České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů

Diplomová práce

Pohledově závislé stínové mapy

Bc. Jan Hink

Vedoucí práce: Ing. David Ambrož

Studijní program: Elektrotechnika a informatika, strukturovaný, Navazující magisterský

Obor: Výpočetní technika

Leden 2010

Poděkování

Děkuji svému vedoucímu za jeho iniciativu, ochotu pomoci a také za jeho nekonečnou trpělivost. Dále děkuji svým nejbližším za podporu v dobách psaní této práce.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 4. ledna 2010 .....

Abstrakt

Stíny jsou nedílnou součástí vnímání světa kolem nás, jelikož nám poskytují informaci o vzájemné poloze těles a jejich velikosti v trojrozměrném prostoru. Jsou tedy nepostradatelné i v počítačové grafice při vykreslování realisticky vypadajících scén. Přesný výpočet stínů v reálném čase je však časově velmi náročná operace, a proto se v praxi používají alternativní metody, které kompenzují tuto náročnost určitým snížením vizuální kvality.

Mezi nejčastěji používanou metodu patří v dnešní době metoda stínových map, jejíž rychlost a jednoduchost je však vykoupena řadou nedostatků. Patrně největším z nich je závislost na omezené rozlišovací schopnosti grafických karet, způsobující diskretizační chyby. Ty se projevují nepřesnostmi na hranicích generovaných stínů, známými pod pojmem aliasing. Tato práce prezentuje algoritmus, který využívá původní metodu stínových map, a který díky možnostem dnešních grafických karet eliminuje tuto omezující závislost a generuje přesné stíny pro jakékoliv rozlišení.

Abstract

Shadows play an important role in human perception of the world because they provide us with the information about relative position of objects and their size in three dimensional space. Therefore they are indespensable in rendering realistic images in computer graphics. Precise computation of the shadows in real time is, however, a very expensive operation. In order to compensate for this, several alternative approaches were developed at cost of certain reduction of visual quality.

Among these approaches, the probably most frequently used method is called shadow mapping. However, its speed and simplicity is paid back by several drawbacks. Probably the most significant one is dependency on limited image resolution of graphic cards which causes discretization errors. These errors lead to inaccurate rendering of the shadow boundaries which is known as aliasing. This paper presents an algorithm which uses the original shadow mapping method, and by utilizing the capabilities of modern graphic cards removes the limiting dependency and renders correct shadows in any resolution.

Obsah

Kapitola 1

Úvod

Stíny jsou nepostradatelné pro správné vnímání našeho okolí, jelikož nám poskytují důležité informace o prostorovém uspořádání předmětů kolem nás. Pomáhají nám orientovat se v prostoru a určovat tvar, velikost a vzdálenost věcí, které vidíme. Zároveň mnoho vypovídají i o charakteru světelných zdrojů, jejich poloze či intenzitě.

Nejinak je tomu i v počítačové grafice, kde je kladen stále větší důraz na co nejvyšší vizuální kvalitu. V dnešní době již není téměř možné se setkat například s počítačovou hrou, ve které by stíny nebyly, nemluvě o animovaných filmech, které by bez stínů dozajista nemohly slavit žádný komerční úspěch. Stíny přidávají na realističnosti a kvalitě jakéhokoliv počítačem vytvořeného obrazu a bez jejich přítomnosti působí obraz velmi nepřirozeně a uměle.

Věrné stíny například v počítačových hrách tak, jak je známe dnes, nebyly v dobách prvotního rozvoje počítačové grafiky žádnou samozřejmostí a stíny se kvůli své výpočetní náročnosti buď vůbec nepoužívaly, nebo se používaly různé náhrady, které však měly k věrným a realistickým stínům velmi daleko – vzpomeňme například na často využívané statické pseudo-stíny v podobě tmavých oválů pod předměty či postavami. S překotným rozvojem možností a výkonnosti grafických karet, který se datuje prakticky od poloviny devadesátých let minulého století, je však v dnešní době poměrně snadné obohatit uměle vytvořenou scénu o stíny v reálném čase. Nalézt však kompromis mezi dostačující vizuální kvalitou a výpočetní náročností je přesto stále jedna z největších výzev, se kterou se počítačová grafika odjakživa potýká.

Jednou z metod, které se dnes pro přidání stínů do scény v reálném čase nejčastěji používají, je metoda stínových map. Její největší předností je nezávislost na složitosti geometrie scény, protože pracuje v obrazovém prostoru. Tato výhoda, spolu z ní plynoucí rychlostí a snadnou implementací, je však vykoupena řadou nedostatků, které z práce s diskrétními obrazovými daty plynou.

Pravděpodobně nejvíce patrnými z nich jsou takzvaný *aliasing*, čili nepřesnosti na hranicích stínů, které vznikají v důsledku omezené rozlišovací schopnosti grafických karet, a potom takzvaný *self-shadowing*, který je zase způsoben numerickými nepřesnostmi při porovnávání hodnot v plovoucí řádové čárce. Ten se projevuje chybným zastíněním povrchů, které mají být osvětlené. Mezi další nevýhody potom patří také závislost na typu a orientaci světla ve scéně – problematické je zejména generování stínů pro všesměrové světelné zdroje.

Existuje sice řada metod, které výše zmiňované problémy zmírňují, ale s jejich samotnou podstatou se žádná z nich nevypořádává. Dnešní grafické karty však nabízejí možnosti, díky kterým by mělo být možné se těmto problémům vyhnout úplně. Tato práce si tedy klade za cíl po prozkoumání těchto možností navrhnout a vytvořit algoritmus, který nebude výše zmíněnými problémy trpět a bude produkovat přesné stíny v jakémkoliv rozlišení.

Samozřejmě tím budeme muset obejít podstatu metody stínových map, ze které její problémy plynou, ale která jí také zároveň propůjčuje onu vysokou rychlost – tedy obrazový prostor. Algoritmus, který tato práce prezentuje, tedy nebude schopen dosahovat takové rychlosti, neboť bude narozdíl od metody stínových map závislý na složitosti geometrie scény. Obrazového prostoru se však úplně nevzdáme, neboť informace z něj získatelné jsou cenné a mohou být využity při optimalizacích našeho algoritmu. Právě optimalizace budou hlavní náplní této práce a jejich cílem bude zvýšit rychlost našeho algoritmu pokud možno tak, aby byl použitelný v reálném či interaktivním čase pro scény obsahující řádově desetitisíce polygonů.

Ač se bude tato práce věnovat zejména vlastnímu algoritmu pro výpočet stínů, tak princip, na kterém bude tento algoritmus pracovat, má dopad širší. Abychom nastínili, o co se bude jednat, vezměme si obecně grafické karty; víme, že umějí počítat osvětlení, míchat barvy, texturovat objekty a provádět další podobné operace. To všechno však dokáží dělat pouze pro pevně danou množinu bodů, kterými jsou středy pixelů obrazovky, jelikož „alfou a omegou“ zobrazování dnešních grafických karet je proces, kterému říkáme rasterizace. Prostřednictvím našeho algoritmu však ukážeme, jak lze využít dnešní grafické akcelerátory k tomu, abychom toto omezení obešli. Svým způsobem totiž řekneme grafické kartě, nad kterou množinou bodů má tyto výpočty provádět, a tím ji vlastně donutíme vzorkovat nikoliv ve středech pixelů, ale v libovolných bodech. V našem případě budeme nad množinou takto vybraných bodů provádět výpočty zastínění, ale princip, na kterém námi prezentovaný algoritmus pracuje, má obecně mnohem širší použití.

Po krátkém úvedení do problematiky se tedy s tímto principem seznámíme a s jeho využitím implementujeme algoritmus počítající přesné stíny pro jakékoliv rozlišení, který bude ke svým optimalizacím mimo jiné využívat i původní metodu stínových map. K implementaci bude použito grafické rozhraní OpenGL verze 3.0+ a grafická karta GeForce osmé řady od společnosti *NVidia*, která nabízí právě potřebné možnosti.

Kapitola 2

Stíny v počítačové grafice

2.1 Význam stínů a jejich přínos pro vizuální kvalitu obrazu

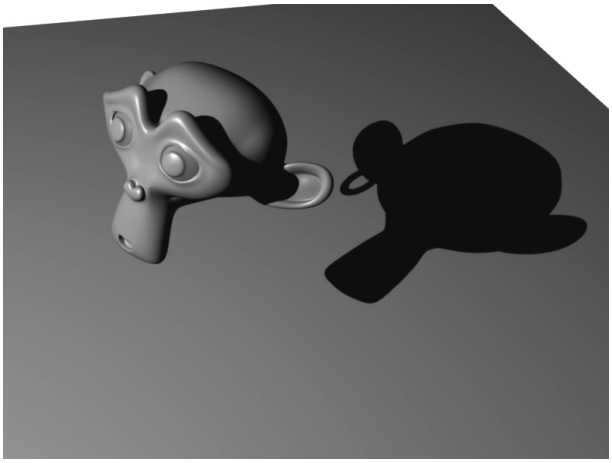
Stíny hrají velmi důležitou roli při orientaci člověka v prostoru. Při každodenní činnosti nám jejich přítomnost přijde samozřejmá a velmi málo si uvědomujeme, kolik informací o našem okolí nám poskytují. Jako je světlo důležité pro vnímání barev předmětů kolem nás, jsou stíny nepostradatelné při vnímání jejich tvaru, povrchu, rozměrů a vzájemné polohy v trojrozměrném prostoru. Navíc nám poskytují informaci o počtu, poloze a vlastnostech světelných zdrojů, i když jsou mimo naše zorné pole.

Intuitivně stín chápeme jako oblast, kam nedopadají světelné paprsky z důvodu jejich (částečného) pohlcení stínícím předmětem. Tvar a velikost stínu jsou dány vzájemnou polohou světelného zdroje a stínícího objektu, stejně jako polohou a tvarem objektu, na který stín dopadá. Dle charakteru a velikosti světelného zdroje pak rozeznáváme jednotlivé typy stínů.

Stíny nemusí být nutně tmavé oblasti bez světla. Pokud máme například ve scéně více než jeden zdroj světla, nemusí být stín nutně tmavý, pokud je zastíněn jen vzhledem k jednomu zdroji. Dalším případem může být stín vznikající zastíněním oblasti (polo)průhledným tělesem, kdy nemusí být stín vůbec vidět či mohou vznikat stíny různých barev. Z přirozeného významu slova však budeme stínem rozumět oblast za předmětem, který leží na cestě mezi světlem a touto oblastí, ať je tato oblast zatemněná či nikoliv.

Pro ilustraci důležitosti stínů se podívejme například na obrázek 2.1. Těleso je zde natočeno ke kameře způsobem, který znemožnuje rozpoznání jeho celkového tvaru a bez přítomnosti stínu nemá pozorovatel šanci tento tvar určit.

V počítačové grafice jsou stíny o to důležitější, jelikož zobrazovací zařízení — počítačová obrazovka — umožňuje zobrazování pouze dvourozměrných dat. Beze stínů je poloha těles ve dvou rozměrech dána nejednoznačně. Při pohledu na obrázek 2.2(a) není jasné, kde se



Obrázek 2.1: Bez stínu bychom nebyli schopni určit přesný tvar tělesa

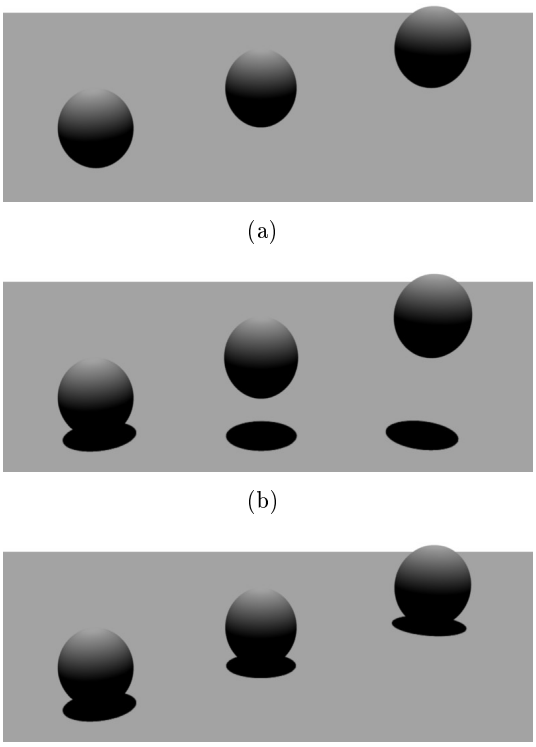
v trojrozměrném prostoru zobrazené koule nacházejí. Nejsme schopni jednoznačně určit, zda jsou všechny stejně veliké a ve stejné hloubce, či se velikostně liší a jsou umístěny v různé hloubce. Pokud však ke dvojrozměrnému obrázku přidáme stín, jak je naznačeno na obrázcích 2.2(b) a 2.2(c), je snadné jejich pozici v prostoru určit. Různě vržený stín nám dokáže napovědět polohu těles v prostoru, aniž by se jakkoliv změnila geometrie scény. Více o důležitosti stínů a jejich dopadu na kvalitu vnímání se lze dozvědět například v [Wan92, MKK98]

2.2 Druhy stínů z pohledu počítačové grafiky

2.2.1 Ostré a měkké stíny

Dle tvaru a velikosti světelného zdroje rozlišujeme dva druhy stínů — takzvané ostré stíny (*hard shadows*) a měkké stíny (*soft shadows*). Ostré stíny vznikají při osvětlení scény bodovým světelným zdrojem a jak vyplývá z jejich názvu, mají ostrou, přesně vymezenou hranici. Přítomnost ostrého stínu ve scéně je pro každý její bod dána binární informací podle toho, zda je z tohoto bodu světelný zdroj vidět (světlo = 1), či je zastíněn jiným předmětem (stín = 0). Výsledná barva bodu je pak dána původní barvou objektu ve scéně vynásobenou touto binární informací. Jak vznikají ostré stíny ilustruje obrázek 2.3 a ukázku takového stínu můžeme pozorovat na obrázku 2.4(a). My se v kapitole 5 blíže seznámíme s metodou stínových map a metodou stínových těles, které se používají pro generování ostrých stínů v reálném čase. Algoritmus pohledově závislých stínových map, prezentovaný v této práci, se pak také zabývá generováním tohoto druhu stínů.

Ostré stíny jsou však pouze zjednodušením reálných stínů, jelikož ideální bodové světelné

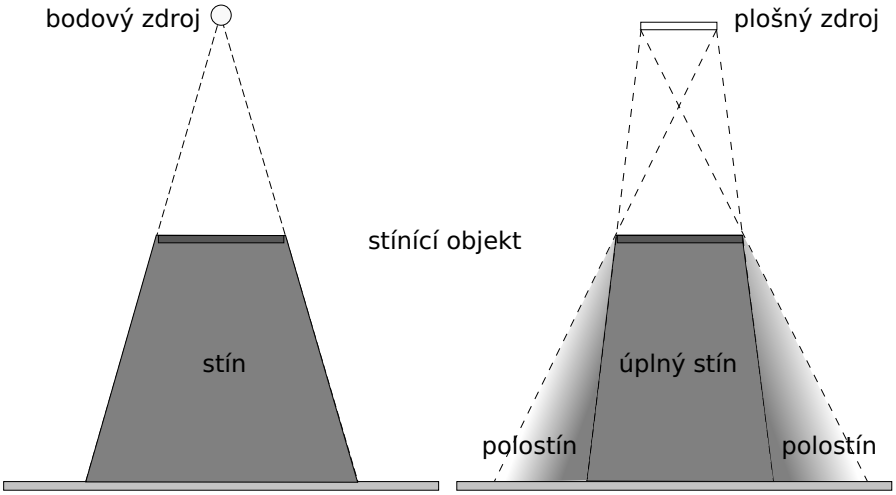


(a)

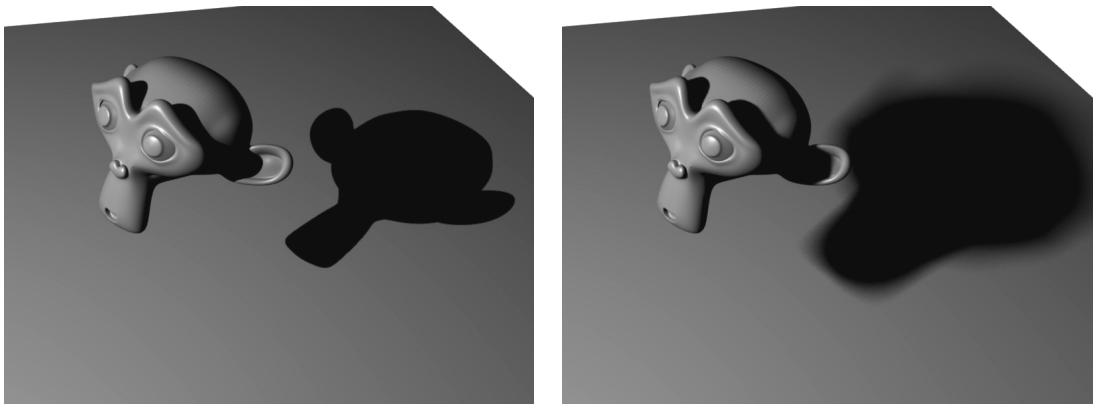
(b)

(c)

Obrázek 2.2: Vliv stínu na vnímání pozice těles



Obrázek 2.3: Ostrý stín pocházející z bodového zdroje světla (vlevo) a měkký stín (úplný stín s polostínem) pocházející z plošného zdroje světla (vpravo)

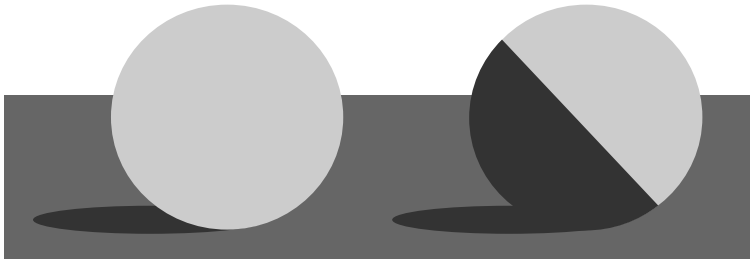


(a) (b)

Obrázek 2.4: Ostrý stín (a) vržený bodovým světelným zdrojem a měkký stín (b) vržený plošným světelným zdrojem

zdroje se ve skutečném světě nevyskytují. Proto tyto stíny působí nerealistickým a nepřirozeným dojmem. Naproti tomu měkké stíny působí mnohem přirozeněji, jelikož pocházejí z plošných či objemových světelných zdrojů, které se v různých podobách vyskytují všude kolem nás. Tyto stíny jsou charakteristické tím, že se skládají jak z úplného stínu (*umbra*), tak z takzvaného polostínu, čili oblasti zastíněné pouze částečně (*penumbra*). Úplný stín vzniká tam, kde z daného bodu není vidět žádná část světelného zdroje. Polostín začíná na hranici úplného stínu a jeho intenzita postupně klesá směrem k okraji stínu, až může úplně vymizet. Intenzita stínu v daném bodě pak může být dána jak velikostí plochy světelného zdroje viditelné z tohoto bodu, tak i jeho vyzařovacími charakteristikami. Algoritmy pro výpočet měkkých stínů pracující v reálném čase většinou situaci zjednodušují a tyto charakteristiky zanedbávají – předpokládají tedy například konstantní intenzitu záření po celé ploše (či v celém objemu) světelného zdroje. Pokud ale chceme počítat přesné měkké stíny, musíme brát v potaz i tyto charakteristiky. Vznik měkkého stínu je ilustrován na obrázku 2.3 a ukázku takto generovaného stínu můžeme pozorovat na obrázku 2.4(b).

Vzhledem k výše popsaným skutečnostem by se mohlo zdát, že měkké stíny jsou ve všech směrech o mnoho lepší než stíny ostré, a tedy nemá smysl se generováním ostrých stínů vůbec zabývat. Toto je však třeba uvést na pravou míru. Ač jsou měkké stíny vizuálně přitažlivější a přidávají na realističnosti obrazu, jejich přesný výpočet je i v dnešní době stále velmi náročný. Metody pro generování měkkých stínů se pak často zakládají na algoritmech výpočtu stínů ostrých, nebo je přímo využívají.



Obrázek 2.5: Příklad jednoduché scény bez vlastního stínu (vlevo) a s vlastním stínem (vpravo)

2.2.2 Vlastní a vržené stíny

Další klasifikace stínů z pohledu počítačové grafiky zahrnuje stíny vlastní (*self shadows*) a stíny vržené (*cast shadows*). Ve vlastním stínu leží všechny plochy tělesa, které jsou odvrácené od světla a dále ty plochy, které jsou zastíněné jinými částmi tohoto tělesa. Určení stínu na plochách odvrácených od světla probíhá již ve fázi stínování, a proto ho často nevnímáme jako stín. Příklad jednoduchého vlastního stínu můžeme pozorovat na obrázku 2.5. Za zmínku ještě stojí, že ne každý stínový algoritmus je schopen generovat i vlastní stíny.

Vržené stíny jsou takové stíny, kdy jedno těleso vrhá stín na druhé a často jsou pro vnímání scény jako celku důležitější, jelikož obsahují informace o vzájemné poloze více těles a jejich velikosti. Na obrázcích 2.4(a), 2.4(b) a 2.5 se jedná o stíny vržené tělesy na plochu pod nimi. Tyto stíny nám dokáží vypovědět také mnoho o zdroji světla, o jeho pozici, vzdálenosti či intenzitě.

Tato práce se tedy bude zabývat algoritmem generování ostrých stínů, podporujícím i stíny vlastní. Celkové zaměření této práce však není takto jednostranné – prostřednictvím našeho algoritmu totiž budeme demonstrovat řešení obecnějšího problému, a sice řešení viditelnosti pomocí metody vzorkování na grafické kartě.

Kapitola 3

Stručná historie 3D akcelerované grafiky

Vznik prvního 3D grafického akcelerátoru pro osobní počítač se datuje do druhé poloviny devadesátých let minulého století. Počítačem generovaná trojrozměrná grafika sice již na osobních počítačích existovala, ale operace spojené s jejím vykreslováním se prováděly na hlavním procesoru (CPU) a ukrajovaly tak drahocenný výpočetní čas operacím, které neměly s grafikou nic společného. Z tohoto důvodu nebylo možné vykreslovat v reálném čase líbivou 3D grafiku s efekty, jaké dnes považujeme za samozřejmost – to bylo do té doby výsadou pouze vysoce specializovaných a velmi drahých grafických stanic, i když samozřejmě v omezené míře. Mezi nimi vynikaly zejména stroje společnosti *Silicon Graphics (SGI)*, která v té době dominovala na trhu s grafickým hardware.

Díky počínajícímu zájmu širší veřejnosti o virtuální realitu, zejména pak hráčů počítačových her, si společnosti produkující hardware začaly uvědomovat, že budoucnost počítačové grafiky neleží v rukou grafických studií, která si mohla výkonný grafický hardware dovolit, ale v rukou majitelů osobních počítačů. Díky tomu začalo v roce 1995 svítat na lepší časy i jim, jelikož byl na trh uveden jeden z prvních 3D grafických čipů zaměřených na běžné uživatele – ViRGE S3. Ten již hardwarově podporoval bilineární a trilineární filtrování textur, *mipmapping*[1], míchání barev (*alpha blending*), výpočet viditelnosti (*z-buffering*) a různé další efekty. Tato funkční vybavenost se mu však stala i osudnou, jelikož trpěl velmi nízkou výkonností a v některých případech pracoval i pomaleji, než samotné CPU. V této době byly na trh uvedeny také například grafické karty ATI 3D Rage, Matrox Mystique a Rendition Verité V1000, a společně tak zahájily překotný rozvoj grafických akcelerátorů, který pokračuje dodnes.

[1] *Mipmapping* je metoda používaná při mapování textur ve 3D grafice, která spočívá v tom, že z původní textury v plném rozlišení vytvoříme sadu jejích verzí v rozlišeních nižších. Při mapování textury na povrch je

O opravdový „boom“ v oblasti 3D grafiky se však postarala společnost 3dfx Interactive, založená bývalými zaměstnanci *SGI*, když v roce 1996 vytvořila čip pro grafickou akceleraci zvaný Voodoo Graphics, později označovaný jako Voodoo 1. Čip zvládal perspektivně korektní mapování textur za pomocí jedné texturovací jednotky, dále hardwarový *anti-aliasing*, *triple-buffering* a prakticky vše, co uměly i konkurenční grafické karty. Výkonově však nechával veškerou tehdejší konkurenci daleko za sebou, a díky tomu se stal senzací mezi hráči počítačových her, a to i přesto, že trpěl několika nedostatky. Prvním z nich bylo, že byl zaměřen čistě na 3D grafiku a neobsahoval žádnou podporu pro 2D, a tedy vyžadoval přítomnost druhé grafické karty, což prodražovalo jeho případnou koupi. Tato VGA karta byla propojena s kartou Voodoo, která byla přímo spojena s monitorem. Další nevýhoda spočívala v tom, že komunikace s čipem byla zajišťována pomocí proprietárního API Glide3D a aplikace, které chtěly využívat jeho možnosti grafické akcelerace, musely podporovat toto rozhraní. Pokud však měly fungovat i na jiných kartách, které komunikovaly hlavně pomocí API Direct3D, musely podporovat zároveň obě rozhraní, což samozřejmě prodražovalo vývoj. Podotkněme jen, že OpenGL nebylo tehdy mezi výrobci rozšířené.

Téměř monopolní postavení chipu *Voodoo* na trhu s grafickými akcelerátory bylo ohroženo až v roce 1997, kdy byla na trh uvedena grafická karta Riva 128 společnosti *NVidia*. Tato karta narozdíl od karty Voodoo 1 integrovala funkcionalitu pro vykreslování 2D a 3D, a navíc podporovala tehdy novou a rychlejší sběrnici AGP (*Accelerated Graphics Port*). Ve srovnání s kartou od společnosti *3dfx* však disponovala horší kvalitou obrazu a mnoho vývojářů 3D aplikací dávalo stále přednost čipu Voodoo a jeho proprietárnímu API Glide3D.

Aby společnost *3dfx* ulevila zákazníkům od nutnosti zakoupit ke svému produktu i 2D kartu, vyvinula čip Voodoo Rush, který její funkcionalitu za cenu sníženého výkonu obsahoval. Následně společnost ještě utvrdila svoji neohroženou pozici uvolněním čipu Voodoo 2, který se sice parametry příliš nelišil od své první verze, ale umožňoval propojit dvě stejné karty v módu SLI (*Scan Line Interleave*). Takto zapojené karty fungovaly tak, že každá vykreslovala polovinu řádků obrazovky, čímž mohly dosáhnout až dvojnásobného výkonu. Tento koncept později využila i řada budoucích karet. Posledním počinem *3dfx*, který zřejmě ještě držel společnost na výsluní, byl čip s přízviskem Banshee, který již měl první plnohodnotnou a výkonnou podporu 2D zobrazování v historii této řady. Z konkurence stojí za zmínku karta Riva TNT od *NVidie*.

S příchodem Voodoo 3 však již *3dfx* nestačí držet krok se stále úspěšnější společností *NVidia* a jejím novým grafickým akcelerátorem Riva TNT2, který podporoval 32-bitové zobrazení barev a který slavil velký komerční úspěch. Zatímco se společnost *3dfx* snažila vyrovnat s touto porážkou, dostala od *NVidie* finální zásah v podobě vydání revoluční karty

---

potom v závislosti na vzdálenosti texturované plochy od pozorovatele vybrána odpovídající mipmapa, čímž lze urychlit proces vykreslování a zároveň i zvýšit vizuální kvalitu.

GeForce 256 v roce 1999, která se stala pravděpodobně jedním z nejdůležitějších milníků v historii grafických akcelerátorů.

GeForce 256 totiž jako první nabídla možnost nechat hardwarově spočítat část zobrazovacího řetězce týkající se transformací a osvětlení, přestože hry, které toho byly schopné využít, přišly až o rok později. Tato schopnost se označuje zkratkou HW T&L, neboli *Hardware Transform and Lighting*. Jako první také obsahovala kompletní hardwarovou podporu fixní grafické pipeline (viz kapitola 4.2.2) podle standardu OpenGL. Byla to také tato grafická karta, díky které se začaly všechny následující grafické čipy označovat zkratkou GPU (*Graphics Processing Unit*).

Následovala druhá řada čipů GeForce a poté třetí, která na počátku roku 2001 realizovala koncept programovatelné grafické pipeline (viz kapitola 4.2.3), který se stále vyvíjí a používá dodnes. V současnosti mají za sebou grafické čipy GeForce již více než devět generací.

Pro ilustraci pokroku ve vývoji grafických akcelerátorů za posledních čtrnáct let proveďme srovnání například tehdejšího špičkového grafického akcelerátoru Voodoo 1[2] z roku 1996 a jedné z dnešních nejvýkonnějších grafických karet od *NVidie*, kterou je GeForce 9800 GTX+[3]. Grafický čip Voodo 1 měl 4 až 6 MB paměti a pracoval na frekvenci jádra 50 MHz. Jeho výkon měřený v počtu texelů vykreslených za vteřinu (*texture fill rate*) se pohyboval okolo 50 milionů při všech zapnutých efektech. O čtrnáct let později má GeForce 9800 GTX+ 512 MB paměti, frekvence grafického procesoru je 738 MHz a jeho výkon se pohybuje kolem 47 miliard pixelů za vteřinu, přičemž samozřejmě dokáže spočítat mnohem více grafických efektů. Jen z hlediska *fill rate* tedy můžeme pozorovat téměř tisíci násobné zrychlení.

[2]Technické parametry grafického čipu Voodoo převzaty z `www.accelenation.com/?ac.id.123.1`
[3]Technické údaje grafického čipu GeForce převzaty z `www.nvidia.com/object/geforce_family.html`

Kapitola 4

Úvod do programování dnešních GPU

4.1 Grafické rozhraní OpenGL

4.1.1 Stručná historie

V době rozvoje počítačové grafiky v osmdesátých letech minulého století neexistoval žádný jednotný způsob komunikace s grafickým hardware. Pro každý nový model grafického čipu se tedy musel programovat specifický ovladač a k němu komunikační rozhraní. Aplikace, které potom měly za úkol pracovat s různými grafickými zařízeními, pak musely všechna tato rozhraní podporovat, což bylo velice neefektivní.

Na počátku devadesátých let bylo proto vyvinuto a jako otevřený standard používáno aplikační rozhraní (API) PHIGS (*Programmer's Hierarchical Interactive Graphics System*), které však rychle ustoupilo dalšímu, vyspělejšímu, avšak proprietárnímu rozhraní IRIS GL (*Integrated Raster Imaging System Graphics Library*), vyvinutému společností *Silicon Graphics (SGI)*, která v té době dominovala na poli producentů grafického hardware.

V této době se však začaly objevovat i další společnosti schopné vyrábět zařízení pro vykreslování grafiky, a tudíž vznikla potřeba vyvinout jednotné, otevřené a standardizované rozhraní, které odstraní nutnost přizpůsobovat každou aplikaci všem modelům dostupným na trhu. Kandidátem bylo samozřejmě API IRIS GL, které tehdy představovalo špičku technologického pokroku, ale kvůli problémům s licencemi a nedostatečně formálně definovanou specifikací ho nebylo možné použít. Navíc toto rozhraní obsahovalo řadu funkcí, které s vykreslováním počítačové grafiky přímo nesouvisely, jako například správa oken nebo zpracování vstupu z periferních zařízení. Bylo tedy třeba navrhnout rozhraní nové.

V roce 1992 tak byla dokončena specifikace první verze OpenGL (*Open Graphics Library*)[1], o níž se zasloužili pánové *Mark Segal* a *Kurt Akeley*, a která se dodnes používá jako

[1] Oficiální stránky knihovny OpenGL jsou na adrese `www.opengl.org`

otevřený standard pro komunikaci s grafickými kartami. Současně byla pod názvem *OpenGL Architecture Review Board* (ARB)[2] ustavena skupina společností, které měly za úkol definovat kvalitativní testy nových specifikací, tyto poté schvalovat, a tím pečovat o kvalitu standardu. V roce 2006 tuto úlohu převzalo neziskové konsorcium *Khronos Group*[3].

4.1.2 Základní charakteristika OpenGL

OpenGL je standardizované, multiplatformní, hardwarově a jazykově nezávislé nízkoúrovňové (*low-level*) grafické rozhraní pro vytváření interaktivních aplikací využívajících dvou- a trojrozměrnou grafiku. Aby bylo možno těchto vlastností dosáhnout, neobsahuje žádné platformově specifické příkazy pro práci s okny či se vstupními zařízeními. Místo toho OpenGL specifikuje, jak má vypadat grafický zobrazovací řetězec (*graphic pipeline*, více v kapitole 4.2) a definuje množství funkcí pro komunikaci mezi aplikací a grafickým hardware. Samotné API je implementováno jako relativně složitý stavový automat[4] a většina funkcí, které uživateli nabízí, slouží k posílání grafických primitiv do zobrazovacího řetězce a k ovlivňování jejich zpracování.

Díky tomu, že je OpenGL nízkoúrovňové grafické rozhraní (*low-level API*), je nad ním možné vystavět sofistikované knihovny, které mohou například popisovat celé trojrozměrné modely. Jednou z knihoven, která poskytuje v určité míře tuto funkcionalitu, je GLU (*OpenGL Utility Library*), jež je přímo součástí OpenGL. Pro správu oken či vstupních zařízení však musí uživatel použít některou z externích knihoven, jako například multiplatformní Glut (*OpenGL Utility Toolkit*) či SDL (*Simple DirectMedia Layer*). Druhá ze jmenovaných byla použita v implementační části této práce.

Vývoj OpenGL je spíše než kompletním přepisováním jádra zajištěn systémem revizí, kdy jsou do jádra postupně přidávána takzvaná rozšíření, která pocházejí od výrobců grafických karet a spočívají ve zpřístupnění nových možností GPU přidáním nových metod a symbolů k API. Po dosažení určité kvality, jejíž měřítko je posuzováno konsorciem ARB, jsou tato rozšíření zapracována přímo do následující verze OpenGL.

Od svého prvního vydání na počátku devadesátých let pokročilo OpenGL do verze 3.2, a za tu dobu urazilo společně s vývojem GPU dlouhou cestu. V počáteční verzi se OpenGL v některých aspektech podobalo IRIS GL, ale například ještě neobsahovalo podporu texturovacích objektů, která byla přidána až ve verzi 1.1. Za další milník ve vývoji OpenGL lze považovat verzi 2.0, kdy do něj byl zapracován jazyk GLSL (*OpenGL Shading Language*), a tím přibyla uživatelům významná možnost přeprogramovat chování transformační a stínovací fáze – vznikla tím tedy podpora programovatelného grafického zobrazovacího řetězce

[2] O uskupení ARB se lze dozvědět více na `www.opengl.org/about/arb/`

[3] Oficiální stránky *Khronos Group* jsou `www.khronos.org`

[4] Schéma stavového automatu OpenGL verze 1.1 lze nalézt na `www.opengl.org/documentation/specs/version1.1/state.pdf`

(viz kapitola 4.2.3). S verzí 3.0 byl zaveden takzvaný *depreciation model* a byla zavedena notace dopředné (*forward*) a zpětné (*backward*) kompatility. V režimu dopředné kompatibility zastarávají některé příkazy a jsou nahrazeny novými a efektivnějšími.

Nyní obsahuje OpenGL na 75 schválených rozšíření[5], která jsou již součástí samotného API (tato mají předponu ARB, například GL_ARB_depth_texture) a dále přes 380 často hardwarově specifických rozšíření výrobců grafických karet (například GL_EXT_framebuffer_object či GL_NV_depth_clamp).

4.2 Grafický zobrazovací řetězec OpenGL

4.2.1 Úvod

Převedení trojrozměrné scény dané geometrií těles na pixely zobrazené na dvourozměrné ploše monitoru je složitý proces zahrnující množství kroků, které je třeba postupně a v pevně daném pořadí vykonat. Jednotlivé fáze tohoto procesu se hromadně nazývají grafickým zobrazovacím řetězcem (*graphic pipeline*).

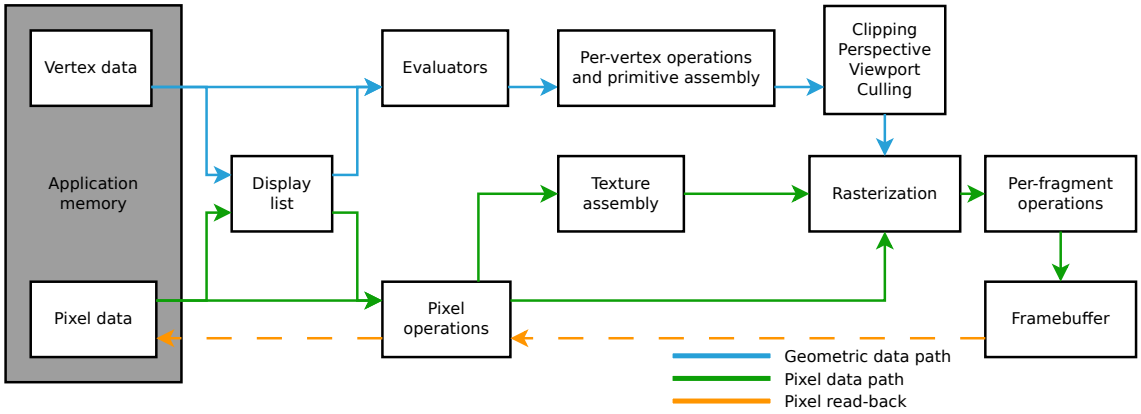
Před OpenGL verze 2.0 bylo možné využívat pouze grafický zobrazovací řetězec s takzvanou „pevnou funkčností“ (*fixed functionality pipeline*, více v kapitole 4.2.2), kdy bylo možné ovlivňovat průběh zpracování grafických primitiv v jednotlivých jeho částech pouze ve velmi omezené míře. Od této verze má však uživatel možnost přepsat toto limitující chování pomocí takzvaných *shaderů* (viz 4.2.3) s použitím jazyka GLSL (*OpenGL Shading Language*) a implementovat tak vlastní chování určitých částí. Zobrazovací řetězec, který toto umožňuje, se nazývá programovatelným (*programmable pipeline*) a dnes je již standardem v grafickém průmyslu. Blíže se s ním seznámíme v kapitole 4.2.3.

4.2.2 Fixní grafický zobrazovací řetězec

Zjednodušené, avšak pro účely tohoto úvodu dostačující schéma fixního grafického zobrazovacího řetězce OpenGL můžeme pozorovat na obrázku 4.1. Hned si můžeme všimnout, že schéma obsahuje dvě nezávislé cesty – cestu geometrických dat, kudy putují údaje o jednotlivých vrcholech, a cestu rastrových dat, kudy putují pixelová data – například textury či bitmapy. Tyto cesty jsou realizovány jednotlivými moduly, které jsou na sobě nezávislé a narozdíl od CPU nevyžadují řízení externím programem. Mohou proto být zřetězeny a vysoce paralelizovány, díky čemuž mohou dnešní grafické karty dosahovat vysoké výkonnosti.

Cesta geometrických dat začíná v klientské paměti (paměti CPU), kde jsou uložena spolu s daty rastrovými. Tato data lze dále volitelně ukládat do takzvaných *display listů*, což jsou

[5] Kompletní seznam rozšíření se specifikacemi lze nalézt v registru OpenGL na `www.opengl.org/registry/`



Obrázek 4.1: Grafický zobrazovací řetězec OpenGL 1.1

datové struktury fungující jako *cache* paměť pro příkazy, kterými data definujeme. Tyto příkazy se při použití *display listu* uloží v paměti serveru (v tomto případě grafické karty), což kromě výhodného umístění poskytuje i možnost tyto příkazy optimalizovat pro danou grafickou kartu.

Evaluátory slouží k převodu grafických primitiv na vrcholy. Používají se hlavně u primitiv zadaných parametricky, jako jsou například parametrické plochy. Vstupem do *evaluátorů* jsou řídící body těchto ploch a výstupem potom jednotlivé vrcholy.

Per-vertex operace je souhrnný název pro množinu operací, které se provádějí s každým vrcholem v této fázi zobrazovacího řetězce. Každý vrchol je transformován[6] pomocí modelové a pohledové matice (*modelview matrix*) do pohledového prostoru kamery a dále do ořezového prostoru kamery pomocí projekční matice (*projection matrix*). Nakonec se pomocí texturovací matice transformují texturovací souřadnice a na základě normál vrcholů a nastavení materiálů a světel se vypočítává osvětlení (používá se *Blinn-Phongův* osvětlovací model [Bli77]). Tato fáze zobrazovacího řetězce se také označuje jako T&L, neboli *Transform & Lighting*.

Před přechodem do další části zobrazovacího řetězce se z jednotlivých vrcholů sestavují grafická primitiva, která pak putují dále. Jak vypadají transformační matice se lze dočíst například v [ŽBSF04].

Následuje fáze ořezání primitiv – takzvaný *clipping*. Každé primitivum je ořezáno podle toho, zda spadá do prostoru viditelného kamerou či nikoliv. Tento prostor je vymezen šesti rovinami ohraničujícími viditelnou oblast a nazývá se pohledový jehlan (*view frustum*). Primitiva, která leží celá mimo tento prostor, jsou automaticky zahazována, a primitiva ležící celá uvnitř pohledového jehlanu pokračují do dalších částí zobrazovacího řetězce nezměněna.

[6] Podrobnější popis transformací v OpenGL je popsán například v `www.glprogramming.com/red`

A nakonec ta, která leží uvnitř viditelné oblasti pouze částečně, musí být ořezána, přičemž během tohoto procesu mohou vzniknout i nové vrcholy.

Dále následuje perspektivní dělení (*perspective divide*), které způsobí, že se vzdálené objekty budou jevit menší než ty blízké. Po tomto kroku jsou již vrcholy připraveny na takzvanou transformaci okno-formát (*viewport transformation*), díky které získáme souřadnice vrcholů přímo na obrazovce monitoru v pixelech. V závislosti na nastavení parametrů zobrazovacího řetězce může ještě proběhnout takzvaný *face culling*, kdy jsou zahozeny přivrácené či odvrácené plochy.

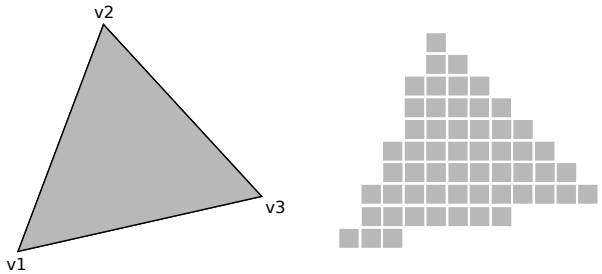
Před samotnou rasterizací se ještě podíváme na cestu rastrových dat. Operace s pixely zahrnují jejich načtení z paměti, dékodování jejich formátu a „rozbalení“ (*unpack*) na jednotlivé barevné složky. Ty pak mohou být prohozeny (*swap*), posunuty (*bias*), přemapovány (*map*) či může být změněn jejich kontrast (*scale*). Dále se tyto pixely uloží buď do texturovací paměti jako textura, nebo pokračují přímo do rasterizační fáze.

Pokud jsou používány textury, další fází může být spuštění algoritmů jako například kombinování textur či mipmapping. Následuje rasterizační fáze.

Zde se setkávají cesty geometrických a rastrových dat. V rasterizační jednotce se transformovaná a ořezaná grafická primitiva převádějí na takzvané *fragmenty*. *Fragment* je část grafického primitiva odpovídající svou velikostí pixelu na obrazovce, která však musí ještě podstoupit řadu testů, než se vůbec může stát pixelem. OpenGL pojem *fragment* používá právě k rozlišení mezi skutečnými pixely, z nichž je poskládán výsledný obraz, a těmito entitami vznikajícími v procesu rasterizace (viz obrázek 4.2). Narozdíl od pixelu, který je již reprezentován pouze svou souřadnicí na obrazovce a barvou, musí fragment nést navíc ještě další informace, které jsou následnými testy vyhodnocovány. Mezi tyto informace patří například hloubka fragmentu, hodnota alfa kanálu či texturovací koordináty. Dalšími částmi zobrazovacího řetězce již tedy putují pouze fragmenty.

Následně se provádějí takzvané *per-fragment* operace, jejichž úkolem je získat výslednou barvu každého fragmentu. Ta může být reprezentována přímo barvou, která byla fragmentu přidělena při rasterizaci, kde byla vypočtena pomocí interpolace barev jednotlivých vrcholů grafického primitiva. V případě, že je povoleno texturování, může být tato barva dále modulována hodnotou určitého *texelu*[7] z textury, jehož souřadnice jsou dány texturovacími koordináty. Mezi tyto operace se řadí také případný výpočet mlhy (*fog*), který může mít na výslednou barvu fragmentu taktéž vliv. Navíc můžeme použít takzvaný *alpha blending*, tedy smíchání barvy fragmentu s barvou pixelu nacházejícího se na stejném místě v rastru, a to v poměru daným hodnotou v alfa kanálu.

Před samotným uložením do *framebufferu* podstupuje dále každý fragment sérii takzvaných *per-fragment* testů, mezi něž patří například test hloubky (*depth test*) a test s hodnotou ve *stencil bufferu* (*stencil test*). Všechny tyto operace a testy lze povolit či zakázat, případně



Obrázek 4.2: Rasterizace trojúhelníku

upravit jejich chování skrze příkazy OpenGL. Je zřejmé, že různé nastavení může velmi razantně ovlivnit hodnotu, která se do *framebufferu* nakonec uloží.

Framebuffer si lze zjednodušeně představit jako zásobník na příchozí fragmenty. Ve skutečnosti se však jedná o sadu samostatných *bufferů* stejných rozměrů, z nichž pravděpodobně nejdůležitější je *color buffer*, jehož hodnoty zpravidla určují barvy pixelů na obrazovce. Mezi další *buffery* patří paměť hloubky (*Z-buffer*), šablonový buffer (*stencil buffer*) a akumulační buffer (*accumulation buffer*).

Poslední cestou, které si můžeme na obrázku 4.1 všimnout, je cesta z *framebufferu* zpět do klientské paměti. Díky ní máme možnost si hodnoty uložené ve *framebufferu* přečíst a poté si je uložit pro další použití.

4.2.3 Programovatelný grafický zobrazovací řetězec

Nevýhody výše popsané architektury grafického zobrazovacího řetězce jsou zřejmé, pokud si takto vystavěný řetězec představíme jako montážní linku a jednotlivé moduly jako obráběcí stroje. Uživatel může některé stroje vypínat či zapínat, případně jim nastavovat v omezené míře parametry, jak s daty putujícími po lince zacházet, ale vždy jen v rámci předem specifikované funkcionality. Příkladem může být například výpočet osvětlení ve fázi *per-vertex* operací, kdy můžeme nastavit vlastnosti světla a materiály objektů, ale vždy se nám nad těmito parametry bude počítat stejný osvětlovací model – v tomto případě *Blinn-Phongův*. A takovýchto příkladů se dá nalézt mnoho.

Vývojáři OpenGL si byli tohoto omezení vědomi, a tak byl roku 2001 ve verzi 2.0 implementován první koncept programovatelného zobrazovacího řetězce, jehož zjednodušené schéma vidíme na obrázku 4.3. Tento se od zobrazovacího řetězce s fixní funkcionalitou tehdy lišil dvěma zásadními věcmi. Jednalo se o nahrazení chování jednotek provádějících

[7] Tak, jako se používá termín *pixel* pro označení elementu na obrazovce, používá se pojem *texel* v souvislosti s texturami.

per-vertex a *per-fragment* operace takzvanými *vertex* a *fragment procesory* (blíže viz následující sekce této kapitoly), což jsou jednotky umožňující pomocí uživatelem definovaných programů – takzvaných *shaderů* – přepsat chování transformační a osvětlovací fáze, respektive fáze operací s fragmenty.

Pro implementaci těchto programů pak OpenGL nabízí jazyk GLSL (*OpenGL Shading Language*), jehož syntaxe je velice podobná jazyku C. Zkompilované[8] *shadery* se nahrávají do paměti grafické karty, kde jsou poté prováděny právě *vertex*, respektive *fragment* procesorem. V následujícím textu bude pojem *shader* používán jak pro označení těchto uživatelem definovaných programů, tak i pro jejich konkrétní instance právě zpracovávané na příslušných procesorech.

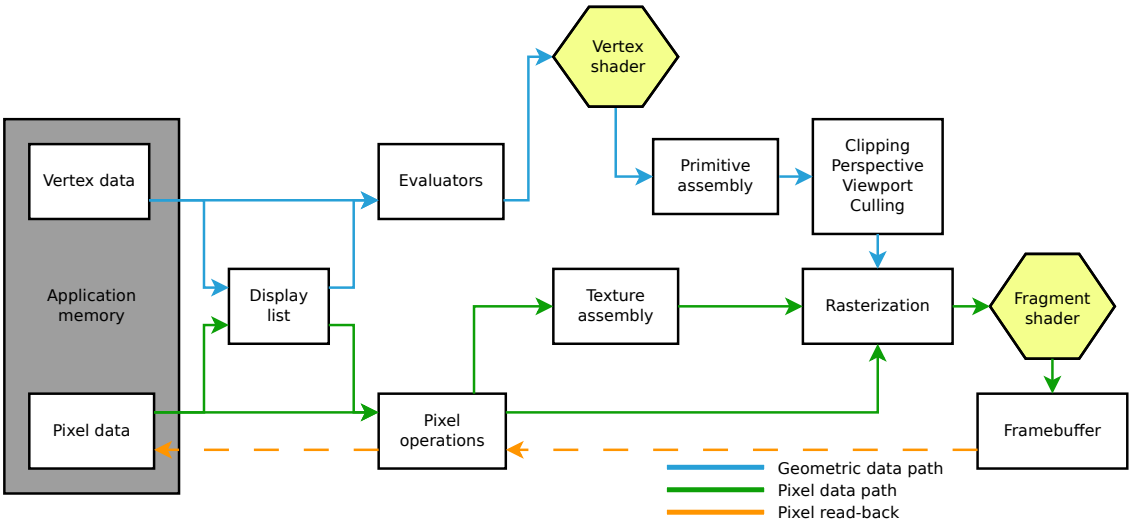
Využití funkcí programovatelného zobrazovacího řetězce, tedy využití *vertex* a *fragment shaderů*, je v OpenGL volitelné a pokud je použít nechceme, aplikuje se průchod fixním zobrazovacím řetězcem. Pokud se ale rozhodneme pro použití *shaderů*, máme nejen možnost, ale i povinnost alespoň minimální množinu z původní funkcionality fixního zobrazovacího řetězce implementovat. To tedy znamená, že ve *vertex shaderu* nelze například spočítat pouze transformace a nechat výpočet osvětlení na fixní funkcionalitě. Veškerou původní funkcionalitu musíme v tomto případě implementovat sami, přičemž musíme implementovat zároveň jak *vertex*, tak i *fragment shader*.

Jelikož jsou *vertex* i *fragment* procesory zapojeny do jinak fixního zobrazovacího řetězce, musí mít pevně definované vstupní a výstupní proměnné. Vstupní proměnné mohou být buď „uniformní“ (*uniform*), tedy neměnné a společné pro všechny vrcholy či fragmenty daného primitiva, případně „atributové“ (*attribute*), které jsou definovány zvlášť pro každý vrchol. Zvláštním typem jsou pak takzvané *varying*[9] proměnné, díky kterým lze předávat hodnoty z *vertex shaderu* do *fragment shaderu*, přičemž tyto hodnoty mohou být při přechodu mezi *shadery* interpolovány. Význam každé *varying* proměnné lze ještě navíc přesněji specifikovat uvedením klíčového slova `in` pro vstupní a `out` pro výstupní proměnné, obecně lze pak použít klíčové slovo `const` pro konstanty, čímž můžeme (ale také nemusíme) usnadnit práci kompilátoru.

Pro jednodušší práci se *shadery* nám GLSL poskytuje přístup k některým stavovým proměnným OpenGL prostřednictvím takzvaných vestavěných proměnných. Nejedná se však o nic jiného než o sadu klíčových slov, po jejichž použití kompilátor vytvoří proměnné odpovídajícího typu (tedy *uniform*, *attribute* či *varying*) a zajistí jejich naplnění příslušnými hodnotami z OpenGL či z předcházejících částí zobrazovacího řetězce.

[8] Příklady, jak vytvořit a zkompilovat shader, můžeme nalézt například na `www.lighthouse3d.com/opengl/glsl`, případně v tutorialech k OpenGL SDK na `www.opengl.org`.

[9] Podle depreciation modelu zavedeném ve verzi OpenGL 3.0 je užívání klíčového slova *varying* zastaralé a má se místo něj používat terminologie *in/out*.



Obrázek 4.3: Programovatelný grafický zobrazovací řetězec OpenGL 3.0

Aby nebyl uživatel omezen pouze na vestavěné vstupní proměnné, lze si z OpenGL aplikace do *shaderu* předat jakoukoliv další *uniform* či *attribute* proměnnou, což poskytuje vysokou volnost v samotné implementaci *shaderů*. Výstupní proměnné pro každý druh *shaderu* jsou potom pevně dány a musí být naplněny, jinak může být chování *shaderů* nepředvídatelné.

Pro implementaci samotné logiky *shaderu*, která se umisťuje do metody s názvem `main` bez parametrů, nám pak GLSL poskytuje nejen primitivní datové typy, jakými jsou například *integer*, *float* a *boolean*, nýbrž máme k dispozici i řadu komplexnějších a specifických struktur. Těmi jsou například dvou-, troj- a čtyřrozměrné vektory (`vec2-4`) a matice (`mat2x2-4x4`) vytvořené z primitivních datových typů a dále speciální struktury například pro materiál a světlo. Dále nám GLSL poskytuje řadu vestavěných matematických funkcí a množství funkcí pro práci s vektory a maticemi. Nemusíme tedy sami implementovat operace jako skalární či vektorový součin, normalizaci vektorů, násobení matic či výpočet směru odraženého paprsku. Zároveň si můžeme definovat vlastní funkce.

4.2.3.1 Vertex shader

Vertex shader je uživatelem definovaný program, který nahrazuje chování jednotky provádějící *per-vertex* operace, a má tedy na starost implementovat chování T&L fáze. Tento program se spouští pro každý vrchol poslaný skrze volání OpenGL do zobrazovacího řetězce a jeho vstupními proměnnými jsou atributy každého vrcholu tak, jak jsou v naší OpenGL aplikaci definovány.

Pro implementaci transformační fáze máme tedy k dispozici vestavěné vstupní atributové proměnné jako souřadnice vrcholu (`gl_Vertex`), jeho normálu (`gl_Normal`) a texturovací souřadnice každé texturovací jednotky (`gl_MultitexCoord0-7`). K implementaci osvětlení pak atributovou proměnnou barvy vrcholu (`gl_Color`) a uniformní struktury s vlastnostmi světel (`gl_LightSource[MAX_LIGHTS]`) a aktuálně používaného materiálu (`gl_FrontMaterial`).

Dále máme jako uniformní proměnné k dispozici transformační matice, a to „modelově-pohledovou“ (`gl_ModelViewMatrix`), projekční (`gl_ProjectionMatrix`) a normálovou matici (`gl_NormalMatrix`), která slouží k transformaci normál do pohledového prostoru kamery, a nakonec texturovací matici pro každou texturovací jednotku (`gl_TextureMatrix[MAX_TEXTURE_UNITS]`).

Na výstupní proměnné je kladena jediná podmínka, a sice, že musíme naplnit alespoň proměnnou `gl_Position`, do které musíme uložit transformovanou pozici vrcholu. Pokud tak neučiníme, nemůže dojít k rasterizaci a tím pádem se nic nezapíše do *framebufferu*. Dále můžeme specifikovat výstupní barvu vrcholu v proměnné `gl_FrontColor`, transformované (či vygenerované) texturovací souřadnice v `gl_TexCoord[MAX_TEXTURE_UNITS]` a další.

Zde je ukázka jednoduchého *vertex shaderu*, který pouze transformuje příchozí vrchol do ořezového prostoru kamery a nastaví jeho barvu:

```
void main()
{
    // nastavení barvy
    gl_FrontColor = gl_Color;

    // transformace pozice vrcholu
    // Pozn.: následující tři příkazy jsou ekvivalentní
    gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;

    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    gl_Position = ftransform();
}
```

Podrobnější výčet možností lze nalézt v registru OpenGL ve specifikaci rozšíření GL_ARB_vertex_shader.

4.2.3.2 Fragment shader

Podobně jako *vertex shader* dovoluje přepsat chování T&L jednotky, dokáže *fragment shader* přepsat chování jednotky pro *per-fragment* operace. Tento program se spouští pro každý fragment, který vznikl rasterizací grafických primitiv poslaných do zobrazovacího řetězce prostřednictvím příkazů OpenGL. Principielně jsou si oba *shadery* velmi podobné, liší se pouze sadou vstupních a výstupních proměnných.

Vstupem do *fragment shaderu* jsou souřadnice fragmentu (`gl_FragCoord`) a interpolované hodnoty z výstupu *vertex shaderu*, jako barva (`gl_Color`) a texturovací souřadnice (`gl_TexCoord[MAX_TEXTURE_UNITS]`). Zároveň jsou zde vstupem všechny *varying* proměnné produkované *vertex shaderem*.

Minimálním výstupem *fragment shaderu* je pak informace o finální barvě fragmentu v podobě zápisu do proměnné `gl_FragColor`, případně můžeme použít speciálního klíčového slova ***discard***, které způsobí zahození fragmentu a ukončení programu. Dále je možné upravit hloubku fragmentu zápisem do proměnné `gl_FragDepth`, přestože samotné souřadnice fragmentu v rastru již v této fázi měnit nelze. To však není nutné, neboť hloubka fragmentu je již vypočítána při rasterizaci.

Jednou z nejdůležitějších schopností *fragment shaderu* je možnost přístupu do paměti textur[10]. Ta je zprostředkována pomocí speciálních proměnných, takzvaných *samplerů*, které reprezentují aktivní texturovací jednotky s příslušnými texturami. GLSL pak defunuje sadu funkcí, pomocí kterých můžeme do těchto *samplerů*, respektive k jejich texturám, přistupovat.

Následuje příklad jednoduchého *fragment shaderu*, který zkombinuje barvu příchozího fragmentu s barvou texelu z textury v poměru jedna k jedné:

```
uniform sampler2D tex;

void main()
{
    // získání hodnoty texelu
    vec4 value = texture2D(tex, gl_TexCoord[0].st);

    // nastavení barvy
    gl_FragColor = mix(value, gl_Color, 0.5);
}
```

[10] Nové grafické karty umožňují přímý přístup do paměti textur i ve *vertex shaderu*.

Všimněme si zvláštního zápisu `gl_TexCoord[0].st`. Jedná se o takzvaný *swizzle-mask*, pomocí kterého se můžeme odkazovat na jednotlivé složky vektoru. Čtyři složky texturovacích souřadnic se zpravidla označují písmeny {s, t, p, q}, souřadnice pozic jsou {x, y, z, w} a složky barevného vektoru zase {r, g, b, a}. Jednotlivá písmena lze v rámci jedné množiny písmen libovolně kombinovat a přeskupovat (například `value.bgra` nebo `value.xxyy`), napříč množinami však nikoliv (například `value.xxbb` způsobí chybu).

Podrobnější výčet možností *fragment shaderu* lze nalézt v registru OpenGL ve specifikaci rozšíření GL_ARB_fragment_shader.

4.3 Možnosti a techniky používané v dnešních GPU

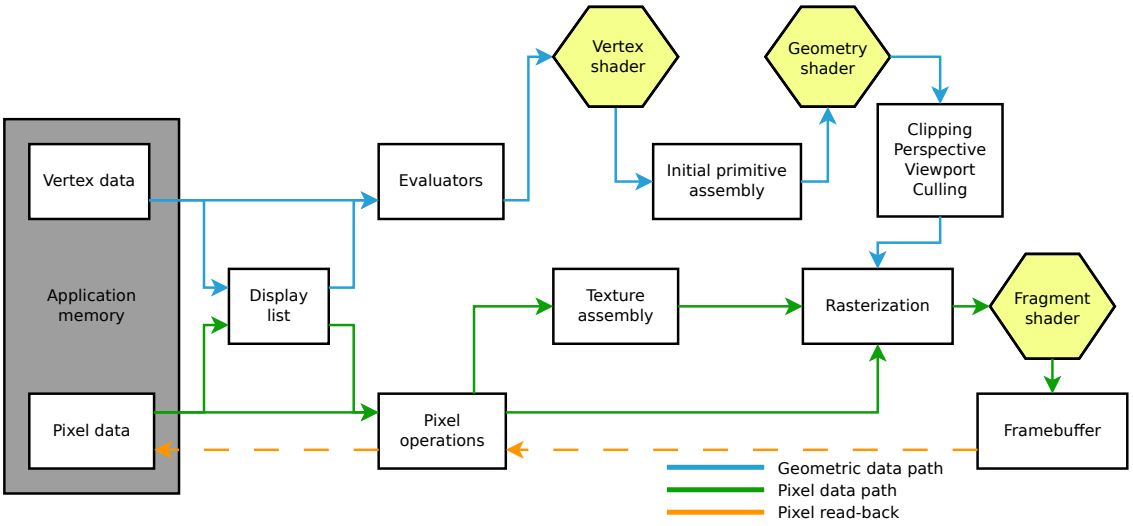
Tato kapitola má za úkol stručně, avšak dostatečně seznámit čtenáře s možnostmi a technikami, které nabízejí dnešní grafické karty a které mají souvislost s touto prací a implementací prezentovaného algoritmu pohledově závislých stínových map.

4.3.1 Geometrický procesor

Předchozí kapitola 4.2.3 popisuje, jak vypadal programovatelný grafický řetězec před příchodem verze OpenGL 3.0. V této verzi totiž došlo k další změně, a sice o rozšíření tohoto řetězce o nový prvek – tím je takzvaný geometrický procesor (*geometry processor*). Jeho úkolem je dále zpracovávat grafická primitiva poté, co byly jejich jednotlivé vrcholy zpracovány vertexovým procesorem. Kromě toho, že může geometrický procesor tato primitiva, respektive jejich vrcholy, dále transformovat nebo počítat jejich osvětlení, může také přidávat nové vrcholy či vrcholy zahazovat. Díky tomu je schopen generovat i grafická primitiva jiného typu, než jsou ta, která přijímá na svém vstupu.

Principielně na tom není *geometry shader* jinak, než *vertex* a *fragment shader*. Jeho použití v programovatelném zobrazovacím řetězci je opět volitelné, tentokrát však můžeme použít buď jen *vertex* a *fragment shader*, anebo všechny tři *shadery* dohromady. Při použití *geometry shaderu* musíme navíc po kompilaci specifikovat, jaká grafická primitiva očekáváme na jeho vstupu a jaká na jeho výstupu. Na vstupu jsou přijímány buď body (GL_POINTS), úsečky (GL_LINES), trojúhelníky (GL_TRIANGLES) nebo trojúhelníky se sousedností (GL_TRIANGLE_ADJACENCY), což je nová struktura přidaná tímto rozšířením. Ta dovoluje kromě trojúhelníku specifikovat i informaci o jeho sousedech. Na výstupu potom mohou být opět body, úsečky či trojúhelníky, ale také série navazujících úseček (GL_LINE_STRIP) nebo série navazujících trojúhelníků (GL_TRIANGLE_STRIP).

Pro lepší představu zařazení *geometry shaderu* do stávajícího programovatelného zobrazovacího řetězce se podívejme na obrázek 4.4. Jak zde můžeme vidět, tak z vrcholů, které



Obrázek 4.4: Zařazení geometry shaderu do programovatelné pipeline

vystupují z *vertex shaderu*, se sestaví grafická primitiva a tato pokračují právě do *geometry shaderu*. Na jeho vstupu již tedy nejsou samostatné vrcholy, ale množina vrcholů, jejíž velikost je dána tím, jaké jsme specifikovali vstupní primitvum. V případě trojúhelníku to tedy budou tři vrcholy, v případě úsečky dva.

Aby bylo možné vytvořit například z trojúhelníku sérii navazujících trojúhelníků, musí *geometry shader* umět generovat nové vrcholy – a právě to je jeho hlavní schopností. S jejich využitím můžeme nyní přímo na GPU implementovat algoritmy jako generování stínových těles [SWK07], displacement mapping [SKU08] a další, jež do této doby potřebovaly zásadní asistenci CPU.

Jelikož *geometry shader* již nepracuje s jedním vrcholem, jako tomu bylo u *vertex shaderu*, nýbrž s celým grafickým primitivem, nejsou jeho vstupem jednotlivé atributové proměnné, ale rovnou pole těchto proměnných. Jejich přesný počet je definován v konstantě `gl_VerticesIn` a souřadnice jednotlivých vrcholů jsou uloženy v poli `gl_PositionIn[gl_VerticesIn]`. Podobně jsou například uloženy i barvy jednotlivých vrcholů primitiva, a sice v poli `gl_FrontColorIn[gl_VerticesIn]`. Taktéž veškeré příchozí *varying* proměnné přicházejí jako pole a ne jako jednotlivé hodnoty, jak tomu bylo při předávání hodnot mezi *vertex* a *fragment* shaderem. Aby *geometry shader* mohl rozlišit, které vrcholy jsou vstupní a které výstupní, používají se před deklarací proměnných klíčová slova `in` a `out`. Dále jsou v *geometry shaderu* stejně jako ve *vertex shaderu* k dispozici modelově-pohledová, projekční i texturovací matice a stejně jako tento shader má i *geometry shader* přístup k informacím o světlech ve scéně a nastavených materiálech. Obdobně je také možné přímo přistupovat do paměti textur.

Výstupní proměnné *geometry shaderu* jsou v podstatě totožné s výstupními proměnnými *vertex shaderu* a nastavují se pro každý vrchol, který je následně odeslán („emitován“) k dalšímu zpracování zobrazovacím řetězcem. K tomuto účelu slouží funkce `EmitVertex()`, která je specifická pouze pro *geometry shader*, a kterou je tedy třeba zavolat po každém nastavení výstupních proměnných. Druhou specifickou funkcí je `EndPrimitive()`, jejímž zavoláním oznámíme, že jsme emitovali všechny vrcholy pro dané výstupní primitivum. Toto je důležité, neboť tento mechanismus *geometry shaderu* umožňuje emitovat i více na sobě nezávislých primitiv. Pokud se opět vrátíme k obrázku 4.4, uvidíme, že takto emitovaná grafická primitiva pokračují dále do fáze ořezání, perspektivního dělení, *viewport* transformace a následné rasterizace stejně, jako je tomu u fixního zobrazovacího řetězce.

Na následujícím příkladu vidíme jednoduchou implementaci takzvaného *pass-through geometry shaderu*, který nedělá nic jiného, než že přeposílá data z *vertex shaderu* do *fragment shaderu*:

```
// zde je nutné explicitně pomocí direktivy povolit rozšíření
#version 120
#extension GL_EXT_geometry_shader4 : enable

void main()
{
        for(int i = 0; i < gl_VerticesIn; ++i)
        {
                // uložení barvy vrcholu do výstupní proměnné
                gl_FrontColor = gl_FrontColorIn[i];

                // uložení pozice vrcholu do výstupní proměnné
                gl_Position = gl_PositionIn[i];

                // odeslání vrcholu
                EmitVertex();
        }

        // ukončení primitiva - zde nepovinné, jelikož generujeme
        // pouze jedno
        EndPrimitive();
}
```

Na tomto místě ještě podotkněme, že *geometry shader* je pro svou schopnost generovat

nová grafická primitiva také stěžejní jednotkou pro implementaci algoritmu, jenž je předmětem této práce, viz kapitola 6.2.

Základní verze *geometry shaderu* je poskytována rozšířením OpenGL s označením GL_EXT_geometry_shader4. Dále existuje hardwarově specifické rozšíření s názvem GL_NV_geometry_shader4, které přidává k původnímu rozšíření určitou funkcionalitu. Bližší informace lze nalézt ve specifikacích těchto rozšíření v registru OpenGL. Příklady lze nalézt také například v oficiální OpenGL Wiki[11].

4.3.2 Framebuffer objekt

Framebuffer objekt (FBO) je rozšíření definující rozhraní pro renderování obrazu do jiných bufferů než těch, které jsou OpenGL poskytovány systémem. Tyto buffery se obecně nazývají „připojitelné objekty“ (*framebuffer-attachable images*) a dělí se na dva druhy. Jedním z nich jsou textury (přesněji objekty textur) a druhým takzvaný *renderbuffer objekt*, což je nový objekt definovaný tímto rozšířením, který slouží jako buffer k uložení jednoho 2D obrazu.

FBO disponuje množstvím takzvaných „přípojných bodů“ (*attachment points*), ke kterým lze tyto objekty připojovat. Tyto body se dělí na tři typy podle své funkce – prvním je GL_COLOR_ATTACHMENTi, což je bod sloužící pro připojení objektů schopných uchovávat barevnou informaci a brát tak na sebe úlohu *color bufferu*, to znamená texturu v kompatibilním barevném formátu jako například GL_RGBA. Maximální počet bodů tohoto druhu, který lze k FBO připojit, je definován v proměnné GL_MAX_COLOR_ATTACHMENTS a může se lišit v závislosti na hardware. Dalším typem je GL_DEPTH_ATTACHMENT, což je bod, ke kterému lze připojit objekt schopný ukládat informaci o hloubce fragmentů. Tím může být opět textura, tentokrát v hloubkovém formátu GL_DEPTH_COMPONENT, nebo také *renderbuffer objekt*. Posledním typem je GL_STENCIL_ATTACHMENT, kam lze připojit *stencil buffer*. Připojení těchto objektů k přípojným bodům ve *framebuffer objektu* ilustruje obrázek 4.5.

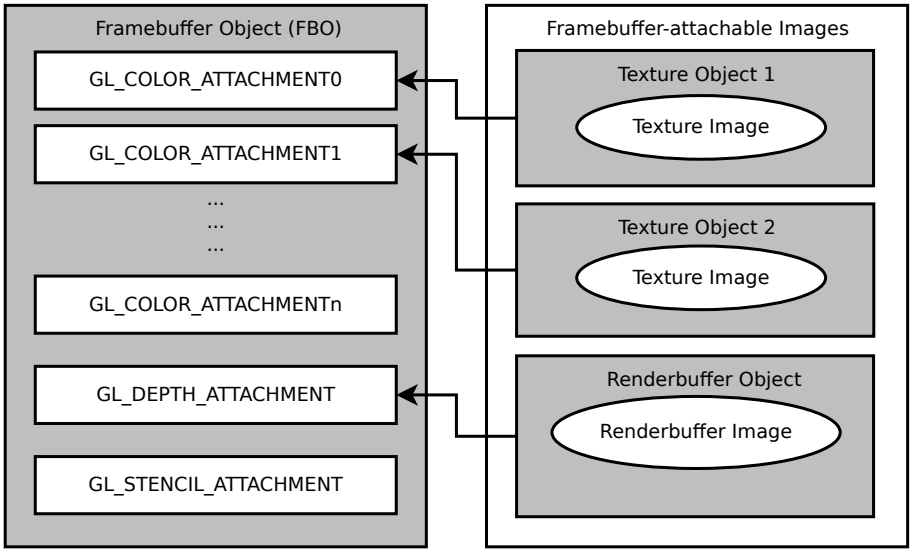
Framebuffer objekt je poskytován rozšířením GL_EXT_framebuffer_object. Jeho specifikaci spolu s příklady[12] použití lze pod tímto označením nalézt v registru OpenGL.

4.3.2.1 Vykreslování do textur

Vytvořit obraz scény s komplexním osvětlením, stíny a jinými efekty, lze za použití dnešních algoritmů jen stěží v jednom vykreslovacím průchodu. Často musíme provést mezikroky,

[11] Oficiální OpenGL Wiki se nachází na adrese `www.opengl.org/wiki`

[12] Příklady ilustrující práci s FBO lze nalézt také v oficiální Wiki OpenGL na adrese `www.opengl.org/wiki`, další pak na `www.songho.ca/opengl`



Obrázek 4.5: Připojení objektů k FBO

jako například zobrazení scény z jiného, pomocného pohledu za účelem získání informace o prostorovém uspořádání objektů ve scéně, jejichž výsledek nechceme zobrazovat na obrazovku, nýbrž někam uložit k dalšímu použití. Právě k tomuto účelu slouží technika vykreslování do textur (*render-to-texture*, RTT), jež nám umožňuje takto získaná data snadno a rychle uložit do paměti grafické karty v podobě textury, kterou můžeme využít v dalších krocích algoritmu.

Tato technika využívá schopnosti framebuffer objektu „přesměrovat“ vykreslované pixely přímo do textury, a to bez účasti hlavního procesoru. Za běhu aplikace tak můžeme po aktivaci FBO určit, do kterého místa (připojeného objektu) se bude vykreslovat, k čemuž slouží standardní OpenGL funkce `glDrawBuffer`, která díky tomuto rozšíření přijímá nový parametr GL_COLOR_ATTACHMENTi, kde `i` je pořadové číslo přípojného bodu. Za normálních okolností musí mít FBO připojen alespoň jeden objekt k bodu ukládajícímu barevnou informaci a jeden objekt k bodu ukládajícímu informaci o hloubce fragmentů. Vypnutím hloubkového testu a nastavením `glDepthMask` na GL_FALSE však lze ukládat i jen barevnou informaci a nastavením `glDrawBuffer` na GL_NONE lze ukládat pouze informaci o hloubce.

Díky výše popsaných mechanismům je tedy možné vyhnout se doposud používanému a neefektivnímu způsobu kopírování pixelů přímo z *framebufferu* do paměti, kdy se používala funkce `glReadPixels`. Podotkněme jen, že tato funkce má však stále svá použití.

Technika vykreslování do textur je hojně využívána v implementační části této práce, blíže viz kapitola 6.

4.3.2.2 Vykreslování do více textur

Důvodem, proč FBO poskytuje uživateli připojit ne jeden, ale více připojitelných objektů, je jeho schopnost renderovat do více textur zároveň. Tato technika se nazývá *multiple-render-target* (MRT).

Vykreslování do více textur zároveň připojených k FBO lze nastavit použitím funkce `glDrawBuffers`. Funguje podobně jako standardní metoda `glDrawBuffer`, avšak s tím rozdílem, že tentokrát nepředáváme identifikátor pouze jednoho přípojného bodu, ale pole těchto bodů.

Aby mohla být tato funkcionalita využita i při použití programovatelného zobrazovacího řetězce, je rozšíření *framebuffer objekt* úzce spjato s *fragment shaderem*. Pokud totiž máme správně definované přípojné body, do kterých se má vykreslovat, lze ve *fragment shaderu* selektivně určit, do kterého z nich bude výstup přeposílán. Namísto výstupní proměnné `gl_FragColor` nám tedy GLSL poskytuje novou proměnnou `gl_FragData[i]`, kde `i` je indexem do pole přípojných bodů, které jsme předali funkci `glDrawBuffers`.

4.3.3 Vertex Buffer Objekt

Vertex Buffer Objekt (VBO) je rozšíření, které bylo vyvinuto za účelem zvýšení výkonu OpenGL tím, že poskytuje rychlejší způsob vkládání vrcholů do zobrazovacího řetězce. Dosahuje toho tím, že si bere to nejlepší z *display listů* a takzvaných vertexových polí, přičemž netrpí jejich nedostatky.

V prvních verzích OpenGL bylo možné do grafického zobrazovacího řetězce posílat vrcholy, respektive jejich atributy (souřadnice, normály, barvy, texturovací koordináty), pouze pomocí speciálních funkcí, volaných pro každý tento atribut. Tento režim definování geometrie se označoval termínem *immediate* a jak lze vytušit, již pro vykreslení trochu složitější geometrie bylo třeba stovek, ba i tisíců takto zavolaných funkcí, což vykazovalo nezanedbatelné zpoždění. Vývojáři OpenGL si toho byli vědomi, a tak vznikla právě výše zmíněná vertexová pole.

Vertexová pole (*vertex arrays*, VA) umožňují uložení atributů vrcholů do polí, která lze poté celá vykreslit jedním voláním funkce, čímž dokáží ušetřit režii jinak potřebnou na početná volání funkcí v režimu *immediate*. Data uvnitř těchto polí lze navíc jednoduše modifikovat. Nevýhodou potom je, že jsou tato datová pole uložena v klientské paměti počítače (tedy CPU) a po každém vykreslení musí být přenesena do paměti GPU, spravované OpenGL. Na druhé straně máme *display listy*, o kterých již byla řeč v souvislosti s fixním zobrazovacím řetězcem v kapitole 4.2.2. Ty sice pracují v rychlé paměti GPU, ale po svém vytvoření již nedovolují měnit svůj obsah.

Vertex Buffer Objekt je tedy kombinací obou předchozích přístupů. Dovoluje vytvořit souvislý blok vysoce optimalizované a rychlé paměti na straně grafické karty, do kterého lze ukládat data vrcholů podobně jako za použití *display listů*, přičemž poskytuje funkce pro přístup k těmto datům, jako to bylo možné v případě *vertexových polí*.

VBO je součástí OpenGL od verze 1.5 a pod názvem GL_ARB_vertex_buffer_object je veden v registru OpenGL.

4.3.4 Pixel Buffer Objekt

Pixel Buffer Objekt (PBO) je v podstatě pouze rozšířením VBO, které vzniklo za účelem vytvoření možnosti ukládat do optimalizované paměti GPU nejen vrcholy a jejich atributy, ale i pixely. To nám potom dává například možnost načítat textury bez účasti CPU, kdy máme v PBO uloženy jednotlivé pixely a můžeme je pomocí DMA (*Direct Memory Access*) nahrát do textury. Toto funguje samozřejmě i opačně.

Další výhodou je potom asynchroní přenos dat. Při zavolání funkce `glReadPixels` bez PBO – tedy standardním způsobem – se zablokuje vlákno provádějící tuto funkci, neboť ovladač musí zajistit, abychom po jejím proběhnutí měli v klientské paměti platná data. Kontrola je proto aplikaci vrácena až poté, co celý přenos dat proběhne. S použitím PBO se však volání této funkce vrací okamžitě, jelikož přistupovat do PBO není možné jinak než opět pouze přes OpenGL, a tedy přes ovladač. Vlákno tudíž nemusí být blokováno a může pokračovat dále bez zdržení.

Stejně jako VBO není ani PBO v podstatě ničím jiným, než polem *bytů* uloženým v optimalizované paměti GPU, a jako na takové na něj nahlíží i grafická karta, potažmo OpenGL. To totiž mezi těmito *buffery* až na určité výjimky nerozlišuje a za běhu aplikace lze jeden typ přetypovat[13] na druhý a obráceně. Tento mechanismus umožňuje vznik techniky vykreslování do vertexových polí, kterou nám přiblíží následující kapitola.

PBO je součástí OpenGL od verze 2.1 a pod názvem GL_ARB_pixel_buffer_object lze najít jeho specifikaci v registru OpenGL.

4.3.5 Vykreslování do vertexových polí

Ještě nedávno nebylo možné posílat geometrická data scény do zobrazovacího řetězce jiným způsobem, než prostřednictvím řídící aplikace. Grafická karta pak nedělala nic jiného, než že takto předem danou scénu na základě nastavených parametrů zobrazovala. Až nyní

[13] Nejedná se zde o přetypování ve významu, v jakém ho vnímáme například v jazyce C. Ve skutečnosti pouze řekneme OpenGL, aby na data – tedy pixely – v PBO pohlíželo jako na vrcholy VBO.

se nám otevírají v tomto směru nové možnosti, kdy pomocí techniky vykreslování do vertexových polí již grafická karta nemusí pasivně přijímat geometrii od řídící aplikace – může generovat geometrii vlastní, a to bez přítomnosti geometrického procesoru.

Technice vykreslování do vertexových polí (*render-to-vertex-array, RTVA*) dala vzniknout univerzálnost programovatelného zobrazovacího řetězce spolu s možnostmi framebuffer objektu, pixel buffer objektu a vertex buffer objektu. Její princip spočívá v tom, že do textury připojené k FBO vykreslíme pomocí *fragment shaderu* obraz. Nemusí jít samozřejmě o informace o barvě, díky *shaderu* lze do textury uložit jakákoliv data. Tato data poté bez účasti CPU, tedy pouze v rámci paměti grafického procesoru, zkopírujeme do PBO. Díky tomu, že grafická karta, a tedy i OpenGL, pohlíží na PBO i VBO stejně, lze bez problémů data uložená v jednotlivých pixelech interpretovat a následně vykreslit pomocí VBO jako vrcholy.

Příklad na použití RTVA můžeme najít v sekci „Usage examples“ ve specifikaci rozšíření GL_ARB_pixel_buffer_object v registru OpenGL.

Tato technika je použita v optimalizaci algoritmu pohledově závislých stínových map, konkrétně viz kapitola 7.4.

Kapitola 5

Algoritmy pro generování ostrých stínů

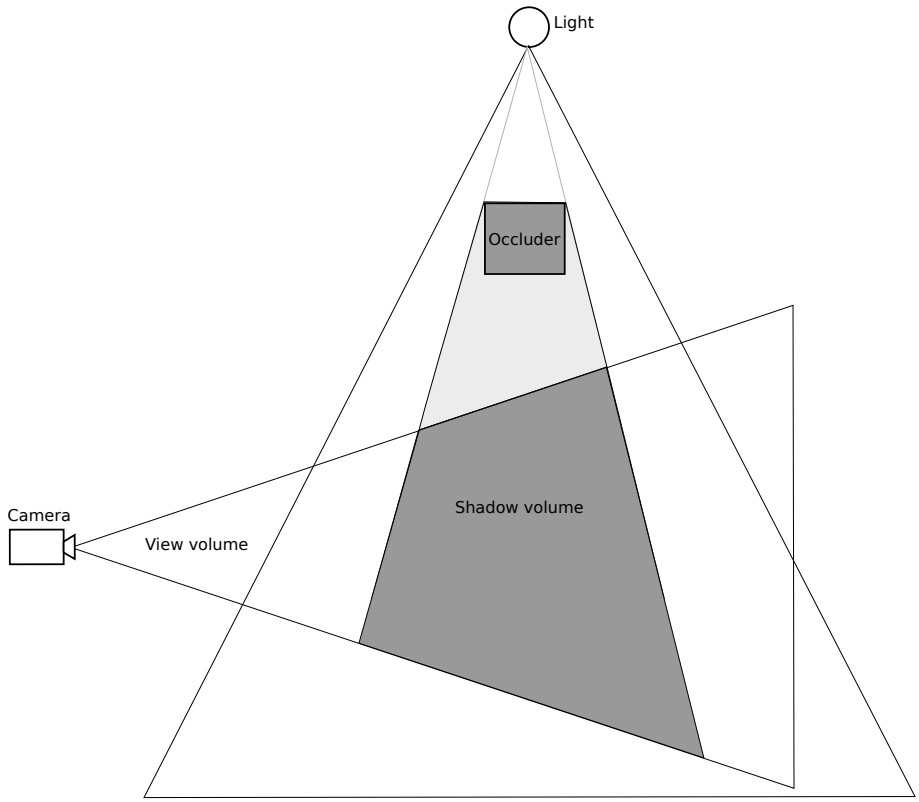
Generování stínů je výpočetně velmi náročná operace a existuje řada algoritmů, které se jejich výpočtem zabývají. Neustále se zrychlující rozvoj hardware, zejména pak vývoj stále výkonnějších grafických karet, nám otevírá nové možnosti jak tyto algoritmy realizovat rychleji a efektivněji, případně navrhnout techniky zcela nové.

V této práci se nebudeme zabývat výčtem dostupných metod pro generování ostrých stínů, neboť to za nás již udělal například *Woo* [WPF90] a měkkým stínům se pak věnuje například článek [HLHS03]. My se místo toho zaměříme na dvě dnes pravděpodobně nejpoužívanější techniky pro generování stínů, kterými jsou metoda stínových těles, neboli *shadow volumes*, a metoda stínových map, takzvaný *shadow mapping*. V první z nich se zaměříme pouze na části relevantní pro algoritmus pohledově závislých stínových map, který je vysvětlen v kapitole 6. Metodu stínových map probereme podrobněji včetně implementace.

5.1 Metoda stínových těles

První z nejpoužívanějších metod pro generování ostrých stínů ve scéně je metoda zvaná *shadow volumes*. Tuto metodu poprvé prezentoval *Frank Crow* v roce 1977 [Cro77] jako geometrickou metodu sloužící k výpočtu takzvaných stínových těles, což jsou mnohostěny, ohraničující oblasti vzniklé zastíněním světelného zdroje objekty ve scéně, viz obrázek 5.1.

Hlavní výhoda metody stínových těles oproti metodě stínových map spočívá v tom, že generuje přesné stíny pro jakékoliv rozlišení. To je však vykoupeno vyšší výpočetní náročností, jelikož metoda stínových těles je závislá na složitosti geometrie scény – pracuje totiž v prostoru objektů – a hlavně omezeními kladenými na reprezentaci scény. Zatímco metoda

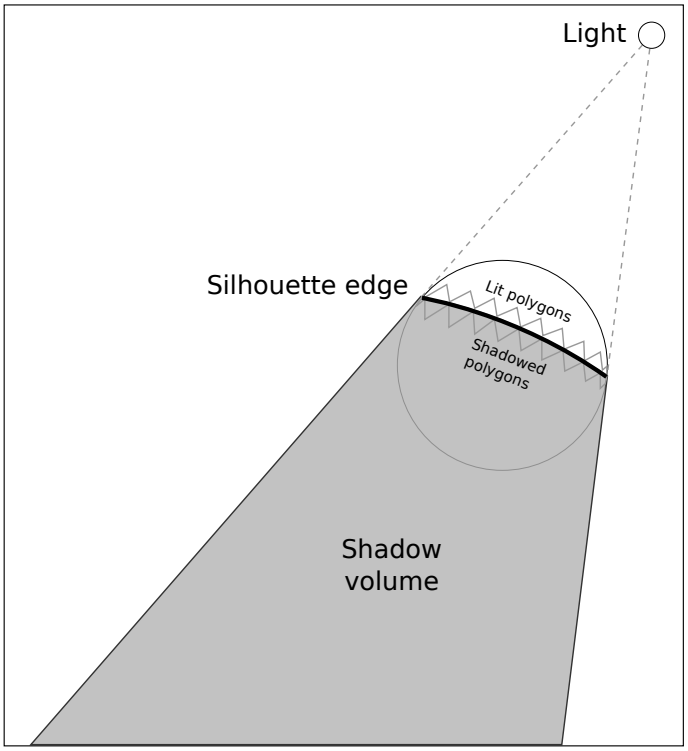


Obrázek 5.1: Stínové těleso

stínových map dokáže generovat i stíny vržené parametrickými plochami, vytvoření příslušného stínového tělesa z takovýchto ploch je přinejmenším velice obtížné, ne-li nemožné v reálném čase.

Konstrukce stínového tělesa pro jeden polygon je snadná. Stínové těleso vytvoříme tak, že vezmeme každou hranu polygonu a protáhneme ji podél směru světelných paprsků pryč od světelného zdroje tak, aby pokrýval veškerou viditelnou plochu – často tedy až do nekonečna. Plochu vzniklou takovýmto protažením pak nazýváme stínovým polygonem a množina těchto polygonů pak tvoří stínové těleso.

Pokud máme objekt složený z více polygonů, ideálním přístupem by bylo nalézt jeho takzvanou siluetu, čili množinu jeho obrysových hran (*silhouette edges*) z pohledu světla. Protažením těchto hran do nekonečna by poté bylo vytvořeno ideální stínové těleso z hlediska počtu generovaných stínových ploch. V praxi se však používá zjednodušená metoda konstrukce stínového tělesa, spočívající v nalezení takzvaných potenciálních obrysových hran (*possible silhouette edges*), které nalezneme snadněji, jelikož se jedná o hrany spojující plochy přivrácené ke světlu s plochami odvrácenými. Protažením těchto hran sice nevytvoříme ideální stínové těleso, nicméně je tento přístup efektivnější než projektovat každý polygon



Obrázek 5.2: Silueta stínového tělesa

objektu.

Typicky vytváříme takovéto stínové těleso pro každý potenciálně stínící objekt. Jakmile máme množinu těchto objektů, stačí potom během vykreslování scény testovat polohu jednotlivých polygonů vůči těmto tělesům. Prakticky mohou nastat tři případy. Polygon leží buď celý mimo stínové těleso a je tedy plně osvětlen, nebo leží celý uvnitř stínového tělesa a leží tedy ve stínu, případně je polygon zastíněn pouze napůl, a tehdy je nutné ho rozdělit na osvětlenou a zastíněnou část.

Test polohy polygonu vůči stínovému tělesu se většinou neprovádí geometricky a používají se rastrové metody. Obecně spočívají v tom, že se každým pixelem obrazovky vrhne paprsek a počítá se, kolikrát tento paprsek vstoupí do stínového tělesa a kolikrát z něj vystoupí. Při vstupu do tělesa se inkrementuje čítač, při výstupu se dekrementuje. Pokud je na konci hodnota čítače nenulová, musel paprsek po svém dopadu k testovanému bodu zůstat ve stínovém tělese a tento bod tedy leží ve stínu.

Podrobnému popisu algoritmů se již věnovat nebudeme, jelikož to není bezpodmínečně nutné pro pochopení algoritmu, který je předmětem této práce, zejména potom jeho optimalizací. Řekněme si jen, že výše popsaný algoritmus, označený jako *depth-pass*, implementoval v roce 1991 *Heidmann* [Hei91] za použití *stencil bufferu* a některé z jeho nedostatků odstra-

nil později Kilgard [Kil01]. Robustní variantu algoritmu, takzvaný *depth-fail*, navrhnul opět Kilgard [EK02].

5.2 Metoda stínových map

5.2.1 Úvod

Metoda stínových map – *shadow mapping*, nebo také *shadow depth map* – byla poprvé prezentována *Lancem Williamsem* v roce 1978 [Wil78] a od té doby, co je toho grafický hardware schopen, je v počítačové grafice hojně využívána k přidání stínů do scény. Používá se jí jak v realtimové grafice, tak i v předrenderovaných scénách – byla použita například při renderování filmu *Toy Story* studia *Pixar*.

Narozdíl od většiny ostatních metod pro generování ostrých stínů, pracuje metoda stínových map v obrazovém prostoru. Díky tomu je nezávislá na geometrii a reprezentaci scény, z čehož plyne její velmi vysoká rychlost. Na druhou stranu však trpí všemi nedostatky, které se při práci s diskretizovaným obrazovým prostorem projevují.

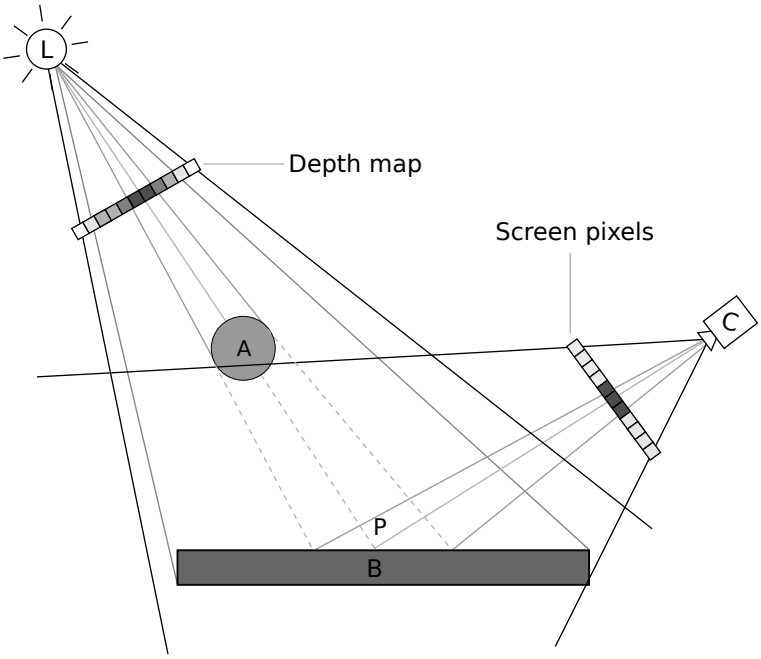
5.2.2 Princip metody

Princip metody stínových map je prostý. Staví na základní myšlence, že veškeré plochy viditelné ze světelného zdroje jsou osvětlené a vše ostatní leží ve stínu. Pokud se tedy na scénu podíváme z pohledu světla, můžeme poměrně snadno určit, ve kterých místech se bude vyskytovat stín a která místa budou osvětlena. Výstupem této metody je potom takzvaná stínová mapa (*shadow map*), což je obraz popisující rozmístění takto určených stínů ve scéně.

Na základě tohoto principu lze potom získání stínové mapy provést ve dvou krocích. V prvním kroku si zobrazíme scénu z pohledu světla a do *bufferu* – hloubkové mapy[1] (*depth map*) – uložíme hodnoty hloubky z viditelné, a tedy osvětlené části scény. To znamená, že v každém pixelu této mapy budeme mít uloženu vzdálenost nejbližšího tělesa vzhledem ke světlu. Ve druhém kroku zobrazujeme scénu v pohledu z kamery a jednotlivé body scény transformujeme do pohledu světla, čímž získáme vzdálenosti těchto bodů v prostoru světla. Tyto vzdálenosti potom porovnáváme s hodnotami uloženými v hloubkové mapě. Pokud je přepočtená vzdálenost větší, než hodnota v hloubkové mapě, leží bod ve stínu.

Tento princip je ilustrován na obrázku 5.3 a algoritmus (pro jeden světelný zdroj) lze zapsat následujícím pseudo-kódem:

[1] Poznamenejme, že termíny stínová mapa a hloubková mapa nelze zaměňovat. Hloubkovou mapou se rozumí mapa s uloženými hodnotami hloubky *z-bufferu*, kdežto stínová mapa je pouze binární mapa zastínění, která je výsledkem algoritmu stínových map.



Obrázek 5.3: Princip algoritmu stínových map

1. Zobraz scénu z pohledu světla L a ulož hodnoty hloubky (*z-buffer*) do hloubkové mapy D.

2. Zobraz scénu z pohledu kamery C a pro každý bod scény prováděj:

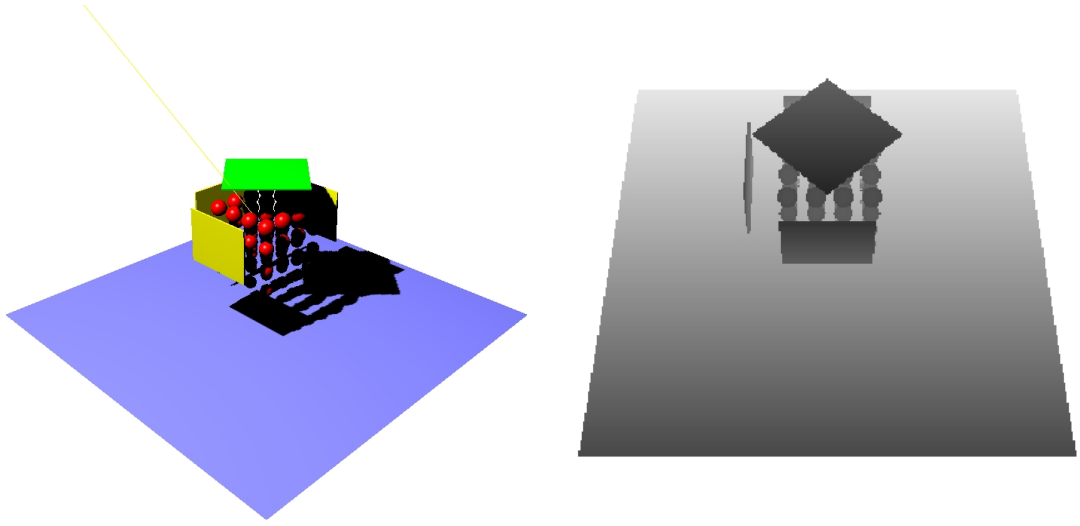
   (a) Převeď bod P_C se souřadnicemi $[x_C, y_C, z_C]$ z pohledu kamery C do pohledu světla L a získej tak nový bod P_L se souřadnicemi $[x_L, y_L, z_L]$.

   (b) Získej hodnotu $z_D = D[x_L, y_L]$.

   (c) Pokud $(z_L > z_D)$, pak je bod P_L zastíněn jiným předmětem a proto i bod P_C leží ve stínu.

Pokud máme více než jeden světelný zdroj, musíme vygenerovat hloubkovou mapu pro každý z nich. Kroky 2a až 2c potom provádíme pro každý světelný zdroj a jeho hloubkovou mapu. Nejlépe metoda funguje pro směrové zdroje ideálně umístěné v nekonečnu, jelikož hloubková mapa může obsáhnout pouze část scény. Na obrázku 5.4(a) můžeme pro ilustraci vidět scénu z pohledu kamery a obrázek 5.4(b) potom ukazuje odpovídající hloubkovou mapu z pohledu světla.

V případě, že máme všesměrový světelný zdroj, musíme vygenerovat až šest hloubkových map, neboli stěn pomyslné krychle obklopující tento zdroj, což by vyžadovalo až šest vykreslovacích průchodů. Jsou však známy i efektivnější přístupy. Algoritmus generování



(a) (b)

Obrázek 5.4: Scéna z pohledu kamery a odpovídající hloubková mapa z pohledu světla

všesměrových stínových map pomocí takzvaných *cube-maps* navrhnul *Gerasimov* ve svém článku [Ger04]. Řešení tohoto problému pomocí dvou stínových map prezentoval *Brabec* [BAS02].

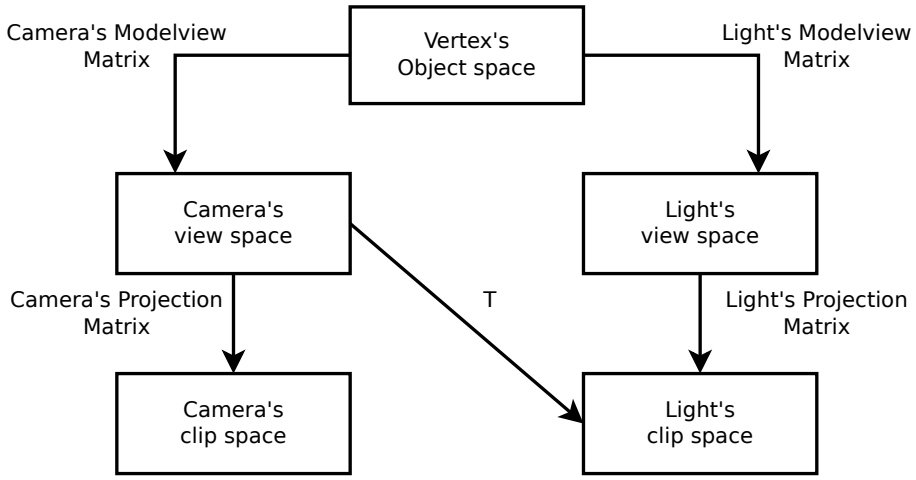
5.2.3 Implementace

Pokud bychom algoritmus implementovali pouze za použití fixního zobrazovacího řetězce, vyžadoval by (v případě jednoho světelného zdroje) tři průchody[2]. První průchod pro vygenerování hloubkové mapy, druhý průchod pro získání stínové mapy a třetí pro vykreslení scény bez stínů a aplikaci stínové mapy. Při aplikaci stínové mapy na scénu lze použít metodu mapování textur [SKW+92].

Díky možnostem programovatelného zobrazovacího řetězce nám však stačí průchody dva, kdy vykreslení scény spojíme se získáním a aplikací stínové mapy do průchodu jednoho. Algoritmus je tedy vhodným adeptem na implementaci na grafické kartě. S odkazem na kapitolu 4.3 nyní naznačíme, jak lze tento algoritmus implementovat s využitím možností v této kapitole popsaných.

K uložení hloubkové mapy můžeme použít FBO a techniku *render to texture* (viz kapitola 4.3.2.1), přičemž ukládáme pouze informaci o hloubce z pohledu světla. V dalším průchodu vykreslujeme scénu z pohledu kamery, přičemž použijeme *vertex* a *fragment shader*. Ve *vertex shaderu* vypočteme texturovací souřadnice do hloubkové mapy, přičemž transformaci T,

[2]Pokud grafický hardware podporuje hloubkové textury a hloubkový test při procesu texturování, lze tento algoritmus za použití fixního zobrazovacího řetězce implementovat i ve dvou průchodech.



Obrázek 5.5: Transformace z pohledového prostoru kamery do ořezového prostoru světla

kterou v tomto případě hledáme, ilustruje obrázek 5.5. Ve *fragment shaderu* poté provedeme porovnání hloubky transformovaného vrcholu, respektive jeho souřadnice z, s hodnotou v hloubkové mapě. To lze provést buď manuálním porovnánám hodnoty z hloubkové textury s touto souřadnicí, nebo lze použít speciální hardwarové porovnání pomocí funkce `shadow2D(Proj)`[3].

5.2.4 Rasterizace a práce v diskretizovaném prostoru

Nyní zdánlivě odbočíme a povíme si něco blíže k rasterizaci a jejích výhodách a nevýhodách. Jak však záhy poznáme, tato vsuvka se dotýká jak metody stínových map, tak i našeho algoritmu pohledově závislých stínových map.

Proces vykreslování na grafických kartách je od jejich počátku (který můžeme datovat od uvedení GeForce 256) až dodnes založen na základním principu, kterému říkáme rasterizace. Jedná se o proces, kdy po transformaci objektu do dvourozměrného prostoru, například na obrazovku monitoru, určujeme, které pixely jsou tímto objektem pokryty a které ne. Rasterizace objektů v OpenGL spočívá v převedení těchto objektů, respektive grafických primitiv, z nichž jsou složeny, na takzvané fragmenty, se kterými jsme seznámili již v kapitole 4.2.2. Jedná se o části grafického primitiva odpovídající velikostí jednomu pixelu, které si s sebou nesou různé hodnoty jednoduše interpolované z vrcholů původního primitiva.

Tento přístup nám, respektive části zobrazovacího řetězce nacházející se za rasterizační jednotkou, umožňuje zapomenout, že existují nějaká grafická primitiva a dovoluje nám dále

[3] K tomu, aby tato funkce prováděla hardwarové porovnání hloubky, musíme mít v OpenGL nastaven porovnávací režim hloubkové textury GL_TEXTURE_COMPARE_MODE na hodnotu GL_COMPARE_R_TO_TEXTURE.

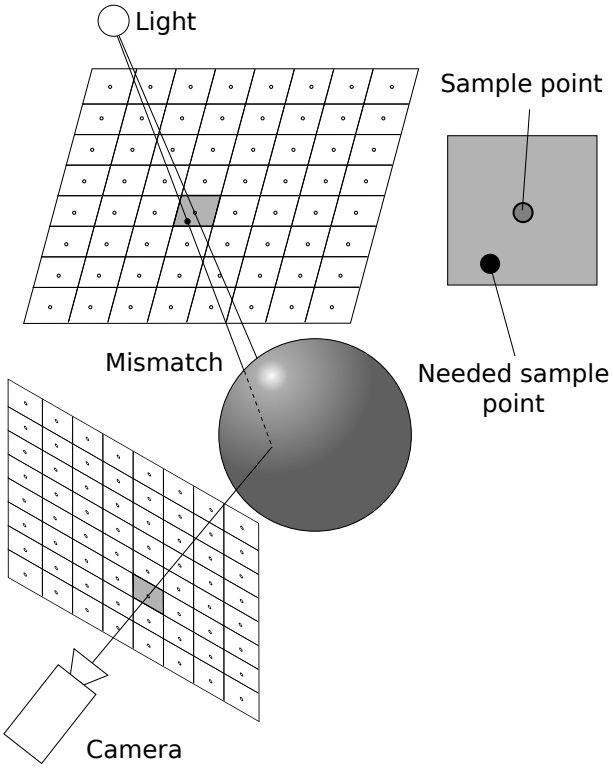
zpracovávat jednotlivé fragmenty zvlášť, čímž se celý proces vykreslování značně zjednodušuje. Navíc je tím umožněno i paralelní zpracování, které dává grafickým kartám jejich příznačnou rychlost. Vzhledem k tomu, že jedno grafické primitivum obvykle pokrývá více než jeden pixel, bývá v dnešních grafických kartách více jednotek paralelně zpracovávajících fragmenty, než jednotek pro zpracování vrcholů.

Rasterizace má však i své nevýhody. Jednou z nich je, že grafická karta dokáže vzorkovat pouze v předem daných bodech, jimiž jsou přesné středy pixelů. Pokud tedy objekt sice zasahuje svojí hranou do daného pixelu, ale nepokrývá jeho střed, nebude pro něj při rasterizaci vygenerován odpovídající fragment. Další nevýhodou je určitá ztráta informace způsobená kvantováním hodnot, jelikož máme k dispozici pouze omezený rozsah přesnosti, se kterou můžeme hodnoty jednotlivých fragmentů ukládat. V praxi si s sebou každý fragment nese například informaci o své hloubce v trojrozměrném prostoru, a tato hloubka je v OpenGL kvantována dokonce nelineárně, tedy s klesající přesností při rostoucí vzdálenosti. To může působit problémy při porovnávání těchto hodnot s hodnotami, které jsou kvantovány s konstantní přesností. Další z významných omezení pak ještě může být, že pozici fragmentu vzniklého rasterizací nelze již měnit – je dána pozicí pixelu, který rasterizovaný objekt pokrýval. Pokud bychom mohli pro každý fragment určit, na jakou pozici se má zapsat, odpadlo by nám dozajista mnoho starostí.

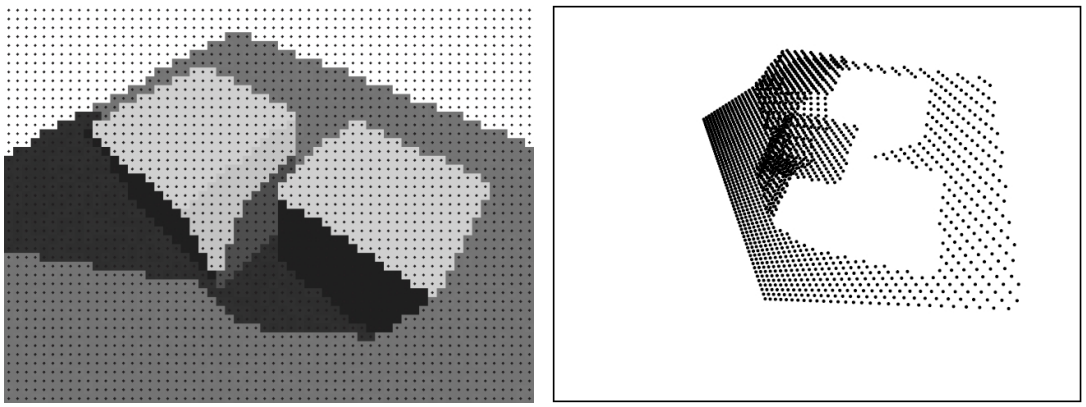
Co se týče metody stínových map, jsou důsledky výše uvedených nevýhod a omezení zřejmé. Kvůli nemožnosti vzorkovat v libovolných bodech, ale pouze ve středech pixelů, dochází k neshodám ve vzorkování z pohledu kamery a světla, což vyústí v nepřesné vykreslení hran generovaných stínů – takzvaný *aliasing*. Tyto neshody jsou ilustrovány na obrázku 5.6. Vidíme, že pro daný pixel obrazovky, pro který testujeme zastínění, bychom potřebovali porovnávat hloubku s jiným vzorkem, než tím, který nám nabízí hloubková mapa z pohledu světla. Pokud bychom mohli vzorkovat v libovolných bodech, dokázali bychom pomocí této jednoduché metody počítat stíny přesné.

Na obrázku 5.7 vidíme, jak může ve skutečnosti vypadat rozložení vzorků viditelných z kamery v pohledu ze světla. Při vykreslování scény z pohledu světla bychom potřebovali vzorkovat právě v místech, která jsou znázorněna na druhém z obrázků (5.7(b)). Vidíme však, že rozložení těchto vzorků je velice nepravidelné – vzorky se dokonce mohou i překrývat – a s tím si grafická karta zkrátka poradit nedokáže.

O shodu ve vzorkování mezi kamerou a světlem, která by umožňovala vykreslování přesných stínů pomocí metody stínových map, se již pokoušeli například *Aila* [AL04] a *Johnson* [JMB04]. Jejich řešení spočívalo principielně právě v získání viditelných vzorků, reprezentovaných jednotlivými body na obrázku 5.7(b), a právě tyto používali při vykreslování stínů k porovnání hloubek. Tím byli schopni pomocí jednoduché metody stínových map vykreslovat přesné stíny podobně, jako má za cíl i tato práce. Narozdíl od našeho algoritmu však



Obrázek 5.6: Neshoda ve vzorkování z pohledu kamery a světla v metodě stínových map



(a) (b)

Obrázek 5.7: Viditelná scéna z pohledu kamery (a) a odpovídající vzorky transformované do pohledu světla (b) (obrázek převzat z [AL04])

k tomu potřebovali složitější vyhledávací struktury – BSP stromy v případě *Aily*, hybridní mřížky v případě *Johnsona* – do kterých vzorky ukládali, a jejich algoritmy tak bylo velmi složité namapovat na grafickou kartu. S rozvojem grafického hardware se však v dnešní době objevují pokusy toto omezení, tedy schopnost grafické karty vzorkovat pouze ve středech pixelů, prolomit a obejít, a to pouze s využitím možností nových grafických karet. O těchto pokusech se zmíníme v úvodu následující kapitoly 6.1.

Náš algoritmus pohledově stínových map je potom jedním z těchto pokusů. K dosažení tohoto cíle – tedy vzorkování v libovolných bodech – nepotřebuje žádné složité struktury, jako výše zmíněné metody. Pracuje tak pouze s využitím schopností dnešních grafických karet.

5.2.5 Metody eliminace artefaktů

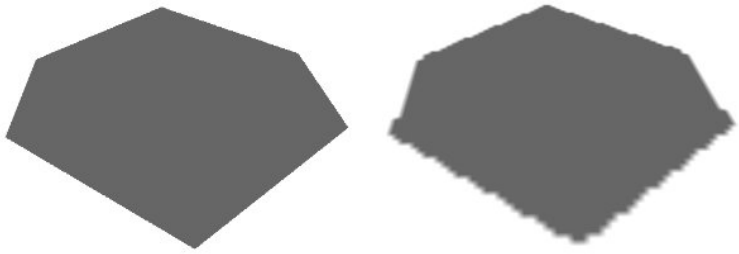
Nyní se ještě vrátíme k metodě stínových map a seznámíme se s technikami, které se používají k odstranění nepřesností na hranách generovaných stínů (*aliasing*) a sebezastínění (*self-shadowing*), jež jsou vizuálně nejpatrnějšími nedostatky této metody. Poté se již zaměříme na samotný algoritmus pohledově závislých stínových map.

5.2.5.1 Nepřesnosti na hranicích stínů

Jak již bylo řečeno, metoda stínových map je ve své základní podobě velmi citlivá na takzvaný *aliasing*, neboli nepřesnosti na hranicích stínů, což je dáno hlavně omezeným rozlišením hloubkové mapy a přístupem do ní. Zvláště patrné jsou tyto nepřesnosti, pokud se výrazně liší rozlišení obrazu z pohledu kamery a světla. Tato citlivost je potom nepřímo úměrná rozlišení (velikosti) hloubkové mapy, kdy s rostoucím rozlišením *aliasing* klesá. Nepřesnosti na hranicích generovaných stínů můžeme pozorovat na obrázku 5.8.

V ideálním případě bychom potřebovali, aby plocha scény, kterou pokrývá jeden pixel na obrazovce, odpovídala přesně jednomu pixelu v hloubkové mapě. V praxi však toto nefunguje a pixel z hloubkové mapy může pokrývat i více než jeden pixel v prostoru obrazovky, čímž vznikají nepřesnosti. Řešení navrhnuté v [RSC87] nepoužívá k vyhodnocení zastínění pouze jednu hodnotu z hloubkové mapy, nýbrž i několik sousedních. S každou takovou hodnotou se provede test zastínění a výsledek se zprůměruje, čímž vlastně *aliasing* principielně neodstraníme, ale jeho projevy převedeme na šum, na nějž je lidské oko méně citlivé. Touto metodou tedy dosáhneme takzvaného *anti-aliasingu* a navíc získáme i (falešný) dojem měkkého stínu.

Další z metod na snížení aliasingu, označovaná jako metoda perspektivních stínových map (*perspective shadow maps*, PSM) [SD02], spočívá v generování stínových map v normalizovaných souřadnicích, tedy až po perspektivním promítání, čímž lze znatelně zvýšit rozlišení objektů v blízkosti ke středu promítání a snížit rozlišení vzdálenějších objektů.



Obrázek 5.8: Vržený stín bez aliasingu (vlevo) a s aliasingem (vpravo)

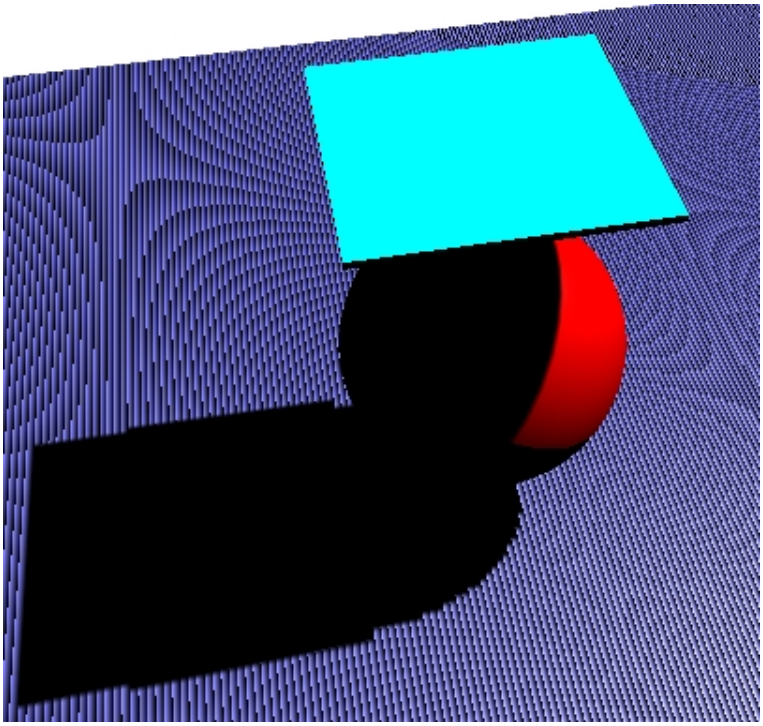
Jinými slovy, vytvářením hloubkové mapy až po perspektivnám promítání metoda PSM zajišťuje, že objekty blíže ke kameře, tedy objekty zabírající více pixelů na obrazovce, budou zabírat více pixelů i v hloubkové mapě. Metoda se tedy snaží přiblížit ideálnímu případu, kdy plocha pokrytá jedním pixelem na obrazovce odpovídá jednomu pixelu v hloubkové mapě. Rozšířeními této metody, snažícími se o odstranění některých jejích nedostatků, jsou pak například perspektivní stínové mapy v prostoru světla (*light-space perspective shadow maps*) [WSP04] nebo logaritmické perspektivní stínové mapy (*logarithmic perspective shadow maps*) [LGQ+08].

Podobnou myšlenku, tedy zvýšení rozlišení hloubkové mapy pro objekty blízko pozorovatele a snížení rozlišení pro objekty vzdálené, používá metoda *cascaded shadow maps* [Dim07]. Ta toho však dosahuje rozdělením pohledového jehlanu a vytvořením hloubkových map v jeho jednotlivých řezech. Metoda je vhodná pro generování stínů v rozsáhlých (většinou venkovních) scénách.

Žádná z výše uvedených metod však z principu neodstraňuje nepřesnosti na hranicích generovaných stínů úplně. Zvyšování rozlišení hloubkových map či generování jejich většího počtu sice může přinést lokální zlepšení ve vizuální kvalitě vykreslovaných scén, žádná z nich však nemůže zaručit, že rozlišení bude vždy dostačující. Narozdíl od metody pohledově závislých stínových map, prezentované v této práci, žádná z těchto metod neodstraňuje *aliasing* z principu, a tedy za všech okolností a úplně.

5.2.5.2 Self-shadowing

Dalším artefaktem, který vzniká při použití metody stínových map, je takzvaný *self-shadowing*, neboli „sebezastínění“. To vzniká v důsledku numerických nepřesností daných kvantováním ukládaných hodnot, kdy se hloubka transformovaného bodu nemusí přesně shodovat s hodnotou v hloubkové mapě, i kdyby se jednalo přesně o tentýž bod v prostoru. Pokud je hodnota v hloubkové mapě nepatrně nižší, může se zastínení bodu chybně vyhodnotit tak, že bod vrhá stín sám na sebe. Tento problém ještě zveličuje fakt, že OpenGL hodnoty



Obrázek 5.9: Self-shadowing

hloubky kvantuje nelineárně – přesnost tedy klesá s rostoucí hloubkou. Příklad sebezastínění můžeme vidět na obrázku 5.9.

Intuitivním řešením tohoto problému může být odečtení malé konstanty od hodnoty transformované souřadnice z a tedy její přiblížení směrem ke světlu. Podobného efektu lze dosáhnout oddálením celé hloubkové mapy o konstantu (k tomu nám OpenGL poskytuje metodu `glPolygonOffset`). Určit velikost této konstanty je však složitější problém. Pokud zvolíme příliš malou hodnotu, artefakty se budou stále objevovat a pokud zvolíme hodnotu příliš vysokou, mohou být některé plochy vyhodnoceny jako osvětlené, i když by správně měly ležet ve stínu.

Řešením tohoto problému může být neukládat do houbkové mapy vzdálenost nejbližšího bodu ve scéně, ale průměr ze dvou nejbližších vzdáleností bodů spadajících do jednoho pixelu mapy. Tím dokážeme na plochách objektů eliminovat *self-shadowing*, protože hodnota v hloubkové mapě bude vždy větší, a zároveň nám zůstanou v mapě korektní hodnoty pro hloubkový test s potenciálně zastíněnými objekty. I tato metoda, prezentovaná v [Woo92], má však své nedostatky. Konkrétně se jedná o neodstraněné artefakty v blízkosti hran objektů, což je způsobeno diskretizací při vytváření hloubkové mapy – pokud hrana objektu při rasterizaci zasahuje do pixelu této mapy, ale nepokrývá jeho střed, do mapy se neuloží a její hloubka se tak do průměru nezapočítá.

Podobným řešením, bohužel i se stejnými nedostatky, může být ukládat do hloubkové mapy pouze plochy odvrácené od světla. Tím zajistíme, že hodnoty v hloubkové mapě budou vždy větší, takže se *self-shadowing* nebude tolik projevovat, a zároveň v mapě zůstanou správné hodnoty pro hloubkový test. Řešení samozřejmě předpokládá uzavřenost objektů.

Jak již bylo řečeno v kapitole předchozí, námi prezentovaný algoritmus generování ostrých stínů ze své podstaty netrpí žádným z těchto problémů. Pracuje totiž na jiném principu, o němž si povíme v následujících kapitolách.

Kapitola 6

Pohledově závislé stínové mapy

6.1 Úvod a motivace

Výpočet přesných stínů ve scéně je speciálním případem řešení viditelnosti. Když řešíme, zda jsou dvě oblasti vzájemně viditelné, musíme určit, zda mezi nimi neleží jiné těleso. V potaz přitom musíme brát velikost testovaných oblastí – testovat vzájemnou polohu dvou bodů je dozajista jednodušší, než testovat polohu dvou polygonů. Při testování polohy dvou bodů nám stačí vypočítat, zda paprsek vržený z jednoho bodu do druhého protíná jiné těleso či nikoliv. Pokud bychom chtěli přesně určit to samé pro polygony, museli bychom vrhnout paprsek z každého bodu jednoho polygonu do každého bodu druhého polygonu, což v případě spojitého prostoru teoreticky vyústí v nekonečný počet paprsků.

Přesný výpočet viditelnosti mezi dvěma povrchy je sice náročný, nikoliv však nemožný. Tento problém lze řešit například pomocí geometrických operací nad vícerozměrnými datovými strukturami propojujícími plošky jednotlivých objektů ve scéně [NBG02], algoritmy pracujícími s kostrami mnohostěnů [HMN05], případně dalšími, obecně však výpočetně náročnými metodami. Pro zjednodušení tohoto problému se proto často používá takzvané vzorkování – *sampling*. To spočívá v získání reprezentativních vzorků – jednotlivých bodů – z obou polygonů, mezi kterými chceme viditelnost určit, a provést test viditelnosti pouze mezi nimi. Problémem je samozřejmě určit počet a rozložení těchto vzorků, přičemž volíme mezi výpočetní náročností a kvalitou výsledného obrazu. Kvalita samozřejmě roste s počtem vzorků, avšak za cenu zvýšené výpočetní náročnosti.

Metoda vzorkování je známa již dlouho. Až donedávna však nebylo možné určit množiny reprezentativních vzorků pouze s využitím grafické karty, a vzorkovat tak přímo na ní. S příchodem geometrického procesoru (viz kapitola 4.3.1) se však situace mění a nám je tak dána možnost přímo na GPU určit oblast, kde se má vzorkovat – můžeme tak teoreticky vzorkovat v libovolných bodech.

Vzorkování na grafické kartě je tedy poměrně mladá disciplína, jelikož tyto možnosti nebyly až donedávna dostupné. Testování viditelnosti pomocí vzorkování na GPU se věnují ve své práci například *Eisemann* a *Décoret* [ED07]. Generováním stínů, konkrétně potom měkkých, založeným na vzorkování na GPU, je věnován například článek [SEA08].

Jak již bylo řečeno, výpočet přesných stínů ve scéně je specifickou variací na řešení viditelnosti. První množina vzorků je v tomto případě reprezentována vzorky na jednotlivých polygonech a druhá množina je množina bodových světelných zdrojů. Pro výpočet zastínění musíme pro každou dvojici z těchto vzorků určit, zda úsečka, jejímiž koncovými body jsou tyto vzorky, protíná některý z polygonů ve scéně. Pokud ano, je testovaný vzorek z první množiny ve stínu, pokud ne, je osvětlen. Obecně tedy implementace výpočtu viditelnosti mezi dvěma množinami vzorků a polygony ve scéně vyžaduje tři vnořené cykly, a díky možnostem dnešních grafických karet lze právě takový výpočet efektivně namapovat přímo na grafickou kartu.

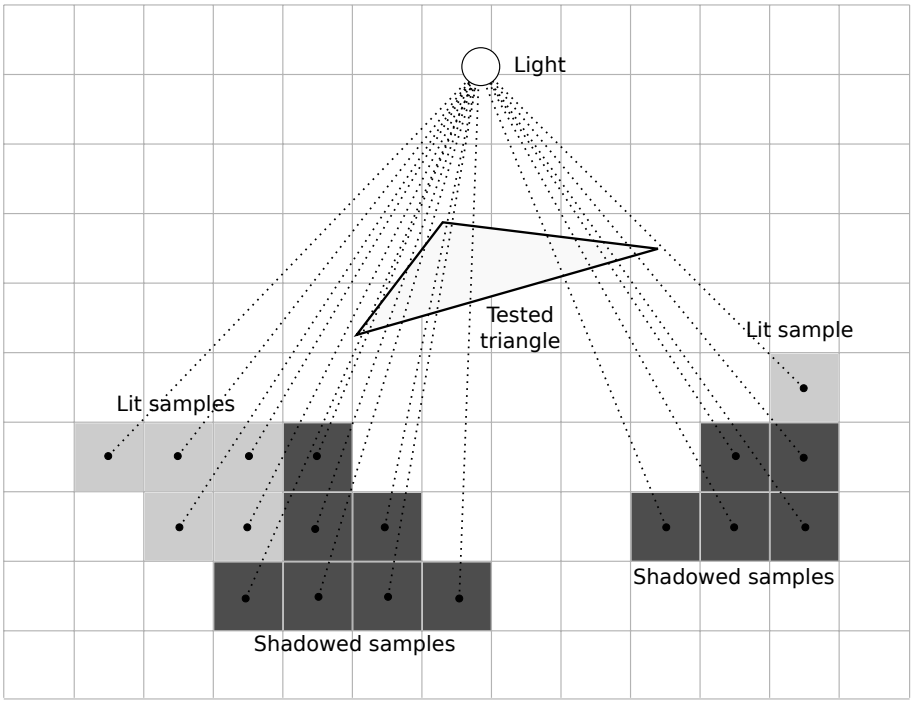
Algoritmus pohledově závislých stínových map tedy využívá principu vzorkování na GPU k výpočtu ostrých stínů ve scéně. Výběr množiny vzorků, kde se má výpočet zastínění provádět, potom umožňuje právě geometrický procesor, na který algoritmus spoléhá.

6.2 Základní implementace algoritmu

Jak již bylo zmíněno, algoritmus pohledově závislých stínových map není v podstatě nic jiného, než řešení viditelnosti mezi množinou bodů získaných pomocí vzorkování na GPU a množinou světelných zdrojů, přičemž zjišťujeme, zda na cestě od těchto bodů směrem ke světelným zdrojům neleží v cestě další polygon. Tento princip ilustruje obrázek 6.1. Algoritmus lze proto implementovat ve třech krocích.

V prvním kroku určíme množinu vzorků – tedy souřadnice bodů, se kterými bude prováděn test viditelnosti vzhledem ke světlům ve scéně. Ve druhém kroku určíme podmnožinu z těchto vzorků, nad kterou budeme výpočet viditelnosti, respektive zastínění, provádět. Výběr těchto vzorků je právě stěžejní částí algoritmu a bez geometrického procesoru by ho nebylo možné implementovat na grafické kartě. Třetím krokem je pak samotný výpočet zastínění, čímž vznikne stínová mapa. Tyto tři kroky vyžadují dva vykreslovací průchody. Ve třetím průchodu vykreslíme scénu normálně bez stínů, přičemž vzniklou stínovou mapu aplikujeme.

V následujících podkapitolách se budeme věnovat konkrétní implementaci algoritmu, tedy jeho třech kroků. Podotkněme jen, že algoritmus tak, jak byl implementován, neklade žádné nároky na reprezentaci scény – pracuje pouze s trojúhelníky a nevyžaduje tedy žádnou další znalost, jako například informaci o sousednosti trojúhelníků (*triangle adjacency*).



Obrázek 6.1: Princip výpočtu zastínění metodou vzorkování na GPU

6.2.1 Získání množiny vzorků - 1. krok

První množinou vzorků účastnících se testu viditelnosti je intuitivně množina bodových světelných zdrojů, které jsou známy již před spuštěním samotného algoritmu. Zbývá tak určit množinu druhou. Ta bude obsahovat vzorky těch ploch, na které může potenciálně dopadat vržený či vlastní stín[1]. Těmito plochami mohou být obecně veškeré polygony ve scéně, relevantními jsou však pouze ty, které jsou viditelné z pohledu kamery. Vzhledem k této skutečnosti můžeme souřadnice vzorků získat při rasterizaci těchto polygonů, kdy budeme namísto barvy fragmentu ukládat jejich interpolované souřadnice. V našem případě je budeme s ohledem na další části algoritmu ukládat v pohledovém prostoru kamery.

Toho lze dosáhnout velmi jednoduše za použití *vertex* a *fragment shaderu* společně s technikou *render to texture* (viz kapitola 4.3). Ve *vertex shaderu* transformujeme příchozí vrcholy do pohledového prostoru kamery a tyto souřadnice posíláme do *fragment shaderu*. Pozici vrcholů přitom převádíme do ořezového prostoru kamery, aby se provedla správně rasterizace. Ve *fragment shaderu* zapisujeme namísto barvy fragmentů nyní již interpolované souřadnice vrcholů v pohledovém prostoru kamery a výsledek ukládáme do předem připravené textury, jež má stejnou velikost jako okno. Dnešní GPU nám pro tyto účely nabízejí

[1] Vlastním stínem nyní myslíme stín vržený jedním polygonem na druhý, avšak v rámci jednoho tělesa.

širokou škálu formátů, díky kterým lze v každém texelu textury uchovávat jedno- až čtyřsložkový vektor v plovoucí řádové čárce, a to až s 32-bitovou přesností. V naší implementaci používáme formát GL_RGBA32F. Na konci tohoto průchodu máme tedy texturu obsahující všechny viditelné body, přičemž v každém jejím texelu je uložena souřadnice bodu v pohledovém prostoru kamery, který do tohoto texelu spadá. Na tuto část algoritmu se budeme ve zbytku textu odkazovat jako na *první krok*, neboli také krok generování *textury vzorků*.

6.2.2 Výběr vzorků k testu zastínění - 2. krok

Nyní se musíme zamyslet, jak realizovat výpočet zastínění na vzorcích, které máme po prvním kroku uložené v textuře. Intuitivní řešení nabízí algoritmus, který pro každý vzorek otestuje, zda na spojnici tohoto vzorku se světlem (pro jednoduchost uvažujeme pouze jeden světelný zdroj) neleží některý z polygonů ve scéně a zda je daný vzorek tímto polygonem zastíněn. Takovýto algoritmus bude vyžadovat dva cykly – jeden přes všechny vzorky, druhý přes všechny polygony. Pořadí těchto cyklů je zaměnitelné, a tak lze tento algoritmus zapsat i následujícím pseudo-kódem:

```
for (each polygon P)
{
        for (each visible point V)
        {
                Line L = create_line(V, Light);
                ShadowMap[V] = is_collision(L, P) ? 0 : 1;
        }
}
```

Implementovat tento algoritmus tak, aby běžel sekvenčně na hlavním procesoru, by jistě nebyl žádný problém. My však tento algoritmus chceme provádět na grafické kartě, a to pokud možno s využitím paralelismu, který je grafickým kartám vlastní. Zamyslíme se tudíž, jak jednotlivé části algoritmu s ohledem na tento fakt co nejefektivněji realizovat.

První částí je cyklus přes všechny polygony, respektive trojúhelníky, ve scéně, což lze realizovat prostým posláním veškeré geometrie, od které očekáváme, že bude vrhat stín, do grafického zobrazovacího řetězce. Následuje druhý cyklus, a tím je spouštění testu zastínění pro každý z viditelných vzorků, které máme po prvním kroku uloženy v textuře. Test zastínění, který je blíže popsán v následující kapitole, budeme provádět ve *fragment shaderu*. Za předpokladu, že má textura s viditelnými vzorky rozlišení MxN pixelů, tedy musíme provést MxN těchto testů. Tyto testy lze sice provést sekvenčně, ale díky možnostem dnešních grafických karet, konkrétně pak přítomnosti geometrického procesoru v grafickém zobrazovacím řetězci, jsme schopni tento proces paralelizovat a využít tak plného potenciálu grafické karty. Paralelismu dosáhneme tím, že pro každý trojúhelník vstupující do geometrického procesoru

vygenerujeme obdélník o velikosti MxN pixelů – tedy přes celou obrazovku – a pošleme jej spolu s informacemi o původním trojúhelníku dále do fragment procesoru. Díky tomu se pak program prováděný na tomto procesoru, tedy *fragment shader*, spustí pro každý trojúhelník právě MxN krát. V každé instanci tohoto programu pak provedeme právě jeden test na průsečík původního trojúhelníku se spojnicí světla a bodu z textury vzorků, jehož souřadnice odpovídají souřadnicím fragmentu v rastru.

Tím jsme tedy dosáhli našeho cíle – pro každý trojúhelník ve scéně se nám provede test zastínění každého vzorku z textury vytvořené v prvním kroku. Navíc použitím geometrického procesoru plně využíváme paralelismu, který grafické kartě umožňuje pracovat rychleji.

Samotná realizace tohoto kroku probíhá následovně. Ve *vertex shaderu* pouze transformujeme každý vrchol do pohledového prostoru kamery. Na vstupu *geometry shaderu* pak máme jednotlivé trojúhelníky, přičemž místo každého z nich generujeme obdelník přes celou obrazovku, který realizujeme jako pás trojúhelníků (*triangle strip*). Ke každému z jeho čtyř vrcholů, jejichž souřadnice mají tedy rozsah od -1 do 1 v osách X a Y, navíc spočítáme texturovací souřadnici, na jejímž základě se poté ve *fragment shaderu* vybere vzorek z textury, který bude příslušný fragment vzniklý rasterizací námi generovaného obdélníku pokrývat. Do *fragment shaderu* si zároveň s každým vrcholem obdélníku posíláme ve *varying* proměnných i všechny souřadnice původního trojúhelníku, abychom s ním následně byli schopni provést test na průsečík. Implementace *geometry shaderu* je ukázána na příkladu 6.1.

```
varying out vec4 texCoord;
varying out vec4 tVec[3];

void main(void)
{
  // souradnice vrcholu puvodniho trojuhelniku ulozime
  tVec[0] = gl_PositionIn[0];
  tVec[1] = gl_PositionIn[1];
  tVec[2] = gl_PositionIn[2];

  // nyni vytvorime ctyri vrcholy obdelniku pres celou obrazovku
  // jako pas trojuhelniku a k nim texturovaci souradnice

  // vytvorime prvni vrchol
  gl_Position = vec4(-1.0, -1.0, 0.0, 1.0);
  texCoord = vec4(0.0, 0.0, 0.0, 1.0);
  EmitVertex();

  // vytvorime druhy vrchol
  gl_Position = vec4(1.0, -1.0, 0.0, 1.0);
  texCoord = vec4(1.0, 0.0, 0.0, 1.0);
  EmitVertex();
```

```
  // vytvorime treti vrchol
  gl_Position = vec4(-1.0, 1.0, 0.0, 1.0);
  texCoord = vec4(0.0, 1.0, 0.0, 1.0);
  EmitVertex();

  // vytvorime ctvrty vrchol
  gl_Position = vec4(1.0, 1.0, 0.0, 1.0);
  texCoord = vec4(1.0, 1.0, 0.0, 1.0);
  EmitVertex();

  EndPrimitive();
}
```

Příklad 6.1: Generování obdélníku přes celou obrazovku v *geometry shaderu*

6.2.3 Test zastínění - 3. krok

Díky tomu, že jsme v předchozím kroku zajistili spuštění *fragment shaderu* pro každý bod obrazovky, jsme nyní schopni z textury vzorků získat texel na pozici aktuálně zpracovávaného fragmentu, jehož souřadnice jsme obdrželi z *geometry shaderu*. Každý takový texel tedy obsahuje buď souřadnice bodu scény v pohledovém prostoru kamery, nebo je v něm uložena souřadnice neplatná, tedy taková, kterou nemůže žádný ze vzorků viditelných z kamery mít. Pokud jsme tedy získali reálný bod scény, budeme testovat jeho viditelnost směrem ke světelnému zdroji. V opačném případě můžeme instanci programu ukončit a nezapisovat do fragmentu žádnou hodnotu – k tomu lze použít klíčové slovo `discard`, specifické pouze pro *fragment shader*.

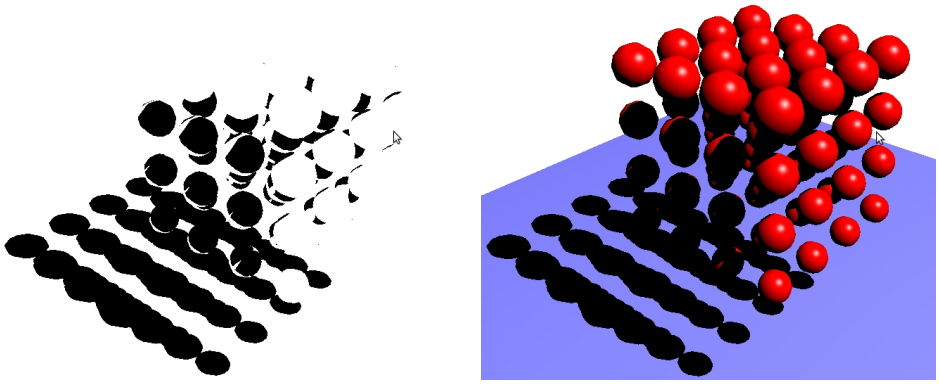
Pokud přistoupíme k testu viditelnosti, zrekonstruujeme ve *fragment shaderu* ze souřadnic vrcholů předaných *geometry shaderem* původní trojúhelník. Poté provedeme test na průsečík tohoto trojúhelníku se spojnicí dříve získaného bodu z textury vzorků se světlem. Důraz je kladen samozřejmě na rychlost výpočtu, a proto je vhodné použít co nejefektivnější algoritmus pro nalezení průsečíku úsečky s trojúhelníkem, například *Moller-Trumborův* [MT97]. Pokud je průsečík nalezen, leží bod ve stínu a do fragmentu zapíšeme příznak zastínění. Pokud průsečík nalezen není, můžeme program ukončit, aniž bychom do fragmentu cokoliv zapisovali (`discard`). Výše popsanou implementaci *fragment shaderu* můžeme vidět na příkladu 6.2.

```
uniform sampler2D coordMap;
uniform float offset;
varying vec4 texCoord;
varying vec4 tVec[3];

void main()
{
        // ziskame texel z textury vzorku
        vec4 point = texture2DProj(coordMap, texCoord);

        // pokud je v nem ulozena neplatna hodnota
        // (v nasem pripade 1), koncime test
        if (point.xyz == vec3(1.0)) {
                discard;
        }

        // ziskame usecku spojujici svetlo se vzorkem
        vec3 lightDir = gl_LightSource[0].position.xyz - point.xyz;

        // testujeme, zda puvodni trojuhelnik lezi na teto usecce
        bool intersection = intersectionTest(point.xyz, lightDir.xyz,
                        tVec[0].xyz, tVec[1].xyz, tVec[2].xyz);

        // pokud ano, zapiseme priznak stinu, jinak ukoncime program
        if (intersection) {
                gl_FragColor = SHADOW;
        }
        else {
                discard;
        }

}
```

Příklad 6.2: Test zastínění ve *fragment shaderu*

Pro každý trojúhelník ve scéně nám tedy vznikne binární mapa zastínění, kde mohou například černé oblasti reprezentovat zastíněné body a bílé oblasti nezastíněné. Sjednocení těchto map, které ukazuje obrázek 6.2(a), nám tedy poskytne úplnou stínovou mapu. Tohoto sjednocení dosáhneme zapnutím „minimalizačního“ míchání barev (*alpha blending*), kdy do textury ukládáme hodnotu fragmentu pouze tehdy, pokud je menší než hodnota v textuře již uložená. Na tuto část algoritmu se budeme dále v textu odkazovat jako na *třetí krok* algoritmu, čili krok s *testem zastínění*.

Ve třetím průchodu znovu vykreslíme celou scénu bez stínů, tentokrát již standardně se zapnutým osvětlením i veškerými požadovanými efekty. Vytvořenou stínovou mapu poté opět se zapnutým mícháním barev, tentokrát multiplikativním, naneseme na dvourozměrný



(a) (b)

Obrázek 6.2: Stínová mapa vygenerovaná algoritmem pohledově závislých stínových map

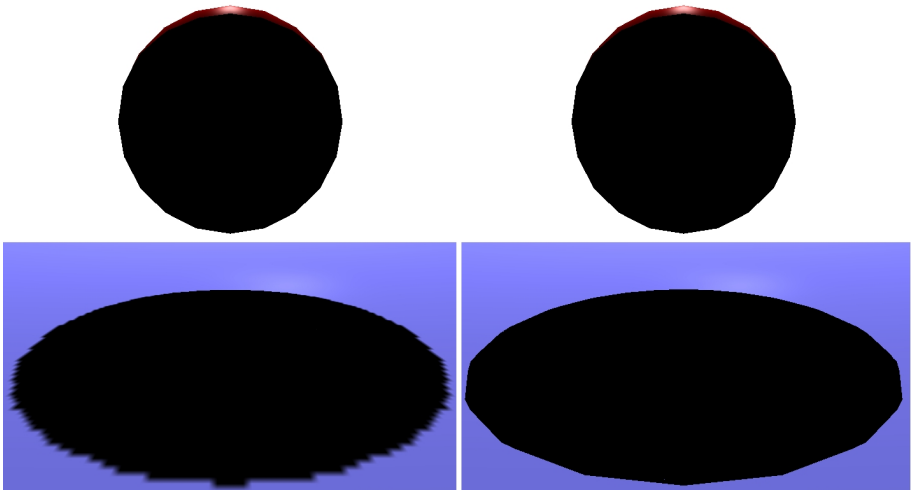
obdélník pokrývající celou obrazovku. Výsledek ukazuje obrázek 6.2(b). Srovnání tohoto algoritmu s metodou stínových map ilustruje obrázek 6.3.

6.3 Nedostatky základní implementace

Algoritmus, který jsme se v této kapitole snažili popsat, přistupuje k řešení problému v podstatě hrubou silou. Ta spočívá ve skutečnosti, že zastínění jedním polygonem testujeme vždy se všemi viditelnými vzorky – zdaleka ne vždy však bude polygon vrhat stín na všechny tyto vzorky. V případě, že plocha, na kterou může trojúhelník potenciálně vrhat stín, zabírá například pouze 10% všech pixelů obrazovky, provádíme devět z deseti testů na průsečík naprosto zbytečně.

Optimální by samozřejmě bylo provádět tento test pouze na těch částech scény, kde se stín vržený daným polygonem může potenciálně nacházet. Jinak řečeno bychom potřebovali určit takovou oblast, která co nejtěsněji pokrývá veškeré potenciálně zastíněné vzorky, a tedy spouštět *fragment shader*, respektive test zastínění, jen a pouze na těchto vzorcích. Určení takovéto oblasti je však při obecně dané geometrii scény netriviální problém. Vzhledem k požadavku na vysokou rychlost algoritmu tedy nebudeme tuto oblast počítat přesně, nýbrž použijeme konzervativní odhad.

Následující kapitoly se proto věnují optimalizacím zaměřeným zejména na druhý krok algoritmu, tedy na zredukování obdélníku – obecně potom oblasti doposud generované přes celou obrazovku, jež má za následek spouštění testu zastínění na všech viditelných vzorcích.



Obrázek 6.3: Srovnání vizuální kvality stínů generovaných metodou stínových map (vlevo) s metodou pohledově závislých stínových map (vpravo)

Většina těchto optimalizací se tedy bude odehrávat právě v *geometry shaderu*, který je za generování této oblasti zodpovědný.

Kapitola 7

Optimalizace algoritmu

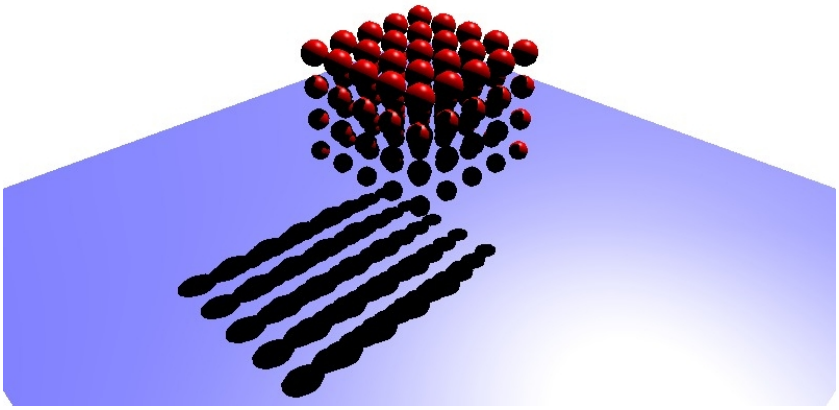
V následujících kapitolách je prezentováno několik optimalizací, které lze rozdělit na dva základní typy. Prvním jsou optimalizace mající za úkol včas eliminovat trojúhelníky, jimiž vržený stín nemůže přispět k výsledné stínové mapě, a test zastínění s těmito trojúhelníky by se tedy počítal zbytečně. Druhým typem jsou pak optimalizace, které se pokoušejí snížit počet vzorků, se kterými bude test zastínění pro daný trojúhelník prováděn, čehož dosahují zmenšením oblasti generované geometrickým procesorem. Oba druhy optimalizací mají tedy ve výsledku společný cíl, a sice co možná nejvíce zredukovat počet spuštění *fragment shaderu*, a tedy testů zastínění. A až na výjimky se budou všechny odehrávat právě v geometrickém procesoru, kde ke generování této oblasti dochází.

7.1 Metodika měření přínosu optimalizací

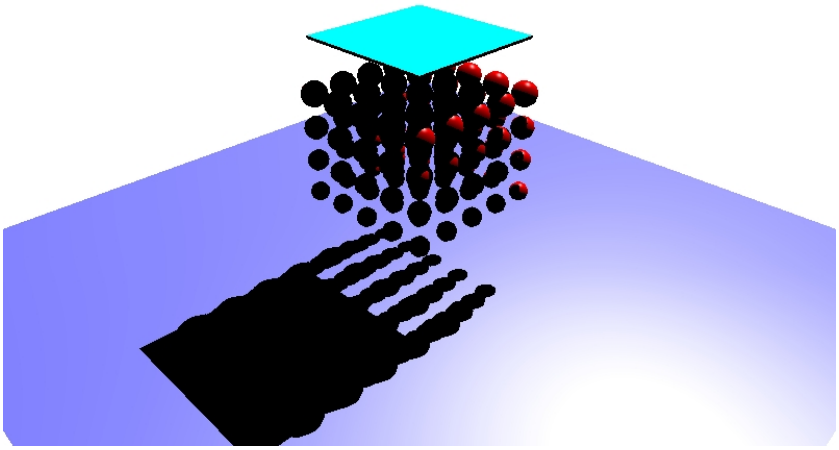
Za účelem zjištění, zda je daná optimalizace přínosem pro algoritmus pohledově závislých stínových map či nikoliv, byl vyvinut program, který tento algoritmus implementuje. Program dále dokáže přepínat mezi jednotlivými optimalizacemi, které jsou uvedeny v následujících kapitolách, a provádět měření jejich efektivity.

Abychom byli schopni tuto efektivitu měřit a provést případné porovnání, musíme vyvinout určitou metodiku měření. Jelikož jsou algoritmus a jeho rychlost závislé na pohledu, ze kterého se na scénu díváme, je nutné všechna měření provádět vždy ze stejného místa. Námi vytvořená umělá scéna a vybraný pohled jsou zachyceny na obrázku 7.1, přičemž její varianty ukazují různé stupně zastínění. Na tuto scénu se budeme odkazovat i v dalších částech této práce. Program dále dovoluje měnit složitost geometrie scény, a to ve třech krocích: cca 18 tisíc, 76 tisíc a 174 tisíc trojúhelníků[1].

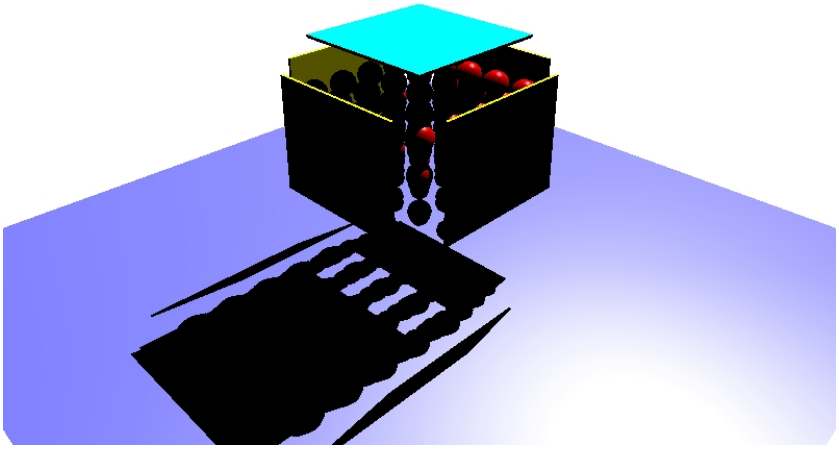
[1] Zanedbáme-li stínící plochy, obsahuje scéna vždy 100 koulí vytvořených pomocí knihovny GLUT funkcí `glutSolidSphere(a, b)`. Složitost geometrie je dána jedním parametrem $P = a = b$, který nabývá hodnot 10, 20 nebo 30. Počet trojúhelníků v jedné této kouli je potom $2P(P-1)$.



(a)



(b)



(c)

Obrázek 7.1: Měřená scéna s malým (a), středním (b) a vysokým (c) stupněm zastínění

Jako měřítko rychlosti algoritmu, a tedy i jeho optimalizací, je brán čas, jenž algoritmus stráví výpočty na grafické kartě – měření času na hlavním procesoru by totiž nebylo tak přesné, neboť může být mnohem významněji ovlivněno chodem jiných programů či operačního systému. Čas strávený výpočty na grafické kartě lze získat pomocí funkcí, které nabízí rozšíření OpenGL s názvem GL_EXT_timer_query. Dále je možné spočítat, kolik času algoritmus alespoň přibližně stráví v určitých fázích zobrazovacího řetězce, a to díky dalšímu rozšíření GL_EXT_transform_feedback.

Samotné měření je prováděno tak, že je pro každou měřenou fázi provedeno třicet vykreslovacích cyklů (snímků) a je spočtena průměrná doba výpočtu na jeden cyklus. První krok algoritmu pohledově závislých stínových map, tedy získání textury vzorků, je pro všechny optimalizace stejný, a tudíž zanedbatelný při porovnávání jejich efektivity. Měření času navíc nebere v potaz finální aplikaci stínové mapy na obraz scény, které se též provádí vždy. Díky výše zmíněným rozšířením jsme schopni měřit následující fáze zpracování:

1. Měření celkové doby výpočtu algoritmu, tedy času stráveném ve vertexovém i geometrickém procesoru (VG), pak následnou rasterizací a spouštěním fragmentového procesoru (RF) a nakonec výpočtem zastínění (Z) v tomto procesoru. Tento čas lze tedy vyjádřit jako:
   $T_1 = VG + RF + Z$

2. Měření doby výpočtu bez spouštění rasterizace[2], čímž získáme hrubý odhad doby výpočtu pouze na vertexovém a geometrickém procesoru, tedy:
   $T_2 = VG$.

3. Měření celkové doby výpočtu algoritmu, přičemž tentokrát ve fragmentovém procesoru na základě proměnné zaslané z řídící aplikace ihned zahazujeme všechny fragmenty (`discard`), takže se neprovádí test zastínění. Toto lze vyjádřit jako:
   $T_3 = VG + RF$.

Na základě možnosti měřit čas strávený ve výše uvedených fázích lze potom odvodit následující hodnoty, které jsou brány jako kritéria posouzení efektivity jednotlivých optimalizací:

1. Měření doby celého výpočtu.
   $T = T_1 = VG + RF + Z$.

2. Měření doby výpočtu strávené ve vertexovém a geometrickém procesoru.
   $T_{VG} = T_2 = VG$.

[2]Zakázání rasterizace a tedy i veškerého následného zpracování zobrazovacím řetězcem dosáhneme povolením GL_RASTERIZER_DISCARD, což umožňuje právě rozšíření GL_transform_feedback

3. Měření doby rasterizace a spouštění fragmentového procesoru.
   $T_{RF} = T_3 - T_2 = RF$.

4. Měření doby samotného výpočtu zastínění.
   $T_Z = T1 - T_3 = T - T_{VG} - T_{RF} = Z$.

Další měřenou hodnotou je potom maximální dosažený počet snímků za vteřinu (FPS, *frames per second*), který naopak měří vykreslení celé scény, a to včetně času potřebného na získání textury vzorků na počátku algoritmu a aplikaci stínové mapy na scénu na jeho konci. Počet FPS bude tedy vždy nižší, než prostá převrácená hodnota z výše uvedeného času T. Doba vykreslení jednoho snímku je též měřena pouze na grafické kartě.

Poslední měřenou hodnotou je počet primitiv generovaných geometrickým procesorem, což může dodat další užitečnou informaci o efektivitě optimalizací – samozřejmě pouze těch, které eliminují celé trojúhelníky a nedochází tak vůbec ke generování oblasti pokrývající viditelné vzorky. Informace o tomto počtu bude u relevantních optimalizací uváděna jako P_c (primitive count).

Doba výpočtu ostrých stínů pomocí základní verze algoritmu pohledově závislých stínových map, tedy verze řešící problém hrubou silou, nebyla měřena, neboť tento algoritmus vykazuje velmi špatné výsledky již pro scény obsahující řádově stovky trojúhelníků. Pro námi testovanou scénu je tedy zcela nepoužitelný, jelikož doba vykreslení jednoho snímku se pohybuje v desítkách vteřin. Za první použitelnou verzi tedy budeme považovat algoritmus s první optimalizací, která je popsána v následující kapitole.

Veškerá měření byla prováděna v okně s rozlišením 800x600 na následující hardwarové konfiguraci:

CPU:	Intel Core 2 Duo T8100 2.10 GHz
GPU:	GeForce 8600M GT
OpenGL:	3.0.0 NVIDIA 185.18.36
GLSL:	1.3 via Cg compiler

Nakonec ještě podotkněme, že řazení následujících kapitol nesouvisí s pořadím, ve kterém jsou optimalizace použity za běhu algoritmu. Jsou řazeny podle metod a principů, ze kterých vycházejí. Pořadí, v jakém jsou ve výsledku použity, je uvedeno v kapitole 7.6.

7.2 Využití metody stínových těles

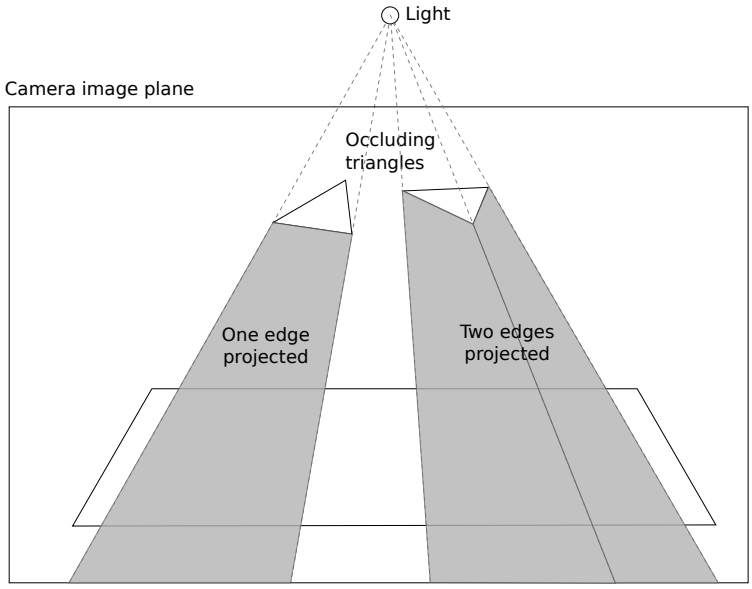
7.2.1 Projekce stínového polygonu

Jak již víme, problém algoritmu pohledově závislých stínových map tak, jak byl doposud prezentován a implementován, je bezesporu v řešení problému hrubou silou. To je způsobeno tím, že oblast, generovaná pro každý trojúhelník ve scéně za účelem spuštění testu zastínění, pokrývá všechny viditelné vzorky ve scéně. Tento přístup tedy nevyhnutelně do testu zastínění zahrne i ty vzorky, na které nemůže daný trojúhelník nikdy vrhat stín.

Při hledání optimalizace tohoto přístupu se inspirujeme metodou stínových těles (viz kapitola 5.1), kdy se pro každý objekt ve scéně určí takzvané stínové těleso, tedy objem, uvnitř kterého se může stín vržený objektem vyskytovat. Pokud podobný princip namapujeme na algoritmus pohledově závislých stínových map, objektem vrhajícím stín bude vždy trojúhelník a stínové těleso z něj vytvořené bude tedy ohraničeno třemi polygony – takzvanými stínovými polygony – které vzniknou protažením jednotlivých hran trojúhelníku do nekonečna směrem od světla. Pokud bychom dále postupovali stejně, jako metoda stínových těles, hledali bychom, které z viditelných vzorků se nacházejí uvnitř takto vytvořeného stínového tělesa.

V našem případě je však situace jednodušší, jelikož žádný test na přítomnost viditelných vzorku ve stínovém tělese provádět nemusíme. My potřebujeme pouze za použití tohoto principu vygenenerovat takovou oblast, která bude v pohledu z kamery pokrývat ty vzorky, kam může stín vržený trojúhelníkem potenciálně dopadnout – a tuto oblast tvoří právě výše zmíněné stínové polygony. Jimi, respektive plochou, kterou pokrývají, pak můžeme nahradit doposud vytvářený obdélník přes celou obrazovku, čímž dozajista ušetříme mnoho zbytečných spuštění testu zastínění.

Právě díky tomu, že nemusíme vytvářet celé stínové těleso, však není ani nutné vytvářet stínový polygon z každé hrany trojúhelníku. Pro účely rasterizace a následného spouštění *fragment shaderu* postačí posílat dále do zobrazovacího řetězce pouze takové stínové polygony, které jsou přivrácené ke kameře. Tento přístup ilustruje obrázek 7.2 a zároveň ukazuje, že takto bude pro každý trojúhelník v závislosti na poloze jeho vrcholů generován pouze jeden, maximálně dva stínové polygony. V praxi toho lze dosáhnout dvěma způsoby. V prvním případě bychom mohli přímo v *geometry shaderu* počítat, zda je daný stínový polygon přivrácen[3] ke kameře či nikoliv, a dále do zobrazovacího řetězce bychom ho posílali pouze tehdy, pokud by toto splňoval. V druhém případě vygenerujeme vždy všechny tři polygony a spolehneme se na hardwarovou eliminaci odvrácených ploch (*back face culling*), který všechny odvrácené plochy automaticky zahodí. Vzhledem ke skutečnosti, že by první z přístupů vyžadoval několik výpočtů navíc, jeví se druhý přístup jako efektivnější.



Obrázek 7.2: Stínový polygon vzniklý projekcí hran trojúhelníku přivrácených ke kameře

Část kódu *geometry shaderu*, která slouží k vytváření stínového polygonu z hran trojúhelníku, můžeme vidět na příkladu 7.1.

```
// biasMatrix je matice mapujici rozsah [-1,1] na rozsah [0,1]
mat4 biasProjectionMatrix = biasMatrix * gl_ProjectionMatrix;

// vytvarime tri stinove polygony (jako pas trojuhelniku),
// hardwarovy face culling se postara o to, ze se rasterizuji
// jen polygony privracene ke kamere
for (int i = 0; i < gl_VerticesIn; ++i)
{
  // index nasledujici hrany
  int next = (i+1) % gl_VerticesIn;

  // vrcholy hrany trojuhelniku, kterou budeme projektovat do nekonecna
  vec4 position0 = gl_PositionIn[i];
  vec4 position1 = gl_PositionIn[next];

  // projekce teto hrany do nekonecna (parametr w je nula)
  // v poli dir jsou ulozeny smerove vektory od svetla k bodum
  // trojuhelniku
  vec4 position2 = vec4(dir[i], 0.0);
  vec4 position3 = vec4(dir[next], 0.0);

  // emitujeme prvni a druhy vrchol hrany trojuhelniku
```

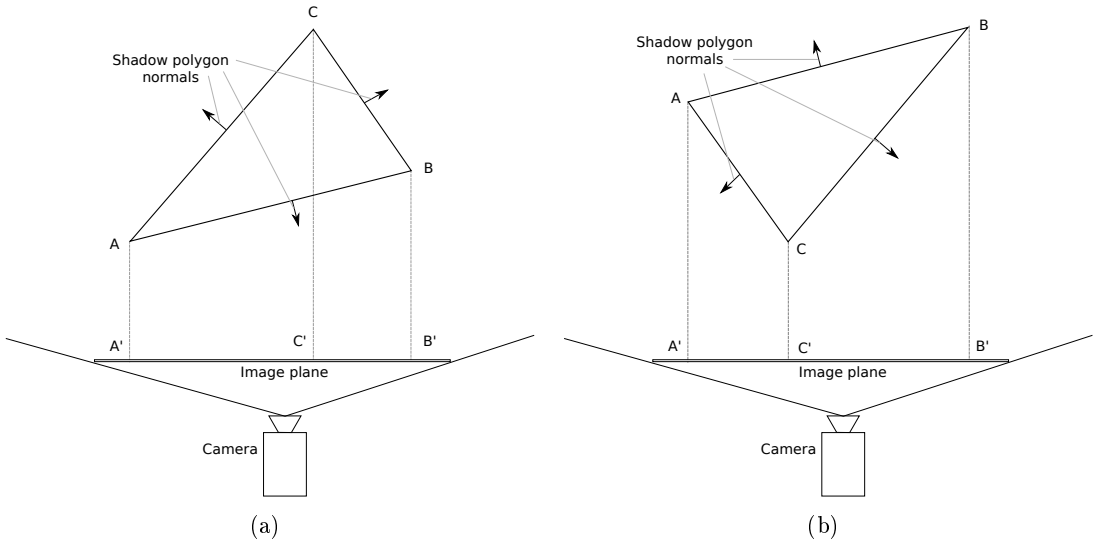
```
    gl_Position = gl_ProjectionMatrix * position0;
    texCoord = biasProjectionMatrix * position0;
    EmitVertex();
    gl_Position = gl_ProjectionMatrix * position1;
    texCoord = biasProjectionMatrix * position1;
    EmitVertex();

    // emitujeme hranu projektovanou do nekonecna
    gl_Position = gl_ProjectionMatrix * position2;
    texCoord = biasProjectionMatrix * position2;
    EmitVertex();
    gl_Position = gl_ProjectionMatrix * position3;
    texCoord = biasProjectionMatrix * position3;
    EmitVertex();

    EndPrimitive();
  }
```

Příklad 7.1: Generování stínového polygonu z hran trojúhelníku

Při hlubším zamyšlení se nad tímto problémem se však naskytuje ještě jedna otázka, a sice, zda by nestačilo pro každý trojúhelník generovat vždy jen jeden stínový polygon. Dvojice obrázků 7.3 zobrazuje trojúhelník z pohledu směrového světelného zdoje a kameru s její přední ořezávací rovinou, na kterou se stínové polygony promítají. Stínový polygon je v tomto pohledu reprezentován každou hranou trojúhelníku a jejím promítnutím do nekonečna směrem „do obrazovky“. Na obou obrázcích vidíme, že stínový polygon tvořený úsečkou AB – respektive jejím průmětem $A'B'$ do přední ořezávací roviny kamery – je v tomto promítnutí dán sjednocením stínových polygonů tvořených úsečkami BC a CA, respektive jejich průměty $B'C'$ a $C'A'$. Díky tomu by tedy mělo být teoreticky možné generovat vždy jen takový stínový polygon, jenž je z trojice možných polygonů jako jediný přivrácený (obrázek 7.3(a)), případně jediný odvrácený (obrázek 7.3(b)). V praxi se však setkáváme s problémy, které použití tohoto přístupu brání.

Jeden z těchto problémů se týká omezení rozsahu hloubky, do které můžeme za účelem vytvoření stínového polygonu projektovat hranu trojúhelníku. Ideálně bychom potřebovali, aby byl stínový polygon protáhnut až do nekonečna, ale v praxi se musíme vejít jen do omezého číselného rozsahu. Tento problém se projevuje pouze, když je kamera umístěna v blízkosti světla a stínový polygon tedy v pohledu z kamery směřuje směrem „do obrazovky“, a zároveň pouze pokud je tento polygon odvrácen od kamery. Na obrázku 7.4 je znázorněn právě tento případ, kdy jsou stínové polygony vzniklé projekcí jednotlivých hran trojúhelníku

[3] S ohledem na další optimalizace uvedené v této práci tedy budeme předpoklat, že normály generovaných stínových polygonů budou vždy směřovat směrem ven ze stínového objemu, pokud nebude výslovně uvedeno jinak.



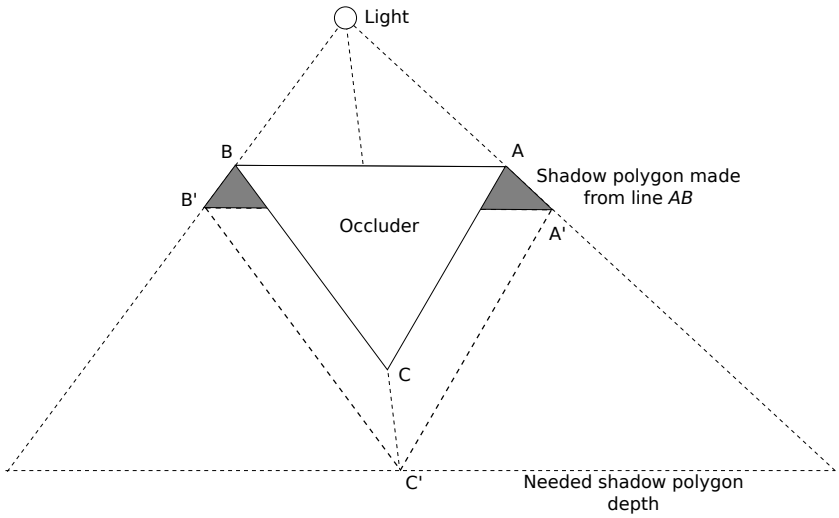
Obrázek 7.3: Výběr hrany k projekci

do „nekonečna“ omezeny maximální možnou hloubkou.

To se projeví tak, že možné stínové polygony mohou být vykresleny maximálně do hloubky, která je na obrázku vymezena trojúhelníkem $A'B'C'$. Nebýt tohoto omezení, pokrýval by odvrácený stínový polygon $ABB'A'$ svou plochou oba přivrácené polygony $CBB'C'$ i $ACC'A'$. S tímto omezením toho však není schopen a tudíž nemůže pokrýt všechny vzorky, které mohou být potenciálně ve stínu. Omezení hloubky však není problém, generujeme-li stínové polygony přivrácené ke kameře, jelikož reálný stín může opravdu ležet pouze uvnitř plochy vymezené body $ABB'C'A'$.

Tento problém samozřejmě není neřešitelný, vyžadoval by však například rozšíření odvráceného stínového polygonu až po hranici, která je znázorněna ve spodní části obrázku. Tím bychom však generovali stínový polygon s celkovou plochou větší, než zabírají oba přivrácené polygony dohromady, a docházelo by tedy ke zbytečnému spouštění testu zastínění na vzorcích, které ve stínu ležet nemohou. Dalším problémem je potom nemožnost použít optimalizaci ořezávání stínového polygonu maximální hloubkou z hloubkové mapy, viz kapitola 7.3.2. Toto ořezání maximální hloubkou by se na odvrácených stínových polygonech projevilo stejně, jako výše popsané omezení hloubky.

I při generování stínových polygonů přivrácených ke kameře však můžeme narazit na další úskalí, tentokrát v podobě nechtěného ořezání stínového polygonu zadní ořezávací rovinou kamery (*far clipping plane*). S tímto se však již potýkal při vylepšování původní metody stínových těles *Kilgard* [EK02]. Jeho řešení spočívá v použití speciální projekční matice, která nebere v potaz zadní ořezávací rovinu a dovoluje tedy vykreslit i geometrii nacháze-



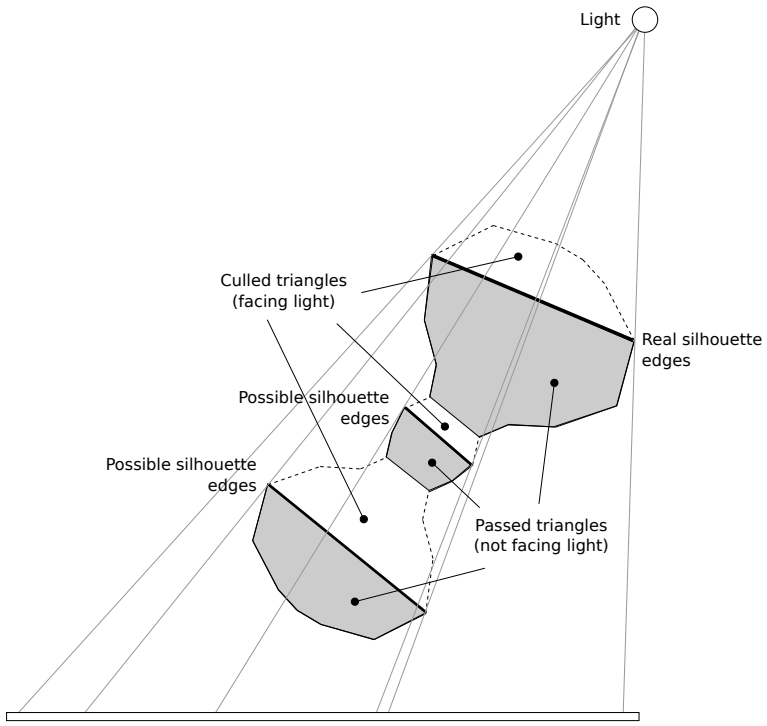
Obrázek 7.4: Problém s omezením hloubky při projekci odvráceného stínového polygonu

jící se za ní. Na grafických kartách *NVidia* lze přesně tohoto dosáhnout použitím rozšíření GL_NV_depth_clamp.

7.2.2 Eliminace přivrácených ploch

Optimalizací popsanou v předchozí kapitole jsme (s odkazem na výsledky měření v kapitole následující) dosti razantně omezili velikost generovaných stínových polygonů a tím také počet spuštění *fragment shaderu* a tedy testů zastínění. Nyní přistoupíme k dalšímu kroku, který by měl pomoci zrychlit práci algoritmu, tentokrát pomocí včasné eliminace trojúhelníků, které nemohou přispět k výsledné stínové mapě.

Algoritmus pohledově závislých stínových map pracuje pouze s trojúhelníky (respektive souřadnicemi jejich vrcholů) bez jakékoliv další přidané informace o závislostech mezi nimi či jejich uspořádání ve scéně. Pokud bychom však chtěli algoritmus výrazně zrychlit, mohli bychom – samozřejmě za cenu snížení jeho univerzálnosti – přidat informaci o sousednosti trojúhelníků (*triangle adjacency*). Pokud by algoritmus měl k dispozici tuto informaci, bylo by možné namísto vytváření stínových polygonů ze všech trojúhelníků tělesa určit pouze jeho potenciální obrysové hrany (*possible silhouette edges*) a jen ty protáhnout do nekonečna – podobně, jako se to v praxi dělá u metody stínových těles. Trojúhelníky, jejichž hrany nejsou možnými obrysovými hranami tělesa, by tak bylo možné okamžitě eliminovat. Jak již ale bylo řečeno, algoritmus je implementován tak, aby mohl pracovat i bez této informace, i když se tím pravděpodobně připravujeme o značnou výhodu. Na druhou stranu tím však zároveň poskytujeme prostor pro výzkum dalších optimalizací, které by se sousedností trojúhelníků mohly pracovat.

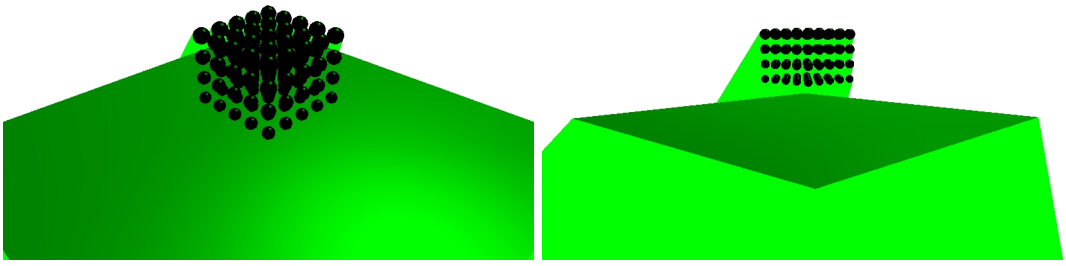


Obrázek 7.5: Optimalizace odstraněním ploch přivrácených ke světlu

Základní myšlenku tohoto přístupu však použijeme i k optimalizaci našeho algoritmu. Využijeme zde faktu, že potenciální obrysové hrany tělesa jsou takové hrany, které jsou společné jak trojúhelníkům přivráceným ke světlu, tak i těm odvráceným. Jinak řečeno oddělují osvětlenou část tělesa od neosvětlené (pro připomenutí se můžeme podívat zpět do kapitoly 5.1 na obrázek 5.2). I přesto, že množinu těchto hran nemáme k dispozici, můžeme přímo v *geometry shaderu* snadno určit, který trojúhelník je ke světlu přivrácený a který není.

V případě uzavřených těles lze pak tedy eliminovat například všechny přivrácené trojúhelníky, jelikož za každým z nich se musí (právě díky uzavřenosti těles) nacházet jeden či více odvrácených trojúhelníků, které svou celkovou plochou pokryjí v pohledu ze světla jeho plochu. Jinými slovy tedy stín vržený trojúhelníkem přivráceným ke světlu bude ležet ve stínu vrženém jedním či více trojúhelníky odvrácenými od světla.

Tento princip je ilustrován na obrázku 7.5. Teoreticky lze namísto přivrácených trojúhelníků eliminovat i trojúhelníky odvrácené – právě díky skutečnosti, že jsou potenciální obrysové hrany tělesa společné jak přivráceným, tak i odvráceným trojúhelníkům – avšak musíme dát pozor na kompatibilitu s dalšími optimalizacemi, konkrétně těmi, které využívají hloubkovou mapu k porovnávání hloubek trojúhelníků, viz kapitola 7.3.



(a) (b)

Obrázek 7.6: Sjednocení stínových polygonů vytvořených s využitím metody stínových těles. Pohled z místa pro měření efektivity (a) a pohled z dálky (b) pro lepší ilustraci.

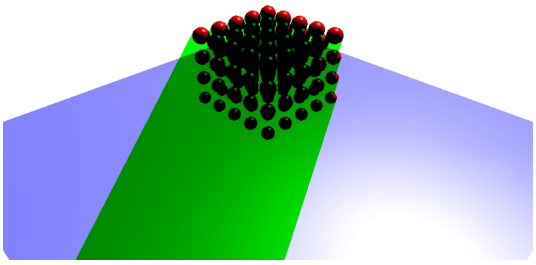
Tuto optimalizaci lze samozřejmě použít pouze pro tělesa uzavřená, tedy tělesa mající přední i zadní stěnu. Pro tělesa pouze s přivrácenými stěnami optimalizace způsobí, že nebudou vrhat stín. I toto lze však vyřešit, například tak, že si do *geometry shaderu* předáme z řídící aplikace proměnnou, signalizující, že se test na přivrácenost trojúhelníku nemá provádět.

7.2.3 Zhodnocení přínosu optimalizací

Přínos první optimalizace považujeme za obrovský, neboť bez projekce stínového polygonu je algoritmus nepoužitelný již pro scény obsahující několik set trojúhelníků. Tato optimalizace posouvá rychlost vykreslování na hranici reálného času, jelikož se pro každý trojúhelník již nemusí počítat test zastínění s každým viditelným vzorkem, ale pouze s těmi, které jsou pokryty generovaným stínovým polygonem.

Sjednocení stínových polygonů, vytvořených ze všech relevantních trojúhelníků v námi testované scéně, která byla popsána a ukázána v úvodu kapitoly o optimalizacích, můžeme pozorovat na obrázku 7.6. Pokud navíc přidáme optimalizaci druhou, tedy eliminaci všech trojúhelníků přivrácených ke světlu, získáme tím stínové polygony zobrazené na obrázku 7.7.

Jak můžeme pozorovat v tabulce 7.1, eliminace ploch přivrácených ke světlu (sloupec EP) nám počet generovaných primitiv, a tedy počet vytvářených stínových polygonů, redukuje téměř na polovinu, vzhledem k pouhému generování stínového polygonu ze všech trojúhelníků (sloupec SP). To se projevuje i na všech následně naměřených dobách výpočtů, přičemž evidujeme zrychlení algoritmu přibližně o 67%. Pro složitější scénu potom můžeme pozorovat přibližně lineární zpomalení. Vliv rozsahu zastínění na algoritmus je minimální. Pro nejvíce



Obrázek 7.7: Sjednocení stínových polygonů vytvořených s využitím metody stínových těles, navíc s eliminací trojúhelníků přivrácených ke světlu.

zastíněnou scénu (viz obrázek 7.1(c) na začátku kapitoly o optimalizacích) pozorujeme zpomalení vykreslování asi o 0.5 FPS při počtu trojúhelníků 18 tisíc.

Scéna 7.1(a) 18k		SP	EP
P_c		108012	54288
FPS		10.7	17.9
T_{VG}	$[ms]$	36	23
T_{RF}	$[ms]$	1.3	0.8
T_Z	$[ms]$	52	28
T	$[ms]$	89	51

Scéna 7.1(a) 76k		SP	EP
P_c		456012	229452
FPS		3.7	6.2
T_{VG}	$[ms]$	155	96
T_{RF}	$[ms]$	3.3	1.9
T_Z	$[ms]$	108	61
T	$[ms]$	266	159

Tabulka 7.1: Porovnání přínosu přidání eliminace trojúhelníků přivrácených ke světlu.

Přínos obou těchto optimalizací je tedy nesporný a pouze díky nim má smysl v optimalizování dále pokračovat. Je zřejmé, že jako první zařadíme samozřejmě eliminaci přivrácených trojúhelníků, a teprve potom budeme generovat stínový polygon z těch, které jsou odvrácené.

V následujících kapitolách diskutujících další optimalizace je automaticky předpokládáno generování stínového polygonu, jelikož bez této optimalizace nejsme schopni naměřit rozumné

hodnoty.

7.2.4 Vznikající artefakty

Při použití projekce stínového polygonu se ojediněle objevují artefakty v podobě jednopixelových světlých míst nacházejících se uvnitř zastíněných oblastí. Tyto artefakty jsou způsobeny nesprávnou rasterizací stínových polygonů, ke které dochází tehdy, když hrana polygonu zasahuje do pixelu, na jehož pozici je potenciálně zastíněný vzorek, ale neprotíná střed pixelu. Rasterizace, vzorkující pouze ve středech pixelů, takovýto pixel vyhodnotí jako nepokrytý stínovým polygonem a odpovídající fragment tak není vytvořen. V důsledku toho se tedy nespustí *fragment shader* a následný test zastínění na odpovídajícím vzorku. Pokud je pak takovýto vzorek pokryt jen a pouze tímto stínovým polygonem, zůstane pixel vyhodnocen jako osvětlený.

Řešením tohoto problému je pak například takzvaná konzervativní rasterizace, kdy bychom v *geometry shaderu* generovali větší stínový polygon – ideálně o půl pixelu z každé strany – čímž bychom zajistili, že bude polygon pixel pokrývat. Nepodařilo se nám však implementovat dostačně rychlou verzi, která by celý algoritmus výrazně nezpomalovala. Případná implementace efektivního algoritmu je tedy ponechána na čtenáři.

7.3 Využití metody stínových map

Díky využití základních principů metody stínových těles se nám podařilo velmi znatelně snížit počet generovaných stínových polygonů a zároveň i omezit jejich velikost, což má za výsledek optimálnější spouštění *fragment shaderu* a následný test zastínění. Až doposud jsme však nebrali v potaz zastínění z pohledu světla, a tak se tento test nadále spouští i pro trojúhelníky, které jsou z pohledu světla zastíněny jinými tělesy a jimi vržený stín tak nemůže více přispět k výsledné stínové mapě. Pokud bychom před vytvořením stínového polygonu měli informaci o tom, které plochy jsou z pohledu světla osvětleny a které zastíněny, bylo by možné algoritmus dále optimalizovat. Přesně tuto informaci nám poskytuje hloubková mapa z pohledu světla, která je základním stavebním kamenem metody stínových map, popsané blíže v kapitole 5.2. Právě proto je však také zdrojem většiny problémů s touto metodou spojených. Náš přístup ale hloubkovou mapu využívá pouze jako optimalizační strukturu, a tak těmito problémy netrpí.

7.3.1 Eliminace trojúhelníků zastíněných z pohledu světla

Jak již víme, hloubková mapa – konkrétně pak textura ukládající hloubku scény z pohledu světla – obsahuje vzdálenosti bodů nejblíže ke světlu. Body v této vzdálenosti jsou tedy

osvětleny a vše, co se nachází ve vzdálenosti větší, leží ve stínu. Původní metoda stínových map potom pracuje s těmito hodnotami tak, že pro každý bod skrytý za pixelem obrazovky z pohledu kamery nalezne odpovídající texel v hloubkové mapě a porovná hodnoty vzdáleností od světla. Podobný princip bychom mohli použít i v algoritmu pohledově závislých stínových map. Na základě hodnot z hloubkové mapy bychom tak mohli eliminovat trojúhelníky, které jsou vzhledem ke světlu zastíněné, a tudíž stínový polygon z nich vygenerovaný by nepřinesl žádnou novou informaci o zastínění ve scéně.

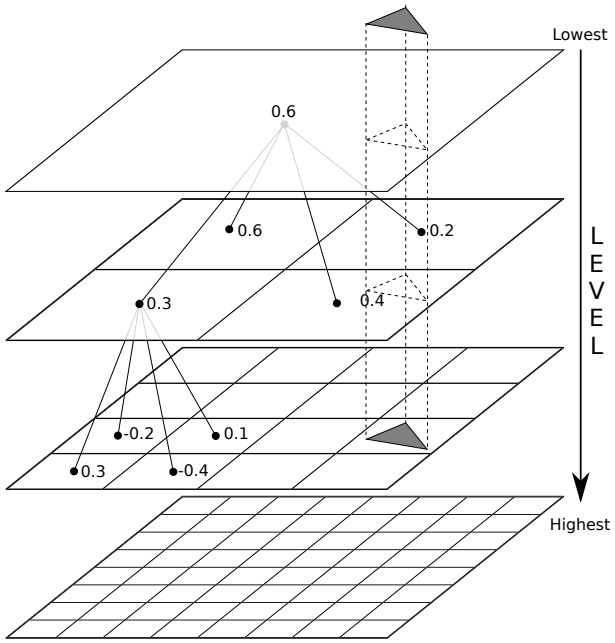
Při aplikaci tohoto principu však narážíme na drobný problém. Algoritmus pohledově závislých stínových map totiž nepracuje v obrazovém prostoru jako metoda stínových map, a tudíž nelze porovnávání hloubek provádět ve *fragment shaderu* na úrovni jednotlivých fragmentů. My musíme porovnávat celé trojúhelníky, a to již v *geometry shaderu*. Ideálně bychom potřebovali získat z hloubkové textury maximální z hodnot texelů, které v projekci do přední ořezové roviny světla trojúhelník pokrývá. Pouze s takovou hodnotou lze jednotlivé vrcholy trojúhelníku porovnat a zjistit tak, zda je celý osvětlený, zastíněný či částečně zastíněný. Získat tyto hodnoty by však bylo výpočetně náročné, a tak se uchýlýme ke konzervativnější metodě.

Ta spočívá v tom, že z jednoduché hloubkové textury vytvoříme hierarchickou vyhledávací strukturu – a vzhledem k tomu, že pixely textury jsou uspořádány v rastru, bude se jednat o hierarchickou víceúrovňovou mřížku. Tato mřížka bude též umístěna v přední ořezávací rovině světla, stejně jako dříve jednoduchá hloubková textura. Na nejvyšší úrovni této mřížky budeme mít hloubky jednotlivých bodů z pohledu světla a každá nižší úroveň potom bude uchovávat maximum z určité podmnožiny hloubek na úrovni předcházející. Nejníže tedy bude jediná hodnota, reprezentující maximum ze všech hloubek – tedy hloubku nejvzdálenějšího bodu viditelného z pohledu světla. Tuto strukturu ukazuje obrázek 7.8 a zároveň naznačuje jeden z možných postupů vyhledávání správné úrovně pro daný trojúhelník. Těmto postupům se věnuje detailněji následující kapitola.

7.3.1.1 Vyhledání referenční hodnoty v hloubkové mapě

Hloubkovou texturu a nad ní postavenou hierarchickou vyhledávací strukturu, jež byla popsána v předchozí kapitole, máme k dispozici v *geometry shaderu*. Jeho úkolem je nyní pro každý trojúhelník vybrat mřížku na správné úrovni, načíst odpovídající (referenční) hodnotu hloubky z hloubkové mapy a porovnat ji s hloubkou každého vrcholu trojúhelníku z pohledu světla. Tímto způsobem zjistíme, které trojúhelníky jsou z pohledu světla zastíněny a můžeme je tak okamžitě eliminovat.

Nyní, pro snazší pochopení souvislostí mezi výše popsanou vyhledávací strukturou a její konkrétní implementací, prozradíme, že tuto strukturu spolu s veškerou požadovanou funkcionalitou nám umožňuje velmi efektivně realizovat samotná grafická karta – a to pomocí



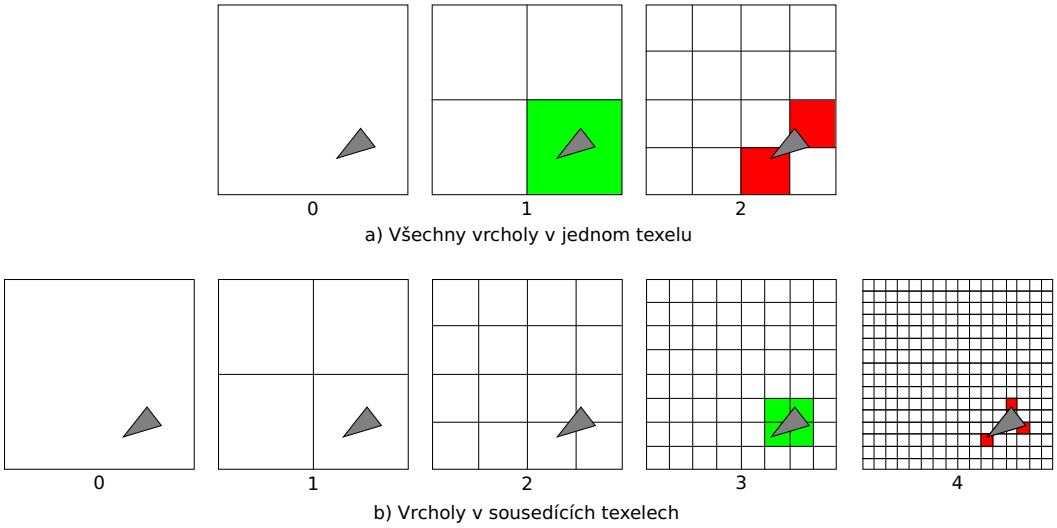
Obrázek 7.8: Hierarchická víceúrovňová mřížka pro vyhledávání referenční hloubky

mipmap. Jednotlivé úrovně mřížky jsou tedy mipmapy hloubkové textury, přičemž na nejvyšší úrovni stojí hloubková textura v nejvyšším rozlišení. Buňky jednotlivých mřížek potom odpovídají jednotlivým texelům těchto mipmap. Vytváření mipmap z hloubkové textury se blíže věnujeme v kapitole 7.3.1.3.

Algoritmů pro výběr správné úrovně mřížky – tedy mipmapy hloubkové textury – lze dozajista vymyslet více. Problémem je však dosáhnout rovnováhy mezi jejich výpočetní složitostí a přesností, z níž vyplývá efektivita takového algoritmu. Efektivitou v tomto případě myslíme poměr počtu eliminovaných trojúhelníků ku počtu trojúhelníků, které by byly eliminovány za použití přesné metody určení referenční hloubky. V rámci této práce byly vyzkoušeny dva různé postupy, jeden rychlý a jednoduchý s nižší efektivitou, druhý složitější a výpočetně náročnější s vyšší efektivitou.

Postup prvního a jednoduššího algoritmu je takový, že začínáme od nejméně detailní mipmapy a postupujeme směrem k nejvíce detailní, přičemž hledáme takovou úroveň (LOD, *level of detail*), kdy ještě všechny vrcholy trojúhelníku, promítnuté do přední ořezávací roviny světla, spadají do stejného texelu. Hodnota v tomto texelu je poté referenční hodnotou, vůči které porovnáváme hloubky jednotlivých vrcholů trojúhelníku. Tento algoritmus výběru mipmapy je ilustrován na obrázku 7.9(a).

Nesporná výhoda tohoto algoritmu je, že je velice jednoduchý na implementaci a hlavně vyžaduje pouze jeden přístup do hloubkové textury, což je jinak velmi „drahá“ operace.



Obrázek 7.9: Algoritmy pro výběr úrovně LOD

Nevýhodou je potom nízká optimálnost, pokud se trojúhelník nachází na rozmezí několika texelů – tehdy se vybírá mipmapa na zbytečné nízké úrovni. Kritická situace potom nastává, pokud se trojúhelník nachází přímo uprostřed hloubkové textury, kdy se automaticky vybírá nějméně detailní mipmapa bez ohledu na to, jak malý daný trojúhelník může být.

Druhý, složitější algoritmus, také postupuje od mipmapy na nejnižší úrovni směrem k nejvyšší, kritériem pro výběr je však to, zda se vrcholy trojúhelníku ještě nacházejí v sousedních texelech – čili rozdíl v souřadnicích texelů, do kterých se vrcholy trojúhelníku promítnou, nesmí v žádné z os x ani y překročit 1. Diagonálně umístěné vrcholy jsou tedy považovány za sousední. Jakmile určíme správnou úroveň, musíme načíst hodnotu každého z texelů a provést nad nimi podobnou operaci, jako při samotné tvorbě mipmapy - musíme tedy vybrat maximum z těchto hodnot. Vypočítanou hodnotu pak lze použít jako referenční hodnotu pro porovnávání hloubky. Tento algoritmus je ilustrován na obrázku 7.9(b).

Tento algoritmus je mnohem efektivnější, co se týče výběru úrovně LOD, což můžeme pozorovat právě na obrázku 7.9. Zde je pro daný příklad vybrána o dvě úrovně detailnější mipmapa, než za použití předchozího algoritmu. Tato efektivita je však vykoupena tím, že budeme potřebovat minimálně dva, často však až čtyři přístupy k hloubkové textuře. Dva přístupy potřebujeme, pokud jsou vrcholy ve dvou sousedních texelech umístěných buď vedle sebe či nad sebou. Čtyři přístupy budeme potřebovat, pokud jsou umístěny ve dvou texelech diagonálně, případně pokud je každý vrchol v jiném texelu. Teoreticky by bylo možné omezit v některých případech tento počet na tři přístupy, ale to by vyžadovalo další sérii testů k potvrzení, že plocha trojúhelníku zasahuje pouze do tří texelů.

Pravděpodobně právě kvůli těmto přístupům do textury a celkově složitější implementaci,

vyžadující nejen určit, kterou mipmapu použít, ale i spočítat, kolik přístupů a kam bude potřeba, je ve výsledku druhý algoritmus pro námi testované scény o mnoho pomalejší než algoritmus první – respektive nepodařilo se tento algoritmus implementovat tak, aby byl rychlejší než algoritmus první, přestože potenciálně eliminuje více trojúhelníků. Pro účely této optimalizace byl tedy nakonec použit algoritmus první, který v zastíněných scénách vykazuje i přes výše zmíněnou negativní vlastnost velmi dobré výsledky.

7.3.1.2 Porovnání hloubky trojúhelníku

Jakmile nalezneme správnou úroveň mipmapy a určíme tak referenční hodnotu hloubky z hloubkové mapy pro daný trojúhelník, porovnáme postupně jeho vrcholy s touto hodnotou, což ilustruje obrázek 7.10. Pokud je hodnota v hloubkové mapě menší, než hloubky všech vrcholů trojúhelníku, můžeme ho eliminovat, jelikož víme, že tento trojúhelník leží celý ve stínu vrženém jiným tělesem. Pokud je referenční hodnota naopak větší, než hloubky všech vrcholů trojúhelníku, je tento trojúhelník plně osvětlen a musíme generovat stínový polygon. Pokud dochází k pouze částečnému zastínění trojúhelníku, tedy pokud se některé vrcholy nacházejí v porovnání s referenční hodnotou z hloubkové textury blíže ke světlu a jiné dále, musíme rozhodnout o dalším postupu. V ideálním případě by bylo třeba určit hranici mezi osvětlenou a zastíněnou částí trojúhelníku a podle této hranice trojúhelník ořezat. Ořezáním však mohou vzniknout nové vrcholy a proces generování stínového polygonu by se tak zesložitil. V případě částečného zastínění tedy neořezáváme a generujeme tak stínový polygon z celého trojúhelníku. Část kódu, ukazující implementaci výše zmíněného, nalezneme v příkladu 7.2.

Díky použití mipmapované hloubkové textury jsme tedy schopni eliminovat některé trojúhelníky, které jsou vzhledem ke světlu zastíněny jinými objekty. Efektivita této eliminace, čili počet doopravdy eliminovaných trojúhelníku ku počtu trojúhelníků, které by mohly být eliminovány za použití přesné metody, závisí na zvoleném algoritmu pro výběr úrovně LOD pro každý trojúhelník. Celkový přínos této optimalizace je diskutován v kapitole 7.3.1.4.

7.3.1.3 Implementace

V této kapitole se dostáváme ke konkrétní implementaci postupu získání hloubkové mapy a vygenerování potřebných mipmap.

Mapu hloubky vytvoříme podobně, jako je popsáno v kapitole 5.2.3. K tomu použijeme FBO s připojenou texturou, přičemž se však nejedná o texturu hloubky v pravém slova smyslu, jelikož její formát nebude GL_DEPTH_COMPONENT, nýbrž barevná textura ve vhodném formátu, například GL_RGBA32F – to proto, že následně budeme s touto texturou pracovat a manipulace s barevnou texturou je jednodušší[4]. Před samotným vykreslováním

```
// ziskame texel na vybrane urovni mipmapy, texturovacimi souradnicemi
// jsou souradnice libovolneho vrcholu transformovaneho do orezoveho
// prostoru svetla a prevedene do rozsahu [0,1]
vec4 lodZValue = vec4(texture2DLod(depthMap,
        vertexInLightSpaceCoord[0].st, selectedLODLevel)).rgba;

// pricteme malou konstantu, abychom zamezili nespravnemu porovnani hodnot
// v plovouci radove carce
lodZValue.r += 0.001;

// porovname hloubku kazdeho vrcholu trojuhelniku, a pokud je hloubka
    vsech
// vetsi nez referencni hloubka, muzeme ho eliminovat
if (vertexInLightSpace[0].z > lodZValue.r
    && vertexInLightSpace[1].z > lodZValue.r
    && vertexInLightSpace[2].z > lodZValue.r) {
  return;
}
```

Příklad 7.2: Získání referenční hodnoty hloubky a porovnání hloubky trojúhelníku v *geometry shaderu*

do této textury ještě vytvoříme prázdné mipmapy, do kterých budeme později generovat obsah. K tomu lze použít funkci `glGenerateMipmap`, jež byla přidána s rozšířením *framebuffer objekt*.

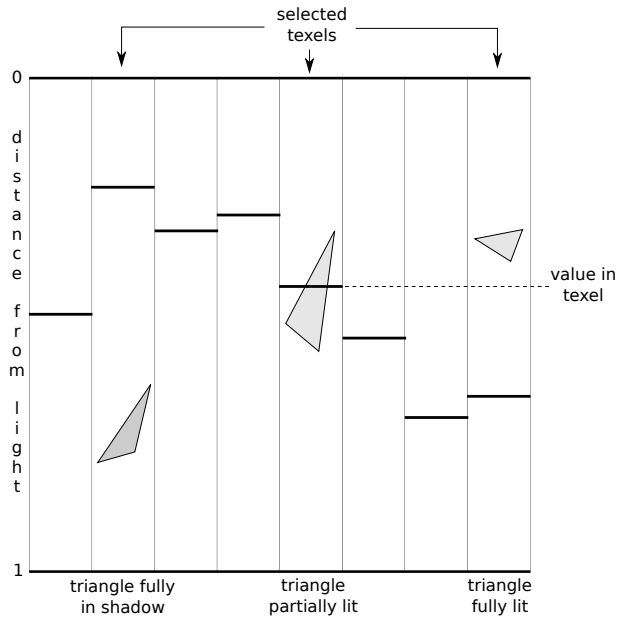
Zatímco tedy máme aktivované FBO s připojenou texturou, jejíž texely obsahují hodnotu 1 (tedy nekonečno), ukládáme pomocí *fragment shaderu* do kanálu R hodnoty souřadnice z v normalizovaném ořezovém prostoru světla. S ohledem na další optimalizaci, uvedenou v kapitole 7.3.2, máme vypnutý hloubkový test a pomocí „minimalizačního“ míchání barev[5] (*alpha blending*) ukládáme do textury pouze minimální hodnoty hloubky – hloubkový test tedy provádíme sami. Vzhledem k tomu, že při optimalizaci v kapitole 7.2.2, pomocí které eliminujeme všechny ke světlu přivrácené trojúhelníky a ponecháváme tedy pouze ty odvrácené, musíme do hloubkové mapy ukládat taktéž jen odvrácené stěny. Jinak by následné porovnávání hloubek nemohlo fungovat.

Jelikož funkce, které nám OpenGL poskytuje ke generování mipmap z existujících textur, nejsou v našem případě použitelné, musíme si mipmapy z hloubkové textury vytvořit sami. Toho dosáhneme použitím techniky vykreslování do textur pomocí sady FBO[6], kdy ke každému z nich připojíme jednu úroveň mipmapované hloubkové textury. V každém kroku i,

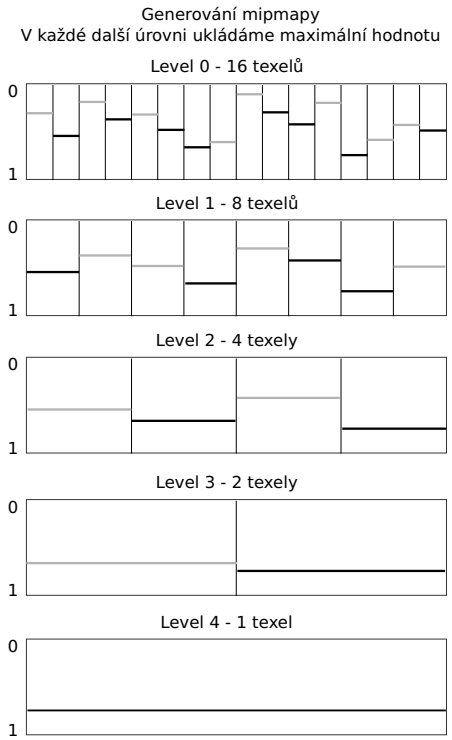
[4] Do hloubkové textury ve formátu GL_DEPTH_COMPONENT nelze ve *fragment shaderu* zapisovat, což je však operace potřebná k vytváření mipmap. Proto ukládáme do barevné textury a tento problém odpadá.

[5] Do textury ukládáme hodnotu fragmentu pouze tehdy, pokud je menší než hodnota, kterou již odpovídající pixel v textuře obsahuje.

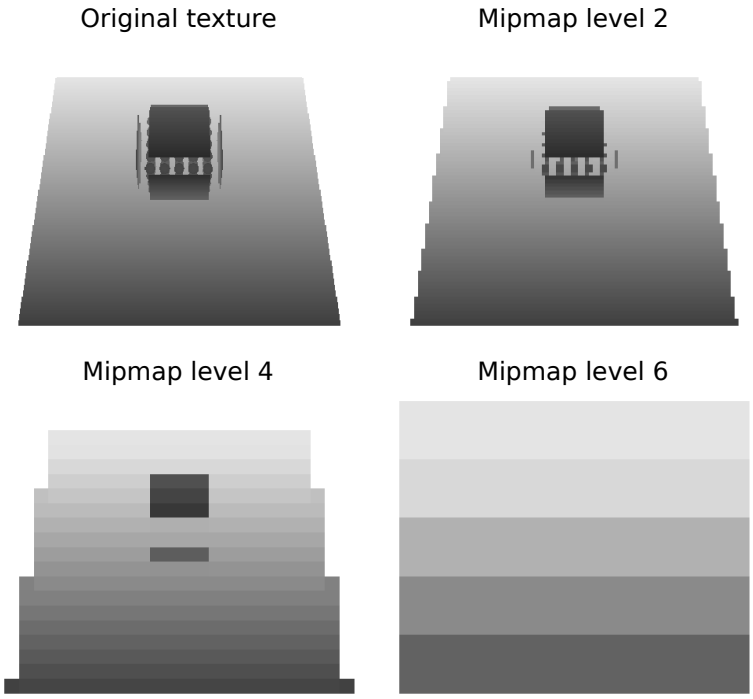
[6] Připojovat jednotlivé mipmapy k jednomu FBO je velmi pomalé, jelikož mipmapy mají v každé úrovni LOD jinou velikost a OpenGL má vysokou režii s ověřováním jeho stavu. Přepínání mezi několika FBO je proto mnohem rychlejší.



Obrázek 7.10: Porovnání hloubky trojúhelníku s referenční hodnotou v LOD mapě



Obrázek 7.11: Algoritmus vytváření mipmapy z hloubkové textury



Obrázek 7.12: Hloubková textura a její mipmapy

kterých je celkem $log_2(max(M, N))$, kde MxN je rozlišení hloubkové textury, vykreslíme obdélník přes celou obrazovku, přičemž velikost vykreslované oblasti (*viewport*) nastavíme na $MxN/(i+1)$. V každé následující úrovni tedy bude výsledná mipmapa poloviční. Obdélník poté zpracujeme pomocí *fragment shaderu*. V něm vezmeme z úrovně mipmapy vytvořené v předchozím kroku hodnoty ze čtyř jejích texelů, které jsou pokryty jedním texelem v právě vytvářené mipmapě. Jejich maximum pak uložíme jako výslednou barvu fragmentu. Zde je velmi důležité, abychom při braní maxima nepropagovali do dalších úrovní nekonečno[7], v našem případě hodnotu 1 – v takovém případě bychom totiž měli po několika krocích v textuře pouze samé jedničky. Tuto hodnotu tedy propagujeme pouze v těch případech, kdy je obsažena ve všech čtyřech texelech z mipmapy na předchozí úrovni.

Na konci algoritmu máme tedy jednu hloubkovou texturu a k ní sadu mipmap, které můžeme používat. Výše uvedený postup generování mipmap z hloubkové textury ilustruje obrázek 7.11, který pro názornost zobrazuje pouze jednorozměrnou texturu. Jak takto vygenerované mipmapy mohou vypadat, vidíme na obrázku 7.12. Část kódu zodpovědného za tvorbu mipmap ukazuje příklad 7.3.

[7] Nekonečnem myslíme obecně hodnotu v takovém texelu, do kterého se nepromítnul žádný z objektů ve scéně. V našem případě bude takováto hodnota rovna kladné jedničce.

```
// onePixel je velikost jednoho pixelu textury, ze ktere vytvarime mipmapu
// halfPixel je potom vzdalenost do stredu tohoto pixelu
// offset definuje aktualni pozici 2x2 bloku sousednich pixelu v texture

// ziskame 4 hodnoty z pixelu textury, kterou mipmapujeme
vec4 color[4];
color[0] = texture2D(tex, vec2(halfPixel.x, halfPixel.y) + offset);
color[1] = texture2D(tex, vec2(onePixel.x + halfPixel.x, halfPixel.y) +
    offset);
color[2] = texture2D(tex, vec2(halfPixel.x, onePixel.y + halfPixel.y) +
    offset);
color[3] = texture2D(tex, vec2(onePixel.x + halfPixel.x, onePixel.y +
    halfPixel.y) + offset);

float mipmappedValue;

// pokud jsou vsechny ctyri hodnoty rovny 1, ulozime tez 1
if (color[0].r == 1.0 && color[1].r == 1.0 && color[2].r == 1.0
                && color[3].r == 1.0) {
        mipmappedValue = 1.0;
}
else {
        // jinak tyto hodnoty prevedeme tak, aby nasledny
        // test na maximum nemohly ovlivnit => nepropagujeme 1
        for (int i = 0; i < 4; ++i) {
                color[i].r = (color[i].r == 1.0) ? -1.0 : color[i].r;
        }

        // ukladame maximum z hodnot
        mipmappedValue = max(color[0].r, max(color[1].r, max(color[2].r,
                color[3].r)));
}

// do kanalu R ukladame toto maximum
gl_FragColor = vec4(mipmappedValue, 0.0, 0.0, 0.0);
```

Příklad 7.3: Vytváření mipmap hloubkové textury ve *fragment shaderu*

7.3.1.4 Zhodnocení přínosu optimalizace

Lze se domnívat, že přínos této optimalizace již nebude konstantně tak vysoký, jako tomu bylo u předchozích optimalizací, diskutovaných v kapitole 7.2. Ve scénách, kde nedochází k žádnému, nebo jen malému zastínění, bude použití této optimalizace zbytečné a naopak se projeví celkovým zpomalením vykreslování. To proto, že je potřeba pro každý snímek vytvořit novou mapu hloubky – samozřejmě pouze v dynamických scénách, tedy pokud se změnila geometrie scény či pozice světla – a poté provést výběr mipmapy a následné porovnání hloubek. Pokud je však ve scéně málo stínů, potenciál této optimalizace k eliminaci trojúhelníků zůstane nevyužit, přičemž potřebné výpočty budou stále probíhat.

Přesně tuto situaci ilustruje tabulka 7.2, ukazující naměřené hodnoty pro málo zastíněnou scénu s 18 tisíci trojúhelníky. Můžeme zde porovnat přínos minulé optimalizace, tedy

Scéna 7.1(a) 18k		EP	SM
P_c		54288	44760
FPS		17.9	17.5
T_{DM}	$[ms]$	-	4
T_{VG}	$[ms]$	23	30
T_{RF}	$[ms]$	0.8	0.8
T_Z	$[ms]$	28	20
T	$[ms]$	51	51 (55)

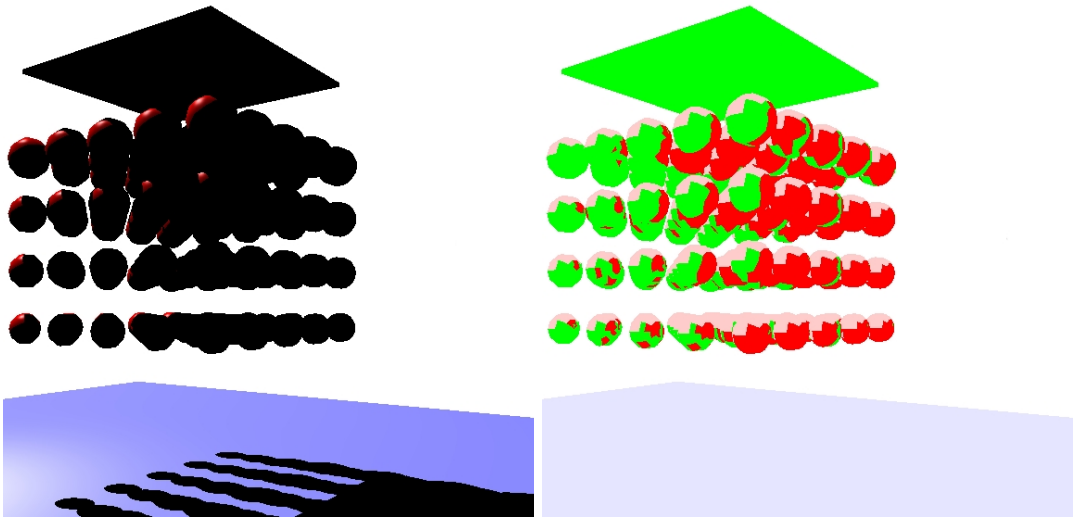
Tabulka 7.2: Porovnání přínosu přidání eliminace trojúhelníků zastíněných z pohledu světla na málo zastíněné scéně.

eliminaci trojúhelníků přivrácených ke světlu (sloupec EP), s přínosem přidání této (sloupec SM). Jelikož alespoň malé zastínění existuje, můžeme pozorovat snížení počtu generovaných primitiv, což způsobuje snížení doby rasterizace a následného testu zastínění ve *fragment shaderu*, protože se celkově nebude spouštět tolikrát. Zároveň je zde však vidět, že narostla složitost implementace *geometry shaderu*, která v této scéně způsobí, že optimalizace nemá ve výsledku žádný přínos. Naopak se nám ještě sníží počet FPS, jelikož je navíc potřeba vytvářet hloubkovou mapu.

Scéna 7.1(b) 18k		EP	SM
P_c		54336	27984
FPS		17.6	22.6
T_{DM}	$[ms]$	-	4
T_{VG}	$[ms]$	23	24
T_{RF}	$[ms]$	0.9	0.6
T_Z	$[ms]$	29	13
T	$[ms]$	53	38 (42)

Scéna 7.1(b) 76k		EP	SM
P_c		229500	108696
FPS		6.1	7.2
T_{DM}	$[ms]$	-	4
T_{VG}	$[ms]$	95	102
T_{RF}	$[ms]$	2	1.5
T_Z	$[ms]$	62	30
T	$[ms]$	159	133 (137)

Tabulka 7.3: Porovnání přínosu přidání eliminace trojúhelníků zastíněných z pohledu světla na středně zastíněné scéně.



(a) (b)

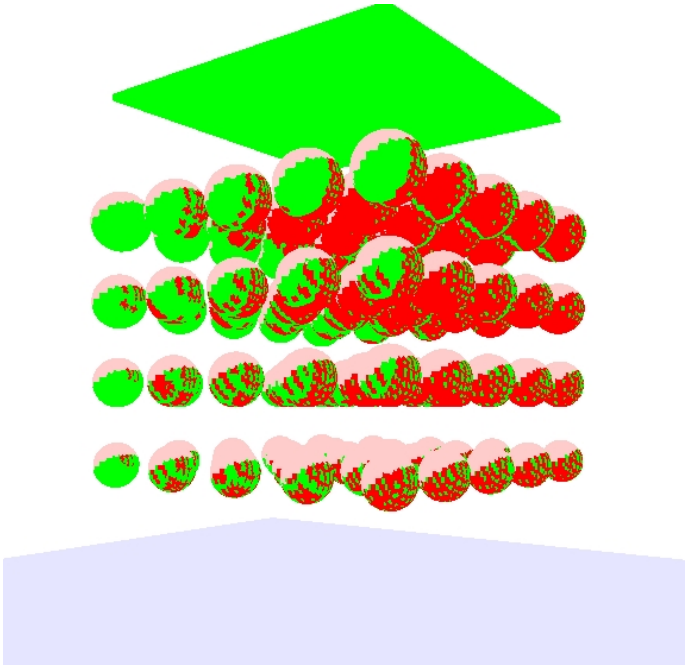
Obrázek 7.13: Efektivita eliminace trojúhelníků zastíněných z pohledu světla.

Pokud však máme více zastíněnou scénu, je situace jiná. Tabulka 7.3 nahoře nám ukazuje naměřené hodnoty pro středně zastíněnou scénu se stejnou geometrickou složitostí, jako scéna předchozí. Díky vyššímu zastínění však dochází k eliminaci zastíněných trojúhelníků ve větší míře, než v předchozím případě, a můžeme tak pozorovat razantně snížený počet generovaných primitiv, tedy stínových polygonů, oproti samotné eliminaci trojúhelníků přivrácených ke světlu. Následkem toho je pak zvýšení celkové rychlosti algoritmu, v našem případě přibližně o 28%. Pokud použijeme scénu zastíněnou nejvíce, dojde ke zrychlení až o 31%. Situaci při složitější geometrii středně zastíněné scény ukazuje tabulka dole, přičemž lze opět sledovat přibližně lineární zpomalení.

Vizuální představu o efektivitě této optimalizace nám poskytuje obrázek 7.13. Vlevo vidíme pohled na scénu, ve které obdélník vrhá stín na objekty pod ním. Vpravo je potom znázorněno, které trojúhelníky jsou díky této optimalizaci eliminovány, jelikož jsou vzhledem ke světlu zastíněny jiným objektem – v tomto případě tedy buď jinou koulí, nebo spíše tímto obdélníkem.

Při složitější geometrii lze na obrázku 7.14 pozorovat následky, které má na efektivitu této eliminace právě algoritmus pro výběr mipmapy. Mezi červenými plochami, tedy eliminovanými trojúhelníky, můžeme vidět zelené oblasti připomínající mřížku. To jsou právě ty trojúhelníky, které nebyly eliminovány na základě výběru příliš nízké úrovně mipmapy, jelikož leží na rozhraní několika sousedních texelů.

Benefity této optimalizace se tedy zásadně projevují ve více zastíněných scénách, zejména když máme jeden či více velkých objektů, vrhajících na zbytek scény stín, v blízkosti světla.



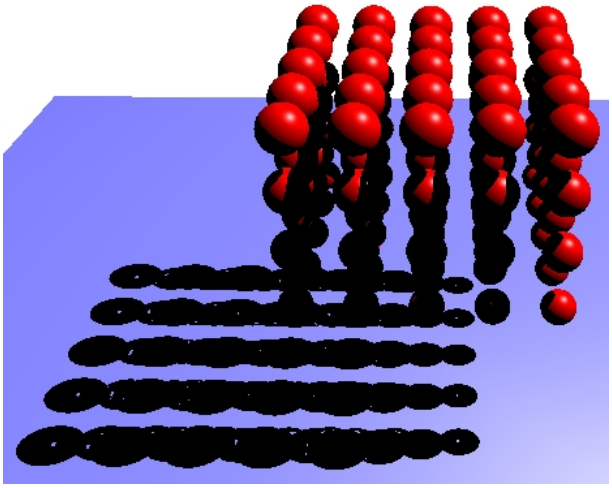
Obrázek 7.14: Efektivita eliminace trojúhelníků zastíněných z pohledu světla při složitější geometrii.

Trojúhelníky v takovýchto scénách mají potom dobré předpoklady být touto optimalizací eliminovány, tudíž nedochází ke zbytečnému generování z nich vytvořených stínových polygonů a následnému testu zastínění v oblastech, které jsou již zastíněny právě tělesy umístěnými blíže ke světlu. Pokud však máme pouze malé zastínění ve scéně, převáží tyto pozitiva režie, která je potřeba k výpočtu všech náležitostí této optimalizace.

7.3.1.5 Vznikající artefakty

Při použití této optimalizace vznikají artefakty v podobě světlých míst uprostřed stínů, jak ilustruje obrázek 7.15. Důvodem je nesprávná eliminace trojúhelníků, a tedy zrušení testu zastínění pro vzorky, na které můžou potenciálně vrhat stín. Ke vzniku těchto artefaktů dochází pouze při „jemnější“ geometrii, kdy jsou tělesa ve scéně tvořena malými trojúhelníky, a také hlavně tehdy, když jsou tyto trojúhelníky (téměř) kolmé na směr světla. Jejich vrcholy potom mají z pohledu světla stejnou či podobnou hloubku a mohou být na základě porovnání hloubek v plovoucí řádové čárce vyhodnoceny nesprávně jako zastíněné.

Řešení tohoto problému spočívá v konzervativním přístupu, a tedy můžeme buď k hodnotě hloubky z hloubkové mapy přidat určitou konstantu (nejlépe adaptivní, přizpůsobující se „jemnosti“ geometrie), nebo pro takovéto trojúhelníky vybírat nižší (méně detailní) úroveň mipmapy, ze které se následně hloubka získá.



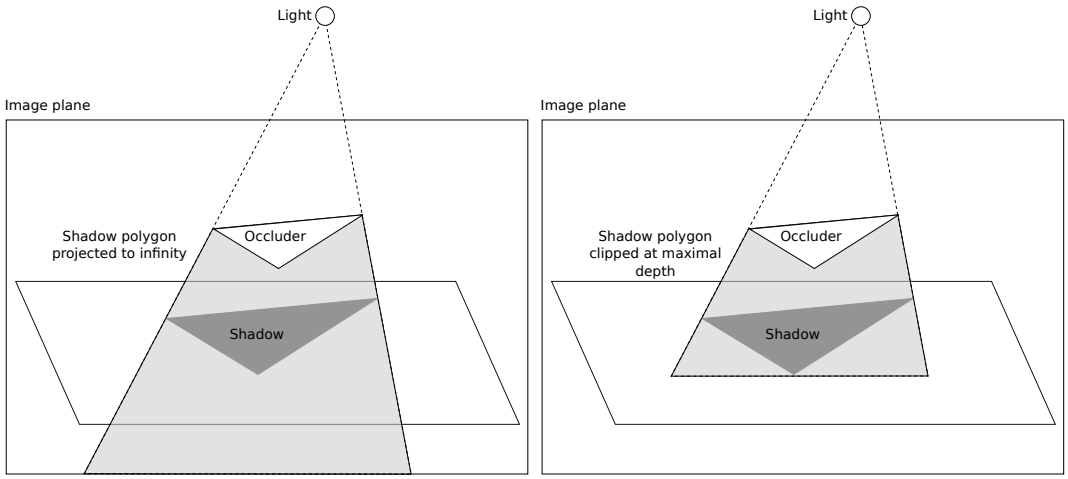
Obrázek 7.15: Artefakty způsobené nesprávnou eliminací trojúhelníků na základě porovnání jejich hloubky s hloubkovou mapou.

7.3.2 Oříznutí hloubky stínového polygonu

Doposud jsme generovali stínový polygon vždy protažený směrem od světla až do nekonečna, a tudíž pokrývající plochu začínající hranou zpracovávaného trojúhelníku a končící na okraji obrazovky. Časté jsou přitom případy, kdy je scéna zespoda omezena nějakou plochou, například podlahou v místnosti či terénem ve venkovní scéně. V těchto případech je tedy zbytečné generovat stínový polygon až do nekonečna, jelikož víme, že se stín nemůže nacházet dál od světla, než se nachází tato plocha. Bylo by tedy vhodné stínový polygon ukončit právě ve vzdálenosti této plochy od světla, což je ilustrováno na obrázku 7.16.

Abychom tedy mohli stínový polygon ořezat hloubkou nejvzdálenějšího objektu ve scéně z pohledu světla, musíme někam tyto největší vzdálenosti ukládat. A tak stejně, jako jsme v předchozí optimalizaci do hloubkové mapy ukládali vzdálenosti nejbližších objektů ke světlu, budeme nyní ukládat i vzdálenosti objektů nejvzdálenějších. Potom podobně, jako při eliminaci trojúhelníků na základě porovnání jejich vzdálenosti s hodnotou v hloubkové mapě, i zde budeme hledat referenční hodnotu hloubky pro každý trojúhelník. Tuto hodnotu však tentokrát nebudeme s ničím porovnávat, nýbrž ji použijeme přímo při generování stínového polygonu – a to jako novou z-ovou souřadnici vrcholů jeho vzdálené hrany z pohledu světla, která doposud ležela vždy v nekonečnu.

Nyní musíme vyřešit, jak uložit do textury zároveň nejbližší i nejvzdálenější hodnoty, a to co nejefektivněji, tedy pokud možno v jednom vykreslovacím průchodu. V předchozí optimalizaci jsme ukládali vzdálenosti bodů nejblíže ke světlu do kanálu R hloubkové textury, tudíž máme k dispozici ještě další tři kanály GBA (v případě použití textury se třemi barevnými



Obrázek 7.16: Stínový polygon projektovaný do nekonečna (vlevo) a ořezaný maximální hloubkou z pohledu světla (vpravo)

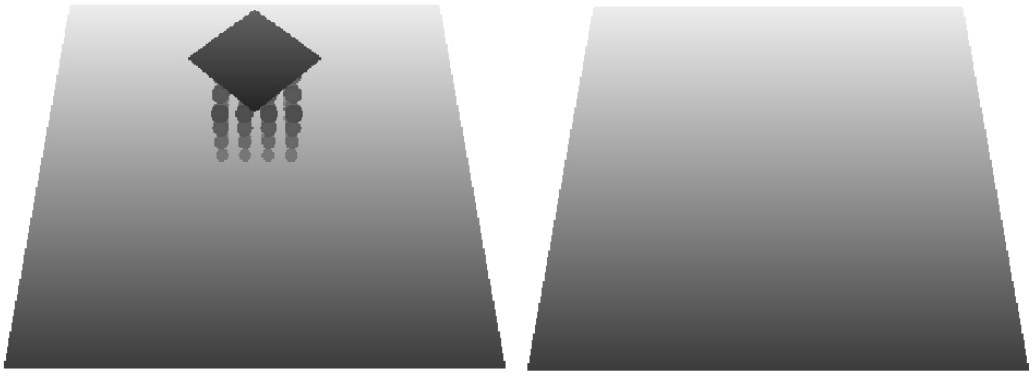
složkami a alfa kanálem). V kapitole 7.3.1.3, věnované generování hloubkové textury, jsme také záměrně uvedli, že s ohledem na právě popisovanou optimalizaci při ukládání nejbližších bodů vypínáme hloubkový test a provádíme ho sami pomocí míchání barev (*alpha blending*), kdy do textury ukládáme pouze nejnižší hodnoty. Pokud zároveň nastavíme výchozí hodnotu (barvu) textury na 1, máme zajištěno, že se nám do textury uloží pouze minimální hodnoty hloubky, přičemž nekonečno bude mít právě hodnotu 1.

Stejný způsob, tedy ukládání minimálních hodnot do hloubkové textury, lze paradoxně použít i při ukládání nejvzdálenějších hodnot – musíme však ukládat souřadnici z vynásobenou -1. Při ukládání minima ze záporných hodnot[8] zajistíme, že se nám bude ukládat vždy nejvzdálenější souřadnice, avšak s obráceným znaménkem. Původní hodnota (barva) textury, tedy hodnota nekonečna, však zůstává 1, což musíme později při práci s takto uloženými vzdálenostmi brát v potaz.

Pro správnou funkčnost musíme ještě rozšířit stávající algoritmus tvorby mipmap. Doposud jsme ve *fragment shaderu* pro každou úroveň LOD načítali hodnoty v kanálu R ze čtyř texelů v mipmapě na předchozí úrovni a do barvy fragmentu jsme zapisovali maximum z těchto hodnot, opět do kanálu R. Nyní potřebujeme číst a zapisovat i kanál G, přičemž tentokrát ukládáme minimum ze čtyř sousedních hodnot, přestože chceme uložit vždy největší vzdálenost od světla – to samozřejmě proto, že souřadnice v kanálu G jsou záporné.

Po provedení tohoto kroku nám tedy vznikne mipmapovaná hloubková textura, v jejímž kanálu R máme uloženy vzdálenosti nejbližších objektů viditelných z pohledu světla a v

[8] Zde je velmi důležité, abychom hodnoty ukládali do jednoho z vhodných formátů, které nám k tomuto účelu OpenGL poskytuje, jako například `GL_RGBA32F`. Při použití standardních formátů totiž dochází k ořezání barev (*color clamp*) do intervalu $[0, 1]$, což by znemožnilo ukládání záporných hodnot.



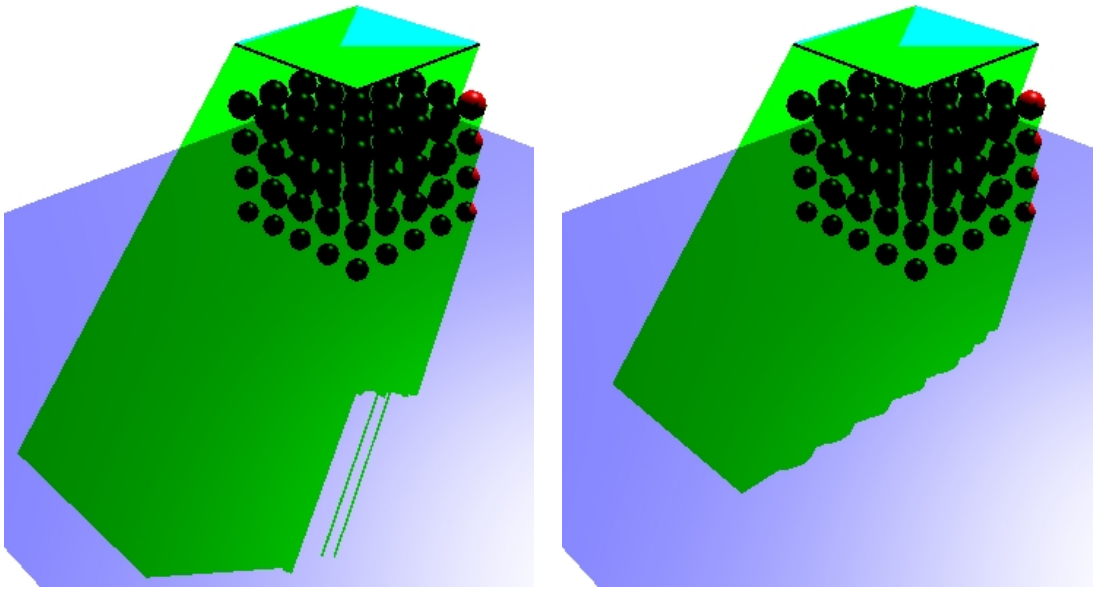
Obrázek 7.17: Hloubková textura ukládající nejbližší hodnoty (vlevo) a nejvzdálenější (vpravo)

kanálu G naopak vzdálenosti objektů, které jsou od světla nejdále, avšak s obráceným znaménkem. Ukázku této hloubkové textury můžeme vidět na obrázku 7.17. Vlevo vidíme právě kanál R, vpravo pak kanál G převedený do kladných hodnot.

Nalézt referenční hodnotu hloubky pro ořezání stínového polygonu lze, podobně jako dříve v kapitole 7.3.1.1, opět více způsoby. První způsob je naprosto stejný, jako dříve popisované získání texelu z mipmapy na takové úrovni LOD, kdy se celý trojúhelník ještě vejde do jednoho texelu. Jediný rozdíl je ten, že nebudeme z texelu číst jen hodnotu v kanálu R, nýbrž i G. Pozor si musíme dát pouze na to, že takto přečtená hodnota má záporné znaménko a tudíž ho musíme otočit. Vyjímkou je pouze hodnota nekonečna, která je reprezentována kladnou 1.

Po ořezání stínového polygonu takto získanou hloubkou mohou stínové polygony vypadat jako na obrázku 7.18 vlevo. Můžeme pozorovat, že zdaleka ne všechny stínové polygony jsou oříznuty přesně na úrovni spodní plochy. To je dáno právě neoptimálním výběrem úrovně LOD, kdy se vybere méně detailní mipmapa obsahující vzdálenější hodnotu, než je ve skutečnosti potřeba. Výsledkem je potom delší stínový polygon, než je nutné.

Stínový polygon lze však ořezat i přesně, jak je ukázáno na obrázku 7.18 vpravo. Tento postup však zahrnuje opět více čtení hloubkové textury. Narozdíl od hledání přesnější úrovně LOD v kapitole 7.3.1.1 však nevyžaduje žádné složité počítání navíc, a tudíž má skutečný potenciál být ve výsledku rychlejší, než přístup první. Můžeme totiž postupovat tak, že pro každý vrchol trojúhelníku přečteme z hloubkové textury odpovídající přesnou hloubku nejvzdálenějšího bodu ve scéně, a to tak, že budeme ignorovat mipmapy a použijeme klasické čtení texelu z textury. Takováto čtení tedy musíme provést tři, což sice může algoritmus zpomalit, ale potenciálně můžeme mnoho stínových polygonů oříznout znatelně více, než za použití přístupu s mipmapami. Při samotném generování stínového polygonu potom použi-



(a) (b)

Obrázek 7.18: Stínové polygony oříznuté maximální hloubkou z pohledu světla pomocí hodnoty získané z LOD mapy (a) a pomocí přesné hodnoty (b)

jeme přečtené hloubky k jeho ořezání.

7.3.2.1 Zhodnocení přínosu optimalizace

Díky této optimalizaci jsme tedy schopni stínový polygon oříznout maximální hloubkou ve scéně z pohledu světla, čímž můžeme zamezit spouštění *fragment shaderu* a následného testu zastínění na bodech, kam se již stín nemůže dostat. Tato optimalizace je však užitečná pouze v případech, kdy tuto plochu, která reprezentuje nejvzdálenější body z pohledu světla, vidíme i z pohledu kamery. Pokud ji totiž z kamery nevidíme, bude se stínový polygon generovat i nadále až k okraji obrazovky. O tomto nedostatku a jeho řešení pojednávají další kapitoly v této části.

Nejprve se však podívejme na tabulku 7.4, která ukazuje výsledky naměřené na středně zastíněné scéně s 18 tisíci trojúhelníky. Vidíme zde srovnání předcházející optimalizace, tedy eliminaci zastíněných trojúhelníků (sloupec SM), s přidáním optimalizace ořezávající stínový polygon pomocí maximální hloubky přečtené z hloubkové LOD mapy (sloupec DCL) a optimalizace ořezávající tento polygon pomocí přesné hodnoty z hloubkové mapy (sloupec DCE). V základním pohledu, tady takovém, jako je zobrazen na obrázku 7.1(b) na začátku kapitoly o optimalizacích, pozorujeme pouze malé zrychlení – přibližně o 4% až 6% – jelikož z tohoto pohledu není příliš velký prostor pro redukci stínového polygonu – ořezaný stínový polygon

Scéna 7.1(b) 18k		SM	DCL	DCE
P_c		27984	27984	27984
FPS		22.6	23.5	24
T_{DM}	$[ms]$	4	4	4
T_{VG}	$[ms]$	24	26	27
T_{RF}	$[ms]$	0.6	0.6	0.5
T_Z	$[ms]$	13	9.8	7.8
T	$[ms]$	38 (42)	36 (40)	35 (39)

Tabulka 7.4: Porovnání přínosu přidání ořezávání stínového polygonu maximální hloubkou.

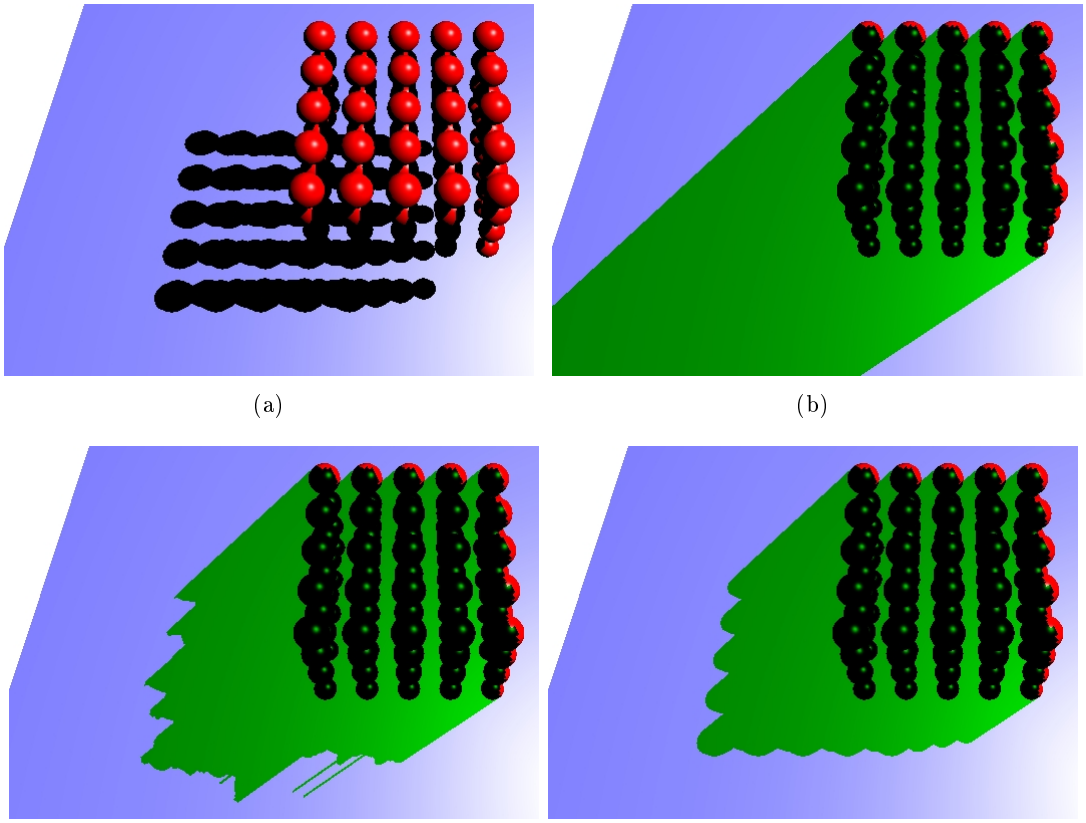
bude v tomto pohledu pouze nepatrně menší, než polygon protažený až k okraji obrazovky. Pokud však natočíme kameru například tak, jako je uvedeno na sérii obrázků 7.19, dosahuje optimalizace větších zrychlení, což můžeme vidět v tabulce 7.5. Konkrétně námi naměřená data nyní vykazují zrychlení až o 40% oproti verzi bez ořezávání hloubky stínového polygonu.

Scéna 7.19(a) 18k		SM	DCL	DCE
P_c		44760	44760	44760
FPS		14.5	20.4	20.5
T_{DM}	$[ms]$	4	4	4
T_{VG}	$[ms]$	30	32	33
T_{RF}	$[ms]$	1.1	0.5	0.4
T_Z	$[ms]$	32	9.7	8.5
T	$[ms]$	63 (67)	43 (47)	42 (46)

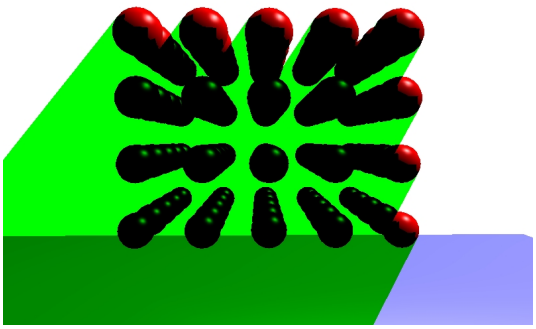
Tabulka 7.5: Porovnání přínosu přidání ořezávání stínového polygonu maximální hloubkou.

Zde se projevuje fakt, že rychlost algoritmu je opravdu závislá na pohledu, z jakého scénu pozorujeme – tedy na poloze a orientaci kamery. Díky těmto optimalizacím lze však výkyvy v rychlosti algoritmu v omezené míře zmírnit. Porovnáme-li hodnoty z této tabulky s tabulkou předchozí, vidíme, že změnou pohledu kamery algoritmus využívající jen optimaizaci s eliminací zastíněných trojúhelníků utrpěl zpomalení o více než 35%. Při využití optimalizace ořezání hloubky stínového polygonu je tato ztráta pouze přibližně 15%. Rozdíl mezi ořezáním stínového polygonu maximální hodnotou hloubky získanou z hloubkové LOD mapy a ořezáním přesnou hotnotou je na námi měřené scéně v obou případech nepatrný – čas, který ušetříme spouštěním testu zastínění je z velké části vykoupen dražším vícenásobným čtením z textury.

Situace, kdy nám tato optimalizace nepomůže, byla zmíněna již dříve. Jedná se o takové případy, kdy část plochy, která určuje maximální hloubku scény z pohledu světla, není z pohledu kamery vidět. Tehdy se stínový polygon i nadále protahuje až do nekonečna, tedy



Obrázek 7.19: Scéna zobrazená z jiného pohledu kamery (a) a stínové polygony bez ořezávání jejich hloubky (b), s ořezáváním maximální hloubkou scény získanou z hloubkové LOD mapy (c) a přesnou hodnotou maximální hloubky získanou z hloubkové mapy pro každý vrchol zvlášť (d).



Obrázek 7.20: Případ ořezání hloubky stínového polygonu maximální hloubkou scény z pohledu světla, kdy optimalizace nepřináší zrychlení.

až k okraji obrazovky, a optimalizace přináší pouze celkové zpomalení algoritmu, jelikož potřebné výpočty stále probíhají. Takovou situaci máme zobrazenu na obrázku 7.20 a její řešení diskutuje následující kapitola.

7.4 Využití informace o viditelné části scény

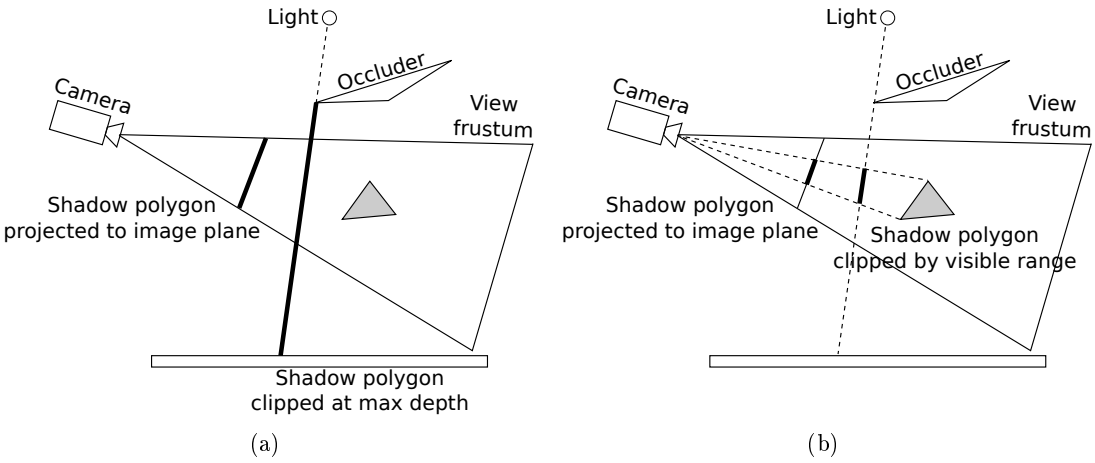
Optimalizace, kterými jsme se doposud zabývali, se na problém dívají pouze z jedné perspektivy – totiž z pohledu světla. Pouze z tohoto pohledu jsme na základě hloubkové mapy zjišťovali, které polygony jsou zajímavé pro výpočet a které ne, a pouze z pohledu světla jsme ukládali nejvzdálenější body scény a ořezávali jejich hloubkou stínový polygon. Problém zastínění scény řešený metodou vzorkování na GPU s následným testem viditelnosti však není takto jednostranný. Velmi hodnotné informace o scéně, respektive o tom, která její část je zajímavá pro výpočet, nám může poskytnout také druhá strana zainteresovaná v řešení tohoto problému – a tou je kamera.

Následující kapitoly popisují, jak lze využít pohled z kamery, lépe řečeno informaci o tom, které objekty jsou z kamery viditelné, k dalším optimalizacím algoritmu pohledově závislých stínových map.

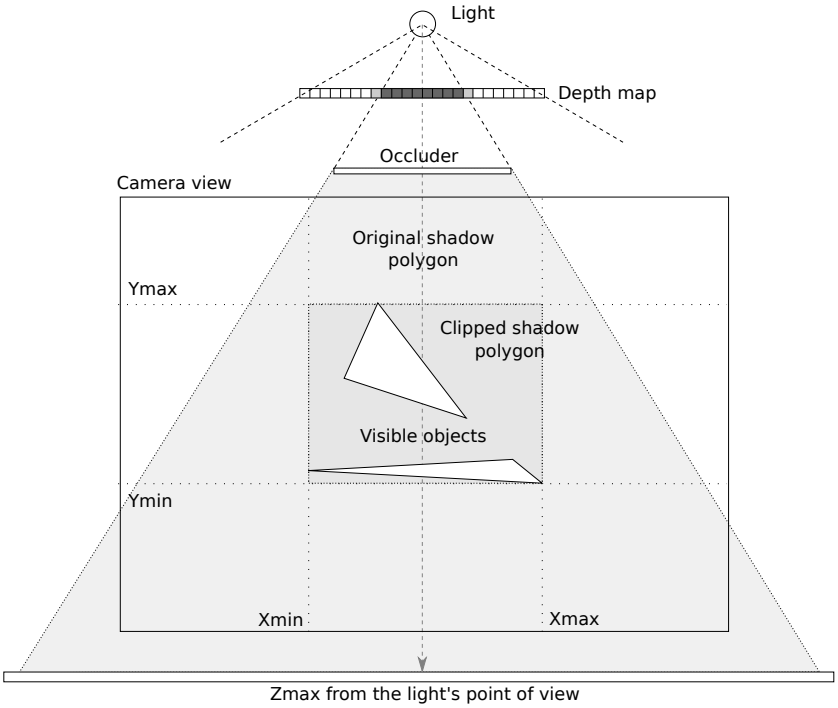
7.4.1 Ořezání stínového polygonu podle rozsahu souřadnic objektů viditelných z kamery

V této kapitole navážeme plynule na poslední optimalizaci, kterou bylo ořezávání stínového polygonu maximální hloubkou ve scéně z pohledu světla (kapitola 7.3.2). V některých případech, ilustrovaných na obrázku 7.21(a), nám však tato optimalizace nepomůže – jedná se o situace, kdy sice stínový polygon oříznemе maximální hloubkou scény z pohledu světla, ale plocha ležící v této hloubce se nachází mimo pohled kamery a stínový polygon se tedy nadále generuje až k okraji obrazovky. Pokud bychom však využili i informaci o objektech viditelných z kamery, mohli bychom dosáhnout lepších výsledků, jak naznačuje obrázek 7.21(b).

Na obrázku 7.22 je znázorněna podobná situace, tentokrát z pohledu kamery. Plocha, pomocí jejíž hloubky jsme díky minulé optimalizaci ořezali stínový polygon, není z pohledu kamery vidět a na stínový polygon tedy tato optimalizace nemá vliv. Z pohledu kamery však vidíme, že rozsah souřadnic objektů viditelných z kamery je X_{min}, X_{max} a Y_{min}, Y_{max}. Ořízneme-li tento polygon zmíněným rozsahem, tedy osově souměrným obdélníkem ohraničujícím viditelné objekty (takzvaným *bounding boxem*), můžeme výsledný polygon ještě o mnoho zmenšit.



Obrázek 7.21: Problém s využitím optimalizace ořezávání stínového polygonu maximální hloubkou z pohledu světla (a) a jeho řešení pomocí využití informace o viditelných objektech z pohledu kamery (b)



Obrázek 7.22: Ořezání stínového polygonu rozsahem souřadnic objektů viditelných z kamery

Abychom mohli tento rozsah souřadnic, získaných nyní pouze z objektů viditelných z kamery, použít k ořezání stínového polygonu, musíme ho opět uložit do textury z pohledu světla. Jedná se o obdobu hloubkové textury, ale vzhledem k tomu, že již neukládáme pouhé vzdálenosti bodů od světla, nýbrž rozsahy souřadnic, budeme tuto texturu dále v textu nazývat *texturou rozsahů*.

Z této textury budeme potom číst stejně, jako jsme četli z hloubkové textury u optimalizace eliminující trojúhelníky zastíněné vzhledem ke světlu – budeme tedy opět stejným způsobem vybírat úroveň LOD a číst texel z vybrané mipmapy. V něm však tentokrát nebude uložena vzdálenost bodů od světla, nýbrž rozsah souřadnic v osách X a Y, odpovídající rozsahu souřadnic objektů viditelných z kamery. V nejméně detailní mipmapě, obsahující jediný texel, bude tedy uložen rozsah v osách X a Y z celé části scény viditelné z kamery. V každé následující úrovni pak budeme tento rozsah dělit.

7.4.1.1 Získání a vykreslení vzorků viditelných z pohledu kamery

K tomu, abychom mohli do hloubkové textury uložit potřebné hodnoty, tedy rozsahy souřadnic v osách X a Y objektů viditelných z pohledu kamery, budeme potřebovat další vykreslovací průchod, při kterém opět transforumjeme příchozí vrcholy do pohledu světla. Standardně se však do grafického zobrazovacího řetězce posílá veškerá geometrie scény a nám by se takto uložily hodnoty získané ze všech objektů viditelných ze světla – což jsme vlastně dělali doposud při generování hloubkové mapy. My však potřebujeme ukládat hodnoty pouze té části scény, kterou vidí kamera, tedy části, která projde zobrazovacím řetězcem a uloží se ve finále do *framebufferu* (respektive její barevná informace). Potřebujeme tedy do zobrazovacího řetězce poslat k dalšímu zpracování pouze omezenou část scény, přičemž předem (na úrovni řídící aplikace) nevíme, která to bude.

Toho bychom nebyli schopni standardním způsobem docílit, nebýt techniky vykreslování do vertexových polí (*render-to-vertex-arrays*, RTVA), která byla představena v kapitole 4.3.5. Ta umožňuje hodnoty (v našem případě souřadnice), jež si uložíme do pixelů textury, vykreslit znovu jako pole vrcholů, a tím do zobrazovacího řetězce poslat pouze tu část geometrie, kterou potřebujeme.

Teoreticky budeme tedy potřebovat dva průchody navíc. První průchod pro vykreslení scény z pohledu kamery a uložení souřadnic jednotlivých bodů do textury, a druhý průchod pro vykreslení takto uložených souřadnic za použití RTVA. Pokud se však na problém podíváme blíže, zjistíme, že první průchod máme již vyřešen. Nejedná se totiž o nic jiného, než jsme již prováděli v prvním kroku algoritmu pohledově závislých stínových map – tedy krok získání textury se vzorky. Tato textura vzorků totiž obsahuje souřadnice všech bodů viditelných z kamery. Nyní tedy stačí, s odkazem na kapitolu 4.3.5, obsah této textury zkopírovat

do *pixel buffer objektu* a tento poté převést na *vertex buffer objekt* a vykreslit jeho obsah jako vrcholy.

Vykreslování provádíme s aktivovaným FBO a výstup tedy rovnou přesměrováváme do textury rozsahů, která je na počátku nastavena na samé jedničky. Jelikož chceme ukládat rozsah hodnot, tedy minimální a maximální hodnotu, postupujeme stejně, jako při generování hloubkové textury, která obsahovala minimální a maximální vzdálenost bodů od světla. Máme tedy opět vypnutý hloubkový test a naopak zapnuté míchání barev, přičemž ukládáme vždy minimum z hodnot.

Pomocí dvojice *vertex a fragment shaderu* následně definujeme, co se do textury rozsahů bude zapisovat. Ve *vertex shaderu* transformujeme příchozí vrcholy do ořezového prostoru světla, přičemž však do *fragment shaderu* posíláme jejich souřadnice v proměnné zároveň i v ořezovém prostoru kamery. Ve *fragment shaderu* tedy potom získáme pohled na viditelné vzorky scény z pohledu světla, ale do hodnoty (barvy) fragmentu ukládáme odpovídající souřadnice v ořezovém prostoru kamery, které navíc normalizujeme. Protože máme zapnuté míchání barev, pomocí kterého ukládáme pouze minimální hodnoty, musíme postupovat tak, že minimální souřadnice v osách X a Y zapisujeme standardně, kdežto maximální souřadnice v těchto osách ukládáme s obráceným znaménkem. Tímto způsobem jsme schopni uložit do čtyř kanálů $RGBA$ jedné textury postupně souřadnice X_{min}, $-X_{max}$, Y_{min} a $-Y_{max}$. Výše popsané lze v jazyce GLSL implementovat tak, jak je ukázáno v příkladě 7.4.

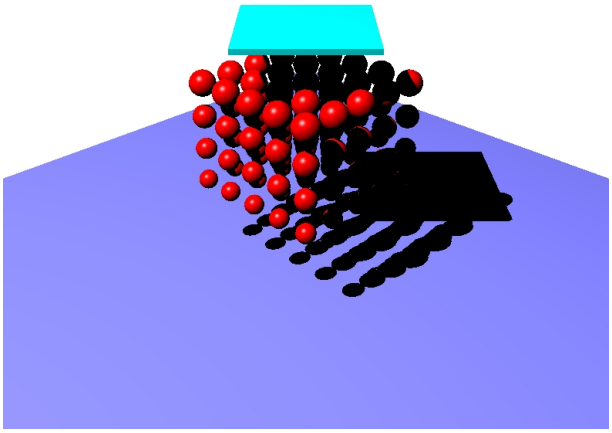
Obrázek 7.23(a) ukazuje scénu, respektive její viditelnou část, z pohledu kamery. Obrázek 7.23(b) potom ukazuje kanály R a G textury rozsahů, tedy X_{min} (vlevo) a X_{max} (vpravo). Vidíme, že na textuře je scéna zobrazena z pohledu světla, ale jednotlivé její texely obsahují rozsah hodnot v ose X z pohledu kamery. To lze poznat podle toho, že se barva textury mění zleva doprava (vzhledem k pozici kamery) postupně z černé na bílou[9]. Na obrázku 7.23(c) vidíme kanály B a A této textury, tedy rozsahy Y_{min} a Y_{max}.

Ke správné práci této optimalizace potřebujeme ještě vytvořit mipmapy z této textury, přičemž postupujeme podobně jako doposud při tvorbě mipmap. Opět v každé úrovni LOD bereme hodnoty čtyř texelů z mipmapy na předchozí úrovni, přičemž tentokrát ukládáme minimum ze všech těchto hodnot, a to v každém ze čtyř kanálů. Tím dosáhneme toho, že budeme mít v každém texelu stále minimální a maximální hodnoty v osách X a Y, přičemž maximální budou mít opačné znaménko.

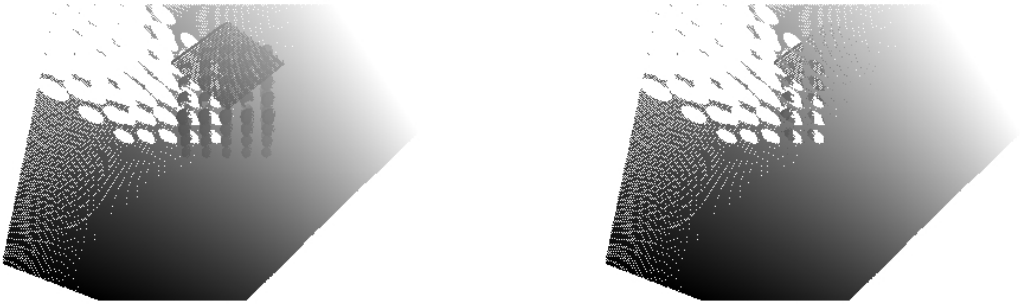
7.4.1.2 Ořezání stínového polygonu

Nyní se vracíme zpět ke generování stínového polygonu, a tedy do *geometry shaderu*. Naším úkolem bude z textury rozsahů získat odpovídající rozsah souřadnic pro právě zpracová-

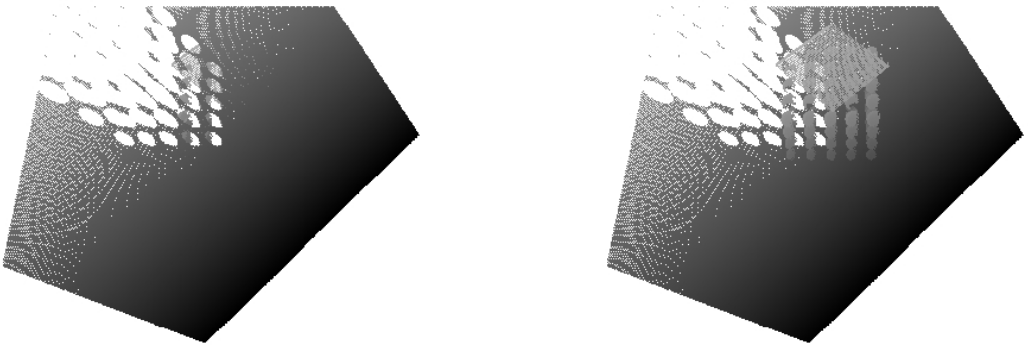
[9] Textura na obrázku je samozřejmě přemapována do rozsahu $[0, 1]$, aby byla zobrazitelná. Skutečná textura má však rozsahy hodnot v každém kanálu $[-1, 1]$



(a)



(b)



(c)

Obrázek 7.23: Scéna z pohledu kamery (a) a odpovídající vygenerovaná textura rozsahů souřadnic v osách X (b) a Y (c) z pohledu světla. Vlevo jsou pak minimální hodnoty, vpravo maximální v dané ose

```
Vertex shader:
----------------------------------------
varying vec4 vertexPositionCamera;

void main()
{
  // prichozi vrchol je jiz diky RTVA v pohledovem prostoru kamery
  // staci tedy provest projekcni promitani
  vertexPositionCamera = gl_ProjectionMatrix * gl_Vertex;

  // texturovaci matice obsahuje transformace potrebne k transformaci
  // vrcholu z pohledoveho prostoru kamery do orezoveho prostoru svetla
  gl_Position = gl_TextureMatrix[0] * gl_Vertex;
}

Fragment shader:
----------------------------------------
varying vec4 vertexPositionCamera;

void main()
{
  // diky michani barev se ukladaji pouze minimalni hodnoty
  // => muzeme tak ulozit jak minimalni, tak i maximalni rozsah XY
  vec2 rangeX = vec2(vertexPositionCamera.x / vertexPositionCamera.w,
                -vertexPositionCamera.x / vertexPositionCamera.w);
  vec2 rangeY = vec2(vertexPositionCamera.y / vertexPositionCamera.w,
                -vertexPositionCamera.y / vertexPositionCamera.w);

  // vysledny rozsah zapiseme
  gl_FragColor = vec4(rangeX, rangeY);
}
```

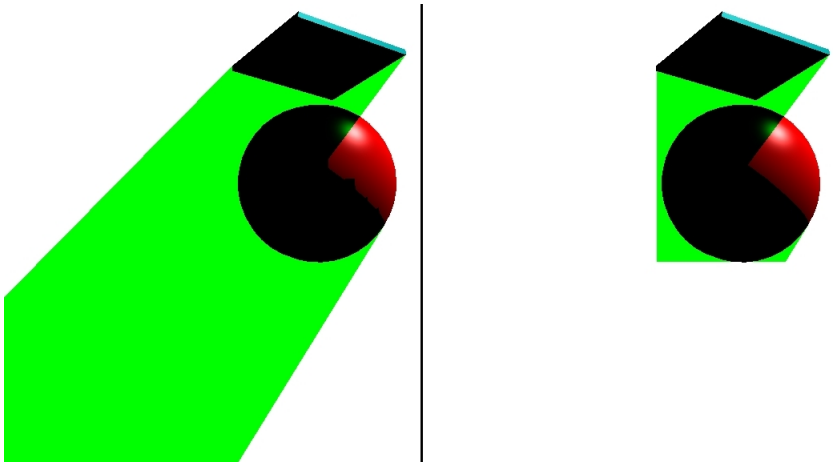
Příklad 7.4: Získání rozsahu XY z viditelných hodnot

vaný trojúhelník za použití správné mipmapy, z tohoto rozsahu vytvořit obdélník ohraničující objekty viditelné z kamery (*bounding box*) a tímto poté stínový polygon oříznout.

K dispozici tedy máme kromě hloubkové textury také texturu rozsahů, jejíž vytvoření popisovala předchozí kapitola. Výběr správné úrovně mipmapy provedeme stejně, jako při optimalizaci eliminující zastíněné trojúhelníky z pohledu světla (viz kapitola 7.3.1.1) – budeme tedy opět postupovat od nejnižší úrovně (nejméně detailní mipmapy) směrem k nejvyšší (nejvíce detailní) a vybereme takovou úroveň, kdy se ještě celý zpracovávaný trojúhelník vejde do jednoho texelu. Tento texel poté načteme z textury rozsahů.

Z kanálů R a B přečteme hodnoty X_{min} a Y_{min}, které tedy reprezentují levý spodní roh ořezávacího obdélníku. Kanály G a A potom, po jejich vynásobení -1, obsahují hodnoty X_{max} a Y_{max}, a tedy reprezentují pravý horní roh tohoto obdélníku. Jakmile tedy máme připraven tento obdélník, je vše připraveno k samotnému ořezání stínového polygonu.

V rámci této práce bylo implementováno ořezávání stínového polygonu algoritmem *Sutherland-Hodgman* [SH74]. Tento velmi jednoduchý algoritmus pracuje tak, že vstupní orientovaný po-



Obrázek 7.24: Ořezání stínového polygonu rozsahem souřadnic objektů viditelných z kamery. Vlevo původní stínový polygon, vpravo poté ořezaný algoritmem *Sutherland-Hodgman*

lygon postupně ořezává přímkami, ležícími na hranách osově souměrného ořezávacího okna. Výsledkem ořezávání konvexního polygonu je pak opět konvexní polygon, jehož vrcholy jsou vytvořeny dokola v pořadí daném jeho orientovanými hranami. Tím pádem je možné takto orientovaný polygon vykreslit bez dalšího řazení jako pás trojúhelníků (*triangle strip*).

Na obrázku 7.24 vlevo můžeme vidět původní stínový polygon a vpravo potom polygon ořezaný pomocí tohoto algoritmu obdélníkem získaným z textury rozsahů.

7.4.1.3 Zhodnocení přínosu optimalizace

Jak můžeme vidět právě na obrázku 7.24, tato optimalizace má potenciál opět velmi zredukovat velikost stínového polygonu, čímž lze značně omezit počet spouštění testů zastínění. Námi implementovaný algoritmus však nevykazuje dobré výsledky, jelikož v základní variantě není příliš vhodný pro implementaci na grafické kartě.

Hlavní funkce `clipPolygon`, zobrazená na příkladu 7.5, ořezává vstupní polygon postupně každou z přímek ležících na hranách ořezávacího okna. Body vzniklé po ořezání jednou přímkou jsou přeposlány do další fáze ořezání přímkou další, přičemž počet vrcholů se může změnit.

```
void clipPolygon(inout int count, inout vec2 point[MAX_CLIP_VERTICES],
    in Rect clipRect)
{
  // polygon postupne orezeme kazdou z orezavacich primek
  // vysledne souradnice vrcholu a jejich pocet budou ulozeny
  // ve vstupne-vystupnich promennych point[] a count
  clipPlane(count, point, clipRect, CLIP_LEFT);
```

```
  clipPlane(count, point, clipRect, CLIP_RIGHT);
  clipPlane(count, point, clipRect, CLIP_TOP);
  clipPlane(count, point, clipRect, CLIP_BOTTOM);
}
```

Příklad 7.5: Funkce ořezávající stínový polygon ořezovým oknem

Funkce *clipPlane*, jejíž implementace je ukázána na příkladu 7.6, ořezává polygon daný vstupními body danou ořezávací přímkou. Výstupem je množina bodů takto ořezaného polygonu.

```
void clipPlane(inout int count, inout vec2 points[MAX_CLIP_VERTICES],
    in Rect clipRect, in int side)
{
  // vytvorime pole orezanych vrcholu o maximalni velikosti
  vec2 newPoints[MAX_CLIP_VERTICES];
  int newCount = 0;

  // ziskame prvni bod
  vec2 prevPoint = points[count - 1];

  // opakujeme pro vsechny hrany prichoziho polygonu
  for(int i = 0; i < count; ++i) {

    // ziskame druhy bod
    vec2 nextPoint = points[i];

    // pokud je druhy bod uvnitr orezavaciho okna z pohledu primky
    if(isPointInside(nextPoint, clipRect, side)) {
      // a zaroven pokud je prvni bod venku
      if(!isPointInside(prevPoint, clipRect, side)) {
        // vypocitame prusecik a zaradime ho do pole orezanych vrcholu
        newPoints[newCount] = computeIntersection(
            nextPoint, prevPoint, clipRect, side); newCount++;
      }
      // do orezanych vrcholu zaradime i druhy bod
      newPoints[newCount] = nextPoint;
      newCount++;
    }
    // jinak pokud je druhy bod venku a prvni bod uvnitr
    else if(isPointInside(prevPoint, clipRect, side)) {
      // vypocitame prusecik a zaradime ho do pole orezanych vrcholu
      newPoints[newCount] = computeIntersection(
          prevPoint, nextPoint, clipRect, side); newCount++;
    }

    // druhy bod se stane prvnim a cyklus opakujeme
```

```
    prevPoint = nextPoint;
  }

  // ulozime pole orezanych vrcholu na vystup k dalsimu zpracovani
  count = newCount;
  for(int i = 0 ; i < count; ++i) {
    points[i] = newPoints[i];
  }
}
```

Příklad 7.6: Funkce ořezávající polygon vždy jednou z ořezových přímek

Funkce *isPointInside*, ukázaná na příkladu 7.7, na základě dané ořezávací přímky určuje, zda bod leží vně nebo uvnitř ořezávacího okna.

```
bool isPointInside(in vec2 p, in Rect r, in int side)
{
  switch(side) {
    case CLIP_LEFT:
      return p.x >= r.left;
    case CLIP_RIGHT:
      return p.x <= r.right;
    case CLIP_TOP:
      return p.y <= r.top;
    case CLIP_BOTTOM:
      return p.y >= r.bottom;
  }
}
```

Příklad 7.7: Funkce určující zda se bod nachází vně nebo uvnitř ořezávacího okna vzhledem k dané přímce

Funkce *computeIntersection* počítá průsečík úsečky dané dvěma body s danou ořezávací přímkou. Tato funkce je ukázána na příkladu 7.8.

```
vec2 computeIntersection(in vec2 p, in vec2 q, in Rect r,  in int side)
{
  // souradnice pruseciku a parametry primky
  vec2 intPoint;
  float a = (q.y - p.y) / (q.x - p.x);
  float b = p.y - p.x * a;

  // prusecik vypocteme na zaklade znalosti orezavaci primky
  switch(side) {
    case CLIP_LEFT:
      intPoint.x = r.left;
      intPoint.y = intPoint.x * a + b;
      break;
```

```
    case CLIP_RIGHT:
      intPoint.x = r.right;
      intPoint.y = intPoint.x * a + b;
      break;
    case CLIP_TOP:
      intPoint.y = r.top;
      intPoint.x = (!isinf(a)) ? (intPoint.y - b) / a : p.x;
      break;
    case CLIP_BOTTOM:
      intPoint.y = r.bottom;
      intPoint.x = (!isinf(a)) ? (intPoint.y - b) / a : p.x;
      break;
  }

  return intPoint;
}
```

Příklad 7.8: Funkce počítající průsečík úsečky s přímkou

Hlavní problém naší implementace tkví patrně v tom, že v základní verzi obsahuje algoritmus cykly s proměnnou délkou, které nemohou být optimalizovány při kompilaci na grafické kartě. Přítomnost z těchto cyklů vyplývá ze skutečnosti, že počet vrcholů ořezávaného polygonu se může měnit během jeho ořezávaní každou z ořezávacích přímek. Konečný počet vrcholů je znám až na konci algoritmu a cylky je tak složité odstranit. Jedním z řešení by mohlo být převést tyto cykly na cykly s pevnou délkou a potom uvnitř každé iterace testovat, zda hodnota v poli je platná či nikoliv. Na základě výsledku tohoto testu bychom potom buď pokračovali, nebo cyklus ukončili – toto však kompilátor též nemůže predikovat a odstranění tohoto problému je tak v základní implementaci algoritmu složité.

Dalším problémem by mohla být přítomnost velkého množství podmínek (`if`) a přepínačů (`switch`), které nám pomáhají určit, jak se má například vypočítávat průsečík pro různé ořezávací přímky, či jaká je poloha bodu vůči této přímce. Částečným odstraněním těchto podmínek a přepínačů jsme však nedocílili výrazného zrychlení algoritmu, a tak hlavním problémem patrně stále zůstává proměnná délka cyklů. Neoptimální pro zpracování grafickou kartou je též fakt, že nemůžeme emitovat vrcholy v průběhu ořezávání, jelikož konečný výsledek je znám, až když ořežeme polygon poslední přímkou. Tato skutečnost také může výrazně zpomalit chod algoritmu.

Pro efektivní implementaci této optimalizace na grafické kartě by tedy bylo potřeba použít algoritmus jiný, jenž dokáže ideálně ořezat v každé iteraci jednu hranu stínového polygonu do její finální podoby, abychom ji mohli ihned poslat k dalšímu zpracování – tímto bychom se mohli cyklům s proměnnou délkou vyhnout. Z ořezaných hran bychom poté za předpokladu, že si budeme pamatovat první emitovaný vrchol, mohli postupně vykreslit vějíř trojúhelníků (*triangle fan*). Takový algoritmus by byl vhodným kandidátem pro implementaci na grafické

kartě. Jeho nalezení, a tudíž efektivní implementace této optimalizace, je však již ponechána čtenáři.

7.5 Eliminace vzorků ležících před rovinou stínového polygonu

Veškeré dosud prezentované optimalizace algoritmu pohledově závislých stínových map pracovaly pouze na úrovni *geometry procesoru*. Buď tedy rovnou eliminovaly trojúhelníky, jimiž vržený stín nemůže přispět k výsledné stínové mapě, nebo omezovaly velikost generovaného stínového polygonu. Následující optimalizace se od těch předcházejících liší tím, že pracuje až ve *fragment procesoru* a bude tedy omezovat počet testů zastínění až těsně před jejich samotným provedením.

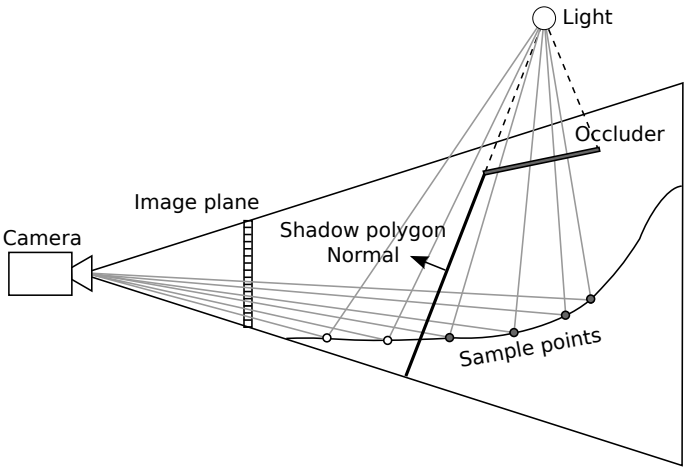
V kapitole 7.2.1 jsme se seznámili s první z optimalizací omezujících velikost oblasti generované v *geometry shaderu* a této oblasti též od této chvíle říkáme stínový polygon. Díky této optimalizaci již nemusíme test zastínění provádět s každým bodem z textury vzorků, ale můžeme z každého trojúhelníku vrhajícího stín vytvořit pouze jeden či dva stínové polygony, které v pohledu z kamery vymezují oblast výskytu možného vrženého stínu. Test zastínění pak provádíme pouze na těch vzorcích, které jsou v tomto pohledu stínovými polygony pokryty.

Výše zmíněná optimalizace tedy může radikálně snížit počet spuštění *fragment shaderu* a tedy počet provedených testů zastínění. Jeden či dva přivrácené stínové polygony vytvořené z daného trojúhelníku a promítnuté do přední ořezávací roviny kamery nám však dokáží určit pouze dvourozměrnou oblast, ve které se může stín nacházet, a nemohou tak postihnout rozměr třetí.

Pro ilustraci výše popsaného se nyní podívejme na obrázek 7.25. Na něm vidíme trojúhelník vrhající stín a jeho stínový polygon přivrácený ke kameře s normálou směřující směrem k ní. Test zastínění se však provádí pro všechny vzorky, tedy i ty, které leží před rovinou stínového polygonu a které nemohou být nikdy daným trojúhelníkem zastíněny. Pro tyto vzorky je tedy test zastínění prováděn zbytečně.

Řešení tohoto problému je jednoduché. Stačí nám určit rovinu každého generovaného stínového polygonu a následně provádět test na vzájemnou polohu této roviny s bodem z textury vzorků. Pokud se daný bod nachází před touto rovinou, nemusíme provádět následný test viditelnosti[10], jelikož ihned víme, že tento vzorek nemůže být zastíněn. V opačném případě provedeme test standardním způsobem.

[10] Testem viditelnosti myslíme pouze část programu počítající průsečík trojúhelníku se spojnicí světla a bodu z textury vzorků, a následnou klasifikaci vzorku jako zastíněného či osvětleného. Testem zastínění pak myslíme celý program včetně získání bodu z textury.



Obrázek 7.25: Princip zahazování vzorků ležících mimo rovinu stínového polygonu

Jelikož předpokládáme, že nejkritičtější fází pro rychlost celého algoritmu je fáze výpočtu ve *fragment shaderu*, počítáme parametry rovnice roviny stínového polygonu již v *geometry shaderu*. Obecná rovnice roviny je dána vztahem $Ax + By + Cz + D = 0$, kde A, B, C jsou souřadnice normály kolmé na tuto rovinu a parametr D musíme vypočítat. Po spočtení pro každý stínový polygon uložíme tyto čtyři parametry do jednoho čtyřsložkového vektoru a ten předáváme do *fragment shaderu*. V něm poté pouze dosadíme do rovnice za x, y, z souřadnice bodu z textury vzorků a přičteme parametr D. Pokud je výsledná hodnota větší než nula, leží bod před touto rovinou a můžeme instanci programu ukončit (`discard`), jelikož tento bod nemůže být nikdy zastíněn. Implementace výše popsaného je uvedena na příkladech 7.9 a 7.10, které ukazují relevantní části *geometry* a *fragment* shaderu.

```
Geometry shader
-----------------------------------------
for (int i = 0; i < gl_VerticesIn; ++i)
{
  // index dalsiho vrcholu
  int next = (i+1) % gl_VerticesIn;

  // vypocteme normalu stinoveho polygonu, promenna dir[] obsahuje
  // souradnice smeroveho vektoru od svetla k vrcholu trojuhelniku
  // (tato normala vzdy smeruje smerem ven ze stinoveho objemu)
  vec3 shadowPlaneNormal = normalize(cross(
      gl_PositionIn[next].xyz - gl_PositionIn[i].xyz, dir[i]));

  // spocitame parametr D = -(Ax + By + Cz), kde xyz jsou souradnice
  // bodu leziciho v rovine
  float paramD = -dot(shadowPlaneNormal, gl_PositionIn[i].xyz)
```

```
  // souradnice XYZ normaly jsou parametry ABC rovnice roviny
  // parametry ABCD ulozime do promenne, ktera je predana
  // do fragment shaderu
  shadowPlaneEquation = vec4(shadowPlaneNormal, paramD);

  // pokracujeme v generovani stinovych polygonu
  ...
}
```

Příklad 7.9: Výpočet parametrů roviny stínového polygonu

```
Fragment shader
----------------------------------------
// do rovnice Ax + By + Cz + D (parametry ABCD mame z geometry shaderu)
// dosadime bod ziskany z textury vzorku.
float dotP = (dot(shadowPlaneEquation.xyz, point.xyz)
    + shadowPlaneEquation.w);

// Pokud je hodnota vetsi nez 0, lezi bod pred rovinou stinoveho
// polygonu a muzeme tedy ukoncit test viditelnosti
if (dotP > 0.0) {
  discard;
}
```

Příklad 7.10: Test polohy vzorku vůči rovině stínového polygonu

7.5.1 Zhodnocení přínosu optimalizace

Účinnost této optimalizace může být sporná, a to i přesto, že může potenciálně vyloučit mnoho vzorků z testu viditelnosti. Jelikož je prováděna až ve fázi *per-fragment* operací, musí být vždy proveden výpočet výše zmíněných parametrů rovnic stínových polygonů v *geometry shaderu*, dále dojde k jejich rasterizaci, spuštění *fragment shaderu* a výpočtu polohy bodu a roviny. Tyto všechny operace se musí provést, aniž předem víme, zda bude vzorek nakonec vyloučen či nikoliv. Efektivita optimalizace tedy bude dána rovnicí, kde na jedné straně bude čas ušetřený tím, že v některých případech nebudeme provádět test viditelnosti, a na druhé straně bude čas potřebný na výše zmíněné výpočty. Pokud tedy budeme například předpokládat, že test viditelnosti je x-krát náročnější než potřebné výpočty, musíme doufat, že se touto optimalizací vyloučí alespoň $1/x$ všech testovaných vzorků.

Tabulka 7.6, která porovnává přínos přidání této optimalizace (sloupec EZ) k optimalizaci eliminace zastíněných trojúhelníků (sloupec SM), ukazuje právě situaci, kdy se obě strany této rovnice rovnají. Čas, který ušetříme spouštěním testu zastínění a který se projeví právě sníženou dobou výpočtu ve *fragment shaderu*, je přesně kompenzován časem potřebným k výpočtu parametrů roviny stínového polygonu v *geometry shaderu* a ve fázi rasterizace s

Scéna 7.1(a) 18k		SM	SM + EZ
P_c		44760	44760
FPS		17.5	17.5
T_{DM}	$[ms]$	4	4
T_{VG}	$[ms]$	30	31
T_{RF}	$[ms]$	0.9	1
T_Z	$[ms]$	19	17
T	$[ms]$	50 (54)	50 (54)

Tabulka 7.6: Porovnání přínosu přidání zrušení testu viditelnosti pro vzorky před rovinou stínového polygonu.

spouštení *fragment shaderu*, kde musíme tyto parametry předávat. Hodnoty byly naměřeny opět v základním pohledu na scénu s malým zastíněním a 18 tisíci trojúhelníky (obrázek 7.1(a)).

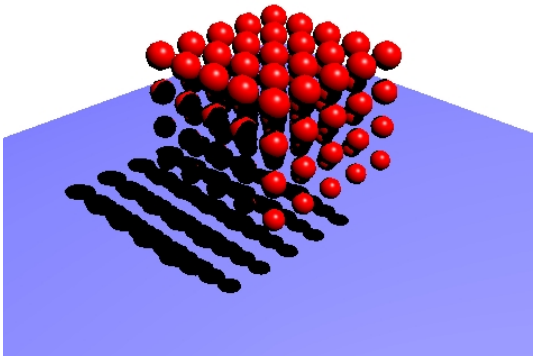
Scéna 7.1(a) 18k		SM	SM + EZ
P_c		44760	44760
FPS		18.7	19.2
T_{DM}	$[ms]$	4	4
T_{VG}	$[ms]$	30	31
T_{RF}	$[ms]$	0.7	0.8
T_Z	$[ms]$	17	14
T	$[ms]$	48 (52)	46 (50)

Tabulka 7.7: Porovnání přínosu přidání zrušení testu viditelnosti pro vzorky před rovinou stínového polygonu – jiný pohled na scénu

Abychom zjistili, zda tato optimalizace může vůbec být přínosem, musíme polohovat kameru do jiné pozice. Ve scéně na obrázku 7.26 již lze pozorovat drobné zrychlení algoritmu, které ilustruje tabulka 7.7. Zde je již čas ušetřený ve *fragment shaderu* natolik významný, že převažuje čas potřebný na „před-výpočet“ v *geometry shaderu* a zvyšuje se tak rychlost algoritmu.

Celkově je však přínos této optimalizace velmi sporný a v závislosti na pohledu se na této scéně s 18 tisíci trojúhelníky pohybuje v rozmezí ± 1 FPS. Na více zastíněných scénách je toto rozmezí podobné, ve scénách se složitější geometrií tato optimalizace naopak ztrácí na efektivitě. To je dáno skutečností, že se generuje mnoho stínových polygonů, pro které se musí počítat parametry roviny, ve které leží, ale polygony mají velmi malou plochu, a tak je počet potenciálně eliminovaných testů viditelnosti malý.

Pokud by se podařilo zrychlit část výpočtu v *geometry shaderu*, a tudíž by tato optimalizace měla vyšší efektivitu, by bylo možné ji rozšířit i o testy s rovinami ostatních dvou



Obrázek 7.26: Scéna s 18k trojúhelníky v pohledu, ve kterém se projeví přínos přidání optimalizace rušící test viditelnosti pro vzorky před rovinou stínového polygonu.

stínových polygonů. Tím bychom se vlastně přiblížili principu metody stínových těles, jelikož bychom pak počítali, zda se vzorek nachází uvnitř stínového objemu ohraničeného těmito stínovými polygony, a teoreticky by tak bylo možné vyloučit ještě více vzorků z testu viditelnosti. Na základě výše naměřených časů se však lze oprávněně domnívat, že toto rozšíření již pravděpodobně nebude přinášet dobré výsledky, jelikož program stráví více času prováděním potřebných výpočtů než ušetří tím, že se v některých případech test viditelnosti neprovede.

7.6 Shrnutí přínosu optimalizací

Optimalizováním původního algoritmu pohledově závislých stínových map, který výpočet ostrých stínů řešil hrubou silou a pro každý trojúhelník ve scéně tak počítal zastínění všech viditelných vzorků, se nám podařilo rychlost vykreslování dostat ze stavu nepoužitelné pro interaktivní komunikaci až na hranici reálného času. Zbývá vyřešit, v jakém pořadí uvedené optimalizace zapojit tak, aby bylo jejich použití co nejefektivnější – tomuto zapojení budeme říkat *optimalizační řetězec*.

Prvním kritériem pro zařazení jednotlivých optimalizací do optimalizačního řetězce je samozřejmě místo v grafickém zobrazovacím řetězci, ve kterém optimalizace probíhají. Pořadí optimalizací tedy musíme přizpůsobit skutečnosti, že se geometrický procesor nachází před fragmentovým procesorem. Je tedy jasné, že optimalizace eliminující vzorky nacházející se před rovinou stínového polygonu, která probíhá až ve *fragment shaderu*, bude na konci optimalizačního řetězce.

Všechny ostatní optimalizace probíhají v geometrickém procesoru, konkrétně potom v programu, který je na tomto procesoru prováděn. Jak jsme popsali v úvodu kapitoly věnující

se optimalizacím, dělíme je na dva typy. První typ optimalizací eliminuje příchozí trojúhelníky, jimiž vržený stín nemůže přispět k výsledné stínové mapě, druhý potom zmenšuje velikost stínových polygonů, které jsou z hran těchto trojúhelníků vytvářeny. Z toho vyplývá další řazení – první typ optimalizací bude zařazen jako první, abychom zbytečně negenerovali stínové polygony z trojúhelníků, které budou nakonec eliminovány. Druhý typ tedy zařadíme až za optimalizace prvního typu. V rámci těchto typů nám potom intuitivně vyplývá jediné možné další řazení, a sice řazení podle potenciálu optimalizací zrychlit běh algoritmu. Jako první budou tedy logicky zařazeny ty optimalizace, které mají tento potenciál nejvyšší.

Finální zařazení jednotlivých optimalizací na geometrickém procesoru ilustruje obrázek 7.27. Sjednocení oblastí, kde se provádí test zastínění, je pro ilustraci vyznačeno zelenou barvou.

Číslo 0 tedy ukazuje původní verzi algoritmu bez optimalizací, kdy se pro každý trojúhelník testovalo zastínění na všech viditelných vzorcích. Na námi měřené scéně však vykreslení jednoho obrázku trvalo několik vteřin, což je v praxi samozřejmě nepoužitelné.

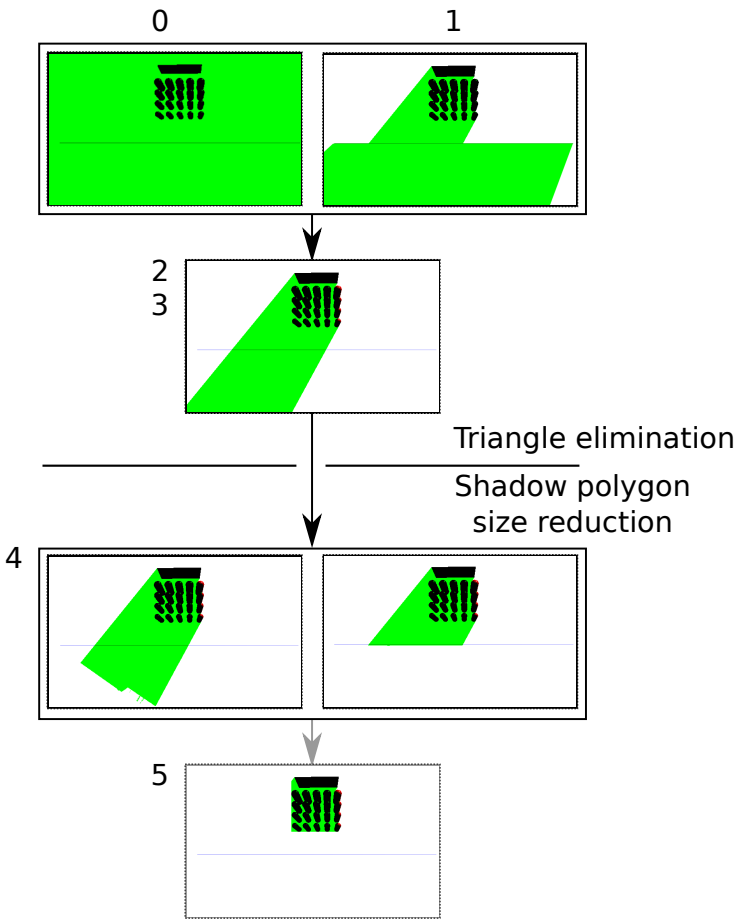
Číslo 1 i následující obrázky ukazují pro názornost oblast stínového polygonu, i když jeho generování se provádí až poté, co jsou eliminovány všechny nadbytečné trojúhelníky. Optimalizace generující stínové polygony (kapitola 7.2.1) však má pravděpodobně největší přínos a bez ní by nebylo možno úspěšně dále optimalizovat. Zrychlení, které přináší, by se dalo pravděpodobně vyjádřit ve stovkách procent.

Čísla 2 a 3 – obě na jednom obrázku, jelikož ve sjednocení stínových polygonů není mezi těmito optimalizacemi vidět rozdíl – potom ukazují stínové polygony po eliminaci trojúhelníků přivrácených ke světlu (kapitola 7.2.2), a zároveň i eliminaci trojúhelníků zastíněných z pohledu světla (kapitola 7.3.1). Eliminace trojúhelníků přivrácených ke světlu přinesla zrychlení algoritmu o více než 60%, eliminace zastíněných trojúhelníků potom v závislosti na velikosti zastínění scény až dalších 30%.

Nyní jsme ukončili fázi eliminace trojúhelníků a vstupujeme do fáze generování stínového polygonu. Ten je nejprve protažen až do nekonečna (k okraji obrazovky), jak vidíme například u čísel 2 a 3 a nyní budeme omezovat jeho velikost.

Číslo 4 ilustruje oříznutí stínového polygonu maximální hloubkou scény z pohledu světla (kapitola 7.3.2), přičemž obrázek vlevo zobrazuje přístup získání této hloubky z hloubkové LOD textury, kdežto obrázek vpravo zobrazuje použití přesné hodnoty pro každý vrchol. Na námi měřené scéně byly oba přístupy téměř rovnocenné a v závislosti na pohledu, tedy umístění kamery a její orientaci, mohou přinést zrychlení až o dalších 40%.

Číslo 5 ukazuje předposlední námi prezentovanou optimalizaci, a sice ořezání stínového polygonu obdélníkem ohraničujícím rozsah objektů viditelných z kamery (kapitola 7.4). Tuto optimalizaci se nám však bohužel nepodařilo efektivně implementovat. Při pohledu na obrá-



Obrázek 7.27: Optimalizační řetězec na úrovni geometrického procesoru

zek je ale zřejmé, že efektivní implementace algoritmu ořezání stínového polygonu by mohla přinést další významné zrychlení.

Poslední optimalizací, která již na obrázku není, je zahazování vzorků ležících před rovinou stínového polygonu (kapitola 7.5). Ta je prováděna až na úrovni fragmentového procesoru a díky ní je možné v některých případech nespouštět test viditelnosti na vzorcích, které nemohou ležet ve stínu vrženém daným trojúhelníkem. Měření však ukázala, že na námi testované scéně je přínos této optimalizace sporný a s rostoucí složitostí scény naopak celý algoritmus zpomaluje.

Následující tabulka 7.8 shrnuje přínos všech optimalizací, u kterých bylo na námi měřené scéně naměřeno nějaké zrychlení. Ukazuje tedy vliv první optimalizace, tedy projekce stínového polygonu, na rychlost algoritmu a porovnává ho s vlivem, který mají na jeho rychlost všechny následující optimalizace (s odkazem na obrázek 7.27 tedy optimalizace 2, 3 a 4). Ve třetím sloupečku jsou pak uvedeny procentuální změny, kterých jsme v jednotlivých měře-

ných hodnotách dosáhli optimalizováním. Údaje byly měřeny na nejvíce zastíněné scéně na obrázku 7.1(c) se 76 tisíci trojúhelníky.

Scéna 7.1(c) 76k		Bez optimalizací	S optimalizacemi	Změna
P_c		456372	89184	-80%
FPS		3.7	7.9	+214%
T_{DM}	$[ms]$	-	4	-
T_{VG}	$[ms]$	153	102	-33%
T_{RF}	$[ms]$	3.1	1.0	-67%
T_Z	$[ms]$	77	17	-78%
T	$[ms]$	233	120 (124)	-47%

Tabulka 7.8: Porovnání přínosu všech optimalizací

Poslední tabulka 7.9 v této sekci ukazuje maximálně dosažený počet FPS pro jednotlivé kombinace zastínění scény a její geometrické složitosti za použití přínosných optimalizací. Používáme tedy projekci stínového polygonu, eliminaci trojúhelníků přivrácených ke světlu, eliminaci zastíněných trojúhelníků z pohledu světla a ořezání stínového polygonu maximální hloubkou scény, získanou přesně pro každý vrchol.

Zastínění / Složitost geometrie	18k	76k	174k
Malé - scéna 7.1(a)	19.6	6.0	2.8
Střední - scéna 7.1(b)	24.4	7.5	3.3
Velké - scéna 7.1(c)	25	7.9	3.4

Tabulka 7.9: Přehled dosažených FPS na měřené scéně a jejích variantách za použití všech přínosných optimalizací.

Při pohledu na tuto tabulku lze říci, že jsme pro scény s 18 tisíci trojúhelníky dosáhli rychlosti vykreslování téměř na hranici reálného času, který bývá definován jako hodnoty rovné a vyšší 25 FPS. Při vyšší složitosti geometrie se pohybujeme v čase interaktivním. Na nejsložitější scéně již algoritmus nevykazuje přijatelné výsledky, ale tato scéna je již nad rámec našeho cíle, který jsme si stanovili v úvodu – tedy dosáhnout reálného, nebo alespoň interaktivního času pro scény s řádově desítkami tisíc polygonů.

Tyto naměřené rychlosti se však týkají námi přesně určeného pohledu na scénu. Pokud kameru přiblížíme tak, že budou stíny pokrývat větší část obrazovky, rychlost vykreslování klesne, jelikož při větším poměrném zastoupení stínů ve výsledném obrazu se musí provádět více testů zastínění. Opačný případ samozřejmě nastane při oddálení. Náš algoritmus je tedy opravdu pohledově závislý.

Kapitola 8

Závěr

V rámci této práce jsme prezentovali a implementovali algoritmus pohledově závislých stínových map, sloužící k obohacení libovolně reprezentované scény o ostré stíny. Ač tento algoritmus jako jednu ze svých optimalizací využívá dnes velmi populární metodu stínových map, netrpí jejími nedostatky, které plynou z diskretizačních chyb vznikajících v průběhu vytváření obrazu. A nejen že jimi netrpí, on je z principu eliminuje a díky tomu je schopen vykreslovat přesné stíny pro jakékoliv rozlišení.

Algoritmus pohledově závislých stínových map se tak, jak byl implementován, zatím nemůže rovnat rychlosti původní metody stínových map. To je však očekávaný výsledek, neboť metoda stínových map pracuje v obrazovém prostoru a není tedy narozdíl od námi prezentovaného algoritmu závislá na složitosti geometrie scény. Pro středně složité scény, tedy scény obsahující řádově tisíce až desetitisíce polygonů, je však i naše metoda schopna pracovat v interaktivním čase, v některých případech dokonce na hranici reálného času. Narozdíl od metody stínových map však produkuje kvalitní ostré stíny bez nepřesností na jejich hranách. Hranice reálného času však dosahujeme při měření na grafické kartě, která je v dnešní době považována již za starší. Dnešní grafické akcelerátory GeForce deváté generace jsou až několikanásobně výkonnější, a tak lze předpokládat, že by na nich bylo možné dosáhnout bez větších problémů plynulosti času reálného.

Mnohem důležitějším přínosem tohoto algoritmu a vůbec celé práce ale je, že princip, na kterém je algoritmus založen, otevírá brány novým a dosud nevídaným možnostem využití dnešních grafických karet. Prostřednictvím tohoto algoritmu jsme totiž zároveň prezentovali jednu z nejmladších disciplín dnešní počítačové grafiky, kterou je metoda vzorkování na GPU. Za využití možností novodobých grafických akcelerátorů se nám prakticky podařilo překonat omezení principu, který byl a dodnes je jádrem veškerého zobrazování v počítačové grafice – principu rasterizace. Ta totiž dovoluje počítat osvětlení, texturování a v neposlední řadě také zastínění pouze na pevně dané množině bodů, a navíc pouze ve středech pixelů.

Prostřednictvím algoritmu pohledově závislých stínových map jsme však ukázali, že díky schopnostem dnešních grafických karet, hlavně tedy poslednímu vývoji programovatelného zobrazovacího řetězce, lze provádět libovolné výpočty pouze na námi zvolených bodech, a tím tedy, oproti zažitému stereotypu, prakticky donutit GPU vzorkovat v libovolných bodech. Díky tomu je tedy nejen možné generovat přesné stíny pro jakékoliv rozlišení, nýbrž lze řešit jakékoliv obecné problémy viditelnosti.

Ač má tedy naše implementace konkrétního využití vzorkování na GPU – tedy počítání zastínění – jisté nedostatky, které jí zatím nedovolují pracovat v časově náročných prostředích, samotný tento princip by mohl stát na cestě vedoucí k překonání omezení, které plynou z rasterizace, a přispět tak k dalšímu rozvoji současných i budoucích grafických karet.

Literatura

[AL04] Timo Aila and Samuli Laine. Alias-free shadow maps. In *Rendering Techniques 2004: 15th Eurographics Workshop on Rendering*, pages 161–166, June 2004. Available from: `www.tml.tkk.fi/~timo/publications/aila2004egsr_paper.pdf`.

[BAS02] Stefan Brabec, Thomas Annen, and Hans-Peter Seidel. Shadow mapping for hemispherical and omnidirectional light sources. In John Vince and Rae Earnshaw, editors, *Advances in Modelling, Animation and Rendering (Proceedings Computer Graphics International 2002)*, pages 397–408, Bradford, UK, 2002. Springer.

[Bli77] James F. Blinn. Models of light reflection for computer synthesized pictures. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 192–198, New York, NY, USA, 1977. ACM.

[Cro77] Franklin C. Crow. Shadow algorithms for computer graphics. *Computer Graphics (SIGGRAPH '77 Proceedings)*, 11(2), Summer 1977.

[Dim07] Rouslan Dimitrov. Cascaded shadow maps, August 2007. PDF. Available from: `http://developer.download.nvidia.com/SDK/10.5/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf`.

[ED07] Elmar Eisemann and Xavier Décoret. Visibility sampling on gpu and applications. *Computer Graphics Forum (Proceedings of Eurographics 2007)*, 26(3), 2007. Available from: `http://artis.imag.fr/Publications/2007/ED07a`.

[EK02] Cass W. Everitt and Mark J. Kilgard. Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering. *GDC2002*, March 2002. PDF. Available from: `http://developer.nvidia.com/attach/6831`.

[Ger04] Philipp S. Gerasimov. Omnidirectional shadow mapping. In *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, chapter 12. Pearson Higher Education, 2004. Available from: `http://http.developer.nvidia.com/GPUGems/gpugems_ch12.html`.

[Hei91] Tim Heidmann. Real shadows, real time. In *Iris Universe*, volume 18, pages 28–31. Silicon Graphics Inc., 1991.

[HLHS03] Jean-Marc Hasenfratz, Marc Lapierre, Nicolas Holzschuch, and François Sillion. A survey of real-time soft shadows algorithms. *Computer Graphics Forum*, 22(4):753–774, dec 2003. Available from: `http://artis.inrialpes.fr/Publications/2003/HLHS03a`.

[HMN05] Denis Haumont, Otso Makinen, and Shaun Nirenstein. A low dimensional framework for exact polygon-to-polygon occlusion queries. In *Rendering Techniques*, pages 211–222, 2005. PDF. Available from: `http://www.eg.org/EG/DL/WS/EGWR/EGSR05/211-222.pdf`.

[JMB04] G. S. Johnson, W. R. Mark, and C. A Burns. The irregular z-buffer and its application to shadow mapping. Technical Report TR-04-09, The University of Texas at Austin, Austin, TX. Department of Computer Sciences, april 2004.

[Kil01] Mark J. Kilgard. Robust Stenciled Shadow Volumes. *CEDEC2001 presentation*, September 2001. PDF. Available from: `http://developer.nvidia.com/object/cedec_stencil.html`.

[LGQ+08] D. Brandon Lloyd, Naga K. Govindaraju, Cory Quammen, Steven E. Molnar, and Dinesh Manocha. Logarithmic perspective shadow maps. *ACM Transactions on Graphics*, 27(4):106:1–106:32, October 2008.

[MKK98] Pascal Mamassian, David C. Knill, and Daniel Kersten. The perception of cast shadows. *Trends in Cognitive Sciences*, 2:288–295, 1998.

[MT97] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *journal of graphics, gpu, and game tools*, 2(1):21–28, 1997.

[NBG02] S. Nirenstein, E. Blake, and J. Gain. Exact from-region visibility culling. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*, pages 191–202, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[RSC87] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, pages 283–291, July 1987.

[SD02] Marc Stamminger and George Drettakis. Perspective shadow maps. *ACM Transactions on Graphics*, 21(3):557–562, July 2002.

[SEA08] Erik Sintorn, Elmar Eisemann, and Ulf Assarsson. Sample-based visibility for soft shadows using alias-free shadow maps. *Computer Graphics Forum (Proceedings of the Eurographics Symposium on Rendering 2008)*, 27(4):1285–1292, June 2008. Available from: `http://artis.imag.fr/Publications/2008/SEA08`.

[SH74] Ivan E. Sutherland and G. W. Hodgman. Reentrant polygon clipping. *Commun. ACM*, 17(1):32–42, 1974.

[SKU08] László Szirmay-Kalos and Tamás Umenhoffer. Displacement Mapping on the GPU - State of the Art. *Computer Graphics Forum*, 27(6):1567–1592, September 2008.

[SKW+92] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadows and lighting effects using texture mapping. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, pages 249–252, July 1992.

[SWK07] Martin Stich, Carsten Wächter, and Alexander Keller. Efficient and robust shadow volumes using hierarchical occlusion culling and geometry shaders. In *GPU Gems 3*. Addison-Wesley Professional, 2007. Available from: `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch11.html`.

[Wan92] Leonard Wanger. The effect of shadow quality on the perception of spatial relationships in computer generated imagery. In *1992 Symposium on Interactive 3D Graphics*, pages 39–42, March 1992.

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 78)*, pages 270–274, August 1978.

[Woo92] Andrew Woo. The shadow depth map revisited. In *Graphics Gems III*, pages 338–342, 582. Charles River Media, 1992.

[WPF90] Andrew Woo, Pierre Poulin, and Alain Fournier. A survey of shadow algorithms. *IEEE Computer Graphics & Applications*, 10(6):13–32, November 1990.

[WSP04] Michael Wimmer, Daniel Scherzer, and Werner Purgathofer. Light space perspective shadow maps. In *Rendering Techniques 2004: 15th Eurographics Workshop on Rendering*, pages 143–152, June 2004.

[ŽBSF04] Jiří Žára, Bedřich Beneš, Jiří Sochor, and Petr Felkel. *Moderní počítačová grafika*. Computer Press s.r.o, Brno, 2nd edition, 2004. In Czech.

Seznam obrázků

Seznam tabulek

Seznam příkladů

Dodatek A

Instalační a uživatelská příručka

A.1 Požadavky

Požadavky na spuštění programu jsou následující:

- Grafická karta GeForce 8xxx a výše, podporující OpenGL 3.0+ a GLSL 1.4+
- Operační systém GNU/Linux
- Instalované knihovny GLUT (freeglut) a SDL
- Pro kompilaci jsou potřeba též zdrojové soubory knihoven GLUT a SDL

A.2 Instalace

Program je dodáván ve spustitelné verzi pro GNU/Linux, která se nachází ve složce `bin`, a jeho spuštění je popsáno v sekci A.4. Pokud je třeba kompilovat, příslušný skript se nachází ve složce `build` a postup pro kompilaci je následující:

```
$ cd build
$ make
```

Po úspěšné kompilaci se vytvoří spustitelný soubor ve složce `bin`.

A.3 Generátor shaderů

Po úspěšné kompilaci se ve složce `bin` bude nacházet i složka `shaders`. Ta obsahuje skript `build.sh`, pomocí kterého lze vytvářet *shadery* pro algoritmus pohledově závislých stínových map. Generování *shaderů* lze ovlivnit několika přepínači, které dovolují přidávat libovolné kombinace implementovaných optimalizací, nebo lze *shadery* vytvářet ve *vizualizačním režimu* (např. pro vizualizaci stínových polygonů). Fungování těchto *shaderů* je potom pevně provázáno se samotným programem, viz kapitola A.5.

Skript `build.sh` přijímá následující přepínače a argumenty:

```
Argumenty:
-c      K definování optimalizací bude použit soubor options.conf.
        Přepínače v sekci Optimalizace budou ignorovány.
-n      Neinteraktivní režim.
        Soubory budou přepisovány bez potvrzení uživatele.
-o FILE Specifikuje název a cestu k výstupnímu souboru.
        Přípony .vs, .gs a .fs budou k souboru připojeny automaticky.

Optimalizace:
-b      Vloží do shaderů kód k provádění benchmarků. Bez tohoto přepínače
        nebude měření rychlosti v aplikaci fungovat správně.
-d      Ořezávání hloubky polygonu maximální hloubkou ve scéně z pohledu
        světla. K jejímu získání se používá hloubková LOD mapa.
        Ke správnému fungování je nutné zapnout generování hloubkové textury
        v aplikaci.
-e      Ořezávání hloubky polygonu maximální hloubkou ve scéně z pohledu
        světla. K jejímu získání se používá přesná hodnota z hloubkové mapy
        pro každý vrchol.
        Ke správnému fungování je nutné zapnout generování hloubkové textury
        v aplikaci.
-f      Eliminace trojúhelníků přivrácených ke světlu.
-l      Eliminace trojúhelníků zastíněných z pohledu světla.
        Ke správnému fungování je nutné zapnout generování hloubkové textury
        v aplikaci.
-p      Zahození vzorků nacházejících se před rovinou stínového polygonu.
-s      Vytváření vždy jen jednoho stínového polygonu.
        Pozor! Tato optimalizace z principu nefunguje a je přidána pouze
        pro demonstraci.
-x      Ořezávání stínového polygonu rozsahem souřadnic objektů viditelných
        z pohledu kamery. Pozor! Chod aplikace se může výrazně zpomalit.
        Ke správnému fungování je nutné zapnout generování textury rozsahů
        v aplikaci.

Vizualizace:
```

```
-u      Vizualizace trojúhelníků eliminovaných pomocí optimalizace -l. Ke
        správnému zobrazení vizualizace je nutné vypnout blending v
        aplikaci.
-w      Vizualizace vytvářených stínových polygonů.
```

Samotná aplikace potom dokáže přepínat mezi devíti vygenerovanými *shadery*. Aby toto bylo možné, musí se tyto *shadery* nacházet ve složce `optimizations` a musí se jmenovat `shader_i.vs|gs|fs`, kde `i` je číslo v rozsahu 1 až 9. Mezi těmito *shadery* lze pak přepínat čísly 1 až 9 na klávesnici.

Příklad: Pokud chceme vygenerovat na pozici osm *shader*, který bude podporovat měření rychlosti, eliminovat trojúhelníky přivrácené ke světlu, ořezávat stínové polygony hloubkou získanou přesně pro každý vrchol z hloubkové mapy, a zároveň takto vytvořené stínové polygony vizualizovat, zavoláme skript `build.sh` s následujícími parametry:

```
$ ./build.sh -n -o optimizations/shader_8 -bfew
```

V aplikaci potom musíme povolit generování hloubkové mapy, viz sekce A.5.

A.4 Spuštění

Spustitelný program s názvem `Pzsm` se nachází ve složce `bin`. Kvůli relativním cestám je nutné ho spouštět právě z této složky, jinak nemusí najít cestu k *shaderům* a nebude fungovat správně.

Program přijímá jeden argument, a sice rozlišení (velikost) okna ve formátu `ŠÍŘKAxVÝŠKA`. Například:

```
$ ./Pzsm 800x600
```

A.5 Ovládání

Aplikace podporuje generování stínů pomocí standardní metody stínových map a potom samozřejmě pomocí algoritmu pohledově závislých stínových map (PZSM). Klávesové zkratky se potom dělí do dvou skupin – takzvané globální, která fungují pro obě metody, a potom ty, které fungují pouze v metodě pohledově závislých stínových map.

Aplikaci lze ovládat následovně:

```
Globální
----------------------------------------------
Levé tlačítko myši  Otáčení scény
Pravé tlačítko myši Přiblížení/oddálení scény
Esc             Ukončení aplikace
Mezerník        Spuštění/pozastavení animace světla
A               Přepíná mezi metodou stínových map a PZSM.
C, Shift + C    Zvyšuje/snižuje geometrickou složitost scény.
F               Ukáže maximální a minimální dosažené FPS.
                Čítač se vynuluje kdykoliv je načten nový shader (0-9) pokud
                je použit algoritmus PZSM.
I               Přepíná zobrazení informací v okně.
L               Uzamkne kameru na pozici světla.
O               Otočí kameru o 90 stupňů.
R               Umístí kameru na výchozí pozici.
S, Shift + S    Zvyšuje/snižuje zastínění scény.

PZSM
----------------------------------------------
0-9     Použije shader s názvem 'shader_[0-9].vs|gs|fs' ze složky
        'optimizations'. Pozor! Shader 0 může způsobit výrazné
        zpomalení vykreslování a celého počítače, jelikož nebude
        použita žádná optimizace.
B       Spustí měření rychlosti algoritmu. Rychlost je měřena vždy
        na třiceti snímcích pro každou fázi, které jsou dohromady tři.
        V závislosti na FPS tedy může měření zabrat více času (aplikace
        přestane v průběhu měření reagovat).
D       Přepíná generování hloubkové mapy (nutnost pro některé
        optimalizace).
P       Přepíná náhledy textur, a to dle následujícího schématu:
        0) Vypnuto
        1) Pokud je zapnuto generování hloubkové mapy, ukazuje vlevo její
           kanál R a vpravo kanál G. Pokud je zapnuto i generování textury
           rozsahů, je hloubková textura zobrazena nahoře a textura rozsahů
           dole.
        2) To samé jako 1, ale ukazují se kanály B a A.
        3) Ukazuje vlevo texturu vzorků a vpravo stínovou mapu.
T       Přepíná míchání barev (blending).
Tab     Pokud je zapnut náhled textur, zobrazí mipmapu na nižší úroveni.
        Shift + Tab zobrazí mipmapu na vyšší úrovni.
U       Přepíná odstraňování odvrácených ploch (back face culling).
V       Přepíná generování textury rozsahů (nutnost pro některé
        optimalizace).
```

Dodatek B

Obsah přiloženého CD

program	Složka s programem
bin	Složka se spustitelným programem
shaders	Složka se shadery
optimizations	Složka s optimalizujícími shadery
build.sh	Generátor shaderů
Pzsm	Spustitelný program
build	Složka s instalačním skriptem (makefile)
doc	Složka s HTML dokumentací (Doxygen)
src	Složka se zdrojovými soubory a shadery
thesis	Textová část diplomové práce
pdf	Diplomová práce ve formátu PDF
source	Zdrojové soubory v LaTeXu