České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačové grafiky a interakce

Diplomová práce

Rozšiřitelný zobrazovací řetězec založený na odloženém stínování

Bc. Daniel Šimek

Vedoucí práce: Ing. Jiří Bittner, Ph.D.

Studijní program: Otevřená informatika, strukturovaný, Navazující magisterský

Obor: Počítačová grafika a interakce

31. prosince 2012

Poděkování

Děkuji tímto vedoucímu diplomové práce Ing. Jiřímu Bittnerovi, Ph.D. za účinnou metodickou, pedagogickou a odbornou pomoc spolu s dalšími cennými radami při zpracování této práce. Dále děkuji Ing. Antonínu Míškovi, Ph.D. za podporu ohledně implementačního prostředí VRUT a pomoc při závěrečném testování. Na závěr bych chtěl poděkovat rodičům za důležitou podporu a zázemí během celého studia.

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 2. 1. 2013 .....

Abstract

Implementation of modern rendering pipeline based on deferred shading is discussed and realized in this work. We focus on extensibility by new advanced algorithms. Firstly methods used in real time are analyzed, it is followed by implementation environment and proposal of architecture of rendering system. The usability was verified in some known methods of computer graphic like shadow maps, depth of field or ambient occlusion. The implementation was realized with respect to simple modifications of rendering pipeline for different rendering in various situations. This system was primarily optimised for real time rendering in modern hardware configurations. Visual quality and time analysis of rendering is disscused at the end.

Abstrakt

Tato práce se zabývá návrhem a implementací moderního zobrazovacího řetězce se zaměřením na snadnou rozšiřitelnost z hlediska aplikace nových pokročilých algoritmů. Diskuzi metod používaných pro zobrazování v reálném čase následuje popis implementačního prostředí s návrhem architektury zobrazovacího systému. Použitelnost návrhu byla ověřena na řadě známých metod počítačové grafiky, například na výpočtu stínů, hloubky ostrosti nebo ambientního zastínění. Návrh a implementace byly realizovány s ohledem na snadnou aplikaci efektů zlepšujících vizuální kvalitu obrazu podle potřeb zobrazované scény. Důraz byl kladen především na schopnost vykreslování v reálném čase na moderních počítačových sestavách. Na závěr je diskutována kvalita zobrazení spolu s časovou náročností implementovaných metod.

Obsah

1 Úvod **1**
1.1 Struktura práce . . . 2

2 Existující metody vykreslování v reálném čase **3**
2.1 Zobrazovací část aplikace . . . 3
2.1.1 Modulární architektura . . . 4
2.1.2 Kompozice zobrazovacího řetězce . . . 4
2.2 Odložené stínování . . . 6
2.3 Stíny . . . 6
2.3.1 Stínové objemy . . . 6
2.3.2 Stínové mapy . . . 8
2.3.2.1 Zaměření pohledového objemu pro stínovou mapu . . . 8
2.3.2.2 Perspektivní stínové mapy (PSM) . . . 8
2.3.2.3 Perspektivní stínové mapy v prostoru světla (LispSM) . . . 10
2.3.2.4 Vylepšení stínových map . . . 10
2.4 Zastínění ve scéně . . . 12
2.4.1 Zastínění okolím (AO) . . . 12
2.4.2 Zastínění okolím v prostoru obrazu (SSAO) . . . 13
2.4.3 Směrové zastínění v prostoru obrazu (SSDO) . . . 13
2.5 Antialiasing . . . 14
2.5.1 Supersample antialiasing (SSAA) . . . 14
2.5.2 Multisample antialiasing (MSAA) . . . 15
2.5.3 Coverage sample antialiasing (CSAA) . . . 15
2.5.4 Antialiasing pro odložené stínování (SRAA) . . . 15
2.6 Hloubka ostrosti . . . 16
2.7 Zrcadlové plochy . . . 16

3 Zobrazovací řetězec založený na odloženém stínování **19**
3.1 Implementační prostředí . . . 19
3.2 Použité technologie . . . 20
3.3 Požadavky na implementaci . . . 21
3.4 Zobrazovací modul . . . 21
3.4.1 Návrh architektury . . . 22
3.4.1.1 Správa průchodů . . . 23
3.4.1.2 Zpracování událostí . . . 23

3.4.1.3 Sdílení dat . . . 24
3.4.1.4 Uživatelské rozhraní . . . 25
3.4.2 Podpora vykreslování . . . 25
3.4.2.1 Zobrazení geometrie . . . 26
3.4.2.2 Druhy materiálů . . . 26
3.4.2.3 Druhy světel . . . 26
3.4.2.4 Pozadí . . . 27
3.4.3 Podpora vývoje nových průchodů . . . 27
3.5 Vytvoření zobrazovacího průchodu . . . 27
3.6 Základní zobrazovací průchody . . . 28
3.6.1 Ořez scény . . . 28
3.6.2 Vytvoření g-bufferu . . . 28
3.6.3 Výpočet osvětlení . . . 29
3.6.4 Průhledné objekty . . . 31
3.7 Doplňkové zobrazovací průchody . . . 32
3.7.1 Vytvoření stínů . . . 32
3.7.1.1 Generování stínových map . . . 32
3.7.1.2 Aplikace stínových map . . . 33
3.7.2 Ambientní zastínění v prostoru obrazu . . . 35
3.7.2.1 Generování vzorků . . . 35
3.7.2.2 Filtrování mapy zastínění . . . 37
3.7.2.3 Aplikace mapy zastínění . . . 37
3.7.2.4 Nedostatky metody SSAO . . . 38
3.7.3 Antialiasing . . . 39
3.7.3.1 Generování normál a hloubky . . . 39
3.7.3.2 Filtrování . . . 40
3.7.3.3 Redukce rozmazání . . . 41
3.7.4 Simulace hloubky ostrosti . . . 42
3.7.4.1 Rozostření obrazu . . . 42
3.7.4.2 Finální efekt hloubky ostrosti . . . 45
3.7.5 Zrcadlové plochy . . . 45
3.7.5.1 Vytvoření odrazu . . . 45
3.7.5.2 Materiál a dokončení . . . 46
3.7.6 Obsah g-bufferu . . . 47
3.7.7 Zobrazení BVH . . . 47
3.8 Doplňkové funkce . . . 48
3.9 Souhrn . . . 49

4 Výsledky **51**
4.1 Testovací scény . . . 51
4.2 Testovací hardware . . . 52
4.3 Výkonnostní testy . . . 52
4.3.1 Srovnání nového modulu s původním . . . 52
4.3.2 Náročnost jednotlivých průchodů . . . 54
4.3.3 Časová analýza metody SRAA . . . 58
4.3.4 Závislost metody SSAO na počtu vzorků . . . 59

4.4 Vizuální kvalita zobrazení 59
4.4.1 Závislost stínových map na rozlišení 60
4.4.2 Kvalita SSAO v závislosti na počtu vzorků 61
4.4.3 Porovnání nového a původního modulu 61

5 Závěr **65**

A Seznam použitých zkratek **75**

B Uživatelská příručka **77**

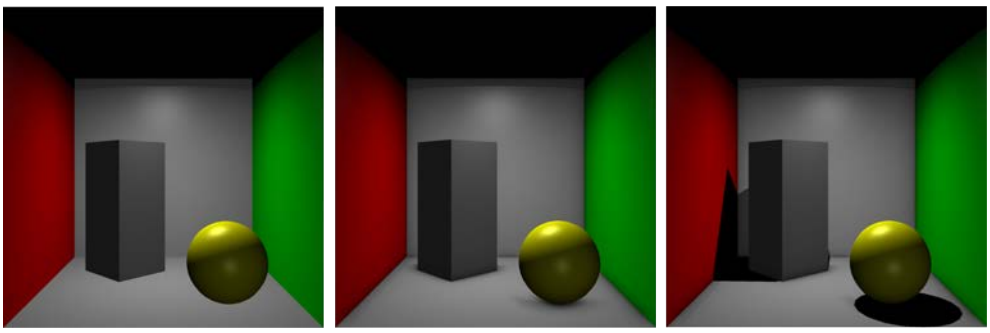
C Obsah přiloženého DVD **79**

Kapitola 1

Úvod

Fotorealistické zobrazování 3D dat je velká výzva pro počítačovou grafiku. Na nejvyšší úrovni v tomto odvětví jsou špičkové herní enginy vyvíjené týmy specialistů pro účely komerčního nasazení. Pro účely výuky a výzkumné činnosti je však potřeba zobrazovací systém umožňující snadnou modifikaci, optimalizaci nebo přidání jednotlivých částí. Pak lze dosáhnout nejen dobrého pochopení algoritmů, které studenti programují, ale vyvinout kvalitní zobrazovací modul podle nejnovějších vědeckých publikací. Moderní zobrazovací systémy dokáží zobrazovat rozsáhlé dynamické scény v reálném čase s velmi dobrou vizuální kvalitou.

Tato práce se zabývá vývojem modulárního zobrazovacího řetězce určeného především pro programátory zabývající se implementací pokročilých grafických algoritmů. Práce demonstruje alternativu architektury zobrazovacího modulu spolu s aplikací řady algoritmů používaných pro pokročilé zobrazování scén, jako je výpočet stínů od přímého osvětlení, zastínění okolím nebo simulace hloubky ostrosti posouvající generovaný obraz o něco blíže realitě. V obecném zobrazovacím systému se scény mohou a obvykle i značně liší, proto je navržený modul schopný aplikovat rozdílné efekty podle potřeby dané scény, viz obrázek 1.1 znázorňující rozdíl syntetizovaného obrazu bez úprav a se zapnutými efekty počítajícími nepřímé zastínění a stíny.



Obrázek 1.1: Porovnání obrazu v závislosti na aplikovaných efektech. Zleva: scéna se základním osvětlením bez efektů, ambientní zastínění a ambientní zastínění se stíny od přímého osvětlení.

1.1 Struktura práce

První část projektu se zabývá existujícími metodami vykreslování v reálném čase. Jedná se o přehled modulárních přístupů k architektuře zobrazovací části aplikace spolu s metodami zvyšujícími kvalitu obrazu.

Další část již obsahuje samotný návrh a implementaci zobrazovacího řetězce založeného na odloženém stínování. Nejprve je rozebráno implementační prostředí, použité technologie a požadavky na implementaci. Následuje diskuze architektury a návrh jádra zobrazovacího systému včetně podpory vykreslování s různými druhy materiálů a světel. Navazuje implementace modulárních částí zobrazovacího řetězce, které jsou rozděleny do dvou skupin. Základní zobrazovací komponenty jsou nezbytné k běžnému vykreslování, zatímco doplňkové přidávají do scény nejrůznější efekty. Nejdůležitější takové části jsou zejména vytvoření stínů od přímého osvětlení nebo ambientní zastínění. Je jich ale implementováno mnohem více, čímž je zároveň demonstrována modularita celého systému.

Závěrečná kapitola obsahuje výsledky a měření. Kromě diskuze rychlosti jednotlivých komponent, je zde i rozbor závislosti důležitých částí zobrazování na nastavených parametrech nebo porovnání kvality zobrazení vzhledem k aktivním doplňkovým efektům.

Kapitola 2

Existující metody vykreslování v reálném čase

Jeden z velmi důležitých faktorů ovlivňujících vnímání scény je kvalita její vizuální reprezentace. Na výsledném dojmu se podílí řada faktorů. Jedná-li se o aplikaci zobrazující nereálnou scénu, je situace značně ulehčena. Naopak, je-li potřeba zobrazovat reálné scény, je situace o poznání složitější. Takový obraz musí působit velmi přirozeně a musí být podobný skutečnému světu respektujícímu fyzikální zákony. Simulovat fyzikální vlastnosti reálného světa a zejména osvětlení je stále poměrně problematické, i když existují velmi pokročilé techniky. V aplikaci pracující v reálném čase je však nutné použít metody hrubě aproximující přesný výpočet i na moderním hardwaru. Tato kapitola se zabývá existujícími technikami a algoritmy používanými při syntéze obrazu. První část je zaměřena na strukturu a možnosti jádra zobrazovací částí aplikace. Následuje rozbor metod využívaných k zobrazování realisticky vypadajících scén v reálném čase, jako je výpočet stínů, nepřímého zastínění, hloubky ostrosti nebo vylepšení kvality obrazu pomocí antialiasingu.

2.1 Zobrazovací část aplikace

Část aplikace starající se o zobrazování hraje velkou či malou roli v závislosti na softwaru a jeho využití. Zatímco v 3D modelářích typu Maya, v simulátorech a v neposlední řadě ve hrách je zobrazování naprosto nepostradatelná vlastnost, v jiných aplikacích se jedná pouze o zanedbatelnou část určenou například jenom k náhledu. Přestože v dřívější době grafika nejlepších aplikacích připomínala cosi hodně vzdáleného od reality, dnes je tomu jinak a zobrazovací modul je v řadě případů nejpodstatnější a nejintenzivněji vyvíjená část aplikace. Nejen ve hrách, ale i ve vědecké vizualizaci a řadě dalších odvětví je nezbytná špičková vizualizace dat. Zvládat navíc zobrazení ve fotorealistické kvalitě v reálném čase je stále i na moderních počítačích poměrně velká výzva.

Přístup k architektuře zobrazovacího řetězce se s postupem času velmi změnil. Na začátku byla zobrazovací část programu pevně implementována a uživatel příliš nemohl ovlivnit kvalitu výstupu. Tato část navíc byla komplexně naprogramována a jakákoliv změna byla velmi náročná. Pevná architektura byla samozřejmě z dlouhodobého hlediska nevyhovující a uživatelé potřebovali mít možnost zasáhnout do zobrazovacího řetězce. Silně parametrizovaný,

pevně naprogramovaný řetězec sice jistou možnost ovlivnit grafický výstup poskytuje, ale kód takové aplikace je poměrně rozsáhlý a těžko udržitelný v rámci dlouhodobého vývoje. Nejmodernější přístup využívá modulární architektury skládající se z menších komplexních částí umožňujících nenáročně měnit a vylepšovat vlastnosti celku.

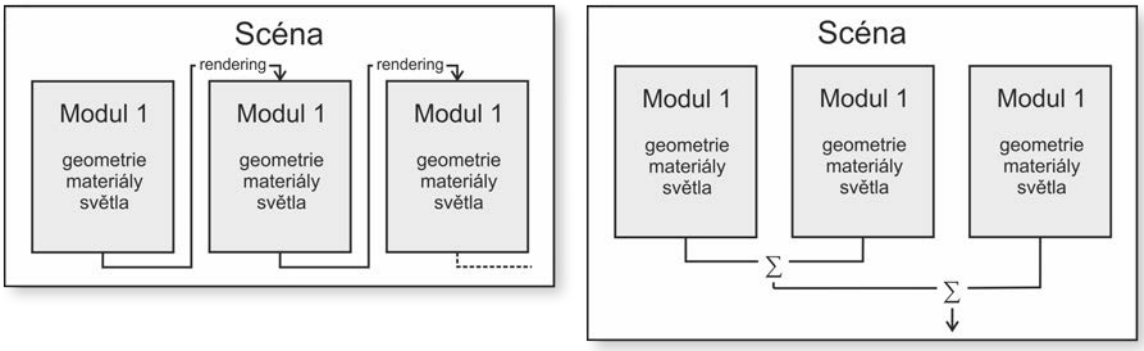
2.1.1 Modulární architektura

Mluvíme-li o modulární architektuře, máme na mysli zobrazovací řetězec, který je rozložen do nezávislých částí, o jejichž složení se stará uživatel. Takovéto řešení má nesporné výhody v přehlednosti a rozšiřitelnosti zobrazovacího modulu. Z jak velkých komplexních částí se skládá, určuje jeho míru volnosti. Pokud se skládá z příliš malých částí, programátoři ztrácí nezanedbatelný čas kompozicí jednotlivých prvků namísto řešení aktuálního problému. Naopak příliš velké celky nedovolí takovou tvůrčí volnost. Správně navržená aplikace umožní fotorealistické renderování stejně jako použití jiných technik a záleží pouze na uživateli, případně programátorovi, jakou metodu zvolí. Vysoce modulární nástroj pro zobrazování nejen vědeckých dat je například Visualization Toolkit (VTK) implementovaný v C++ [Wil].

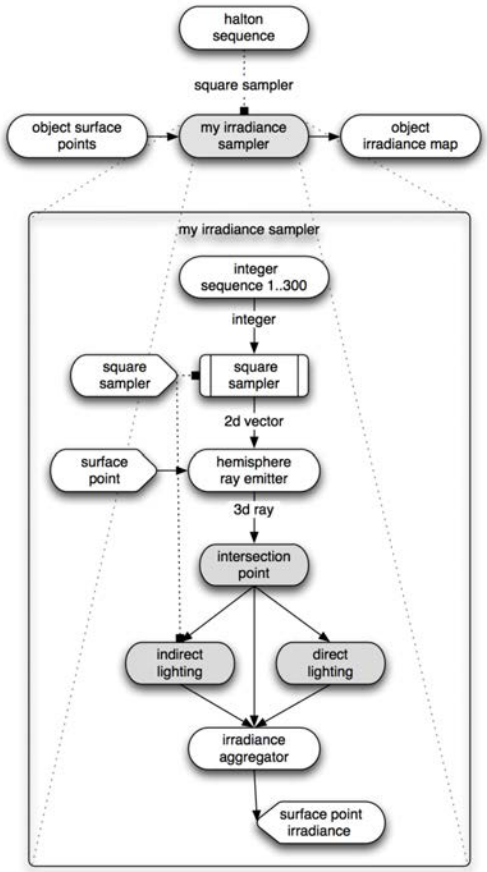
2.1.2 Kompozice zobrazovacího řetězce

Jakým způsobem se modulární části podílejí na tvorbě obrazu a jak jsou řazeny, závisí na konkretní implementaci. Jedna z možností je řazení modulů sériově za sebe, kdy po ukončení jedné části je aktivována část další. Sériové řazení má výhodu nízké náročnosti na režii modulů, ale nenabízí příliš velkou volnost. Naopak paralelní běh komponent s následným složením výsledků se zdá být vhodné řešení v případě potřeby správného začlenění velmi odlišných modulů. Paralelně mohou být zpracovány pouze absolutně nezávislé části. Tím jsou myšleny části nevyužívající žádný závislý vstup, který by mohl být změněn paralelně zpracovávanými moduly. Například post-processing efekty, pracující nad již hotovým obrazem, nemohou být zpracovány paralelně s vykreslováním scény, ale musejí být zařazeny až na konec zobrazovacího řetězce. Paralelní zpracování nezávislých částí může na moderních zařízeních obsahujících více jader CPU přinést značné urychlení vykreslování. Rozdíl mezi sériovým a paralelním zapojením komponent je znázorněn na obrázku 2.1.

Poměrně jednoduché paralelní nebo sériové řazení modulů je ve většině případů naprosto dostačující řešení, ale existují i sofistikovanější metody založené na složitějších vazbách mezi moduly. V článku zabývajícím se velmi flexibilní modulární architekturou [ACOD07] navrhli tři typy vazeb mezi moduly. Datové spojení (data-flow connection links) je hlavním používaným prvkem a reprezentuje běžné předání dat z výstupu jednoho modulu na vstup druhého. Naproti tomu spojení na základě dotazů (query connection links) je využíváno v případě pokročilých zobrazovacích metod, kde by datové spojení bylo neefektivní. Prakticky se jedná o asynchronní dotazy pomocí podporovaného protokolu mezi moduly. Poslední prezentovaný typ je využívání vzorů (template links). Vzory jsou využívány pro prototypování, jestliže je vhodné rychle nahrazovat stávající modul jiným. Namísto kopírování celého schématu a nahrazení modulu, je možné vytvořit vzor a ten pak propojit s moduly měnícími funkcionalitu daného modulu schématu. Situace je názorně zobrazena na obrázku 2.2. Vzor square sampler reprezentuje vzor zodpovědný za propojení modulu square sampler se schématem.



Obrázek 2.1: Sériové zapojení modulů - na vstupu modulů je vždy výstup z předchozích (vlevo). Paralelní zapojení zobrazovacího řetězce - výstupy z jednotlivých částí je nutné správně sloučit (vpravo).



Obrázek 2.2: Příklad definice vzoru (my irradiance sampler) a jeho aplikace ve schématu. Převzato z [ACOD07].

2.2 Odložené stínování

Při klasickém zobrazování je geometrie nejprve převedena do souřadnicového systému kamery, následně rasterizována a pro každý pixel počítáno osvětlení. Při vykreslování klasickou technikou dochází k častému přepisování barvy pixelu podle hloubkového testu a většinou velmi náročný výpočet osvětlení pak přijde vniveč.

Populární metoda urychlující zobrazování v rozsáhlých scénách rozdělením geometrické části od výpočtu osvětlení se nazývá odložené stínování (deferred shading). Zobrazování probíhá minimálně ve dvou průchodech. Nejprve je scéna rasterizována a jsou uloženy informace o geometrii a materiálech v prostoru obrazu, do paměti. V následujícím průchodu jsou tyto informace zpracovány a je vypočteno osvětlení pouze pro viditelné pixely. Tento přístup představil v roce 1988 Deering a kol. [DWS+88]. Koncept použití paměti pro uložení informací o geometrii (g-buffer) byl zaveden v [ST90].

V moderních aplikacích s velmi propracovanými a náročnými metodami výpočtu osvětlení výsledného pixelu je vhodné použít odložené stínování pro zvýšení výkonu a osvětlení počítat pouze ve viditelných pixelech. Naopak v malých scénách nebo při nenáročných výpočtech osvětlení je výhodnější klasické přímé zobrazování, protože pro odložené stínování je nezbytné alokovat na grafické kartě dodatečnou paměť pro vytvoření g-bufferu, který obsahuje veškeré informace o geometrii a materiálech potřebné pro výpočet osvětlení.

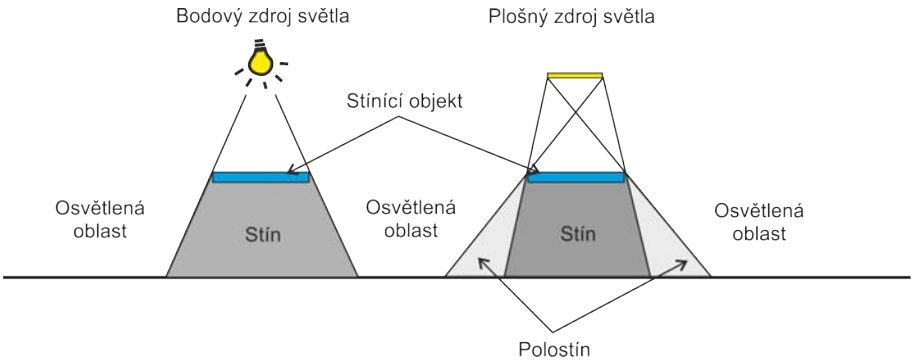
Hlavní nevýhodou odloženého stínování je problém s průhlednou geometrií. Z principu g-buffer nemůže obsahovat pro jeden pixel informace o průhledném objektu i o geometrii, která je skrz objekt viditelná. Tento nedostatek je obvykle řešen vykreslením průhledné geometrie samostatně po výpočtu osvětlení pomocí odloženého stínování. Dalším neméně podstatným problémem je aliasing. S g-bufferem není možné využít hardwarové podpory multisamplingu a antialiasing je nutné řešit pomocí sofistikovaných algoritmů v post-processingu.

2.3 Stíny

V realistické syntéze obrazu stíny hrají významnou roli a alespoň základní implementace je již dlouhou dobu nezbytnou součástí všech herních enginů a 3D modelovacích nástrojů. Stíny vypovídají o rozložení objektů ve scéně stejně jako o vlastnostech světelných zdrojů. V oblasti počítačové grafiky se stíny dělí na ostré a měkké. Ostré stíny vytváří bodové zdroje světla. Paprsek vycházející ze zdroje světla dopadá na stínící neprůhledný objekt a veškerá geometrie ležící za ním je tedy neosvětlena, čímž je vytvořen stín. Plošné zdroje světla naopak vytvářejí přirozenější měkké stíny, kdy se na hranici mezi zastíněnou a nezastíněnou částí scény ještě nachází oblast polostínu způsobená menším počtem dopadajících paprsků přímo ze světla, viz obrázek 2.3. Bodové zdroje světla se v reálném světě nevyskytují, ale jsou často využívány v počítačové grafice pro nižší výpočetní náročnost vytvoření stínů. Pro výpočet stínů v reálném čase jsou nejčastěji používané metody stínových objemů nebo stínových map popsané v následujících sekcích 2.3.1 a 2.3.2.

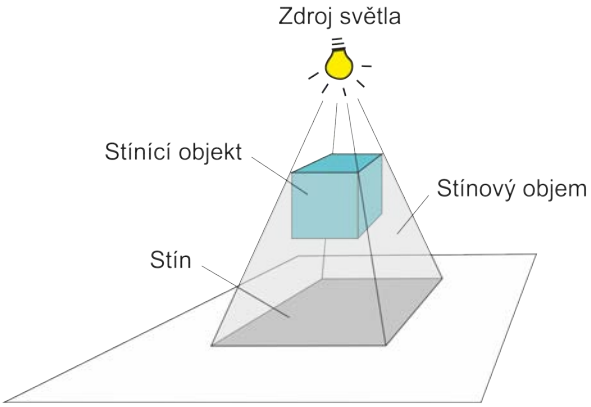
2.3.1 Stínové objemy

Stínové objemy zavedl Frank Crow v roce 1977 [Cro77]. Označují místa, která spadají do stínu a dělí tedy scénu na nezastíněnou a zastíněnou oblast, viz obrázek 2.4. V základní verzi



Obrázek 2.3: Ostré stíny od bodového světla (vlevo), měkké stíny od plošného zdroje světla (vpravo).

je nutné vytvořit stínový objem pro každý stínící objekt. Lepší varianta je však vytvářet stínový objem podle siluety všech stínících objektů, které na sebe navazují. Výhoda stínových objemů spočívá v jejich přesnosti na jeden pixel, i když některé implementace mohou mít problémy se samo-zastíněním na siluetách stínících objektů. Tento problém je nejčastěji řešen malým odsazením stínového objemu od stínícího objektu ve směru světelných paprsků. Vytváření stínových objemů může být poměrně výpočetně náročné, zejména v případě, že stínové objekty zabírají velkou plochu obrazu z pohledu kamery a dochází k častému zápisu dat při rozhodování, zda objekt leží či neleží ve stínu. Stínové objemy jsou navíc většinou počítané na CPU a špatně se výpočet akceleruje na grafické kartě.



Obrázek 2.4: Stínový objem zkonstruovaný podle siluety stínícího objektu.

Pro nalezení stínů je potřeba zjistit, zda objekt leží v nějakém stínovém objemu. Pokud leží alespoň v jednom, pak je zastíněn. Nejprve je vykreslena scéna a vytvořen čítač (obvykle je výhodné použít stencil buffer). Poté jsou vykresleny přivrácené plochy stínových objemů a navýšen čítač pro jednotlivé fragmenty vždy, když úspěšně proběhl hloubkový test. V tuto chvíli čítač obsahuje počet stínových objemů, do kterých vstoupil paprsek vyslaný z kamery.

Následně jsou vykresleny odvrácené plochy stínových objektů a pro úspěšné hloubkové testy čítač snížen. Stav čítače pak určuje, v kolika stínových objemech paprsek zůstal. V případě, že je čítač nulový, pixel není zastíněn žádným objektem, v opačném případě leží ve stínu.

2.3.2 Stínové mapy

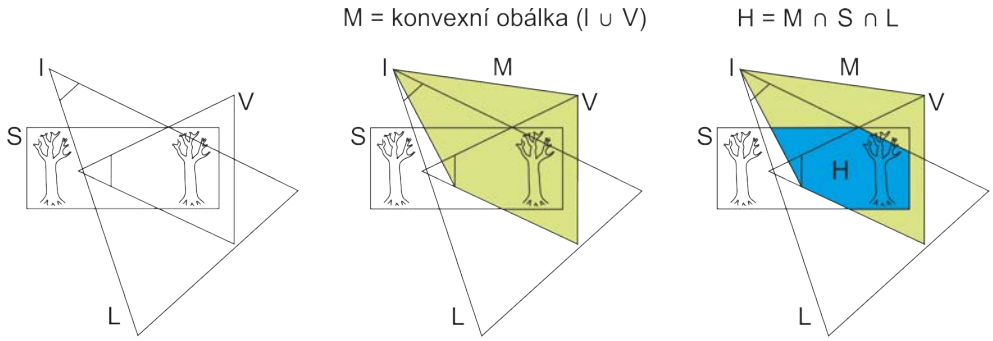
Metoda stínových map pracuje na odlišném principu a zdá se být výhodnější vzhledem k možnosti výpočtu na grafické kartě. Algoritmus pracuje ve dvou průchodech. V první části je vykreslena scéna z pozice světla. Během celého procesu je ukládána pouze informace o hloubce scény do zvláštního bufferu (z-buffer), a tím vznikne stínová mapa. Nachází-li se ve scéně více světel, je nutné vytvořit z-buffer pro každé z nich. V druhé fázi je již vykreslována scéna se stíny. Při zpracování fragmentu je porovnána hloubka z pohledu kamery s hloubkou ve stínové mapě. Jestliže je hloubka ve stínové mapě menší než hloubka aktuálně zpracovávaného fragmentu, fragment je zastíněn, protože se mezi fragmentem a světlem nachází nějaký stínící objekt. Tato metoda je zdánlivě velmi jednoduchá a je možné jí plně implementovat na grafické kartě. Nicméně stínové mapy mají také svá omezení. Velkou roli na kvalitu stínů má zejména rozlišení stínové mapy a efektivita jeho využití. Stínová mapa musí obsahovat informaci o hloubce veškeré geometrie, která vrhá stín. Takové geometrie může být poměrně mnoho a hlavně u exteriérových scén, které obsahují směrový zdroj světla, mnohdy rozlišení stínové mapy nedostačuje. Pokud je rozlišení příliš malé, na hranách stínů se objevuje aliasing. Využití stínových map se stejným rozlišením v celé své ploše se označuje jako uniformní stínové mapy (uniform shadow mapping).

2.3.2.1 Zaměření pohledového objemu pro stínovou mapu

Pro konstrukci stínové mapy je jeden z nejdůležitějších faktorů správné zaměření pohledového objemu na scénu z pozice světla. Při generování stínových map je nezbytné, aby ve stínové mapě byla obsažena všechna geometrie viditelná z pohledu kamery plus geometrie potencionálně vrhající stíny do výsledného obrazu [Sta02], ale zároveň maximálně efektivně využít plochu stínové mapy. Definujme množinu všech objektů ve scéně jako S (obvykle se jedná o AABB obálku scény). Pohledový objem označíme jako V a objekt zahrnující paprsky vržené světlem na pozici l jako L. Nejprve spočítáme konvexní obálku M obsahující l a V. M tedy obsahuje všechny potencionálně stínící objekty nacházející se mezi zdrojem světla a pohledovým objemem. Nakonec ořežeme objekt M obálkou scény S a objektem L ($H = M \cap S \ \cap L$). Nyní H vymezuje prostor, který je nutné obsáhnout do stínové mapy, viz obrázek 2.5.

2.3.2.2 Perspektivní stínové mapy (PSM)

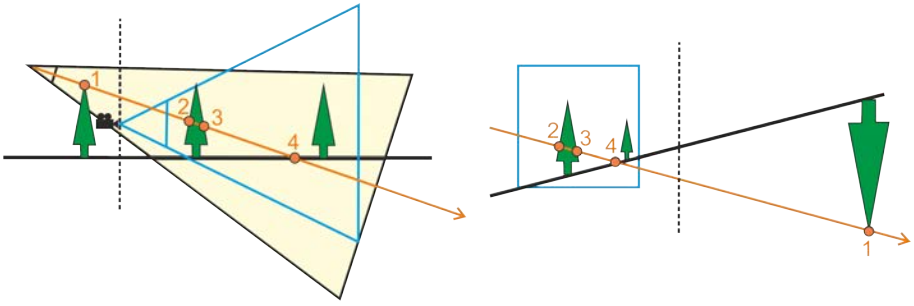
Perspektivní stínové mapy zavedl v roce 2002 Marc Stamminger [Sta02]. U základních uniformních stínových map je veškerým objektům přiděleno stejné rozlišení bez ohledu na vzdálenost od pozorovatele. Běžně se pak stává, že stíny vytvořené blízko u pozorovatele mají velmi špatnou kvalitu způsobenou velkým aliasingem a naopak stínům velmi vzdáleným je ve stínové mapě přiděleno zbytečně velké rozlišení. Vzdálené stíny se pak promítnou do výsledného obrazu pouze jako několik pixelů a prakticky bezvýznamně zabírají místo ve stínové



Obrázek 2.5: Obálka scény S, pohledový objem V, pozice světla l a pohledový objem světla L (vlevo). Rozšíření pohledového objemu V o pozici světla l (uprostřed). Skupina bodů H, která musí být viditelná při generování stínové mapy (vpravo).

mapě. Tento problém řeší perspektivní stínové mapy, u kterých je na základě vzdálenosti od pozorovatele přiděleno adekvátní rozlišení stínové mapy.

Perspektivní stínové mapy jsou počítány v prostoru po perspektivní transformaci pohledové kamery, tedy v normalizovaných souřadnicích zařízení (dále jen PPS). Při běžném vykreslování jsou po perspektivní transformaci objekty blíže ke kameře větší než vzdálené objekty. Myšlenkou perspektivních stínových map je nejprve scénu transformovat do PPS a následně vytvořit standardní stínovou mapu z pozice transformovaného světla. Protože stínová mapa je vytvářena po perspektivní transformaci, dojde k razantnímu snížení, ne-li k úplnému odstranění perspektivního aliasingu. Obecně lze říci, že kvalita stínů u směrového zdroje světla je závislá na úhlu paprsků světla vůči směru pohledu. V ideálním případě jsou na sebe oba směry kolmé a se snižováním úhlu se perspektivní stínové mapy přibližují standardním uniformním.



Obrázek 2.6: Projekce objektů nacházejících se za kamerou se změnou pořadí průsečíků s paprskem.

Perspektivní stínové mapy navržené v [Sta02] mají několik problémů, se kterými je nutné se vypořádat. V případě světla za kamerou se mohou mezi světlem a pozicí kamery nacházet i objekty potencionálně vrhající stíny do prostoru obrazu a musí být obsažené ve stínové mapě. Před vytvořením stínové mapy je však nutné provést perspektivní transformaci podle matice kamery a ta ze své podstaty objekty nacházející se za kamerou transformuje za ro-

vinu v nekonečnu (infinity plane). Tímto se změní i pořadí, v jakém paprsek světla protíná jednotlivé objekty, viz obrázek 2.6. Původní autor navrhuje dvě řešení. Vytvořit dvě stínové mapy, jednu pro scénu před kamerou a druhou pro objekty za kamerou nebo posunout pozici kamery dozadu a zkonstruovat takzvanou virtuální kameru, která obsahuje veškerou potřebnou geometrii scény. Posunutí má pak vliv na míru perspektivního zkreslení a čím více je kamera posouvána směrem dozadu, tím se zmenšuje úhel mezi jejími paprsky. V extrémním případě, při posunutí kamery do nekonečna, se bude jednat o paralelní projekci a prakticky jde u uniformní stínové mapy. V praxi má pak i malý posun výrazný vliv na výsledný aliasing. Další obecný problém stínových map je samozastínění nejčastěji řešené malým konstantním posunutím hodnot hloubky ve stínové mapě ve směru paprsků světla. U perspektivních stínových map, tedy vytvořených po perspektivní transformaci, není možné použít konstantní hodnotu, protože rozložení ve stínové mapě je neuniformní a velmi závislé na pozici světla vůči kameře. V takovém případě může být problematické najít vhodnou hodnotu posunu i ve statické scéně, protože hodnota dostačující pro blízkou geometrii z pohledu kamery nemusí již dostačovat pro vzdálené objekty. Z výše popsaných problémů je zřejmé, že perspektivní stínové mapy se potýkají s řadou problémů zejména spojených s relativní pozicí světla vůči kameře. V ideálním případě je aliasing téměř odstraněn, zatímco jiné případy je nutné speciálně ošetřit, jinak nevykazují takové zlepšení oproti standardním stínovým mapám. Výše zmíněné problémy řeší například Kozlov v [Sim07].

2.3.2.3 Perspektivní stínové mapy v prostoru světla (LispSM)

Metoda perspektivních stínových map v prostoru světla odstraňuje nedostatky PSM a byla poprvé prezentována v roce 2004 [WSP04]. Na rozdíl od PSM je perspektivní chyba distribuována rovnoměrně mezi blízkými a vzdálenými objekty. Metoda využívá konstrukci speciálního perspektivního prostoru podle pozice světla vůči kameře. Míra perspektivního zkreslení je měněna automaticky nebo uživatelem. Tím dochází ke kompenzaci nevhodné pozice světla a výsledky se blíží standardním uniformním mapám v případě pohledu rovnoběžného s pohledem kamery (ideální případ) nebo PSM v situaci, kdy jsou na sebe směry kolmé. Ve výsledku se LispSM snaží kombinovat výhody uniformních stínových map i PSM a vytváří kvalitní stíny pro blízké i vzdálené objekty, viz obrázek 2.7.

2.3.2.4 Vylepšení stínových map

LispSM nebo perspektivní stínové mapy produkují poměrně dobré výsledky s minimálním navýšením výpočetní složitosti proti standardním uniformním stínovým mapám. Stále však existují vylepšující metody, jak z kategorie zvýšení výkonu, tak odstranění aliasingu. Naivní způsob řešení aliasingu, a to zvyšování rozlišení stínové mapy, není příliš vhodný vzhledem k rychlosti vykreslování spolu s velkými nároky na paměť.

Metoda zvaná Tiled shadow maps [Arv04] předpokládá, že celkové rozlišení stínové mapy je dostatečné, ale její plocha je špatně využita. Stínová mapa je rozdělena na malé části a následně vykreslena v malém rozlišení. Výsledek je analyzován a jednotlivým částem přiděleno potřebné rozlišení. Výsledkem je nerovnoměrné rozlišení stínové mapy podle výskytu hran stínů.

Prakticky u všech typů stínových map je vhodné používat při porovnávání hloubky ve stínové mapě takzvané filtrování (PCF). Filtrovací algoritmus jednoduše porovná výsledky

Obrázek 2.7: Porovnání kvality stínů vytvořených různými metodami. Zleva: uniformní stínové mapy, LispSM, PSM. Převzato z [WSP04].

okolních pixelů a vytvoří průměr. Daný pixel v obraze již není pouze zastíněn nebo nezastíněn, ale může být i částečně zastíněn. Takovýto postup razantně zlepší kvalitu stínů, nicméně na odstranění velkého aliasingu, kdy pro jeden pixel stínové mapy připadá mnoho pixelů obrazu, stále nestačí. V dnešní době je nejvýhodnější využívat hardwarovou podporu stínových map, která PCF již obsahuje a na dotaz, zda je pixel zastíněn, vrací úroveň zastínění.

Poslední navrženou metodou prakticky úplně odstraňující aliasing je rozostření stínové mapy [Sim07]. Tuto techniku však nelze použít přímo, protože stínové mapy nejsou ve formě obrázku, který by bylo možné rozmazat, ale obsahují informaci o hloubce. U objektů, na kterých jsou stíny rozostřeny, nelze použít samozastínění, ale vizuální vylepšení stínů je velmi dramatické, proto je technika vhodná zejména na rovné plochy, například zdi nebo na terén, kde je nežádoucí aliasing nejvíce vnímán. Aby bylo možné stíny rozostřit, je přidána k hloubkové mapě ještě klasická barevná textura. Při vytváření stínů je pak do barevné textury zaznamenána hodnota 1 pro terén a hodnota 0 pro ostatní objekty. Tímto způsobem získáme texturu obsahující hodnotu 0 v místech zastínění terénu. Na texturu je pak použit Gaussův filtr pro rozostření. Při zobrazování scény je pro veškeré objekty kromě terénu použita standardní hloubková textura a pro terén rozmazaná textura. Výsledky tohoto přístupu lze vidět na obrázku 2.8.

Rychlost aplikace stínových map je především závislá na tom, jak efektivně jsme schopní zobrazovat objekty scény. Kromě využívání moderních funkcí OpenGL a vyvarování se volání zastaralých metod může být rychlost výrazně vylepšena využitím algoritmů pro ořez neviditelných částí scény. Běžně využívané ořezávání pohledovým jehlanem spolu s metodou CHC++ založenou na hardwarových dotazech zastínění lze z pohledu stínových map vylepšit. Například použitím masky obsahující informaci o rozložení objektů, na které mohou dopadat stíny. Metodu zavedl Bittner a kol. v roce 2011 [BMSW11] a je vhodná zejména u komplexních venkovních scén (typicky městská zástavba). Nejprve je scéna vykreslena z pozice kamery a je vytvořena maska pomocí stencil bufferu obsahující viditelnou geometrii schopnou přijímat stíny. Masku je možné rozdělit na tři typy podle přesnosti zobrazení geometrie, nejméně přesné je vykreslení obálek objektů, přesnější varianta pak obsahuje přesný tvar objektu a do nejlepší varianty jsou zaneseny jenom fragmenty, na kterých potenciálně může vzniknout stín. Při následném vytváření stínové mapy je pak vykreslena jenom geo-

Obrázek 2.8: Stíny s použitím PCF (vlevo). Rozostření stínů na rovných plochách (vpravo). Převzato z [Sim07]

metrie, jejíž obálka zasahuje do masky. Díky tomuto přístupu lze renderování velké městské zástavby urychlit 3x-10x. Hlavní problém této metody je zobrazení velkých objektů zasahujících do celé scény, jako je terén, protože spolehlivě vyplní masku a nemůže dojít k žádnému dalšímu ořezu scény při vykreslování z pozice světla. Takovou geometrii stačí rozdělit na menší části a použít normální postup.

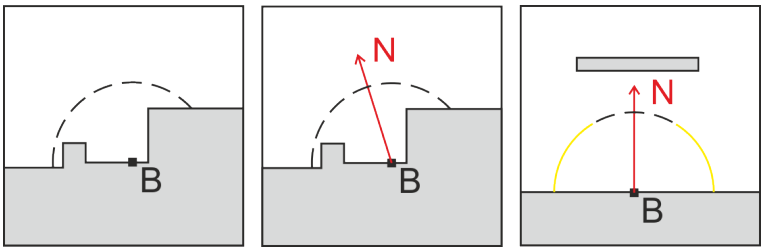
2.4 Zastínění ve scéně

Simulace nepřímého osvětlení scény v reálném čase je nejobtížnější problém v počítačové grafice. Počítat osvětlení přesně podle fyzikálního modelu šíření světla je prakticky nemožné, a proto byla vyvinuta řada algoritmů aproximujících přesný výpočet používaný v aplikacích pracujících v reálném čase. Určité zjednodušení přináší výpočet ambientního zastínění typicky vytvářející měkké stíny v rozích nebo mezi blízkými objekty.

2.4.1 Zastínění okolím (AO)

Zastínění okolím dodává scéně více realističnosti v podobě simulace reálného chování světla v oblastech s horším přístupem světla. Zastínění bodu je pak dáno mírou, jakou ho obklopuje zbytek objektů ve scéně, a tím blokuje přístup paprsků ze světelného zdroje. Typicky se nad bodem vytvoří hemisféra a zastínění bodu určuje plocha hemisféry zasahující do sousedních objektů, viz obrázek 2.9. Výpočet je pak prakticky prováděn vysláním paprsků z počítaného bodu do okolí a následným akumulováním zastiňujících bodů. Při akumulaci je brána v potaz i vzdálenost a úhel vyslaného paprsku, na kterém závisí míra zastínění. Blízký pixel má tedy na zastínění výrazně větší vliv než vzdálený a podobně paprsek s velkým úhlem vůči normále nemá takový vliv jako paprsek vyslaný ve směru normály. Nejčastěji jsou paprsky generovány v náhodném směru s rovnoměrným rozložením. Původní metoda počítá zastínění v prostoru scény a uchovává si informaci o zastíněných a nezastíněných místech. Tento přístup je velmi výpočetně náročný a data musí být obvykle předpočítána, proto není vhodná pro dynamické scény, kde by musel být výpočet opakován při posunu objektů vůči sobě. Naopak

výhoda metody je absolutní nezávislost na pozici světel ve scéně. Často se u této metody vypočítává skloněná normála vytvořená jako průměr všech nezastíněných směrů. Skloněnou normálu lze pak využít k vyhledávání v mapě prostředí, protože přesněji reprezentuje chování světla ve scéně. Nicméně i tato normála v některých případech nebude správně reprezentovat převažující směr dopadu světla, viz obrázek 2.9.



Obrázek 2.9: Hemisféra bodu scény B zasahující do okolních objektů (vlevo). Skloněná normála N vypočítaná z nezastíněných směrů (uprostřed). V případě, že světlo přichází ze stran, skloněná normála neodpovídá převažujícímu směru světla (vpravo)

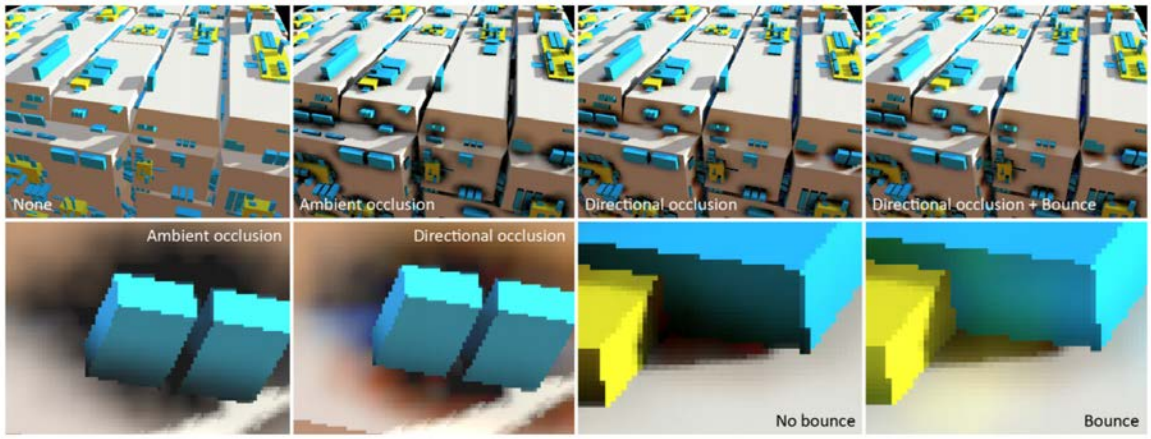
2.4.2 Zastínění okolím v prostoru obrazu (SSAO)

SSAO je vhodná metoda použitá spolu s odloženým stínováním, protože kompletně pracuje v prostoru obrazu. K aproximaci globálního osvětlení, tedy zastínění v těžko dostupných místech, je využívána pouze informace o normálách a hloubkách jednotlivých pixelů obrazu. V případě odloženého stínování jsou tato data již uložena v g-bufferu a SSAO může být počítáno při vykreslení obdélníku přes celou obrazovku pomocí fragment shaderu na GPU. SSAO je velmi oblíbená metoda implementovaná v širokém spektru moderních her pro její velmi dobré výsledky vzhledem k nízké výpočetní složitosti a jednoduchosti implementace. Díky práci v prostoru obrazu lze metodu aplikovat po stínování jako dodatečný efekt naprosto nezávislý na složitosti scény. Poměr mezi kvalitou výstupu a rychlostí lze pak snadno řídit nastavením proměnlivých parametrů, jako je například počet emitovaných paprsků pro test zastínění. Naopak výpočet v prostoru obrazu přináší některé problémy. Typicky se jedná o zastínění určitých částí scény, které by zastíněné být neměly. Tento jev vyvolává geometrie postavená blíže před kamerou, která v prostoru obrazu částečně zastiňuje okolí zpracovávaného pixelu, přitom jsou tyto objekty ve scéně velmi vzdáleny. Efekt působí nepřirozeně, ale naštěstí ho lze poměrně snadno odstranit nastavením prahu vzdálenosti a při kontrole okolí nebrat v úvahu zastínění příliš vzdálenými objekty. Ve výsledku je zastínění aplikováno v několika fázích. Nejdříve je vytvořena textura obsahující zastínění ve scéně. Tato textura musí být následně filtrována, protože náhodné rozmístění vzorků generuje šum a nakonec je použita pro úpravu barvy pixelů v obrázku, aby vznikl zastíněný obraz.

2.4.3 Směrové zastínění v prostoru obrazu (SSDO)

SSAO je vytvářeno nezávisle na výpočtu osvětlení, a proto umožňuje pouze černobílé zastínění. Výsledný obraz je obvykle složen z textury obsahující zastínění a obrazu osvětlené

scény. SSDO je podobné metodě SSAO s tím rozdílem, že je vyhodnocováno přímo při výpočtu osvětlení. Vzorky vygenerované na hemisféře ve směru normály zpracovávaného bodu určují, jaká místa jsou zastíněna. Zastíněné směry jsou pak vyřazeny z výpočtu osvětlení. Díky této směrové informaci mohou stíny dostávat barevný nádech podle mapy okolí a z různě barevných světel. Zastíněné směry mohou být také použity k vytvoření nepřímých odrazů [RGS09]. Přístup spočívá v předpokladu, že objekty v zastíněných směrech vyzařují svou barvu ve směru normály povrchu. Podle vzdálenosti a natočení normály je pak k bodu P přičten vliv nepřímého osvětlení. Normály omezí přičtení nepřímého osvětlení z odvrácených ploch a vzdálenost omezí vliv příliš vzdálených objektů. Metoda je stejně jako SSAO nezávislá na komplexnosti scény a pracuje pouze v prostoru obrazu, na rozdíl od SSAO bere v potaz barvu vyskytující se ve scéně, ale je náročnější na výpočet. SSDO ve srovnání se standardním SSAO je znázorněno na obrázku 2.10.



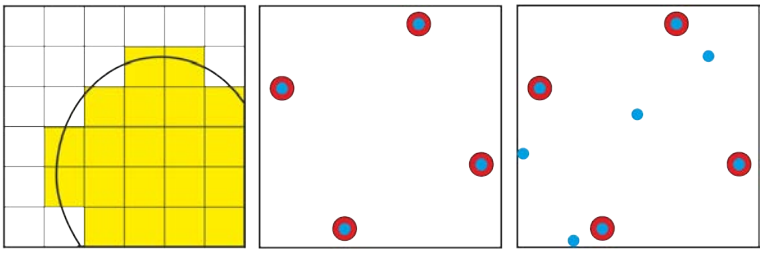
Obrázek 2.10: Nahoře zleva: Scéna bez AO, běžné SSAO, SSDO, SSDO s nepřímými odrazy. Dole zleva: SSAO, SSDO, SSDO bez odrazů, SSDO s odrazy. Převzato z [RGS09]

2.5 Antialiasing

Počítačový obraz prakticky od jeho počátku trápí aliasing způsobený zobrazením spojitých dat na zařízeních obsahujících diskrétní mřížku a s tím spojenou rasterizací. V dnešní době již existují monitory s rozlišením vyšším, než je schopné lidské oko rozpoznat, ale stále naprostá většina monitorů obsahuje rozlišení značně nižší a při rasterizaci dochází, zejména na šikmých hranách objektů, k vytváření zubů, viz obrázek 2.11. Se stoupajícími nároky na kvalitu obrazu musel být tento problém řešen a vzniklo poměrně velké množství metod, z nichž ty nejzákladnější zde budou popsány.

2.5.1 Supersample antialiasing (SSAA)

Jedná se o antialiasing produkující nejlepší výsledky, ale je poměrně pomalý. SSAA funguje na principu zobrazování ve vyšším rozlišení. Tedy pro každý pixel je vytvořeno několik



Obrázek 2.11: Aliasing způsobený rasterizací při zobrazování oblého objektu (vlevo). Více vzorků v jednom pixelu při použití SSAA nebo MSAA (uprostřed). Červeně vzorky, ve kterých je počítáno osvětlení a modře osm dodatečných vzorků použitých v 4xEQAA (vpravo)

vzorků, viz obrázek 2.11. U každého z nich vypočítáno osvětlení a barva výsledného pixelu je určena jejich průměrem. Počet vzorků připadajících na jeden pixel obrazu se liší podle úrovně antialiasingu a typicky se jedná o hodnoty 2, 4, 8 nebo 16. Z toho vyplývá, že pro obraz o rozlišení 800 x 600 px s antialiasingem typu 4xSSAA musí být vypočítáno plné osvětlení v rozlišení 1600 x 1200 px. U moderních her představuje výpočet osvětlení značnou část celkového času, a proto byly vyvinuty úspornější metody, jak na výpočetní čas, tak na paměť zabranou na GPU.

2.5.2 Multisample antialiasing (MSAA)

Osvětlení jednotlivých pixelů je často počítáno v komplexních programech na grafické kartě, které při výpočtu berou v úvahu barvu textury, hloubku fragmentu nebo hodnotu stencil bufferu. V SSAA jsou tyto programy spuštěny pro několikanásobně větší počet pixelů a dochází k razantnímu zvýšení výpočetních nároků. MSAA vylepšuje tuto metodu a program na grafické kartě spouští pouze jednou pro každý pixel a více vzorků je vytvářeno pouze na hranách polygonů. MSAA bývá plně podporován grafickou kartou a je aktuálně nejpoužívanější metodou antialiasingu ve hrách a podobných aplikacích s přímým způsobem zobrazování.

2.5.3 Coverage sample antialiasing (CSAA)

Jedná se o vylepšení MSAA výrobcem grafických karet Nvidia (ATI má ekvivalentní metodu s názvem Enhanced quality antialiasing - EQAA) [Don11]. V principu se jedná o klasické MSAA doplněné o dodatečné vzorky. U vzorků je vyhodnoceno, zda se nacházejí na rasterizovaném polygonu či nikoliv a podle toho je vážena výsledná barva pixelu. Pro grafické karty je poměrně snadné doplnit výsledný výpočet o větší počet vzorků, takže výsledná náročnost není příliš navýšena proti standardnímu MSAA, ale výsledkek může značně přispět ke kvalitě v závislosti na rozložení dodatečných vzorků v pixelu. Příklad dodatečných vzorků, ve kterých není počítáno osvětlení, je znázorněn na obrázku 2.11.

2.5.4 Antialiasing pro odložené stínování (SRAA)

V případě, že aplikace využívá odložené stínování, není možné použít MSAA a je nezbytné zvolit vhodný algoritmus pracující nad hotovým obrázkem. SRAA využívá výpočet osvětlení

pro jeden vzorek pixelu, ale subpixelovou přesnost pro řešení viditelnosti [CML11]. V případě, že máme hotový obraz, tedy vypočítané osvětlení v každém pixelu, metoda dodatečně vygeneruje více vzorků pro každý pixel, ve kterém uloží informaci o hloubce a normále. Podle těchto informací je pak detekována diskontinuita na hranách objektů a spočítána váha příspěvků okolních pixelů k výsledné barvě. SRAA produkuje výsledky srovnatelné minimálně s 4xMSAA a nízkou časovou náročností. Nevýhoda tohoto algoritmu spočívá v nutnosti vytvořit dodatečný buffer obsahující hodnoty normál a hloubky získané multisamplingem. V originálním článku navrhují metodu urychlení s použitím pouze hloubkového bufferu, čímž je značně redukováno čtení a zápis do paměti.

2.6 Hloubka ostrosti

Mocný nástroj pro vyjádření prostoru nejen ve fotografii, ale i v syntéze obrazu, je hloubka ostrosti. Prakticky se jedná o vzdálenost ve scéně, ve které se objekty jeví jako ostré. Objekty mimo tuto vzdálenost jsou rozmazané a lidský mozek je příliš nevnímá. Lidský zrak z principu není schopný vidět celou scénu správně zaostřenou a tato vlastnost platí i pro optické soustavy používané ve fotoaparátech. Simulací hloubky ostrosti lze kromě zvýraznění prostoru určit, na jaká místa člověk zaměří pozornost a bude je detailně sledovat. Správné využití hloubky ostrosti má obrovský význam v digitální fotografii a při správném využití se obraz jeví pro pozorovatele naprosto přirozeně. Z těchto důvodů je tato technika často využívána v moderních počítačových hrách, mnohdy se podobajících celovečerním filmům. Základní princip metod zabývajících se hloubkou ostrosti při syntéze obrazu je poměrně jednoduchý. Důležitá je znalost hloubky ve scéně, podle které je možné dopočítat míru rozostření v oblastech mimo hloubku zaostření. K rozmazání je obvykle využito základní filtrování obrazu Gaussovým filtrem nebo jednoduše průměrování okolních pixelů. Gausián lépe respektuje okolí zpracovávaného pixelu a vzdálenějším částem obrazu nepřikládá takový význam. Na druhou stranu pro stejnou úroveň rozmazání, kterou lze dosáhnout průměrováním okolních bodů, je nutné využít větší okolí (kernel). To představuje v programech na grafických kartách více čtení dat z paměti a patřičné zpomalení procesu.

2.7 Zrcadlové plochy

Zobrazení zrcadel a podobných odrazivých ploch je pokročilý efekt, který již není součástí veškerých aplikací využívající 3D zobrazování, ale v některých případech se jedná o důležitou vlastnost, kterou je nutné implementovat. Pro vykreslení zrcátek a různých odrazivých ploch lze použít dva rozdílné přístupy.

První z nich využívá klasického rasterizačního řetězce a je díky své rychlosti velmi rozšířená. Její princip je velmi jednoduchý a popíšeme si jej na příkladu obyčejného zrcátka. Prakticky se jedná o vykreslení odražené scény z pozice zrcadlové plochy. Nejprve je nutné správně zkonstruovat projekční matici zajištující vykreslení odraženého obrazu scény na místo zrcadlové plochy. Pokud bychom přímo použili tuto matici, dojde nejen k vykreslení zrcadlového obrazu na správné místo, ale i na místa zrcátku neodpovídající. Proto je nezbytné zápis do obrazu omezit pouze na místo odrazivé plochy. Omezení je zajištěno pomocí stencil bufferu, do kterého je na začátku geometrie zrcátka vykreslena a jednoduchý test

zajistí vykreslení pouze na korektní místo. Metoda zobrazení zrcátek pomocí stencil bufferu má hlavní nevýhodu v zobrazení zakřivených zrcátek. Zrcátka musí být rozdělena na menší ploché části nebo je nutné použít jiný způsob.

Další mnohem obecnější, ale výpočetně náročný přístup, je zobrazit zrcátka pomocí sledování paprsku (ray tracing). Metoda je prakticky nezávislá na tvaru zrcátka a různorodě zakřivená zrcátka není problém zobrazit za cenu větší výpočetní náročnosti. V případě sledování paprsku stačí generovat paprsky pouze pro takové části obrazu, kde je viditelná alespoň část odrazivé plochy, vypočítat odražené paprsky a následně jejich průsečík se scénou. V případě nalezení průsečíku může být klasickým způsobem vypočítána barva a přiřazena danému pixelu ve výsledném obrazu.

Kapitola 3

Zobrazovací řetězec založený na odloženém stínování

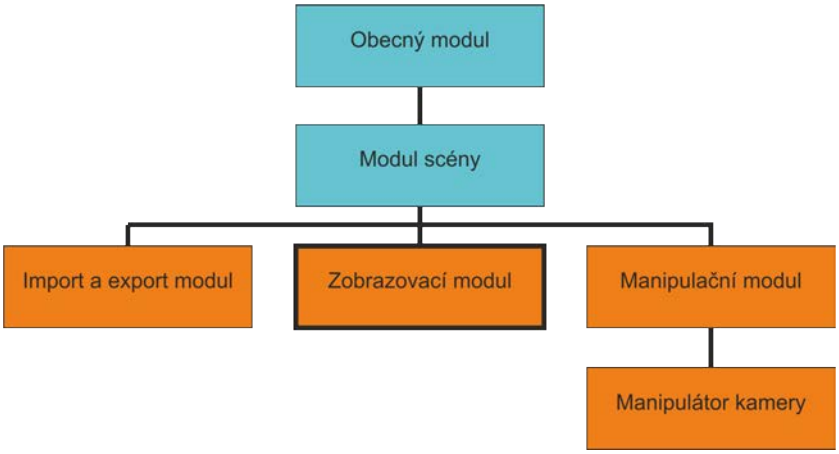
Tato kapitola se zabývá návrhem a realizací zobrazovacího modulu založeného na odloženém stínování. Nejprve je popsáno implementační prostředí následované částí pojednávající o návrhu architektury nového modulu spolu s důležitými vlastnostmi, jako je zpracování událostí nebo sdílení dat. Velká je část věnována návrhu a implementaci základních i doplňkových prvků zobrazovacího řetězce, které zajištující samotné vykreslování.

3.1 Implementační prostředí

Návrh a implementaci bylo potřeba přizpůsobit prostředí VRUT (Virtual Reality Universal Toolkit). Jedná se o modulární multiplatformní aplikaci vyvíjenou ve spolupráci ČVUT a firmou Škoda auto.

Systém se skládá z několika typů základních modulů (IO, manipulátor, zobrazovací, modul scény, obecný) a jádra [Vá09]. Jednotlivé moduly pak tvoří hierarchii znázorněnou na obrázku 3.1. Části aplikace mezi sebou komunikují pomocí událostí, které jsou generovány a následně přeposílány všem registrovaným modulům. Pokud tedy chce například zobrazovací modul zpracovávat nějakou událost (potřebuje detekovat ID scény, kterou má zobrazit), zaregistruje si daný typ události. V případě, že přijde událost tohoto typu (změna ID scény) do jádra, je automaticky přeposlána zobrazovacímu modulu. Tento koncept umožňuje snadnou rozšiřitelnost novými moduly a hlavně relativní nezávislost modulů na sobě. Zobrazovací modul nemusí znát implementaci IO modulu starajícího se o načtení scény a naopak IO modul načte scénu, vygeneruje patřičnou událost a další zpracování scény je pro něj nepodstatné.

V současné době je pro zobrazování ve VRUTu využíván modul s názvem *rendergl*, jehož koncept je zastaralý a nepřehledný. To činí problémy z hlediska dlouhodobé udržitelnosti a hlavně začlenění nových sofistikovaných algoritmů. Díky tomuto nedostatku vznikla ve VRUTu řada modulů jako pouhá kopie, ve které byla následně změněna nebo přidána požadovaná funkcionalita. Pokud takovéto moduly nejsou často využívány, je poměrně pracné a zbytečné je udržovat plně funkční. Tento problém současný systém řeší vyčleněním obecné funkcionality přímo ze zobrazovacího modulu do jádra. Vznikla tak třída *renderGLmodule*



Obrázek 3.1: Hierarchie modulů ve VRUTu se zvýrazněným zobrazovacím modulem, kterým se zabývá tato práce.

obsahující obecné zobrazovací metody a algoritmy, které pak mohou využívat všechny moduly. Tím je omezena duplicita kódů, ale jen částečně. Postupem času se pak stala třída *renderGLmodule* tak rozsáhlou, že provedení jakékoliv změny je poměrně problematickou záležitostí, pro kterou je nutné mít velký přehled o aktuální implementaci.

3.2 Použité technologie

VRUT je postaven na multiplatformní knihovně wxWidgets, kterou vytvořil v roce 1992 Julian Smart [Kom]. WxWidgets je knihovna grafických elementů pro tvorbu grafického uživatelského rozhraní (GUI) implementovaná v C++, ale použitelná v mnoha běžných jazycích. Výhoda wxWidgets spočívá ve využívání nativních grafických prvků na všech platformách namísto napodobování grafiky dané platformy.

Pro zobrazování je použita knihovna OpenGL vyvinutá v roce 1992. OpenGL je rozhraní pro tvorbu 2D i 3D grafických aplikací. Používá se nejen při tvorbě her, ale i ve vědecké vizualizaci nebo aplikacích virtuální reality. Implementace OpenGL existuje prakticky pro všechny široce používané platformy. Jedná se o rozšířený standard s velmi dobrou dokumentací a stabilní podporou výrobci grafických karet. Aktuální verze je OpenGL 4.3, přičemž nové verze jsou úzce spjaté s jazykem GLSL používaným ke psaní programů pro GPU.

Přesněji se jedná o programovací jazyk určený k psaní instrukcí zpracovávaných grafickou kartou při vykreslování objektů. GLSL umožňuje programátorům plně kontrolovat průběh vykreslování, přičemž v moderních verzích OpenGL je tento systém zobrazování přímo vyžadován. Dříve využívaná funkcionalita statického zobrazovacího řetězce, kdy programátor neměl možnost ovlivnit vykreslování jinak než nastavením stavu OpenGL, je v dnešní době stále podporovaná, ale nedoporučuje se ji používat. Existují tři základní druhy programu pro GPU. Jedná se o program zpracovávající jednotlivé vrcholy geometrie (vertex shader), program ovlivňující geometrii (geometry shader) a poslední program pracující nad pixely výsledného obrazu (fragment shader). Všechny části jsou kompilovány aplikací za běhu a z nich

vytvořen program pro GPU, který je v případě potřeby aktivován a veškeré vykreslování pak probíhá přes něj.

3.3 Požadavky na implementaci

Cílem práce byla implementace nového zobrazovacího modulu do systému VRUT. Modul by měl plně nahradit stávající zobrazovací modul *rendergl*, přičemž je nutné implementovat veškerou důležitou funkcionalitu a podporovat všechny druhy materiálů a světel jako současná implementace. Nový modul by měl zároveň rozšířit stávající funkcionalitu o nové vlastnosti, jako je výpočet stínů a podobné efekty zlepšující výslednou kvalitu obrazu. Neméně podstatné vlastnosti jsou rozšiřitelnost, případně snadná úprava funkcionality a v neposlední řadě rychlost zobrazování. VRUT je aplikace používaná pro zobrazování na stereoskopických stěnách i v zařízení typu CAVE v reálném čase, proto je rychlost zobrazování naprosto zásadní aspekt ovliňující výslednou použitelnost.

3.4 Zobrazovací modul

Instance zobrazovacího modulu VRUTu se vytváří pro každé okno určené pro vykreslení scény. V případě, že grafické rozhraní obsahuje více nezávislých oken, každá instance zobrazovacího modulu musí mít i svůj vlastní OpenGL kontext. V současné době VRUT obsahuje nejvíce používanou verzi zobrazovacího modulu *rendergl* a několik experimentálních modulů testujících různorodé efekty vytvořené v rámci školních projektů.

Nový zobrazovací modul vytvořený v této práci nese název *rendergl2* a kompletně mění zavedenou architekturu ve stávajících vykreslovaných modulech. Cílem bylo navrhnout nezávislý samostatný modul, který bude možné vyvíjet a vkládat do něj zobrazovací komponenty bez ovlivnění jiných modulů VRUTu a bez přílišného rizika zavedení chyb do stávajícího kódu. Takový modul bude nejen vhodný pro účely výuky, ale zároveň je možné přizpůsobit jeho skladbu aktuálně zobrazované scéně, a tím maximálně využít jeho potenciál. Zobrazovací modul se skládá z částí dohromady tvořících zobrazovací řetězec. Takovéto části (moduly) budou dále nazývány zobrazovací průchody, aby nedocházelo k záměně s názvem zobrazovací modul určeným pro modul VRUTu jako celek. Pojmem zobrazovací průchod vykreslovacího modulu je tedy myšlena taková část kódu, která představuje komplexní řešení určitého problému. Například komponentu vytvářející g-buffer označíme za zobrazovací průchod, i když komponenta sama o sobě může a často i obsahuje víceprůchodové algoritmy.

Celý zobrazovací modul je založený na využití odloženého stínování, ale jeho struktura dovoluje využít i přímé zobrazování v závislosti na implementovaných vykreslovacích průchodech. Jako základní koncept byla zvolena metoda jednoduchého sekvenčního řazení, viz kapitola 2.1.2. Sekvenční řazení zobrazovacích průchodů neobsahuje výpočetně náročnou režii a pro daný účel se jedná o naprosto dostačující řešení. Pří správné implementaci zobrazování pomocí odloženého stínování navíc dochází k dostatečnému vytížení GPU i při sekvenčním běhu programu.

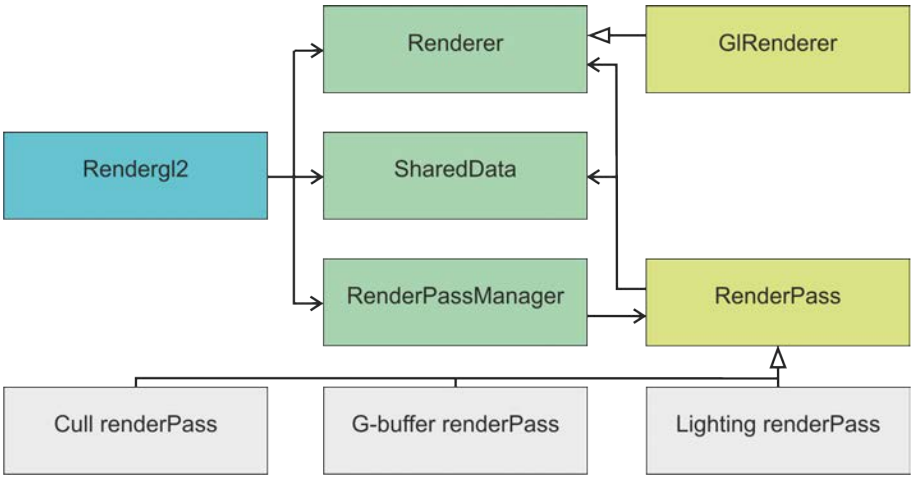
Při zobrazování v reálném čase pomocí odloženého stínování jsou uložena data do g-bufferu, následně vypočítáno osvětlení a pak již závisí jen na kreativitě programátora nebo

uživatele, jaké průchody ovlivňující výsledný obraz zařadí. Je možné obraz přiblížit realitě pomocí sofistikovaných hi-end algoritmů stejně jako použít jakýkoliv umělecký efekt.

Během implementace bylo dbáno na použití moderních metod zobrazování, a proto je nový modul kompletně implementován v čistém OpenGl 4.0 s maximálním využitím programovatelného grafického řetězce.

3.4.1 Návrh architektury

Jádro zobrazovacího modulu bylo navrženo s ohledem na požadavky celého systému. Pro modulární zobrazování v reálném čase je nutné navrhnout strukturu respektující několik zásadních pravidel. Nejen, že struktura musí být přehledná, ale zároveň musí zajišťovat distribuci dat mezi zobrazovacími průchody tak, aby byla omezena nebo zcela odstraněna duplicita v hlavní paměti i paměti GPU. Současně je důležité, aby všechny části zobrazovacího řetězce měly přístup k důležitým funkcím pro vykreslování. Je vhodné vytvořit rozhraní (API) akumulující podstatné odladěné vykreslovací funkce. Jakýkoliv průchod pak může volat dostupnou implementaci a kromě omezení duplicity kódu je přinášen určitý komfort do vývoje nových částí, kdy se programátor nemusí zabývat implementací základních prvků. Zjednodušený návrh architektury modulu je znázorněn na obrázku 3.2.



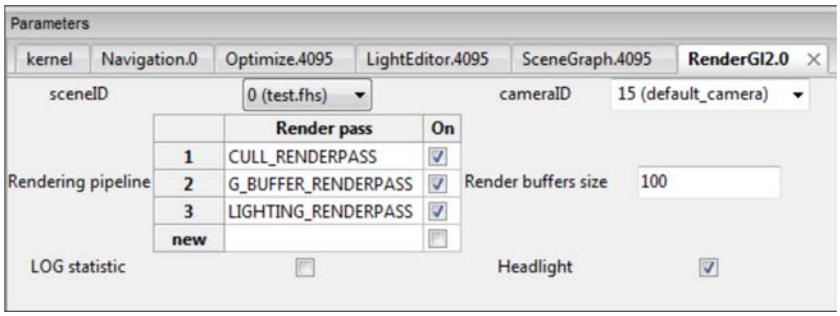
Obrázek 3.2: Architektura nového zobrazovacího modulu.

Jádro se jmenuje shodně s celým modulem *rendergl2* a stará se o běh celého vykreslování. Tato hlavní část distribuuje algoritmickou třídu *GlRenderer* obsahující obecné metody pro zobrazování mezi průchody, viz sekce 3.4.2. Sdílení třídy *GlRenderer* umožňuje souhrnně používaná grafická data uložená na GPU využívat ve všech zobrazovacích průchodech pomocí veřejných metod za tímto účelem vytvořených. Ze schématu je patrné, že jednotlivé zobrazovací průchody na sobě nemají žádnou závislost. Veškerá sdílená data pak nese obecná třída *GlRenderer* nebo třída *SharedData*. Třída *SharedData* je určená pouze pro sdílení dat mezi průchody, kterým ji předává jádro zobrazovacího modulu. Sdílením dat se podrobněji zabývá sekce 3.4.1.3.

Hlavní část *renfergl2* zahrnuje i režii běhu jednotlivých průchodu, reakci na události systému nebo GUI. Všechny důležité části budou popsány v následujících sekcích.

3.4.1.1 Správa průchodů

Zobrazovací průchody mohou být začleněny do průběhu zobrazování přidáním do tabulky pomocí GUI, viz obrázek 3.3. Je zde možné přidávat, odebírat, měnit nebo dočasně zastavit jakýkoliv zobrazovací průchod. Zobrazování pak probíhá podle pořadí v tabulce. Vrátíme-li se k obrázku 3.3, první proběhne ořezávání scény, pak vytvoření g-bufferu a na konec stínování. Pomocí tabulky je možné přehledně reprezentovat sekvenční zobrazovací řetězec, přičemž systém registrace zobrazovacích průchodů je navržen tak, aby nebyl problém zařadit do průběhu vykreslování i několik stejných průchodů s jinak nastavenými parametry. Vyšší uživatelský komfort je dosažen minimalizováním nároku na paměť uživatele a jednotlivé průchody lze měnit pomocí rozbalovací nabídky pouhým výběrem požadované možnosti. Občas je vhodné nastavení uchovat pro další potřebu, například při vývoji nebo během přípravy vizualizační prezentace. V takové situaci by bylo krajně nepraktické pokaždé znovu „naklikávat“ nastavení pomocí GUI, proto lze celý zobrazovací řetězec konfigurovat pomocí jazyku JavaScript automaticky při startu nebo kdykoliv za běhu VRUTu.



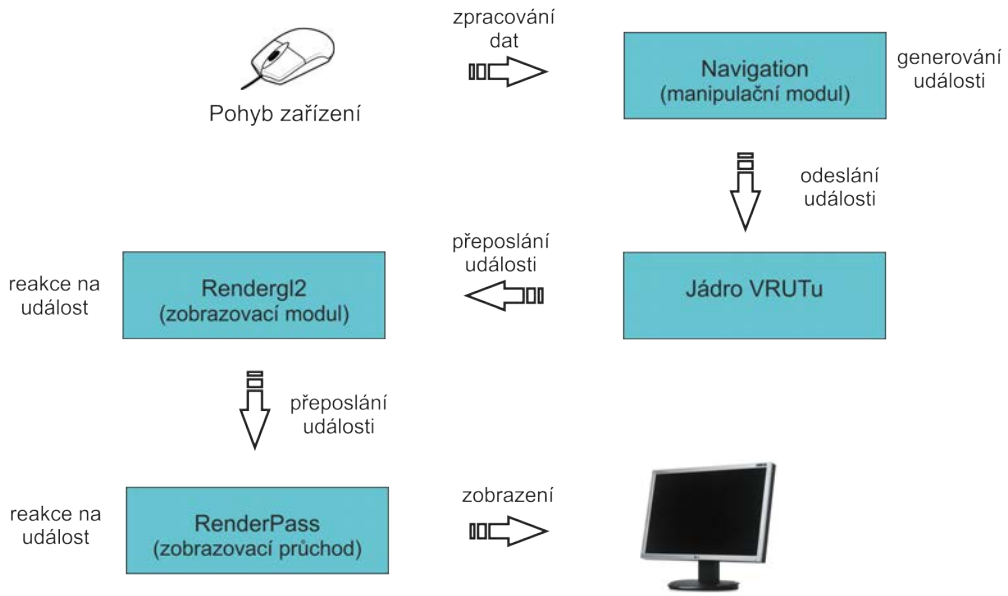
Obrázek 3.3: Uživatelské rozhraní pro nastavení zobrazovacího řetězce. Průchody jsou spouštěny v odpovídajícím pořadí uvedení v tabulce.

Každá změna v zobrazovacím řetězci je detekována, aby mohlo být vykreslování zastaveno v případě chybné konfigurace a bylo tak možno předejít případným problémům. Nastavení je nevalidní, pokud není splněna závislost pro nějaký průchod. Typicky se jedná o nedodržení základních principů nastavení, kdy je například nastaven průchod počítající osvětlení scény před průchodem vytvářejícím g-buffer. Ke správě a validaci konfigurace průchodů byla vytvořena třída *RenderPassManager*. Tato třída udržuje nastavení zobrazovacího řetězce, je schopná jej validovat a kdykoliv vrátit aktuálně nastavenou konfiguraci připravenou k vykreslování.

3.4.1.2 Zpracování událostí

Ve VRUTu veškerá komunikace probíhá na bázi událostí posílaných přes jádro modulům. Závisí tedy pouze na typu modulu, na jaké události by měl reagovat a jaké sám generuje. Například navigační modul obsluhující kameru zareaguje na pokyn od uživatele, odpovídajícím

způsobem nastaví kameru a vygeneruje patřičnou událost, kterou odešle jádru. Zobrazovací modul si zaregistruje typy událostí, které si přeje přijímat a v případě, že se typ shoduje s událostí přijatou jádrem, je tato událost přeposlána dotyčnému modulu. Jak s událostí naloží modul je čistě v režii programátora, v extrémním případě událost zahodí, ale v případě zobrazovacího modulu a pohybu kamery by měl minimálně překreslit scénu. Stejným způsobem jsou ve VRUTu generovány stovky událostí a je povinností zobrazovacího modulu si řádně zaregistrovat a zpracovávat vhodné zprávy. Zejména se jedná o události týkající se stavu scény, jako je zmiňovaný pohyb kamery. Všechny události je vhodné distribuovat i do vykreslovacích průchodů, aby na ně mohly adekvátně zareagovat v metodě *processEvent*, kterou musí povinně implementovat. Situaci předání události od uživatele až do zobrazovacího průchodu znázorňuje obrázek 3.4.



Obrázek 3.4: Přeposílání události pohybu kamery pomocí myši přes jádro VRUTu až do jednotlivých průchodů v zobrazovacím řetězci.

3.4.1.3 Sdílení dat

Máme-li dva průchody, jeden vytvářející g-buffer a druhý počítající osvětlení, logicky mezi sebou musí sdílet data. V tomto případě se jedná o g-buffer, do kterého jeden průchod zapisuje a další informace využívá. Obecně je nutné definovat systém sdílených dat mezi průchody pro dosažení maximální flexibility. Na druhou stranu zobrazovací průchody musí být natolik nezávislé, aby bylo možné jakýkoliv průchod nahradit jiným, v danou chvíli vhodnějším. Sdílení dat umožňuje jádro zobrazovacího modulu, jednotlivé průchody při inicializaci zaregistrují data potřebná pro svůj bezchybný běh a zaregistrují i data, která inicializují. Správce všech průchodů je potom schopný kontrolovat závislosti a inicializaci potřebných dat. Vrátíme-li se k příkladu s průchodem vytvářejícím g-buffer a následným průchodem pro zpracování, první průchod zaregistruje inicializaci g-bufferu a druhý identifikuje g-buffer jako

nutný vstupní parametr. Pokud první průchod nebude aktivní a tím pádem nebude g-buffer vytvořen, nežádoucí situace je detekována a renderování zastaveno.

Ve sdílených datech se rovněž nachází nekontrolovaná data, která automaticky inicializuje jádro modulu a jsou nezbytná ke správnému chodu. Typickým příkladem jsou dva zobrazovací buffery (FBO), jeden pro čtení a druhý pro zápis. V průběhu vykreslování jsou obě FBO k dispozici všem průchodům, z nichž každý zapisuje do bufferu určeného pro zápis a před ukončením kreslení je zodpovědný za zavolání metody, která buffery prohodí. Metoda dvou FBO zamezuje kopírování dat z jednoho FBO do záložního před samotným zápisem dat, které by většina složitějších průchodů musela provádět. Z výkonnostních důvodů mají oba FBO sdílenou část obsahující hloubku. Díky sdílení mohou být prohazovány pouze ukazatele a okamžitě do FBO zobrazována nová data bez kopírování hloubkového bufferu.

3.4.1.4 Uživatelské rozhraní

Podobně jako registrace potřebných dat je nutné vytvořit uživatelské rozhraní. Uživatelské rozhraní ve VRUTu není příliš přehledné a je dobré množství ovládacích prvků omezit na rozumné minimum. Jádro zobrazovacího modulu obsahuje základní ovládací prvky pro nastavení aktivních zobrazovacích průchodů, rozlišení renderovaného obrazu, zapnutí ořezu odvracených ploch, intensitu ambientního světla, nebo zapnutí čelového světla (headlight). Jelikož zobrazovací průchody mohou být naprosto rozdílného charakteru a mít i své vlastní ovládací prvky, musí být jejich ovládání snadno přístupné s logickým uspořádáním. Kdybychom všechny tyto prvky zobrazili najednou, snadno by se mohlo stát uživatelské rozhraní naprosto nepřehledné. Prvky uživatelského rozhraní si jednotlivé průchody na začátku zaregistrují podobně jako v případě sdílených dat a jádro modulu je dynamicky přidává a odebírá z uživatelského rozhraní podle toho, zda je zobrazovací průchod zařazen do vykreslování či nikoliv. Jednotlivé ovládací prvky jsou nejprve označeny číslem zobrazovacího průchodu a až následně svým názvem. Jsou-li do zobrazovacího řetězce zařazeny dva nebo více stejných průchodů, je možné jejich ovládací prvky jednoduše rozeznat podle tohoto předřazeného identifikátoru.

Grafické prvky, jejich registraci i události generované při změně zpracovává jádro VRUTu a přeposílá zobrazovacímu modulu. Aby se o důležitých informacích o stavu ovládacích prvků nebo VRUTu dozvěděly i zobrazovací průchody, jejichž algoritmy na nich mohou být závislé, jsou jim tyto události rovněž přeposílány.

3.4.2 Podpora vykreslování

Pro dosažení snadného vývoje a zamezení plýtvávání časem při implementaci obecných vlastností, jako je například vykreslení geometrie, byla navržena třída *GlRenderer*. Jedná se o třídu implementující základní metody vykreslování včetně zahrnutí materiálů a světel, která je odvozena ze třídy *Renderer*, kam budou postupně přidávány nejobecnější metody použitelné ve všech zobrazovacích průchodech. *GlRenderer* je navržen podobně jako OpenGL, kdy je možné způsob vykreslení řídit pomocí stavů. Lze ovlivnit vykreslování v podobě zapnutí nebo vypnutí materiálů či světel adekvátními příkazy *Enable(vlastnost)* nebo *Disable(vlastnost)*.

3.4.2.1 Zobrazení geometrie

Jelikož je celý zobrazovací modul psaný v moderní verzi OpenGL, muselo tomu být přizpůsobeno i samotné vykreslování. Aktuálně je nutné veškeré zobrazení řešit pomocí shaderů a nevyužívat display listy nebo podobné zastaralé funkce, i když jsou stále podporovány většinou hardwaru. Geometrie je uložena přímo na GPU ve vertex buffer objektech (VBO), aby nedocházelo k žádnému zbytečnému přenosu dat kvůli maximální efektivitě při vykreslování. VBO obsahující pozice vrcholů, normály a texturovací souřadnice jsou pak sdruženy do vertex array objektu (VAO).

3.4.2.2 Druhy materiálů

Původní zobrazovací modul podporuje velké množství nastavení materiálů spojené s používáním fixního zobrazovacího řetězce. Standardní barvu a vlastnosti materiálu doplňuje texturování zvyšující úroveň detailů. Jako základ byl zvolen nejpoužívanější Phongův osvětlovací model počítaný pro každý pixel obrazu zvlášť.

Textury mohou mít nastavený typ kombinace s barvou materiálu podle specifikace OpenGL (např. modulování). Podobně může být definován typ mapování textury na objekt v případě, že nejsou texturovací souřadnice součástí modelu nebo mapování prostředí. Generování texturovacích souřadnic je počítáno přímo na GPU a jsou implementovány následující typy: generování v prostoru objektu, generování v prostoru kamery, sférické mapování a v případě mapy prostředí mapování na krychli.

Specialita VRUTu jsou materiály obsahující vlastní shader pro vykreslení. Pokud materiál obsahuje vlastní shader, není jej možné vykreslit současně s ostatními objekty, protože na GPU může být v jednu chvíli aktivní pouze jeden program. Je tedy nutné podobné objekty vykreslovat samostatně.

3.4.2.3 Druhy světel

V novém zobrazovacím modulu jsou implementovány běžné tři typy světelných zdrojů. Do scény je možné přidávat směrové, bodové nebo reflektorové typy světel. Nastavitelné parametry znázorněné v tabulce 3.1 jsou inspirovány jazykem VRML.

	Druhy světel		
Parametry	Směrové	Bodové	Reflektorové
barva	√	√	√
intenzita	√	√	√
ambientní intenzita	√	√	√
pozice	×	√	√
směr	√	×	√
útlum	×	√	√
úhel	×	×	√
exponent	×	×	√

Tabulka 3.1: Parametry světel inspirované jazykem VRML.

Světla jsou standardně na pozici (0, 0, 0) a mají směr (0, 0, -1), jejich skutečná pozice a natočení je dopočítána pomocí jejich světové transformační matice před nahráním do programu na GPU. Z tabulky je dobře zřetelná část parametrů, ve které se světla liší a které jsou naopak stejné. Odlišných vlastností lze s výhodou využít v libovolném shaderu pro rozpoznání druhu světla následující sérií podmínek.

```
if(length(smer) == 0.0)
    //bodové světlo
else if (uhel == 0.0)
    //směrové světlo
else
    //reflektorové světlo
```

3.4.2.4 Pozadí

Pozadí je ve VRUTu reprezentováno speciálním uzlem scény, který definuje barvu, případně texturu okolí. Do scény je možné vložit dva typy pozadí. První jednoduchý typ je tvořen pouze rovnou plochou s namapovanou texturou. Druhé běžně využívané řešení mapuje speciálně připravenou texturu na krychli, uvnitř které se nachází pozorovatel. Vykreslování pozadí probíhá s maximální možnou hloubkou, která však není zapisována. Tento přístup má výhodu v případě, že je pozadí vykreslováno na závěr. Nedochází totiž ke zbytečnému zapsání barvy pozadí na zastíněná místa jinými objekty.

3.4.3 Podpora vývoje nových průchodů

Jelikož je VRUT stále intenzivně vyvíjená aplikace a je předpoklad vytváření nových zobrazovacích průchodů v rámci semestrálních nebo diplomových prací, bylo podstatné připravit pro nový vývoj podmínky. V grafických algoritmech je nejpodstatnější optimalizace výpočtů na CPU i GPU pro zachování dostatečné zobrazovací rychlosti. Za tímto účelem jádro modulu obsahuje logování, které je možné zapnout pomocí GUI. Jedná se o výpis doby běhu jednotlivých průchodů, zvlášť na CPU a GPU. Náročnost na výpočetní čas GPU je zjišťována pomocí hardwarových dotazů na grafickou kartu vracející čas v nanosekundách. Logování neprobíhá, jak je ve VRUTu zvykem do konzole spolu se standardním logovacím souborem, ale do speciálního souboru s názvem *renderGl2Stats.log*, kam jsou v každém snímku zapisovány časy pro všechny zobrazovací průchody. Zápis do zvláštního souboru má své opodstatnění, protože dat je zapisováno velmi mnoho. Zároveň je pak snadné data automaticky zpracovat a vytvořit například graf rychlosti vykreslování v závislosti na průchodu scénou.

3.5 Vytvoření zobrazovacího průchodu

Vytvoření nového průchodu je poměrně snadná záležitost. Každý nový zobrazovací průchod je odvozen od obecné třídy *RenderPass* a musí implementovat dvě důležité metody *Draw()* a *ProcessEvent()*, které, jak jejich název napovídá, slouží k zobrazování nebo ke zpracování událostí. Metoda *Draw()* je volána v každém snímku v odpovídající době podle konfigurace zobrazovacích průchodů a obvykle zajišťuje vykreslování. Naopak metoda *ProcessEvent()* je

volána pouze v případě, že zobrazovací modul přijme novou událost a zobrazovací průchody v ní mohou například zpracovat událost oznamující změnu z uživatelského rozhraní.

Dále je vhodné, aby každý zobrazovací průchod zaregistroval sdílená data, která potřebuje nebo inicializuje. Kdyby to nebylo provedeno, nemohou být následně kontrolovány závislosti mezi průchody. Podobně si průchod zaregistruje prvky uživatelského rozhraní, které se pak při aktivaci průchodu v zobrazovacím řetězci zobrazí.

3.6 Základní zobrazovací průchody

Základní zobrazovací průchody zajišťují běžné vykreslování scény a většina je v zobrazovacím řetězci implicitně přednastavena. Jedná se o průchody starající se o ořez scény, vytvoření g-bufferu nebo výpočet osvětlení. Dále se mezi ně řadí důležité zobrazení průhledných objektů obvykle zařazené na konec standardního vykreslování.

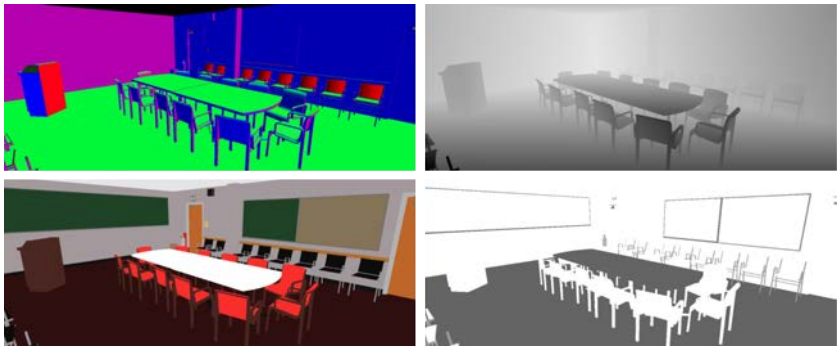
3.6.1 Ořez scény

Základní zobrazovací průchod připravující data pro zobrazení je ořez scény. Správné a efektivní odstranění neviditelné geometrie je nepostradatelná součást každého zobrazovacího systému. Ořezávání scény však nemusí být vždy potřeba, proto je tato část implementace vyčleněna ve zvláštní průchod. Prakticky se jedná o část kódu zajištující ořezávání scény pohledovým jehlanem kamery a inicializování geometrických dat pro další zpracování. Data jsou v zobrazovacím modulu uspořádána v hierarchické struktuře BVH a při ořezu jsou rozdělena do čtyř kategorií podle druhu. První seznam obsahuje běžnou neprůhlednou geometrii, druhý pak geometrii průhlednou, třetí zrcadlové plochy a poslední průhlednou geometrii na základě testu hodnoty průhlednosti (alpha test). Jedná se o speciální případ průhledné geometrie, u které jsou průhledné pixely zcela odstraněny a zbytek zůstává neprůhledný. K tomuto typu geometrie lze v určitých případech přistupovat jako k neprůhlednému a je tedy vhodné vyčlenit ho již při ořezu scény. V případě, že scéna obsahuje průhledné objekty, jsou seřazeny podle vzdálenosti od pozorovatele přímo v tomto průchodu. Podobně jsou viditelné neprůhledné objekty před zobrazováním seřazeny podle ID materiálu pro urychlení zpracování během vykreslování scény.

3.6.2 Vytvoření g-bufferu

Při vykreslování pomocí odloženého stínování je nezbytné vytvořit g-buffer obsahující informace o geometrii ve scéně. Formát a data obsažená v g-bufferu je třeba volit obezřetně. Špatně navržený g-buffer může výrazně zpomalit celou aplikaci a při návrhu je třeba dbát zejména na velikost bufferu. G-buffer musí být obvykle tvořen více texturami, z nichž každá obsahuje rozdílná data, viz obrázek 3.5. Rozložení g-bufferu je znázorněno v tabulce 3.2.

První textura obsahuje normály zakódované sférickým mapováním do dvou hodnot, z nichž každá zabírá 2B podle [Mit09]. Kódování normál tímto způsobem značně zvyšuje přesnost uložených dat vzhledem k velikosti g-bufferu, ačkoliv kódování a dekódování stojí grafickou kartu výpočetní čas. Další textury obsahují informaci o hloubce ve scéně, difuzní a spekulární barvu povrchu. V poslední textuře jsou pak obsaženy dodatečné informace, jako je typ materiálu a hodnota určující lesk (shininess). Ukládaná difuzní barva v sobě již



Obrázek 3.5: Zobrazení komponent g-bufferu. Zleva shora: Dekódovaná normálová, hloubková, difuzní a spekulární textura.

	Byte			
RT	0	1	2	3
0	depth			
1	normal.x		normal.y	
2	diffuse.r	diffuse.g	diffuse.b	diffuse.a
3	specular.r	specular.g	specular.b	free
4	material type	shininess	free	free

Tabulka 3.2: Hloubka, normály, difuzní barva, spekulární barva a typ materiálu spolu s hodnotou lesku v pěti 4B texturách.

musí obsahovat aplikované textury, které by v prostoru obrázku nemohly být namapovány na objekty.

G-buffer však může obsahovat informace jen o neprůhledné geometrii. Kdyby byla průhledná geometrii vykreslena do g-bufferu a prošla by hloubkovým testem, byla by do textury na odpovídající pixely zapsána informace o hloubce, materiálech a normále, ale v tu chvíli by byly ztraceny údaje o neprůhledných objektech nacházejících se za průhlednou geometrií. V takovém případě by nebylo možné správně sloučit barvu pixelu průhledné geometrie s objekty, které překrývá. G-buffer tedy obsahuje pouze neprůhledné objekty a objekty s vlastními shadery v materiálu. Hloubku a normály objektů s vlastními shadery je vhodné zapsat do g-bufferu i když musí být vykreslovány samostatně, protože jsou tato data obvykle využívána v post-procesu různými efekty. Navíc je do g-bufferu přidána geometrie s testem průhlednosti, u které jsou zcela průhledné části odstraněny a může tedy využívat výhod odloženého stínování.

3.6.3 Výpočet osvětlení

Výpočet osvětlení je základní zobrazovací průchod pracující v prostoru obrazu s využitím pouze g-bufferu. Každý pixel obrázku je nutné zpracovávat zvlášť na GPU pomocí fragment shaderu. G-buffer obsahuje informace pouze o viditelné neprůhledné geometrii včetně objektů obsahujících vlastní shadery v materiálu, a proto pro odpovídající zpracování na GPU stačí

vykreslit obdélník přes celou obrazovku. Fragment shader přistupuje k datům z g-bufferu pomocí texturovacích jednotek na základě texturovacích souřadnic interpolovaných mezi vrcholy vykreslovaného obdélníku. Výpočet osvětlení pro jeden světelný zdroj probíhá pomocí Phonogova osvětlovacího modelu obecně definovaného následujícím vztahem [rBSF04]:

$$I_v = I_a + I_d + I_s \tag{3.1}$$

Kde celkové osvětlení I_v se skládá z ambientní složky I_a, difuzní složky I_d a zrcadlové (spekulární) složky I_s.

Ambientní složka určuje množství okolního světla a bývá obvykle shodná pro celou scénu. Ambientní osvětlení zabraňuje zobrazení odvrácených povrchů od světla zcela černě a je vypočítána následovně:

$$I_a = I_A r_a \tag{3.2}$$

Kde I_A je množství okolního světla a r_a je koeficient ambientního odrazu. Během odloženého výpočtu osvětlení však nemáme k dispozici ambientní složku materiálu, proto je tato barva nahrazena složkou difuzní. Množství ambientního světla ve scéně je definováno uživatelem pomocí GUI.

Difuzní složka udává zastoupení difuzní složky v odraženém světle a je určena vztahem:

$$I_d = I_L r_d (\vec{l} \cdot \vec{n}) \tag{3.3}$$

Kde I_L je barevné složení dopadajícího paprsku, r_d koeficient difuzního odrazu, $\vec{l}$ vektor ke světlu a $\vec{n}$ normála povrchu. Difuzní složka nabývá vyšších hodnot, když se směr dopadu světla blíží normále, ale zároveň musí platit $\vec{l} \cdot \vec{n} > 0$, v opačném případě se jedná o povrch odvrácený od světla.

Poslední zrcadlovou složku definuje vztah:

$$I_s = I_L r_s (\vec{v} \cdot \vec{r})^h \tag{3.4}$$

Kde r_s reprezentuje koeficient zrcadlového odrazu, $\vec{v}$ vektor ve směru pozorovatele, $\vec{r}$ odražený paprsek světla od povrchu a h vyjadřuje ostrost zrcadlového odrazu.

Z rovnic je patrné, že pro výpočet je nutné znát pohledový vektor pro všechny pixely obrazu. Pohledové vektory jsou počítány na CPU pro všechny čtyři vrcholy obdélníku kresleného přes obrazovku a následně předány do vertex shaderu. Interpolace pak zajistí správné hodnoty v jednotlivých zpracovávaných fragmentech a následný korektní výpočet osvětlení. Osvětlení je počítáno v prostoru kamery, a tudíž je nezbytné ve stejném prostoru zkonstruovat i pohledové vektory. Jedná se o poměrně jednoduchý proces, ve kterém jsou nejdříve vytvořeny pozice všech čtyř rohových pixelů v obraze: ((0,0), (šířka obrazu,0), (šířka,výška), (0,výška)). Zjistit vektory v souřadnicích kamery je následně možné pomocí funkce OpenGL *gluUnProject()* transformující pixely z prostoru obrazu do prostoru objektu podle zadané modelovací a projekční matice. Tímto způsobem dostaneme vektory, které stačí jen posunout do středu souřadnicového systému a správně natočit pomocí pohledové matice kamery [PF].

Phongův osvětlovací model je aplikován na každý zpracovávaný pixel a výsledná barva zapsána na výstup. V případě, že scéna obsahuje několik světel, musí být výpočet osvětlení proveden pro každé zvlášť a hodnota osvětlení sečtena. Každý druh světla, viz sekce 3.4.2.3, má jiné vlastnosti a výpočet musí být proveden rozdílným způsobem na fragment shaderu. Na příklad bodové nebo reflektorové světlo obsahují navíc parametr určující konstantní, lineární nebo kvadratický útlum světla se vzdáleností od zdroje. Jelikož směrové světlo neobsahuje pozici, je logické, že u něj útlum nebude počítán a naopak u bodového světla je pro správný výpočet nepotřebná informace o směru. Zobrazení scény s různými druhy světel znázorňuje obrázek 3.6.

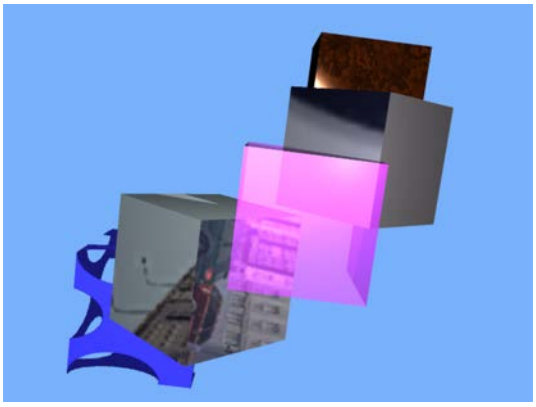
Obrázek 3.6: Implementované druhy světel. Zleva: bodové, směrové a reflektorové.

Jak již bylo zmíněno, objekty obsahující vlastní shadery jsou v g-bufferu také obsaženy. Není tomu tak zcela, protože na GPU nelze aktivovat více jak jeden program současně, a proto geometrie obsahující shader musí být vykreslena samostatně na závěr. G-buffer v textuře s typem materiálu obsahuje informaci, že se jedná o takový objekt a výpočet osvětlení pomocí odloženého stínování je v těchto pixelech přeskočen. Objekty se shadery jsou vykreslovány technikou přímého zobrazování s tím, že je pro hloubkový test použita hloubková textura g-bufferu, což zajistí nejen správné vykreslení viditelných pixelů přes hotový obraz z odloženého stínování, ale také urychlení vykreslování, protože nedochází k velkému přepisování hodnot v obraze.

Jednoduchá scéna obsahující rozdílné typy materiálů včetně objektu s vlastním shaderem je zobrazena na obrázku 3.7.

3.6.4 Průhledné objekty

Průhledné objekty jsou specifická součást scény, se kterou řada algoritmů vůbec nepočítá nebo je pro ni nutné dělat výjimky. Z principu nemohou být obsažené v klasickém g-bufferu a jejich vykreslení tedy probíhá způsobem přímého zobrazování. Právě z důvodů přílišné rozlišnosti průhledných objektů od ostatních byl pro její vykreslení vytvořen speciální zobrazovací průchod. To má své výhody zejména v zařaditelnosti zobrazení průhledné geometrie kamkoliv do vykreslovacího řetězce. Efekty s průhlednou geometrií nepočítající tak mohou bez problémů pracovat před jejím přidáním. Při vykreslování průhledných objektů je nutné dbát na pořadí, aby byly průhledné objekty v zákrytu za jinými viditelné. Toto seřazení na základě vzdálenosti od pozorovatele zajišťuje průchod pro ořez scény, viz sekce 3.6.1.



Obrázek 3.7: Scéna testScene.fhs obsahující pět krychliček, každou s jiným materiálem. Zleva: průhlednost definovaná v textuře, mapování prostředí, celá průhledná krychlička, generování texturovacích souřadnic v prostoru kamery, krychlička s vlastním shaderem v materiálu.

3.7 Doplňkové zobrazovací průchody

Doplňkové zobrazovací průchody, na rozdíl od základních, představují dodatečnou funkcionalitu a k běžnému běhu vykreslování nejsou nezbytné. Jejich aktivace je zcela v režii uživatele, s tím, že vhodnost užití může být závislá na typu scény. Jedná se o průchody přidávající výslednému obrazu na vizuální kvalitě nebo průchody využívané při vývoji nových vlastností, jako je zobrazení struktury BVH nebo komponent g-bufferu. Za efekty dodávající obrazu vyšší kvalitu je zde zobrazení stínů, simulace hloubky ostrosti, ambientní zastínění, odrazivé objekty nebo důležitý antialiasing.

3.7.1 Vytvoření stínů

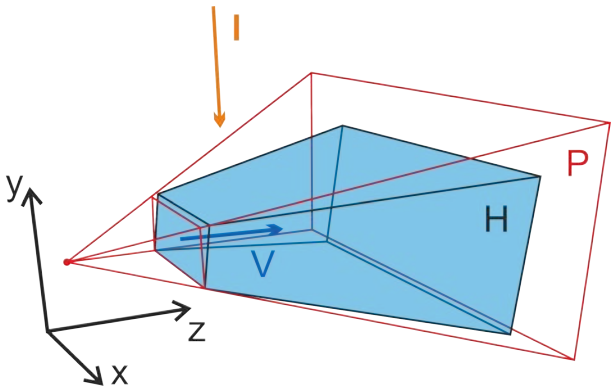
Zobrazovací průchod starající se o vykreslení stínů byl implementován pomocí nejrozšířenější metody stínových map. Konkrétně se jedná o metodu LispSM, viz sekce 2.3.2.3, která byla zvolena pro její dobré vlastnosti z hlediska aliasingu a zanedbatelné navýšení výpočetní náročnosti proti standardním uniformním stínovým mapám. Výhoda stínových map je nezávislost na geometrii ve scéně, jelikož nevyžadují žádné předzpracování. Díky tomu jsou vhodné pro statické i dynamické scény stejně jako pro všechny druhy světel. Algoritmus stínových map pracuje ve dvou fázích, v první jsou generovány stínové mapy v závislosti na typu a pozici světla a druhá část aplikuje stíny do scény.

3.7.1.1 Generování stínových map

Nejprve musí být vypočtena konvexní obálka H uzavírající pohledový objem spolu se všemi objekty potenciálně do něj vrhající stíny. Konstrukce této obálky je implementována podle původního článku [Sta02], postup je podrobně vysvětlen v sekci 2.3.2.1.

Máme-li konvexní obálku H, potřebujeme sestavit perspektivní prostor světla P (light space), který má pohledový vektor paralelní se stínovou mapou. Prostor světla je znázorněn

na obrázku 3.8. Osa y je definována vektorem světla, ale ukazuje v opačném směru. Osa z je kolmá na vektor světla a leží na rovině obsahující pohledový vektor a vektor definovaný směrem světla. Osa x je pak doplněna podle předchozích tak, aby vznikl ortogonální souřadnicový systém. Tímto způsobem vznikne levotočivý systém souřadnic, který je v praxi nutný transformovat do pravotočivého používaného v OpenGL [WSP04].



Obrázek 3.8: Konstrukce prostoru světla P, který zahrnuje konvexní obálku H světelných paprsků, které mohou vrhat stíny do pohledového objemu.

Prostor světla je definován standardně pomocí pohledové a projekční matice, které jsou pak použity při vytváření stínové mapy. Stínová mapa je reprezentována pomocí FBO, kam je ukládána pouze informace o hloubce, tedy o vzdálenosti objektů od světla. Vykreslování objektů do stínové mapy probíhá klasickým přímým zobrazováním, pomocí jednoduchého programu na GPU, který pouze transformuje vrcholy pohledovou, projekční a modelovací maticí pro získání pozice bodů ve stínové mapě. Během vykreslování je zakázán zápis barevné složky a kvůli maximální rychlosti generování stínové mapy je zapnut pouze hloubkový test. Stejným způsobem musí být postup opakován pro všechna světla ve scéně, protože pro každé světlo musí být stínová mapa vytvářena s adekvátní pohledovou a projekční maticí zajištující transformaci do prostoru světla. Podobně jako do g-bufferu je i zde průhledná geometrie na základě masky průhlednosti přidána do stínové mapy. Přidání těchto objektů umožní vrhat stíny jinak nezpracovávaným objektům, jako jsou stromy nebo plot definovaný texturou namapovanou na plném objektu. Z hlediska aliasingu, kromě vhodného využití plochy stínové mapy, samozřejmě hraje roli i její rozlišení. V některých případech dostačující rozlišení se šířkou 1024 pixelů může být v jiných naprosto nevyhovující. Z těchto důvodů průchod zobrazující stínové mapy umožňuje pomocí GUI nastavit rozlišení stínových map od šířky 512 pixelů až po 8192 pixelů.

3.7.1.2 Aplikace stínových map

Ve chvíli, kdy máme vytvořené všechny stínové mapy, je možné na jejich základě aplikovat stíny do scény. Výpočet opět probíhá v prostoru obrázku s využitím dat z g-bufferu. Pro zjištění, zda se pixel nachází ve stínu či nikoliv, je nejdříve rekonstruována pozice bodu ve světových souřadnicích pomocí informace o hloubce pixelu. Tuto pozici jsme pak schopni

transformovat stejnou pohledovou a projekční maticí jako při vytváření stínové mapy a normalizovat. Tímto postupem dostaneme pozici bodu v normalizovaných souřadnicích zařízení, ale pro přístup do stínové mapy uložené ve formě textury je nutné vypočtené souřadnice ještě posunout a vydělit, abychom dostali rozsah od nuly do jedné. Za tímto účelem se používá speciální matice nazvaná $scaleBias$, kterou stačí souřadnice vynásobit a vypadá následovně:

$$scaleBias = \begin{bmatrix} 0.5 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.5 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.5 & 0.0 \\ 0.5 & 0.5 & 0.5 & 1.0 \end{bmatrix} \tag{3.5}$$

Získáme-li správnou pozici zpracovávaného bodu ve stínové mapě, je jeho hloubka ze stínové mapy porovnána s hloubkou v obraze. V situaci, kdy je hloubka ve stínové mapě menší, bod je zastíněn. V praxi se k hodnotám stínové mapy ještě přidává malý posun, odstraňující samozastínění. Posun je přímo závislý na měřítku scény a je tedy vhodné umožnit jeho nastavení uživatelem přímo z GUI.

Pozici světla je samozřejmě opět nutné porovnávat ve všech vytvořených stínových mapách, ale co dělat v případě, že je pixel obrazu zastíněný? Stínící objekt obvykle stíní dopadající paprsky z jednoho nebo více světel. Pokud by byla hodnota jasu konstantně snížena, mohla by nastat situace, kdy by byl vytvořen stín od světla, které sice není zastíněné, ale prakticky neovlivňuje barvu pixelu v obraze. Tato situace může nastat například, pokud je nastaven útlum světla a světlo se nachází příliš daleko, takže výsledný pixel neovlivní nebo v případě reflektorového světla pixel nespadá do světelného kuželu. Jedná se o zdánlivě jednoduchý problém, ale v konceptu zcela samostatného průchodu pracujícího v post-procesu není řešení tak triviální. Jedna z možností je aproximovat původní osvětlení zjednodušeným výpočtem Phongova osvětlovacího modelu, ale to by mohlo způsobit poměrně značný výkonnostní problém. Proto jsme se rozhodli v případě zobrazovacího průchodu zajištujícího výpočet stínových map udělat výjimku a sloučit jej s výpočtem osvětlení technikou odloženého stínování. V praxi to pak znamená, že pokud je aktivní průchod pro stíny, může být ze zobrazovacího řetězce odstraněn průchod počítající osvětlení, protože ho průchod s výpočtem stínů nahradí. Tento přístup sice integruje dva zobrazovací průchody do jednoho, ale díky poměrně jednoduché implementaci se nestává kód příliš komplexním a nepřehledným. Díky spojení dvou průchodů není zbytečně počítáno osvětlení pro fragmenty obrazu, ze kterých by následně bylo odstraněno, protože se nacházejí ve stínu.

Další problém představují objekty s vlastními shadery, u kterých nemáme k dispozici žádnou informaci o průběhu osvětlení. Jelikož vlastní shadery jsou přidávány do materiálů pouze v případě netypického požadavku na vlastnosti povrchu, je poměrně pracné jakkoli osvětlení odhadovat a zobrazit tak správné stíny, ale nezobrazit na nich stíny žádné by bylo naprosto nevyhovující. Pokud tedy neznáme absolutně nic o typu osvětlení, lze situaci vyřešit tím nejpřirozenějším způsobem. Nejprve je vykreslena geometrie s vlastními shadery technikou přímého zobrazování. Ačkoli by se tak mohlo zdát, nedochází v této fázi k příliš velkému přepisování hodnot ve srovnání s technikou odloženého stínování, protože je aktivován hloubkový test proti hloubkové mapě g-bufferu, ve které je geometrie se shadery obsažena. Následně při standardním stínování a aplikaci stínových map je fragmentům vykresleným v předchozím kroku snížena hodnota osvětlení na základě výpočtu difusní složky

Phongova osvětlovacího modelu. Phongův osvětlovací model respektuje polohu světla vzhledem ke směru normály a je velká pravděpodobnost, že výsledný rozdíl odečteného osvětlení se bude dostatečně blížit správné hodnotě. Zobrazení stínů v několika scénách je znázorněno na obrázku 3.9. V této práci byly implementovány stínové mapy pro dva typy nejpoužívanějších světel a to reflektorový a směrový zdroj světla.

Obrázek 3.9: Zobrazení stínů v malé scéně. Zleva: scéna bez stínů, metoda LispSM se směrovým typem světla a uniformní stínové mapy s reflektorovým světlem.

3.7.2 Ambientní zastínění v prostoru obrazu

Ambientní zastínění hrubě aproximuje šíření nepřímého osvětlení a dodává tak scéně reálnější vzhled, podrobněji viz sekce 2.4.2. SSAO, jak již název vypovídá, pracuje v prostoru obrazu a nezávisí na komplexnosti scény. Metoda pracuje podobně jako vytvoření stínové mapy ve dvou krocích. V první fázi je vytvořena mapa zastínění v prostoru obrazu, která je posléze ve druhé fázi aplikována na zpracovávaný obraz.

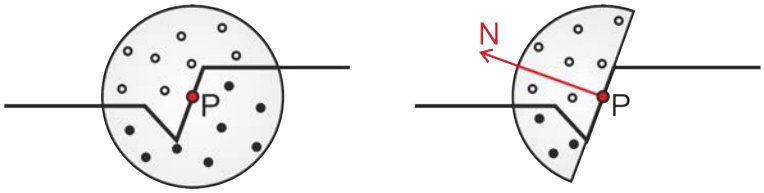
Mapa zastínění určuje v každém bodě obrazu míru zastínění okolím. Pixely nacházející se uvnitř složité geometrie jsou více zastíněny než pixely, které žádná geometrie neobklopuje, protože je příchozí světlo více tlumeno okolními objekty. V prostoru obrazu je zastínění vyhodnocováno na základě hloubkového testu. Pro každý pixel jsou vygenerovány vzorky v okolí daném nastaveným poloměrem. Tyto vzorky musí být následně převedeny do prostoru obrázku a jejich hloubka porovnána s hloubkovou mapou scény. Je-li hloubka vygenerovaného vzorku větší než hodnota v hloubkové mapě, vzorek je zastíněn. Vyhodnocením všech vygenerovaných vzorků je získána celková hodnota zastínění bodu obrazu. Tento postup je opakován pro každý pixel a postupně vznikne mapa zastínění.

3.7.2.1 Generování vzorků

Generování vzorků je nejdůležitější část, která významně ovlivňuje kvalitu a rychlost výpočtu ambientního zastínění. Existuje několik přístupů k této problematice, ale ve všech hraje roli poloměr koule, ve které jsou vzorky generovány. Poloměr je plně závislý na měřítku scény a určuje maximální vzdálenost objektů ovlivňujících zastínění.

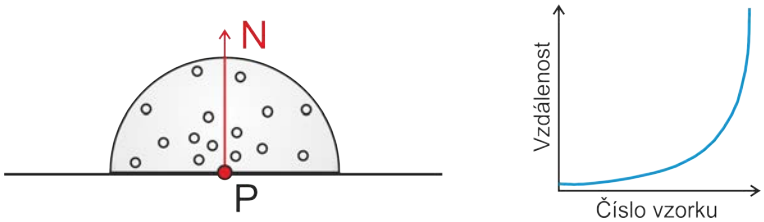
Základní přístup vytváří vzorky rovnoměrně v celém prostoru koule ohraničující daný bod v prostoru, viz obrázek 3.10. Hlavním nedostatkem této metody je částečně zastíněná veškerá geometrie ve scéně, protože i na rovinných objektech leží část vzorů pod povrchem a je vyhodnocena jako zastíněná. Všechny tyto vzorky na odvrácené hemisféře jsou navíc generovány zcela zbytečně a plýtvají výkonem GPU.

Kvůli výše zmíněným nedostatkům byla zvolena nejběžnější metoda generování paprsků na hemisféře natočené podle normály povrchu, viz obrázek 3.10. Získat informaci o normálách může být problematické, pokud SSAO pracuje v post-procesu po přímém zobrazování, ale v případě odloženého stínování s uloženou normálovou mapou se jedná o nejvhodnější metodu s minimálním navýšením náročnosti výpočtu.



Obrázek 3.10: Generování vzorků v okolí bodu. Vzorky rozložené v celém okolí bodu (vlevo). Výhodněji generované vzorky na hemisféře (vpravo).

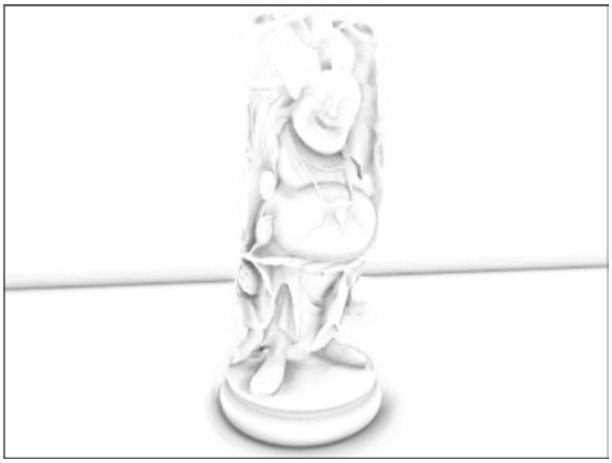
Dalším aspektem ovlivňujícím zastínění je rozložení vzorků uvnitř hemisféry a způsob generování. Uniformní rozložení vzorků může být nahrazeno Gaussovým rozložením kolem normály lépe respektujícím fyzikální vlastnosti světla, kdy je přikládán větší důraz směrům blížícím se normále, které více přispívají k celkovému osvětlení bodu [Mir10]. Každý z těchto směrů je generován náhodně, a pak uniformním způsobem vzorkován v několika vzdálenostech pro vhodnější pokrytí hemisféry. Jiný přístup naopak generuje vzorky náhodně v celé hemisféře, ale upravuje jejich množství vzhledem ke vzdálenosti od zpracovávaného bodu, viz obrázek 3.11. Více vzorků je tedy generováno v menší vzdálenosti od středu, čímž dojde k vhodnějšímu pokrytí bližšího okolí, které teoreticky více přispívá zastínění [Joh11].



Obrázek 3.11: Distribuce vzorků vzhledem ke vzdálenosti od zpracovávaného bodu P.

V naší implementaci byly testovány obě metody generování vzorků. První jednoduchá metoda podle [Mir10] s Gaussovým rozložením podle normály s uniformním vzorkováním vzdálenosti až po nastavený poloměr u které byly vzorky generovány pro každý pixel obrazu náhodně na GPU. Tato metoda se nezdá být vhodná vzhledem k vyšší zátěži GPU a poměrně velkému šumu vznikajícímu zcela náhodným generováním. Z těchto důvodů byla implementována další pokročilejší metoda původně pocházející ze hry Crysis. Nejprve je na CPU náhodně vygenerována sada vzorků v jednotkové hemisféře s větší hustotou okolo středu. Dále je vytvořena malá textura s náhodnými hodnotami (šum), která je při zpracování periodicky opakována přes celý obraz. V každém bodě obrazu během vytváření mapy zastínění je pak náhodně vygenerovaná sada vzorků natočena podle normály a pootočena kolem své osy o hodnotu šumu. Textura se šumem má rozměr 4 x 4 pixelů a díky opako-

vání se v mapě zastínění objeví šum s vysokou frekvencí, který je snadné odstranit filtrem v následující fázi zpracování. Nefiltrovaná mapa zastínění je zobrazena na obrázku 3.12.



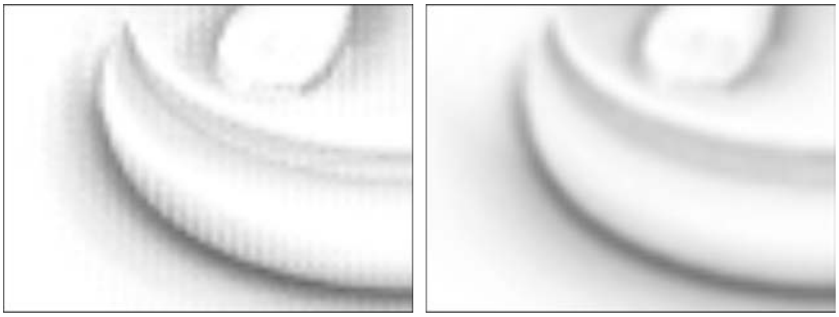
Obrázek 3.12: Vytvořená mapa zastínění metodou SSAO.

3.7.2.2 Filtrování mapy zastínění

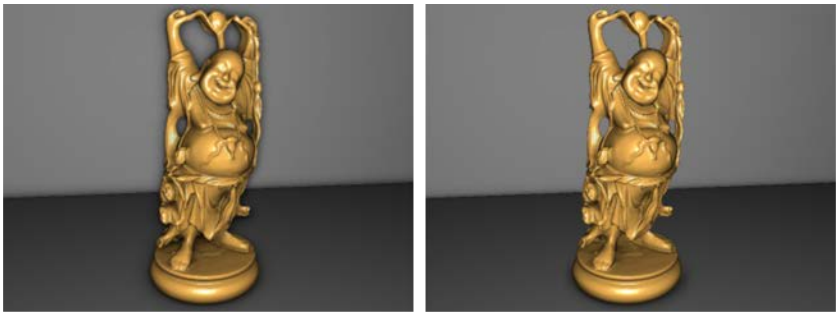
Před aplikací stínové mapy na scénu je nezbytné odstranění šumu způsobené nedostatečným počtem vzorků, který je nutné držet co nejnižší kvůli výpočetním nárokům. Za tímto účelem je mapa zastínění filtrována. Jako filtrovací funkce je zvolen průměr okolních hodnot s filtrovacím jádrem čtyři pixely. Jedná se o separabilní filtr, který je na GPU vhodné implementovat ve dvou průchodech, kdy nejprve je obraz filtrován vertikálně a potom horizontálně nebo naopak. Výsledek pak vypadá naprosto shodně jako při filtrování celého okolí najednou, ale zpracování je výrazně rychlejší. U metody s náhodným generováním vzorků pro každý pixel byla mapa zastínění velmi zašuměná a bylo nezbytné zvolit velké jádro filtru. Původní autor dokonce aplikoval dvojí filtrování. Během filtrování velkým jádrem pak dochází k rozmazání mapy zastínění v oblasti hran a je nutná jejich detekce. U před-generovaných vzorků s opakujícím se pootočením stačí filtrovat pouze velmi malé okolí. Menší filtrovací jádro této metody ušetří výpočetní čas a pořád je naprosto dostatečné k výborným výsledkům. Rozdíl mezi filtrovanou a nefiltrovanou mapou zastínění je znázorněn na obrázku 3.13.

3.7.2.3 Aplikace mapy zastínění

Máme-li vytvořenou mapu zastínění a barevný obraz z předchozích průchodů, je možné zastínění aplikovat. Jedná se o poslední průchod, ve kterém je opět vykreslen obdélník přes celou obrazovku a na fragment shaderu sestaven výsledný obraz. Mapa zastínění obsahuje tmavé pixely v zastíněných místech a k její aplikaci stačí pouhé vynásobení barevného pixelu v původním obraze hodnotou zastínění, viz obrázek 3.14.



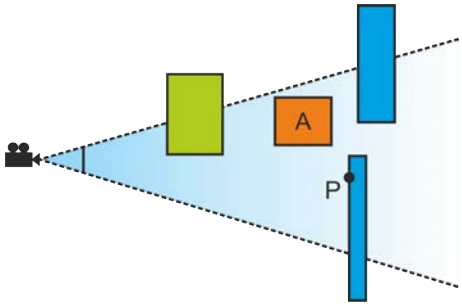
Obrázek 3.13: Zvětšený detail nefiltrované mapy zastínění (vlevo). Stejný detail po aplikaci filtru o šířce čtyři pixely (vpravo).



Obrázek 3.14: Objekt v popředí nesprávně zastiňuje vzdálenou plochu (vlevo). Aplikovaná mapa zastínění ve scéně s kontrolou vzdálenosti (vpravo).

3.7.2.4 Nedostatky metody SSAO

Výpočet zastínění v prostoru obrázku se potýká s několika problémy. Díky porovnávání hloubky v obrazovém prostoru je ztracena informace o objektech z pohledu pozorovatele zastíněných. Neviditelné objekty pak nepřispívají k zastínění tak, jak by měly a vzniká chyba ve výpočtu, viz obrázek 3.15. Ze stejné situace zpracování pouze hloubkové mapy vyplývá podobný problém zastínění bodu vzdálenými objekty. V případě, že je testována hloubka vzorku vůči hodnotě v hloubkové mapě, je vyhodnoceno zastínění i v případě, kdy je stínící objekt velmi vzdálen a neměl by k zastínění přispívat. Efekt se typicky projevuje zastiňující aurou kolem objektů, viz obrázek 3.14. Nežádoucí zastínění vzdálených objektů lze poměrně snadno odstranit pomocí testování rozdílu hodnot při porovnávání hloubek. Liší-li se vzdálenost mezi body více než je nastavený poloměr okolí braného v úvahu při zastínění, vyhodnotí se vzorek jako nezastiňující.



Obrázek 3.15: Z pohledu kamery neviditelný objekt A by měl přispívat k celkovému zastínění zpracovávaného bodu P.

3.7.3 Antialiasing

Minimalizace aliasingu obrazu je nezbytnou součástí každého pokročilého zobrazovacího řetězce. V této práci byla implementována moderní pokročilá metoda SRAA, vyvinutá speciálně pro účely her a jiných aplikací pracujících v reálném čase, viz sekce 2.5.4. Jedná se o antialiasing určený pro použití v post-procesu v zobrazovacích řetězcích založených na odloženém stínování a je tedy velmi vhodný pro implementaci do samostatného zobrazovacího průchodu.

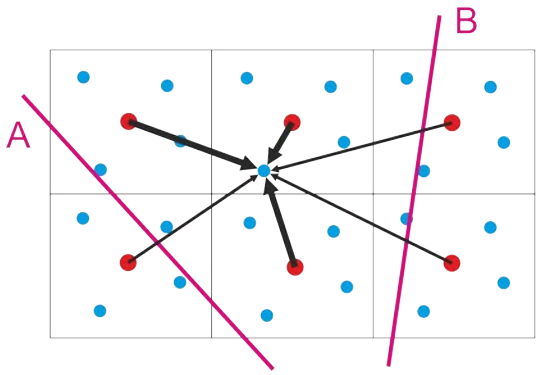
SRAA je založený na faktu, že se barva v prostoru obrazu často mění pomaleji než geometrie a vytváří hloubkový a normálový buffer ve vyšším rozlišení než je samotný obraz. Klasický supersampling zobrazuje celou scénu ve vyšším rozlišení a následně interpoluje hodnoty pro pixely obrazovky. V takovém případě musí být počítáno obvykle náročné osvětlení v každém pixelu, což představuje poměrně velkou výpočetní zátěž. Aby v případě odloženého stínování nemusel být vytvářen celý g-buffer ve vysokém rozlišení, pracuje SRAA pouze s informací o geometrii ve scéně. SRAA v první fázi získá informace o hloubce a normálách ve vyšším rozlišení a v druhém kroku jsou detekovány hrany s následnou filtrací obrazu. Výsledkem celého procesu je obraz v rozlišení obrazovky s výrazně redukovaným aliasingem vhodný pro další zpracování v post-procesu.

3.7.3.1 Generování normál a hloubky

SRAA hledá diskontinuity v obraze podle geometrie ve scéně. Pro přesné vyhlazení je běžné rozlišení g-bufferu nedostačující a je nezbytné vygenerovat data v dostatečném rozlišení pro další zpracování. Potřebujeme tedy normálovou a hloubkovou texturu obsahující několik vzorků pro každý pixel obrazu. Jako nejvhodnější vzhledem k přijatelné rychlosti a kvalitě vyhlazení se jeví generovat čtyři dodatečné vzorky na jeden pixel. Výsledek je pak srovnatelný se čtyřikrát počítaným osvětlením pro všechny body obrazu [CML11]. Pro uložení dodatečných dat byl využit klasický multisampling pro normálovou i hloubkovou mapu. Na začátku zobrazovacího průchodu jsou s pomocí maximálně jednoduchých shaderů vytvořeny oba buffery, které pak mohou být použity pro fázi vyhlazování.

3.7.3.2 Filtrování

Při rekonstrukci každého bodu obrazu se vychází z hodnot pixelů v nejbližším okolí. Poloměr okolí je pevně daný a v našem případě se jedná o zahrnutí sousedních pixelů. Zvětšením poloměru může teoreticky dojít ke zlepšení kvality rekonstrukce, ale nejhorší případ chybné rekonstrukce se zvyšuje a navíc je načítáno více dat z paměti. Během výpočtu z nejbližšího okolí je maximální chyba rekonstrukce jeden pixel. Metoda využívá bilaterální filtr zachovávající ostré hrany. Nejprve jsou spočítány váhy pro každý vzorek pixelu z okolních bodů obrazu postupem znázorněným na obrázku 3.16. Váhy jsou posléze zkombinovány a použity pro výpočet výsledné barvy pixelu.



Obrázek 3.16: Rekonstrukce jednoho vzorku pomocí SSRA. Červená barva v každém pixelu určuje vzorek, ve kterém je počítáno osvětlení, modrá pak dodatečné čtyři vzorky s informací o normále a pozici. Přímky A a B označují hrany ve scéně. Pro každý geometrický vzorek jsou počítány váhy z okolních vzorků s vypočítaným osvětlením. Vzorky nacházející se za hranou bilaterální filtr ohodnotí malou váhou a naopak vzorky bez velké změny v pozici nebo normále povrchu výrazně přispívají výsledné barvě pixelu.

Váha vzorku je určena z rozdílu v normále povrchu a v pozici ve scéně. Rozdíl v natočení normály d_n a v pozici d_p je počítán podle vztahů:

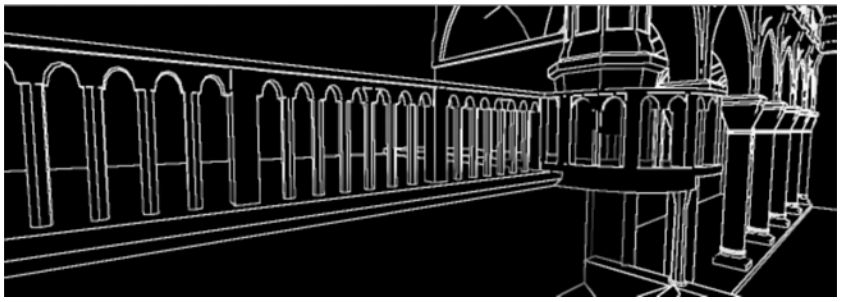
$$d_n = 1 - (\vec{n_s} \cdot \vec{n_t}) \tag{3.6}$$

$$d_p = \sigma_p |p_s - p_t| \tag{3.7}$$

Kde $\vec{n_s}$ je normála zdrojového vzorku, $\vec{n_t}$ normála cílového vzorku, p_s pozice zdrojového vzorku a p_t pozice cílového. Hodnota σ_p pak určuje měřítko hloubky závislé na hloubkovém rozsahu scény. Celková váha vzorku w je pak počítána podle následujícího vzorce, kde s určuje rychlost snížení váhy při nalezeném rozdílu.

$$w = e^{-s*max(d_n,d_p)} \tag{3.8}$$

Vypočítáním váhy okolních pixelů pro každý vzorek zpracovávaného bodu obrazu získáme informaci o případných hranách v okolí. Podle váhy jsme pak schopni určit míru ovlivnění barvy v obraze okolními pixely a korektně zrekonstruovat obraz. Obrázek 3.17 zobrazuje



Obrázek 3.17: Detekce hran ve scéně. Bílá barva označuje místa, kde dochází k filtrování, černě jsou pak označeny uniformní oblasti.

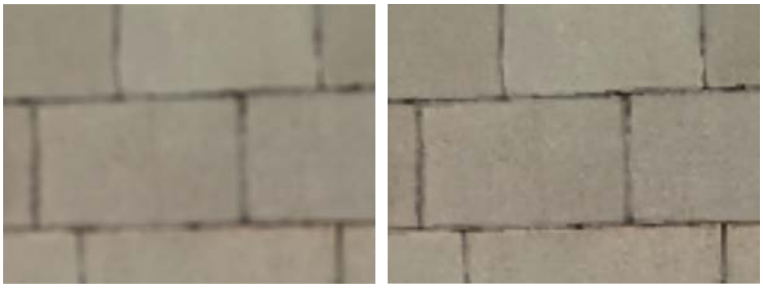
místa, kde je detekována diskontinuita v obraze a dochází k zamezení příspěvku barvy od některých pixelů v okolí, čímž dochází k vyhlazení.

Výše zmíněný filtr je možné zjednodušit a urychlit celou fázi rekonstrukce vypuštěním normálové složky z výpočtu. Zahrnutím pouze hloubkového bufferu je snížen počet čtení a zápisu dat na polovinu, z čehož vyplývá značná úspora času. V našem případě vůbec nemusí dojít ke generování normálové mapy ve vysokém rozlišení, ale je uložena pouze pozice vzorků ve scéně. Aliasing je nejvíce vnímaný na obrysech objektů, a proto se zdá být kvalita poměrně vyhovující. Zejména pak v dynamických scénách s velkým množstvím textur není rozdíl prakticky postřehnutelný. Vyhlazení metodou SRAA a porovnání jednotlivých přístupů je znázorněno na obrázku 3.18.

Obrázek 3.18: Vyhlazení hran pomocí SRAA. Bez vyhlazení (vlevo). Vyhlazeno pouze podle hloubkové mapy (uprostřed). Vyhlazeno pomocí hloubkové i normálové mapy (vpravo).

3.7.3.3 Redukce rozmazání

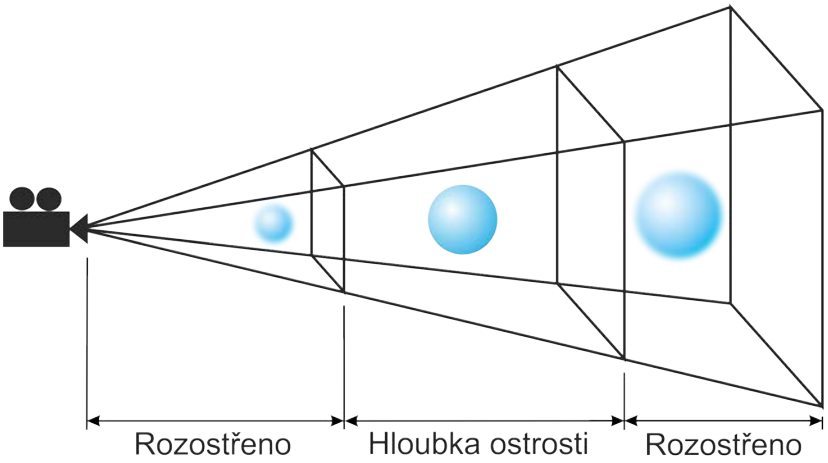
Z principu filtru popsaného výše je patrné, že v případě stejného okolí zpracovávaného pixelu dochází k nechtěnému rozmazání. Jedná se o obecný problém všech filtrů vycházejících z okolí zpracovávaného bodu. V takové situaci všechny okolní pixely mají stejnou váhu a rovným dílem přispívají k výsledné barvě. Tento nežádoucí jev je nejvíce patrný na texturách. Detekováním uniformních oblastí jsme schopni rozmazání eliminovat, viz obrázek 3.19. Jestliže není při běhu shaderu zpracovávajícího jeden fragment detekována změna v normále nebo hloubce proti okolním pixelům, je oblast prohlášena za uniformní a není na ní aplikován filtr.



Obrázek 3.19: Rozmazaná textura filtrem SRAA na uniformní části scény (vlevo). Zabránění filtrování stejné uniformní oblasti (vpravo).

3.7.4 Simulace hloubky ostrosti

Hloubka ostrosti (DOF) je místo, ve kterém se objekty jeví jako ostré, jestliže se objekty nachází před nebo za touto oblastí, dochází k rozostření, viz obrázek 3.20. Tento fakt je daný fyzikálními vlastnostmi průchodu světla přes čočku. V počítačové grafice však scénu nesledujeme přes čočku a nejsme tak limitováni fyzikálními vlastnostmi, ale jsou případy, kdy je vhodné obrazu dodat více realističnosti a simulovat reálný svět natočený běžnou kamerou. Ve hrách je například hloubka ostrosti využívána pro přitažení pozornosti hráče, podobně jako ve filmu během animačních sekvencí. Hloubka ostrosti byla implementována podle [Nat12] s počítaným rozostřením na základě parametrů kamery.



Obrázek 3.20: Objekty uvnitř hloubky ostrosti se z pohledu pozorovatele jeví jako ostré na rozdíl od objektů před nebo za touto oblastí.

3.7.4.1 Rozostření obrazu

Míru rozostření lze implementovat dvěma způsoby. První přístup předává počáteční a koncové hloubky, ve kterých jsou objekty ostré, do shaderu, který data zpracuje a oblasti mimo

hloubku ostrosti přiměřeně rozostří. Tato metoda je poměrně jednoduchá, ale při animaci je velmi složité správně reagovat na nastavení kamery při pohybu ve scéně. Další možnost obsahuje sice složitější výpočet, ale míra rozostření je odvozená z reálných parametrů kamery a je tedy vhodnější pro animaci typu přibližování a podobně.

Rozostření je aplikováno podobně jako v případě mapy zastínění, viz sekce 2.4.2 separabilním filtrem ve dvou průchodech (vertikálním a horizontálním). Na rozdíl od mapy zastínění metoda DOF potřebuje filtrovat obraz nerovnoměrně podle hloubky objektů ve scéně vzhledem k nastavení parametrů kamery. Míra rozostření je určena poloměrem filtru vypočítaným z následujícího vztahu:

$$b = \frac{f m_s}{N} \frac{x_d}{D_s \pm x_d} \tag{3.9}$$

Kde f je ohnisková vzdálenost, $m_s = \frac{f}{D_s - f}$, N je ohnisková vzdálenost vydělená poloměrem čočky, x_d je vzdálenost fragmentu od vzdálenosti plně zostřeného objektu D_s. Z rovnice je získán výsledný poloměr rozostření b pomocí $D_s - x_d$, jestliže se objekt nachází v popředí hloubky ostrosti nebo $D_s + x_d$ nachází-li se za hloubkou ostrosti. V dalším kroku je míra rozostření převedena na poloměr filtru v pixelech podle velikosti snímače pomocí vzorce:

$$r = b \frac{\sqrt{w_s^2 h_s^2}}{\sqrt{w_f^2 h_f^2}} \tag{3.10}$$

Kde w_s a h_s označují velikost obrazu v pixelech a podobně w_f a h_f velikost snímače v milimetrech. Výsledek r je pak konečná velikost filtru použitého k rozostření.

Z hlediska nasazení efektu hloubky ostrosti v aplikacích pracujících v reálném čase musí být podobně jako předchozí efekty maximálně optimalizován, aby zbytečně nenavyšoval čas potřebný k výpočtu jednoho snímku. Kromě již zmíněného separabilního filtru pro navýšení rychlosti je důležité množství zpracovávaných dat při rozostření. Máme-li obraz ve vysokém rozlišení, musíme pro stejnou míru rozmazání zahrnout do výpočtu větší okolí bodů než v situaci s nižším rozlišením. Této vlastnosti lze využít a obraz filtrovat v mnohem nižším rozlišení a zajistit tak výrazné ušetření načítání dat z paměti. Míru snížení rozlišení je nutné volit rozumně s ohledem na kvalitu zobrazení v závislosti na navýšení výkonu. V naší implementaci jsme použili pro rozostření FBO s poměrem 0,75 šířky a výšky proti původní velikosti. Na první pohled velmi malé snížení celkového rozlišení umožní použít výrazně menší filtrovací jádro se stejně razantním rozostřením a značně tak ušetřit výpočetní čas na GPU. Rovněž je vhodné výpočetní nároky a míru rozostření řídit parametrem maximální možné šířky filtru.

Využití jednoduchého, ale rychlého filtru může způsobovat zdánlivé rozostření objektů, které by měly být plně zaostřeny. Stává se tak v případě velkého rozdílu hloubky sousedních objektů, protože i když zaostřený objekt není filtrován, dochází k přejímání barvy sousedními pixely, které jsou více filtrovány a jejich okolí zasahuje do ostrého objektu. Nežádoucí jev se zvyšuje s mírou rozostření a je patrný zejména v jednoduchých scénách bez textur.

GLSL program pro rozostření

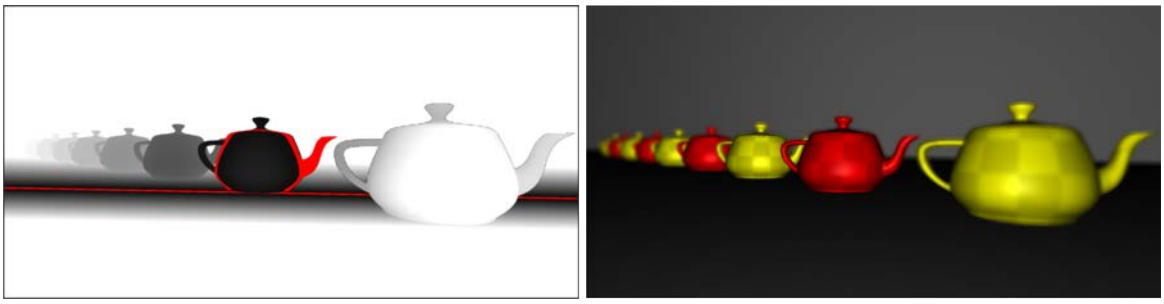
Fragment shader zajištující horizontální rozostření je typická ukázka výpočtu efektu v prostoru obrazu. Během vykreslení obdélníku přes celou obrazovku dojde ke zpracování čtyř

vrcholů vertex shaderem, přičemž jsou data během rasterizace interpolována pro výpočet na fragment shaderu. Z důvodu přehlednosti kód neobsahuje uniformní proměnné, do kterých jsou předávána data z CPU, ale jen vstupní texturovací souřadnice z vertex shaderu a výstupní textury pro uložení rozmazaného obrazu a koeficientu rozostření. V hlavní metodě *main()* je nejprve načtena informace o hloubce fragmentu, která musí být převedena do prostoru kamery, protože OpenGL neukládá hloubku v lineárním stavu. Následuje výpočet míry rozostření s uložením této hodnoty pro zpracování v dalších částech efektu DOF. Konec ukázkového kódu spočítá barvu výsledného pixelu průměrem z okolních pixelů daných velikostí rozostření.

```
//texturovací souřadnice z vertex shaderu
smooth in vec2 VS_TexCoord;

//výstupní textury
layout(location = 0) out vec4 colorOut;
layout(location = 1) out vec4 blurAmountOut;

void main()
{
    //načtení hloubky ve scéně
    float depthMapDepth = texture(depthMap, VS_TexCoord).x;
    float depthReal = planes.y / (planes.x + depthMapDepth);

    //výpočet míry rozostření
    float xd = abs(depthReal - focusDistance);
    float xdd = (depthReal < focusDistance) ?
        (focusDistance - xd) : (focusDistance + xd);
    float b = blurCoefficient * (xd / xdd);
    float blurAmount = clamp( b * PPM, 1, maxBlurRadius);

    //zápis dat pro následující průchody
    blurAmountOut.x = blurAmount;

    //horizontální rozostření obrazu
    const vec2 texelOffset = vec2(texelSize.x, 0.0);
    colorOut = vec4(0.0);
    float halfBlur = blurAmount * 0.5;
    for(int i = 0; i < blurAmount; ++i)
    {
        vec2 vOffset = VS_TexCoord + (texelOffset * (i - halfBlur));
        colorOut += texture(colorMap, vOffset);
    }

    //zápis výsledné barvy
    colorOut = colorOut / ceil(blurAmount);
}
```

3.7.4.2 Finální efekt hloubky ostrosti

V první fázi byla vytvořena rozostřená textura s nižším rozlišením na základě nastavení parametrů kamery. Rozmazaný obraz pak může být spojen s původním v plném rozlišení podle míry rozostření. Během zpracování každého fragmentu shaderem je barva interpolována na základě hodnoty rozmazání mezi barvou originálního pixelu a barvou z rozmazané textury. Ve výsledku je pak textura s nižším rozlišením použita jen u rozostřených částí. Plně zostřené objekty jsou nezměněny a nedochází k žádnému zhoršení kvality obrazu. Správné nastavení parametrů může být obtížné, proto byl implementován speciální mód zobrazující míru rozostření scény a vzdálenost plného zaostření. Mód určený pro nastavení a scéna s aplikovanou hloubkou ostrostí je znázorněna na obrázku 3.21.



Obrázek 3.21: Nastavení efektu DOF s červeně zobrazenou vzdáleností plného zaostření. Bílá barva označuje plně rozostřené oblasti a černá naopak ostré (vlevo). Aplikovaná hloubka ostrosti na scénu, kde dochází k rozmazání blízkých i vzdálených objektů podle nastavení parametrů kamery (vpravo).

3.7.5 Zrcadlové plochy

Odrazy jsou ve VRUTu velmi důležitá vlastnost vzhledem k zobrazení zrcátek u modelů automobilů ve firmě Škoda Auto. Pro implementaci byla zvolena metoda zobrazení odrazů využívající stencil buffer. Metoda je vhodná pouze pro zobrazení rovinných tvarů a případná složitější geometrie musí být rozdělena na oddělené plochy, každá představující samostatný objekt. Směr odrazu zrcadlového objektu je dán jeho normálou povrchu. V případě, že se kamera nachází za objektem, je normála otočena, aby byl vytvořen korektní odraz i na opačné straně. Z normály a bodu povrchu je pak vypočítána matice použitá při generování odrazu.

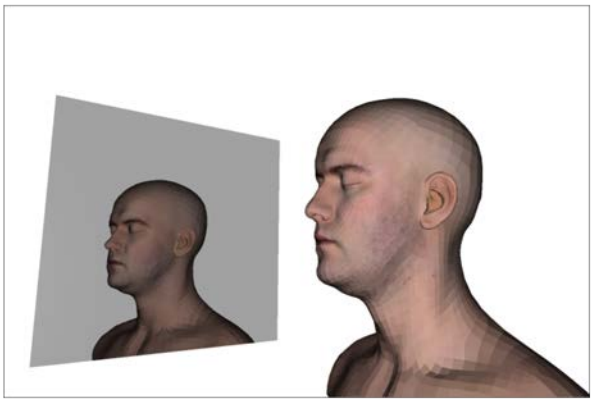
3.7.5.1 Vytvoření odrazu

Vykreslení probíhá do zvláštního FBO se spojeným hloubkovým a stencil bufferem v poměru 24/8 bitů. Do FBO je nejprve zapsána hloubka všech zrcátek se zapnutým hloubkovým testem, v dalším kroku je nastaven stencil test pro zápis jedniček na místa, kde je vyhodnocen hloubkový test kladně. Po nastavení stencil testu je do stencil bufferu vykreslena geometrie zpracovávaného zrcátka. Díky předchozímu zápisu všech zrcátek do hloubkového bufferu nedochází k nesprávnému překrytí případného zrcátka umístěného blíže k pozorovateli a

jedničky jsou zapsány pouze v místě, kde projde hloubkový test. Obsahuje-li stencil buffer informaci o pozici zrcátka, můžeme vykreslit samotný odraz. K tomu je zapotřebí přenastavit stencil test, aby docházelo k zápisu hodnot pouze na místa obsahujících jedničku, tedy v místě zrcátka. Kdybychom v tuto chvíli začali vykreslovat odražený obraz pomocí transformační matice zrcátka, byla by geometrie nacházející se za zrcadlovou plochou rovněž zobrazena. Abychom zamezili tomuto efektu, je před samotným vykreslováním důležité nastavit ořezovou rovinu shodnou s rovinou zrcátka. Veškerá geometrie za zrcátkem bude odstraněná a scéna může být vykreslena. Zobrazování odražené scény probíhá přímým vykreslováním pouze s ořezem pohledovým jehlanem speciálně zkonstruovaným podle tvaru, velikosti a natočení zrcátka. Stejným způsobem je samostatně zpracováno každé zrcátko a postupně vzniká FBO obsahující správné odrazy scény.

3.7.5.2 Materiál a dokončení

Z předchozího postupu máme FBO s odrazy na místě všech zrcátek. Takovéto odrazy však simulují stoprocentně odrazivý povrch a ignorují samotný materiál objektu. Naše implementace umožňuje vytvářet nejen stoprocentně odrazivé plochy, ale i odrazivost částečnou. V případě částečné odrazivosti je barva materiálu kombinována s odraženým obrazem. Sloučení barev je provedeno pomocí průhlednosti, kdy odrazivé objekty jsou znovu vykresleny do FBO s nastavenou průhledností podle míry odrazivosti. Zapneme-li průhlednost v OpenGL, grafická karta se o sloučení postará sama. Před vykreslením objektů s materiály musí být smazána hloubková paměť obsahující data z vytváření odrazů. Hloubka zrcátek je pak znovu zapsána, aby nemohlo dojít k přepsání následujícím vykreslováním scény z pohledu kamery. Jestliže je ve scéně více odrazivých ploch, musí být zápis hloubky a materiálu oddělen kvůli správnému zápisu barvy materiálu vždy pouze na viditelné pixely obrazu. Obsahuje-li scéna pouze jednu zrcadlovou plochu, nemusela by být hloubka znovu zapisována, protože výpočet osvětlení v samostatném průchodu využívá informaci o hloubce uložené v g-bufferu, ve kterém se odrazivá geometrie nachází. Je tedy zajištěno, aby nedošlo k nesprávnému přepsání zrcátka jinými objekty, ale jelikož by měly být průchody psány maximálně nezávisle, je tento krok nezbytný. Výsledný efekt je znázorněn na obrázku 3.22.

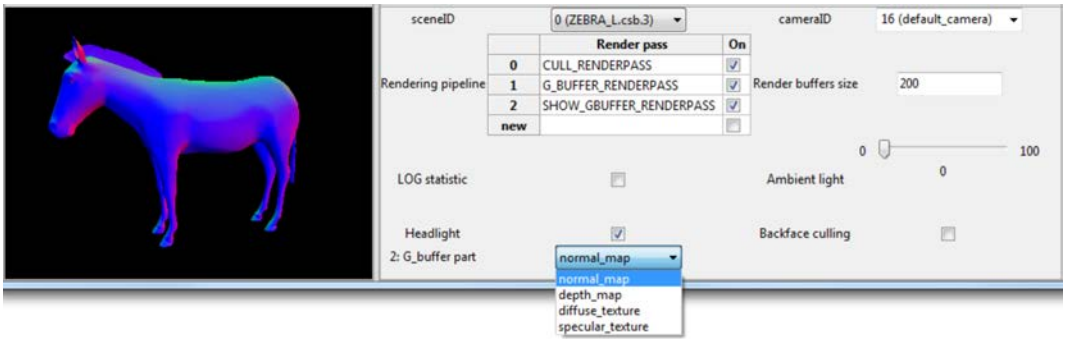


Obrázek 3.22: Jednoduchá testovací scéna s objektem simulujícím zrcadlo.

3.7.6 Obsah g-bufferu

Jelikož se jedná o zobrazovací řetězec založený na odloženém stínování, je jeho nedílnou součástí g-buffer, viz sekce zabývající se jeho vytvořením 3.6.2. Pro pohodlný vývoj a správu aplikace je v některých případech vhodné ověřit obsah g-bufferu. Za tímto účelem byl vyvinut samostatný průchod vykreslující jednotlivé komponenty g-bufferu přímo na obrazovku. Vývojář si pak kdykoliv za běhu programu může veškeré informace zobrazit a například se ubezpečit o korektnosti normál sledovaného povrchu.

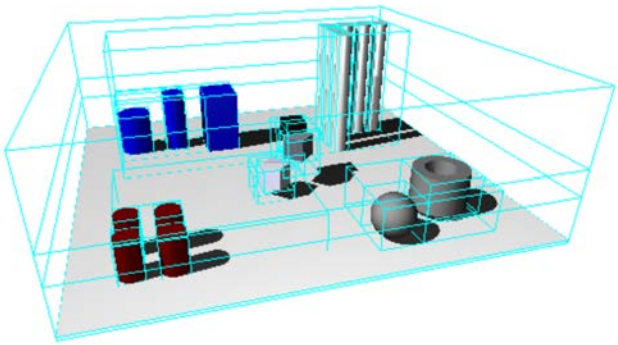
Tato jednoduchá komponenta je zároveň vhodná pro ukázku typického prvku uživatelského rozhraní zaregistrovaného zobrazovacím průchodem. Prvek grafického rozhraní je zobrazen pouze v případě, nachází-li se vykreslovací průchod v tabulce specifikující zobrazovací řetězec a to i v případě, že není aktivní. Obrázek 3.23 znázorňuje ovládací prvek určený ke zvolení části g-bufferu pro zobrazení. Před samotným názvem ovládací komponenty je předřazeno číslo přiřazeného zobrazovacího průchodu z tabulky.



Obrázek 3.23: Ovládací prvek GUI určený ke zvolení části g-bufferu pro zobrazení. Název prvku obsahuje předřazené číslo přesně specifikující, k jakému zobrazovacímu průchodu náleží.

3.7.7 Zobrazení BVH

Během vývoje nových algoritmů nebo optimalizace stávajících, je často důležité být co nejpřesněji informován o dění ve scéně. Pro reprezentaci scény je použita hierarchie obálek, která je dynamicky aktualizována při změnách ve scéně. Právě pro zobrazení BVH slouží speciální zobrazovací průchod. BVH je ve VRUTu reprezentováno binárním stromem, který je při vykreslování obálek nutné projít. Procházení stromem probíhá do hloubky s postupným vykreslováním všech listů. Zobrazení hierarchie obálek ve scéně je zobrazeno na obrázku 3.24.



Obrázek 3.24: Zobrazení BVH v testovací scéně.

3.8 Doplňkové funkce

Jádro zobrazovacího modulu, kromě již popsané správy zobrazovacích průchodů, umožňuje využít několik dalších funkcí. Asi nejpodstatnější je možnost zobrazování v rozdílném rozlišení oproti velikosti vykreslovacího okna. Tato vlastnost umožňuje vytvořit obraz o výrazně vyšším rozlišení, než dosahuje zobrazovací zařízení a neměl by být problém takový obraz patřičně zařazeným zobrazovacím průchodem uložit, ať již jako statický obrázek nebo zaznamenat videosekvenci.

Další vlastností je podpora aktivního stereoskopického zobrazování. Tento mód je možné zapnout v modulu *Navigation*, kde se rovněž nachází nastavení všech parametrů (např. separace očí). Obraz je pak generován pro pravé i levé oko samostatně s patřičně nastavenou pohledovou a projekční maticí kamery. Uložíme-li výsledný obraz pro levé i pravé oko do quad bufferu, grafická karta automaticky zajišťuje synchronizaci přepínání obrazu s brýlemi.

Poslední zde zmíněnou vlastností je podpora práce zobrazovacího modulu v clusteru. VRUT je často využíván k zobrazování v různých stereoskopických systémech (např. CAVE). K zobrazování na takto moderních zařízeních je obvykle zapotřebí práce více počítačů pro sestavení výsledného obrazu. Vykreslování je pak nutné správně synchronizovat mezi všemi počítači v síti, které se na zobrazení podílejí. Modul *Cluster* zajišťuje přenos dat přes síť, ale povinností zobrazovacího modulu je korektně reagovat na zprávy o stavu ostatních počítačů a ve vhodnou chvíli překreslovat obraz.

3.9 Souhrn

Tato kapitola se zabývala implementací nového zobrazovacího modulu do systému VRUT. Modul umožňuje rozšíření o nové algoritmy a efekty pracující v reálném čase v podobě na sobě nezávislých částí nazývaných zobrazovací průchody. Množství různorodých průchodů ukazuje potenciál nového modulu. Přidání nového, třeba jen testovacího kódu, nezhoršuje přehlednost a rychlost implementace, jelikož mohou být v danou chvíli zapnuté jen ty průchody, které jsou skutečně potřebné. Realizované zobrazovací průchody obsahuje tabulka 3.3.

Základní průchody	**Doplňkové průchody**
Ořez scény	Vytvoření stínů
Vytvoření g-bufferu	Ambientní zastínění v prostoru obrazu
Výpočet osvětlení	Antialiasing
Průhledné objekty	Simulace hloubky ostrosti
	Zrcadlové plochy
	Obsah g-bufferu
	Zobrazení BVH

Tabulka 3.3: Seznam realizovaných zobrazovacích průchodů.

Kapitola 4

Výsledky

Tato kapitola se věnuje testování nové implementace, jak z hlediska rychlosti, tak z hlediska kvality zobrazení v porovnání s původním zobrazovacím modulem. Testování probíhalo na několika scénách lišících se komplexností a druhy použitých materiálů. Měření bylo provedeno na třech rozdílných hardwarových konfiguracích s grafickými kartami od AMD i Nvidie.

4.1 Testovací scény

Scény použité při testování jsou znázorněny na obrázku 4.1 s parametry uvedenými v tabulce 4.1. Scéna TestScene reprezentuje geometricky jednoduchou scénu, která však obsahuje netypické materiály včetně průhledné geometrie. Krychličky uprostřed scény vyžadují automatické generování souřadnic pro mapu okolí nebo je zde i krychlička vykreslovaná pomocí vlastního shaderu. Závodní trať Forza původně vytvořená Williamem Butlerem a Seanem Staffordem, kterou přepracoval Andrew Sumner, představuje rozsáhlejší typ scény. Scénu Crytek Sponza vytvořil Frank Meinl podle předlohy vymodelované Markem Dabrovicem. Model konferenční místnosti byl ručně vytvořen oměřením reálné konferenční místnosti a je obecně používaný pro testování globálního osvětlení. Fabia2 reprezentuje komplexní scénu s více jak pěti milióny trojúhelníky, velkým množstvím materiálů a karoserie je vykreslována pomocí vlastního shaderu. Scény Crytek Sponza a Conference byly staženy z <`http://graphics.cs.williams.edu/data/`>. Do všech scén bylo pro testování přidáno jedno směrové světlo.

Obrázek 4.1: Obrázky testovacích scén.

	Trojúhelníky	Uzly BVH	Materiály	Textury
TestScene	1954	30	11	4
Forza	72 082	2811	132	132
Crytek Sponza	262 267	383	26	25
Conference + Dragon	1 202 485	44	36	0
Fabia2	5 855 169	3113	86	30

Tabulka 4.1: Parametry testovacích scén.

4.2 Testovací hardware

Měření časové náročnosti probíhalo na třech hardwarových konfiguracích podporujících alespoň OpenGL verze 4.0, viz tabulka 4.2. Jednalo se o dva klasické stolní počítače s výkonnými grafickými kartami Nvidia a slabší notebook s kartou výrobce AMD. Na všech počítačích běžel 64 b operační systém Windows 7.

	HW5730	HW560	HW6000
GPU	ATI Radeon HD 5730	NVIDIA GeForce GTX 560 Ti	2x NVIDIA QUADRO 6000, SLI vypnuté
CPU	Intel Core i7 720QM 1.6 GHz	Intel Core i5 2500K 3.30 GHz	2x Intel Xeon X5680 3.33 GHz
Paměť GPU	1 GB GDDR3	4 GB GDDR5	6 GB GDDR5
RAM	4 GB	8 GB	24 GB
Operační systém	Win 7, 64 b	Win 7, 64 b	Win 7, 64 b

Tabulka 4.2: Parametry testovacích sestav.

4.3 Výkonnostní testy

Veškeré měření probíhalo pomocí skriptů definujících přesné animační sekvence a nastavení parametrů pro jednotlivé typy scén. Výsledné hodnoty byly v průběhu animace zaznamenávány a na závěr vytvořen průměr z nejméně 1200 hodnot. Měření bylo prováděno s přesností na mikrosekundy s následným zaokrouhlením na desetinu milisekundy. Pokud nebude uvedeno jinak, všechny časy obsažené v tabulkách jsou v milisekundách.

4.3.1 Srovnání nového modulu s původním

Nejprve byla porovnána výpočetní náročnost původního modulu *rendergl* a nového *rendergl2*. Původní modul využívá klasické přímé zobrazování pomocí display listů a fixní OpenGL řetězec, zatímco nový modul veškeré operace provádí modernějším přístupem pomocí shaderů a VBO. Aby byly výsledky alespoň částečně srovnatelné, jsou v *rendergl2* aktivovány pouze základní zobrazovací průchody, jmenovitě: ořez scény, vytvoření g-bufferu, výpočet osvětlení a zobrazení průhledné geometrie. Měření probíhalo s výchozím nastavením obou modulů s aktivovaným čelovým světlem ve dvou rozlišeních. Jako základní rozlišení bylo zvoleno

800 x 600 px a další trochu netypické rozlišení 1366 x 768 px. Širokoúhlé rozlišení bylo zvoleno, jelikož modul *rendergl* neumožňuje vykreslovat ve vyšším rozlišení, než obsahuje zobrazovací zařízení a velikost tedy byla přizpůsobena maximálnímu rozlišení nejmenšího monitoru ze tří testovacích počítačů. Naměřené hodnoty jsou pak obsaženy v tabulkách 4.3 a 4.4.

800 x 600 px	TestScene	Forza	Crytek Sponza	Conference + Dragon	Fabia2
HW5730					
rendergl	5,0	8,7	5,4	6,2	57,8
rendergl2	5,6	10,9	8,0	10,2	31,8
HW560					
rendergl	6,5	9,9	12,4	42,6	240,1
rendergl2	2,8	8,7	6,4	7,4	24,8
HW6000					
rendergl	4,3	9,4	1,9	2,0	28,0
rendergl2	1,4	3,9	2,1	2,6	16,86

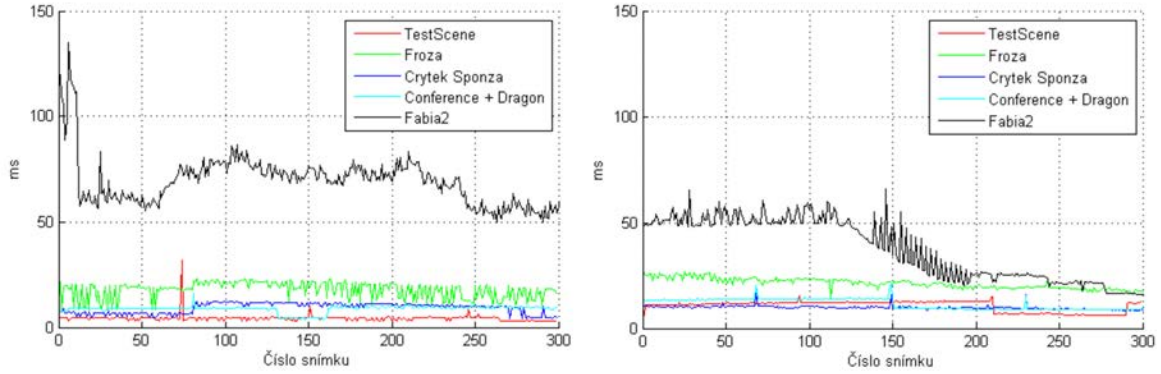
Tabulka 4.3: Doba vykreslení v milisekundách pro modul *rendergl* i *rendergl2*. Tabulka obsahuje naměřené hodnoty pro všech pět testovacích scén ve dvou rozlišeních.

1366 x 768 px	TestScene	Forza	Crytek Sponza	Conference + Dragon	Fabia2
HW5730					
rendergl	5,2	10,5	7,0	6,9	76,4
rendergl2	9,9	16,6	15,7	15,9	44,2
HW560					
rendergl	7,1	13,2	14,6	45,9	302,9
rendergl2	3,9	11,6	10,2	9,4	25,9
HW6000					
rendergl	4,4	12,4	2,6	2,2	32,0
rendergl2	2,0	5,9	3,6	3,5	22,4

Tabulka 4.4: Stejné měření jako v tabulce 4.3, pro rozlišení 1366 x 768 px.

Z výsledků je patrná vyšší režie modulu *rendergl2* zejména pak vytvoření g-bufferu, která se projevuje na pomalejší grafické kartě AMD. Naopak nový modul vykazuje lepší výsledky na obou moderních kartách výrobce Nvidia a jeho náskok se díky odloženému stínování zvyšuje s velikostí scény. Modul rendergl2 navíc může být lehce znevýhodněn výpočtem Phongova osvětlení pro každý pixel obrazu samostatně.

Graf 4.2 znázorňuje zaznamenaný průchod jednotlivými scénami. Vodorovná osa obsahuje pořadí snímku a horizontální osa čas v milisekundách potřebný pro jeho vykreslení. Zajímavé je sledovat pokles doby zpracování snímku nejnáročnější scény Fabia2 (černě), způsobený množstvím geometrie odstraněným ořezem pohledovým jehlanem. V první fázi animace pohled kamery zabírá celou přední část i s neviditelným motorem a postupně přechází směrem k zadní části, kde již tuto geometrii není nutné zpracovávat a je zcela odstraněna.



Obrázek 4.2: Rychlost vykreslení snímku v závislosti na průchodu scénou. Měření průchodu všemi scénami zobrazenými modulem *rendergl* (vlevo). Shodné měření pro modul *rendergl2* (vpravo).

4.3.2 Náročnost jednotlivých průchodů

Tabulky 4.5 a 4.6 obsahují naměřené časové nároky v milisekundách všech důležitých zobrazovacích průchodů. Ořez pohledovým jehlanem (Cull) podle očekávání není příliš náročný na rozdíl od vytvoření g-bufferu (G-buffer). Poměrně vysoké nároky jsou způsobeny vykreslováním velkého množství geometrie, které by bylo v některých případech vhodné omezit doplněním ořezu zastíněné geometrie, která nepřispívá do výsledného obrazu. Zobrazovací průchody pro výpočet osvětlení (Lighting a Shadow map), z nichž druhý v pořadí navíc vytváří stíny od přímého osvětlení, jsou především závislé na rozlišení obrazu. V případě zobrazení stínů hraje roli i počet geometrie vykreslované do stínové mapy a její rozlišení, které bylo při testech nastaveno na 2048 x 2048 px. Nejnáročnější průchod antialiasingu (SRAA) v sobě obsahuje časově náročné vytvoření dodatečných vzorků a jeho podrobnější analýzou se zabývá následující sekce. Během tohoto měření byla aktivována verze hledající diskontinuity v hloubkovém i normálovém bufferu. Zobrazovací průchody pro simulaci hloubky ostrosti (DOF) nebo ambientní zastínění (SSAO) jsou závislé na měřítku scény, proto byly jejich parametry, jako je poloměr generování vzorků nebo vzdálenost zaostřených objektů, nastaveny pro každou scénu samostatně. V metodě SSAO bylo generováno 30 vzorků na jeden pixel obrazu. Rychlost poslední metody vykreslující průhlednou geometrii do scény (Transparency) je závislá především na počtu této geometrie, kdy v testovacích scénách se jí objevuje minoritní množství.

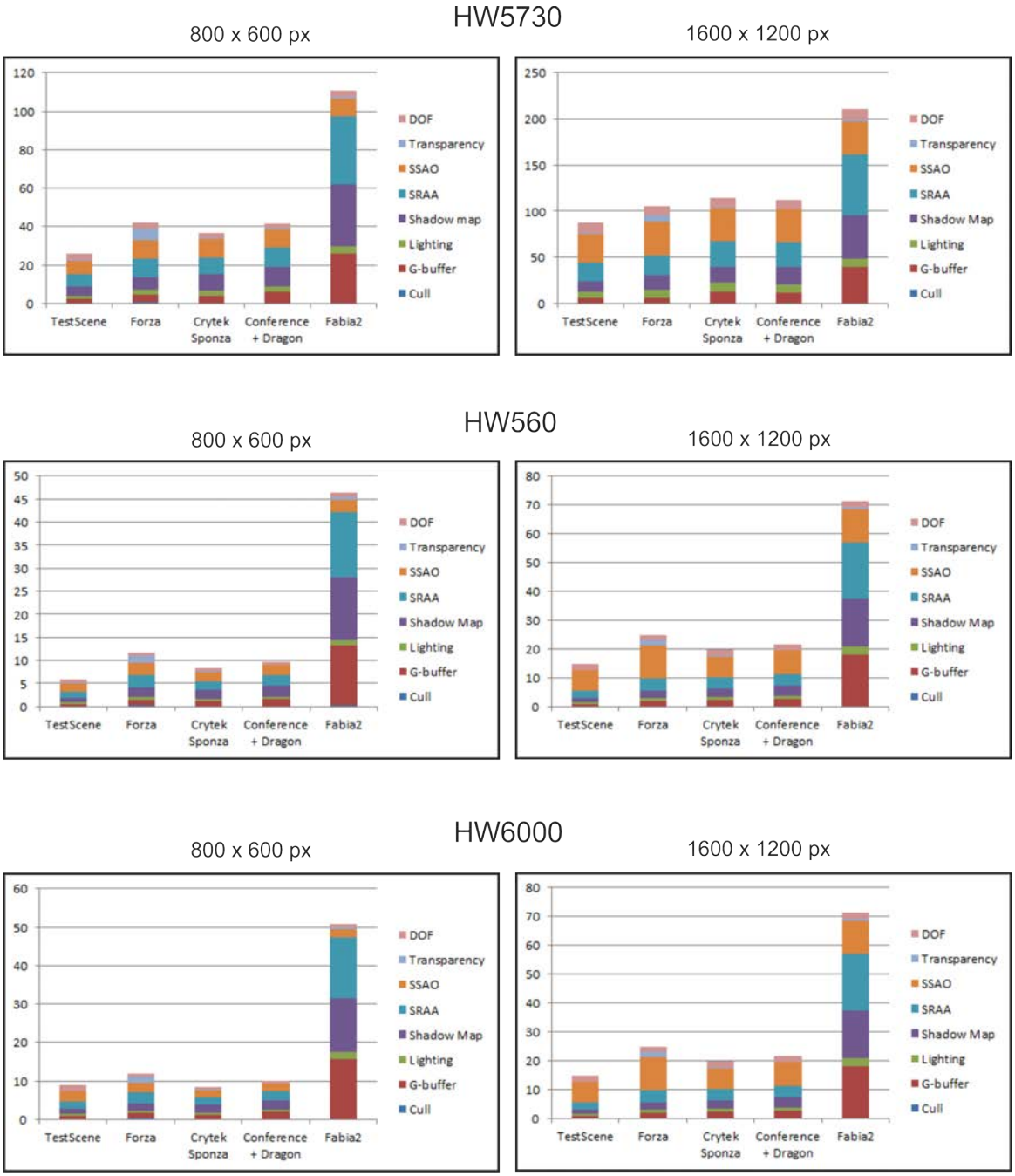
Porovnání tabulek 4.5 a 4.6, z nichž ve druhé byly měřeny stejné údaje pro dvakrát vyšší rozlišení, odhaluje závislost rychlosti některých průchodů na velikosti obrazu. U náročných průchodů pracujících převážně v prostoru obrazu, jako je SRAA nebo SSAO, dochází k výrazné změně rychlosti. Naopak ořez scény je na rozlišení prakticky nezávislý. Grafické znázornění stejných hodnot formou grafů obsahuje obrázek 4.3.

800 x 600 px	TestScene	Forza	Crytek Sponza	Conference + Dragon	Fabia2
HW5730					
Cull	0,2	0,5	0,2	0,2	0,4
G-buffer	1,9	3,9	4,0	6,1	25,7
Lighting	2,1	2,6	2,6	2,6	3,5
Shadow map	4,8	6,7	8,3	10,2	32,2
SRAA	6,1	9,5	8,9	10,1	35,5
SSAO	7,2	9,9	9,3	9,3	9,0
Transparency	0,4	5,6	0,5	0,1	1,5
DOF	3,2	3,1	3,1	3,1	3,2
HW560					
Cull	0,02	0,3	0,1	0,02	0,1
G-buffer	0,8	1,5	1,1	2,0	15,5
Lighting	0,6	0,4	0,6	0,5	2,0
Shadow map	1,4	1,9	2,1	2,5	14,0
SRAA	1,7	2,9	1,9	2,3	15,6
SSAO	2,7	2,5	1,8	1,8	2,4
Transparency	0,2	1,6	0,2	0,1	0,5
DOF	1,4	0,7	0,7	0,7	0,6
HW6000					
Cull	0,01	0,2	0,05	0,02	0,2
G-buffer	0,5	1,3	1,1	1,6	13,1
Lighting	0,4	0,5	0,5	0,5	1,2
Shadow map	1,0	2,0	2,0	2,4	13,5
SRAA	1,3	2,7	1,9	2,3	14,1
SSAO	1,8	2,7	2,0	2,1	2,8
Transparency	0,2	1,7	0,2	0,1	0,6
DOF	0,7	0,7	0,7	0,7	0,8

Tabulka 4.5: Čas v milisekundách potřebný pro jednotlivé zobrazovací průchody. Sloupce obsahují výpočetní časy v závislosti na scéně a řádky měřené zobrazovací průchody. Výsledky byly naměřeny při rozlišení 800 x 600 px na třech hardwarových sestavách.

1600 x 1200 px	TestScene	Forza	Crytek Sponza	Conference + Dragon	Fabia2
HW5730					
Cull	0,2	0,5	0,2	0,2	0,4
G-buffer	5,3	5,6	13,0	11,2	38,7
Lighting	6,9	8,8	9,2	9,2	9,4
Shadow map	11,2	15,2	17,1	19,1	47,0
SRAA	20,1	22,1	28,0	26,3	66,0
SSAO	32,1	36,2	35,9	35,9	35,5
Transparency	0,8	7,0	0,9	0,2	2,7
DOF	11,2	10,4	10,5	10,6	10,8
HW560					
Cull	0,02	0,2	0,05	0,01	0,2
G-buffer	0,7	1,7	2,2	2,7	17,7
Lighting	0,8	1,0	1,1	1,1	3,0
Shadow map	1,5	2,7	2,7	3,4	16,4
SRAA	2,4	4,1	4,1	4,1	19,7
SSAO	7,2	11,5	7,3	8,2	11,5
Transparency	0,2	1,8	0,3	0,1	0,7
DOF	2,1	1,9	1,9	1,9	2,0
HW6000					
Cull	0,01	0,2	0,05	0,02	0,2
G-buffer	1,2	1,8	2,9	2,6	14,1
Lighting	1,0	1,3	1,4	1,4	2,3
Shadow map	1,9	3,2	3,4	3,8	17,2
SRAA	3,2	4,6	5,1	4,9	18,7
SSAO	8,0	12,7	8,4	9,3	12,8
Transparency	0,2	2,1	0,3	0,1	0,9
DOF	2,3	2,2	2,1	2,2	2,5

Tabulka 4.6: Tabulka obsahující stejné měření jako 4.5, ale v rozlišení 1600 x 1200 px.



Obrázek 4.3: Graficky znázorněná data z tabulek 4.5 a 4.6, které obsahují časy v milisekundách pro jednotlivé zobrazovací průchody. Z grafů je zřetelná závislost některých průchodů na velikosti scény nebo rozlišení obrazovky.

4.3.3 Časová analýza metody SRAA

Tabulka 4.7 obsahuje výpočetní čas metody SRAA rozdělený na vytvoření bufferu pomocí multisamplingu a samotný filtr. Měření probíhalo na všech testovacích scénách kvůli analýze rychlosti antialiasingu v závislosti na velikosti scény. Z tabulky vyplývá, že rychlost filtru je nezávislá na velikosti scény a je velkou mírou ovlivněná rychlostí čtení dat z paměti. SRAA pracující pouze nad hloubkovým bufferem pak dosahuje dvakrát rychlejších časů než plná verze pracující i nad normálami. Naopak se objevuje velká závislost vytvoření dodatečných bufferů na velikosti scény, protože je nutné podobně jako u g-bufferu vykreslit veškerou viditelnou geometrii. Rozdíl mezi zpracováním pouze hloubky nebo hloubky a normál je logicky daný pouze zpracováním normál na GPU, kde jsou ukládány pomocí sférického mapování do příslušného bufferu. Ve výsledku je filtrování pouze pomocí hloubky vhodnější pro dynamické texturované scény, kde není rozdíl příliš patrný, ale dojde ke značné úspoře výpočetního výkonu.

HW5730	Vytvoření bufferu		Filtrování		Celkem	
1600 x 1200 px	H	H+N	H	H+N	H	H+N
TestScene	1,3	4,1	9,4	16,1	10,7	20,2
Forza	2,8	5,6	9,1	16,6	11,9	22,2
Crytek Sponza	3,9	8,7	10,1	20,4	14,0	29,1
Conference + Dragon	5,5	7,5	10,4	18,9	15,9	26,4
Fabia2	36,0	45,2	8,9	16,8	44,9	62.9

Tabulka 4.7: Rychlost jednotlivých fází metody SRAA. Tabulka obsahuje naměřené časy v milisekundách při použití pouze hloubkového bufferu (H) a při filtrování pomocí hloubky i normál (H+N).

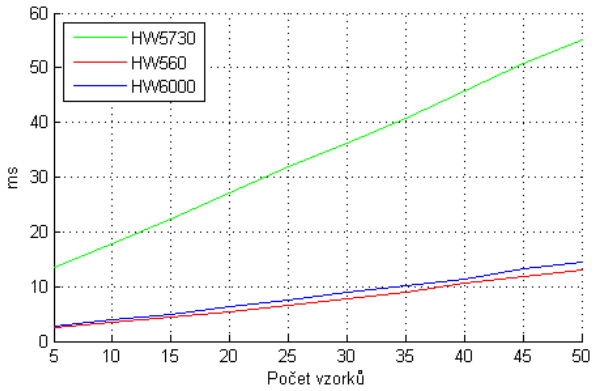
Díky ořezu pouze pohledovým jehlanem je generování dodatečných vzorků poměrně náročné, a proto proběhlo další měření porovnávající rychlost zobrazování v dvakrát větším rozlišení, kdy se prakticky jedná o supersampling (4xSSAA) a SRAA využívající čtyři dodatečné vzorky na pixel (4xSRAA). Tabulka 4.8 ukazuje naměřené hodnoty pro základní rozlišení 800 x 600 px. Supersampling tedy zobrazuje v rozlišení 1600 x 1200 px. SRAA hledá diskontinuity v obraze pomocí hloubky i normál. Měření nejprve probíhalo pro základní zobrazování bez přidaných efektů a následně se zapnutým SSAO a DOF. Z tabulky je zřejmé že SRAA vykazuje značnou úsporu času v případě aktivovaných efektů. Je tomu tak, protože v naší implementaci zobrazovacího řetězce probíhá zobrazování ve vyšším rozlišení v celé fázi a všechny efekty pracující v post-procesu pracují rovněž v tomto rozlišení. Tento přístup je samozřejmě nevhodný pro antialiasing, kdy by bylo výhodnější obraz nejprve zmenšit a pak dále zpracovávat, ale umožňuje vytvářet statické snímky nebo video sekvence v rozlišení, jaké nepodporuje použité zařízení.

HW5730	Bez efektů		SSAO + DOF	
800 x 600 px	4xSSAA	4xSRAA	4xSSAA	4xSRAA
TestScene	16,4	11,3	60,9	21,0
Forza	22,6	19,8	84,6	31,4
Crytek Sponza	27,3	16,4	73,0	28,1
Conference + Dragon	24,8	20,5	71,0	31,7
Fabia2	55,0	62,6	105,6	77,3

Tabulka 4.8: Porovnání rychlosti zobrazování ve větším rozlišení (supersampling) a metody SRAA v milisekundách. Během měření bylo u algoritmu SRAA aktivováno filtrování pomocí hloubky i normál. Výhoda SRAA se naplno projeví zejména při aktivaci dodatečných efektů pracujících v prostoru obrazu. V tomto případě byly aktivovány stínové mapy a hloubka ostrosti.

4.3.4 Závislost metody SSAO na počtu vzorků

Hlavní aspekt ovlivňující náročnost metody SSAO je počet vzorků generovaných v hemisféře určené pozicí bodu a normálou povrchu. Počet vzorků zároveň určuje přesnost výpočtu efektu zastínění a jedná se tedy o parametr, který je důležité přizpůsobit zobrazované scéně a výkonu GPU. Graf 4.4 zobrazuje lineární závislost rychlosti zobrazování na počtu generovaných vzorků. Měření probíhalo na scéně Crytek Sponza s nastaveným poloměrem hemisféry 25 v deseti měřeních pokaždé s jiným počtem vzorků a rozlišením 1600 x 1200 px.



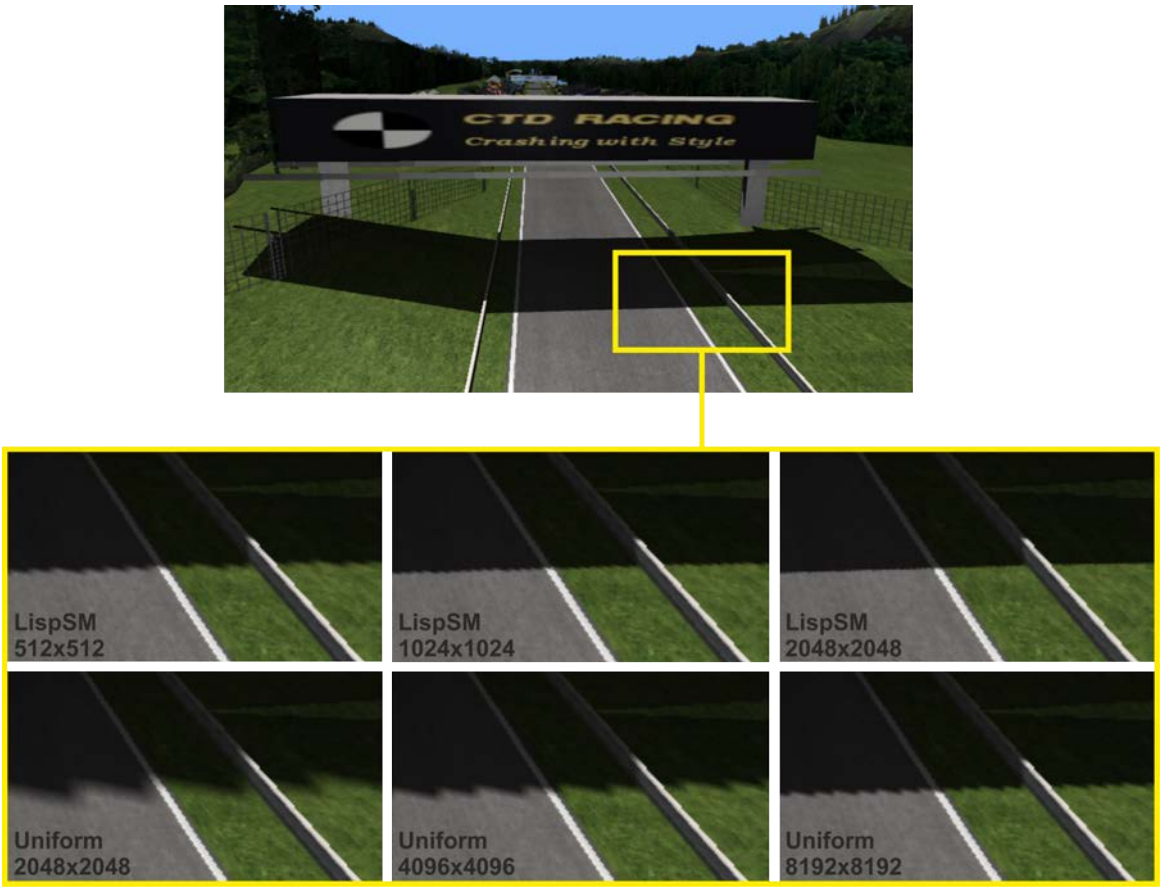
Obrázek 4.4: Závislost rychlosti metody SSAO na počtu vzorků. Svislá osa znázorňuje dobu výpočtu metody SSAO v milisekundách a vodorovná osa počet vzorků.

4.4 Vizuální kvalita zobrazení

Stejně důležitou součástí, jako je rychlost vykreslování, je kvalita zobrazení. Tato sekce se zabývá diskuzí kvality některých efektů v závislosti na nastavení jejich parametrů. Nechybí zde ani porovnání kvality obrazu vytvořeného novým a původním zobrazovacím modulem.

4.4.1 Závislost stínových map na rozlišení

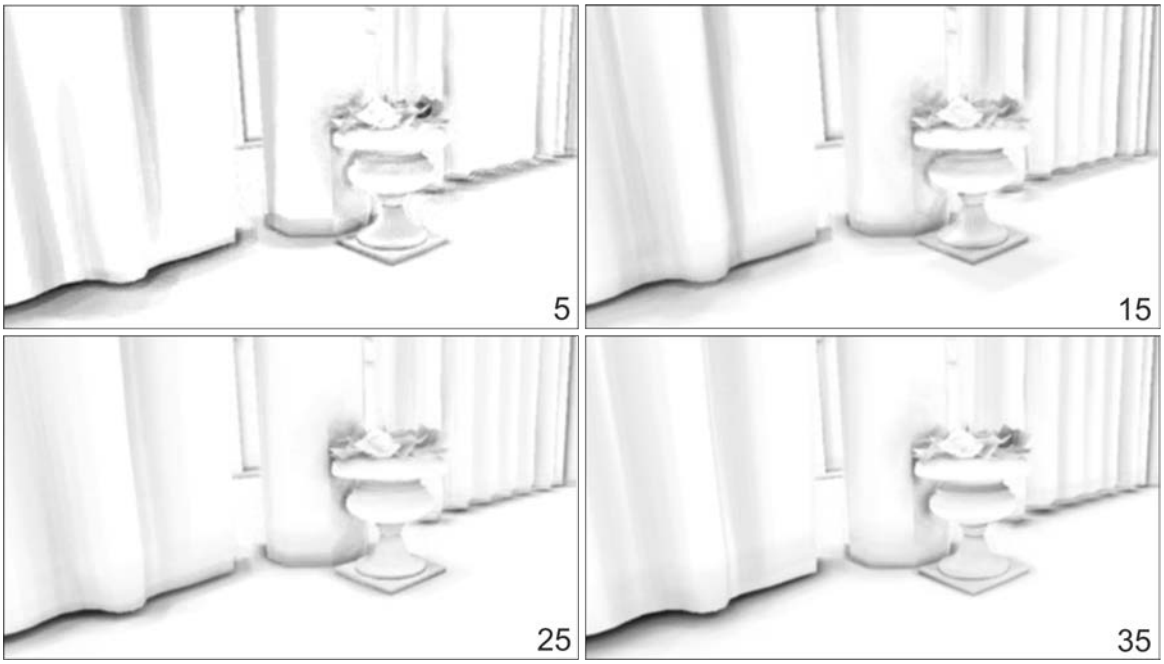
Kvalitu i rychlost vytvoření stínů ovlivňuje zaměření a rozlišení stínových map. Obrázek 4.5 znázorňuje porovnání uniformních stínových map s metodu LispSM ve scéně Forza s jedním směrovým zdrojem světla. Kvalita se projevuje zejména na hranách stínů, kde nedostatečné rozlišení způsobuje aliasing. Z porovnání obou metod je patrné, že uniformní stínové mapy dosahují prakticky stejného výsledku jako LispSM v rozlišení 512 x 512 px až při rozlišení 8192 x 8192 px, které znamená významnou výkonnostní ztrátu. LispSM naproti tomu dosáhne ideálního výsledku již při rozlišení 2048 x 2048 px. Výsledek je samozřejmě závislý na typu scény a pozici světla, proto pro porovnání byla zvolena rozsáhlá scéna se směrovým světlem, u které by měl být s aliasingem výraznější problém. Obecně lze říci, že LispSM dosahuje vždy výrazně lepších výsledků než uniformní stínové mapy.



Obrázek 4.5: Porovnání uniformních stínových map s metodou LispSM v rozsáhlé scéně Forza. Detail stínu znázorňuje, jakým způsobem se mění aliasing na hranách v závislosti na použité metodě a rozlišení stínové mapy. Zatímco LispSM v tomto případě dosahuje hladké linie stínu již při rozlišení 2048 x 2048 px, uniformní stínové mapy stejné kvality nedosáhnou ani při dvakrát větším rozlišení stínové mapy.

4.4.2 Kvalita SSAO v závislosti na počtu vzorků

Přesnost výpočtu ambientního zastínění ovlivňuje zejména počet vzorků generovaných uvnitř hemisféry pro jeden pixel obrazu. Jelikož jsou vzorky generované náhodně, u malého množství může dojít k poměrně výrazným rozdílům mapy zastínění podle pozice vzorků. Obrázek 4.6 obsahuje porovnání mapy zastínění v závislosti na počtu generovaných vzorků.



Obrázek 4.6: Mapa zastínění v závislosti na počtu vzorků generovaných pro jeden pixel obrazu. U extrémně malého počtu pěti vzorků není mapa příliš detailní a díky náhodnému generování se může značně lišit podle umístnění vzorků v hemisféře. Se zvětšujícím se počtem kvalita stínové mapy stoupá, ale roste výpočetní náročnost. Poměrně uspokojivých výsledků metoda dosahuje již při 25 vzorcích.

4.4.3 Porovnání nového a původního modulu

Nový modul si klade za cíl, kromě zlepšení udržitelnosti a rozšiřitelnosti kódu, vylepšení vizuální kvality obrazu. Obrázky 4.7 a 4.8 obsahují scénu Sponza vykreslenou novým a původním zobrazovacím modulem. V novém modulu byly aktivovány efekty stínových map a SSAO. Následující dvojice obrázků 4.9 a 4.10 obsahuje rovněž porovnání obou modulů, tentokrát pro scénu Forza s přidaným modelem Fabia2. Opět jsou aktivovány efekty SSAO a stínové mapy stejně jako v poslední dvojici porovnávající kvalitu obrazu ve známé scéně Conference s přidaným modelem Dragon, viz obrázky 4.11 a 4.12. Scéna Conference, na rozdíl od předchozích dvou, obsahuje reflektorový zdroj světla umístěný nad konferenčním stolem napodobující běžné osvětlení místnosti.

Obrázek 4.7: Scéna Sponza se směrovým a čelovým zdrojem světla zobrazená modulem *rendergl2* s aktivními efekty SSAO a stínovými mapami. SSAO generuje 30 vzorků uvnitř hemisféry s poloměrem 40. Stínové mapy o rozlišení 2048 x 2048 px byly vytvořeny metodou LispSM. Ambientní světlo ovlivňující osvětlení celé scény bylo nastaveno na hodnotu 40.

Obrázek 4.8: Scéna Sponza se stejnými zdroji světla jako na obrázku 4.7 zobrazená modulem *rendergl*. Modul *rendergl* nepodporuje ambientní světlo, proto je scéna méně nasvícená.

Obrázek 4.9: Scéna Forza s přidaným modelem Fabia2 vykreslená modulem *rendergl2*. Celou scénu nasvěcuje směrové světlo spolu s čelovým. Ambientní světlo je nastaveno na hodnotu 9 a dále jsou aktivní zobrazovací průchody SSAO a stínové mapy. SSAO generuje 30 vzorků s nastaveným poloměrem 700 a stínovou mapu o rozlišení 2048 x 2048 px počítá metoda LispSM.

Obrázek 4.10: Stejná scéna jako na obrázku 4.9, tentokrát zobrazená původním modulem *rendergl* rovněž s aktivním čelovým světlem.

Obrázek 4.11: Známá scéna Conference s přidaným modelem Dragon zobrazená modulem *rendergl2*. Efekty SSAO a stínové mapy vylepšují celkovou kvalitu zobrazení. SSAO generuje 30 vzorků s poloměrem hemisféry 0,4. Scéna obsahuje reflektorový zdroj světla umístěný nad konferenčním stolem směřujícím směrem dolů s širokým světelným kuželem o šířce 170°. Stíny jsou pak počítány metodou uniformních stínových map o velkém rozlišení 8192 x 8192 px.

Obrázek 4.12: Scéna Conference s přidaným modelem Dragon jako na obrázku 4.11 zobrazená modulem *rendergl*

.

Kapitola 5

Závěr

V této práci byl navržen a implementován nový zobrazovací modul do systému VRUT. Struktura modulu je navržena s ohledem na snadnou rozšiřitelnost a údržbu kódu, která je zajištěna rozdělením vykreslovacího procesu na menší celky nazývané zobrazovací průchody. Dělení zobrazování na menší celky zajišťuje možnost vhodného přizpůsobení vykreslování danému typu scény v podobě aktivace efektů značně vylepšujících kvalitu obrazu. Celkem bylo implementováno jedenáct zobrazovacích průchodů, které byly rozděleny do dvou kategorií, základní a doplňkové. Základní vykreslovací průchody řeší standardní vykreslování a doplňkové zpravidla vylepšují obraz nebo zobrazují dodatečné informace. Koncept takto navrženého modulu umožňuje stejným způsobem zařadit do procesu vykreslování moderní sofistikované algoritmy určené k fotorealistickému zobrazování i průchody vhodné při zobrazování technických zařízení.

Zobrazovací modul byl navrhován s požadavkem na práci v reálném čase, a proto rychlost vykreslování hraje významnou roli. Jako hlavní přístup bylo zvoleno odložené stínování, u kterého výpočet osvětlení probíhá pouze u viditelných pixelů obrazu. Rychlost zobrazovacích průchodů byla ověřena a porovnána na třech rozdílných hardwarových konfiguracích. Výsledky měření ukazují mírné rezervy u některých zobrazovacích průchodů, ale celkově jsou všechny části použitelné při zobrazování v reálném čase. Nezbytností bylo zlepšení vizuální kvality zobrazování, které bylo docíleno právě implementací několika efektů, jako je ambientní zastínění, vytvoření stínů nebo antialiasing.

Tématem pro navazující práce může být celá řada vylepšení. Díky modularitě se může jednat o implementaci nových zobrazovacích algoritmů nebo negrafických průchodů, například exportující videosekvence přímo do běžně používaných formátů. Dalším námětem pro vylepšení je implementace ořezu zastíněné geometrie ve scéně, která by velmi výrazně zrychlila práci několika realizovaných průchodů. Na závěr bych zmínil námět pro vylepšení týkající se celého VRUTu a to úpravu grafického rozhraní pro lepší přehlednost a vyšší stabilitu.

Literatura

[ACOD07] Toma Agoston, Csaba Csuprai, Juraj Onderik, and Roman Durikovic. Design of modular rendering pipeline. In *Proceedings of the 7th IEEE International Conference on Computer and Information Technology*, pages 322–328, 2007.

[Arv04] Jukka Arvo. Tiled shadow maps. In *Proceedings of the Computer Graphics International*, pages 240–247, 2004.

[BMSW11] Jiří Bittner, Oliver Mattausch, Ari Silvennoinen, and Michael Wimmer. Shadow caster culling for efficient shadow mapping. In *Proceedings Symposium on Interactive 3D Graphics and Games*, pages 81–88, 2011.

[CML11] Matthäus G. Chajdas, Morgan McGuire, and David Luebke. Subpixel reconstruction antialiasing for deferred shading. In *Symposium on Interactive 3D Graphics and Games*, pages 15–22, 2011.

[Cro77] Franklin C. Crow. Shadow algorithms for computer graphics. In *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 242–248, 1977.

[Don11] Don Woligroski. Coverage sampling modes: Nvidia's csaa and amd's eqaa, 2011. <`http://www.vtk.org`>, stav z 7. 11. 2012.

[DWS+88] Michael Deering, Stephanie Winner, Bic Schediwy Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: a vlsi system for high performance graphics. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '88, pages 21–30, New York, NY, USA, 1988. ACM.

[Joh11] John Chapman. Ssao tutorial, 2011. <`http://www.john-chapman.net/content.php?id=8`>, stav z 12. 10. 2012.

[Kom] Komunita wxWidgets. About the wxwidgets project. <`http://www.vtk.orghttp://www.wxwidgets.org/about/`>, stav z 9. 11. 2012.

[Mir10] Miroslav Novotný. Diplomová práce - přibližný výpočet efektů globálního osvětlení na GPU, ČVUT, 2010. <`https://dip.felk.cvut.cz/browse/pdfcache/novotm21_2011dipl.pdf`>, stav z 19. 11. 2012.

[Mit09] Martin Mittring. A bit more deferred-CryEngine 3. In *Triangle Game Conference*, volume 4, 2009. <http://www.crytek.com/cryengine/presentations>.

[Nat12] Nathaniel Meyer. Shader effects: Depth of field, 2012. <http://devmaster.net/posts/3021/shader-effects-depth-of-field>, stav z 25. 11. 2012.

[PF] Fabio Policarpo and Francisco Fonseca. Deferred shading tutorial. <http://www710.univ-lyon1.fr/~jciehl/Public/educ/GAMA/2007/Deferred_Shading_Tutorial_SBGAMES2005.pdf>.

[rBSF04] Jiří Žára, Bedřich Beneš, Jiří Sochor, and Petr Felkel. *Moderní počítačová grafika.* Computer Press, nám. 28 dubna 48, 635 00 Brno, 2th edition, 2004.

[RGS09] Tobias Ritschel, Thorsten Grosch, and Hans-Peter Seidel. Approximating dynamic global illumination in image space. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 75–82, 2009.

[Sim07] Simon Kozlov . Perspective shadow maps: Care and feeding, 2007. <http://http.developer.nvidia.com/GPUGems/gpugems_ch14.html>, stav z 2. 11. 2012.

[ST90] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph.*, 24(4):197–206, September 1990.

[Sta02] Marc Stamminger. Perspective shadow maps. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 557–562, 2002.

[Vá09] Václav Kyba. Diplomová práce - modulární 3d prohlížeč, ČVUT, 2009. <https://dip.felk.cvut.cz/browse/pdfcache/kybav1_2009dipl.pdf>, stav z 31. 10. 2012.

[Wil] Will Schroeder and Ken Martin and Bill Lorensen. Vizualization toolkit. <http://www.vtk.org>, stav z 5. 11. 2012.

[WSP04] Michael Wimmer, Daniel Scherzer, and Werner Purgathofer. Light space perspective shadow maps. In *Proceedings Eurographics Symposium on Rendering*, pages 143–151, 2004.

Seznam obrázků

1.1 Porovnání obrazu v závislosti na aplikovaných efektech. Zleva: scéna se základním osvětlením bez efektů, ambientní zastínění a ambientní zastínění se stíny od přímého osvětlení. 1

2.1 Sériové zapojení modulů - na vstupu modulů je vždy výstup z předchozích (vlevo). Paralelní zapojení zobrazovacího řetězce - výstupy z jednotlivých částí je nutné správně sloučit (vpravo). 5

2.2 Příklad definice vzoru (my irradiance sampler) a jeho aplikace ve schématu. Převzato z [ACOD07]. 5

2.3 Ostré stíny od bodového světla (vlevo), měkké stíny od plošného zdroje světla (vpravo). 7

2.4 Stínový objem zkonstruovaný podle siluety stínícího objektu. 7

2.5 Obálka scény S, pohledový objem V, pozice světla l a pohledový objem světla L (vlevo). Rozšíření pohledového objemu V o pozici světla l (uprostřed). Skupina bodů H, která musí být viditelná při generování stínové mapy (vpravo). 9

2.6 Projekce objektů nacházejících se za kamerou se změnou pořadí průsečíků s paprskem. 9

2.7 Porovnání kvality stínů vytvořených různými metodami. Zleva: uniformní stínové mapy, LispSM, PSM. Převzato z [WSP04]. 11

2.8 Stíny s použitím PCF (vlevo). Rozostření stínů na rovných plochách (vpravo). Převzato z [Sim07] 12

2.9 Hemisféra bodu scény B zasahující do okolních objektů (vlevo). Skloněná normála N vypočítaná z nezastíněných směrů (uprostřed). V případě, že světlo přichází ze stran, skloněná normála neodpovídá převažujícímu směru světla (vpravo) 13

2.10 Nahoře zleva: Scéna bez AO, běžné SSAO, SSDO, SSDO s nepřímými odrazy. Dole zleva: SSAO, SSDO, SSDO bez odrazů, SSDO s odrazy. Převzato z [RGS09] 14

2.11 Aliasing způsobený rasterizací při zobrazování oblého objektu (vlevo). Více vzorků v jednom pixelu při použití SSAA nebo MSAA (uprostřed). Červeně vzorky, ve kterých je počítáno osvětlení a modře osm dodatečných vzorků použitých v 4xEQAA (vpravo) 15

3.1 Hierarchie modulů ve VRUTu se zvýrazněným zobrazovacím modulem, kterým se zabývá tato práce. 20

3.2 Architektura nového zobrazovacího modulu. 22
3.3 Uživatelské rozhraní pro nastavení zobrazovacího řetězce. Průchody jsou spouštěny v odpovídajícím pořadí uvedení v tabulce. 23
3.4 Přeposílání události pohybu kamery pomocí myši přes jádro VRUTu až do jednotlivých průchodů v zobrazovacím řetězci. 24
3.5 Zobrazení komponent g-bufferu. Zleva shora: Dekódovaná normálová, hloubková, difuzní a spekulární textura. 29
3.6 Implementované druhy světel. Zleva: bodové, směrové a reflektorové. 31
3.7 Scéna testScene.fhs obsahující pět krychliček, každou s jiným materiálem. Zleva: průhlednost definovaná v textuře, mapování prostředí, celá průhledná krychlička, generování texturovacích souřadnic v prostoru kamery, krychlička s vlastním shaderem v materiálu. 32
3.8 Konstrukce prostoru světla P, který zahrnuje konvexní obálku H světelných paprsků, které mohou vrhat stíny do pohledového objemu. 33
3.9 Zobrazení stínů v malé scéně. Zleva: scéna bez stínů, metoda LispSM se směrovým typem světla a uniformní stínové mapy s reflektorovým světlem. 35
3.10 Generování vzorků v okolí bodu. Vzorky rozložené v celém okolí bodu (vlevo). Výhodněji generované vzorky na hemisféře (vpravo). 36
3.11 Distribuce vzorků vzhledem ke vzdálenosti od zpracovávaného bodu P. 36
3.12 Vytvořená mapa zastínění metodou SSAO. 37
3.13 Zvětšený detail nefiltrované mapy zastínění (vlevo). Stejný detail po aplikaci filtru o šířce čtyři pixely (vpravo). 38
3.14 Objekt v popředí nesprávně zastiňuje vzdálenou plochu (vlevo). Aplikovaná mapa zastínění ve scéně s kontrolou vzdálenosti (vpravo). 38
3.15 Z pohledu kamery neviditelný objekt A by měl přispívat k celkovému zastínění zpracovávaného bodu P. 39
3.16 Rekonstrukce jednoho vzorku pomocí SSRA. Červená barva v každém pixelu určuje vzorek, ve kterém je počítáno osvětlení, modrá pak dodatečné čtyři vzorky s informací o normále a pozici. Přímky A a B označují hrany ve scéně. Pro každý geometrický vzorek jsou počítány váhy z okolních vzorků s vypočítaným osvětlením. Vzorky nacházející se za hranou bilaterální filtr ohodnotí malou váhou a naopak vzorky bez velké změny v pozici nebo normále povrchu výrazně přispívají výsledné barvě pixelu. 40
3.17 Detekce hran ve scéně. Bílá barva označuje místa, kde dochází k filtrování, černě jsou pak označeny uniformní oblasti. 41
3.18 Vyhlazení hran pomocí SRAA. Bez vyhlazení (vlevo). Vyhlazeno pouze podle hloubkové mapy (uprostřed). Vyhlazeno pomocí hloubkové i normálové mapy (vpravo). 41
3.19 Rozmazaná textura filtrem SRAA na uniformní části scény (vlevo). Zabránění filtrování stejné uniformní oblasti (vpravo). 42
3.20 Objekty uvnitř hloubky ostrosti se z pohledu pozorovatele jeví jako ostré na rozdíl od objektů před nebo za touto oblastí. 42

3.21 Nastavení efektu DOF s červeně zobrazenou vzdáleností plného zaostření. Bílá barva označuje plně rozostřené oblasti a černá naopak ostré (vlevo). Aplikovaná hloubka ostrosti na scénu, kde dochází k rozmazání blízkých i vzdálených objektů podle nastavení parametrů kamery (vpravo). 45
3.22 Jednoduchá testovací scéna s objektem simulujícím zrcadlo. 46
3.23 Ovládací prvek GUI určený ke zvolení části g-bufferu pro zobrazení. Název prvku obsahuje předřazené číslo přesně specifikující, k jakému zobrazovacímu průchodu náleží. 47
3.24 Zobrazení BVH v testovací scéně. 48

4.1 Obrázky testovacích scén. 51
4.2 Rychlost vykreslení snímku v závislosti na průchodu scénou. Měření průchodu všemi scénami zobrazenými modulem *rendergl* (vlevo). Shodné měření pro modul *rendergl2* (vpravo). 54
4.3 Graficky znázorněná data z tabulek 4.5 a 4.6, které obsahují časy v milisekundách pro jednotlivé zobrazovací průchody. Z grafů je zřetelná závislost některých průchodů na velikosti scény nebo rozlišení obrazovky. 57
4.4 Závislost rychlosti metody SSAO na počtu vzorků. Svislá osa znázorňuje dobu výpočtu metody SSAO v milisekundách a vodorovná osa počet vzorků. 59
4.5 Porovnání uniformních stínových map s metodou LispSM v rozsáhlé scéně Forza. Detail stínu znázorňuje, jakým způsobem se mění aliasing na hranách v závislosti na použité metodě a rozlišení stínové mapy. Zatímco LispSM v tomto případě dosahuje hladké linie stínu již při rozlišení 2048 x 2048 px, uniformní stínové mapy stejné kvality nedosáhnou ani při dvakrát větším rozlišení stínové mapy. 60
4.6 Mapa zastínění v závislosti na počtu vzorků generovaných pro jeden pixel obrazu. U extrémně malého počtu pěti vzorků není mapa příliš detailní a díky náhodnému generování se může značně lišit podle umístnění vzorků v hemisféře. Se zvětšujícím se počtem kvalita stínové mapy stoupá, ale roste výpočetní náročnost. Poměrně uspokojivých výsledků metoda dosahuje již při 25 vzorcích. 61
4.7 Scéna Sponza se směrovým a čelovým zdrojem světla zobrazená modulem *rendergl2* s aktivními efekty SSAO a stínovými mapami. SSAO generuje 30 vzorků uvnitř hemisféry s poloměrem 40. Stínové mapy o rozlišení 2048 x 2048 px byly vytvořeny metodou LispSM. Ambientní světlo ovlivňující osvětlení celé scény bylo nastaveno na hodnotu 40. 62
4.8 Scéna Sponza se stejnými zdroji světla jako na obrázku 4.7 zobrazená modulem *rendergl*. Modul *rendergl* nepodporuje ambientní světlo, proto je scéna méně nasvícená. 62
4.9 Scéna Forza s přidaným modelem Fabia2 vykreslená modulem *rendergl2*. Celou scénu nasvěcuje směrové světlo spolu s čelovým. Ambientní světlo je nastaveno na hodnotu 9 a dále jsou aktivní zobrazovací průchody SSAO a stínové mapy. SSAO generuje 30 vzorků s nastaveným poloměrem 700 a stínovou mapu o rozlišení 2048 x 2048 px počítá metoda LispSM. 63

4.10 Stejná scéna jako na obrázku 4.9, tentokrát zobrazená původním modulem *rendergl* rovněž s aktivním čelovým světlem. 63
4.11 Známá scéna Conference s přidaným modelem Dragon zobrazená modulem *rendergl2*. Efekty SSAO a stínové mapy vylepšují celkovou kvalitu zobrazení. SSAO generuje 30 vzorků s poloměrem hemisféry 0,4. Scéna obsahuje reflektorový zdroj světla umístěný nad konferenčním stolem směřujícím směrem dolů s širokým světelným kuželem o šířce 170°. Stíny jsou pak počítány metodou uniformních stínových map o velkém rozlišení 8192 x 8192 px. 64
4.12 Scéna Conference s přidaným modelem Dragon jako na obrázku 4.11 zobrazená modulem *rendergl* 64

Seznam tabulek

3.1 Parametry světel inspirované jazykem VRML. 26
3.2 Hloubka, normály, difuzní barva, spekulární barva a typ materiálu spolu s hodnotou lesku v pěti 4B texturách. 29
3.3 Seznam realizovaných zobrazovacích průchodů. 49

4.1 Parametry testovacích scén. 52
4.2 Parametry testovacích sestav. 52
4.3 Doba vykreslení v milisekundách pro modul *rendergl* i *rendergl2*. Tabulka obsahuje naměřené hodnoty pro všech pět testovacích scén ve dvou rozlišeních. . 53
4.4 Stejné měření jako v tabulce 4.3, pro rozlišení 1366 x 768 px. 53
4.5 Čas v milisekundách potřebný pro jednotlivé zobrazovací průchody. Sloupce obsahují výpočetní časy v závislosti na scéně a řádky měřené zobrazovací průchody. Výsledky byly naměřeny při rozlišení 800 x 600 px na třech hardwarových sestavách. 55
4.6 Tabulka obsahující stejné měření jako 4.5, ale v rozlišení 1600 x 1200 px. . . . 56
4.7 Rychlost jednotlivých fází metody SRAA. Tabulka obsahuje naměřené časy v milisekundách při použití pouze hloubkového bufferu (H) a při filtrování pomocí hloubky i normál (H+N). 58
4.8 Porovnání rychlosti zobrazování ve větším rozlišení (supersampling) a metody SRAA v milisekundách. Během měření bylo u algoritmu SRAA aktivováno filtrování pomocí hloubky i normál. Výhoda SRAA se naplno projeví zejména při aktivaci dodatečných efektů pracujících v prostoru obrazu. V tomto případě byly aktivovány stínové mapy a hloubka ostrosti. 59

Příloha A

Seznam použitých zkratek

AO Ambient occlusion

API Application programming interface

BVH Bounding volume hierarchy

CSAA Coverage sample antialiasing

DOF Depth of field

EQAA Enhanced quality antialiasing

FBO Frame buffer object

LispSM Light space perspective shadow maps

MSAA Multisample antialiasing

PCF Percentage-closer filtering

PPS Post-perspective space

PSM Perspective shadow maps

SRAA Subpixel reconstruction antialiasing

SSAA Supersample antialiasing

SSAO Screen space ambient occlusion

SSDO Screen space directional occlusion

VRUT Virtual Reality Universal Toolkit

3D Three-Dimensional

Příloha B

Uživatelská příručka

Tato část obsahuje stručný popis základního ovládání VRUTu a zobrazovacího modulu *rendergl2*. Nezabývá se však samotnou kompilací VRUTu, ke které je možné najít dostatek informací přímo na přiloženém DVD v dokumentech u zdrojových kódů.

Spuštění

Máme-li VRUT přeložený, pomocí souboru vrut.exe je možné spustit aplikaci. VRUT je z velké části ovládán z konzole pomocí příkazů, které je možné zadávat ručně nebo je načíst automaticky ze skriptu. Název skriptu s posloupností příkazů lze zadat jako parametr při spuštění VRUTu z příkazové řádky. Zároveň pokud v pracovním adresáři existuje skript autoexec.cfg, je automaticky načten.

Základní příkazy

importscene "nazev sceny" - načtení scény zadaného názvu
renderscene 0 - zobrazení scény s ID 0
usemodule rendergl2 - nastavení zobrazovacího modulu, který je aktivován při zavolání příkazu renderscene
fit 0 - nastavení kamery pro záběr na celou scénu
runmodule SceneGraph - spuštění modulu SceneGraph, stejným způsobem lze spustit jakýkoliv modul VRUTu (například LightEditor, MaterialEditor)

Manipulace

Základní pohyb ve scéně pomocí myši zajišťuje modul Navigation, ve kterém je důležité i nastavení blízké a vzdálené ořezávací roviny pohledového jehlanu. Korektní nastavení těchto parametrů vyžadují některé zobrazovací průchody pro správnou práci. Pomocí dvojkliku prostředním tlačítkem myši je pak možné nastavit střed souřadnicového systému využívaného při manipulaci.

Modul *rendergl2* je ovládán intuitivně pomocí grafického rozhraní. Důležitými prvky jsou: tabulka pro nastavení nebo aktivaci zobrazovacích průchodů, nastavení intenzity ambientního světla, zapnutí čelového světla, ořez odvrácených ploch a nastavení rozlišení zobrazování. Rozlišení je nastavováno relativně k velikosti okna v procentech. Další prvky jsou dynamicky přidávány do grafického rozhraní a nastavují speciální vlastnosti jednotlivých vykreslovacích průchodů.

Příloha C

Obsah přiloženého DVD

- readme.txt - popis obsahu DVD a návod ke spuštění aplikace
- bin/ - spustitelná verze s ukázkovými skripty a daty
- text/ - text diplomové práce
- doc/ - programátorská dokumentace vygenerovaná programem Doxygen
- latex/ - zdrojové soubory pro latex
- src/ - zdrojové soubory celé aplikace