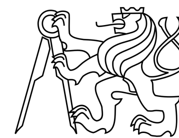
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁ SKÉ PRÁCE

Název:	Metodika pro srovnání deferred a forward renderer
Student:	Tomáš Malý
Vedoucí:	Ing. Ji í Chludil
Studijní program:	Informatika
Studijní obor:	Softwarové inženýrství (bakalá ský)
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2014/15

Pokyny pro vypracování

- Navrhn te metodiku pro m ení výkonu grafických renderer . Sestavte sadu m itelných parametr pro klasifikaci scény podle vlivu na výkon.
- S ohledem na zvolenou metodiku a sledované parametry navrhn te vhodné testovací scény. Scény sestavujte tak, abyste pokryl velké rozsahy všech sledovaných parametr . Pro každou vytvo enou scénu zm te všechny sledované parametry.
- Pomocí metod softwarového inženýrství navrhn te a implementujte deferred a forward renderery, na kterých podle zvolené metodiky provedete srovnání výkonu nad navrženými scénami.
- Program implementujte pro MS Windows. Využijte OpenGL a další vhodn zvolené knihovny. Rozdíly mezi grafickými kartami, ostatním hardwarem, r znými knihovnami ani opera ními systémy neuvažujte.
- Pro m ení využijte také infrastrukturu SAGE laborato e.
- Výsledky m ení programu zhodno te a vhodnou formou vizualizujte. P iložte vyrendrované náhledy všech scén.

Seznam odborné literatury

Dodá vedoucí práce.

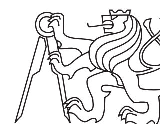
L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof.Ing. Pavel Tvrdík, CSc.
d kan

V Praze dne 18. b ezna 2014

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra . . . (softwarového inženýrství)



Bakalářská práce

Metodika pro srovnání deferred a forward rendererů

Tomáš Malý

Vedoucí práce: Ing. Jiří Chludil

25. června 2014

Poděkování

Rád bych poděkoval Ing. Jiřímu Chludilovi a Ing. Radku Richtrovi za kvalitní a odborné vedení a oponenturu mé bakalářské práce. Dále děkuji Ing. Jiřímu Melnikovi za pomoc s technologií SAGE. Nakonec děkuji sdružení CESNET za poskytnutí výpočetního výkonu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 25. června 2014 ....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2014 Tomáš Malý. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Malý, Tomáš. *Metodika pro srovnání deferred a forward rendererů.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2014.

Abstrakt

Deferred renderer se v posledních letech stal velmi populární technikou používanou v mnoha počítačových hrách a herních nebo grafických enginech. Cílem této práce je experimentálně ověřit použitelnost této techniky v budoucnosti s nástupem obrazovek s velkými rozlišeními (4k nebo 8k) a obecně, pro které scény a nastavení je vhodná.

Klíčová slova deferred, forward, render, OpenGL, SAGElib

Abstract

Deferred renderer have become a very popular technique used in several computer games or even game and graphical engines. Main goal of this thesis is to experimentally check out its usability in near future where high definition (4k or 8k) is standard; or what scenes and configurations is it suitable for.

Keywords deferred, forward, render, OpenGL, SAGElib

Obsah

Úvod 1
Rozbor zadání . . . 2
Poznámka k angličtině . . . 2

1 Fungování deferred rendereru 3
1.1 Forward renderer . . . 3
1.2 Deferred renderer (deferred shading) . . . 4
1.3 Nevýhody . . . 7
1.4 Souhrn . . . 8
1.5 Varianty . . . 8

2 Metodika 9
2.1 Vstupy . . . 9
2.2 Výstupy . . . 11

3 Systém pro měření 13
3.1 Funkční požadavky . . . 13
3.2 Nefunkční požadavky . . . 15
3.3 Případy užití . . . 15
3.4 Zvolené vlastnosti metodiky . . . 18
3.5 Architektura . . . 24
3.6 Datový relační model databáze . . . 27
3.7 Model komponent . . . 29
3.8 Grafické znázornění výsledků . . . 30
3.9 Zdrojové kódy . . . 30

4 Výsledky 33
4.1 Počet volání . . . 33
4.2 Časy renderování . . . 34
4.3 Průměr na pixel respektive trojúhelník . . . 38

4.4 Deferred renderer 40

Závěr **43**
Možnosti pokračování 43

Literatura **45**

A Pseudokódy **47**
A.1 Forward renderer 47
A.2 Deferred renderer 50

B Seznam použitých zkratek a cizích slov **55**

C Obsah přiloženého CD **57**

Seznam obrázků

1.1 Znázornění průběhu forward rendereru 3
1.2 Znázornění průběhu deferred rendereru 5
1.3 Znázornění obsahu g-bufferu. Vlevo hloubka, uprostřed diffuse barva a vpravo normály. Obrázky převzaty z [9]. 6
1.4 Sponza Atrium — z hodnoty normál (zelená — záporné) 6
1.5 Vlevo správně osvětlený obrázek. Vpravo ukázka efektu kvantování normál. 7

3.1 Sponza Atrium 1 19
3.2 Sponza Atrium 2 20
3.3 Sponza Atrium 3 — starší verze 20
3.4 Head 21
3.5 Xeno Queen 22
3.6 Conference Room 22
3.7 Sibenik 23
3.8 Rozložení g-bufferu v deferred rendereru. 25
3.9 Model architektury 25
3.10 Model SAGE 26
3.11 Relační model databáze 28
3.12 Model komponent 29
3.13 Diagram dědičnosti rendererů 29
3.14 Ukázka formuláře webové aplikace 30

4.1 Graf počtu volání OpenGL funkcí. 34
4.2 Časy renderování scény Sponza Atrium 34
4.3 Časy renderování scény Xeno Queen 35
4.4 Časy renderování scény Head 35
4.5 Časy renderování scény Conference Room 35
4.6 Časy renderování scény Sibenik 36
4.7 Časy renderování v rozlišení nHD 36

4.8 Časy renderování v rozlišení FullHD 37
4.9 Časy renderování v rozlišení FUHD 37
4.10 Časy renderování s 8 světly 38
4.11 Časy renderování s 32 světly 38
4.12 Časy renderování s 128 světly 39
4.13 Časy renderování s 512 světly 39
4.14 Časy renderování průměrně na pixel 39
4.15 Časy renderování průměrně na trojúhelník 40
4.16 Časy přípravy g-bufferu 41
4.17 Časy osvětlovacího průchodu 41

Seznam tabulek

3.1 Tabulka s napočítanými parametry scén. 19

Úvod

Počítačová grafika je velmi mladý a dynamicky se rozvíjející obor. Proto obvykle každá dobrá nová myšlenka bývá rychle využita a implementována. V případě deferred rendereru tomu tak ale nebylo a tato technika se začala používat až po více jak desetiletí od jejího prvního návrhu.

Proč se deferred renderer stal tak populární až v posledních pár letech i přes to, že tato myšlenka tu byla už tak dlouho? Na tuto otázku je odpověď překvapivě jednoduchá. Nejdůležitějším faktorem v této obměně byl velký nárůst kapacity paměti.

Spousta dalších otázek už ale tak jednoduchá není, a právě jim se věnuje tato práce:

1. Jak deferred renderer funguje? Jaké jsou jeho výhody a nevýhody oproti jiným technikám? A jaké jsou k němu alternativy nebo jeho varianty? Těmto otázkám se věnuje kapitola Fungování deferred rendereru.

2. Jak lze „měřit“ grafické techniky? Na čem všem závisí naše měření a co ho může ovlivnit? Více v kapitole Metodika.

3. Jaké parametry mě konkrétně zajímají? Jaké jsem měl požadavky na program pro experimentální porovnání deferred a forward rendererů? Jak jsem tento program navrhl a implementoval? Odpovědi na tyto otázky jsou v kapitole Systém pro měření.

4. Na jakých počítačích jsem provedl měření? Jaké hodnoty jsem naměřil? Jaké závěry usuzuji z naměřených údajů? A který renderer bych doporučil? O tomto se více dozvíte v poslední kapitole Výsledky.

Rozbor zadání

*Renderer*em mám na mysli základní kostru programu pro renderování. Různé renderery se tedy neliší tím, zda používají například HDR ale tím, kolikrát iterují přes geometrii či přes světla.

Slovem *scéna* myslím konkrétní rozprostření (naaranžování) 3D objektů (modelů) v prostoru, umístění kamery, rozmístění světel a dalších podobných předmětů. *Model*em označuji tvar, materiál a další přidružené vlastnosti jednoho konkrétního objektu (např. váza).

Ačkoliv slovo SAGE (viz B) samo o sobě má jiný význam, souslovím *infrastruktura SAGE* rozumím technické vybavení Síťové a multimediální laboratoře FIT ČVUT, která se běžně přezdívá právě podle slova SAGE. Více o naší laboratoři na stránkách [5].

Poznámka k angličtině

V textu používám mnoho slov přímo v anglickém jazyce nebo v jejich počeštěné (vyskloňované) podobě. Tato slova nepřekládám obvykle z jednoho nebo obou ze dvou důvodů:

- Některá anglická slova vůbec nemají český překlad, nebo případný český překlad původnímu anglickému slovu nedostatečně odpovídá. Příkladem takového slova je „render“, jehož překlad je „udělat, učinit, poskytnout, prokázat“, ačkoli v angličtině se toto slovo používá i ve významu „tvorba realisticky vypadajícího obrazu na základě počítačového modelu“. Já od tohoto slova odvodil (počeštil) i další tvary jako „renderer“, „renderování“, apod.

- Některé algoritmy, techniky a další věci mají ustálené anglické pojmenování, které by po přeložení do češtiny mohlo být matoucí. Příkladem takového pojmenování je „hardware“.

Kapitola 1

Fungování deferred rendereru

Ještě než si vysvětlíme deferred renderer, popíšeme si jak funguje klasický forward renderer. Následovat bude princip deferred rendereru, jeho výhod a nevýhod a stručný popis jemu podobných alternativ.

V příloze Pseudokódy je zjednodušený výtah implementace obou těchto rendererů.

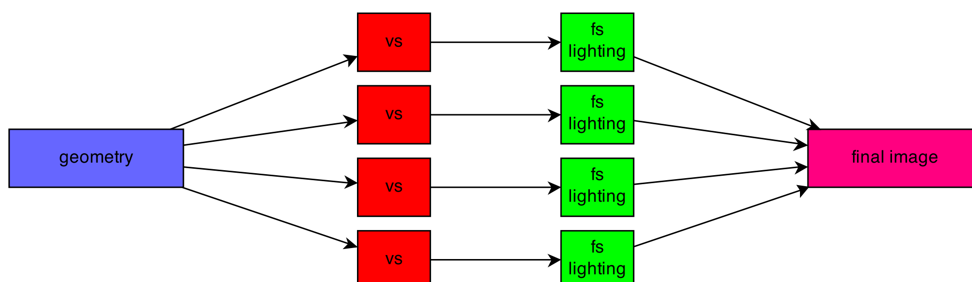
1.1 Forward renderer

Toto je nejpřímočařejší postup renderování scény. Jednoduše pro každý objekt počítá příspěvky všech světel, což vede na asymptotickou časovou složitost

$$O(L * M)$$

kde L je počet světel a M je počet modelů. Názorná ukázka postupu je na obrázku 1.1.

V podstatě jsou dvě možnosti jak zpracovat více světel:



Obrázek 1.1: Znázornění průběhu forward rendereru

1.1.1 Jedno-průchodový

Každý objekt je jednou renderován. Ve fragment shaderu jsou aplikována všechna světla v cyklu. S velkým počtem světel je při tomto postupu vyžadováno velké množství uniform proměnných pro data světel. Další problém je se stíny. Nejběžněji používaná metoda - shadow mapping — vyžaduje jednu texturu pro každé světlo. Takto můžeme velmi rychle narazit na limitovaný počet texturovacích jednotek v grafické kartě.

1.1.2 Více-průchodový

Každý objekt je renderován s každým světlem samostatně. Výsledná barva se akumuluje v color-bufferu. Tento postup, ačkoli řeší problém s limitovaným počtem texturovacích jednotek a s pamětí na uniform proměnné, má také své nevýhody. Vyžaduje velké množství příkazů na grafickou kartu a může snadno překročit propustnost sběrnice. Navíc velké množství výpočtů je zbytečně opakováno (transformace na vrcholech, skeletární animace).

1.1.3 Souhrn

Největším problémem obou těchto postupů zůstává velké množství nadbytečných výpočtů s každým překrytím fragmentu (depth-test), kdy všechny složité výpočty osvětlení jsou bez užitku zahozeny. Tento problém může být u jednoduchých scén vyřešen správným řazením objektů (nejbližší renderovat jako první) nebo technikou depth-prepass.

1.2 Deferred renderer (deferred shading)

Deferred renderer řeší problémy s drahými výpočty osvětlení tak, že je, jak název napovídá, odloží na později. Celý proces renderování rozděluje do dvou průchodů. Výsledná asymptotická časová složitost je

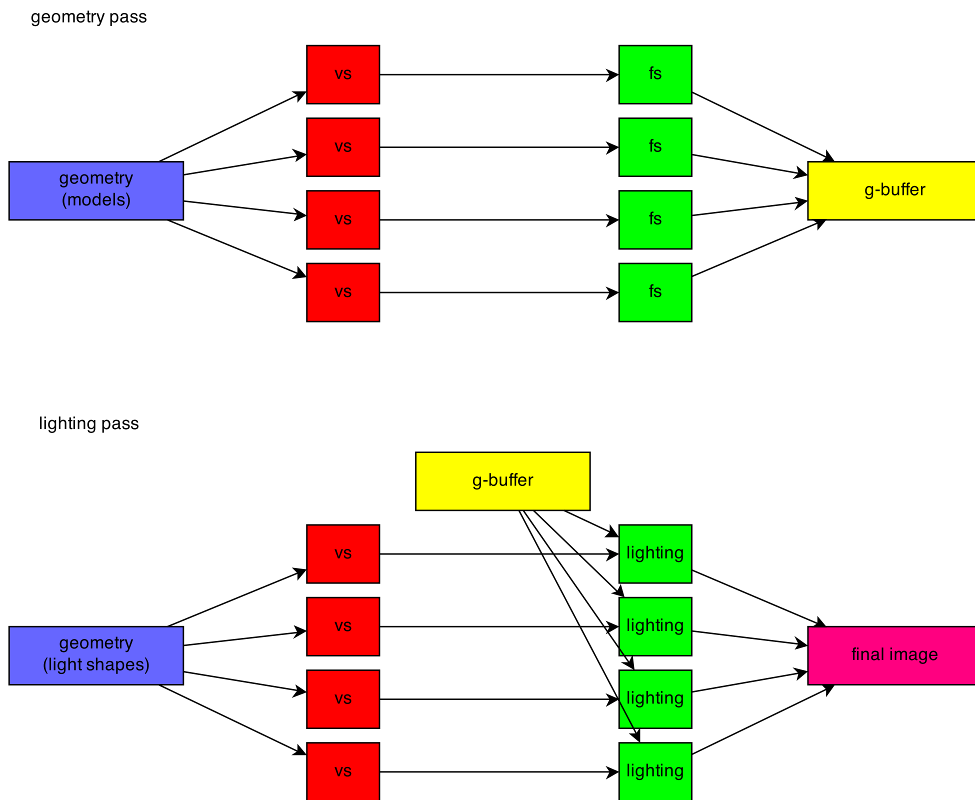
$$O(L + M)$$

kde L je počet světel a M počet modelů. Náhled průchodu deferred rendereru je na obrázku 1.2.

1.2.1 Geometry-pass

V první fázi se postupně vyrenderují všechny objekty. Ve fragment shaderu se místo výpočtu osvětlení všechny materiálové vlastnosti uloží do MRT (multiple render targets), tedy několika textur souhrnně označovaných jako g-buffer, a které budou v druhé fázi použity pro samotný výpočet osvětlení.

G-buffer typicky obsahuje pozici, normálu, difuzní a speculární barvu a speculární exponent (shininess), id materiálu a další. Některé z těchto údajů

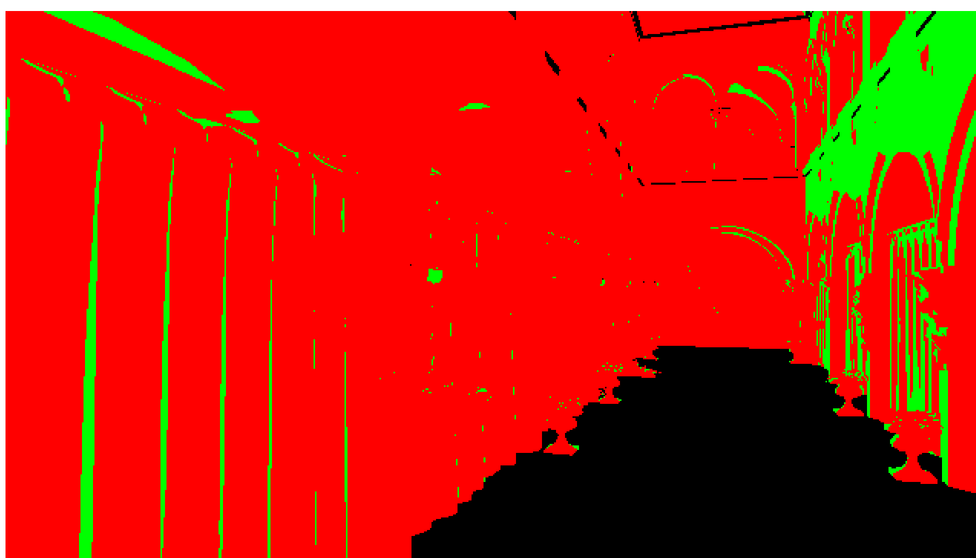


Obrázek 1.2: Znázornění průběhu deferred rendereru

lze vynechat z g-bufferu a při osvětlování je dopočítávat. Například pozici lze vynechat úplně a dopočítat ji ze souřadnic fragmentu a z hodnoty z hloubkového bufferu. Normály lze komprimovat do jiných souřadných systémů nebo jinak upravit tak, aby byl využit celý rozsah použitelných hodnot. Tyto úspory se typicky využijí ke zvýšení přesnosti některých položek místo ke snížení spotřeby paměti. Výsledkem tak jsou obvykle 3 až 4 textury s 4 bajty na pixel. Obrázek s náhledem bufferů je 1.3.

Pro full-HD rozlišení je, při této konfiguraci, potřeba kolem 32 MB paměti grafické karty, což dnes není problém. Pro rozlišení 8k už potřebujeme přibližně 512 MB. Zde je potřeba upozornit, že hlavní problém nespočívá v samotném uložení dat, ale v jejich čtení a zapisování, což rychle spotřebuje všechnu dostupnou přenosovou kapacitu.

Obrázek 1.3: Znázornění obsahu g-bufferu. Vlevo hloubka, uprostřed diffuse barva a vpravo normály. Obrázky převzaty z [9].



Obrázek 1.4: Sponza Atrium — z hodnoty normál (zelená — záporné)

1.2.1.1 Normály v g-bufferu

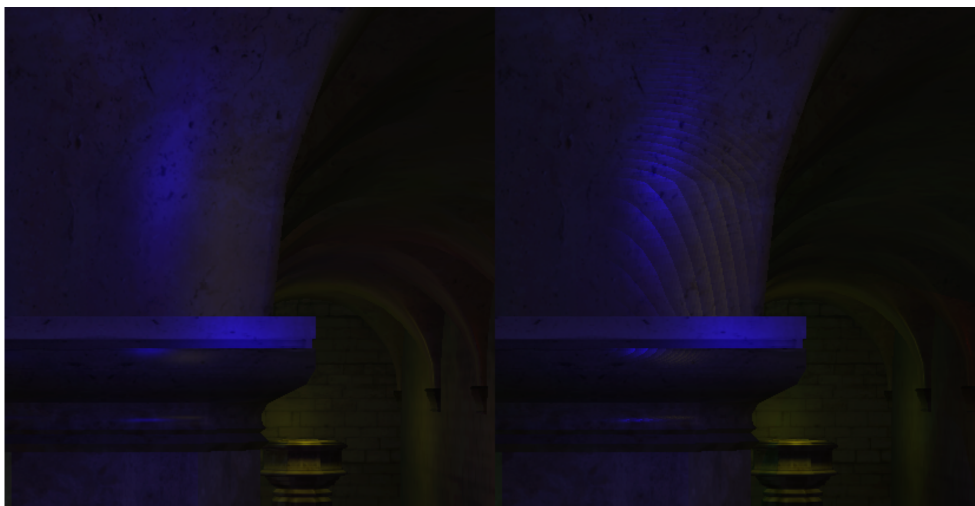
Ohledně uložení normál v g-bufferu existuje mýtus, že stačí ukládat souřadnice x a y s tím, že z se dopočítá podle vzorce

$$z = sqrt(1 - x * x - y * y)$$

tato metoda ovšem předpokládá, že žádné normály ve scéně (v souřadném systému kamery) nesměřují od kamery — jinými slovy předpokládá, že z souřadnice je vždy nezáporná. Bohužel tento předpoklad není vždy pravdivý, toto jsem si ověřil a na obrázku 1.4 znázornil.

Dobrá diskuse k této a dalším možnostem kompaktnějšího (nebo přesnějšího) uložení normál je na webové stránce [11].

S uložením normál v g-bufferu souvisí ještě jeden problém — takzvané kvantování. Normály jsou vždy jednotkové délky a tím pádem se při uložení



Obrázek 1.5: Vlevo správně osvětlený obrázek. Vpravo ukázka efektu kvantování normál.

využívá jen málo z dostupných kombinací v daném počtu bajtů. Výsledkem jsou viditelné pruhy vzniklé při nasvícení. Ukázka efektu je na obrázku 1.5.

1.2.2 Light-pass

V druhé fázi se skutečně provádí výpočet příspěvků světel na výslednou barvu fragmentů. Jelikož shadery musí vždy pracovat nad geometrií, tak se v této fázi pro každé světlo vyrenderuje co nejjednodušší obálka světla. Pro bodové světlo koule, pro spot světlo jehlan a pro směrové světlo obdélník přes celou obrazovku. Ve fragment shaderu se pak z g-bufferu vytahají všechny potřebné údaje pro výpočet osvětlení a výsledná barva se akumuluje v color-bufferu.

1.3 Nevýhody

Jako obvykle, ani tato technika není zcela bez poskvrny. Významným nedostatkem je, jak již bylo diskutováno dříve, velká spotřeba paměti pro g-buffer.

Další problém je, že osvětlení lze počítat pouze pro jeden fragment na každé pozici. Takže se deferred renderer nedá použít na částečně průhledné objekty. Ty je potřeba dodatečně dopočítat klasickým forward rendererem.

Také často zmiňovaným problémem je nemožnost využít hardwarové vyhlazovací (anti-aliasing) techniky.

1.4 Souhrn

Deferred renderer řeší adresovaný problém popsaný u forward rendereru tím, že výpočet osvětlení odkládá až do doby, kdy o daném fragmentu má jistotu, že bude viditelný.

Ve srovnání s jedno-průchodovým forward rendererem netrpí ani problémem s počtem texturovacích jednotek. V první fázi (geometry-pass) není problém. Všechny texturovací jednotky jsou k dispozici pro čtení povrchových vlastností objektů. V druhé fázi (lighting-pass) jsou použity (typicky) čtyři texturovací jednotky pro čtení z g-bufferu a všechny ostatní jsou k dispozici například pro použití na stíny.

1.5 Varianty

Kromě zde popsané verze (deferred shading) existují i další možnosti:

- **Deferred lighting**

  Deferred lighting používá dokonce tři fáze. V 1. fázi do g-bufferu uloží hloubku a normálu. Ve druhé fázi provádí výpočet osvětlení — výsledkem je buffer s irradiancí. Ve třetí fázi opět projde geometrii a s pomocí irradiance dopočítá finální barvy. Hlavní pointou této alternativy je snížit velikost g-bufferu a tím i snížit množství čtených a zapisovaných dat. Nevýhodou je dvojí průchod přes geometrii. Někdy je tato technika označována také jako light pre-pass.

- **Indexed lighting**

  Indexed lighting, stejně jako defered lighting, používá tři fáze. V první fázi se provede depth pre-pass — uloží se hloubky do depth-bufferu. Ve druhé fázi se renderují obálky světel a do bufferu se uloží indexy světel, které daný fragment zasahují. Ve třetí fázi se renderuje geometrie scény a ve fragment shaderu se údaje z bufferu použijí jako indexy do tabulky světel.

  Celkově je tato technika velmi podobná té předchozí, ale má menší spotřebu paměti. Také umožňuje větší rozmanitost materiálových i světelných vlastností.

  Problémem této techniky je otázka, jak efektivně ukládat indexy světel do bufferu.

  Více informací na [12].

- **Tiled lighting**

  Podobné jako indexed lighting, ale místo ukládání seznamu světel pro každý pixel jsou ukládány pro dlaždice. Více informací na [10].

Kapitola 2

Metodika

Druhá kapitola obecně popisuje metodiku pro srovnání rendererů. Jsou zde shrnuty vstupy — co vše může ovlivnit rychlost renderování. A výstupy — jaké hodnoty lze při měření získávat.

2.1 Vstupy

Zde jsou rozepsány charakteristiky scény a možná nastavení rendereru. Tento seznam není nijak řazený a nemusí být úplný, ale měl by zachytit ty položky, které mohou výkon rendereru ovlivnit nejcitelněji.

- **Algoritmy**

  Výkon je samozřejmě ovlivněn použitými postupy. Zvolený renderer je jen jedno z mála možných rozdílností. Kromě toho existuje spousta algoritmů používaných pro zrychlení nebo naopak pro zlepšení výsledného obrazu. Žádný z těchto algoritmů není vždy lepší než jiné. Často záleží na typu aplikace a požadavcích (očekáváních). Krátký, neúplný a neuspořádaný výčet:

  - organizace objektů ve scéně: bsp, octree, tunel, portal
  - scene occlusion
  - depth pre-pass
  - ambient occlusion
  - depth of field
  - bloom
  - motion blur
  - HDR
  - antialiasing: fxaa, msaa

  - animace
  - stíny: stencil shadows, volume shadows, shadow maps

- **Geometrie**

  Jeden z nejdůležitějších faktorů je geometrie, tedy zobrazovaný model nebo scéna. Ty se liší v počtu trojúhelníků (nebo jiných geometrických primitiv), v pravidelnosti teselace primitiv, jejich velikosti a v míře vzájemného překrytí.

- **Hardware**

  Rychlost záleží na výkonu GPU, CPU, velikosti RAM a grafické paměti a dalších součástkách.

  Pro srovnání by potenciálně stačilo provést všechna měření na jedné konkrétní sestavě. Takové výsledky ovšem nebudou nutně korelovat s jinými sestavami právě z důvodu komplexnosti celého procesu renderování. HW se neliší jen rychlostí (taktem) a velikosti paměti, ale i v propustnosti sběrnice, typu cache, zapojení součástek atd.

  S hardwarem souvisí také nainstalované ovladače.

- **Rozlišení**

  Rozlišení obrazu udává počet pixelů, pro které je potřeba určit výslednou barvu, tedy provést velké množství výpočtů. Dnes je standardem vypočítávat osvětlení a všechny další efekty per-pixel (dříve per-vertex). V případě deferred rendereru je právě rozlišení velmi stěžejní i pro spotřebu paměti.

  Malá rozlišení, která jsou dnes běžně na data-projektorech a přenosných zařízeních, mají kolem půl miliónu pixelů. Velká rozlišení (full HD), která běžně mají dnes prodávané televizory i monitory, mají přes dva miliony pixelů. Filmy v kině jsou obvykle promítány v rozlišení 4k s více než osmi miliony pixelů. Telestěna na naší fakultě s rozlišením 8k má více než 32 miliónů pixelů.

- **Světla a osvětlovací model**

  Dalším faktorem ovlivňujícím výkon je počet, druh a nastavení světel a celkově použitý osvětlovací model. Jako základ se světla rozlišují na tři typy:

  - bodové (point) světlo má pozici, útlum a svítí všemi směry
  - reflektorové (spot) světlo má navíc směr a ořezový úhel
  - směrové (directional) světlo nemá pozici ani útlum, má jen směr a svítí z nekonečné dálky

Kromě těchto základních jsou i další druhy světel (například plošné), ty jsou ale příliš různorodé a už se s nimi dále nebudu zabývat.

Významným faktorem u každého světla je také fakt, zda vrhá stíny. Ty obvykle vyžadují extra průchod přes všechnu geometrii a můžou tak významně ovlivnit výkon.

Dále je rozdíl, zda jsou světla počítána ve vertex shaderu nebo ve fragment shaderu. Tato možnost volby není možná u deferred rendereru. Počítání osvětlení ve vrcholech je také krajně nevhodné pro spekulární složku osvětlení.

- **Rozlišení a filtrování textur**

  Vliv rozlišení textur na výkon neumím odhadnout, ale rozhodně má vliv na spotřebu paměti. Naproti tomu použité filtrování má na výkon rozhodně nezanedbatelný vliv.

- **Renderovací middleware**

  Dnes jsou dostupné v podstatě jen dvě možnosti: OpenGL a DirectX.

- **Hloubka barvového prostoru**

  Dříve se používalo 16 bitů na pixel. Dnes se typicky používá 32 bitů na pixel — 8 bitů na každou barevnou komponentu a 8 bitů na průhlednost. Protože jednotlivá světla se počítají samostatně a jejich výsledné příspěvky se sčítají, sčítají se také zaokrouhlovací chyby při převodu do tohoto barvového rozlišení. Toto lze vylepšit použitím techniky nazývané HDR, kde se v průběhu renderování renderuje do bufferu s vyšším barvovým rozlišením a teprve při převodu do front-bufferu se převede na 32 bitové.

- **Vertikální synchronizace**

  V-sync plní dvě funkce. Za prvé omezuje rychlost renderování na tolik fps, jakou mají monitory (šetří energii, neboť rychlejší renderování je zbytečné). Za druhé zabraňuje takzvanému tearingu — zobrazení směsice dvou za sebou jdoucích snímků s viditelným zlomem při jedné aktualizaci obrazu. Tato technika vnáší prodlevy do prohození předního a zadního bufferu.

2.2 Výstupy

Nejdůležitějším výstupem z měření je doba trvání jednoho snímku nebo (ekvivalentně) počet snímků za sekundu (frames per second — fps).

Kromě toho lze v rámci každého snímku měřit časy jednotlivých fází vykreslování.

2.2.1 Fps

Podle vlastní zkušenosti jsou počítačové 3D hry hratelné od cca 15 fps, ale není to příliš příjemné a po delší době (řádově jednotky hodin) z toho bolí oči. Nejnižší frekvence, která nezpůsobuje bolení očí je kolem 30 fps. Rychlost 60 fps poskytuje maximální komfort a zároveň je to horní hranice, kterou podporují dnes běžná zobrazovací zařízení (televize, monitory).

Fluktuace v fps mohou být způsobovány například pohybem nebo otočením kamery, změnou scény, realokací interních polí, spuštěním garbage-kolektoru, zpožděním na síti a dalšími.

2.2.2 Vliv měření na měřené hodnoty

Samotné měření spočívá ve zjištění systémového času mezi každou dvojicí snímků. Rozdíl dvou sousedních hodnot dá časy jednotlivých snímků. Čas na zjištění systémového času je ve srovnání s dobou trvání jednoho snímku naprosto zanedbatelný.

Pro zjištění doby trvání jednotlivých fází v rámci jediného snímku je nutné procesor opakovaně synchonizovat s grafickou kartou. Tato synchronizace (pomocí příkazu glFinish) může naměřené hodnoty ovlivnit mnohem výrazněji. Odhaduji až stovky mikrosekund při každém volání.

Teoreticky by šlo měřit zvlášť čas procesoru a zvlášť čas GPU, ale právě kvůli synchronizaci tuto variantu nedoporučuji.

2.2.3 Další výstupy

Kromě již rozebraného počtu snímku za sekundu lze měřit i další faktory. Zajímavým údajem je počet volání OpenGL příkazů. Vhodným výstupem by také byla spotřeba paměti, ideálně jak pro CPU, tak i pro GPU.

Kapitola 3

Systém pro měření

Cílem je vytvořit platformu pro srovnávání renderovacích technik. Porovnávání by mělo být možné z mnoha úhlů pohledů — na základě několika charakteristik. Jde o návrh softwaru potřebného pro všechny fáze porovnávání technik, počínaje přípravou dat přes provedení měření až po samotnou reprezentaci a grafické znázornění výsledků.

Veškerý software potřebný pro tuto úlohu se dá rozdělit na dvě části. První je aplikace pro provedení měření a druhá pro zobrazení výsledků.

V této kapitole je detailně rozepsaná analýza, návrh a samotná implementace takového softwaru.

Součástí analýzy je popis funkčních a nefunkčních požadavků na obě části softwaru a dále případy užití obou částí.

V návrhu softwaru jsou zde sepsané konkrétní zvolené vlastnosti dříve rozepsané metodiky, dále model architektury, použité externí knihovny, model databáze a model komponent.

3.1 Funkční požadavky

1. **Načtení konfigurace ze souboru**

   Program po spuštění načte konfiguraci ze souboru *measurement.txt*. Tento soubor je jednoduchý textový soubor s jedním nastavením na řádek ve formátu *<název volby>* = *<hodnota volby>*. Konfigurační soubor může obsahovat komentáře (uvozené znakem #). Na pořadí voleb v souboru nezáleží.

   Konfigurační soubor může obsahovat tyto volby (s těmito výchozími hodnotami):

   - Nastavení připojení k databázi: *host=localhost*, *user=measurement*, *password=measurement*, *schema=measurement*
   - Nastavení cesty, kam se mají ukládat screenshoty: *screenshots-path=screenshots*

- Nastavení intervalu případů testů: *tests=0–0*
- Nastavení rozlišení okna pro průběžné zobrazování: *resolution=1280*720*
- Nastavení módu (žádné, okno, fullscreen) okna pro průběžné zobrazování: *display=window*
- Nastavení, zda se má používat sage: *sage=false*
- Nastavení, zda se mají ukládat screenshoty: *screenshots=false*
- Nastavení, zda ukládané screenshoty ukládat v png nebo raw formátu: *screenshots-raw=false*

2. **Načtení konfigurace z parametrů**

   Program může nastavení přijímat i z parametrů jemu předaných při spuštění. Tyto hodnoty jsou ve stejném formátu jako ta ze souboru, ale mají vyšší prioritu.

3. **Validace konfigurace**

   Program musí ověřit zadanou konfiguraci. Například ověřuje, zda některý z parametrů není mimo rozsah.

4. **Načtení případu testu**

   Pokud načtení a validace konfigurace proběhla úspěšně, přejde program ke stažení údajů o zpracovávaném případu testu. Všechny tyto údaje stahuje z databáze. Použitá struktura databáze je rozepsána dále v této kapitole.

5. **Provedení testu**

   Když má program všechny potřebné údaje k dispozici, začne přípravou k provedení testu. Příprava sestává z načtení všech assetů (modelů, textur, shaderových programů, ...) a spustí inicializační kód testovaného algoritmu.

   Po dokončení přípravy začne program měřit. Postupně renderuje všechny snímky z daného případu testu a obraz zobrazí v okně. Podle nastavení také obraz uloží do souboru nebo zpracuje v knihovně SAGElib pro zobrazení na velkoplošné telestěně.

   V průběhu renderování měří všechny sledované parametry (časy jednotlivých fází, počty volání OpenGL funkcí apod.) a po skončení každého snímku tyto údaje uloží do databáze.

   Po dokončení aktuálního případu testu přejde program k dalšímu podle nastavení.

6. **Zobrazení výsledků**

   Všechny výsledky získané měřením jsou uložené v databázi. Pro snadné zpracování těchto výsledků databáze obsahuje několik pohledů pro zjednodušení načítání dat.

Kromě zobrazení výsledků přímo v databázi bude k dispozici také webová stránka. Na této stránce bude možnost si specifikovat dotaz. Lze si zvolit hodnoty na ose x, hodnoty na ose y a případné operace na ostatních hodnotách (min, max, avg, ...) včetně případného seskupení (group by) nebo podmínění (where).

V neposlední řadě jsou některé vybrané snímky z každého případu testu ukládány jako obrázky. To umožní následné porovnání vizuální kvality nebo hledání případných rozdílů.

7. **Změna testovaných algoritmů**

   Program lze upravit a doplnit o další renderery nebo obecně jakékoli algoritmy pro jejich srovnání. Provedené změny je nutné reflektovat také v databázi.

   Po úpravě algoritmů může být nutné program překompilovat.

3.1.1 Poznámka k databázi

Požadavky, jejichž smyslem je úprava struktury nebo dat v databázi nejsou zahrnuty mezi funkčními požadavky i přes to, že tato funkcionalita je v seznamu případů užití předpokládána.

Na správu databáze (struktury i dat samotných) předpokládám použití nástrojů třetích stran.

3.2 Nefunkční požadavky

1. Měřící program napsaný v c++
2. Měřící program musí běžet na operačním systému Microsoft Windows
3. OpenGL minimální verze 3.3 (může vyžadovat úpravu zdrojového kódu), doporučená verze je 4.2
4. Knihovna SAGElib
5. Data ukládána do databáze MySQL
6. Webový server s podporou PHP

3.3 Případy užití

1. **Správa porovnávaných algoritmů**

   a) Přidání, odebrání a přejmenování algoritmu

   b) Úprava implementace algoritmu (c++)

c) Úprava shaderových kódů algoritmu

2. **Správa rozlišení**

   a) Přidání, odebrání, změna a přejmenování rozlišení

3. **Správa scén**

   a) Přidání, odebrání, přejmenování, změna doby trvání a nastavení screenshotu kamery
   b) Přidání, odebrání, nastavení frame indexu, nastavení pozice a nastavení orientace kamery
   c) Přidání, odebrání, změna ořezových rovin a změna ambientní barvy scény
   d) Přidání, odebrání a změna typu světla ve scéně
   e) Přidání, odebrání, nastavení frame indexu, pozice, směru, útlumu, úhlu, exponentu a difusní i spekulární barvy světla
   f) Přidání, odebrání, nastavení pozice a orientace instancí modelů ve scéně
   g) Přidání, odebrání, přejmenování a nastavení počtu vrcholů a indexů modelu

4. **Správa assetů**

   a) Přidání, odebrání, změna a přejmenování assetu
   b) Procesování assetů do formátů používaných aplikací

5. **Správa případů testů**

   a) Smazání případů testů a všech přidružených dat
   b) Vygenerování případů testů pro všechny přípustné kombinace kamer, scén, počtů světel, rozlišení a algoritmů
   c) Vygenerování zvolené podmnožiny všech přípustných kombinací případů testů

6. **Nastavení měření**

   a) Nastavení připojení k databázi
   b) Nastavení cesty, kam se mají ukládat screenshoty
   c) Nastavení intervalu případů testů
   d) Nastavení rozlišení okna pro průběžné zobrazování
   e) Nastavení módu (žádné, okno, fullscreen) okna pro průběžné zobrazování

   f) Nastavení, zda se má používat sage
   g) Nastavení, zda se mají ukládat screenshoty
   h) Nastavení, zda ukládané screenshoty ukládat v png nebo raw formátu

7. **Správa měření**
   a) Uložení nebo smazání parametrů počítače
   b) Provedení měření celého rozsahu případů testů
   c) Provedení měření na zvoleném intervalu případů testů
   d) Provedení měření těch případů testů, které ještě na aktuálním počítači nebyly dokončeny
   e) Smazání záznamů měření

8. **Zobrazení výsledků**
   a) Nastavení zvětšení
   b) Výběr závislosti čeho (svislá osa)
      - čas renderování (pět fází + čas prohození bufferů)
      - čas renderování v průměru na pixel
      - čas renderování v průměru na trojúhelník
      - čas přípravy rendereru
      - čas renderování konkrétní fáze (1 až 5)
      - součet časů renderování pěti fází
      - čas prohození bufferů
      - celkový čas snímku (pět fází + čas prohození bufferů + čekání na další vlákna)
      - celkový počet volání OpenGL funkcí (počet funkcí uniform + kreslení + ostatní)
      - počet volání funkcí nastavujících uniformní proměnné
      - počet volání funkcí pro kreslení
      - počet volání ostatních OpenGL funkcí
      - celkový počet položek (místo agregace)
      - zadání vlastního výrazu
   c) Výběr agregace závislosti čeho (svislá osa)
      - průměr
      - rozptyl
      - minimum
      - maximum

- součet

d) Možnost povolit logaritmickou škálu

e) Výběr závislostí na (vodorovná osa) — výběr více hodnot

- renderer / algoritmus
- rozlišení
- počet světel
- kamera
- scéna
- počítač
- index snímku

f) Výběr seskupení (rozdělení na barvy) — výběr více hodnot

- stejné možnosti jako u „Výběr závislostí na (vodorovná osa)“

g) Nastavení filtrů

- stejné možnosti jako u „Výběr závislostí na (vodorovná osa)“
- vlastní filtr

h) Nastavení vlastního uspořádání

i) Zobrazit výsledný SQL dotaz

j) Zobrazit tabulku s výsledky dotazu

k) Zobrazit graf

3.4 Zvolené vlastnosti metodiky

Ze všech vstupních parametrů mě zajímá především rozlišení, počet světel, počet trojúhelníků, zvolený renderer a to vše pro několik různých scén.

Sledované parametry jsou časy jednotlivých fází renderování, celkový čas renderování, čas prohození bufferů a celkový čas snímku. Dále chci počítat počty volání OpenGL funkcí rozdělených do tří kategorií — počet nastavení uniformních proměnných, počet vykreslení (glDrawElements, glDrawArrays) a počet ostatních (glUseProgram, glActiveTexture, glBindTexture, glBindVertexArray, glBindFramebuffer, . . .).

Jako osvětlovací model používám tradiční Phongův model. Filtrování textur je nastaveno na lineární s lineárními mip-mapami. Používám 32 bitový barevný prostor (RGBa kanály, každý po 8 bitech). Všechny výpočty na CPU budou v single přesnosti (32 bitů na jedno číslo v plovoucí desetinné čárce). Vertikální synchronizace se řídí nastavením systému.

Scéna	Počet trojúhelníků	Textury (rozlišení)	Normálové mapy	Alpha mapy
Sponza Atrium	262 267	1 024	ano	ano
Head	17 684	4 096	ano	ne
Xeno Queen	19 448	2 048	ano	ne
Conference Room	331 179	žádné	ne	ne
Sibenik	75 283	512	ne	ne

Tabulka 3.1: Tabulka s napočítanými parametry scén.

Obrázek 3.1: Sponza Atrium 1

3.4.1 Scény

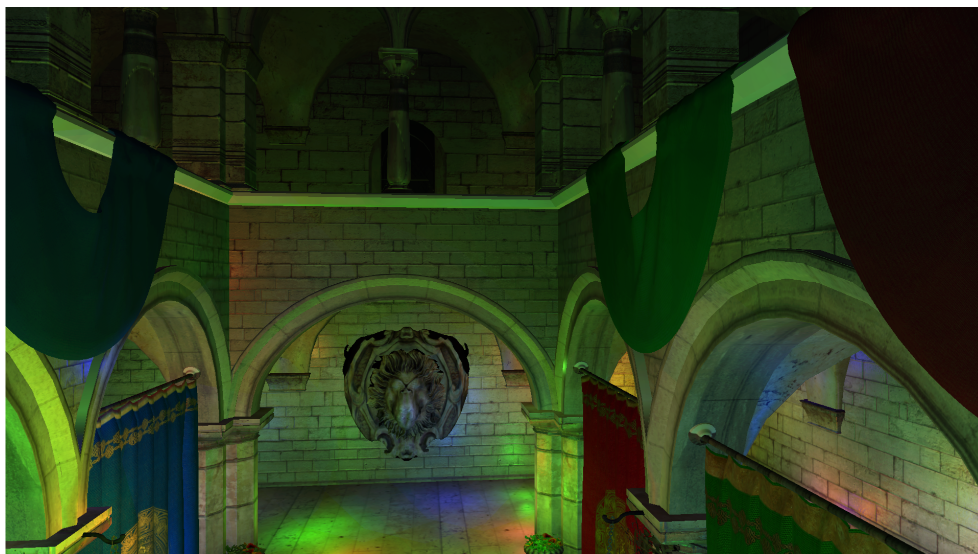
Zde je seznam scén vybraných pro provedení měření. Náhledy v této sekci jsou zpětně doplněné výsledné rendery z měřícího programu s použitím deferred699 rendereru. Scény jsou uvedeny ve stejném pořadí v jakém jsou zpracovávány v programu.

Detaily ke každé scéně jsou shrnuty v tabulce 3.1.

1. **Sponza Atrium**

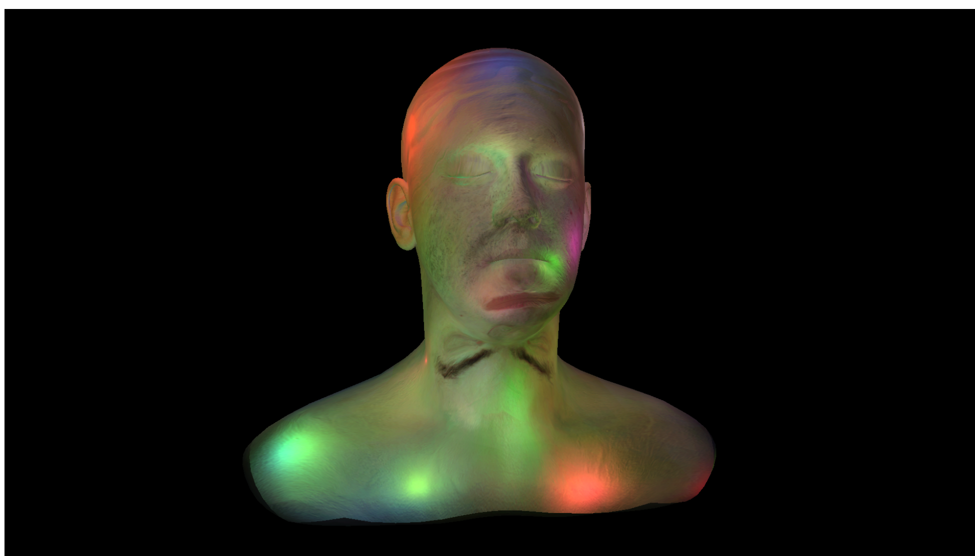
   Átrium paláce Sponza v Dubrovníku vymodelovaný Frankem Meinlem z Cryteku. Model upraven Morganem McGuirem. Model je dostupný pod licensí Creative Commons. Stažený ze stránky [2].

   Obrázky: 3.1, 3.2 a 3.3.



Obrázek 3.2: Sponza Atrium 2

Obrázek 3.3: Sponza Atrium 3 — starší verze



Obrázek 3.4: Head

2. **Head**

   Lee Perry-Smith je cenou odměněný umělec s celoživotní vášní pro záznam a snímání lidské podoby. Model této hlavy je jeho Morganem McGuirem upravený 3D sken. Model je také dostupný s licensí Creative Commons. Stažený ze stránky [2].

   Obrázek 3.4.

3. **Xeno Queen**

   Model xenomorph královny — jeden z vývojových článků fiktivní mimozemské rasy ze série Vetřelec, původně navržený H. R. Gigerem. Model stažený ze stránek TF3DM z adresy [7]. Licence dovoluje osobní použití.

   Obrázek 3.5.

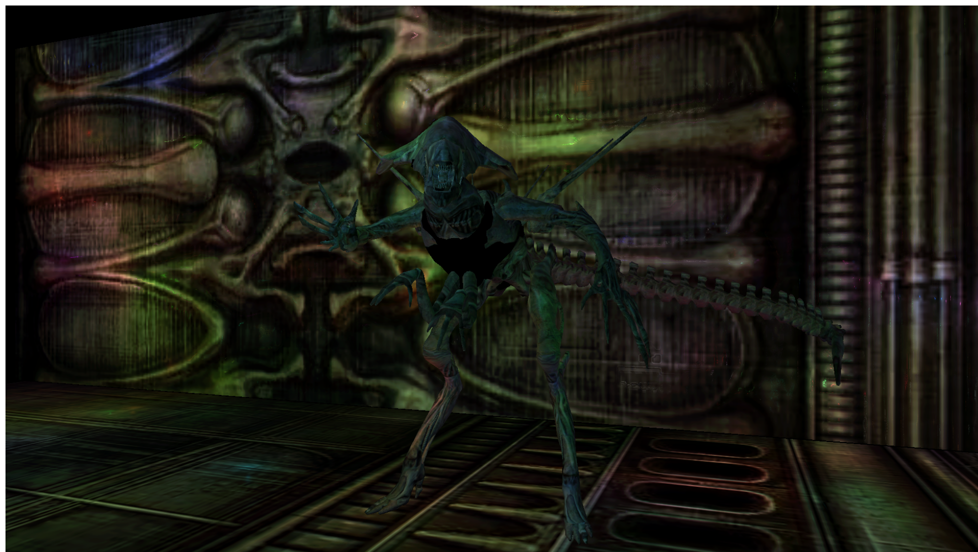
4. **Conference Room**

   Konferenční místnost vymodelovaná podle skutečné místnosti v Lawrance Berkeley National Laboratory 90, 3. patro, původně vymodelovaný Anatem Grynbergem a Gregem Wardem a opravený Kanzie Lamarem. Model stažený z [2] s licencí Creative Commons.

   Obrázek 3.6.

5. **Sibenik**

   Model interiéru Šibenické katedrály. Kdo vymodeloval první verzi nevím. Tato verze je upravená Kenzie Lamarem a Morganem McGuirem. Model stažený ze stránky [2] s licencí Creative Commons.



Obrázek 3.5: Xeno Queen

Obrázek 3.6: Conference Room

Obrázek 3.7: Sibenik

Obrázek 3.7.

3.4.2 Světla

Měření provádím s geometricky rostoucím počtem světel, tedy s 1, 2, 4, 8, ...512. Hranici 512 světel jsem určil s ohledem na, podle mých zkušeností, horní hranici užitečného počtu (například ve hrách). Dalším hlediskem je konečná doba provádění celého měření.

3.4.3 Rozlišení

Měření provádím na těchto rozlišeních:

1. nHD (640 * 360)
2. qHD (960 * 540)
3. WSVGA (1024 * 576)
4. HD (7280 * 720)
5. HD+ (1600 * 900)
6. FullHD (1920 * 1080)
7. QWXGA (2048 * 1152)
8. WQHD (2560 * 1440)

9. WQXGA+ (3200 * 1800)

10. UHD (3840 * 2160)

11. UHD+ (5120 * 2880)

12. FUHD (7680 * 4320)

Všechna tato rozlišení mají poměr stran 16:9.

Maximální rozlišení 8k jsem zvolil s ohledem na rozlišení podporované na telestěně na naší fakultě. Kromě toho větší rozlišení (16k) sice jde renderovat na mé grafické kartě pomocí forward htmlrendereru, ale v případě deferred rendereru by se potřebný g-buffer nevešel do paměti mé grafické karty.

3.4.4 Renderery

Původně jsem chtěl měření provést celkem na čtyřech rendererech: jedno-průchodový forward renderer s a bez hloubkového průchodu, více-průchodový forward renderer a samozřejmě deferred renderer. Konečné měření jsem ale provedl pouze s více-průchodovým forward rendererem a s deferred rendererem, protože na NVIDIA kartě na počítači od CESNETu jsem měl problém zkompilovat shader pro jedno-průchodový forward renderer. Podle předběžných měření na mém počítači ani jedna z variant jedno-průchodového forward rendereru nemohla zbývajícím dvěma konkurovat, takže výsledky lze i nadále považovat za relevantní.

Rozložení g-bufferu deferred rendereru je znázorněno na obrázku 3.8 (první dva render targety jsou použity i pro forward renderery). Abych předešel problému kvantování normál (nebo ho alespoň minimalizoval), používám na uložení normál formát R10G10B10U2 a poslední dva bity nevyužívám.

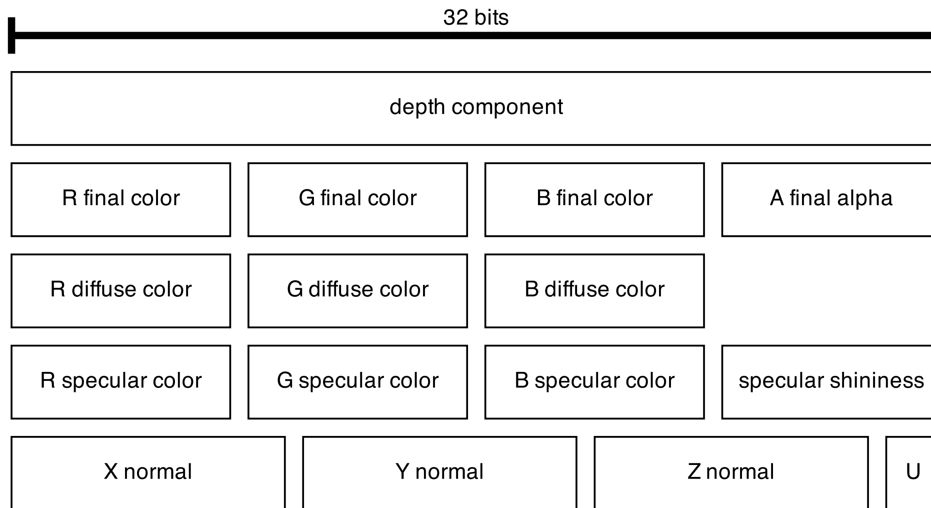
3.5 Architektura

V tomto modelu je znázorněna závislost jednotlivých knihoven na ostatních. Vyšší vrstva vždy využívá pouze rozhraní nižších vrstev. Grafické znázornění je na obrázku 3.9.

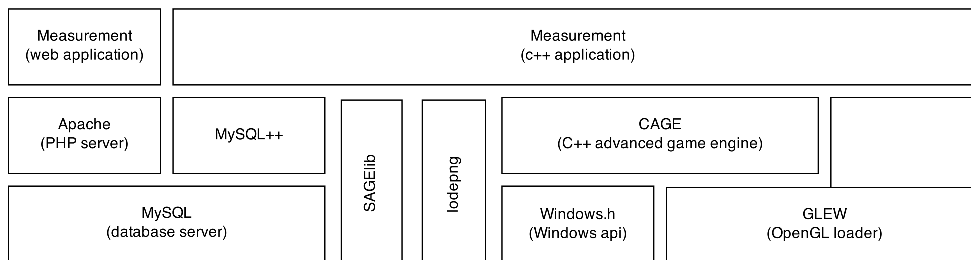
Seznam a popis použitých externích knihoven:

1. **CAGE**

   Všechny volně dostupné herní enginy, které znám, mají pouze forward renderer, nebo vyžadují velké množství úprav pro zprovoznění deferred rendereru. Také obvykle neumožňují dostatečně detailně řídit proces vykreslování tak, abych mohl přesně zaznamenávat všechny potřebné údaje. Z tohoto důvodu jsem se rozhodl použít knihovnu CAGE.



Obrázek 3.8: Rozložení g-bufferu v deferred rendereru.



Obrázek 3.9: Model architektury

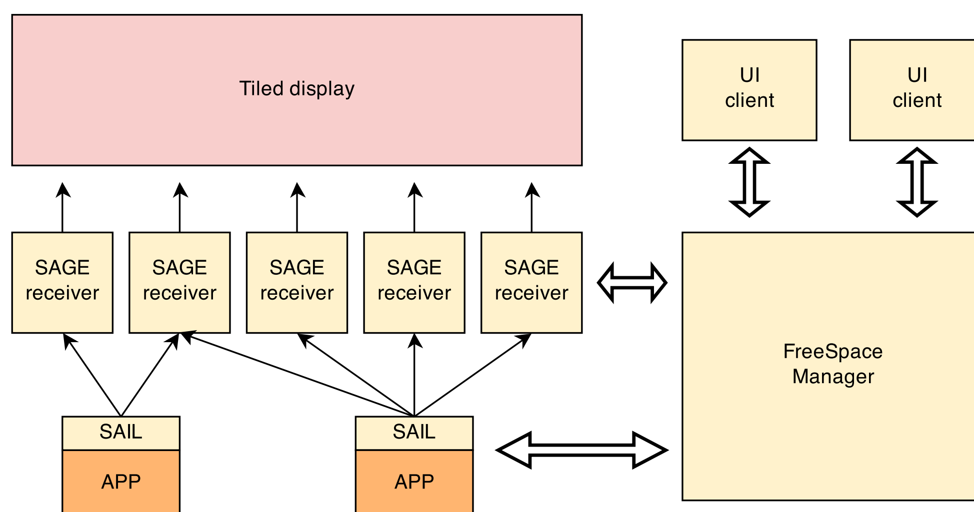
Knihovna CAGE poskytuje objektovou nadstavbu nad OpenGL, zahrnuje kompletní správu assetů (meshe, materiály, textury, shadery), poskytuje správu vláken, synchronizaci, io i oken a zjednodušuje některé další často prováděné nebo opakované akce. Přitom zachovává naprostou volnost a flexibilitu v samotném procesu renderování.

CAGE není veřejně dostupná.

2. **SAGElib**

   Pro práci s velkoplošnými zobrazovacími zařízeními je používána knihovna SAGElib.

   Na tomto modelu (obrázek 3.10) je znázorněno možné rozložení infrastruktury SAGE. *UI Client* slouží k organizaci oken na virtuální ploše. *SAGE Receiver* slouží k zobrazování přijímaných obrazových dat na monitorech, které jsou součástí velkoplošné složené obrazovky. Samotné aplikace pak vrstvě *SAIL* (komponenta knihovny SAGElib) předávají



Obrázek 3.10: Model SAGE

kompletní obrazové informace a *SAIL* je pak podle instrukcí z *FreeSpace Manageru* rozřezává a rozesílá jednotlivým *SAGE Receiverům*.

Jak je vidět, vrstvu *SAIL* je potřeba do aplikace přímo přidat ve formě úpravy zdrojového kódu nebo v podobě pluginu.

Alternativou k úpravě zdrojového kódu aplikace nebo k psaní doplňku by bylo použití grabberu. Grabber zachytává přímo video paměť a její obsah předává do své vlastní *SAIL* vrstvy. Grabování nemusí u všech aplikací fungovat.

V případě OpenGL je další možností plně obejít knihovnu SAGElib a použít Chromium pro distribuované zpracování OpenGL příkazů přímo na všech zobrazovacích nodech. Bohužel Chromium již delší dobu není vyvíjeno a podporuje pouze archaické OpenGL verze 1.5. Více informací o Chromium na adrese [1].

Více o knihovně SAGElib na stránkách [6] a [4].

Bohužel problémy s kompilací knihovny na Microsoft Windows v release konfiguraci mě donutily provést měření v debug buildu s vypnutými optimalizacemi. Jelikož se tyto optimalizace týkají pouze procesoru, mělo by to mít na výsledky zanedbatelný dopad.

3. **MySQL++**

   Pro připojení k MySQL serveru používám knihovnu MySQL++. Tato knihovna poskytuje velmi příjemné objektové rozhraní velmi blízké používání STL kontejnerů. Navíc díky dlouhé době komunitního vývoje je knihovna velmi vyzrálá a stabilní, proto jsem jí dal přednost před

použitím oficiálního MySQL c++ konektoru, se kterým jsem měl velké problémy.

Více informací o MySQL++ na adrese [3].

4. **LodePNG**

   Knihovnu LodePNG používám pro zakódování čistých obrazových dat do formátu png, abych je mohl uložit na disk v komprimované a zobrazitelné podobě. Více informací o knihovně na adrese [13].

3.6 Datový relační model databáze

Vstupním bodem celé databáze (obrázek 3.11) je tabulka *testcase*. Id řádků v této tabulce tvoří rozsah případů užití, s kterými pak pracuje měřicí aplikace. Pro každý z těchto řádek aplikace vytvoří jeden záznam v tabulce *test* a záznamy v tabulce *result* — jeden pro každý snímek v případu testu. Všechny řádky z tabulky *test* jsou nakonec sjednoceny záznamem v tabulce *measurement* — jeden záznam za každé spuštění aplikace — a svázány se záznamem v tabulce *computer*.

Sama tabulka *testcase* vytváří vazbu mezi tabulkou *camera* (rekurzivně tabulky *scene*, *model*, *instance* a *light* i *light key*) a tabulkami *resolution* a *renderer*.

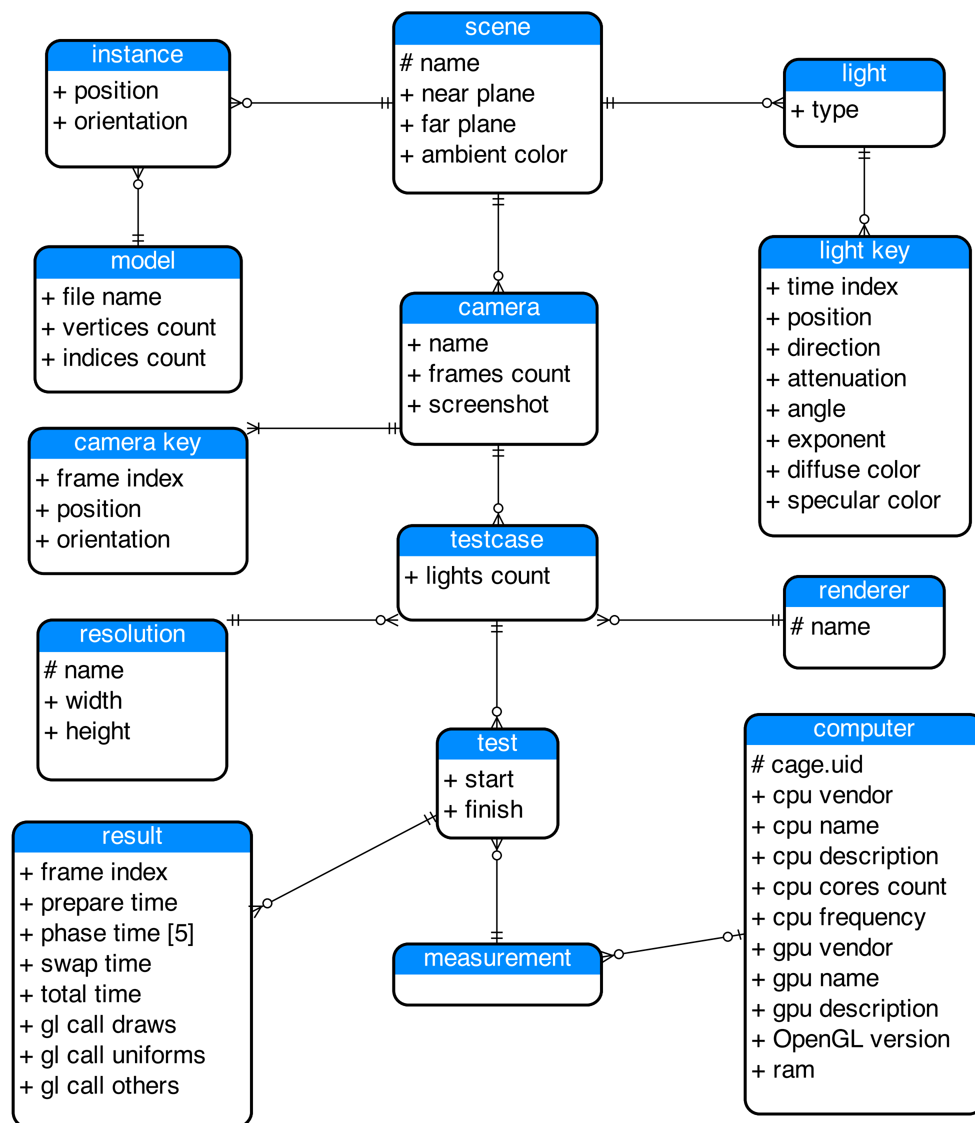
3.6.1 Pohledy

Kromě datových tabulek databáze obsahuje také několik pohledů, které slouží ke zjednodušení práce.

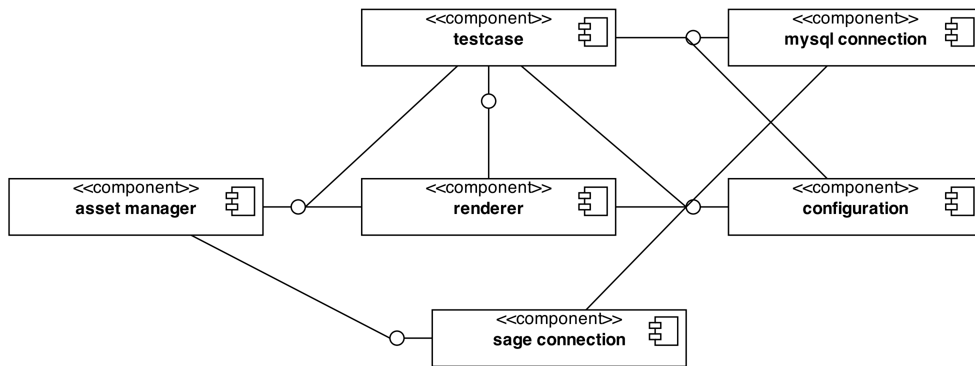
Jedním z takových pohledů je *view_triangles*, který pro zadanou scénu spočítá celkový počet trojúhelníků (za předpokladu, že všechny použité modely se skládají právě z trojúhelníků). Druhým je *view_todo*, který vypisuje pro každý počítač ty případy testů, které na daném počítači nebyly dokončeny. Dále pohled *view_times*, který pro každý řádek výsledků dopočítává součet dob všech fází renderování (což je pravděpodobně ten údaj, který bude nejvíce žádaný).

Další pohled *view_result* shrnuje údaje z téměř všech tabulek a pohledů. Jsou v něm ke každému řádku z tabulky *result* přiřazeny údaje scény, rozlišení, použitý renderer, kamera a další. Díky němu není nutné všechny klauzule pro spojení tabulek psát v každém příkazu znova.

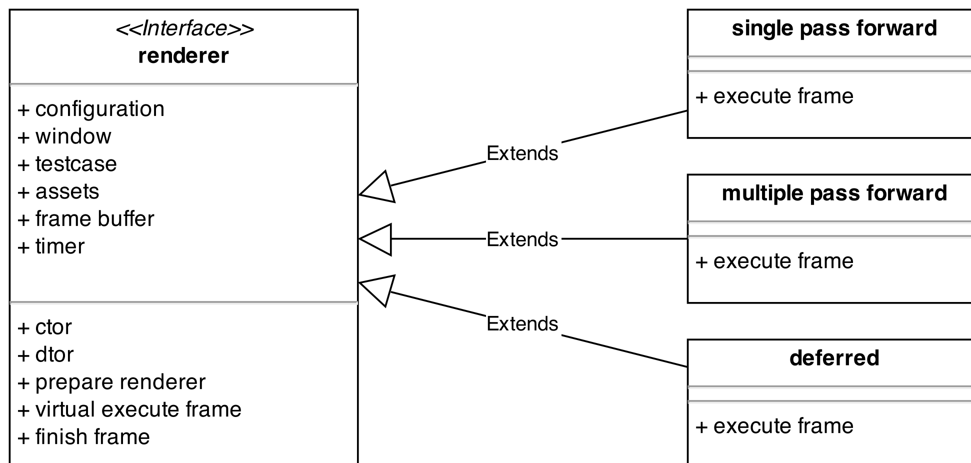
Nakonec pohled *view_testcase* se dá, v případě změn, použít pro snadné přegenerování řádků tabulky *testcase*.



Obrázek 3.11: Relační model databáze



Obrázek 3.12: Model komponent



Obrázek 3.13: Diagram dědičnosti rendererů

3.7 Model komponent

Tento model (obrázek 3.12) znázorňuje nejdůležitější komponenty c++ aplikace a zároveň znázorňuje vazby, která komponenta využívá rozhraní které komponenty. Jednotlivé komponenty přibližně odpovídají patřičným třídám, ačkoli kompletní doménový (nebo třídní) model by pro potřeby této práce byl zbytečně komplikovaný a rozsáhlý.

Samotná komponenta renderer je implementována jako abstraktní a jednotlivé renderery (nebo obecně porovnávané algoritmy) jsou implementovány díky dědičnosti, obrázek 3.13.

Measurement - result viewer. Font scale: 200 %. Content: Form

Graph of:	render time average
Log scale:	
Depends on:	renderer resolution lights camera scene computer frame
Colors by:	renderer resolution lights camera scene computer frame
Filters:	renderer · resolution · lights · camera · scene · computer · frame · Custom
Order by:	
Submit:	Show me the graph, please

Obrázek 3.14: Ukázka formuláře webové aplikace

3.8 Grafické znázornění výsledků

Pro zobrazení výsledků jsem připravil webovou stránku. Na straně serveru se zpracovává v PHP a výsledek je u klienta ještě dokončen v Javascriptu. Při přístupu na stránku, se vám zobrazí formulář pro výběr sledované vlastnosti (osy Y), závislosti (osy X), výběr, podle čeho se mají výsledky třídit do kategorií (obarvovat) a které výsledky se mají přeskočit. Dále formulář obsahuje možnost logaritmického zobrazení (o základu 2). A v neposlední řadě možnost zadat vlastní možnosti. Náhled formuláře je na obrázku 3.14.

Po nastavení a odkliknutí se, v případě, že nedošlo k chybě, zobrazí žádaný graf. Kromě něj je možné si na stránce zobrazit použitý SQL dotaz a také neupravenou tabulku s výsledkem dotazu.

Pro zobrazení grafu je použita javascriptová knihovna Chartjs. Tato knihovna je k dispozici pod MIT licencí. Více informací o knihovně na adrese [8].

3.9 Zdrojové kódy

Na přiloženém CD (viz přílohu Obsah přiloženého CD) jsou kompletní zdrojové kódy aplikace i webové stránky. Na CD jsou přiložené i všechny potřebné knihovny pro překompilování programu a také assety pro vlastní provedení měření. Pro kompilaci je, z důvodu kompatibility s c++ knihovnami, nutné

použít Microsoft Visual Studio 2010 SP1.

Pro transformaci assetů v originálních formátech (.obj, .mtl, .tga, .png, .tif, ...) do formátů používaných knihovnou CAGE použijte program cage-asseter.exe, který je také součástí přiloženého CD.

V příloze Pseudokódy je v pseudokódu zjednodušený a zkrácený přepis obou rendererů včetně shaderů.

Kapitola 4

Výsledky

V této závěrečné kapitole prezentuji a diskutuji naměřená data. V grafech jsem zcela úmyslně, pro větší přehlednost, vynechal některá rozlišení a některé počty světel. Pro zobrazení libovolného grafu použijte přiloženou webovou aplikaci.

Ačkoli to zadání nevyžaduje, měl jsem možnost provést měření na více počítačích:

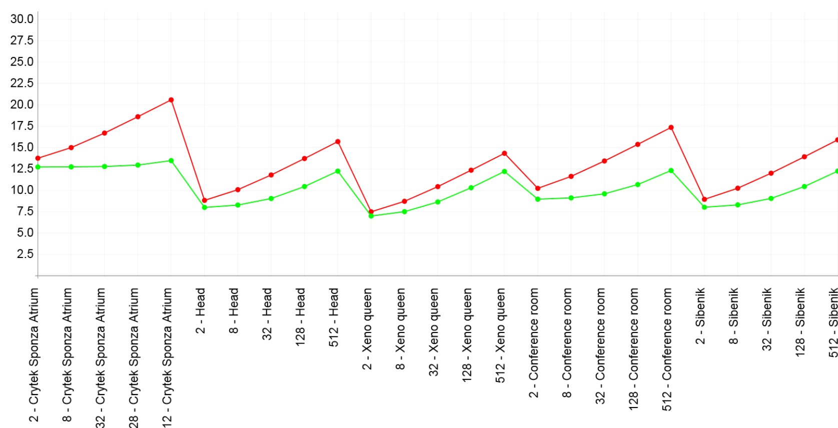
1. Můj notebook HP Pavilion dv6
    - cpu: Intel i7-2670QM, 2.2 GHz
    - ram: 8 GB
    - gpu: AMD Radeon HD 6770M
2. Počítač od CESNETu.
    - cpu: Intel Xeon ES-2640, 2.5 GHz
    - ram: 8 GB
    - gpu: NVIDIA GeForce GTX 780

V grafech jsou výsledky průměrovány ze všech měření.

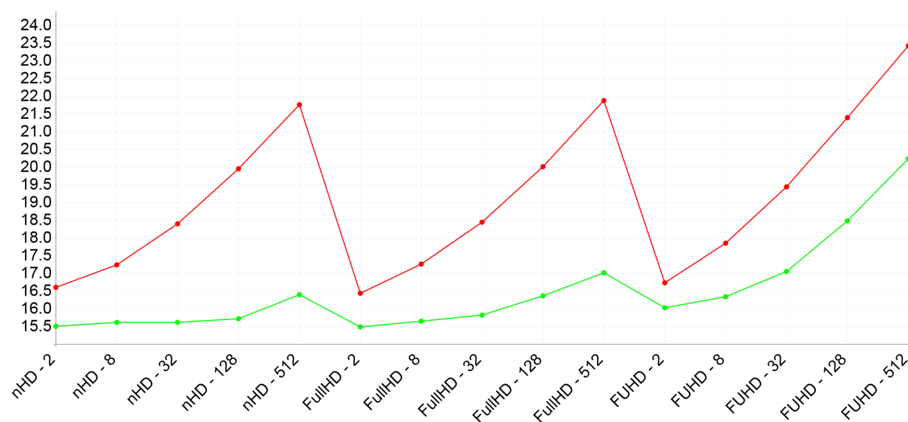
4.1 Počet volání

Jako první je graf 4.1 znázorňující (na logaritmické škále o základu 2) počet volání OpenGL funkcí v průměru na jeden snímek. Červená je forward multipass renderer a zelená je deferred renderer.

Z grafu je patrné, že Sponza Atrium je v tomto ohledu nejnáročnější.



Obrázek 4.1: Graf počtu volání OpenGL funkcí.



Obrázek 4.2: Časy renderování scény Sponza Atrium

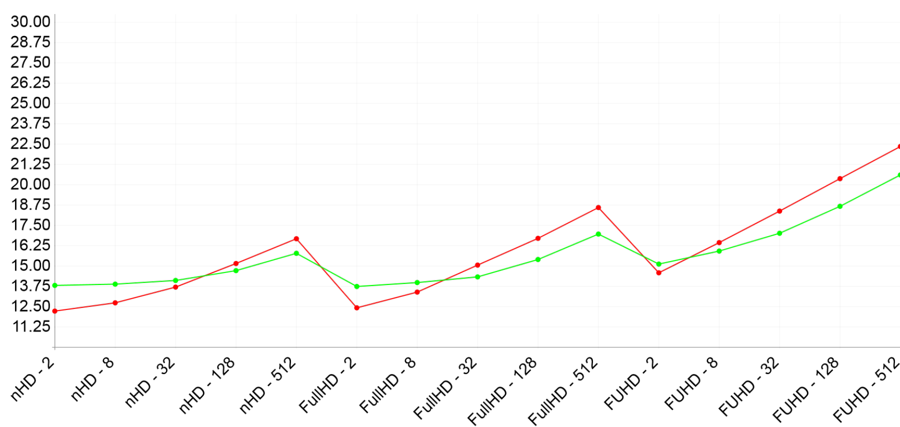
4.2 Časy renderování

V této sekci jsou, v jednotlivých podsekcích, postupně grafy, které znázorňují (opět na logaritmické škále o základu 2) časy renderování (čas přípravy plus čas všech fází — doba trvání prohození bufferů tedy není zahrnuta) v mikrosekundách. Čím menší hodnoty tím lepší.

4.2.1 Podle scény

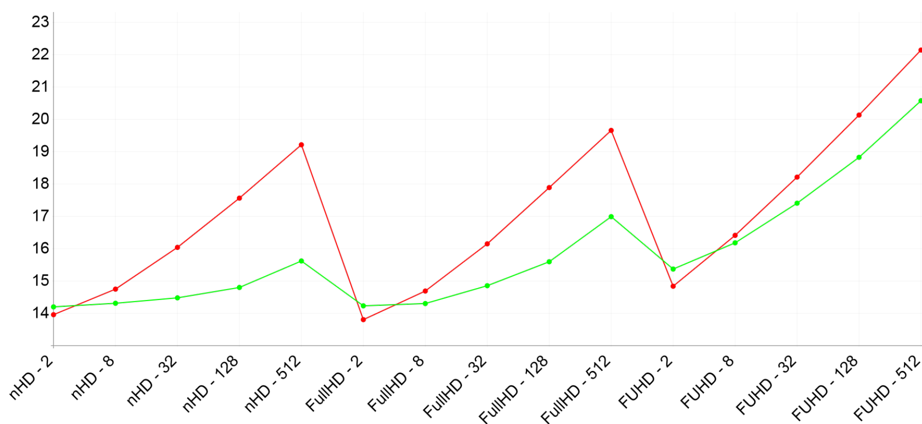
V této podsekci jsou grafy (4.2 4.3 4.4 4.5 4.6) závislosti času renderování na rozlišení a počtu světel a jsou rozdělené jednotlivě podle scén. Červená značí forward multipass renderer a zelená značí deferred renderer.

Z grafů je patrné, že u scén s obdobnou složitostí jako je Sponza Atrium, se vždy vyplatí použít deferred renderer bez ohledu na rozlišení nebo počet

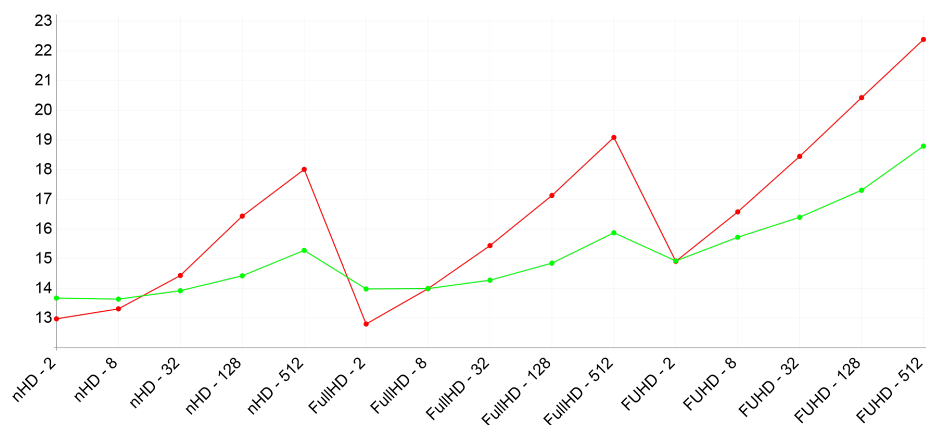


Obrázek 4.3: Časy renderování scény Xeno Queen

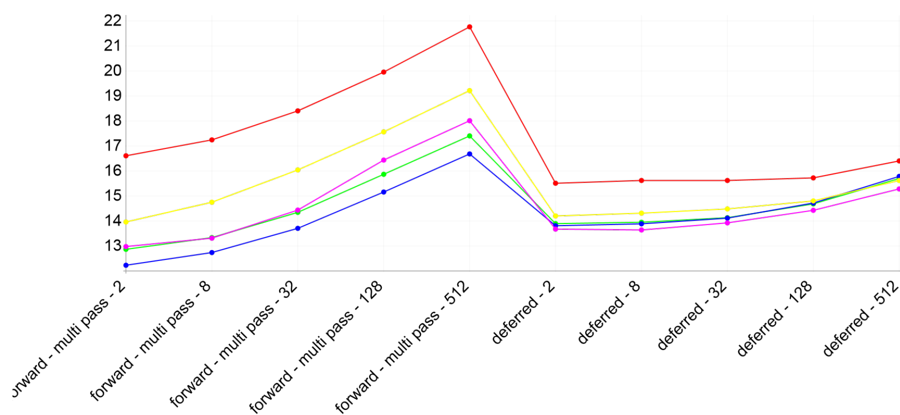
Obrázek 4.4: Časy renderování scény Head



Obrázek 4.5: Časy renderování scény Conference Room



Obrázek 4.6: Časy renderování scény Sibenik



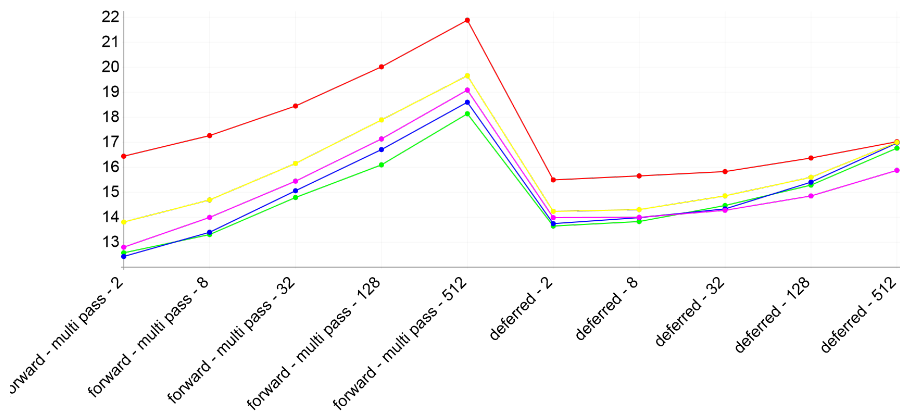
Obrázek 4.7: Časy renderování v rozlišení nHD

světel. U méně náročných scén se deferred vyplatí až od určitého počtu světel ve scéně a tento počet se většinou liší podle cíleného rozlišení.

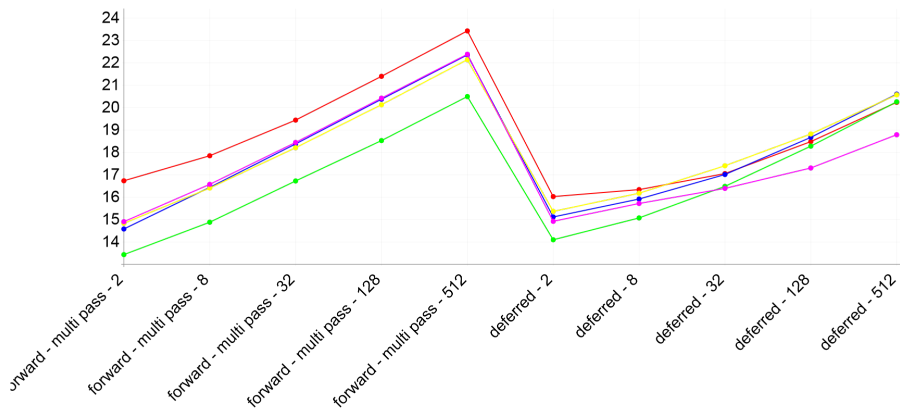
Dalším zajímavým poznatkem z grafů je, že, nezávisle na scéně, mezi nHD $(640 * 360)$ a FullHD $(1920 * 1080)$ je podstatně menší rozdíl než mezi FullHD a FUHD $(7680 * 4320)$. A tento rozdíl je neproporční vzhledem k počtu pixelů.

4.2.2 Podle rozlišení

V této podsekci jsou grafy (4.7 4.8 4.9) závislosti času renderování na rendereru a počtu světel a jsou rozděleny podle rozlišení. Jednotlivé barvy odpovídají jednotlivým scénám. Červená je Sponza Atrium, zelená je Head, modrá je Xeno Queen, žlutá je Conference room a konečně růžová je Sibenik.



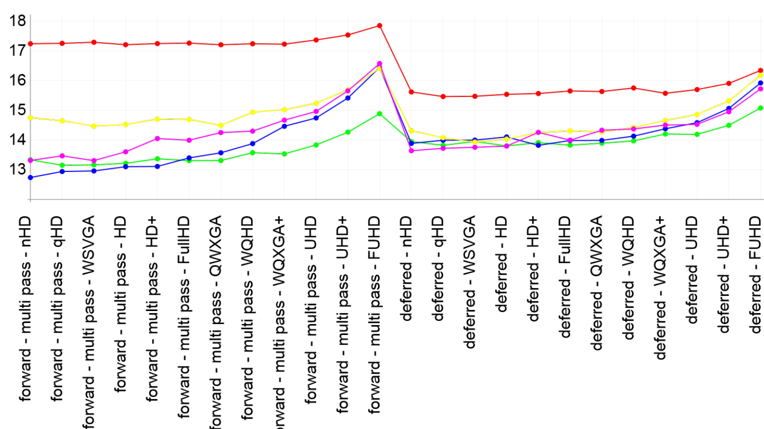
Obrázek 4.8: Časy renderování v rozlišení FullHD



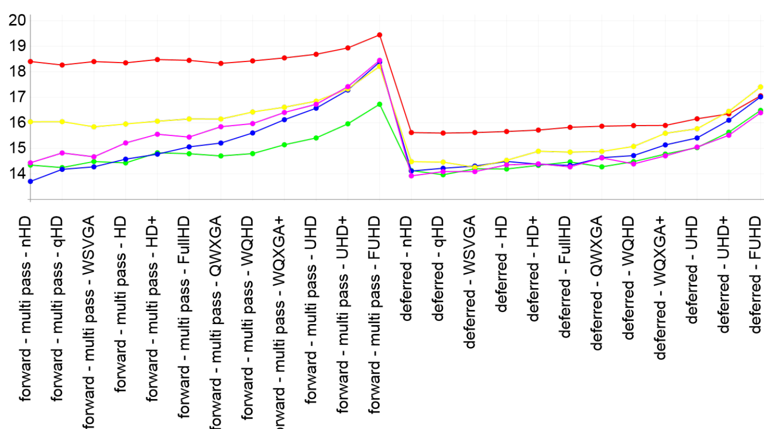
Obrázek 4.9: Časy renderování v rozlišení FUHD

Grafy opět potvrzují, že rozlišení menší než FullHD ($1920 * 1080$) se navzájem liší jen velmi málo. Současně u těchto rozlišení je patrné dosažení cíle deferred rendereru, tedy čas potřebný k renderování narůstá pouze lineárně (v grafech logaritmicky) vzhledem k počtu světel. Naproti tomu časy renderování u forward rendereru narůstají exponenciálně (v grafech lineárně) vzhledem k počtu světel.

Zajímavý je graf pro FUHD ($7680 * 4320$). Pro počty světel větší než 8 stále platí, že deferred renderer se vyplatí. Ovšem nárůst času renderování v závislosti na počtu světel už roste strměji, než bylo pozorováno u nižších rozlišení.



Obrázek 4.10: Časy renderování s 8 světly



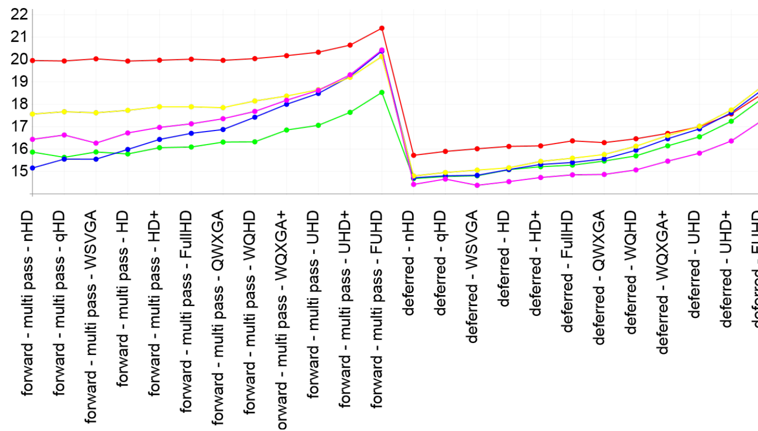
Obrázek 4.11: Časy renderování s 32 světly

4.2.3 Podle počtu světel

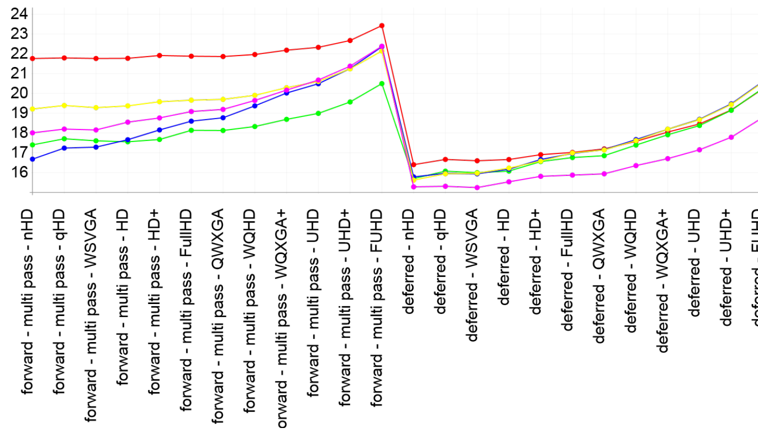
V této podsekci jsou grafy (4.10 4.11 4.12 4.13) závislosti času renderování na rendereru a rozlišení a jsou rozděleny podle počtu světel. Jednotlivé barvy, stejně jako v sekci Podle rozlišení odpovídají jednotlivým scénám. Červená je Sponza Atrium, zelená je Head, modrá je Xeno Queen, žlutá je Conference room a konečně růžová je Sibenik.

4.3 Průměr na pixel respektive trojúhelník

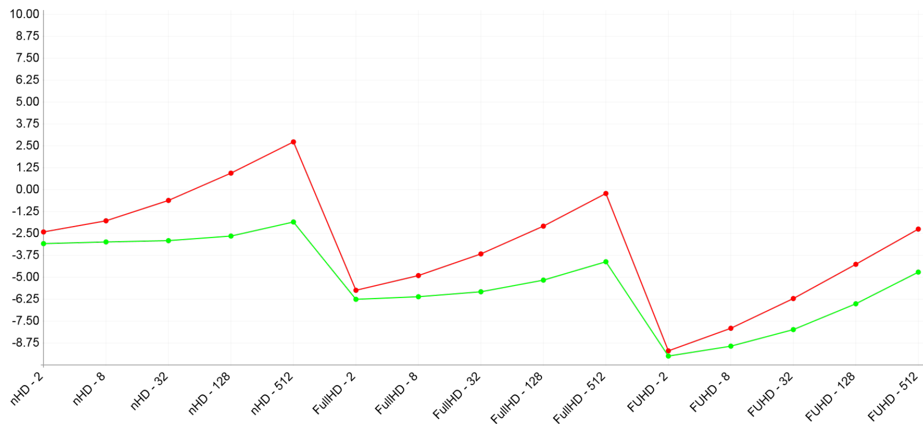
Zde jsou grafy (4.14 4.15) závislosti času renderování na rozlišení a počtu světel, ovšem tentokrát zobrazují průměr na jeden pixel respektive trojúhelník. Červená značí forward multipass renderer a zelená deferred renderer.



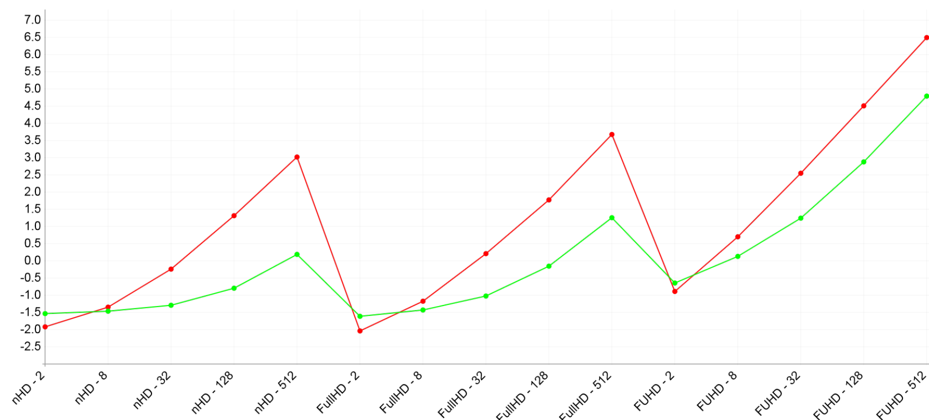
Obrázek 4.12: Časy renderování s 128 světly



Obrázek 4.13: Časy renderování s 512 světly



Obrázek 4.14: Časy renderování průměrně na pixel



Obrázek 4.15: Časy renderování průměrně na trojúhelník

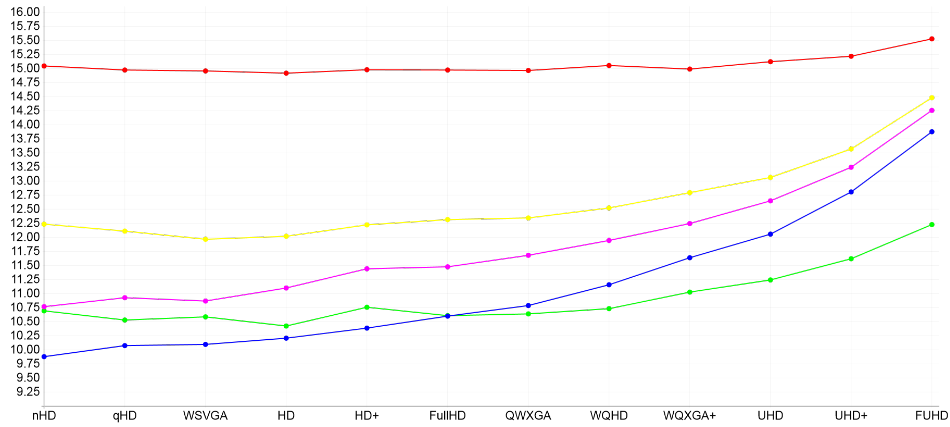
Se zvětšujícím se rozlišením očividně klesá čas na výpočet jednoho pixelu. To se dá vysvětlit zvětšující se periodou mezi režijními funkcemi jako je vyčištění nebo prohození bufferů.

S rostoucím rozlišením přirozeně roste čas potřebný na zpracování každého trojúhelníku. Užitečným poznatkem je, že se zvětšujícím se rozlišením klesá hranice počtu světel, od které je deferred renderer výhodnější než forward renderer.

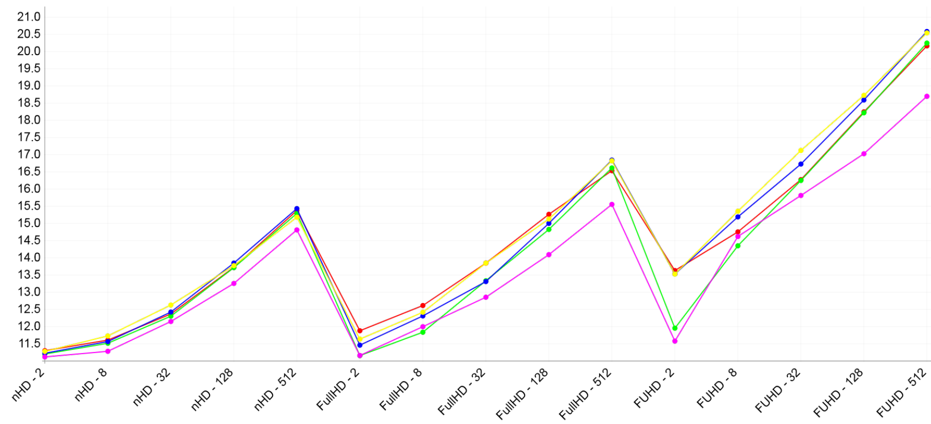
4.4 Deferred renderer

V těchto dvou grafech (4.16 4.17) jsou znázorněny časy v mikrosekundách (logaritmická škála o základu 2) jednotlivých fází deferred rendereru v závislosti na rozlišení respektive rozlišení a počtu světel. Jednotlivé barvy, stejně jako v sekci Podle rozlišení, odpovídají jednotlivým scénám. Červená je Sponza Atrium, zelená je Head, modrá je Xeno Queen, žlutá je Conference room a konečně růžová je Sibenik.

Je patrné, že časy osvětlovacího průchodu jsou výrazně závislé na počtu světel a rozlišení, ale mezi jednotlivými scénami už se příliš neliší. Což je přesně na opak než u průchodu přípravy g-bufferu, kde čas závisí kromě rozlišení velmi výrazně i na scéně, tedy na použitých normálových mapách, texturách a podobně.



Obrázek 4.16: Časy přípravy g-bufferu



Obrázek 4.17: Časy osvětlovacího průchodu

Závěr

V této práci jsem se věnoval srovnání rychlosti dvou populárních grafických rendererů, jejichž fungování jsem zkráceně vysvětlil v kapitole Fungování deferred rendereru.

Pro provedení srovnání jsem v kapitole Metodika navrhl a popsal metodiku.

Podle této metodiky jsem následně zvolil scény a další vstupní parametry. Dále jsem napsal program, který podle dané metodiky provedl všechna měření a výsledky uložil do databáze, ze které na začátku načetl i vstupní nastavení. Program při svém běhu, podle nastavení, také ukládal zvolené snímky jako png obrázky. Všechny tyto informace jsou rozepsané v kapitole Systém pro měření.

Po napsání programu a provedení měření na dvou počítačích jsem ještě napsal webovou stránku pro interaktivní grafové zobrazení výsledků, ze které jsou vyexportované grafy prezentované v tomto dokumentu v kapitole Výsledky.

Všechny cíle stanovené v zadání se mi podařilo úspěšně naplnit.

Možnosti pokračování

Práce by se dala rozšířit v několika možných ohledech.

- Pro zpřesnění výsledků by bylo vhodné provést více měření na různých počítačových sestavách. Z takových měření by se pak dal odvodit i vliv různých hardwarových komponent na rychlost renderování. Nebo třeba znázornit, která ze dvou největších firem vyrábějících grafické karty (NVIDIA a AMD) má navrch při použití různých algoritmů nebo scén.

- Ačkoliv deferred renderer, nepočítaje původní forward renderer, je podle mého odhadu nejpopulárnější mezi real-time grafickými (herními) renderery, rozhodně není jedinou možností. Další alternativy jsem rozepsal

v sekci Varianty. Zde popsané ale i další alternativy by také bylo vhodné naimplementovat a zahrnout je do měření.

- Vedle samotné podoby rendereru je také možné do práce zahrnout další grafické techniky. Obzvlášť zajímavé možnosti by byly techniky pro vrhání stínů a nějaké screen-space efekty jako anti-aliasing, motion-blur, light-shafts nebo pro renderování využít techniky HDR, indirect-lighting a další. Oproti různým druhům rendererů by bylo možné tyto techniky nezávisle na ostatním nastavení zapínat a vypínat a tím měřit vliv těchto technik samostatně nebo naopak sledovat vliv těchto technik na ostatní algoritmy.
- Posledním směrem, kterým by se práce mohla dále rozvíjet, je doplnění dalších scén. Mezi scénami rozhodně chybí exteriér, který nebyl zahrnut z důvodu složitosti implementace techniky prořezávání scény, bez které takové scény nemá smysl do měření zanášet.

Literatura

[1] Chromium Documentation. Dostupné z: http://chromium.sourceforge.net/doc/index.html

[2] McGuire Graphics Data. Dostupné z: http://graphics.cs.williams.edu/data/meshes.xml

[3] MySQL++. Dostupné z: http://tangentsoft.net/mysql++/

[4] SAGE Documentation. Dostupné z: http://www.evl.uic.edu/cavern/sage/documentation/SAGEdoc.htm

[5] SAGELab. Dostupné z: http://sage.fit.cvut.cz/about/

[6] Scalable Adaptive Graphics Environment. Dostupné z: http://www.sagecommons.org/

[7] TF3DM. Dostupné z: http://tf3dm.com/3d-model/acm-xeno-queen-57998.html

[8] Downie, N.: Chart.js. Dostupné z: http://www.chartjs.org/

[9] Hargreaves, S.: Deferred Shading. Dostupné z: http://www.talula.demon.co.uk/DeferredShading.pdf

[10] Olsson, O.; Assarsson, U.: Tiled Shading. Dostupné z: http://www.cse.chalmers.se/~olaolss/main_frame.php?contents=publication&id=tiled_shading

[11] Pranckevičius, A.: Compact Normal Storage for Small G-Buffers. Dostupné z: http://aras-p.info/texts/CompactNormalStorage.html

[12] Trebilco, D.: Light Indexed Deferred Lighting. Dostupné z: https://code.google.com/p/lightindexed-deferredrender/

[13] Vandevenne, L.: Lode PNG. Dostupné z: http://lodev.org/lodepng/

Příloha A

Pseudokódy

A.1 Forward renderer

```
void createMRT ()
{
  // create textures for use in MRT
  depthTexture = newTextureBufferComponent (maxRenderWidth,
      maxRenderHeight, GL_DEPTH_COMPONENT32, GL_DEPTH_COMPONENT,
      GL_FLOAT);
  colorTexture = newTextureBufferComponent (maxRenderWidth,
      maxRenderHeight, GL_RGBA8, GL_RGBA, GL_UNSIGNED_BYTE);
  // frame buffer for final image
  colorBuffer = newFrameBuffer ();
  frameBufferSetDepthTexture (depthTexture);
  frameBufferSetColorTexture (0, colorTexture);
  frameBufferCheckStatus ();
}

void prepare ()
{
  createMRT ();
  colorBuffer.bind();
  glViewport (0, 0, renderWidth, renderHeight);
  glEnable (GL_DEPTH_TEST);
  glDepthFunc (GL_LEQUAL);
  glEnable (GL_CULL_FACE);
  glEnable (GL_BLEND);
}

void renderFrame ()
{
  glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  foreach (model in models)
  {
    shader = model.getShader();
```

```
32      shader.bind();
33      shader.setUniform("view", view);
34      shader.setUniform("projection", projection);
35      material = model.getMaterial ();
36      material.dispatch(shader);
37      mesh = model.getMesh ();
38      mesh.bind();
39      foreach (instance in model.instances)
40      {
41        mat4 model = mat4 (instance.orientation) * mat4 (vec4
              (instance.position, 1));
42        mat4 modelview = view * model;
43        shader.setUniform("modelview", modelview);
44        shader.setUniform("normal", modelToNormal(modelview));
45
46        uint32 processed = 0;
47        foreach (light in pointLights)
48        {
49          real radius = lightRadius (ceil (light.diffuse,
                light.specular), light.attenuation);
50          vec3 pos (radius, radius, radius);
51          aabb lb = aabb (vec3(light.position) - pos,
                vec3(light.position) + pos);
52          if (!intersects (mesh.getBox(), lb))
53            continue;
54
55          if (processed == 0)
56          {
57            shader.setUniform ("light_ambient", ambient);
58            glBlendFunc (GL_ONE, GL_ZERO);
59          }
60          else if (processed == 1)
61          {
62            shader.setUniform ("light_ambient"), vec3 ());
63            glBlendFunc (GL_ONE, GL_ONE);
64          }
65
66          shader.setUniform ("light_diffuse", light.diffuse);
67          shader.setUniform ("light_specular", light.specular);
68          shader.setUniform ("light_attenuation",
                light.attenuation);
69          shader.setUniform ("light_position", view *
                light.position);
70          shader.setUniform ("light_angle", cos (light.angle));
71          shader.setUniform ("light_exponent", light.exponent);
72
73          mesh->dispatch ();
74        }
75
76      }
77    }
78
79    // finalize
80    // perform any screen-space effects
```

```
81    // swap buffers
82 }
```

Listing A.1: forward renderer c++

```
1
2  uniform mat4 modelview;
3  uniform mat4 projection;
4  uniform mat3 normal;
5
6  out vec3 var_position;
7  out vec2 var_uv;
8  out vec3 var_normal;
9
10 layout(location = 0) in vec3 in_position;
11 layout(location = 1) in vec3 in_normal;
12 layout(location = 2) in vec3 in_tangent;
13 layout(location = 3) in vec3 in_bitangent;
14 layout(location = 4) in vec2 in_uv;
15
16 void main()
17 {
18    var_position = (modelview * vec4(in_position, 1.0)).xyz;
19    var_uv = in_uv;
20    var_normal = normal * in_normal;
21    gl_Position = uni_projection * uni_modelview *
          vec4(in_position, 1.0);
22 }
```

Listing A.2: forward renderer vertex shader

```
1
2  uniform vec3 light_diffuse;
3  uniform vec3 light_specular;
4  uniform vec3 light_direction;
5  uniform vec3 light_attenuation;
6  uniform vec4 light_position;
7  uniform float light_exponent;
8  uniform vec3 light_ambient;
9
10 uniform vec3 material_diffuse;
11 uniform vec4 material_specular;
12 uniform vec3 material_ambient;
13 uniform vec3 material_emission;
14
15 in vec3 var_position;
16 in vec2 var_uv;
17 in vec3 var_normal;
18
19 void main ()
20 {
21    vec3 C = -normalize(var_position);
22    vec3 N = normalize(var_normal);
23
```

```
24    // do calculate lighting (diff & spec)
25
26    out_color = vec4(vec3(
27        material_ambient * light_ambient
28      + material_diffuse * diff
29      + material_specular.rgb * spec
30      + material_emission
31      ), 1.0);
32  }
```

Listing A.3: forward renderer fragment shader

A.2 Deferred renderer

```
1
2   void createMRT ()
3   {
4     // create textures for use in MRT
5     depthTexture = newTextureBufferComponent (maxRenderWidth,
          maxRenderHeight, GL_DEPTH_COMPONENT32, GL_DEPTH_COMPONENT,
          GL_FLOAT);
6     colorTexture = newTextureBufferComponent (maxRenderWidth,
          maxRenderHeight, GL_RGBA8, GL_RGBA, GL_UNSIGNED_BYTE);
7     rtDiff = newTextureBufferComponent (maxRenderWidth,
          maxRenderHeight, GL_RGB8, GL_RGB, GL_FLOAT);
8     rtSpec = newTextureBufferComponent (maxRenderWidth,
          maxRenderHeight, GL_RGBA8, GL_RGBA, GL_FLOAT);
9     rtNorm = newTextureBufferComponent (maxRenderWidth,
          maxRenderHeight, GL_RGB10_A2, GL_RGB, GL_FLOAT);
10    // frame buffer for use as g-buffer
11    gBuffer = newFrameBuffer ();
12    frameBufferSetDepthTexture (depthTexture);
13    frameBufferSetColorTexture (0, colorTexture);
14    frameBufferSetColorTexture (1, rtDiff);
15    frameBufferSetColorTexture (2, rtSpec);
16    frameBufferSetColorTexture (3, rtNorm);
17    frameBufferSetDrawAttachments (4);
18    frameBufferCheckStatus ();
19    // frame buffer for final image
20    colorBuffer = newFrameBuffer ();
21    frameBufferSetDepthTexture (depthTexture);
22    frameBufferSetColorTexture (0, colorTexture);
23    frameBufferCheckStatus ();
24  }
25
26  void prepare ()
27  {
28    createMRT ();
29    glViewport (0, 0, renderWidth, renderHeight);
30    glDepthFunc (GL_LESS);
31    glEnable (GL_CULL_FACE);
32    glBlendFunc (GL_ONE, GL_ONE);
```

```
33 }
34
35 void renderFrame ()
36 {
37   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
38
39   // g-buffer pass
40
41   gBuffer.bind();
42   glEnable (GL_DEPTH_TEST);
43   glDisable (GL_BLEND);
44   glCullFace (GL_BACK);
45   foreach (model in models)
46   {
47     shader = model.getShader();
48     shader.bind();
49     shader.setUniform("view", view);
50     shader.setUniform("projection", projection);
51     material = model.getMaterial ();
52     material.dispatch(shader);
53     mesh = model.getMesh ();
54     mesh.bind();
55     foreach (instance in model.instances)
56     {
57       mat4 model = mat4 (instance.orientation) * mat4 (vec4
           (instance.position, 1));
58       mat4 modelview = view * model;
59       shader.setUniform("modelview", modelview);
60       shader.setUniform("normal", modelToNormal(modelview));
61       mesh.dispatch();
62     }
63   }
64
65   // lighting pass
66
67   colorBuffer.bind();
68   glDisable (GL_DEPTH_TEST);
69   glEnable (GL_BLEND);
70   glCullFace (GL_FRONT);
71
72   glActiveTexture (GL_TEXTURE0);
73   rtDiff.bind();
74   glActiveTexture (GL_TEXTURE1);
75   rtSpec.bind();
76   glActiveTexture (GL_TEXTURE2);
77   rtNorm.bind();
78   glActiveTexture (GL_TEXTURE3);
79   depthTexture.bind();
80
81   shader = deferredLightShader;
82   shader.bind();
83   shader.setUniform ("window_resolution", vec2 (renderWidth,
       renderHeight));
84   shader.setUniform ("projection", projection);
```

```
85    shader.setUniform ("lights_tranformation",
        projection.inverse());
86    shader.setUniform ("diffuse", 0);
87    shader.setUniform ("specular", 1);
88    shader.setUniform ("normal", 2);
89    shader.setUniform ("depth", 3);
90
91    mesh = deferredLightSphere;
92    mesh.bind();
93
94    foreach (light in pointLights)
95    {
96      real radius = lightRadius (ceil (light.diffuse,
          light.specular), light.attenuation);
97      mat4 modelview = mat4 (radius) * mat4 (light.position) * view;
98      shader.setUniform ("modelview", modelview);
99      shader.setUniform ("light_diffuse", light.diffuse);
100     shader.setUniform ("light_specular", light.specular);
101     shader.setUniform ("light_attenuation", light.attenuation);
102     shader.setUniform ("light_position", view * light.position);
103     shader.setUniform ("light_angle", cos (light.angle));
104     shader.setUniform ("light_exponent", light.exponent);
105     mesh->dispatch ();
106   }
107
108   // finalize
109   // perform any screen-space effects
110   // swap buffers
111 }
```

Listing A.4: deferred renderer c++

```
1
2  uniform mat4 modelview;
3  uniform mat4 projection;
4  uniform mat3 normal;
5
6  out vec3 var_position;
7  out vec2 var_uv;
8  out vec3 var_normal;
9
10 layout(location = 0) in vec3 in_position;
11 layout(location = 1) in vec3 in_normal;
12 layout(location = 2) in vec3 in_tangent;
13 layout(location = 3) in vec3 in_bitangent;
14 layout(location = 4) in vec2 in_uv;
15
16 void main()
17 {
18   var_position = (modelview * vec4(in_position, 1.0)).xyz;
19   var_uv = in_uv;
20   var_normal = normal * in_normal;
21   gl_Position = uni_projection * uni_modelview *
       vec4(in_position, 1.0);
```

```
22 }
```

Listing A.5: deferred renderer gbuffer vertex shader

```
1
2  #define SHININESS_MAX_VALUE 1024.0
3
4  layout(location = 1) out vec3 out_diffuse;
5  layout(location = 2) out vec4 out_specular;
6  layout(location = 3) out vec3 out_normal;
7
8  uniform vec3 material_diffuse;
9  uniform vec4 material_specular;
10 uniform vec3 material_ambient;
11 uniform vec3 material_emission;
12
13 in vec3 var_position;
14 in vec2 var_uv;
15 in vec3 var_normal;
16
17 void main ()
18 {
19   out_diffuse = material_diffuse;
20   out_specular = material_specular;
21   out_color = vec4 (material_ambient * light_ambient +
       material_emission, 1.0);
22   out_specular.a /= SHININESS_MAX_VALUE;
23   out_normal = normalize(var_normal) * 0.5 + 0.5;
24 }
```

Listing A.6: deferred renderer gbuffer fragment shader

```
1
2  uniform mat4 modelview;
3  uniform mat4 projection;
4
5  layout(location = 0) in vec3 in_position;
6
7  void main()
8  {
9    gl_Position = projection * modelview * vec4(in_position, 1.0);
10 }
```

Listing A.7: deferred renderer lighting vertex shader

```
1
2  #define SHININESS_MAX_VALUE 1024.0
3
4  in vec3 var_position;
5  in vec2 var_uv;
6  in vec3 var_normal;
7
8  layout(location = 0) out vec4 out_color;
```

```
9
10 uniform mat4 light_tranformation;
11 uniform vec2 window_resolution;
12
13 uniform sampler2D tex_depth; // mrt textures
14 uniform sampler2D tex_normal;
15 uniform sampler2D tex_diffuse;
16 uniform sampler2D tex_specular;
17
18 void main ()
19 {
20   ivec2 coord = ivec2(gl_FragCoord.xy);
21
22   // recompute position in view-space from fragment coordinates
       and depth
23   vec4 tmp = uni_lights_tranformation * (vec4(gl_FragCoord.xy /
       uni_window_resolution, texelFetch(tex_depth, coord, 0).x,
       1.0) * 2.0 - 1.0);
24   vec3 position = tmp.xyz / tmp.w;
25   vec3 C = -normalize(position);
26
27   // fetch normal from mrt
28   vec3 N = texelFetch (tex_normal, coord, 0).xyz * 2.0 - 1.0;
29
30   // fetch specular from mrt
31   vec4 specular = texelFetch (tex_specular, coord, 0);
32   specular.a *= SHININESS_MAX_VALUE;
33
34   // do calculate lighting (diff & spec)
35
36   out_color = vec4(vec3(
37       texelFetch(tex_diffuse, coord, 0).rgb * diff
38     + specular.rgb * spec
39     ), 1.0);
40 }
```

Listing A.8: deferred renderer lighting fragment shader

Příloha B

Seznam použitých zkratek a cizích slov

CPU central processing unit — hlavní procesor

DirectX proprietární knihovna pro akcelerované grafické programy od firmy Microsoft

fps frames per second — snímky za sekundu

fs fragment (pixel) shader

GPU graphics processing unit — grafická karta

HDR high-dynamic-range — technika, která pro uložení mezivýsledků používá buffer s větším rozsahem možných hodnot a tedy pracuje s větší přesností

MySQL databázový server v současnosti spravovaný firmou Oracle

OpenGL knihovna pro akcelerované grafické programy — otevřený standard v současnosti spravovaný skupinou KHRONOS GROUP, OpenGL je ochranná známka společnosti SGI

PHP PHP hypertext preprocessor — skriptovací jazyk a interpretr pro webové stránky běžící na straně serveru, v současnosti spadá pod firmu The PHP Group

png portable network graphics — formát obrázku vytvořený společností PNG Development Group

RAM random access memory — paměť s přímým přístupem — v této práci touto zkratkou označuji hlavní operační paměť

RGB red, green, blue — červená, zelená, modrá

SAGE Scalable Adaptive Graphics Environment — rozšiřitelné přizpůsobivé grafické rozhraní. Obecně se jedná o zařízení (nebo několik zařízení) s napojením na několik obrazovek najednou, které dohromady tvoří jedinou obrazovou plochu s velkým rozlišením (telestěna). Dále existuje knihovna SAGElib určená právě pro podporu aplikací na takovémto zařízení — podle této knihovny občas označuji celé zařízení. Více informací na [6].

STL standard template library — standardní knihovna šablon v jazyku c++

vs vertex shader

Příloha C

Obsah přiloženého CD

- `application` .... aplikace pro měření
  - `bin` .... spustitelná verze
  - `conf` .... ukázkové konfigurační skripty
  - `dep` .... předpřipravené závislostmi pro kompilaci
  - `resources` .... všechny assety nutné pro zopakování měření — v této složce jsou v původních formátech a před použitím je nutné je zpracovat do formátů používaných knihovnou CAGE
  - `src` .... zdrojové kódy
- `images` .... všechny vyrenderované obrázky
- `mysql` .... MySQL skripty
- `thesis`
  - `images` .... obrázky použité v tomto dokumentu
  - `sources` .... pseudokódy použité v tomto dokumentu
  - `BP_Malý_Tomáš_2014.pdf` .... tento dokument ve formátu pdf
  - `BP_Malý_Tomáš_2014.tex` .... zdrojový kód tohoto dokumentu v LaTeX
- `webpage` .... zdrojové kódy webové aplikace pro zobrazování grafů