VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ZOBRAZENÍ STÍNŮ METODAMI DĚLÍCÍMI POHLEDOVÉ TĚLESO V ROZHRANÍ OPENGL

SHADOW RENDERING USING VIEW FRUSTUM SPLITTING METHODS IN OPENGL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **DOMINIK ROHÁČEK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. JOZEF KOBRTEK,**
SUPERVISOR

BRNO 2018

Zadání bakalářské práce/21308/2017/xrohac04

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Roháček Dominik**
Obor: Informační technologie
Téma: **Zobrazení stínů metodami dělícími pohledové těleso v rozhraní OpenGL**
Shadow Rendering Using View Frustum Splitting Methods in OpenGL
Kategorie: Počítačová grafika

Pokyny:

1. Prostudujte knihovnu OpenGL.
2. Nastudujte algoritmus pro zobrazování stínů Shadow Mapping a jeho dvě varianty - Parallel-Split Shadow Mapping a Sample-Distribution Shadow Mapping.
3. Obě metody naimplementujte.
4. Vytvořte 3D scénu s benchmarkem (průlet kamerou).
5. Vykonejte měření na různých grafických kartách.
6. Porovnejte obě metody z hlediska výkonu, kvality stínů a implementační náročnosti.

Literatura:

- https://software.intel.com/en-us/articles/sample-distribution-shadow-maps
- http://http.developer.nvidia.com/GPUGems3/gpugems3_ch10.html

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Kobrtek Jozef, Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

L.S. VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Tato práce pojednává o generování kaskádových stínových map ve 3D aplikacích. Popisuje dvě vybrané metody dělení pohledového tělesa na tyto kaskády a rozebírá jejich rozdílnosti. Pro analýzu byla vyvinuta aplikace za použití rozhraní OpenGL, která na třech scénách ukazuje výhody a nevýhody těchto metod. Práce se zaměřila na porovnání vykreslování každé kaskády zvlášť a pomocí vícevrstvých textur pro shadow mapy. Analýza dat prokázala některé výhody vrstového vykreslování. Metoda PSSM byla dle analýzy rychlejší, ovšem metoda SDSM na druhou stranu produkovala lepší výsledky.

Abstract

The content of this thesis discusses the generation of cascaded shadow maps in 3D applications. It describes two chosen methods of view frustum splitting into sub-frustum and discusses their differences. It was developed an application for analysation purposes in OpenGL which shows their advantages and disadvantages on three testing scenes. This work is focused on a comparison of drawing each split separately and using layered rendering. Data analysis proved some advantages of layered rendering. The PSSM was faster in accord to data analysis, but SDSM, on the other hand, produces more accurate shadows.

Klíčová slova

Vykreslování v reálném čase, Shadow mapping, Aliasing stínů, Parallel-Split Shadow Maps, Sample-Distribution Shadow Maps

Keywords

Real-time rendering, Shadow mapping, Aliasing stínů, Parallel-Split Shadow Maps, Sample-Distribution Shadow Maps

Citace

ROHÁČEK, Dominik. *Zobrazení stínů metodami dělícími pohledové těleso v rozhraní OpenGL.* Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jozef Kobrtek,

Zobrazení stínů metodami dělícími pohledové těleso v rozhraní OpenGL

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jozefa Kobrtka. Další informace mi poskytl Ing. Tomáš Barák. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Dominik Roháček
16. května 2018

Poděkování

Rád bych zde poděkoval panu Ing. Jozefovi Kobrtkovi za odbornou pomoc v průběhu práce a panu Ing. Tomáši Barákovi za konzultace technických detailů implementace.

Obsah

Kapitola 1

Úvod

Při projekci 3D scén, na dvou rozměrné plochy monitorů našich počítačů je světlo jedním z našich hlavních způsobů reprezentace hloubky obrazu. Světlo nám pomáhá naznačit úhel, ve kterém se objekt nachází vůči pozorovateli, pozici světla díky odleskům nebo třeba pozměněním barev jednotlivých objektů. Ovšem hloubku scény nenaznačuje jen světlo, ale stejně výrazně i jeho nedostatek, stín. Díky stínu jsme schopni při pohledu na scénu odlišit, který objekt se nachází před kterým právě díky stínu, který na něj vrhá.

Uvěřitelná reprezentace 3D světa je klíčová jak v profesionálních aplikacích, jako jsou vizualizace staveb, skenů objektů a fotomontáže tak v real time aplikacích vykreslující 3D scény. Dalším odvětvím, kde se metody vykreslování stínů používají, jsou různé real time aplikace. V těchto aplikacích však musíme balancovat mezi kvalitou obrazu a obnovovací frekvencí tak, aby pozorovatel měl pocit plynulého obrazu a nepoznal propady (ideálně je omezit na minimum) obnovovací frekvence. Stálá obnovovací rychlost je také nutná v případě použití technik jako je V-Sync (tento požadavek již není nutný v případě monitorů s technologií G-Sync).

Během posledních let probíhá v oblasti zdokonalení algoritmů pro výpočet stínů rozsáhlý výzkum a vzniká mnoho nových metod. Tato práce si proto neklade za cíl zdokumentování všech těchto metod, ale vybírá si dvě podobné metody vycházející z metody shadow mapping a rozšiřující ji o použití několika takových map. Cílem této práce je jejich implementace, demonstrace na vhodně zvolené scéně a následné porovnání jak z hlediska výkonu, tak kvality vykreslení.

Práce je rozdělená na kapitoly tak, aby čtenář dostal postupně přehled o celé práci. V následující kapitole 2 se čtenář seznámí s teorií shadow mappingu a obou metod, které v této práci porovnávám. Kapitola 3 obsahuje návrh implementace a ihned následující kapitola 4 popíše i samotnou implementaci tohoto návrhu. Další kapitola 5 shrnuje způsoby měření metod a jejich porovnání. V poslední kapitole 6 se nachází závěry z porovnání metod a návrh dalších vylepšení.

Výstupem této práce je aplikace v jazyce C++ ve standardu C++14 za použití knihovny OpenGL a knihovny GLEW. Tato aplikace bude sloužit k porovnání obou metod.

Kapitola 2

Teorie

Shadow mapping je v jádru jednoduchá metoda výpočtu stínů. Pro pochopení optimalizačních metod, na které je tato práce zaměřená, je důležité pochopit původní myšlenku, ze které se vyvinuly. Tomuto je věnována tato kapitola.

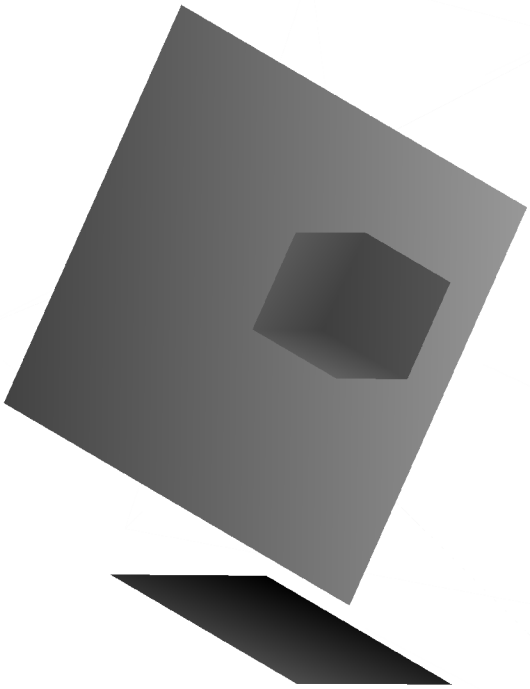
2.1 Shadow mapping

Pokud se na stíny podíváme čistě z fyzikálního hlediska můžeme říct, že stín je projevem absence světla. Cambridgeský slovník definuje stín jako "an area of darkness, caused by light being blocked by something"[1]. Pokud se pokusíme tuto definici zalgoritmizovat, dovede nás to k problému, jak zjistit zda právě kreslený bod je osvětlen, nebo světlo blokuje jiný objekt. Jinými slovy zda existuje z daného bodu přímka taková, aby bez přerušení šla směrem do světla. Toto by nás navedlo nejspíše na techniky z oblasti ray tracingu. Pokud však uvažujeme běžnou OpenGL render pipeline, kde renderujeme geometrii objektů, musíme zvolit jiný postup. Můžeme náš dotaz otočit a zjišťovat viditelnost objektů z pozice světla. [11]

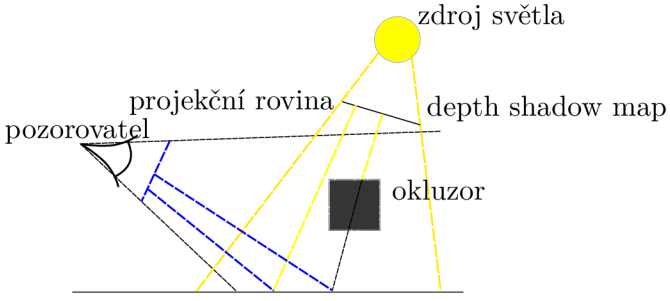
Toto se dá realizovat mnohem jednodušeji. Stačí nám scénu vykreslit z pohledu světla a poté přečíst informaci z depth bufferu (hloubková mapa). Příklad takové textury můžete vidět na následujícím příkladu 2.1 a schéma scény s shadow mapou je zobrazeno ve schématu 2.2. Toho se docílí tak, že si vytvoříme tzv. light-view projekční matici, která se sestaví tak jako bychom kameru umístili do místa, kde se nachází světlo a zkreslí se projekčním zkreslením dle vlastností světla. Toto zkreslení bude například pro směrové (directional) světlo stejné jako pro ortogonální projekci. Poté při finálním vykreslení potřebujeme znát jak view projekční matici pro aktivní kameru, tak i pro transformaci bodu do light view space a k tomu mít přístup k depth bufferu dříve vykresleném při pohledu z pozice zdroje světla, který již máme uložený do textury. Zda je bod zastíněn pak zjistíme tak, že jeho pozici ve world space vynásobíme light view-projekční maticí. Pokud takto získaná souřadnice má komponentu v ose z větší než ta, která je v shadow mapě (je vzdálenější od světla), můžeme říct, že je námi vykreslovaný bod zastíněn jiným objektem, který leží blíže ke světlu. Pokud je komponenta v ose z větší nebo rovna té v shadow mapě, tak vykreslujeme objekt, který je osvícen daným světlem [7].

Pokud bychom chtěli touto metodou vykreslovat scénu nasvícenou N světly, budeme potřebovat N takovýchto shadow map, které vykreslíme v N průchozích grafem scény. Poté při finálním kreslení budeme potřebovat N shadow map a N light-projekčních matic. Naštěstí

[1] https://dictionary.cambridge.org/dictionary/english/shadowed



Obrázek 2.1: Depth buffer shadow mapy pro scénu obsahující dva čtverce a jednu krychli



Obrázek 2.2: Schéma scény zobrazující shadow mapu

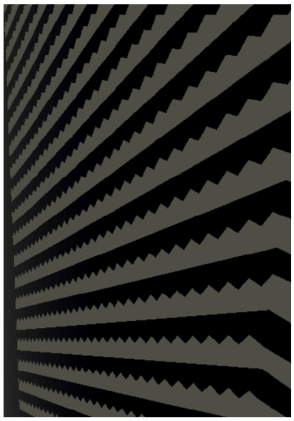
během kreslení depth map můžeme přeskočit v shaderech veškeré grafické operace, které neovlivňují siluetu objektu. Nemusíme tudíž počítat barvy, normály, odrazivost objektu, jejich hrubost atd. Toto může tato vykreslení velmi zrychlit. I přesto musíme být uvážliví s počtem světel, které chceme mít ve scéně.

2.1.1 Nedokonalosti shadow map

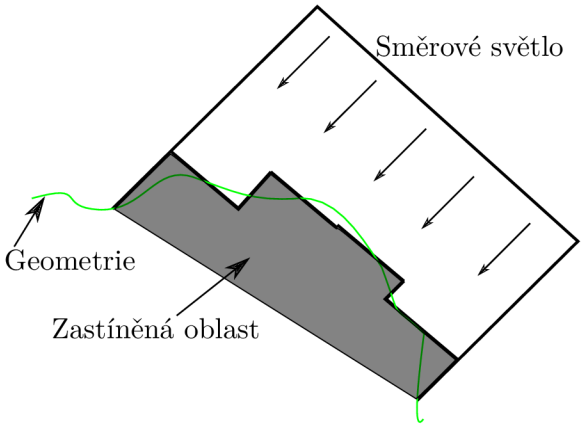
Protože textury jsou pouze diskretizovanou aproximací světa, který modelujeme ze spojitých modelů, vznikají nám nedokonalosti. Běžně je můžeme omezit tím, že obětujeme více výkonu a nebo je vyměníme za jiné nedokonalosti, které ale již nemusí být tolik viditelné nebo je pozorovatel nezahlédne vůbec. Právě tyto cíle mají metody prezentované v následujících odstavcích, nejdříve si však můžeme zmínit ostatní nedokonalosti, které zatěžují i mnou použité metody.

2.1.2 Akné

Povrchové akné (angl. shadow acne/shadow moire) je artefakt (viz obrázek 2.3) na povrchu 3D objektů způsobené granularitou shadow map. Toto může být způsobeno tzv. z-fightingem nebo tím, že stínová mapa nemá dostatečné rozlišení a tak texel v ní reprezentuje mnohem více pixelů ve výsledném obraze. Z-Fighting je proces, kdy je více pixelů rasterizováno do stejné hloubky a tak není jasné, který má větší/menší hloubku[7]. Druhý uvedený problém je zobrazený na obrázku 2.4. Jak můžete vidět, tak části geomterie se v shadow mapě promítly do jednoho pixelu textury i přesto, že část geometrie je před a část za touto hladinou. Projevem této chyby jsou pravidelné proužky nebo trojúhelníky na jednotlivých rovných plochách geometrie, které jsou zastíněné. Tomuto jevu se také anglicky říká "self-shadowing".



Obrázek 2.3: Shadow acne na povrchu objektu



Obrázek 2.4: Ukázka sebezastínění

Proti tomuto problému existuje několika možnostmi. První možností je pro shadow mapu použít takzvaný "front-face culling", tedy vykreslování pouze odvrácených ploch od kamery. Toto však způsobí, že veškerá naše geometrie musí být uzavřená a mít obě strany. Další možností je konstatní bias při porovnání pozice rasterizovaného bodu v shadow mapě a ve výsledné scéně. Toto však způsobí, že stíny mohou začínat kus od skutečné geomterie. Vylepšením této metody, které jsem implementoval ve své práci, je adaptivní bias s ohledem na orientaci plochy vůči pozici slunce [1].

2.1.3 Peter-Panning

Tento problém je přímo spojen s předchozím. Tím, že se přidá, se stíny odsadily od stěn a tak veškerá geometrie jakoby levituje ve vzduchu. Toto lze odstranit právě tím, že veškerá naše geometrie bude uzavřená a vypneme ořezávání předních stran.

2.1.4 Percentage-Closer Filtering

Vzrokováním pouze jediného pixelu z shadow mapy získáme pouze lokální informaci. Abychom dosáhli co největšího vyhlazení výsledných stínů, používáme různé metody vyhlazení stínů. Percetage-Closer Filtering[5] navrhuje vzorkování v okolí pixelu. Citovaná práce navrhuje několik metod výběru pixelů, které budeme vzorkovat. Můžeme například ohraničit náš bod oblastí, ve které budeme náhodně vzrokovat fixní počet bodů. Nebo použít Gaussovo rozložení. Můžeme také vzrokovat všechny body v okolí.

Problém této metody je, že každé vzorkování z shadow mapy samožejmě prodlužuje dobu nutnou pro vykreslení každého pixelu. Toto se potvrdilo i během měření výsledné aplikace a výsledky můžete najít v odstavci 5.4.

Důvodem, proč tuto metodu zmiňuji je, že ve výsledné aplikaci chci naznačit způsob, jakým je možno použít vyhlazování stínů. Zároveň také změřit, jak zásadní vliv bude mít filtrování na výslednou dobu vykreslení.

2.1.5 Plural sunlight depth buffers

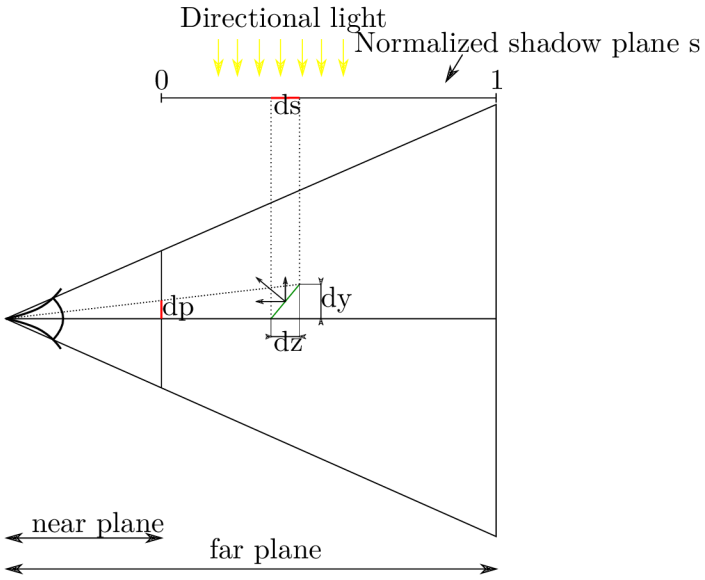
Tato práce si vzala za cíl dělit pohledové těleso na základě relativní pozice a směrů slunce vůči kameře [6]. Tato metoda zahrnuje netriviální výpočet jednotlivých ploch rozdělujících pohledové těleso v závislosti na tom, jakou část obrazu zabírají pixely v daném intervalu vzdálenosti od kamery. Počet takovýchto ploch se může lišit dle aktuálních nastavení kamery a jejího úhlu vůči slunci. Jednotlivé shadow mapy pak mohou mít různá rozlišení.

Hlavními nevýhodami této metody jsou dle Fan Zhang [9] následující

1. Náročnost výpočtu jednotlivých rovin dělící pohledové těleso
2. Měnící se rozlišení jednotlivých shadow map, které není podporováno hardwarem grafických karet. (Jejich proměnný počet může být řešen pomocí ring bufferů [2], stále však budeme provádět mnoho alokací a dealokací).
3. Nutnost několikanásobných shadow a vykreslovacích průchodů.

2.2 Parallel-Split Shadow Maps

První zkoumaná metoda vychází z vylepšení shadow map zvaného Cascaded shadow maps. Myšlenka této metody bylo omezení aliasingu v shadow mapách zvýšením rozlišení, které shadow mapa má. Toto nelze již z architektonických důvodů grafických karet dělat donekonečna. Vykreslit shadwow mapu v rozlišení 8K, 16K a více je nejen nepraktické, ale zabere i velkou část paměti na grafické kartě. Navíc tím rozložíme rovnoměrně texture space shadow mapy po celé scéně, přičemž ho nezužitkujeme stejně efektivně ve vzdálených částech scény jako částech bližších pozorovateli. V ideálním případě bychom tudíž chtěli, rozložit texture space shadow mapy rovnoměrně dle toho, kolik prostoru je danému místu věnováno ve screen space.



Obrázek 2.5: Schéma zobrazující vztahy mezi pohledovým tělesem, geometrií ve scéně a její projekcí do normalizované roviny shadow mapy

Metodu můžeme popsat následujícím zápisem. $PSSM(m, res)$, kde vstupními parametry jsou:

m Počet částí na které bude pohledové těleso rozděleno

res Rozlišení jednotlivých shadow-map.

2.2.1 Algoritmus

Samotný postup výpočtu stínů lze rozdělit do následujících kroků. Popisuji obecnou implementaci.

1. Celé pohledové těleso rozdělíme dle hloubky.
2. Pohledové těleso směrového světla rozdělíme na části tak, aby každá pokrývala právě jednu z částí pohledového tělesa.
3. Vykreslení jednotlivých částí pohledového tělesa světa.
4. Výpočet stínů při renderování scény.

Zásadní otázkou je, jak specifikovat dělící plochy pohledového tělesa. Jedna z cest je rozdělit prostor uniformně. Tímto však vyčleníme stejný prostor texture space všem hloubkám scény. To ovšem nereflektuje správně poměr texelů zabraných v shadow mapě a jejich podíl na výsledném obrazu. Co ovšem opravdu chceme je minimalizovat míra aliasu, kterou nám toto zanese do stínů. Míru aliasu lze spočítat následovnou rovnicí [8].

$$\frac{dp}{ds} = \frac{1}{\tan(\phi)} \frac{dz}{zds} \frac{\cos(\phi)}{\cos(\theta)} \tag{2.1}$$

Tuto chybu lze rozdělit na dvě hlavní složky - perspektivní a projekční alias. Z předchozí rovnice můžeme vyjmout perspektivní alias $\frac{dz}{zds}$ a projekční alias (způsobený zkreslením projekce kamery) $\frac{\cos(\phi)}{\cos(\theta)}$. Jak můžeme vidět projekční alias je závislý na danné geometrii. Hlavně pak její orientaci vůči světlu a kameře. Proto bychom museli pro jeho snížení provádět netriviální analýzu modelů. Perspektivní alias však není ovlivněn scénou, pouze perspektivní deformací prostoru.

Schéma, které navrhuje Zhang [9], je kombinací logaritmického rozdělení prostoru C^{log} a uniformního rozdělení C^{uni}. Podle následující rovnice:

$$C_i = \frac{C_i^{log} + C_i^{uni}}{2} \tag{2.2}$$

Tuto rovnici je možné zobecnit parametrem λ, kterým je možné nastavit poměr mezi jednotlivými rozloženími. Toto umožňuje nastavit dělcí plochu přesně pro naši scénu. To může výrazně vylepšit kvalitu výsledých stínů, pokud například scéna nemá v přední části mnoho geometrie (není potřeba tolik rozlišení v shadow map textuře), ale naopak má mnoho geometrie v prostřední části. Rovnice je upravena následovně.

$$C_i = \lambda C_i^{log} + (1-\lambda)C_i^{uni}, 0 \leq \lambda \leq 1 \tag{2.3}$$

Přičemž nastavením λ do hraničních hodnot schéma změníme na čistě logaritmické/uniformní. Předchozí rovnice pak je vlastně specifický případ použití této pro $\lambda = 0.5$.

Takovéto schéma zaručuje konstantní rozložení perspektivního aliasu po celém pohledovém tělese.

Logaritmické schéma si klade za cíl rozložit perspetivní alias konstatně po celé hloubce obrazu. Tudíž můžeme postavit rovnici:

$$\frac{dz}{zds} = \rho,\ \rho = const. \tag{2.4}$$

Z čehož dostaneme integrací podle s, s předpokladem ideální situace, že plýtvat žádným prostorem v textuře shadow mapy, kdy $s[0,1]$:

$$s = \int_0^s \mathrm{d}s = \frac{1}{\rho}\int_n^z \frac{dz}{z} = \frac{1}{\rho}\ln\frac{z}{n} \tag{2.5}$$

S předpokladem, že $s[0,1]$ můžeme získat $\rho = \ln f/n$. Díky podpoře jediné nelineární transformaci podporované aktuálním hardware, tj. perpektivně-projekční $s = \frac{A}{z} + B$[9], musíme provést vzorkování logaritmické funkce v různých hloubkách. Čím více takovýchto vzorků je provedeno, tím samožřejmě získáme přesnější aproximace takové funkce. Vzorkování tedy můžeme provést v $z = \{C_i^{log}\}$:

$$s_i = s(C_i^{log}) = \frac{\ln \frac{C_i^{log}}{n}}{\ln \frac{f}{n}} \tag{2.6}$$

Toto schéma však nahustí dělící plochy poblíž near plane kamery příliš blízko a v těchto podprostorech bude minimum objektů. Proto v praxi používáme i uniformní rozložení definované následující rovnicí:

$$C_i^{uni} = n + (f-n)\frac{i}{m} \tag{2.7}$$

Díky nutnosti manuálně nastavit poměr mezi uniformním a logaritmickým rozložením řezů pohledovým tělesem se tato metoda nehodí jako metoda pro nepředpřipravené scény. Pro ideální stíny je potřeba, aby umělec chystající scénu nastavil vzádelnosti dělících ploch a rozlišení shadow map. A to pokud možno pro každý snímek nebo pomocí tzv. keyframes, ale zároveň s důrazem na to, aby se textury zbytečně nepřealokovávaly příšliš často.

2.3 Sample-Distribution Shadow Maps

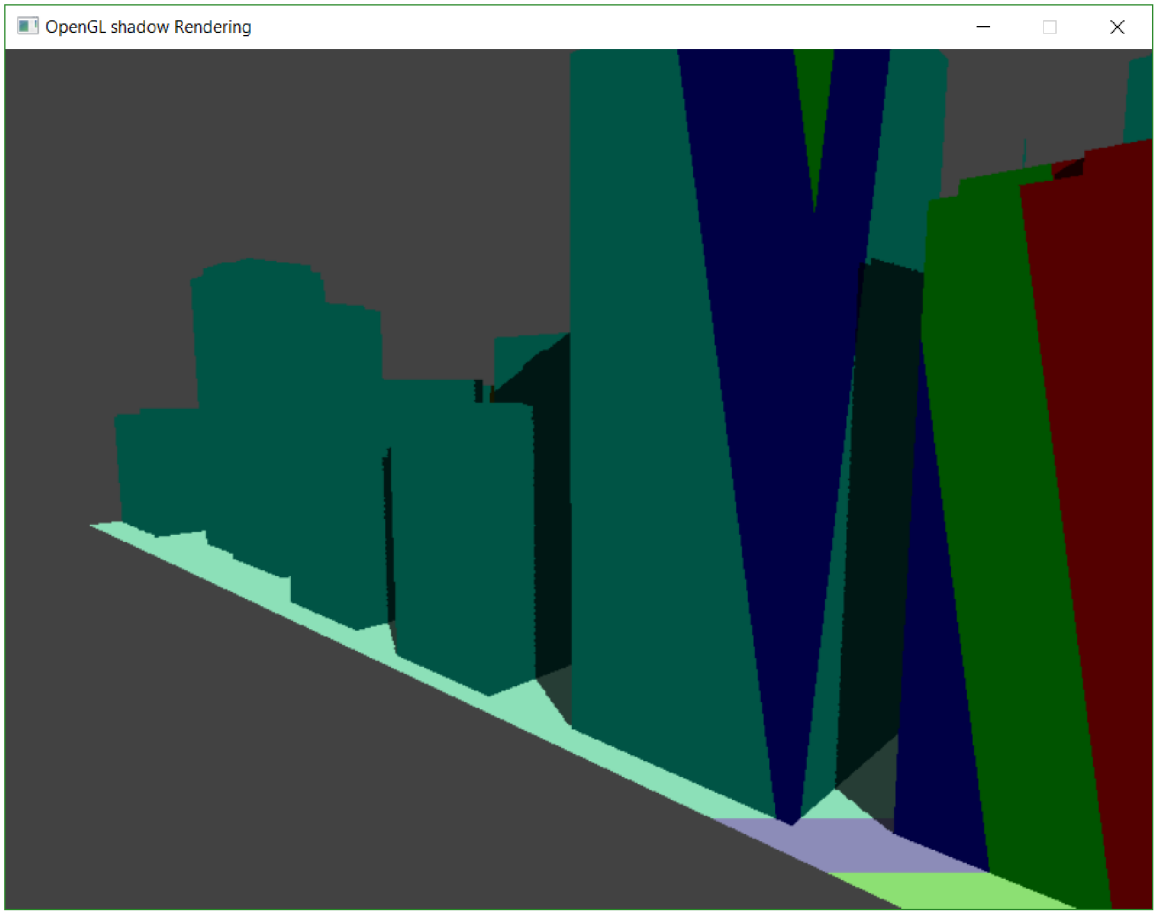
Tato metoda je velmi silně inspirovaná předchozí popisovanou a snaží se být jejím zobecněním a zároveň vylepšením. Její hlavní cíl je odstranit nutnost nastavení poměru mezi uniformním a logaritmickým rozdělením dle aktuální scény. Toho dosahuje analýzou depth bufferu předchozího snímku a dle počtu vzorků v určité hloubce rozmístí vzdálenosti jednotlivých řezů.

Prvním krokem, stejně jako v předchozí metodě, je výpočet vzdáleností k dělícím plochám. Oproti předchozí metodě můžeme ze vstupních informací zjistit opravdu těsnou near a far plochu. Zároveň můžeme mezi podprostory vytvořit mezery v místech, kde se nenachází žádné vzorky, které bychom měli zastínit. Toho dosáhneme min/max redukcí depth bufferu. Tímto minimalizujeme chybu způsobenou aliasingem tím, že zužitkujeme maximum texture space v shadow mapě.

Nejdříve si vypočítáme histogram hloubek z depth bufferu předchozího snímku. Z předchozího snímku ho počítáme, pokud víme, že kamera se nebude přemisťovat skokově nebo můžeme jeden snímek po přemístění kamery, který bude mít potencionálně vysoký aliasing v projekci stínů, zanedbat. Pokud před samotným vykreslením snímku provádíme Z-Pass zjednodušené scény, pak můžeme samozřejmě, a zároveň bychom měli, využít depth buffer právě z tohoto kroku.

Na obrázku 2.6 můžete vidět příklad použití metody SDSM na reálné scéně. Jak si můžete všimnout, nejbližší (červená) část pohledového tělesa obsahuje nejbližší budovy a to i přes vyšší vzdálenost od kamery. Pokud bychom použili metodu PSSM, tak by v nejbližší kaskádě neležel nejspíše žádný objekt, protože by obsahovala pouze volný prostor mezi kamerou a budovami.

Tuto metodu můžeme dále vylepšit tím, že provedeme tuto minimalizaci i v druhé ose. Tím vytvoříme opravdu minimální bounding box okolo daných vzorků, který následně obalíme pohledovým tělesem, ve kterém budeme renderovat shadow mapu. [3]



Obrázek 2.6: Ukázka SDSM na scéně, s prázdným popředím scény

Kapitola 3

Návrh aplikace

Ještě před samotnou implementací popsaných algoritmů, je nutné se zamyslet nad požadavky aplikace. To proto, abychom si mohli určit způsob, jakým chceme tyto požadavky naplnit. Tato kapitola vysvětlí čtenáři volby jednotlivých knihoven, nástrojů a přístupů, které jsem zvolil.

Základním požadavkem na aplikaci je možnost porovnat dvě metody výpočtu stínů v trojrozměrné scéně. Toto měření bude probíhat na dvou úrovních - výkonnostní a kvalitativní. Protože každý z těchto aspektů vyžaduje rozdílný způsob používání aplikace, bude potřeba aplikaci moci spustit s jinými rozhraními. Další z bodů zadání vyžaduje možnost předdefinovaného průletu scénou po trase, tudíž i tento požadavek musí výsledná implementace zohlednit. Dle mne by aplikace také měla sloužit k názorné prezentaci rozdílů mezi oběma metodami, proto i toto bude zohledněno ve výsledné implementaci. Míra aliasu stínů je také závislá na dané scéně, jak jsme si mohli přečíst v odstavci 2.2.1, proto by bylo vhodné, mít nějaký jednoduchý způsob, jak načíst rozdílné scény.

První strukturou, kterou musíme navrhnout v takovéto aplikaci, je reprezentace samotné scény. Nejjednodušší metodou reprezentace scény je stromová struktura. Na toto má standardní knihovna C++ dostatek nástrojů a proto jsem si zvolil tuto cestu. Tato struktura by měla dokázat dle daných parametrů vykreslit scénu a také podporovat frustum culling. Implementaci nám zlehčuje i to, že výsledná scéna nemusí být dynamická. Tudíž jediný požadavek na strukturu je kvůli frustum cullingu to, aby rodičovský prvek stromu měl AABB box, který bude obalovat všechny jeho potomky.

Dalším důležitým bodem, který musíme vyřešit, je vytváření a správa shaderů. Vzhledem k tomu, že tato práce bude zaměřena právě na programování shaderů, je nutné mít dobrý způsob jak je vytvářet, definovat a spojovat jednotlivé části do ucelených celků. Také díky absenci direktivity `#include` v jazyce GLSL je vhodné vyřešit způsob, jak mezi jednotlivými programy v tomto jazyce sdílet datové struktury a společné funkce. Důležitým přínosem hromadné správy shaderů je možnost je centrálně vyměnit za jiné za běhu programu nebo kontrolovat počet změn stavů pipeline během tvoření fronty příkazů pro vykreslení scény.

3.1 Rozhraní

Pro názornější prezentaci aplikace budeme potřebovat nějaký způsob, jak uživateli umožnit ovládání aplikace. Pro jeho implementaci jsem zvolil volně dostupnou knihovnu ImGUI distribuovanou pod MIT licencí. Tato knihovna umožňuje mimo jiné vytvářet plovoucí

Úroveň	Barva	RGB
0	Červená	#FF0000
1	Zelená	#00FF00
2	Modrá	#0000FF
3	Tyrkysová	#00FFFF
(Mimo rozsah)	Žlutá	#FFFF00

Tabulka 3.1: Tabulka barev reprezentující jednotlivé řezy pohledovým tělesem

panely s ovládacími prvky. Další funkcí, kterou v aplikaci použiji pro přehlednější prezentaci, je vykreslení textury uložené v paměti grafické karty v plovoucím panelu.

3.2 Dvě kamery

Pro názornější prezentaci obou metod bude vhodné mít možnost použít mimo hlavní kamery, ke které se dělící metody vztahují, ještě jednu. Tak budeme moci sledovat detaily stínů, pozici pohledového tělesa hlavní kamery nebo pozici jednotlivých dělících ploch. Tyto plochy mají přiřazené barvy, které jsou definovány v tabulce 3.1.

3.3 Vizualizace

V metodě SDSM se mohou měnit pozice dělících ploch. Proto bude pro názornost užitečné mít možnost nějak sledovat pozice těchto ploch a vztahovat je k počtu pixelů, nacházejících se v dané hloubce pohledového tělesa.

Kapitola 4

Implementace

Následující kapitola má za cíl čtenáři popsat některé implementační detaily a nástrahy, které během vývoje aplikace nastaly nebo stojí za mínku.

Aplikace byla napsaná v jazyce C++ ve standardu C++14. Projekt byl kompilován kompilátorem MSVC 2015 distribuovaném spolu s IDE Microsoft Visual Studio 2015, v operačním systému Windows 10. Celá aplikace je psaná platformně nezávisle a nepoužívá žádné platfomně závislé knihovny, proto by neměl být žádný problém ji přeložit i pod jiným OS.

4.1 Vytvoření kontextu

Pro vytvoření okna a inicializaci kontextu jsem využil kódu poskytnutého vedoucím práce Ing. Josefem Kobrtkem. Šablona využívá knihovny SDL2 pro odstítnění od platformních závislostí okenního systému. OpenGL kontext je inicializovaný ve verzi 4.5.

4.2 Frustum culling

Frustum culling je technika snižující počet příkazů posílaných po sběrnici z CPU do grafické karty tím, že analyzujeme scénu a vykreslujeme pouze tu geometrii, která je ve výsledném pohledovém tělese opravdu vidět[10]. Tato technika má dva efekty, kterými může přispět ke zrychlení procesu renderu. Zaprvé se uvolní komunikační pásmo na sběrnici a zadruhé se odstraní nutnost rasterizovat mnoho pixelů, které nijak nezasáhnou do výsledné scény. Grafická karta musí spustit rasterizaci i pro geometrii, jejíž vrcholy jsou mimo pohledové těleso, protože samotné pixely by mohly do frusta zasáhnout.

Má implementace techniky frustum culling je nejzákladnější variantou a proto nemusí být zcela efektivní. Důvod proč jsem zvolil právě tuto je nejen její jednoduchost, ale i jistota, že se neprovede ořezávání geometrie, která by do výsledné scény zasáhla. Naopak však může dojít k rasterizaci nepotřebné geometrie (overdraw). Funguje na základě kontroly kolizí dvou koulí obalující AABB (axis aligned bounding boxe) scény a geometrie, kterou chceme kreslit. Tato kontrola je výpočetně nenáročná. Jedná se pouze o kontrolu, zda je vzdálenost středů koulí větší, než součet jejich poloměrů.

Tato optimalizace zrychlila dobu vykreslování snímků na testovací scéne "Desert city" ze 4000ms na 66ms. Jak můžete vidět, dopad této metody na výslednou rychlost vykreslování je zásadní.

4.3 Generování kaskády shadow map

Ač implementujeme dvě rozdílné metody generování kaskády stínových map, můžeme říct, že mají společný celý proces vykreslení. Liší se pouze ve způsobu, jakým se počítá umístění jednotlivých dělicích ploch. Samotný výpočet light view-projekčních matic probíhá stejně, ať se jedná o PSSM nebo SDSM.

Pro vykreslení shadow map si musíme nejdříve vytvořit FBO (framebuffer object). V OpenGL je to typ bufferu, ke kterému můžete nalinkovat jednotlivé textury nebo renderbuffery (tzv. attachements), kterým definujete jejich úlohu při vykreslování, např. color, depth nebo stencil. V našem případě si vystačíme s hloubkovým (depth) bufferem. V našem případě nepotřebujeme vykreslovat barevnou komponentu, proto ji můžeme příkazem `glDrawBuffer(GL_NONE)` vypnout.

Původní práce navrhuje postup vykreslování, při které se vykreslí první část kaskády, následně další a tak dál.

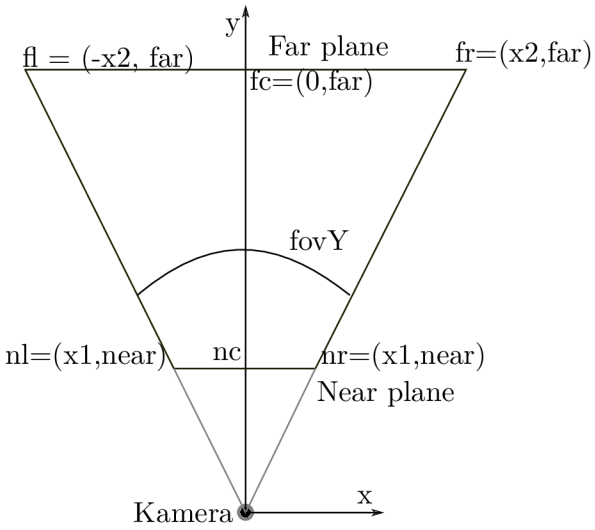
Pro generování kaskád shadow map je vhodná funkcionalita OpenGL umožňující práci s vrstvenými texturami. Poté nemusíme vykreslovat jednotlivé úrovně oddělenými draw cally, ale můžeme vykreslit vše, co by mělo být viditelné z pohledu světla. Odpovídající vrstvu poté můžeme určit pomocí vestavěné proměnné `gl_Layer`[1]. Samotné vykreslení objektů pro výpočet shadow map je zjednodušené o veškeré vlastnosti materiálu, které neovlivňují jeho siluetu. Díky vykreslování do vrstev se nemusí provádět celá pipeline shader programu zvlášť pro každý model a každou úroveň, ale do uniformní proměnné si uložíme view-projection matici. Poté provede pro každý model, který bude vrhat stíny právě jeden draw call, vertex shader se provede právě jednou, a až v geometry shader se model pošle do rasterizátoru pro každou úroveň zvlášť s příslušnou view-projection maticí.

4.4 Výpočet view-projection matic

Postup výpočtu view-projekčních matic je podobný, jako u běžného shadow mappingu. Proto prvně popíšu tento postup a na konci si popíšeme rozdíly.

To co ideálně vyžadujeme od tohoto výpočtu, je definici pohledového tělesa, těsně obalující pohledové těleso naší kamery a začínající v bodě kde se nachází světlo. Abychom tohoto dosáhli musíme nejdříve získat pozice rohů našeho pohledového tělesa v souřadnicovém systému světa (world space). Tento výpočet je znázorněn na následujícím diagramu 4.1. Diagram je zjednodušený do plochy XY. Vše, co známe o pohledovém tělese a kameře, je pozice kamery (označme si tento jako `camPos`), vektor směřující směrem z kamery dopředu `viewVec` a vektro směřující obrazem přímo nahoru, který je kolmý na předchozí vektor. Tomuto vektoru říkejme `upVec`. Dále známe vzdálenost k ploše tvořící nejbližší vykreslované objekty `near` (označované v anglické literatuře jako near plane) a vzdálenost k zadní ploše pohledového tělesa `far` (far plane). Dále známe poměr stran obrazu `aspect` a šířku záběru `fovY`.

[1] https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/gl_Layer.xhtml



Obrázek 4.1: Pohledové těleso zobrazené na XY ploše

Ze schématu 4.1 lze odvodit následující rovnice:

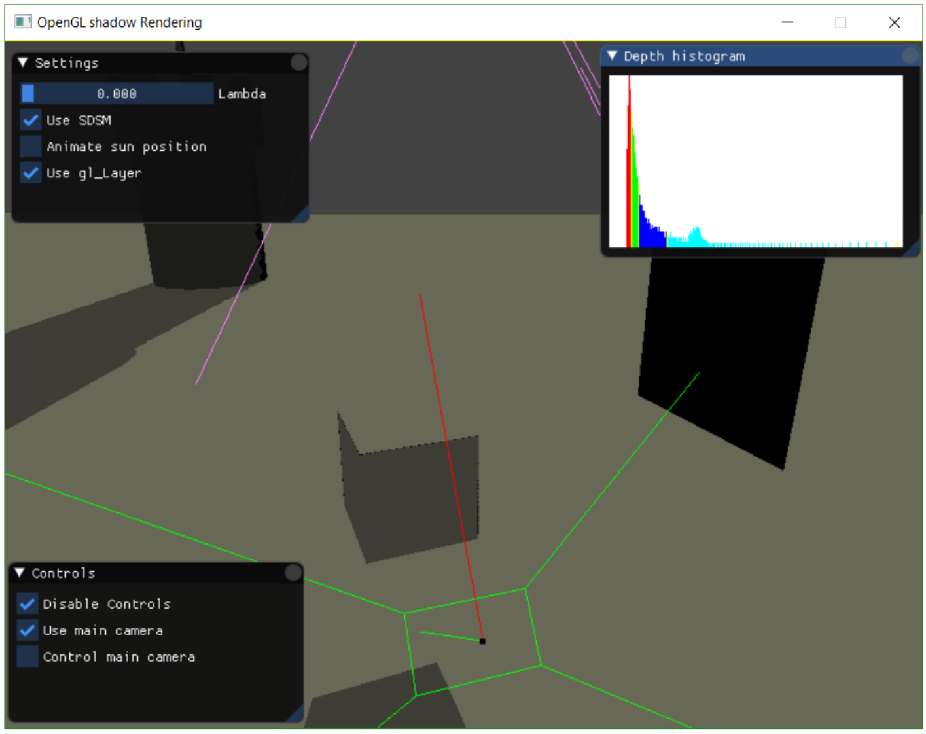
$$\begin{aligned} \tan(\frac{fovY}{2}) = \frac{x_1}{near} &\implies x_1 = near * \tan(\frac{fovY}{2}) \\ \tan(\frac{fovY}{2}) = \frac{x_2}{far} &\implies x_2 = far * \tan(\frac{fovY}{2}) \end{aligned} \tag{4.1}$$

Z těchto dat můžeme spočítat pozice jednotlivých rohů následovně:

$$\begin{aligned} xn\delta &= near * \tan(fovY/2) \\ yn\delta &= near * \tan((fovY * aspect)/2) \\ xf\delta &= near * \tan(fovY/2) \\ yf\delta &= near * \tan((fovY * aspect)/2) \\ nc &= \vec{camPos} + \vec{viewVec} * near \\ fc &= \vec{camPos} + \vec{viewVec} * far \\ left &= \vec{upVector} * \vec{upVector} \\ nrt &= nc + \hat{left} * xn\delta + \hat{upVector} * yn\delta \\ frb &= nc - \hat{left} * xf\delta - \hat{upVector} * yf\delta \\ &\dots \end{aligned} \tag{4.2}$$

Body jsou značeny tak, že **n** v názvu znamená near plane, **f** far plane, **r** značí pravou stranu, **l** levou stranu. Pro určení zda se jedná o horní panle se používá písmeno **t** a pro spodní **b**. Zbytek bodů lze jednoduše dopočítat.

Nyní když známe souřadnice okrajů pohledového tělesa můžeme hledat view matici pohledového tělesa světla. Protože v této práci používáme pouze směrové světlo reprezentující slunce, můžeme tento problém zjednodušit na hledání AABB boxu obalující těchto osm bodů transformovaných do souřadnicového systému světla (light space). Takto získaný AABB box nám pak udává hranice ortogonálního pohledového tělesa. Pozici kamery



Obrázek 4.2: Ukázka GUI aplikace

pro pohledovou matici můžeme získat buď výpočtem z pevně stanovené plochy, na které se světlo nachází nebo hledáním nejvzdálenějšího objektu vrhajícího stín a zároveň se nacházejícího v AABB boxu, který jsme našli a prodloužili do nekonečna směrem ke zdroji světla. Tímto postupem můžeme velmi přesně ohraničit pohledové těleso bez toho, abychom ztráceli přesnost hloubkové mapy.

4.5 Grafické rozhraní

Uživateli jsem pomocí uživatelského rozhraní umožnil upravovat parametry právě probíhajícího vykonávání prezentace pomocí několika panelů. Zde můžete vidět jejich popis.

Z panelu *Settings* je možné ovládat argument λ pro výpočet dělících ploch C_i, vypnout nebo zapnout animaci slunce a použití vykreslování po vrstvách. Také je zde možné přepínat mezi jednotlivými metodami. Při přepnutí na metodu SDSM se zobrazí panel *Depth histogram*. Tento panel zobrazuje histogram pro depth mapu, která je obarvená barvami dle tabulky 3.1. Všechny panely můžete vidět na obrázku 4.2.

Poslední panel obsahuje ovládání kamer. Umožňuje vypnout jejich ovládání pro jednodušší práci s rozhraním GUI, kdy můžeme nastavit scénu dle uvážení a poté upravovat parametry v rozhraní. Umožňuje také přepnout do debugové kamery a v **Debug** konfiguraci.

4.6 Překladové cíle a define

V přiloženém projektu můžete nalézt 3 hlavní konfigurace překladu - Debug, Release a Speed. Každá z konfigurací má jinak nastavenou úroveň optimalizace, předpřipravené macro definice a výsledná aplikace má rozdílné ovládání.

Debug

Pro jednoduší vývoj aplikace mimo systému definic shaderů popsaného v dodatku Bm implementuje periodický update shaderů dle zdrojových souborů na disku každou sekundu nebo čas definovaný ve třídě `C_ShaderManager`. Toto umožňuje prototypovat shadery bez nutnosti vypnutí aplikaci. V této konfiguraci byla také vytvořena sada debugových funkcí, které je možno použít a jsou automaticky nahrazeny za prázdná makra nebo ,implementace a tak jsou jejich volání v finální aplikaci optimalizátorem kompletně vymazána z vygenerovaného strojového kódu.

Jedná se o makro `ErrorCheck()`, které volá OpenGL funkci `glGetError()` umožňující odhalit chybná volání knihovního API. A o celou třídu class `C_DebugDraw`, která umožňuje vykreslování 3D primitiv.

Speed

V této verzi byla aplikace měřena, neobsahuje žádné kontrolní výpisy, GUI. V této verzi se neobnovují shadery a negenerují debugové textury. Zároveň tato konfigurace má zaplé takzvané "Link time optimalizace"spolu s nejagresivnějším optimalizátorem kódu. Toto by mělo zaručit co nejoptimálnější kód na straně procesoru, tak abychom při měření mohli vyloučit, že aplikaci spomaluje procesor.

Kapitola 5

Měření

Tato kapitola rozebírá zvolenou metodiku měření a odůvodnění její volby. Následuje přehled použitých počítačů a následně samotné hodnoty.

5.1 Metodika měření

Vzhledem k použití multitarget renderingu, je zajímavé porovnat, jak tuto funkcionalitu implementují jednotliví výrobci grafických karet. Bohužel z největších výrobců, musím vyřadit Intel, jehož grafické karty, nepodporují funkci `glCreateQueries`, kterou sem využil. I přes delší snahy, se mi nepodařilo najít, zda jde o nějaký chybný výklad standardu OpenGL z mé strany, nebo na straně implementace OpenGL v ovladači na mém počítači.

Díky tomuto zjištění, mi přijde zajímavé porovnávat nejen algoritmy, ale i výkon mé implementace na grafických kartách jednotlivých výrobců grafických karet. Zároveň nás může zajímat porovnání jednotlivých řad modelů grafických karet, jednotlivých výrobců.

Jednotlivé parametry obou metod se dají rozdělit mezi měřitelné a neměřitelné. Mezi měřitelné můžeme zařadit datovou náročnost (jak na straně CPU tak na straně GPU) a průměrný interval mezi vykresleními jednotlivých snímků. Z neměřitelných lze uvést počet chybně vykreslených pixelů na jednotlivých snímcích.

Průměrnou délku snímku bych chtěl měřit na dvou úrovních. Kontrolní bod programu bude prohození back a front bufferu na grafické kartě. Jedna z cest měření je využití co nejpřesnějšího měření přímo v C++ programu. Toto však není opravdová doba, kterou grafické kartě zabere vykreslení snímku, ale doba, po kterou procesor chystá render příkazy pro grafickou kartu, která je poté vykonává. Tato doba nás může zajímat také, protože čím rychleji jsme schopni připravit tuto frontu pro grafickou kartu, tím více času ze snímků můžeme věnovat ostatním výpočtům eg. fyzikální výpočty, AI, logiky aplikace etc. Ovšem opravdový čas již není tak jednoduché měřit, pro tento účel bych rád využil externího programu, nejlépe od samotného výrobce dané grafické karty.

Datovou náročnost bych mohl měřit také, ale vzhledem k tomu, že obě implementace využívají skoro stejné struktury, tak by toto měření bylo bezpředmětné. Jediný rozdíl je textura použitá v metodě SDSM pro uložení hloubkového bufferu. Předem známe velikost shadow map a jejich počet tudíž je tento parametr konstantní pro obě metody.

Počet chybně vykreslených pixelů sice nelze měřit, ale chybu lze vypočítat: vložit výpočet.

Samotné měření proběhlo pomocí funkce OpenGL query. Ta funguje na principu přidání požadavku na zjištění aktuálního času na grafické kartě do fronty volaných funkcí knihovny

Jméno	Význam
Frame ID	Identifikátor snímku
progress	Vyjadřuje postup předdefinovanou scénou číslem normalizovaném do intervalu $< 0, 1 >$
DrawCall	Počet vykreslených objektů ve daném snímku
Cascade render	Doba potřebná pro vykreslení všech kaskád
Render	Doba vykreslení finálního scény

Tabulka 5.1: Popis hodnot sledovaných pro algoritmus PSSM

OpenGL. Ve chvíli, kdy grafická karta svým zpracováním dojde k tomuto požadavku tak ho uloží do dříve vygenerovaného "query objectu"[1]. Pokud chceme v kódu, zpracovávaném na procesoru, zjistit hodnotu vyžádáme si tuto informaci. Na tuto informaci ovšem musí procesor počkat a to v případě, že náš požadavek ještě nebyl grafickou kartou vykonán. Tudíž dochází k aktivnímu čekání procesoru na grafickou kartu.

Proto jsem implementoval statistický kód tak, aby se takovémuto čekání co nejvíce předcházelo. V moderních grafických aplikacích, je běžné, že aplikace vytváří frontu OpenGL funkcí několik snímků dopředu. Pokud bychom po každé části vykreslení zastavili vykonávání kódu na CPU, přišli bychom o tuto výhodu a po získání výsledku našeho dotazu, bychom na několik desítek dalších instrukcí na CPU, pozastavili vykonávání kódu na GPU.

Proto první metodou, kterou jsem použil pro měření výkonu aplikaci, bylo čekání až po dokončení celého snímku. Toto však způsobí stejný problém jako dříve, ovšem až po výměně front a back bufferu grafické karty. Proto jsem došel k metodě, při které měření provádím tak, že měřím zároveň tři snímky. Započnu vykonávání prvního snímku a do cyklického pole si uložím první pole požadavků, které jsem během tohoto snímku uskutečnil. Před započetím dalšího snímku si vyžádám výsledky požadavků, které se v cyklickém poli nacházejí na další pozici.

Výsledné hodnoty aplikace vypisuje do souboru ve formátu CSV do souboru pojmenovaném *zadanáCesta-X-S.csv*, kde X je hodnota argumentu zvolené vykreslovací metody a S je název definičního souboru použitého shaderu bez přípony.

Pro lepší získání přehledu o tom, která část vykreslování zabrala nejdelší dobu a kde vzniká rozdíl mezi metodami, aplikace vypisuje pro každou metodu nejen délku každého snímku během po předdefinované cestě, ale i několik další informací. V tabulkách 5.1 a 5.2 můžete vidět jednotlivé hodnoty s popisky.

5.2 Použité přístroje

Pro měření jsem si vybral stroje reprezentující aktuální generaci techniky. Také jsem byl samozřejmě nucen vybírat z hardware, ke kterému jsem měl v době psaní této práce přístup. Přes to se mi povedlo vybrat jeden stroj s grafickou kartou AMD, jeden počítač s instalovanou špičkou v oblasti grafických karet pro běžné používání GTX 1080 Ti. Dále jsem vybral další dva stroje s grafickými kartami střední třídy a jednu notebookovou grafickou kartu. Všechny testy probíhaly na operačním systému Windows 10 a na daných strojích bylo vypnuto maximum ostatních aplikací tak, aby výsledky byly co nejméně zkresleny jejich vlivem. Tímto jsem se pokusil pokrýt většinu spektra používaných karet, na kterých

[1] https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glQueryCounter.xhtml

Jméno	Význam
Frame ID	Identifikátor snímku
progress	Vyjadřuje postup předdefinovanou scénou číslem normalizovaném do intervalu $< 0, 1 >$
DrawCall	Počet vykreslených objektů ve daném snímku
z-pass	Vykreslení pouze hloubkové mapy z pohledu kamery
min-max-reduction	Výpočet histogramu a vypočítání pozic dělících ploch
Cascade render	Doba potřebná pro vykreslení všech kaskád
Render	Doba vykreslení finálního scény

Tabulka 5.2: Popis hodnot sledovaných pro algoritmus SDSM

Jméno	Grafická karta	Grafická RAM	Proceso	RAM
PC1	Nvidia GeForce GTX 1080Ti	12GB	i7-4790K 4.00GHz	32GB
PC2	Nvidia GeForce GTX 970	2GB	i7-4790 3.60GHz	16GB
PC3	AMD Radeon R9 200	4GB	i7-4790 3.60GHz	16GB
PC4	Nvidia GeForce 840M	2GB	i5-4210M 2.60GHz	8GB

Tabulka 5.3: Tabulka použitých počítačů k testování aplikace

by tyto metody mohly být používané. Seznam a parametry jednotlivých strojů najdete v tabulce 5.3. Jméno z tabulky odpovídá složce v adresáři *results* na přiloženém CD.

V průběhu měření jsem však narazil na několik problémů. První byl, že při měření na PC3 se aplikaci nepodařilo správně načíst modely a tudíž kamera prolétá prázdnou scénou. Bohužel na tomto stroji, jsem neměl přístup k debugovým nástrojům a nemohl na ní dlouhodobě práci ladit, ani na jiném stroji s kartou značky AMD. Proto tato měření nemají žádnou vypovídací hodnotu a nebudou v této práci dále zkoumána.

5.3 Jednotlivé scény

Pro měření výkonu jednotlivých metod jsem zvolil tři volně dostupné scény. Všechny tři jsem připravil dle popisu v příloze D tak, že ve scéně se nachází jediný model. Ten je označen (implicitně) jako vrhač stínu (shadow caster) a umístěn do souřadnic $(0, 0, 0)$. Scény byly vybraány tak, aby otestovaly možné nedokonalosti. Prnví scénou je pouštní město (*bin/models/DesertCity.xml*). Tato scéna obsahuje rovný rozsáhlý terén a nízké, většinou kvádrové, budovy. Tato scéna je složena z 536326 trojůhelníků, které mají jednoduchou geometrii. Na této scéně, jde vidět nutnost frustum culling optimalizace, která byla popsána v kapitole 4.2.

Další scéna se jmenuje Sehir (*bin/models/Sehir.xml*). Tato scéna obsahuje blok mrakodrapových budov evokující oblast Manhattanu. Tato scéna ukazuje důvod, proč musí být pohledové těleso pro vykreslení shadow map zvětšeno směrem ke zdroji světla. Pokud by pouze těsně obepínalo část pohledového tělesa, pro které chceme shadow mapu generovat, pak bychom uprostřed úzkých uliček, kde jsme obklopeni výškovými budovami, neviděli stíny budov mimo naše pohledové těleso. V této scéně se nenachází žádná složitější geomterie a je složena pouze z 6896 trojůhelníků.

Jméno	Definiční soubor	Definiční soubor cesty	Délka cesty[s]
City	models/City.xml	CityPth1.xml	25s
Desert City	models/DesertCity.xml	DesertPth1.xml	50s
Sehir	models/Sehir.xml	SehirPth1.xml	25s

Tabulka 5.4: Tabulka scén použitých k testování.

Poslední scéna pojmenovaná City (*bin/models/City.xml*) obsahuje futuristicky vzhlížejících budov s velmi složitou geometrií. Tato scéna je větších rozměrů a ulice zde jsou více otevřené. Má méně samostatných objektů (15154 trojůhelníků), ovšem tyto objekty mají složitější geometrii. Tato scéna má být komplexním prověřením mé implementace, mnoho převisů a daleké výhledy zde vyžadují opravdu precizní výpočet stínů do výsledného obrazu.

V tabulce 5.4 můžete vidět přehled těchto scén, korespondujících cest a jejich délek, tak jak byly použity pro testování[2].

5.4 Naměřené hodnoty

Během měření vzniklo více dat, než je možné prezentovat zde na stránkách této práce. Zároveň by to bylo zbytečné, protože výsledky se velmi podobají. Proto si zde rozebereme jen nejzajímavější hodnoty. Veškerá naměřená data je možno najít na přiloženém disku.

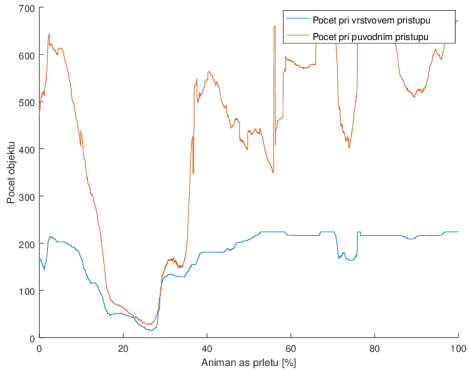
První měřenou hodnotou je počet vykreslených modelů. Ty jsou měřeny jako takzvané *draw cally*. Tímto termínem myslíme počet volání funkcí, které provádí samotné vykreslení. Každá scéna byla měřena dvakrát. Jednou vykreslovaná s optimalizací pospanou v sekci 4.3 pomocí OpenGL funkce vykreslování po vrstvách. Druhý průchod scénou je pak měřen s původní metodou. Všechny scény byly změřeny na jednom stroji, protože výsledky nejsou závislé na daném hardware, ale pouze na implementaci algoritmu. Počet draw callů se mění spolu s průchodem scénou díky frustum culling, popsaném v sekci 4.2, tudíž jejich počet budeme sledovat v závislosti na postupu cestou. Tento postup je normalizovaný do intervalu $< 0, 1 >$.

Jak můžeme vidět, dle očekávání se počet vykreslovaných modelů při použití vrstvového vykreslování snížil výrazně. Přesná čísla můžete vidět v tabulce 5.5. Zároveň z této tabulky lze vidět, že mnohem výraznější zlepšení nastalo v případě metody PSSM. Toto lze vysvětlit zaprvé nedokonalostí implementace frustum cullingu, ale také tím, že již samotná metoda napomáhá vykreslování jen té geometrie během výpočtu stínů, která má opravdový dopad na výslednou scénu.

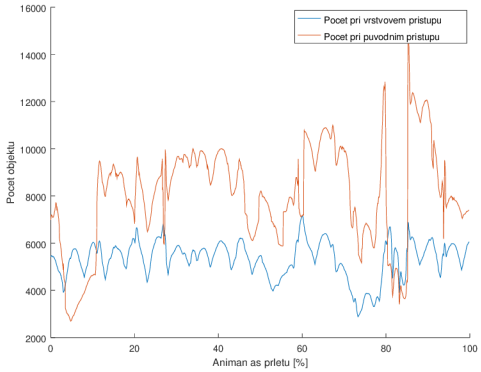
V těchto grafech můžeme najít intervaly stojících za zmínku. První, na který bych se rád zaměřil, se nachází v grafu korespondujícím ke scéně Sehir.xml a odpovídá přibližně intervalu $< 0.7, 0.8 >$. Zde se křivka průběhu bez vrstvového přístupu zvyšuje výrazně rychleji než křivka odpovídající vrstvovému přístupu. Toto je dáno tím, že pokud se v daném snímku ocitne do prostoru, ve kterém vykreslujeme shadow mapy, dvakrát více objektů, pak v přístupu bez vykreslování ve vrstvách proběhne čtyřikrát více vykreslování.

V tom samém grafu můžeme najít další zajímavý interval a to přibližně $< 0.35, 0.50 >$. Zde se několikrát stane, že křivka průchodu, při kterém vykreslujeme každou kaskádu zvlášť, klesne pod křivku průchodu s vrstvovým vykreslováním. Toto by mohlo čtenáře vést na myšlenku, že ve specifických případech může být generování každé kaskády zvlášť optimálnější co do počtu draw callů. Ovšem analýzou pozice kamery v této části průchodu scénou zjis-

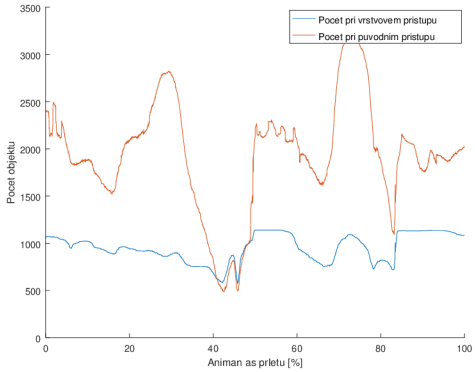
[2]Cesty jsou uváděny relativně k spustitelnému souboru, ten se nachází ve složce bin.



Obrázek 5.1: Počet draw callů při průchodu scénou City.xml



Obrázek 5.2: Počet draw callů při průchodu scénou Desert.xml



Obrázek 5.3: Počet draw callů při průchodu scénou Sehir.xml

Scéna	PSSM	SDSM
City	47.278%	41.657%
Desert City	37.927%	12.935%
Sehir	49.0073%	26.486%

Tabulka 5.5: Tabulka procentuálního snížení počtu draw callů u jednotlivých metod a scén použitím vykreslování po vrstvách

Metody	City	Desert	Sehir
SDSM / SDSM+gl_Layer	0.74397	0.70471	1.044
PSSM+gl_Layer / PSSM+gl_Layer+PCF	0.96751	0.99845	0.97761
SDSM / PSSM	0.91573	0.99695	0.79693

Tabulka 5.6: Poměry průměrncých dob vykreslení snímků jednotlivých metod měřených na PC1 v různých scénách

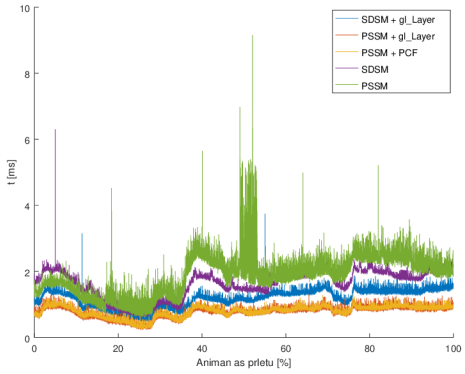
tíme, že právě kamera opouští scénu a v záběru je minimum objektů. Důvodem, proč zde vyjde lépe tato metoda je, že implementace frustum cullingu v této práci je neoptimální. Díky tomu, že nedochází k přesnému prořezávání, ale pouze na základě koule opsané okolo AABB boxu objektu nebo pohledového tělesa, je tato metoda úspěšnější v metodě, která vykresluje kaskády zvlášť. Pokud totiž opisujeme koule okolo menších pohledových těles místo jednoho velkého, výsledný součet objemů koulí je menší, a tak se v nich nachází menší počet objektů.

Závěr tohoto měření tedy je, že co do počtu draw callů vychází obecně lépe přístup s vykreslováním po vrstvách. Pokud bychom implementovali přesné ořezávání scény, rozhodně by vycházelo lépe použít toto.

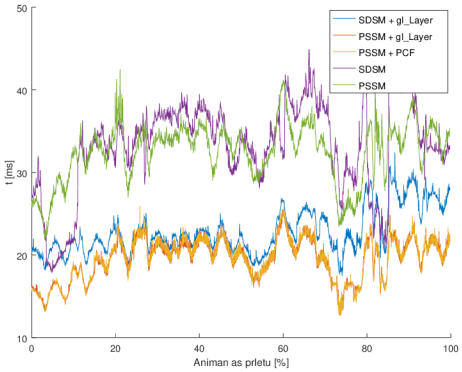
Další z měřených parametrů je délka vykreslení jednoho snímku (frame time). Tento čas udávám v milisekundách. Na každém z měřených počítačů jsem měřil na každé scéně pět průchodů stejnou trasou. Každá trasa má v tabulce 5.4 pevně přiřazený čas, za který má být scéna projita. Díky rozdílné době, kterou zabralo vykreslení jednoho snímku každé metodě, nemohu vztahovat hodnoty vůči danému snímku. Namísto toho následující grafy, vztahují dobu vykreslení k části trasy, na které se nacházela kamera. Tím můžeme v následujících grafech vidět, jak si jednotlivé části trasy vedly s vykreslením stejné geometrie a stejnému nastavení slunce.

Jak si na grafech průchodu scénami můžeme všimnout, výsledné hodnoty jsou v očekávaných poměrech. Nejrychlejší metodou je implementace PSSM spolu s vykreslováním shadow map po vrstvách. Zajímavé je, jak korelují křivky této implementace a implementace PSSM spolu s PCF vylepšením. Detailnější přehled o této podobnosti lze získat v tabulce 5.6. Toto je nejspíše dáno tím, jak grafická karta úspěšně cachuje shadow mapu. Zde nejspíše dochází k vysokému cache hit rate díky dobré lokalitě přístupu do paměti. To znamená, že při přečtení prvního ze čtyř pixelů, ze kterých nakonec průměrujeme informaci zda je pixel zastíněn, dokáže grafická karta úspěšně uložit okolní body v textuře do cache. Při čtení zbylých tří bodů tak čteme z rychlé paměti namísto z grafické RAM nebo hůře ze sdílené paměti.

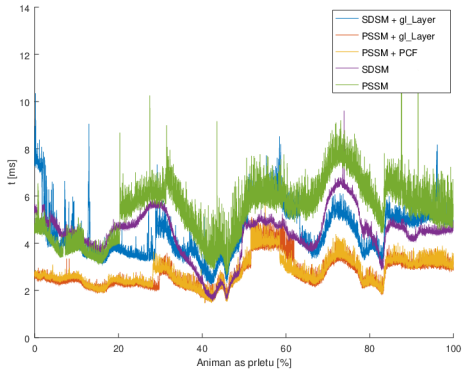
Dalším údajem, jež z měření můžeme vyčíst, jsou procentuální doby, jaké zabíralo provedení jednotlivých kroků obou metod na grafické kartě. Tyto statistiky jsou mnohem detailnější pro metodu SDSM, protože zde je nutno provést více kroků k výpočtu dělících ploch. Výsledky můžete vidět v tabulce 5.7.



Obrázek 5.4: Graf výkonu během průchodu scénou City.xml na PC1



Obrázek 5.5: Graf výkonu během průchodu scénou Desert.xml na PC1



Obrázek 5.6: Graf výkonu během průchodu scénou Sehir.xml na PC1

Počítač	Metoda	Scéna	ZPass	Min-max	Výpočet stínů	Vykreslení
PC1	SDSM	City	19.882%	9.9201%	34.361%	35.837%
		Desert	21.817%	1.3277%	47.294%	29.562%
		Sehir	22.147%	4.4709%	37.053%	36.329%
	PSSM	City	x	x	46.582%	53.418%
		Desert	x	x	63.256%	36.744%
		Sehir	x	x	50.018%	49.982%
PC2	SDSM	City	19.479%	9.1880%	46.067%	25.266%
		Desert	20.677%	1.0395%	50.121%	28.162%
		Sehir	22.218%	3.1958%	41.166%	33.421%
	PSSM	City	x	x	62.556%	37.444%
		Desert	x	x	66.470%	33.530%
		Sehir	x	x	51.916%	48.084%

Tabulka 5.7: Poměry průměrncých dob prováděných jednotlivých částí algorimtů u různých metod a počítačů

V tabulce můžete vidět výrazný rozdíl mezi PC1 a PC2 při použití metody SDSM v poměru doby výpočtu stínů a vykreslení finální scény. Nejvýraznější je tento rozdíl u scény City. Toto by šlo přiřknout rozdílné implementaci použitých operací v driverech daného zařízení. I díky tomu, že předchozí části (ZPass, Min-max reduction) se nijak výrazně nezměnili. Bohužel však až během vyhodnocení výsledků jsem zjistil, že do času výpočtu stínů se započítává i možné čekání CPU na informaci z předchozích kroků. Proto nemůžeme s přesností říct, čím je tento zásadní rozdíl způsobený.

Kapitola 6

Závěr

Čtenář se dozvěděl o vývoji vedoucím k metodám Paralel-Split Shadow Maps a Sample Distribution Shadow Maps a nedostatcích předchozích metod, které vedly k vývoji těchto metod. Zároveň se dozvěděl o způsobech, jak zmírnit nedostatky uvedených metod a o možnostech implementace filtračních metod ve spojení s uvedenými metodami.

Měřením jsme zjistili, že navržený přístup vykreslování shadow map do vrstvových textur je většinou rychlejší, než vykreslování každé úrovně kaskády. Ovšem připravení se o možnost použití jednodušších modelů v závislosti na vzdálenosti od pozorovatele (LOD[4]), může být obrovskou nevýhodou. V kapitole 5 se čtenář mohl dozvědět o výkonnostních rozdílech mezi jednotlivými metodami. Z těchto výsledků vyplývá, že ztráta výkonu při použití metody SDSM není příliš vysoká. Pohybovala se mezi 0-4% při využití vykreslování do vrstvové textury a 1-20% při vykreslení každé vrstvy zvlášť. Dále jsme došli ke zjištění, že použití PCF mělo minimální dopad na výkon obou metod.

Implementačně je metoda Sample Distribution Shadow Maps o něco složitější. Avšak rozdíl je minimální.

Výsledná aplikace by jistě mohla být stále vylepšena. Variabliní počet kaskád v obou metodách by měl jít lehce docílit za pomoci přiloženého preprocesoru GLSL souborů. Další vylepšení by mohlo spočívat v lehkém překrývání jednotlivých podprostorů. Tento překryv by následně během vykreslování stínů mohl být použit pro filtrování mezi dvěmi kaskádami tak, aby přechod mezi nimi byl méně znatelný.

Dalším směrem této práce by mohlo být ustálení dělících ploch v případě SDSM tak, aby chybovost jednotlivých kaskád nebyla měněna příliš rychle. Tímto by nebylo pozorovatelovo oko na tuto nedokonalost tak lehce vedeno. Dále by bylo možno celý výpočet matic pro vykreslení shadow map v metodě Sample Distribution Shadow Maps přesunout na grafickou kartu. Tím by se zamezilo možnému čekání CPU na výsledky výpočtu rozložení vzorků dle vzdáleností.

Literatura

[1] Dou, H.; Yan, Y.; Kerzner, E.; aj.: Adaptive Depth Bias for Shadow Maps. *Journal of Computer Graphics Techniques (JCGT)*, ročník 3, č. 4, December 2014: s. 146–162, ISSN 2331-7418.
URL http://jcgt.org/published/0003/04/08/

[2] Giesen, F.: Ring buffers and queues. 2010.
URL https://fgiesen.wordpress.com/2010/12/14/ring-buffers-and-queues/

[3] Lauritzen, A.; Salvi, M.; Lefohn, A.: Sample Distribution Shadow Maps. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, New York, NY, USA: ACM, 2011, ISBN 978-1-4503-0565-5, s. 97–102, doi:10.1145/1944745.1944761.
URL http://doi.acm.org/10.1145/1944745.1944761

[4] Luebke, D.; Reddy, M.; Cohen, J. D.: *Level of detail for 3D graphics*. Boston: Morgan Kaufmann, vyd. 1 vydání, 2003, ISBN 1-55860-838-9.

[5] Reeves, W. T.; Salesin, D. H.; Cook, R. L.: Rendering Antialiased Shadows with Depth Maps. *SIGGRAPH Comput. Graph.*, ročník 21, č. 4, Srpen 1987: s. 283–291, ISSN 0097-8930, doi:10.1145/37402.37435.
URL http://doi.acm.org.ezproxy.lib.vutbr.cz/10.1145/37402.37435

[6] Tadamura, K.; Qin, X.; Jiao, G.; aj.: Rendering optimal solar shadows with plural sunlight depth buffers. *The Visual Computer*, ročník 17, č. 2, Mar 2001: s. 76–90, ISSN 1432-2315, doi:10.1007/PL00013400.
URL https://doi.org/10.1007/PL00013400

[7] Williams, L.: Casting Curved Shadows on Curved Surfaces. *SIGGRAPH Comput. Graph.*, ročník 12, č. 3, Srpen 1978: s. 270–274, ISSN 0097-8930, doi:10.1145/965139.807402.
URL http://doi.acm.org/10.1145/965139.807402

[8] Wimmer, M.; Scherzer, D.; Purgathofer, W.: Light Space Perspective Shadow Maps. In *Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques*, EGSR'04, Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2004, ISBN 3-905673-12-6, s. 143–151, doi:10.2312/EGWR/EGSR04/143-151.
URL http://dx.doi.org/10.2312/EGWR/EGSR04/143-151

[9] Zhang, F.; Sun, H.; Xu, L.; aj.: Parallel-split Shadow Maps for Large-scale Virtual Environments. 2006: s. 311–318, doi:10.1145/1128923.1128975.
URL http://doi.acm.org/10.1145/1128923.1128975

[10] Zhou, J.-W.; Wan, W.-G.; Cui, B.; aj.: A method of view-frustum culling with OBB based on octree. In *IET Conference Publications*, 533, 2007, ISBN 9780863418365, s. 680–682, doi:10.1049/cp:20070239.

[11] Žára, J.; Beneš, B.; Felkel, P.: *Moderní počítačová grafika*. Praha: Computer Press, vyd. 1 vydání, 1998, ISBN 80-7226-049-9.

Příloha A

Obsah CD

Přiložené CD obsahuje zdrojové kódy výsledné aplikace, této textové práce a data vzniklá měřením.

Adresář `results`, obsahuje skript, kterým byly data vyhodnocována a poté složky pojmenované dle tabulky 5.3 v kapitole 5.

V adresář `src`, obsahuje zdrojové kódy výsledné aplikace. Pokud uživatel chce provést překlad na svém PC, je nutné pomocí nástroje **CMake**, vygenerovat projekt pro knihovnu PugiXML. Ta se nachází v adresáři `external/pugixml-1.8`, spolu s tímto adresářem zde můžete najít předpřipravené adresáře pro vygenerování projektu `pugixml-1.8-build` a instalační adresář `pugixml-1.8-install`. Projekt určený pro Visual Studio 2015 můžete najít v adresáři `/src`.

V adresáři `/src/bin` můžete najít dva přednastavené projekty pro program *RenderDoc*, kterým je možno blíže analyzovat průběh vykreslení určitého snímku.

```
/
├── results ..... výsledky provedených měření
│   ├── PC1
│   ├── PC2
│   ├── PC3
│   └── PC4
├── src ..... zdrojové soubory aplikace
│   ├── bin ..... zde naleznete spustitelný soubour
│   │   ├── models ..... scény a jejich definice
│   │   ├── results ..... složka kam jsou vygenerovány výsledky měření
│   │   └── shaders ..... shadery a jejich definice
│   ├── external
│   │   ├── pugixml-1.8
│   │   ├── pugixml-1.8-build
│   │   └── pugixml-1.8-install
│   ├── include
│   ├── lib ..... použité knihovny
│   └── src ..... zdrojové kódy aplikace
└── text ..... zdrojové soubory textové části práce
```

Výpis A.1: Adresářová strukura na disku CD

Příloha B

Ovládání aplikace

Aplikace má dva módy ve kterých může být spuštěna. Prvním je interaktivní mód. V tomto módu uživatel ovládá pohyb kamery, nastavení použitých metod nebo například chystat trasu pro průlet scénou. Druhým módem je právě tento průlet scénou, ten nemůže uživatel nikterak ovládat.

Aplikaci lze spustit spolu s několika argumenty, z toho je několik povinných. Zde můžete vidět seznam argumentů.

```
./"Opengl Shadow Rendering-Target.exe" SceneDef Method Path Shader
```

Target je v tomto příkladu nutno nahradit za verzi programu, kterou chce spustit. Ty jsou difnovány v kapitole 4.6.

Argumenty nemohou měnit své pořadí. Jediný povinný argument je první z uvedených **SceneDef**. Tento argument slouží k určení lokace definičního soubrou scény (formát je popsán v příloze D) a musí být zapsán relativně k umístění spustitelného souboru. Několik scén je předpřipraveno v adresáři `models`. Druhým argumentem je použitá metoda výpočtu stínů. Tento argument může nabývat hodnot *pssm* nebo *sdsm*. K oběma variantám je možné připsat postfix *-noLayers*, kterým můžete vypnout používání vrstvového vykreslování. Argument **Path** definuje trasu, kterou bude prolétat kamera, během takovéhoto průletu se vygenerují naměřené hodnoty do adresáře `results` a to do formátu definovaného v kapitole 5.

Poslední argumetn definuje použitý GLSL program pro vykreslení barev ve scéně. Musí nabývat hodnot odpovídajícím jménům definičních souborů popsaných v příloze C. Tento definiční soubor se musí nacházet v adresáři `shaders`. Argument může být zapsán bez přípony. Několik takových shaderů je zde již nachystaných. Výchozí se nazývá `basic`. Tento program vykresluje modely s barvami a texturami definovanými v .mtl souborech. Další zajímavý program je `basic-planes`. Tento program vykresluje modely s barvami dle toho, v jaké vrstvě se nacházejí. Tyto barvy jsou definovány v tabulce 3.1. Dalším z používaných programů je `basic-poisson`. Tento program aplikuje filtrování na stínovou mapu, takže jsou stíny vyhlazené, tento program generuje kvalitativně nejlepší výsledky.

Pokud by si uživatel nebyl jist použitím jednotlivých argumentů, může najít jejich užití v souboru `bin/test.bat`.

Pokud uživatel spustí program v interaktivním módu, může v konfiguraci Debug, má k dispozici GUI, pomocí kterého může nastavovat veškeré parametry vykreslování. Pozor by si hlavně měl dát, kterou kameru v interaktivním módu používá. Jak jsem rozebíral v sekci 3.2 v aplikaci existují dvě kamery. Ta která se spustí jako výchozí je debuggová. Do pohledu z hlavní kamery se přepne zmáčknutím klávesy **Tab**, pokdu chce pouze ovládat

hlavní kameru, ale dále pozorovat scénu pomocí kamrey debuggové lze tak učinit současným držením klávesy **v**. K pohybu po scéně slouží ustálená kombinace kláves **wasd**.

V interaktivním módu také uživatel může nahrát trasu pro automatické průlety. Toho lze učinit tak, že se uživatel pohybuje po scéně a pokud dorazí na místo, kam by chtěl vložit takzvaný klíčový snímek(key frame), stlačí klávesu **k**. K uložení trasy stačí zmáčkount klávesu **m**, poté je uživatel vyzván, aby zadal jméno souboru, kam bude trasa uložena. Pokud chce změnit délku trasy, musí tak učitnit přímo v XML zápisu.

Příloha C

Definice shaderů

Každý shader se skládá z několika podrogramů, který každý definuje jeden krok v shader pipeline. Každý z těchto podprogramů se zvlášť kompiluje a následně musí být nalinkovány do celého shader programu, definujícího pipeline. Pokud je rozhraní takového podprogramu dostatečně obecné, je možné takový podprogram použít v několika oddělených pipeline. Na základě těchto předpokladů, jsem se pro jednodušší práci se shadery rozhodl implementovat jednoduchý systém jejich definice pomocí XML souborů. XML je jednoduchý značkovací jazyk jednoduše čitelný člověkem. Výhoda tohoto přístupu je, že není nutné pro výměnu jednoho podrogramu v určité pipeline kompilovat spustitelný soubor (a díky výše popsanému systému přenačítání shaderů za běheu v Debug konfiguraci ani vypínat běžící program), ale postačí měnit jeho definici v XML.

Takový soubor pak má jednoduchou strukturu. Kořenový element souboru se musí jmenovat `<pipeline>`, který pak obsahuje jednotlivé elementy `<shader>`. Takový element obsahuje cestu ke zdrojovému souboru shaderu, určenou relativně k xml souboru. Povinný je atribut `stage`. Ten definuje, kteoru část pipeline zastává daný soubor a může nabývat hodnot "vertex", "fragment", "geometry"a "compute".

V kódu se dá takovouto pipeline vyžádat od singleton třídy `C_ShaderManager` metodou `GetProgram` s řetězcovým parametrem shodným se jménem xml souboru. Tento soubor se musí nacházet v adresáři "shaders", který je v pracovním adresáři spuštěné instance souboru. `C_ShaderManager` jednotlivé pipeline kešuje, tudíž nedochází k opakované kompilaci programu kdykoli si ho vyžádáte. Pokud chcete využít funkce živého aktualizování pipeline za běhu programu, tak je vhodné si vyžádat tento `std::shared_ptr` před každým použitím programu znova a nedržet ukazatel na tento program v instanci dané třídy, protože potom nebudete mít aktuální program.

Současně implementace vlastního prepocesoru jazyka GLSL v této práci umožňuje používat direktivu `#include`. Ta funguje pouze s relatvními cestami vůči umístění zdrojového souboru .glsl. Dalším omezením této spočívá v její nerekurzivnosti. Pokud tuto direktivu použijete ve vloženém souboru, nebude brána v potaz. Tím se preprocesor vyhýbá cyklickým závislostem souborů. Důvod tohoto omezení je dostatečnost takovéto funckce v tomto projektu. Jinak je funkcionalitou stejná, jako očekáváte v C++.

```
<?xml version="1.0" encoding="UTF-8"?>
<pipeline>
  <shader stage="vertex">basic/vertex-shadow-layered.glsl</shader>
  <shader stage="geometry">basic/geometry-shadow.glsl</shader>
  <shader stage="fragment">basic/fragment-shadow.glsl</shader>
</pipeline>
```

Výpis C.1: Ukázka definice shaderu

Příloha D

Definice scény

V mém projektu jsem se rozhodl použít pro definici jednoduchý soubor psaný v XML jazyce. Soubor obsahuje povinný kořenový element `<scene>`. Ten může obsahovat dva typy potomků `<terrain>` a `<model>`. Protože elementy typu `<terrain>` tato nikde nevyužívá, nebudu se o nich dále rozepisovat více, než že v kódu jsou reprezentovány pomocí tzv. render node `C_Terrain`.

Celá práce využívá elementy `<model>`, které musí obsahovat atribut "file". Ten určuje relativní cestou umístění souboru ȯbj. Tento element může dál obsahovat elementy `<position>`, který definuje pozici modelu ve světových souřadnicích a `<obejct>`, který upřesňuje atributy pro jednotlivé objekty tohoto assetu. Objekt je určen atributem "name", který se odkazuje na jméno objektu přesně dle definice v souboru .obj a aktuálě může definovat booleovským atributem `isShadowCaster` zda bude objekt vrhat stín. Tato funkce je projektu pro demonstraci praktického použití projektu, kdy toto je to, co bychom od běžného engine chtěli.

```
<?xml version="1.0" encoding="UTF-8"?>
<scene>
	<!-- <terrain file="profile.jpg" tile-size="1" /> -->
	<model file="testScene/scene.obj">
		<position x="0.0" y="0.0" z="0.0" />
		<object name="Plane" isShadowCaster="false" />
	</model>
	<model file="testScene/castle.obj">
		<position x="1.0" y="0.0" z="10.0" />
		<object name="Floor_Plane" isShadowCaster="false" />
	</model>
</scene>
```

Výpis D.1: Ukázka definice scény