
//-------------------------------------
   ...
//-------------------------------------
// Transformation matrices
//-------------------------------------
   ...
//-------------------------------------
// Static variables
//-------------------------------------
   ...
//-------------------------------------
// Volume and Lights processing
//-------------------------------------
#include "Volume.fxh"
#include "Lighting.fxh"
   ...
//-------------------------------------
// Geometry processing
//-------------------------------------
#include "Geometry.fxh"
//-------------------------------------
// I/O structures
//-------------------------------------
   ...
//-------------------------------------
// Vertex Shaders
//-------------------------------------
   ...
//-------------------------------------
// Fragment Shaders
//-------------------------------------
   ...
//-------------------------------------
// Script entry point
//-------------------------------------
   ...
//-------------------------------------
// Drawing
//-------------------------------------
   ...
```

Výpis 5.5: Struktura efektů

Následují všechny transformační matice v části nazvané, jak jinak, *Transformation matrices*. Tyto matice jsou dodávány FXComposerem pomocí DXSAS sémantik, které tyto standardní matice označují.

Další na řadě jsou statické proměnné *Static variables*. Tyto proměnné jsou specifické v tom, že jsou inicializovány pouze jednou při zavedení efektu a jejich inicializátor může obsahovat i komplikované HLSL konstrukce. Tato inicializace je prováděna na CPU, nikoliv na GPU. Těchto vlastností lze s výhodou využít k předpočítání některých „drahých“ hodnot, které se během práce efektu nemění, ale jejichž výpočet zahrnuje například dělení či odmocňování, apod.

Následující část *Volume and Lights Processing* již obsahuje kromě parametrů také HLSL funkce. Nejdříve je vložen soubor s funkcemi, které se starají o výběr vstupních hodnot z 3D textury. Dále je vložen soubor, kde jsou deklarována vnější

světla a funkce pro práci s nimi. Následují funkce, které mapují vstupní hodnoty na fyzikální vlastnosti vyjádřené v optických modelech, jako například útlum či vyzařování. Oddíl je zakončen funkcemi, které se starají o zpracování vnějšího osvětlení, přesněji jedna o útlum při průchodu objemem a druhá o odraz světla při dopadu na částice látky. U efektů, které neobsahují interakci s vnějším osvětlením je tento oddíl redukován pouze na práci se vstupními daty.

Další část *Geometry processing* pouze vkládá soubor, kde jsou funkce pro výpočet průsečíků paprsku s proxy geometrií.

I následující část *I/O structures* je krátká. Jsou v ní definovány dvě struktury. První z nich `vertexInput` udává tvar vstupu do vertex shaderu a druhá `vertexOutput` naopak výstup vertex shaderu a zároveň vstup do fragment shaderu. Struktura pro výstup fragment shaderu není nutná, protože fragment shader vrací vždy pouze čtyř-dimensionální vektor barvy fragmentu.

Následují dvě důležité části, *Vertex Shaders* a *Fragment Shaders*. Jejich obsah je zřejmý z názvů. Dodejme pouze, že každá obsahuje právě jeden vertex nebo fragment shader. Předposlední krátká část *Script entry point* obsahuje povinný parametr vyžadovaný standardem DXSAS. Tento parametr musí mít sémantiku `STANDARDSGLOBAL` a v anotaci obsahuje vstupní deklarace pro DXSAS skript a jako hodnotu má použitou verzi DXSAS.

Poslední část nazvaná *Drawing* obsahuje popis technik a průchodů. Tato část se liší podle způsobu, jak jsou generovány paprsky (viz §4.2.1).

```
technique RayCast
<
       string Script =
              "ClearSetColor=clearColor;"
              "Pass = BackPass;"
              "Pass = MainPass;";
>
{
    pass BackPass
       <
              string Script=
                     "RenderColorTarget = rayTexture;"
                     "ClearSetColor=texClearColor;"
                     "Clear=color;"
                     "Draw=Geometry;";
       >
       {
              …
              VertexShader = compile vs_2_0 XYZColor_VS();
              PixelShader  = compile ps_2_0 BackRay_PS(cameraPosition);
       }
    pass MainPass
    <
       string Script =
              "RenderColorTarget=;"
              "ClearSetColor=clearColor;"
              "Clear=color;"
              "Draw=Geometry";
    >
    {
    …
    VertexShader = compile vs_3_0 Volume_VS();
    PixelShader=compile ps_3_0 Volume_PS(rayTextureSampler,cameraPosition);
    }
}
```

Výpis 5.6: Zápis dvou průchodu a jejich spolupráce

Při generování paprsku geometrickým přístupem je prováděn pouze jeden průchod a je definován jeden vertex shader a jeden fragment shader, které jsou oba kompilovány pro Shader Model 3.0.

Při generování paprsku pomocí dvou průchodu jsou evidentně průchody dva. Tyto jsou uloženy v jedné technice a kontrolovány pomocí DXSAS skriptu.

Výpis 5.6 ukazuje, jak jsou pomocí DXSAS skriptu spuštěny oba dva průchody postupně za sebou. Všimněme si, že v prvním průchodu `BackPass` je nastavena dočasná textura `rayTexture` jako cíl pro vykreslování a následně je v druhém průchodu vykreslovací cíl opět vrácen na hlavní barevný buffer. Každý z průchodů má také vlastní sadu vertex a fragment shaderů.

5.4 Implementační zajímavosti

V této podkapitole se zaměřím na nejzajímavější části celé implementace. První z nich se zabývá způsobem rozdělení práce mezi vertex a fragment shader a obsahem vertex shaderu. §5.4.2 obsahuje popis konstrukce hlavního fragment shaderu, který obstarává většinu „práce“. Nakonec jsou v §5.4.3 představeny geometrické výpočty průsečíku paprsku s proxy geometrií, které slouží jako pěkná ukázka výhod vektorových procesorů.

5.4.1 Vertex shader

Rozdělení práce mezi vertex a fragment shader je velmi nerovnoměrné. Vertex shader, jak ukazuje Výpis 5.7, pouze předává dál do fragment shaderu světové a modelové souřadnice proxy geometrie. Zbytek práce se provádí uvnitř fragment shaderu.

```
vertexOutput Volume_VS(vertexInput input)
{
    vertexOutput o;
    o.pos = mul(float4(input.pos,1),worldViewProj);
    o.fragObjPos = input.pos;
    o.fragTexPos = o.pos;
       return o;
}
```

Výpis 5.7: Hlavní vertex shader

Toto nerovnoměrné rozdělení vzniká z povahy použité metody vrhání paprsku. Potřebujeme totiž vrhat paprsek skrze každý fragment a určovat v něm výslednou barvu nezávisle na ostatních. Pokud bychom ten samý postup provedli pouze v několika málo vrcholech proxy geometrie, byly by výsledky nepoužitelné.

Faktem ovšem je, že toto nerovnoměrné zatížení není ideální z výkonnostního hlediska. Většinu času, kdy fragmentové jednotky pracují na plný výkon, tak vertex procesory stojí. Zajímavým směrem pro další výzkum je, pokusit se zátěž rovnoměrněji rozdělit mezi vertex a fragment shader. To by šlo například udělat tak, že proxy geometrie bude obsahovat daleko více rovnoměrně rozprostřených vrcholů. Potom by se ekvivalentní proces mohl provést také ve vertex shaderu a ve fragment shaderu pak pouze dopočítávat potenciálně zajímavá místa. Stojí za to podotknout, že tento postup je možný pouze v Shader Modelu 3.0. Abychom mohli výše zmíněnou

myšlenku provést, je totiž nutné umět uvnitř vertex shaderu číst z textur. A to nebylo až do příchodu Shader Modelu 3.0 možné.

5.4.2 Fragment shader

Přes velké množství práce, kterou fragment shader provádí, je jeho struktura jednoduchá.

```
1 float4 Volume_PS(vertexOutput input,
2                           uniform float4 objSpaceCamera): COLOR
3 {
4     float3 v = normalize(input.fragObjPos - objSpaceCamera.xyz);
5     float3 s = objSpaceCamera.xyz;
6     float2 rayBounds = RayAnalytical(s,v,bbDim);
7
8     float3 sampleCoords;
9     float3 samplePoint;
10    float4 volumeSample;
11    float extinction = 0.0f;
12    float currentExt;
13    float3 emission = 0.0f;
14    float t = rayBounds.x;
15
16    while(t < rayBounds.y)
17    {
18           samplePoint = s+t*v;
19           sampleCoords=PrepareSampleCoords(samplePoint,bbDimHalf,bbDim);
20           volumeSample = SampleVolume(sampleCoords);
21           currentExt = TransferExtinction(volumeSample);
22           extinction += currentExt;
23           emission +=
24                 (TransferEmission(volumeSample,currentExt)+
25                  Scattering(currentExt,v,samplePoint))
26                  *exp(tau*sampleDist*(-extinction));
27           t+=sampleDist;
28    }
29    return float4(emission,exp(tau*sampleDist*(-extinction)));
30 }
```

Výpis 5.8: Hlavní fragment shader

V úvodu je nejprve určen směrový vektor (řádek 4) a počáteční bod (řádek 5) paprsku, který určí hodnotu právě zpracovávaného fragmentu. Následně na řádku 6 je analyticky určen průsečík tohoto paprsku s proxy geometrií, která je zadána třírozměrným vektorem délek stran `bbDim`.

Další akce se začíná dít až na řádku 16. Zde je stěžejní moment celé implementace. Tím je použití while cyklu s podmíněným ukončením. Počet iterací toho while cyklu nelze určit při kompilaci, a proto musí být skutečně realizován instrukcí pro podmíněný skok `break_ge x y`, která je umístěna uvnitř bloku `rep/endrep`.Tato funkcionalita je dostupná uvnitř fragment shaderů až od Shader Modelu 3.0. Bez těchto podmíněných skoků by bylo nutné realizovat celou tuto smyčku násobnými průchody.

Vnitřek smyčky již přesně kopíruje strukturu modelů z §4.3. Výpis 5.8 ukazuje nejsložitější model, tedy rozptyl s vrženými stíny. U jednodušších modelů se liší řádky 21 až 26. Na řádku 21 a 22 je akumulován útlum osvětlení přicházejícího zpoza objemu, který je nakonec aplikován na řádku 29. Vyzařování, jak vlastní tak externí, je sčítáno v proměnné `emission`. Jak vidíme, vlastní vyzařování je sečteno z rozptýleným světlem a výsledná hodnota je upravena doposud nasčítaným

útlumem. Je tedy vidět, že vzorkování paprsku běží od pozorovatele směrem dovnitř objemu.

5.4.3 Geometrické výpočty

Zajímavou ukázkou práce s vektory jsou výpočty průsečíku paprsku z proxy geometrií. Výpis 5.9 ukazuje způsob výpočtu vztahu (4.2).

```
void ComputeSegments(float3 S, float3 v, float3 dims,
                                          out float3 tn, out float3 tf)
{
      float3 invV = 1/v;
      tn  = (-dims/2.0f - S)*invV;
      tf = tn + dims*invV;
}
```

Výpis 5.9: Výpočet vztahu (4.2)

Vidíme, že průsečíky pro všechny tři osy jsou počítány najednou pomocí vektorových operací. Instrukce GPU jsou navíc opravdu vektorové, takže uvedené operace se nepřeloží do trojnásobného počtu skalárních operací, ale opravdu jsou provedeny tak, jak jsou zapsány.

Na Výpis 5.10 vidíme, že se vyplatí vektorizovat i skalární výpočty. Je tedy lepší uložit skalární hodnoty do vektoru a provést jednu vektorovou operaci, než dvě oddělené skalární.

```
1 float2 Intersect(float3 tn, float3 tf)
2 {
3     float2 final;
4     float2 t = max(tn.xx,tn.yz);
5     final.x = max(t.x,t.y);
6     t = min(tf.xx,tf.yz);
7     final.y = min(t.x,t.y);
8     return final;
9 }
```

Výpis 5.10: Výpočet průniku intervalů

Výpis 5.10 ukazuje výpočet správné dvojice průsečíků paprsku s proxy geometrií. Detaily viz §4.2.1. Uvedená vektorizace je vidět např. na řádku 4, kde dvě skalární porovnání jsou provedena na dvourozměrných vektorech, navíc s pomocí tzv. swizzlingu, který umožňuje přeuspořádávat a vybírat komponenty vektorů podle potřeby.

5.5 Experimentální data a práce s nimi

Na závěr se zmíním o způsobu implementace algoritmu pro přípravu experimentálních dat tak, jak je popsán v §4.4. Budu se také věnovat způsobu přenosu a práci s těmito daty uvnitř fragment shaderů.

5.5.1 Implementace šumu a turbulence

Při tvorbě experimentálních data bylo hlavním cílem rychle a jednoduše vytvořit rozumnou množinu experimentálních objemových dat. Proto jsem zvolil jednoduchý objem vyplněný turbulencí Perlinova šumu.

Implementaci algoritmu pro generování šumu jsem převzal podle [Perlin]. Zde je uvedena kompletní implementace algoritmu z [Perlin 02] v jazyce C.

Tato implementace byla doplněna o kód, který zajišťuje převedení vypočítaných dat do formátu DDS a jejich uložení na disk. Formát DDS se ukázal jako nejvhodnější pro uložení potřebných dat z několika důvodů:

- Je schopen ukládat objemové textury jako sadu dvou-dimensionálních průřezů.
- Existuje rozumné API pro práci s tímto formátem, které je přímo dostupné v [DirectX9.0c 05] a umožňuje snadno formát číst a ukládat.
- Formát podporuje několik různých způsobů a velikostí uložení dat. Je schopen pracovat jak s formáty používajícími 32bitů na pixel, tak s formáty používajícími 64bitů na pixel. Tyto 64bitové formáty ukládají hodnoty ve formě 16bitových čísel s plovoucí řádovou čárkou.
- Experimentální prostředí FXComposer poskytuje dobrou podporu toho formátu, včetně jeho objemových 64bitových variant.
- V [DirectX9.0c 05] je také prohlížeč formátu DDS, který opět zvládá i práci s objemovými texturami.

5.5.2 Práce s objemovými daty

Jazyk HLSL přímo podporuje 3D textury. Je zde definovaný speciální druh textury a k ní příslušející sampler, který umí s 3D texturami pracovat. Pomocí DXSAS lze pak do této textury přímo načíst objemová data uložená ve formátu DDS.

```
texture noise3D
<
string ResourceType = "3D";
string ResourceName = "..\\Volumes\\SphereTurb.dds";
>;

sampler noise3DSampler = sampler_state
{
texture = <noise3D>;
AddressU  = WRAP;
AddressV  = WRAP;
AddressW  = WRAP;
MIPFILTER = LINEAR;
MINFILTER = LINEAR;
MAGFILTER = LINEAR;
};
```

Výpis 5.11: Použití 3D textur

Takto načtená data nejsou ještě v dalším kódu přímo používána. Pro manipulaci s nimi slouží dvě metody: `PrepareSampleCoords` a `SampleVolume`. První z nich má na starosti převod mezi souřadnicovými systémy. Souřadnice do textur v HLSL jsou vždy v rozmezí $[0,1]$. Z výpočtu při generování paprsku ovšem přicházejí data v souřadném systému proxy geometrie. Funkce `PrepareSampleCoords` převádí právě mezi těmito souřadnými systémy pomocí znalosti velikosti proxy geometrie.

Funkce `SampleVolume` pouze pomocí funkcí standardní knihovny HLSL vyzvedává hodnoty z 3D textury a vrací je jako čtyř-dimensionální vektor. Je zde ovšem jeden drobný trik, který není na první pohled zcela zřejmý.

```
float4 SampleVolume(float3 samplePoint)
{
	return tex3Dlod(noise3DSampler,float4(samplePoint,0));
}
```

Výpis 5.12: Metoda `SampleVolume`

Běžně by v této metodě byla standardní funkce `tex3D`, která vyzvedává hodnoty z 3D textur. Tato funkce ovšem implicitně pracuje s mip-úrovněmi, a to i když je textura explicitně neobsahuje. Výběr správné mip-úrovně zahrnuje odhad gradientu texturovacích souřadnic, který je prováděn aproximací pomocí diferencí v sousedních fragmentech výsledného obrazu. Funkce `SampleVolume` je ovšem volána uvnitř smyčky, která obsahuje dynamické větvení. Uvnitř dynamického větvení není ale použití aproximace pomocí diferencí povoleno. Je to z toho důvodu, že je možné, že kód pro sousední fragment, jehož texturovací souřadnice potřebuje znát, se vydal jinou větví dynamického větvení a tedy požadovaná hodnota nemusí být vůbec k dispozici. Pokud bychom se pokusili použít funkci `tex3D`, skončí kompilace s chybovou hláškou, která naneštěstí navíc ohlásí příčinu chyby kompilace jako překročení maximálního počtu iterací smyčky.

Naproti tomu funkce `tex3Dlod` nechává výběr mip-úrovně na uživateli. Jak si lze všimnout na Výpis 5.12, funkce má jako druhý parametr čtyř-dimensionální vektor, i když k vyhledání hodnoty v textuře stačí souřadnice tři. Čtvrtá hodnota je právě mip-úroveň, která se má použít. *0* značí základní úroveň.

5.6 Testování a výsledky

Testování rychlosti metody jsem prováděl přímo v prostředí FXComposeru, který umožňuje sledovat počet snímků za vteřinu.

Konfigurace testovacího počítače byla následující:

AMD Athlon 64 3000+
512 MB RAM
MSI TD128E NX6600GT, 128MB RAM
Microsoft WindowsXP Professional, ServicePack 2
FXComposer v1.6

Testovaní jsem prováděl na třech scénách a ve třech konfiguracích. Všechny testovací scény měly stejný obsah, lišily se pouze použitým shaderem, tedy simulovaným optickým modelem. V každé scéně bylo obsaženo jedno bodové světlo a jedna krychle obsahující objemová data. Použité optické modely byly následující:

- **Emission**. Vyzařující optický model. Co se týče výpočetní náročnosti, je z použitých modelů nejjednodušší.
- **HG**. Pohlcující a vyzařující model používající Henyey-Greensteinovu fázovou funkci. Leží uprostřed spektra výpočetní náročnosti.
- **HGAttenuated**. Pohlcující a vyzařující model používající Henyey-Greensteinovu fázovou funkci. Osvětlení z vnější zdrojů je na své dráze

objemem utlumeno. Tento model koresponduje s Rozptylujícím modelem a vrženými stíny, podle §4.3. Z použitých modelů je nejvíce výpočetně náročný.

Všechny tyto tři modely byly testovány v různých konfiguracích:

1. Jako objemová data je použita turbulence v mřížce 64x64x64. Výstup je vykreslován do okna 512x512 pixelů.

Model	100 vzorků	64 vzorků	50 vzorků
Emission	50	74	92
HG	21	32	57
HGAttenuated	<1	2	5

Tabulka 5.1: Výsledky při konfiguraci 1

2. Jako objemová data je použita turbulence v mřížce 64x64x64. Výstup je vykreslován do okna 256x256 pixelů.

Model	100 vzorků	64 vzorků
Emission	203	303
HG	91	139,7
HGAttenuated	4	9

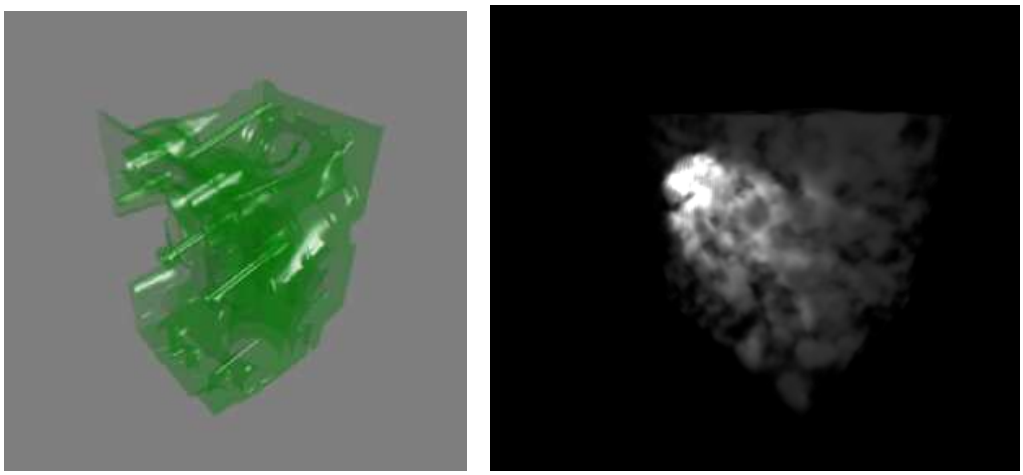
Tabulka 5.2: Výsledky při konfiguraci 2

3. Jako objemová data je použita turbulence v mřížce 128x128x128. Výstup je vykreslován do okna 512x512 pixelů.

Model	50 vzorků
Emission	87
HG	44
HGAttenuated	4

Tabulka 5.3: Výsledky při konfiguraci 3

V rámci každé konfigurace ještě byly měřeny hodnoty s různým počtem vzorků na paprsku, jmenovitě 100, 64 a 50 vzorků.

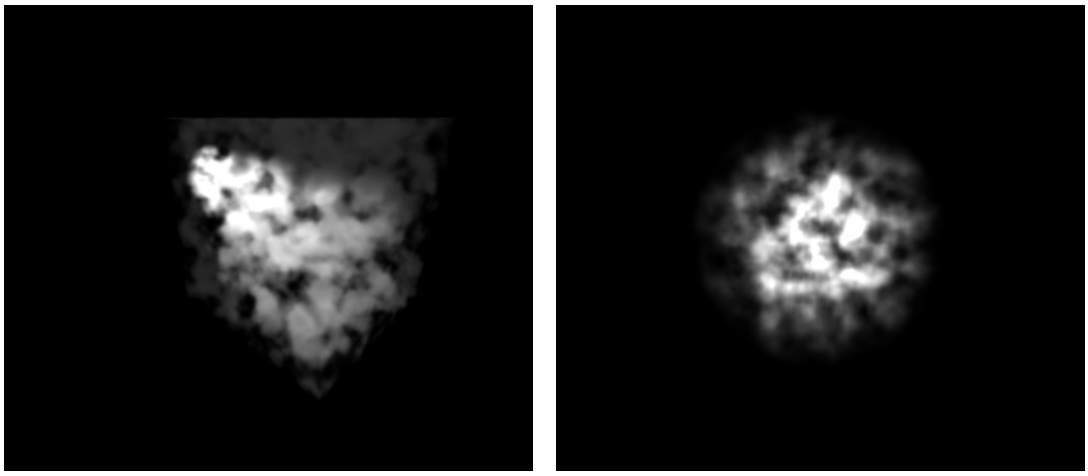


Obrázek 5.2: Testovací scéna ze [Stegmaier et al 05], odtud převzato. Testovací scéna HGAttenuated

Tabulky 5.1 – 5.3 ukazují naměřené hodnoty fps. Měření bylo prováděno tak, že na testovacím počítači běžela pouze aplikace FXComposer. Po načtení scény jsem počkal 30 vteřin a poté jsem odečetl naměřené fps.

Výsledky z konfigurace 1 nejsou příliš překvapivé. Ukazují, že metoda je skutečně použitelná pro zobrazování v reálném čase. Ve shodě s očekáváním je počet vzorků na paprsku jedním ze zásadních činitelů. Naneštěstí je také zásadním faktorem při kvalitě výstupu. 64 vzorků je ještě přípustných, aby kvalita výsledného obrazu nebyla příliš nízká. 100 vzorků je ideálních. Při zvyšování nad 100 vzorků je vzrůst kvality znatelný jen minimálně. Zajímavé jsou extrémně nízké hodnoty u nejsložitějšího optického modelu. Vzhledem ke způsobu implementace nejsou ale příliš překvapením a ani není možné z nich odvozovat nějaké zásadní závěry. Způsob práce se „stínovacími“ paprsky má velké rezervy a lze očekávat řádový nárůst při použitích některých optimalizací (viz §6.1).

Konfigurace 2 byla zahrnuta do testování, aby bylo možné odhadnout, do jaké míry je implementace limitována výkonem fragmentových procesorů karty. Konfigurace 2 je totožná s konfigurací 1, pouze vykresluje čtyřikrát méně pixelů. Pokud se podíváme na naměřené hodnoty, zjistíme, že jsou s drobnými odchylkami čtyřikrát větší. To znamená, že implementace je skutečně limitovaná výkonem fragmentových procesorů. Je zde tedy prostor pro optimalizaci navrhovanou v §5.4.1 a její realizace zřejmě přinese významné zvýšení výkonu. Je důležité také vzít v úvahu, že použitá grafická karta řady 6600GT je karta vyšší střední třídy a oproti nejvyšší třídě karet řady GeForce 6, 6800 Ultra má dvojnásobně menší výkon. V současné době se objevila navíc nová řada karet GeForce 7, která má podle testů ještě o 70% vyšší výkon, že GeForce 6800 Ultra. Pokud k tomu přidáme ještě technologii SLI od firmy Nvidia, která umožňuje použití dvou grafických karet v jednom systému s nárůstem výkonu také řádově o 60-70%, dostáváme se k teoretickým hodnotám okolo 110fps pro středně obtížnou scénu HG vykreslovanou se 100 vzorky do okna 512x512.



Obrázek 5.3: Testovací scéna HG (vlevo), Testovací scéna Emission (vpravo)

Konfigurace 3 byla do testů zařazena pro porovnání s ostatními prácemi podobného zaměření. Úplně korektní přímé srovnání není možné, pokusme se alespoň o odhad. V červnu 2005 byla na konferenci Volume Graphics prezentována práce [Stegmaier at al. 05], která se zabývá totožnou problematikou a implementuje postupy velmi podobné těm prezentovaným v této práci. Autoři prezentují výsledky na následující konfiguraci:

- Standartní PC s GeForce6800 Ultra
- Velikost dat 256x256x110
- Ve scéně je vykreslen jeden model shaderem, který napodobuje průhledný materiál.
- Paprsek je vzorkován přibližně 50 vzorky.

V této konfiguraci dosahují průměrné rychlosti 5.7 fps a maximální 6.7fps.V konfiguraci 3 byl zvětšen objem vstupních dat na 128x128x128 tak, aby výsledky byly rozumně srovnatelné.

Mřížka 64x64x64 obsahuje 256 144 bodů a mřížka 128x128x128 obsahuje 2 097 152 bodů. Přestože tedy konfigurace 3 má osmkrát více vstupních dat, naměřené hodnoty jsou v průměru pouze 15% nižší než v konfiguraci 2.

Výsledky z konfigurace 3 lze, alespoň přibližně srovnat s výsledky Stegmaiera et al. Jejich testovací sada má 3,5krát více vzorků, ovšem jejich systém má řádově dvakrát rychlejší grafickou kartu. Je tedy vidět, že i nejpomalejší z modelů HGAttenuated dosahuje srovnatelných nebo vyšších rychlostí. Středně komplexní model dosahuje dokonce o řád větších rychlostí.

Je ovšem nutné poznamenat, že výše uvedené srovnání je přibližné a nelze z něj vyvozovat přesné závěry, zejména z následujících důvodů:

- Je možné, že zvýšením objemu vstupních dat na 256x256x110 bychom dosáhli kritické úrovně a zpomalení by mohlo být řádové.
- Dvojnásobná rychlost karty GeForce 6800 Ultra je teoretická.
- Na rychlosti zobrazování se výrazně projevuje velikost oblasti okna, kterou objem pokrývá. Toto pokrytí není možné zvolit tak, aby si scény přesně odpovídaly.

5.7 Diskuse

Na závěr kapitoly o implementaci bych rád uvedl diskusi této implementace ve světle původně stanovených cílů. Projdeme každý z cílů samostatně:

Rychlost zobrazování (c1)
Ve světle předchozí sekce lze konstatovat, že rychlost metody je pro experimentální implementaci velmi uspokojivá. Dosahuje srovnatelných výsledků s ostatními prácemi a dokonce se zdá, že je schopna je předstihnout. Pro použití ve vědecké vizualizaci jsou již současné rychlosti, kromě nejsložitějšího modelu, zcela dostatečné. Pro použití v počítačových hrách by bylo potřeba dosahovat rychlostí jednou až dvakrát takových, řekněme tedy okolo 120fps pro HG v konfiguraci 3. Nicméně i tak je možné být optimistický, neboť jak bylo zmíněno v minulých sekcích, tato experimentální metoda má ještě velké rezervy v efektivitě a rychlosti, zejména v těchto oblastech:

- Měření je prováděno na kartě střední vyšší třídy. Špičky současného konzumního hardware jsou schopny dosahovat přibližně pětkrát většího výkonu
- Nejpomalejší shader HGAttenuated má značné rezervy v efektivitě implementace a lze po úpravě očekávat řádový nárůst.

Je ovšem třeba zmínit fakt, že rychlost zobrazování je značně závislá na velikosti oblasti, kterou objem na obrazovce pokrývá. To by se mohlo u interaktivních aplikací či her ukázat jako nepříjemný problém.

Kombinace s jinými zobrazovacími metodami(c2)
Implementovaná metoda je schopna bez problémů pracovat dohromady se standardními metodami pro polygonální objekty. Jediné omezení současné implementace je objemová data se nesmějí protínat s jinou geometrií, ať je to jiný objem nebo polygonální data.

Implementace výhradně uvnitř shaderů(c3)
Tento cíl splněn beze zbytku. Celá metoda je implementovaná pomocí jednoho efektu, který obsahuje právě jeden vertex shader a jeden fragment shader a sadu globálních parametrů, které zajišťují navázání na externí aplikaci.

Samostatnost a přenositelnost(c4)
Tím, že je celá metoda implementována pomocí efektu, stává se tak přenositelnou, jak to jen je v současné době možné. Drobnou nevýhodou je, že musí být kompilována pro Shader Model 3.0, který v současné době podporují pouze dvě řady karet firmy Nvidia. Lze ale v nejbližší době očekávat podporu tohoto modelu i ze stranu druhého velkého výrobce grafický karet, firmy ATI.

Celkově lze tedy říci, že práce dosáhla stanovených cílů, v některých ohledech dokonce překvapivě dobře.

Zajímavé jsou zkušenosti, které byly během tvorby této práce získány. Jedním z přínosů této práce je získání zkušeností s platformou programovatelného stínování na konzumních počítačích a získaní velmi dobré představy o jejích možnostech. Ukázalo se, že tato platforma během své krátké šestileté existence dosáhla značného stupně vyspělosti. Šíře implementovatelných algoritmů je daleko větší, než se v začátcích této práce zdálo. Stejně tak programovací prostředí je relativně pohodlné a produktivní. Lze jenom uzavřít, že další výzkum v oblasti zobrazování objemových dat na platformě konzumních grafických systémů má mnoho zajímavých možností k rozvoji a skutečnému uplatnění.

Kapitola 6

Závěr

Tato diplomová práce prezentuje zobrazování objemových dat jako ucelenou oblast počítačové grafiky, která stojí na dobrém teoretickém základě. Ukazuje také šíři přístupů, které byly pro řešení tohoto problému vyvinuty. Na základě těchto znalostí a studia prostředí současných konzumních grafických systémů prezentuje implementaci metody zobrazování objemu pro tuto platformu.

Podrobněji lze výsledky této práce shrnout do následujících bodů:

- Nejprve je podán ucelený úvod do problémů, kterých se tato práce dotýká, tedy zobrazování objemových dat, zobrazování v reálném čase a architektury současných konzumních grafických systémů.
- V následujících dvou kapitolách je systematicky prezentován vývoj oblasti zobrazování objemových dat. Je představena celá šíře používaných přístupů, od počátků studia tohoto problému až po současnost. Důraz je kladen na obecné principy těchto metod, vysvětlení jejich vzájemných odlišností, výhod a nevýhod.
- V kapitole 3 je podán teoretický základ pro fotorealistické zobrazování objemových dat. Na základě předchozích teoretických prací je ukázáno, že principy, kterými se zobrazování objemových dat řídí, lze budovat jednak systematicky na základě fyzikálních zákonů, ale i empiricky podle potřeb aplikační oblasti. To ukazuje sílu metod pro zobrazování objemu a jejich širokou použitelnost.
- Za použití všech získaných informací je navržena metoda pro zobrazování objemových dat na konzumním hardware. Jejími hlavními rysy jsou universálnost, co se týče použitelných optických modelů a přenositelnost mezi různými grafickými aplikacemi. Při návrhu byl kladen důraz na její správnost a teoretický základ pro použité optické modely.
- Dále je navržená metoda ověřena experimentální implementací na cílové platformě. Implementace sloužila zejména k přímému ověření použitelnosti platformy konzumních grafických systému pro metody zobrazování objemu. Významných přínosem je také zhodnocení problémů a nedostatků této architektury.

- Při hledání vhodného implementačního prostředí a nástrojů byla prověřena většina z dostupných softwarových balíků pro oblast programovatelného stínování.

Mezi vlastní příspěvky této práce patří zejména:

- Na základě studia dostupné literatury byl přesněji formulován, mnohdy jen vágně používaný, pojem zobrazování v reálném čase.
- Shrnutí celé oblasti zobrazování objemových dat na základě původních vědeckých prací. Základ většiny doposud používaných metod je vysvětlen a jednotlivé přístupy jsou porovnány a zhodnoceny jejich výhody a nevýhody.
- Optické modely jsou implementovány přesně podle teoretických modelů a je do nich přidána sada konstant tak, aby bylo možno rychle získat, co nejlépe vypadající výstup.
- Implementace navržené metody s použitím nejnovějších postupů a metod v oblasti programovatelného stínování. Mnoho použitých technologií a softwarových balíků je ještě ve vývoji. To dává implementaci velký potenciál do budoucna.
- Implementace pomocí Shader Model 3.0 patří mezi první svého druhu, přitom její přínos je značný. Umožňuje totiž celou metodu implementovat pomocí jednoho průchodu, což bylo dříve nemyslitelné. Obdobný přístup byl prezentován až těsně před dokončením této práce na konferenci Volume Graphics v červnu 2005.
- Implementovaná metoda je v rámci dostupných možností otestována a jsou prezentovány její výsledky.

6.1 Směry další práce

Velmi podstatným rysem této práce je, že po dokončení vyvolala stejně otázek, jako poskytla odpovědí. Ukázalo se, že možnosti platformy konzumních grafických systémů značně pokročily a skutečně se otevírají možnosti pro reálně použitelnou implementaci zobrazování objemu.

Implementace prezentovaná v této práci je experimentální a při jejím zahájením nebylo zcela jasné, zda se jí podaří dovést do funkčního stavu. Během prací bylo zjištěno mnoho optimalizací a nových postupů, které by bylo možné použít k dalšímu zlepšení metody. Některé z nich nebyly do implementace zahrnuty, neboť preference byla na straně přehlednosti a správnosti metody, spíše než na její rychlosti a efektivitě. Jiné nebylo možné implementovat čistě z časových důvodů.

Některé z dalších možností vývoje této metody lze shrnout do následujících bodů:

- Způsob generování „stínovacích“ paprsků (tedy způsob výpočtu útlumu světla na dráze od vnějšího zdroje) je velmi neefektivní. V literatuře jsou navrženy způsoby, jak tento výpočet provádět efektivněji ([Kajiya a Von Herzen 84]), ovšem způsob jejich implementace uvnitř shaderů není úplně zřejmý a je potřeba dalšího výzkumu možností GPU.

- Rozdělení zatížení vertex a fragment shaderů je velmi nerovnoměrné. Převedení většího dílu práce na vertex procesory by určitě přineslo významné zvýšení výkonu. Způsob, jak by se tato distribuce dala provést byl nastíněn v §5.4.1.
- V implementaci není použitá žádná z urychlovacích technik, které se v jiných implementacích vrhání či rekurzivního sledování paprsku používají. V některých případech lze navíc již nyní říci, že jejich implementace nebude nijak obtížná a zvýšení rychlosti by mohlo být znatelné.
- Pro reálnou aplikovatelnost metody je potřeba zlepšit interakci s ostatními objekty ve scéně. Metoda sice bez problému koexistuje s ostatními tělesy ve scéně, ale pouze pokud mají tyto neprázdný průnik se zobrazovaným objemem. Tento nedostatek je závažný především pro použití v herním odvětví. U fenoménů jako je například oheň a kouř je zcela nezbytné, aby se proxy geometrie objemu mohla protínat s ostatními tělesy.
- V prezentované implementaci je použito pouze jedno světlo. Je čistě technickou záležitostí dodat světel více, ovšem přínos, zejména z hlediska kvality výstupu, by byl znatelný.

Literatura

[Almorea] *Domácí stránka projektu Almorea*, http://almorea.jikos.cz

[AnandTech] *Internetový portál o hardware AnandTech*, www.anandtech.com

[Blinn 82] J. F. Blinn: *Light reflection functions for simulation of clouds and dusty surfaces*, Proceedings of SIGGRAPH Conference (1982)

[Cabral et al. 94] B. Cabral, N. Cam, J. Foran: *Accelerated volume rendering and tomographic reconstruction using texture mapping hardware*, ACM Symposium on Volume Visualization (1994), 91-98

[Cameron a Undrill 92] G.G. Cameron, P.E. Undrill: *Rendering volumetric medical image data on SIMD-architecture computer*, Proceedings of the Third Eurographics Workshop on Rendering, (březen 1992), 135-145

[Chandrasekhar 50] S. Chandrasekhar: *Radiative Transfer*, Oxford University Press (1950)

[Cline et al. 88] H.E.Cline, W.E. Lorensen, S.Ludke, C.R. Craword, B.C. Teeter: *Two Algorithms for Three-dimensional Reconstruction of Tomograms*, Medical Physics Volume 15 Number 3, (březen/duben 1988)320-327

[Cullip 93] T. Cullip, U. Neumann: *Accelerating Volume Reconstruction with 3D texture hardware*, Technical Report TR93-027 University of North Carolina, Chapel Hill, N.C.

[DirectX9.0c 05] *Microsoft DirectX9.0c SDK*, April 2005 Release, msdn.microsoft.com/directx/

[Dunne et al. 90] S. Dunne, S. Napel, B. Rutt: *Fast Reprojection of Volume Data*, Proceedings of the First Conference on Visualization in Biochemical Computing (1990),11-18

[Ekoule et al. 91] A.B. Ekoule, F.C. Peyrin, C.L. Odet: *A triangulation algorithm from Arbitrary Shaped Multiple Planar Contours*, ACM Transactions of Graphics Volume 10 Number 2 (duben 1991), 182-199

[Elvins 92] T. Todd Elvins: *A survey of algorithms for volume visualization*, ACM SIGGRAPH Computer Graphics, Volume 26 Issue 3

[Engel et al. 01] K. Engel, M. Kraus, T. Ertl: *High-quality pre-integrated volume rendering using hardware-accelerated pixel shading*, SIGGRAPH/Eurographics Workshop on Graphics Hardware

[Friedlander 00] S.K.Friedlander: *Smoke, Dust and Haze: Fundamentals of Aerosol Dynamics*, Second Edition, Kapitola 5: Light Scattering, Oxford University Press 2000

[Fuchs et al. 77] H. Fuchs, Z. M. Kedem, S.P.Uselton: *Optimal Surface Reconstruction from Planar Contours*, Communications of the ACM Volume 20 Number 10, (Říjen 1977), 693-702

[Green 05] S. Green: *Volume Rendering for Games*, Game Developer Conference (2005), http://developer.nvidia.com

[Guthe et al. 02] S. Guthe, S. Roettger, A. Schieber, W. Strasser, T. Ertl: *High-quality unstructured volume rendering on the PC platform*, SIGGRAPH/Eurographics Hardware Workshop

[Henyey a Greenstein 40] G.L.Henyey, J.L. Greenstein: *Diffuse radiation in the galaxy*, Astrophysical Journal volume 88, 1940, 70-73

[Herman a Liu 79] G.T. Herman, H.K.Liu*: Three-dimensional Display of Human Organs from Computed Tomograms*, Computer Graphics and Image Processing, Volume 9 Number 1 (leden 1979), 1-21

[Kajiya 86] J. T. Kajiya: *The rendering equation*, Computer Graphics Volume 20 Number 4 (srpen 1986),143-150

[Kajiya a Von Herzen 84] J. T. Kajiya, B. P. Von Herzen: *Ray Tracing Volume Densities*, Proceedings of the SIGGRAPH Conference (1984)

[Kniss et al. 02a] J. Kniss, S. Premoze, C. Hansen, D. Ebert: *Interactive translucent volume rendering and procedural modeling*, Proceedings of IEEE Visualization (2002), stránky168-176

[Krüger a Westermann 03] J. Krüger, R.Westermann: *Acceleration techniques for GPU-based Volume Rendering***,** Proceedings of IEEE Visualization (2003)

[Lacroute a Levoy 94] P. Lacroute, M. Levoy: *Fast volume rendering using shear-warp factorization of the viewing matrix*, Proceedings of the 21st annual conference on Computer graphics and interactive techniques (1994),451-458

[Lastra et al. 95] A. Lastra, S. Molnar, Y. Wang: *Real-time programmable shading*, Proceedings of the symposium on Interactive 3D graphics (duben 1995)

[Levoy 90] Marc Levoy: *Volume Rendering: A hybrid Ray Tracer for Rendering Polygon and Volume Data*, IEEE Computer Graphics and Applications (březen 1990), 33-40

[Levoy 92] M. Levoy: *Volume Rendering using the Fourier Projection-Slice Theorem*, Proceedings of the Graphics Interface (1992), 61-69

[Lorensen a Cline 87] W.E. Lorensen, H.E. Cline: *Marching Cubes: A High Resolution 3D Surface Construction Algorithm*, Computer Graphics Volume 21 Number 4, (červenec 1987),163-169

[Magnor et al. 04] M. Magnor, G. Kindlmann, Ch. Hansen: *Constrained Inverse Volume Rendering for Planetary Nebulae*, Proceedings of the Conference on Visualization (2004)

[Malzbender 93] T. Malzbender: *Fourier Volume Rendering*, ACM Transactions on Graphics Volume 12 Number 3, (červenec 1993)

[Max 83] N. Max: *Panel on the simulation of natural phenomena*, Proceedings of SIGGRAPH Conference (1983)

[Max 95] N. Max: *Optical Models for Direct Volume Rendering*, IEEE Transaction on Visualization and Computer Graphics 1,2 (červen 1995), 99-108

[McCarthy et al. 04] J. D. McCarthy, M. A. Sasse, D. Miras: *Sharp or Smooth ?: comparing the effects of quantization vs. frame rate for streamed video*, Proceedings of the SIGCHI on Human factors in computing systems (duben 2004)

[Meehan et al. 02] M. Meehan, B. Insko, M. Whiton, F. P. Brooks: *Psychological measures of presence in stressful virtual environments*, ACM Transaction of Graphics, Proceedings of th 29th annual conference on Computer graphics and interactive techniques (červenec 2002)

[Meissner et al. 99] M. Meissner, U. Hoffman, W. Strasser: *Enabling classification and shading for 3d texture mapping based volume rendering using OpenGL and extensions*, Proceedings of IEEE Visualization (1999), stránky 110-119

[OpenGL 2.0 04*] The OpenGL Graphics System: A specification*, October 22, 2004, www.opengl.org

[Owen 99] G. S.Owen: *Volume Visualization and Rendering*, Výukové materiály ACM Education Committee, (modifikováno 1999), www.siggraph.org/education/materials/HyperVis/vistech/volume/volume.htm

[Perlin 02] K. Perlin, *Improving Noise*, Computer Graphics, volume 35, No. 3

[Perlin 84] K. Perlin, course in *Advanced Image Synthesis,* ACM SIGGRAPH (1984)
[Perlin] Domácí stránka Kena Perlina, http://mrl.nyu.edu/~perlin

[PG III] J. Pelikán, *Skripta k přednášce Počítačová grafika III,* cgg.ms.mff.cuni.cz/~pepca

[Phong 75] Bui-Tuong Phong, *Illumination for Computer-Generated Pictures,* CACM (červen 1975), 311-317

[Purcell et al. 02] T. J. Purcell, I. Buck, W.R. Mark, P. Hanrahan: *Ray tracing on programmable graphics hardware*, Proceedings of the 29th annual conference on Computer graphics and interactive techniques (červenec 2002), 703-712

[Rezk-Salama et al. 00] C. Rezk-Salama, K. Engel, M. Bauer, G. Geiner, T. Ertl: *Interactive Volume rendering on standard PC graphics hardware using multi-textures and multi-stage rasterization,* SIGGRAPH/Eurographics Workshop on Graphics Hardware, stránky 109-119

[Sabella 88] P. Sabella: *A Rendering Algorithm for Visualizing 3D Scalar Fields,* Computer Graphics Volume 22 Number 4 (červenec 1988), 51-58

[Shirley a Tuchman 90] P. Shirley, A. Tuchman: *A polygonal Approximation to Direct Scalar Volume Rendering*, Computer Graphics Volume 24 Number 5 (listopad 1990), 63-70

[Stegmaier et al. 05] S. Stegmaier, M.Strengert, T.Klein, T.Ertl, *A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-based Raycasting*, Volume Graphics (2005)

[Totsuka a Levoy 93] T.Totsuka, M. Levoy: *Frequency domain volume rendering*, Proceedings of the 20th annual conference on Computer graphics and interactive techniques, 271-278

[Upson a Keeler 88] C. Upson, M. Keeler: *The V-Buffer: Visible Volume Rendering*, Computer Graphics Volume 22 Number 4 (červen 1988), 59-64

[Van Geldern a Kwansik 96] A. Van Geldern, K. Kwansik: *Direct Volume Rendering with Shading via Three-Dimensional Textures,* Proceedings of ACM Symposium on Volume Visualization (1994), stránky 23-30
[Voss 83] R.Voss: *Fourier synthesis of Gaussian fractals: 1/f noises, landscapes and flakes,* Tutorial on the State of the Art Images Synthesis, volume 10, SIGGRAPH (1983)

[Westermann a Ertl 98] R. Westermann, T. Ertl: *Efficiently using graphics hardware in volume rendering applications*, Proceedings of SIGGRAPH (1998), stránky 291-294.

[Westover 90] L. Westover: *Footprint Evaluation For Volume Rendering*, Computer Graphics Volume 24 Number 4, (srpen 1990), 267-376

[Yuan et al. 97] P.Yuan, M.Green, R.W.H.Lau: *A framework for performance evaluation of real-time rendering algorithms in virtual reality*, Proceedings of the ACM Symposium on Virtual Reality software and technology (září 1997)

Příloha A: Obsah CD

Doprovodné CD je vybaveno systémem automatického spouštění. Po vložení CD do mechaniky se spustí webová stránka *index.htm*. Pokud je automatické spouštění v operačním systému vypnuto, lze stránku *index.htm* nalézt přímo v kořenovém adresáři CD. Na této stránce je uveden obsah CD spolu s odkazy na jednotlivé adresáře.

CD má následující obsah:

- **Adresář DataPreparation**. V tomto adresáři je uložen pomocný program pro tvorbu experimentálních dat. Program je ve formě úplného projektu (solution) pro prostředí Microsoft Visual Studio. NET 2003.
- **Adresář Effects**. Tento adresář obsahuje všechny vytvořené efektové soubory. Jsou zde také uloženy testovací scény (/Effects/Scenes), experimentální objemová data (/Effects/Volumes) a pomocné bitmapy (/Effects/Images).
- **Adresář Text**. Zde je uložen text této diplomové práce ve formátu PDF. Přiložen je také zdrojový soubor ve formátu Microsoft Word.