GYMNASIUM JANA KEPLERA

Parléřova 2/118, 169 00 Praha 6

Pokročilé techniky rasterizace vícevrstvých scén

Maturitní práce

Autor: Čeněk Řehoř

Třída: R8.A

Školní rok: 2021/2022

Předmět: Informatika

Vedoucí práce: Ing. Šimon Schierreich

Praha, 2022

GYMNASIUM JANA KEPLERA
Kabinet informatiky

ZADÁNÍ MATURITNÍ PRÁCE

Student:	**Čeněk Řehoř**
Třída:	**R8.A**
Školní rok:	**2021/2022**
Platnost zadání:	**30. 9. 2022**
Vedoucí práce:	**Šimon Schierreich**
Název práce:	**Pokročilé techniky rasterizace vícevrstvých scén**

Pokyny pro vypracování:

Stěžejním cílem práce je sestavit soběstačnou knihovnu pro hardwarové vykreslování 3D scén zaměřenou na podporu odloženého stínování několika vrstev objektů používajících průhledné či průsvitné materiály. Jelikož je však takový systém pro většinu případů méně efektivní než kombinace odloženého a přímého stínování, je dalším důležitým bodem projektu libovolně kombinovatelná implementace obou metod stínování pro rozmanité konfigurace scén a terénní zkouška kódu pro jednotlivé postupy s analýzou možných způsobů užití v praxi. Součástí práce bude mimo samotný rendering engine základní implementace vykreslovacího stromu scény a interní binární formát serializace grafických prostředků (modelů, textur) se zaměřením na výkon oproti standardním ASCII-based specifikacím. Projekt bude využívat technologii OpenGL a jazyk shaderů GLSL verze 4.0.

Doporučená literatura:

[1] KESSENICH, John a INTEL, eds. *The OpenGL® Shading Language* **[online]. Revize 9. The Khronos Group, 2010, aktualizováno 24. 7. 2010. Dostupné z: https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.00.pdf**

[2] *OpenGL - The Industry Standard for High Performance Graphics* **[online]. Beaverton, OR, USA: The Khronos Group, 1997. Dostupné z: https://www.opengl.org/**

[3] COMNINOS, Peter. *Mathematical and computer programming techniques for computer graphics.* **London: Springer, 2006. ISBN 978-1-85233-902-9.**

URL repozitáře:

https://github.com/Hello007/UberRender

student *vedoucí práce*

V Praze dne 22. 10. 2021

Prohlášení

Prohlašuji, že jsem svou práci vypracoval samostatně a použil jsem pouze prameny a literaturu uvedené v seznamu bibliografických záznamů. Nemám žádné námitky proti zpřístupňování této práce v souladu se zákonem č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších předpisů.

V Praze dne 25. března 2022 Čeněk Řehoř

Poděkování

Děkuji panu profesoru Mgr. MgA. Filipovi Horákovi, že zajišťuje, aby se všechno stalo tak, jak se stát má, když na to zrovna celý vesmír zapomene (pokud tedy zrovna nejde o opravování esejí ze septimy).

Abstrakt

Práce se zabývá problematikou optimalizace výpočetního výkonu fragment shaderů, osvětluje moderní přístupy ke stínování a navrhuje možná řešení nedostatků současných metod. Hlavním konceptem výstupního projektu je vykreslovací systém podporující odložené stínování několika nezávislých vrstev scény, který je reakcí na nedokonalosti způsobené skládáním průhlednosti v G-Bufferu. Práce mimo jiné obsahuje též jak soběstačné schéma navrhované techniky dočasného vícevrstvého G-Bufferu, tak referenční implementaci v jazyce Java používající knihovnu OpenGL. Samostatná sekce je též vyhrazena detailnímu uživatelskému návodu pro využívání otevřeného API softwarového výstupu projektu.

Klíčová slova

počítačová grafika, odložené stínování, vícevrstvé vykreslování, OpenGL

Abstract

The thesis discusses the current issues with optimization of fragment shader compute performance, analyzes modern approaches to hardware shading and suggests possible solutions to imperfections of contemporary methods. The staple concept of the resulting project is a rendering engine capable of deferred shading of several independent scene layers, which attempts to mitigate the defects caused by alpha blending within G-Buffers. Other than that, the thesis also contains a standalone schema of the proposed technique of temporary multi-layer G-Buffers, as well as a reference implementation written in the Java programming language using the OpenGL graphics library. A separate section is also dedicated to a detailed user manual for utilizing the exposed API of the software part of the project.

Keywords

computer graphics, deferred shading, multi-layer rendering, OpenGL

Obsah

1. Teoretická část

1.1 Úvod do stínovacích technik

Počítačová syntéza obrazu v reálném čase je v dnešní době robustní proces, jenž většinu času běží na dedikovaném hardwaru s extrémní mírou paralelizace a optimalizace. Zejména během posledních dvou desetiletí prošla rapidním vývojem díky nástupu programovatelných grafických procesorů a s nimi nových způsobů provádění výpočtů a zpracování obrazu. Programy grafických procesorů, získavší si jméno „shadery" díky svému primárnímu užití - osvětlování a stínování geometrie - jsou sice v současnosti technologicky roztříštěné mezi několika high-level programovacími jazyky a knihovnami, avšak v základě mají sdílená tři stěžejní stádia [Alf].

1. Vertex shader - zpracovává geometrické vstupy z videopaměti. Výstupem jsou primitivní tvary v normalizovaných souřadnicích zařízení.
2. Geometry shader - dále zpracovává primitivní tvary z vertex stádia; provádí např. subdivizi. Toto stádium je ve většině programů volitelné.
3. Fragment/Pixel shader - tento shader narozdíl od obou předchozích operuje na jednotlivých pixelech/fragmentech obrazu. Slouží tak k výpočtu většiny vizuálních charakteristik povrchů, tj. k texturování, osvětlení či bump mappingu. Výstupem pixel shaderu je finální obraz.

I takto rudimentární charakteristika postačí k tomu si uvědomit, že nejvíc práce v drtivé většině grafických programů odvede fragment shader, a to nejen co se počtu operací týče (fragment shader zpracovává každý vykreslený pixel, odtud též jeho alternativní název „pixel shader"), ale i co do složitosti těchto operací. Výpočetní náročnost *jednoho průchodu* fragment shaderu se běžně optimalizuje na straně klienta - statické osvětlení se nechává vypočítat předem pomocí komerčně dostupného 3D softwaru a vzorkuje se z textury a videoherní middleware dělá, co může, aby pomocí chytrých triků snížil počet objektů či zdrojů světla, když se na ně zrovna nikdo nedívá (tj. jsou mimo vykreslovací frustum) [Vri]. Vcelku se rovná o relativně standardní optimalizaci kódu.

Techniky optimalizace *počtu* jednotlivých průchodů fragment shaderu jsou oproti tomu výrazně specifičtější a komplexnější. Přímočarý způsob, jak jednoduše vykreslit méně pixelů, je selektivně snížit rozlišení některých částí scény. Tento přístup je vysoce účinný, ale často má velmi nevábné výsledky, pročež se používá velmi delikátně a s rozvahou. Druhá, složitější cesta využívá schopnosti dnešních grafických procesorů vykreslovat do několika cílů najednou a místo provádění fragmentových výpočtů na každém fragmentu vystoupivším z prvotní rasterizace je spustí až na prototypu finálního obrazu. Této technice se díky její charakteristice odkládání hlavního stádia fragment shaderu dostalo přízviska *deferred shading*, čili odložené stínování [HH04].

Odložené stínování drasticky mění logistickou stránku vykreslování, přestože samotné výpočty jsou takřka k nerozeznání od tradičního kódu pro přímé stínování *(forward shading)*. Především vyžaduje nezanedbatelně vyšší množství dostupné videopaměti a schopnost vykreslovat do relativně vysokého počtu cílů, což může být problém např. pro mobilní grafické procesory (specifikace OpenGL ES 3.0 vyžaduje povinně jen čtyři [LL19]). Mimo technické zábrany má odložené stínování i několik dalších implikovaných nedokonalostí - ale o tom již více v následující kapitole.

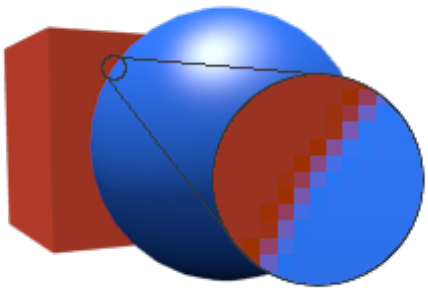
1.2 Stinné stránky odloženého stínování

Abychom lépe porozuměli úskalím odloženého stínování, je předem nutné se alespoň běžně seznámit s principem, na kterém funguje. Celé kouzlo „odkládání“ spočívá v tom, že místo vypočítaného finálního obrazu přesměrujeme do výstupu fragment shaderu pouze okrajově zpracovaná geometrická data - zpravidla je uložíme do textury či textur nazývaných G-Buffer (zásobník geometrie) [HH04]. Tyto jsou v druhém průchodu vykreslování použity jako vstupy komplikovanějšího fragment shaderu, který se teprve stará o výpočty osvětlení apod. Konkrétní obsah a objem těchto dat je závislý na osvětlovacím modelu a vůli programátora, není totiž pokaždé pravidlem, že čím více toho lze odložit, tím lépe. Každý vykreslovací cíl navíc totiž stojí spoustu videopaměti, obzvláště při vysokých rozlišeních. Zároveň je radno uvědomit si, že konečný fragment shader bude jeden a jediný sdílený pro celý vstupní obraz, pročež je nutné zpracovat co nejvíce parametrů materiálu ještě před odesláním do G-Bufferu, neboť předávání takovýchto atribut skrz vykreslovací cíle si vyžádá značné množství hardwarových prostředků - příkladem budiž sestavování T/B/N matice pro bump mapping, jehož odložení by vyžadovalo předání vektorů tečen povrchu i hodnot bump map textury skrz G-Buffer.

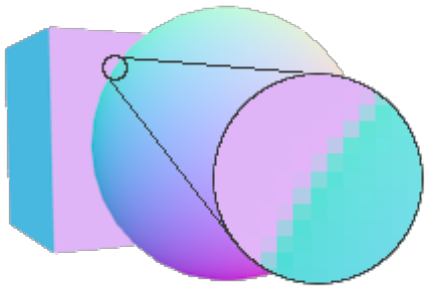
Zde by leckdo mohl oprávněně vznést námitku - čemu konkrétně vlastně ono na první pohled zbytečně komplikované odkládání výpočtů prospěje? Existuje nemálo případů, ve kterých by reakcí na tuto námitku byla prostá odpověď: „Ničemu“. Nicméně se zvyšující se složitostí scén má odkládání výpočtů jednu nespornou výhodu - zajišťuje, že každý pracně spočítaný fragment je právě ten, který se objeví ve finálním obraze, tj. nepřijde vniveč v důsledku zakrytí jiným fragmentem. Překrývající se fragmenty tak spotřebují pouze výpočetní cykly zápisu do G-Bufferu a osvětlovací rovnice už poběží jen na jejich konečné vrstvě. Bohužel, s touto optimalizací přichází i nemalý počet nedostatků.

1.2.1 Nedokonalé vyhlazování hran

Jedním z nejprominentnějších je nemožnost přímého použití hardwarového vícevzorkového antialiasingu (MSAA) [NVI]. Tato metoda vyhlazování hran totiž oproti klasickému nadvzorkování zvyšuje pouze rozlišení zásobníku hloubky, a výstupní obraz je tak vyprodukován pomocí lineární interpolace mezi vzorky, kterážto má pro prokládání souřadnic dvou ničím nespřízněných objektů v G-Bufferu katastrofální následky.



Obrázek 1.1: Nadvzorkování barvy



Obrázek 1.2: Nadvzorkování normál

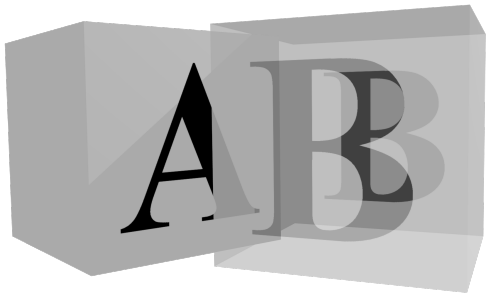
Princip vyhlazování bude podobný jako u standardní vykreslovací procedury - barevné kompo-

nenty framebufferu budou ve finálním obraze interpolovány na základě nadvzorkovaného zásobníku hloubky (obr. 1.1). To samé bude přirozeně platit i pro veškeré součásti G-Bufferu - jako typický příklad nežádoucích účinků MSAA v odloženém stínování si tedy představme třeba interpolaci normálových vektorů. Pokud se pokusíme vypočítat úhel mezi vektorem světla a normálou povrchu, interpolovaná hodnota na okraji (obr. 1.2) v lepším případě způsobí výpočet závislý na průměru normál přilehlých objektů, v tom horším a pravděpodobnějším bude mít za výsledek, způsobený nejednotkovou délkou vstupního vektoru, naprosto nesmyslnou hodnotu.

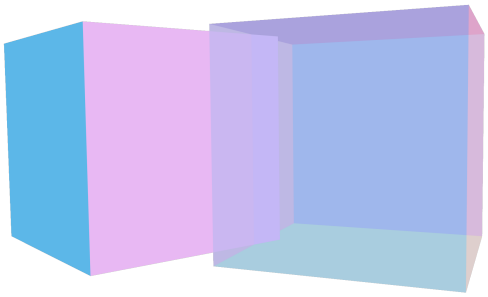
Tento problém je takřka neřešitelný, neboť vícevzorková interpolace je z podstaty neslučitelná s principem G-Bufferů. Z tohoto důvodu se v odloženě stínovaném obraze používají buď screen-space metody vyhlazovaní hran, nezávislé na geometrických charakteristikách dat (např. FXAA nebo SMAA), nebo tradiční supersampling.

1.2.2 Problematické míchání barev

V drtivé většině scén se dříve nebo později objeví nutnost vykreslit nějaký objekt průhledně či průsvitně. Tradičně se pro tuto potřebu využívá zafixované funkce grafických procesorů nazývané skládání či míchání, obvykle na základě alfa kanálu obrazového vstupu a framebufferu. Tento proces je notorickým zdrojem potíží s řazením tvarů na scéně, neboť pro správné vykreslení je třeba zajistit vzestupné hloubkové pořadí všech průhledných a průsvitných objektů - mezi dva průsvitné objekty smíchané ve dvojrozměrném framebufferu totiž není možné vložit další trojrozměrný prvek. Některé vykreslovací postupy používají techniky průhlednosti bez závislosti na pořadí jako např. *depth peeling* [Eve], ty však bohužel vyžadují velké množství průchodů a nejsou tak ideální pro vykreslování v reálném čase. Problémy způsobené přímým skládáním výstupu a framebufferu mají v odloženém stínování podobné důsledky jako multisampling.



Obrázek 1.3: Průhlednost barvy



Obrázek 1.4: Průhlednost normál

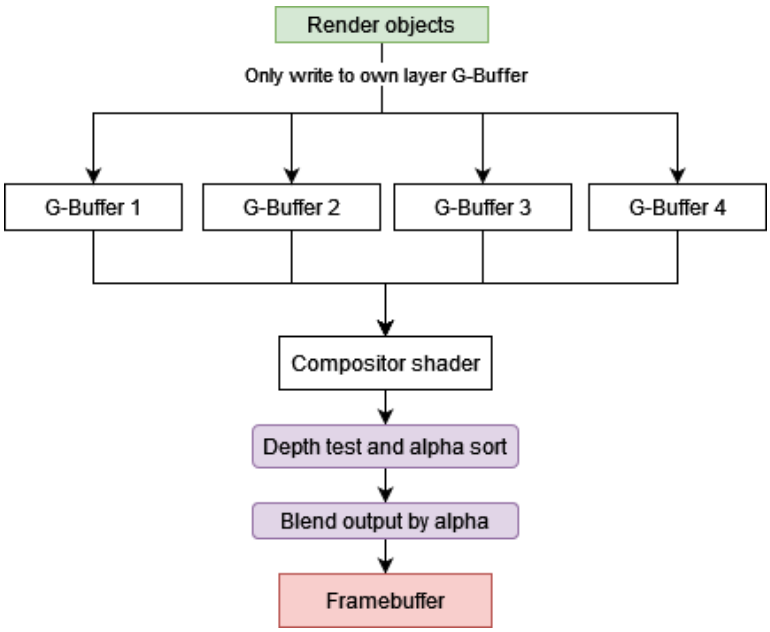
Zjednodušeně bychom mohli proces skládání charakterizovat jako smíchání barvy příchozího fragmentu s aktuálním vykresleným obrazem (obr. 1.3). V jazyce matematiky můžeme říci, že provede lineární interpolaci mezi vstupem a framebufferem, používajíc navzorkovanou hodnotu alfa kanálu vstupu jako interpolační váhu. Tento postup bude za předpokladu správného seřazení fungovat dle očekávání i při odloženém stínování a výstupem budou obdobně interpolované hodnoty v G-Bufferu. V tuto chvíli by nám již mělo být jasné, že jsme narazili na problém takřka shodný s tím, který nám nedovolil používat nadvzorkování zásobníku hloubky. Interpolace albedo složky G-Bufferu sice bude mít žádoucí výsledky, ale geometrické komponenty jako pozice či normála budou nenávratně poškozené obdobným způsobem jako při průměrování sousedních vzorků (obr. 1.4).

Nepříjemnosti způsobené prokládáním G-Bufferu průhledných objektů se běžně řeší oddělením průhledné a matné vrstvy a hybridním vykreslováním kombinujícím jak odložené, tak přímé stínování [Mag12]. Vzhledem k obvykle nevelkému množství průsvitných objektů ve videohrách, vizualizacích a jiných aplikacích real-time počítačové grafiky tato metoda nemá drastický dopad na výpočetní požadavky programu a pro značnou část využití nemá smysl hledat alternativní řešení. S narůstajícími nároky na výkon grafických programů a složitostí scén je však otázka zdokonalování výpočtů stále více než relevantní, pročež jsem na základě několika hypotéz rozhodl přijít s vlastním řešením průhlednosti pro odložené stínování, které by umožňovalo vykreslování téměř nekonečného množství vrstev pomocí libovolné kombinace stínovacích technik či plně odloženě. Jak toho dosáhnout a zda se to vyplatí, to vše se pokusím osvětlit v následujících kapitolách.

1.3 Alternativní řešení průhlednosti

1.3.1 Vykreslování do paralelních G-Bufferů

Má původní myšlenka konceptu systému pro odložené vykreslování několika vrstev byla poněkud netradiční - v jádru totiž stála na odpoutání se od zafixovaných funkcí rasterizéru a provádění operací jako skládání a hloubkové testování v shaderu. Tento kompromis by nám tak dovolil vypnout všechny rané operace s fragmenty a provádět je manuálně později, až na hotových G-Bufferech několika vrstev scény. Celý proces by tak fungoval následujícím způsobem:



Obrázek 1.5: Prototyp průběhu vykreslování

Během prvotního vykreslování je většina standardních per-sample operací na fragmentech, včetně testování hloubky, zakázána. Místo toho se každý fragment zapíše do jednoho z výstupních G-Bufferů podle předem nakonfigurovaného parametru vrstvy a jednotlivé výstupy jsou zpracovány až v kompozitoru, který všechny sám seřadí a provede interpolaci a zahození neviditelných fragmentů vlastní cestou (obr. 1.5).

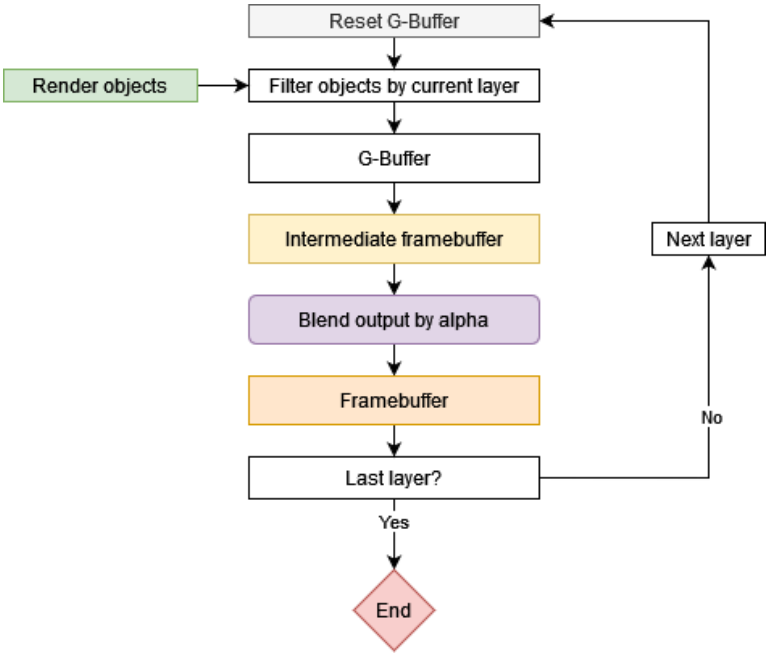
Tento přístup z teoretického hlediska nemá žádné skryté háčky, zato jich má spoustu naprosto oči-

vidných, kterých jsem si všiml již na začátku implementace a kvůli nimž jsem byl nucen tento návrh zásadně přepracovat. Již na první pohled je například až bolestivě neekonomický - vykreslování do několika G-Bufferů by při sebevyšších rozlišeních vesele spořádalo několik set megabajtů videopaměti. Klíčovým nedostatkem však je, že testování hloubky nelze zakázat úplně, pokud se byť jediné dva fragmenty ve scéně překrývají - k řešení takových situací totiž právě tato funkce existuje. Koncept několika souběžných G-Bufferů jsem tak shledal v podstatě nepoužitelným a bylo jasné, že na to musím trochu jinak.

1.3.2 Dočasný vícevrstvý G-Buffer

Došli jsme tedy k závěru, že více paralelních G-Bufferů je takřka nemožné využít, a řešení tudíž musíme hledat pomocí běžného sériového zapojení. To nám mimo jiné dovolí používat i hardwarové hloubkové testování. Přímočarou implementací by v tomto případě bylo pro každou vrstvu vytvořit oddělený G-Buffer, který by se střídal dle aktuálního vykreslovaného objektu. V rámci jednotlivých vrstev by probíhalo hardwarové hloubkové testování a kompozitor by se tak staral jen o srovnávání hloubky mezi vrstvami. Takové řešení by pravděpodobně fungovalo, ale stále u něj přetrvává první kámen úrazu minulého návrhu - náročnost na videopaměť.

Dalším krokem je tedy přijít s co nejefektivnějším způsobem, jak snížit množství použitých G-Bufferů. Náš aktuální model by bez námahy šlo omezit na dva G-Buffery střídající se mezi sebou s každou novou vrstvou, avšak vzhledem k tomu, že jen tento krok by náš systém ochudil o nezávislost na pořadí průhledných objektů, můžeme si dovolit jít ještě o něco dále.



Obrázek 1.6: Návrh vícevrstvého vykreslování s jedním G-Bufferem

Vzhledem k tomu, že pořadí objektů lze předem vypočítat nebo alespoň manuálně nadefinovat (což ve většině praktických případů postačí), můžeme si scénu rozdělit na několik vrstev, které se po průchodu G-Bufferem budou moci chovat jako klasický přímo stínovaný objekt a s nimiž pak budeme snadno moci použít hardwarovou průhlednost (obr. 1.6). Sice tak mírně snížíme účinnost odloženého stínování, neboť budeme vykreslovat více než jen ty nejsvrchnější fragmenty, avšak

tomu se s průhlednými objekty logicky beztak nedá vyhnout. Co nám nyní zbývá vyřešit, je především hloubkové testování a kombinace s přímo stínovanými objekty (např. 2D uživatelské rozhraní nebo částice vyžadující speciální shadery). Tyto otázky by pravděpodobně bylo možné řešit čistě teoreticky, avšak v této fázi už jsem měl dostatek sebejistoty na to, abych přesunul své experimenty do terénu, a tak jsem začal z programováním referenční implementace vícevrstvého odloženého stínování, knihovny a vykreslovacího enginu UberRender.

1.4 Cíl a požadavky

Primárním stanoveným cílem práce je vícevrstvé vykreslování odloženě stínovaných scén, avšak takový systém se neobejde bez spousty podpůrného kódu. Vzhledem ke specifičnosti projektu jsem se rozhodl používat externí knihovny pouze na úrovni backendu a složitější funkce knihovny implementovat vlastním způsobem - bylo tak potřeba nejprve naprogramovat jakousi kostru, abych vůbec mohl začít něco pohodlně vykreslovat.

Prvním požadavkem této úrovně je abstrakce interních grafických API (např. OpenGL) na high-level třídy pro správu grafických prostředků. Sledoval jsem tím záměr odloučení se od přímých volání do grafické knihovny z perspektivy implementora a zároveň lepší využití funkcí objektově orientovaných programovacích jazyků.

Následně je potřeba zajistit bezproblémový převod a načítání prostředků ve standardních formátech podporovaných 3D softwarem. Usoudil jsem, že pro demonstrační účely pravděpodobně postačí formát Wavefront OBJ, který, přestože není ani zdaleka nejnovější, je stále podporovaný většinou moderních 3D programů. Počítaje s nadcházející implementací tohoto formátu mi však také bylo jasné, že načítání větších souborů v tomto formátu založeném na serializaci v prostém textu bude časově velmi nevýhodné a s nárůstem implementačně specifických parametrů materiálů a geometrie bude třeba využít rychlejšího binárního formátu, pročež jsem zatím nepojmenovaný binární formát přidal na seznam požadavků.

Samotný rozšiřitelný formát souborů nám však je nemnoho užitečný, pokud nemáme, jak upravovat jeho rozšířené parametry, a vzhledem k binárnímu schématu úprava pomocí standardních editorů nepřipadá v úvahu. Proto by součástí projektu měl být i grafický editor pokrývající všechna specifika formátu, umožňující intuitivní převod ze standardních schémat.

Jelikož je hlavním výstupem projektu demonstrační implementace, je také důležité neopomenout klíčovou součást každého moderního vykreslovacího enginu - scenegraph. V ideálním případě by měl obsahovat podporu pro skinning a dynamické transformace členů scény, avšak za předpokladu dostatečné rozšiřitelnosti postačí jen základní kompozice a relace mezi členy.

Později, během implementace, vyplavalo na povrch nemalé množství dalších požadavků a potřeb, ale o těch již bude pojednávat příští kapitola.

2. Implementace

2.1 Koncepce a výběr technologií

UberRender, zkráceně URender, je zamýšlen jako knihovna pro experimenty, výzkum a vývoj, pročež jsou jedněmi z jeho primárních cílů srozumitelnost, dostupnost a kompatibilita. Z toho důvodu jsem během implementace ne vždy použil zažité průmyslové standardy pro grafické programování, neboť orientace na výkon byla oproti běžné real-time grafice soustředěna výlučně na stránku programu běžící na GPU bez explicitní snahy o dechberoucí optimalizaci na straně procesoru. Vyhradil jsem proto nejprve zvláštní sekci této kapitoly vedle standardních, univerzálních technologií i těm cíleným exkluzivně na grafický hardware.

2.1.1 GPU technologie

Technologie, jichž jsem využil pro programování na straně grafické karty, byly vybrané s přiměřeným ohledem jak na srozumitelnost a přístupnost, tak na funkcionalitu a výkon. Jedná se výhradně o otevřené standardy a API, které, přestože podporují implementačně specifická rozšíření, jsem se zavázal využít pouze v rámci jejich standardních součástí. Projekt by tudíž měl být velmi snadno přenositelný mezi hardwarem rozličných výrobců, stáří i úrovně podpory.

OpenGL

Grafická knihovna OpenGL skupiny Khronos je sice v dnešní době pomalu ale jistě nahrazována svým nástupcem, knihovnou a API Vulkan, ale pro mnoho využití je stále relevantní. Primární důvod pro volbu OpenGL před jinými řešeními byla především dostupnost veřejné dokumentace a podpory. Ostatní moderní grafické knihovny jsou totiž orientované na průmyslová využití a vyžadují programování na komplexnější úrovni velmi blízké hardwaru, což nejenže zvyšuje vstupní bariéru, ale ani není pro mé záměry žádoucí. OpenGL je oproti tomu ideální pro výzkumné implementace, jelikož má dodnes rozsáhlou komunitu a velmi přístupné rozhraní.

Dalším důvodem pro výběr OpenGL je bezkonkurenční podpora cross-platform vývoje. Oproti sadě DirectX, exkluzivní platformám společnosti Microsoft, a API Vulkan, které přes snahy skupiny Khronos vinou antagonizmu společnosti Apple dosud nemá nativní podporu na nezanedbatelné části trhu, jsou ovladače podporující OpenGL přítomné na drtivé většině zařízení disponujících grafickým procesorem. OpenGL tak zajišťuje téměř bezproblémovou portabilitu mezi operačními prostředími, která dodržují její specifikace (nutno dodat, že to např. společnosti AMD dělá nemalé problémy), poskytujíc mi jednodušší přístup k větší škále měření a diagnostiky.

GLSL

Shading jazyk GLSL, úzce provázaný s knihovnou OpenGL, je z části i důvodem pro volbu zmíněné knihovny. Přestože je v jádru kompatibilní i s API Vulkan, bylo by pro jeho užití nutné zahrnout translační vrstvu do bytekódu SPIR-V (např. knihovnu GLSlang) [Ove].

GLSL je syntaxí podobný jazyku C, pročež je oproti např. GLASM výrazně uživatelsky přívětivější a vhodnější pro demonstrace a referenční implementace. Jeho nevýhodou je delší doba kompilace programů způsobená složitostí syntaxe, což je ale naštěstí ve většině případů včetně mého jen jednorázový problém při zavedení programu.

2.1.2 Standardní technologie

Knihovny a technologie běžící ve standardním operačním prostředí jsem, podobně jako technologie pro GPU, vybíral především pro jejich uživatelskou přístupnost. Tentokrát jsem však snížil požadavky co se optimalizace a výkonu týče a upřednostňoval spíše takové, s nimiž jsem měl v minulosti známost a dobré zkušenosti. Objeví se zde tak několik technologií, které jsou ve sféře počítačové grafiky spíše nevšední, přestože mimo tuto unikátní odnož informatiky znamenají průmyslový standard.

Java

V real-time grafickém programování se téměř bez výjimek používají nativní kompilované jazyky C a C++ pro jejich hloubkový přístup k hardwaru a možnosti optimalizace, pročež je můj výběr jazyka Java možná překvapením. Primárním důvodem výběru pro mne byla familiarita s jazykem a standardní knihovnou, avšak z podobných důvodů bych bez váhání byl schopen zvolit i jazyk C. Skutečnou rozhodující roli nakonec sehrála jednoduchost správy paměti a manipulace datových struktur, která, přestože imponuje nemalou daň na výkon programu, je v prostředí jazyka Java a runtime JVM excelentní a dovoluje tak soustředit se na funkční stránku programu bez nezpočtu řádků low-level systémového kódu. Narozdíl od modernějších jazyků typu Python nebo Rust má Java čitelnější syntaxi a typování, které jsou pro demonstrativní stránku projektu URender klíčové.

Z podobných důvodů jsem ze začátku zvažoval i použití jazyka C#, avšak ve finále jsem dal přednost Javě pro lepší dostupnost multiplatformního programování uživatelského rozhraní, které je ve světě C# dodnes problematické.

Swing

Ve věci grafického uživatelského rozhraní jsem dal přednost knihovně Swing před knihovnou JavaFX zejména pro bezvadnou integraci s vývojovým prostředím NetBeans, jenž drasticky zpříjemňuje design UI prvků díky automatickému generování kódu pro rozvržení a události. Z hlediska čistoty a úhlednosti kódu je sice jakákoli sebevětší aplikace ve Swingu obvykle nejneohrabanější komponentou programů, avšak vzhledem k nastávající roli GUI programování v projektu URender, kterážto nevyžadovala ani rozšiřitelnost uživateli, ani perfektní srozumitelnost zdroje, jsem byl ochoten tuto oběť podstoupit.

JogAmp

Pro komunikaci mezi prostředím JVM a ovladači pro OpenGL v současnosti existují dvě hlavní udržované knihovny - Lightweight Java Game Library (LWJGL), používaná např. v populární vi-

deohře Minecraft a méně robustní knihovna JOGL ze sady JogAmp. Mé rozhodnutí ve prospěch JOGL bylo založené na oproti LWJGL bezproblémové integraci s GUI prvky knihovny Swing a většímu zaměření na obecnou perspektivu grafiky v kontrastu s čistě videoherním cílovým publikem LWJGL.

JOML

Otevřená knihovna pro lineární algebru JOML je volbou takřka bez rozmyslu, neboť se jedná o kompletní sadu tříd pro manipulaci vektorů a matic v prostředí jazyka Java s úctyhodnou optimalizací a vynalézavou vlastní správou paměti téměř bez obdoby. Z předchozích zkušeností mi oproti např. integrované knihovně JogAmpu lépe posloužila při počítání s rotacemi a kvaterniony a narozdíl od klasických knihoven pro lineární algebru je navržená s důrazem na kompatibilitu s grafickými výpočty, zejména s rozhraním OpenGL.

2.2 Průběh a detaily implementace

2.2.1 Struktura knihovny

Přestože jsem se pro rámec této práce rozhodl omezit backend vykreslovacího kódu na knihovnu OpenGL, chtěl jsem zároveň oddělit funkce jejího API od uživateli přístupného rozhraní. Primárním důvodem byla disharmonie programovacích konceptů OpenGL (state machine) a Javy (OOP) a nekompatibilita s potenciální adaptací dalších API v budoucnosti.

URender tak má na nejnižší uživateli otevřené úrovni balíček `api`, který poskytuje abstrakci enumerací a funkcí sdílených pro většinu moderních grafických knihoven (formáty textur, typy interpolace/filtrování, datové typy, testovací rovnice). Adaptér grafické knihovny pak implementuje rozhraní `RenderingBackend` a pomocnou třídu `APITranslator`, které se společně starají o překlad konstant a chování metod pro specifické potřeby knihovny. Důležité je podotknout, že jakákoli high-level abstrakce je z této vrstvy vyloučena - všechny prostředky jako zásobníky, textury nebo shadery jsou zde reprezentovány generickou třídou `UObjHandle`, která se v zákulisí stará o podporu souběžné existence objektu v několika kontextech (např. dvou různých oknech) a udržuje samostatnou referenci pro každý použitý backend.

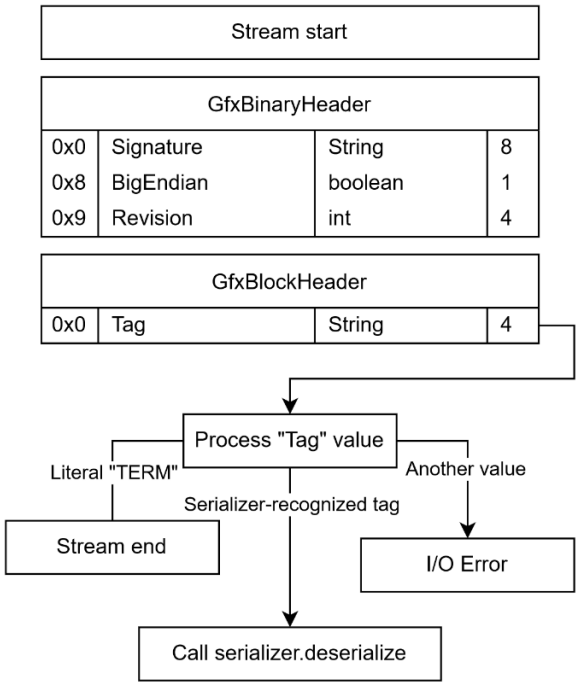
Teprve o úroveň výše, v balíčku `engine`, se tyto abstraktní reference obstarávají v uživatelsky příjemných třídách jako `UTexture` nebo `UShader`, které programátorovi poskytují flexibilní objektově orientované API a předem připravené funkce pro jejich použití ve scéně na ideálně jeden až dva řádky kódu. Každá z těchto tříd disponuje též vlastní Builder třídou pro jednoduchou konstrukci instancí. Tato vrstva je však stále pouze další úrovní abstrakce nad jádrem grafické knihovny - samotná kompozice scény a kontrola průběhu vykreslování probíhá ještě o úroveň výše, v balíčku `scenegraph`.

Úmysl za oddělením těchto dvou vrstev vzešel ze skutečnosti, že za jistou hranicí abstrakce se již mnoha programátorům vyplatí sestavit vlastní implementaci namísto využívání externí knihovny. Balíček `engine` a potažmo i balíček `api` jsou tak plně funkční bez balíčků o úroveň nad nimi a je tedy jen na vůli programátora, zda se rozhodne high-level funkcí zprostředkovaných knihovnou využít, či ne. Balíček `scenegraph` obsahuje struktury pro standardní osvětlovací modely, relace mezi

objekty ve scéně a třídy pro výpočet vykreslovací fronty a transformace geometrie včetně několika druhů kamery. Zároveň obsahuje základní třídu `UGfxRenderer`, která nahradila třídu `RenderingBackend` z nižších úrovní knihovny, a která se stará o správu vykreslování scén včetně manipulace s framebuffery a G-Buffery pro různé kombinace metod stínování. Tato třída vyžaduje implementaci několika virtuálních metod programátorem, neboť nebylo v mém záměru vynucovat na této úrovni specifické rozložení G-Bufferu nebo osvětlovací model.

Poslední součástí knihovny je pomocný balíček `g3dio`, který zjednodušuje převádění rozličných grafických souborů do tříd v rámci běžícího programu. Hlavní komponentou této sady je serializér a deserializér formátu uGFX, který je specificky navržen pro plnou kompatibilitu se všemi atributami tříd prostředků knihovny a umožňuje velkou míru rozšiřování a úprav ze strany programátora. O základní struktuře a návrhu tohoto formátu pojednává následující kapitola.

2.2.2 Formát kontejneru grafických prostředků



Obrázek 2.1: Visualizace kontejneru uGFX

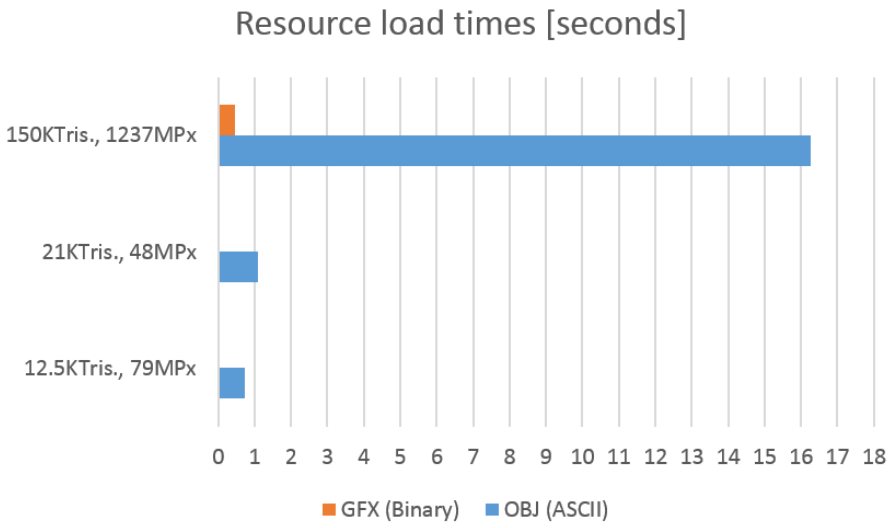
Přestože je uGFX binární formát, oproti běžným implementacím není brán ohled na zarovnání adres hodnot pro přímou dereferenci. Jazyk Java totiž nepodporuje zero-copy načítání dat ze souboru a dereferenci nezpracovaných dat po více než jednom bajtu, nedává tudíž smysl zabývat se výhodami zarovnávání. Formát je tak navržen pro čtení v jednolitém nezarovnaném proudu s minimální vnější strukturou (obr. 2.1) bez nutnosti vracení se či přeskakování vpřed, což potenciálně umožňuje streamování prostředků.

Jednotlivé třídy prostředků disponují vlastními implementacemi serializace a deserializace a poskytují třídě pro zapisování a čtení kontejnerů pouze identifikační štítek a referenci na sebe sama při registraci. Při čtení prostředků z kontejneru se nejdříve přečte zafixovaná hlavička formátu, která kromě kontrolního podpisu (8bajtového řetězce `UGfxRsrc`) obsahuje ještě revizi formátu pro zpětnou kompatibilitu a BOM pro možnost zapisování v big-endian pořadí bajtů, umožňující srozumitelnější

ladění binárního výstupu z perspektivy vývojáře.

Po kontrole hlavičky následuje libovolný počet bloků, začínajících buď rezervovaným identifikátorem `TERM`, označujícím konec proudu, nebo jiným čtyřznakovým identifikačním štítkem unikátním pro třídu deserializéru dat. Pokud se podaří přečtenému štítku přiřadit deserializér, kontrola nad datovým proudem se předá jeho vlastní implementaci, odkud se vrátí na konci bloku. Odtud se cyklus opakuje, dokud rezervovaný identifikátor neukončí proud. Zápis formátu probíhá obdobným způsobem, v podstatě jen s opačným směrem proudu.

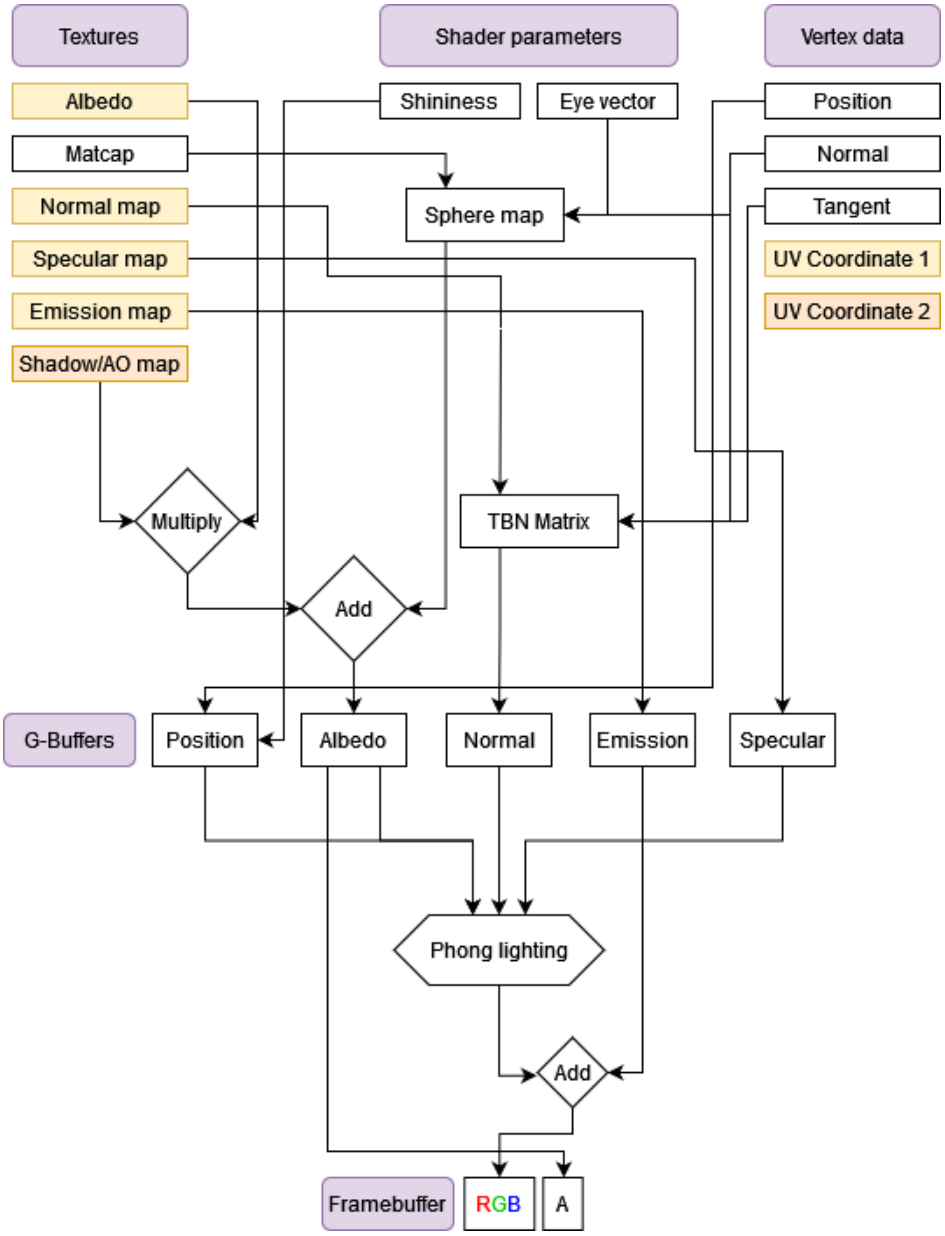
Původním účelem binárního formátu byla především kompletní serializace tříd a rychlejší načítání prostředků, pročež byly součástí finálních stádií implementace zátěžové testy a optimalizace.



Obrázek 2.2: Zátěžový test doby načítání prostředků (nižší čas je lepší)

Testovací sekvence zahrnující tři 3D scény o 21 až 150 tisících trojúhelnících a až přes miliardu pixelů textur dle očekávání vyústila v drastické zlepšení oproti textovému formátu Wavefront OBJ (dokonce se vlivem poměru občas ani nezobrazí na grafu). Načítání z binárních souborů bylo obvykle cca. 30x rychlejší (viz. graf 2.2), často i více při menším objemu textur. Tato část projektu tak dalece předčila má očekávání a spolehlivě splnila svůj účel.

2.2.3 Shader model



Obrázek 2.3: Shader model URender/Turbo

Původně jsem zamýšlel postavit demonstrativní osvětlovací model na principu PBR, avšak při prvních pokusech o implementaci jsem postupně zjistil, že požadavky na kompletní adaptaci tohoto konceptu daleko přesahují rámec projektu a že raději dám přednost modelu, který je lépe srozumitelný. Rozhodl jsem se tak pro starší model osvětlení založený na Phongově stínování s podporou pro předpočítané stíny a sphere-map materiály, který jsem nazval podle kódového označení herního enginu, jehož grafiku jsem se co nejlépe snažil replikovat - Turbo.

Shader model Turbo dovoluje specifikovat všechny klíčové parametry Phongovy rovnice pomocí textury (obr. 2.3) a zároveň regulovat jejich intenzitu pomocí parametrů materiálu či některé z nich úplně zakázat. Ještě před předáním do G-Bufferu se smíchá barva předpočítaného osvětlení a AO, albedo textury a sphere-map textury obvykle užívané pro imitaci kovů. Za pozornost stojí zpraco-

vání parametru lesklosti povrchu (exponenta spekularity v Phongově rovnici). Ten se zapisuje do nevyužité čtvrté komponenty textury pozice fragmentu, neboť vyžaduje škálu hodnot přesahující klasických 255, a tudíž jej není možné zapsat do G-Bufferu spekularity, kam by měl normálně patřit.

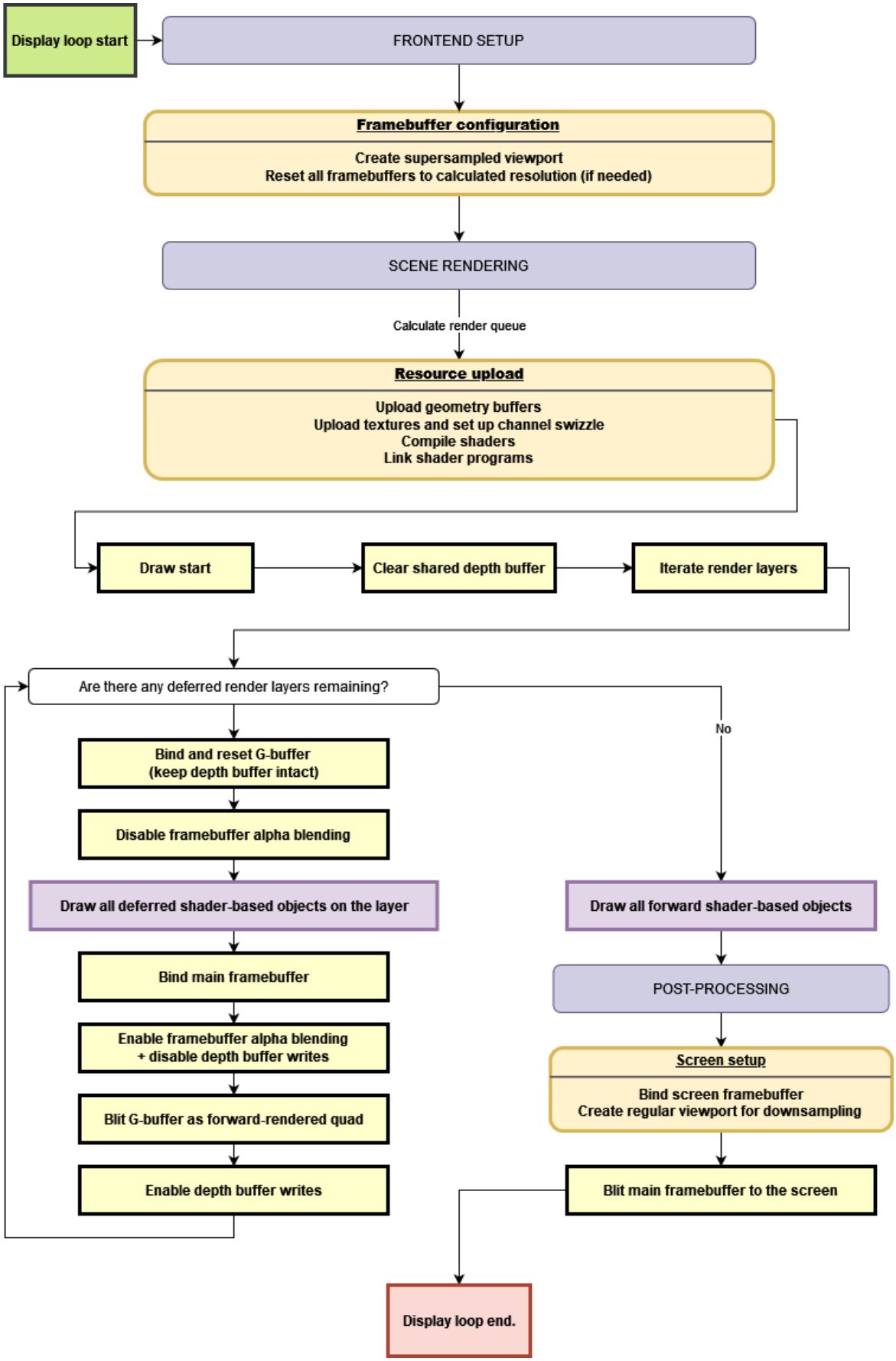
Alfa kanál albedo textury se i při odloženém stínování přechovává od začátku vykreslování skrz G-Buffer až do zkomponovaného obrazu, neboť je klíčový pro určení průhlednosti fragmentu. Po kombinaci alfa kanálu s vypočítanou osvětlenou barvou už výstup putuje do stádia přímého vykreslování.

2.2.4 Vícevrstvé odložené vykreslování

Stěžejního cíle projektu, vícevrstvého vykreslování průhledných objektů, jsem nakonec dosáhl pomocí velmi netradiční manipulace se zásobníkem hloubky. Původně jsem totiž zamýšlel provádět řazení fragmentů až v druhém stádiu fragment shaderu (kompozitoru), to ale mělo nepřijatelný dopad na výkon. Proto jsem využil schopnosti grafických procesorů maskovat zápisy do zásobníku hloubky a dokázal jsem tak použít jeden sdílený zásobník pro jak odloženě, tak přímo stínované objekty.

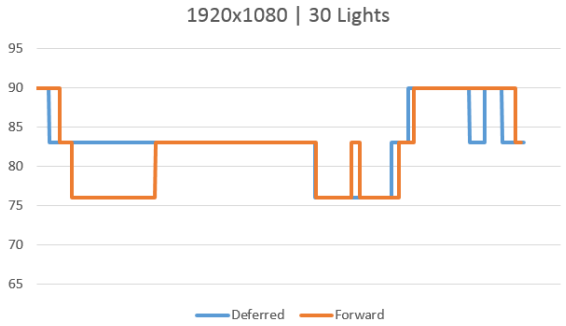
S každou vykreslenou odloženě stínovanou vrstvou se celý G-Buffer překreslí výchozími hodnotami, ale v zásobníku hloubky zůstanou data z minulého průchodu. Následně se zakáže hardwarové skládání průhlednosti, aby nedošlo k nepříjemnostem s interpolovanou geometrií, ale hodnota alfa kanálu se stále ukládá do albedo komponenty G-Bufferu. Během vykreslování do G-Bufferu se hloubka zapisuje běžně do sdíleného zásobníku. Jakmile tato fáze skončí, zakáže se zápis do zásobníku hloubky a G-Buffer se nastaví jako vstup kompozitoru, tentokrát se zapnutým skládáním průhlednosti. V tomto stavu se G-Buffer vykreslí jako obdélník vyplňující celou obrazovku do hlavního framebufferu a průhlednost se spočítá jako při přímém vykreslování interpolací s jeho aktuální hodnotou, pozůstalou z předchozích vykreslených G-Bufferů (obr. 2.4).

Tento proces se opakuje pro všechny odloženě stínované vrstvy, až se nakonec tentokrát již definitivně povolí zápisy do zásobníku hloubky a vykreslí se přímo stínované objekty. Pro grafický procesor je před tímto stádiem zásobník hloubky shodný s takovým, jaký by mělo za výsledek vykreslení předchozích odloženě stínovaných objektů přímo, právě protože jsme během kompozice G-Bufferu vypnuli zápisy do zásobníku hloubky - souřadnice vyplňovací roviny jej tak neovlivnila. Následný proces přímého vykreslování je tudíž shodný s tradičními postupy a plně kompatibilní s předchozími vykreslenými objekty.

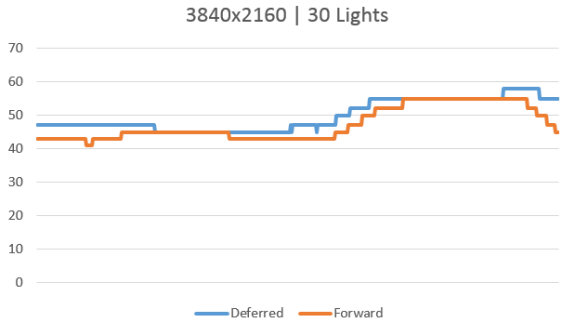


Obrázek 2.4: Finální průběh vykreslování

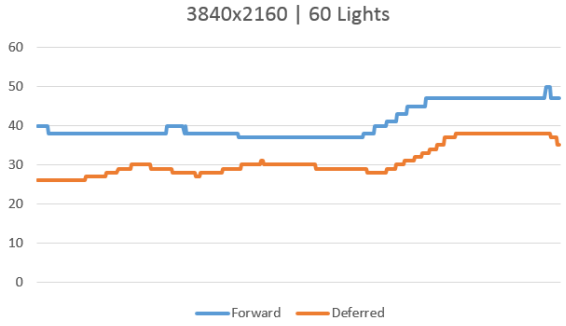
Zátěžové testy



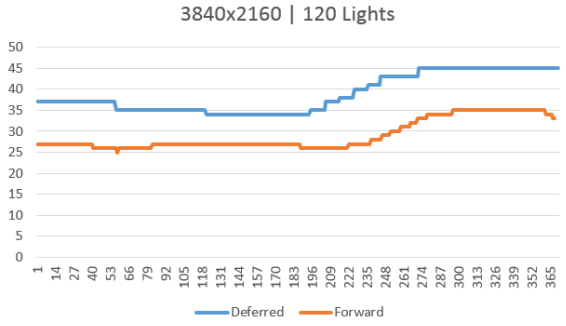
Obrázek 2.5: Zátěžový test 30 světel @ FHD



Obrázek 2.6: Zátěžový test 30 světel @ 4K



Obrázek 2.7: Zátežový test 60 světel @ 4K



Obrázek 2.8: Zátěžový test 120 světel @ 4K

Zátěžové testy vícevrstvého odloženého stínování jsem prováděl na dvou hardwarových konfiguracích, primárně na grafické kartě NVIDIA GeForce GTX 1060 a též na integrovaném grafickém čipu Intel HD Graphics 4000. Pro naprosto otřesné výsledky čipu Intel zde uvádím pouze grafy testů provedených na grafické kartě NVIDIA.

Testy ukázaly, že při nižších rozlišeních a nevelkých počtech zdrojů světla jsou výsledky odloženého stínování téměř do puntíku identické s výkonem klasického přímého stínování (grafy 2.5 a 2.6). Domnívám se, že izolovaný výpočetní výkon shaderu byl při odloženém stínování přeci jen rychlejší, avšak byl zastíněn náročností paměťových operací, zejména pak zápisu a následného čtení G-Bufferu, kterážto fáze je z přímého stínování vyloučena a poskytuje tak nepatrnou výhodu. Rozdíly se začnou ukazovat až při zvýšení počtu světel, kde odložené stínování předčí přímé o přibližně 10 vykreslených snímků za sekundu navíc (grafy 2.7 a 2.8). Jak vyšší počet světel, tak rozlišení poskytují odloženému stínování inkrementální náskok díky pouze jednovrstvému nárůstu v počtu fragmentů. Přímé stínování, kde se výpočty provedou i pro přibyvší fragmenty zakrytých vrstev, tak postupně více a více zaostává.

Další nesporná výhoda použitého odloženého přístupu, která se však v zátěžových testech nemůže projevit, je výrazné uvolnění prostoru pro uniformy shaderu díky oddělení stádia hlavního výpočetního fragment shaderu od prvotního, použitého pouze na vytvoření G-Bufferu. Absence uniforem užitých pro konfiguraci materiálu a jeho textur v kompozitoru umožňuje použití většího počtu uniforem pro světla či speciální samplery. Na první pohled se sice může zdát, že nám jich nepřibude mnoho, avšak zdání klame - specifikace OpenGL 4.0 vyžaduje pouze 1024 povinných

komponent uniforem (status `GL_MAX_FRAGMENT_UNIFORM_COMPONENTS`) [SA10], které struktury světel dokáží slupnout jako maliny. Jakékoli místo navíc je tak překvapivě cenné.

Oproti hybridnímu vykreslování, používajícímu odložené stínování pro neprůhledné objekty a přímé stínování pro průhledné, jsme též ušetřili nezanedbatelné množství systémových volání nutných pro přepínání mezi shader programy a konfiguraci materiálů pro pozměněný shader model.

Zároveň by ale bylo pokrytecké tvrdit, že jsme tímto přístupem naprosto vyloučili přímé stínování ze hry. Výsledky sice potvrdily příznivé účinky odloženého stínování, ale zároveň poukázaly na kompetentnost přímého stínování při nižším rozlišení a jednom průchodu vykreslování. Tato součást projektu tedy splnila své hlavní cíle, ale oproti jiným komponentám přes svou stěžejní roli v projektu má, byť možná příliš ambiciózní, očekávání bohužel významně nepřesáhla.

3. Technická dokumentace

3.1 Sestavení a spuštění

Projekt je dostupný na Git repozitáři, lze ho tudíž jednoduše naklonovat příkazem:

```
git clone https://github.com/Hello007/UberRender
```

Následně je třeba jednotlivé subprojekty otevřít ve vývojářském prostředí - vzhledem ke kompatibilitě s GUI formuláři doporučuji starší, ale spolehlivé IDE NetBeans (testováno s verzí 12.4). Je nutné upozornit, že některé projekty závisejí na jiných, pro vyhnutí se problémům s hledáním závislostí by tudíž měly být otevřeny v pořadí:

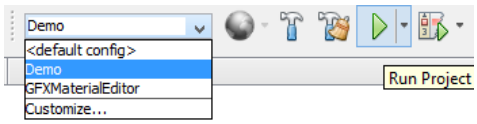
`CommonLibrary`

`LibUberRender`

`UberRenderDemo`

Při načítání některé projekty mohou vyžadovat nalezení kompatibilní vývojářské sady Java - v takovém případě postačí jakákoli verze JDK novější nebo rovná 1.8. Projekty aktuálně používají nativní knihovny cílené na operační prostředí Windows, pakliže používáte jiné, bude nutné změnit referenci na knihovny `gluegen-rt-natives` a `jogl-all-natives` na balíček příslušný cílové platformě ve složce `lib` v kořenové složce repozitáře.

Veškeré spustitelné komponenty projektu se nacházejí v balíčku `urender.demo` projektu `UberRenderDemo`. Projekt obsahuje předpřipravené konfigurace `Demo` a `GFXMaterialEditor` pro spuštění těchto komponent přímo z IDE pomocí rozevíracího seznamu a tlačítka `Run` (obr. 3.1).



Obrázek 3.1: Spuštění demonstračních programů

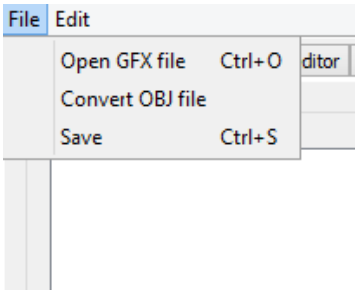
3.2 Demonstrační programy

3.2.1 uGFX MaterialEditor

Editor materiálů pro formát uGFX poskytuje základní funkce pro úpravu parametrů a prostředků materiálů a konverzi 3D modelů a textur do příslušných odnoží tohoto formátu.

Nejprve je třeba načíst zdrojový soubor pro provádění úprav. Lze tak učinit pomocí submenu `File` v nástrojové liště (obr. 3.2).

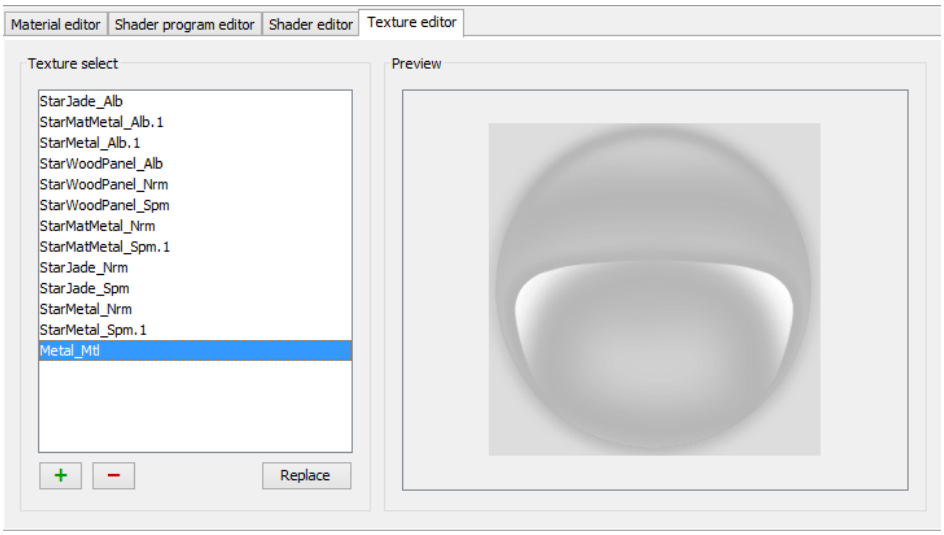
Následně lze buď stisknutím tlačítka `Open GFX file` načíst již existující kontejner formátu uGFX,



Obrázek 3.2: MaterialEditor - načtení souboru

nebo převést 3D model ve formátu OBJ (tlačítko `Convert OBJ file`). Načtené soubory se neukládají automaticky, je tedy důležité je po provedení úprav uložit buď tlačítkem `Save` nebo klávesovou zkratkou `Ctrl+S`.

Editor textur

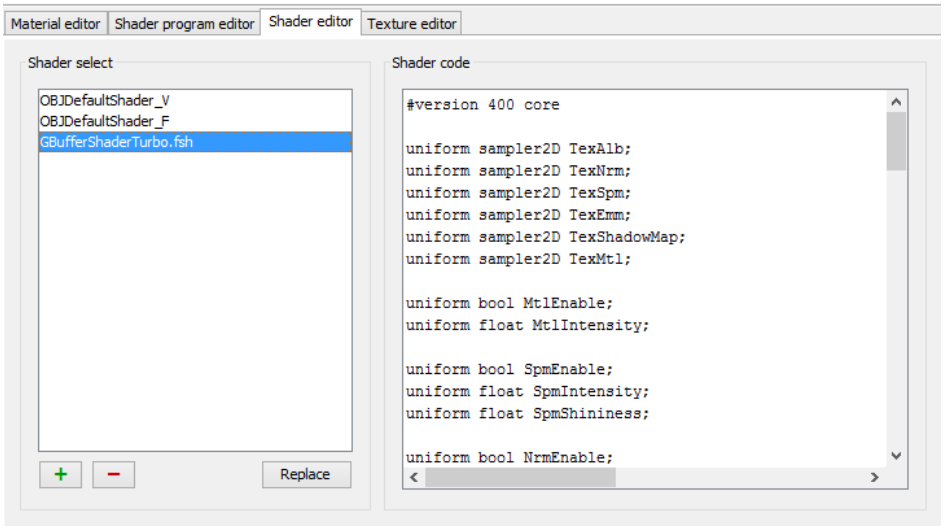


Obrázek 3.3: MaterialEditor - editor textur

Editor textur poskytuje fukncionalitu pro základní úpravy obrazových prostředků. Dovoluje buď importovat textury v běžných formátech obrazu do načteného kontejneru, nebo je vymazat, pakliže nejsou aktuálně použité v materiálu (obr. 3.3). Existující textury lze také jednoduše vyměnit za nové načtené z disku bez odstraňování/znovupřidávání pomocí tlačítka `Replace`.

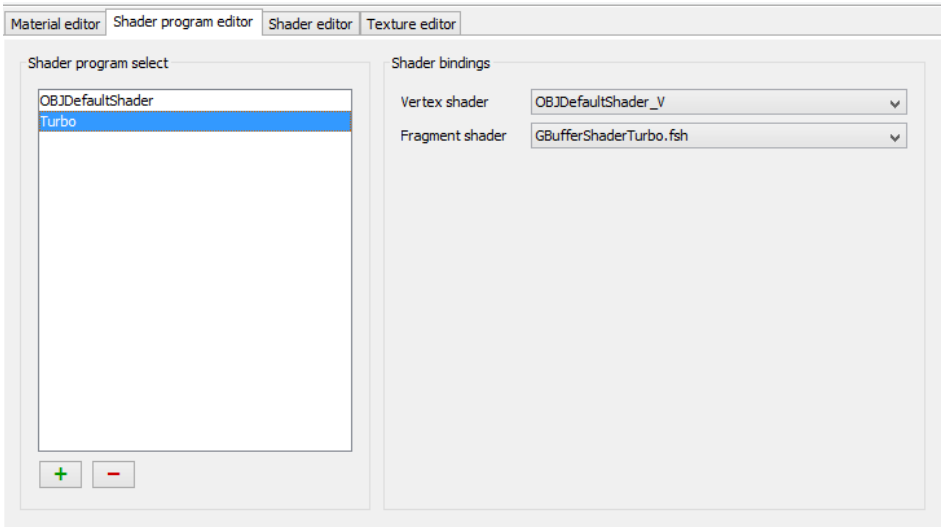
Editor shaderů

Editor shaderů umožňuje importovat a upravovat shadery v jazyce GLSL. Podporovány jsou formáty prostého textu s příponou `.vsh` pro vertex shadery a `.fsh` pro fragment shadery. Přestože editor disponuje základním textovým polem pro úpravu kódu přímo v editoru (obr. 3.4), doporučuje se vzhledem k jeho nepříliš rozvinutým schopnostem formátování používat externí textové editory a změny pouze synchronizovat aktualizací z disku pomocí tlačítka `Replace`. Shader nelze odebrat, pokud je aktuálně používán v shader programu.



Obrázek 3.4: MaterialEditor - editor shaderů

Editor shader programů



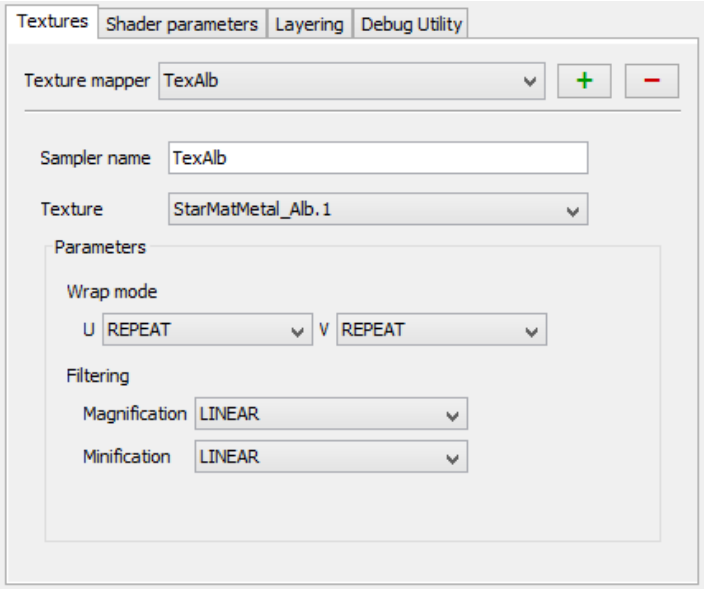
Obrázek 3.5: MaterialEditor - editor shader programů

Vertex a fragment shadery samy o sobě nelze přímo použit s materiálem, je třeba je nejdříve propojit do párů vertex + fragment shader nazývaných programy. Lze tak učinit pomocí karty `Shader program editor`, kde k přiřazení těchto prostředků poslouží dva rozevírací seznamy (obr. 3.5). Shader program není možné odebrat, pokud je aktuálně používán materiálem.

Editor materiálů

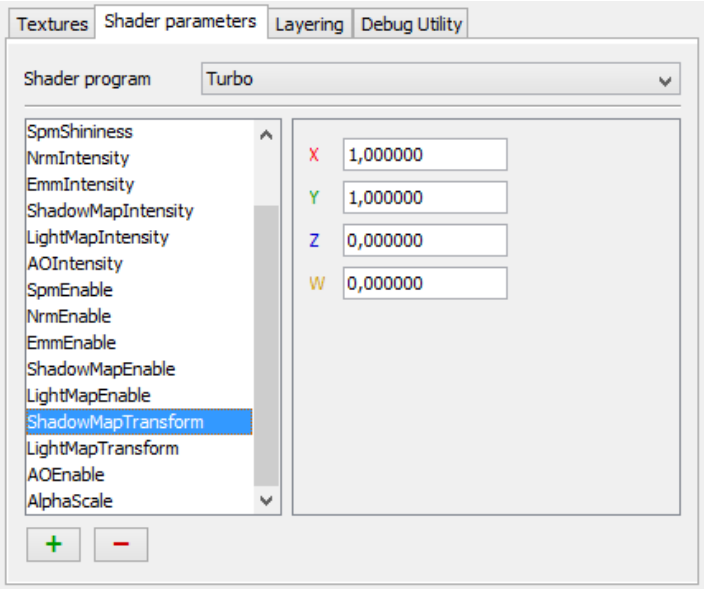
Editor materiálů je sice nekomplexnější, ale ani zdaleka ne složitá součást programu. Umožňuje upravovat libovolné parametry specifické pro podléhající shadery, přidávat či odebírat reference textur a konfigurovat metodu a vrstvu stínování.

Na panelu `Textures` (obr. 3.6) lze přidávat a upravovat samplery textur propojené s texturovými



Obrázek 3.6: MaterialEditor - editor referencí textur

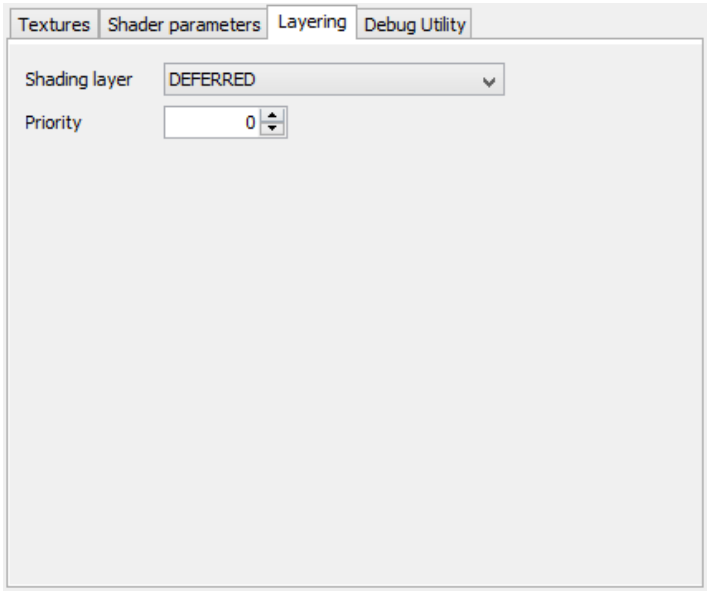
prostředky uvnitř kontejneru. Vedle textového pole pro název sampleru uvnitř shaderu a seznamu dostupných prostředků lze též upravovat parametry pro filtrování textur při zmenšování a zvětšování a nastavení opakování UV souřadnic za hranicemi textury.



Obrázek 3.7: MaterialEditor - editor parametrů shaderu

Panel `Shader parameters` (obr. 3.7) slouží k úpravám hodnot parametrů shaderu a výběru shader programu používaného pro vykreslování materiálu. Parametry mohou mít buď jednoduchou celou či desetinnou skalární hodnotu, nebo mohou reprezentovat až čtyřkomponentový vektor. Datový typ hodnoty po přidání již z bezpečnostních důvodů nelze změnit.

Poslední uživatelský panel editoru materiálů slouží ke konfiguraci vykreslovací vrstvy v rámci scenegraphu UGfxRenderer. Každému materiálu lze přiřadit buď přímý nebo odložený způsob vykreslování a prioritu/vrstvu (obr. 3.8) v rámci všech materiálů využívající stejnou techniku vykreslování (materiály s nižší prioritou budou seřazeny a vykresleny jako první). Tlačítka `Deferred`



Obrázek 3.8: MaterialEditor - editor konfigurace vrstev

`-> Forward` a `Forward -> Deferred` v submenu `Edit` umožňují dávkovou změnu metody stínování pro všechny materiály v kontejneru, je však nutno mít na paměti, že tento proces sám o sobě pouze změní tento atribut materiálu a neprovede např. změny v kódu shaderů potřebné pro vykreslování do G-Bufferu.

Utilita pro ladění

Utilita pro ladění v editoru materiálů je určená pro rychlou operaci s prostředky užívajícími shader model Turbo. Její hlavní funkcí je automatické nastavování parametrů shaderu a generování referencí na textury potřebné pro plnou kompatibilitu s osvětlovacím modelem (bump mapping, specular textury, předvypočítané osvětlení).

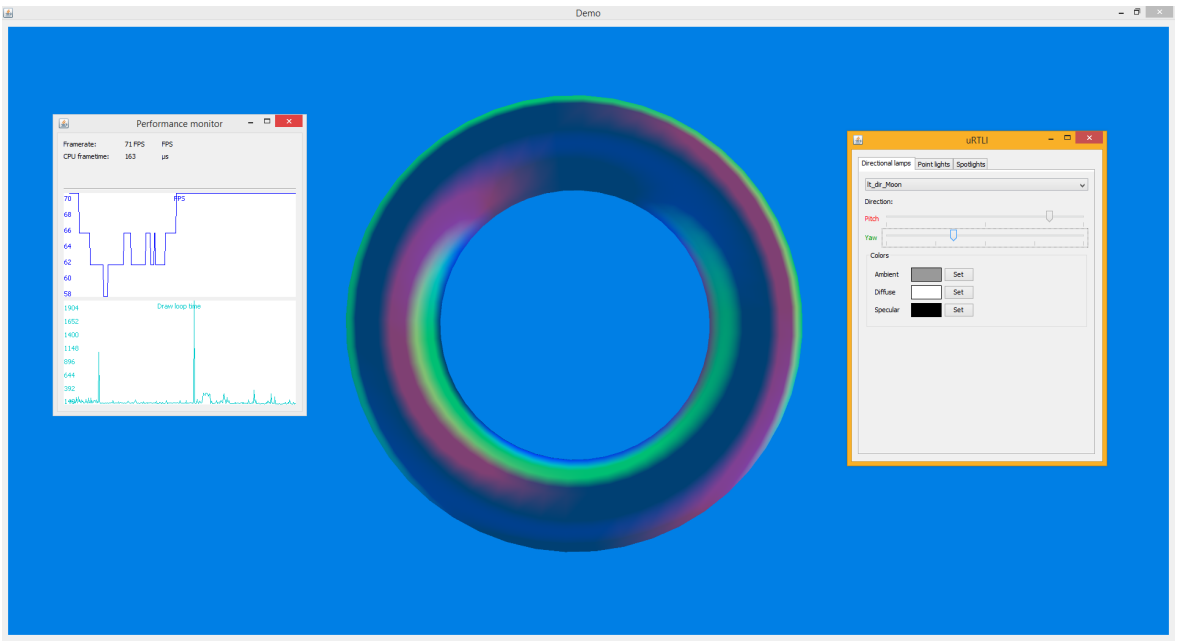
Tlačítko `Set Turbo shader parameters` na základě vybraných zatržítek nastaví následující proměnné v materiálu:

`TexAlb` - reference na Albedo texturu
`TexNrm` - reference na Normal/bump map texturu (jméno Albedo textury s příponou `_Nrm`)
`NrmEnable` - Zapnuto, pokud se Normal texturu podařilo najít v kontejneru a pokud je vybrané zatržítko `Normal/Specular texture`.
`NrmIntensity` - 100%/0% dle hodnoty `NrmEnable`
`TexSpm` - reference na Specular texturu (jméno Albedo textury s příponou `_Spm`)
`SpmEnable` - Zapnuto, pokud se Specular texturu podařilo najít v kontejneru a pokud je vybrané zatržítko `Normal/Specular texture`.
`SpmIntensity` - 100%/0% dle hodnoty `SpmEnable`
`SpmShininess` - 256 (exponent spekularity)
`TexEmm` - reference na Emission texturu (jméno Albedo textury s příponou `_Emm`)
`EmmEnable` - Zapnuto, pokud se Emission texturu podařilo najít v kontejneru a pokud je vybrané zatržítko `Emission texture`.
`EmmIntensity` - 100%/0% dle hodnoty `EmmEnable`

`TexShadowMap` - prázdná hodnota
`ShadowMapEnable` - Zapnuto, pokud je vybrané zatržítko `Shadow bake map`
`AOEnable` - Shodné s hodnotou `ShadowMapEnable`
`ShadowMapIntensity` - 50%
`AOIntensity` - 100%
`ShadowMapTransform` - (0; 0; 0; 0)
`TexLightMap` - prázdná hodnota
`LightMapEnable` - Zapnuto, pokud je vybrané zatržítko `Light bake map`
`LightMapIntensity` - 100%
`LightMapTransform` - (0; 0; 0; 0)
`MtlEnable` - Zakázáno
`AlphaScale` - 1.0 - multiplikátor hodnoty alfa kanálu

Tlačítko `AutoTurbo all` pak provede tyto operace pro všechny materiály v kontejneru.

3.2.2 DemoEngine



Obrázek 3.9: Okno DemoEngine

Demonstrační vykreslovací engine systému URender (obr. 3.9) lze spustit pomocí konfigurace `Demo`. Po spuštění vyzve uživatele k otevření grafického kontejneru ve formátu uGFX, ze kterého po potvrzení načte 3D scénu. Pro konzistenci testů napříč zařízeními je vykreslování zamknuto na rozlišení 4K (3840x2160 pixelů) avšak v případě potřeby jej lze upravit změnou konstant `FIX_RESOLUTION_W` a `FIX_RESOLUTION_H` ve třídě DemoSurface, nebo případně povolit použití dynamického rozlišení dle velikosti okna zakázáním vlajky `FIX_RESOLUTION_USE` té samé třídy. Pro lepší vyhlazování hran při dynamickém rozlišení lze též povolit supersampling (SSAA) nastavením konstanty poměru rozlišení nadvzorkovaného obrazu `SUPERSAMPLING_SCALE` na hodnotu vyšší než 1.

Klávesa `mezerník` přepíná mezi uživatelskou kamerou a předdefinovanou animací. Uživatelskou kameru lze ovládat pomocí myši - pohyb při stisknutém levém tlačítku myši kameru otáčí, ob-

dobný pohyb se stisknutým pravým tlačítkem mění její translaci. Kolečko myši kameru přibližuje a oddaluje.

Vedle hlavního vykreslovacího okna program disponuje i jednoduchým vedlejším panelem pro monitorování výkonu, který zobrazuje graf snímkové frekvence a dobu běhu hlavní vykreslovací rutiny na procesoru (obr. 3.9).

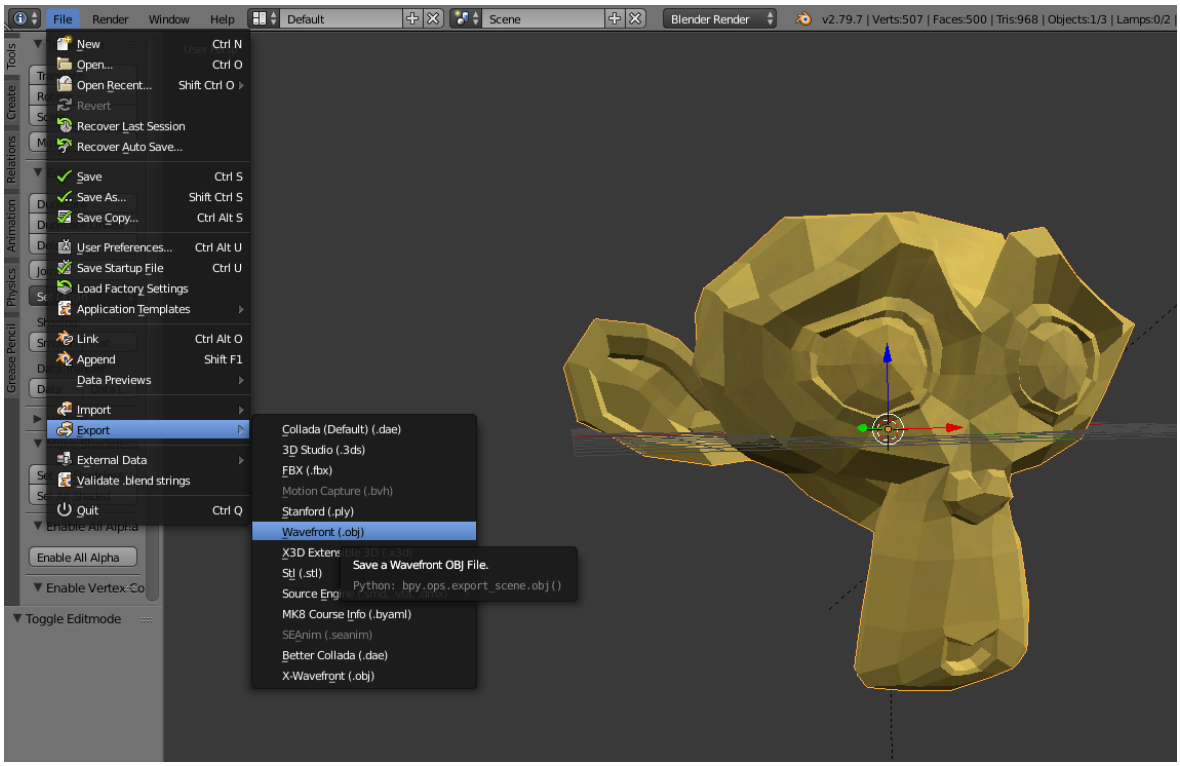
Druhý vedlejší panel slouží k základním úpravám světel, ale jedná spíše o vývojářský nástroj, který je aktuálně propojený s výchozí konfigurací světel pro demonstrační scénu.

3.3 Manuál k shader modelu Turbo

3.3.1 Kompatibilita s 3D softwarem

Konverze 3D modelů pro shader model Turbo využívá rozšíření formátu Wavefront OBJ pro podporu dvou vrstev UV souřadnic. Dokáže sice fungovat bez problému i pouze s jednou vrstvou, avšak i přesto je doporučeno pro export modelů používat rozšíření `io_scene_xobj` pro 3D software Blender verze 2.79 ze složky `blenderplugin` v projektu `UberRenderDemo`.

3.3.2 Export a import 3D modelu



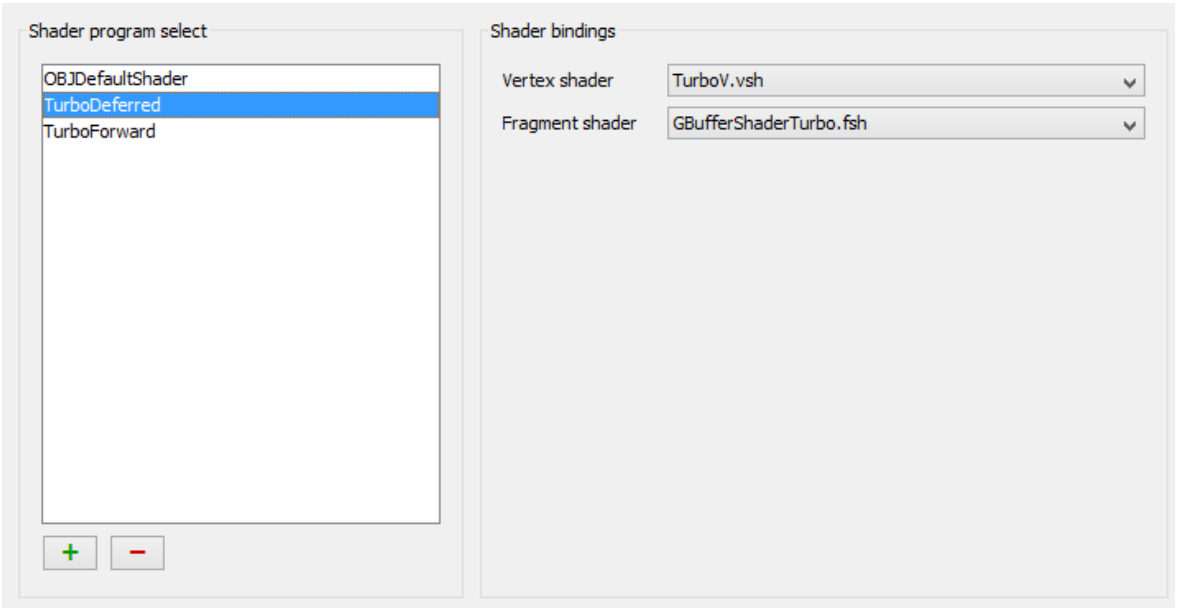
Obrázek 3.10: Export modelu ze 3D softwaru Blender

Model vyexportujte *triangulovaný* ve formátu Wavefront OBJ (obr. 3.10) s up-vektorem rovnoběžným s osou Y a s povoleným zápisem normál a UV souřadnic. Následně spusťte uGFX MaterialEditor a

otevřete vytvořený soubor .obj pomocí `File -> Convert OBJ file`.

3.3.3 Zahrnutí shaderu

Ze složky `shaders/Turbo` v projektu `UberRenderDemo` pomocí tlačítka `+` na kartě `Shader editor` importujte shadery `TurboV.vsh` (společný vertex shader), `ForwardShaderTurbo.fsh` (fragment shader pro přímé stínování) a `GBufferShaderTurbo.vsh` (fragment shader pro odložené stínování).



Obrázek 3.11: Shader program Turbo

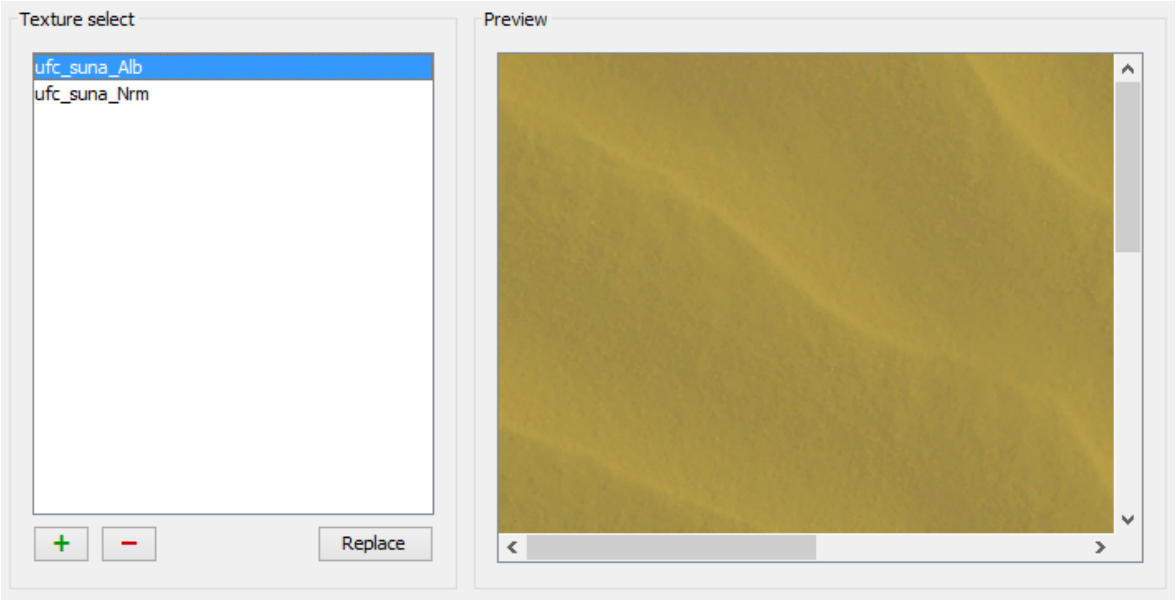
Následně na kartě `Shader program editor` vytvořte libovolnou kombinaci sdíleného vertex shaderu a jednoho z fragment shaderů (obr. 3.11). Pokud si chcete ulehčit práci a plánujete používat pouze jednu z metod stínování pro celý model, hodí se pojmenovat tento program `Turbo` pro rychlejší práci s automatickým generátorem shader parametrů.

3.3.4 Nastavení materiálů

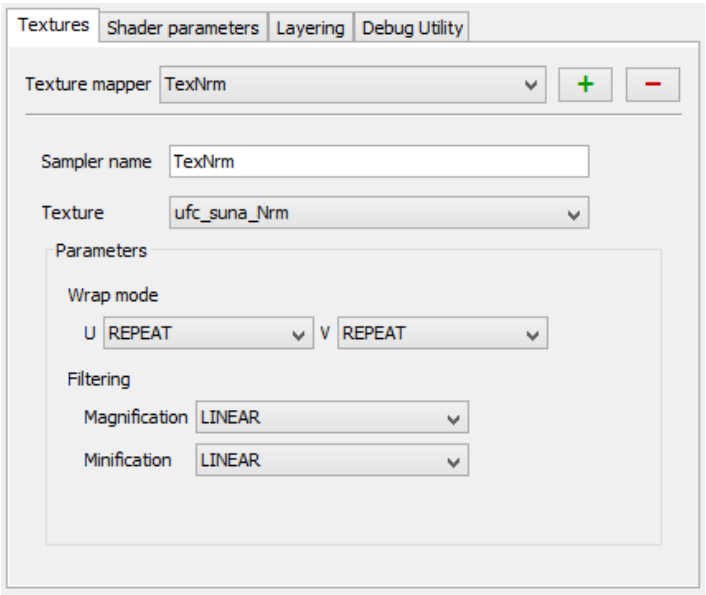
Ještě než začneme nastavovat parametry materiálů, je dobré předem naimportovat všechny potřebné textury včetně textur pro bump mapping nebo spekularitu (obr. 3.12) - urychlí to tak automatické generování parametrů. Jakmile jste si jisti, že máte všechno, co potřebujete, přepněte do editoru materiálů.

Zde nejprve v utilitě pro ladění klikněte na tlačítko `AutoTurbo all`. Program se nejdřív pokusí automaticky odvodit parametry materiálu pro standardní názvy textur. Tyto názvy jsou:

`<název textury>_Alb` - Albedo textura
`<název textury>_Nrm` - Normal textura
`<název textury>_Spm` - Specular textura
`<název textury>_Emm` - Emission textura



Obrázek 3.12: Phong textury



Obrázek 3.13: Normal textura dle konvence Turbo

Pokud daný materiál nemá textury pojmenované dle této konvence (obr. 3.13), operace selže a je třeba použít tlačítko `Set Turbo shader parameters` v konjunkci se zatržítky pro manuální vygenerování parametrů.

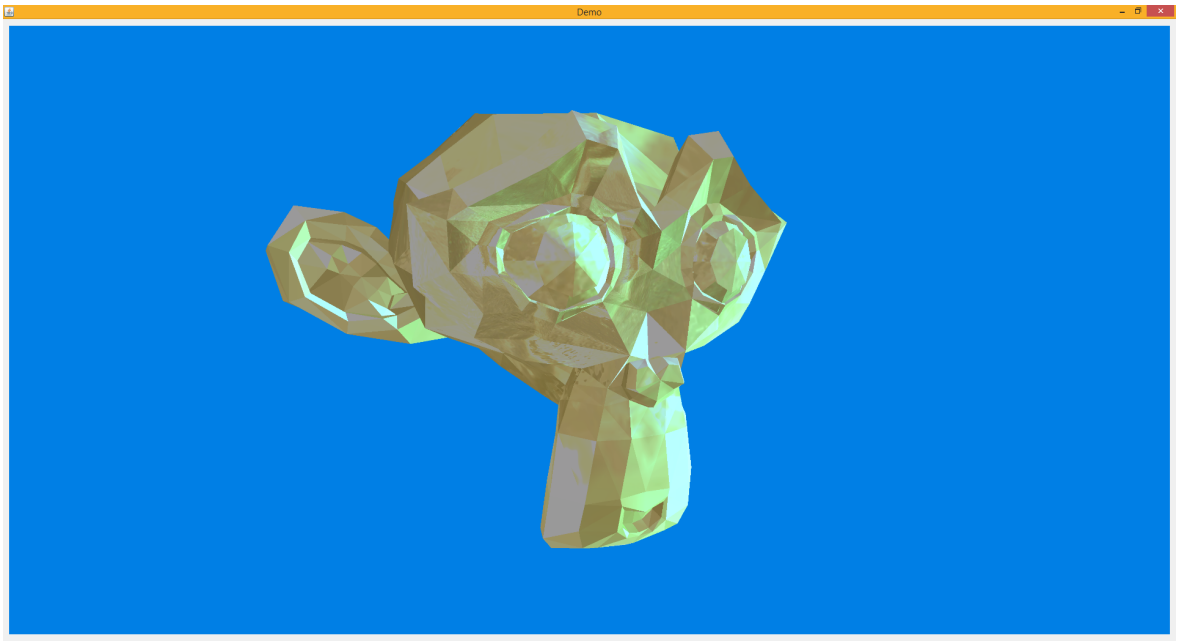
Dále se na panelu `Shader parameters` ujistěte, že materiál používá vámi vytvořený shader program. Pokud ano, můžete nyní pro jednotlivé speciální textury upravit intenzitu jejich vlivu na osvětlovací rovnici (proměnné `Intensity`), případně některé z nich úplně vypnout (proměnné `Enable`). Též můžete pro libovolný materiál nastavit konstantní multiplikační faktor pro alfa kanál albedo textury, využívaný pro průhlednost, v proměnné `AlphaScale`.

Imitace kovů

V shader modelu Turbo existuje základní podpora pro imitaci kovů pomocí techniky běžně nazývané *matcap*. Pokud váš materiál disponuje sphere-map texturou pro tuto techniku, lze ji použít připojením do sampleru `TexMtl`, povolením parametru `MtlEnable` a nastavením `MtlIntensity` na patřičnou hodnotu.

3.3.5 Testování

Modely využívající shader model Turbo lze přímo otestovat v demonstračním programu DemoEngine (obr. 3.14).



Obrázek 3.14: DemoEngine - Suzanne

3.4 Programátorská příručka knihovny URender

3.4.1 API a správa referencí

Low-level API knihovny URender disponuje rozsáhlou dokumentací ve formátu Javadoc, která pokrývá většinu důležitých funkcí pro interakci s grafickým backendem. Tato kapitola se tedy bude zabývat především návodem, jak propojit jádro API s poskytovatelem rozhraní OpenGL.

Třída `GLRenderingBackend` má sama o sobě výchozí konstruktor, avšak vyžaduje dodatečné nastavení po zavedení vykreslovacího systému. Při každé změně objektu GL4, která může dle JOGL proběhnou z libovolných implementačních důvodů, je nutné na instanci `GLRenderingBackend` zavolat funkci `setGL(GL4 gl, int identity)`, která jí poskytne „jádro“ pro provádění OpenGL operací. Parametr `identity` slouží jako unikátní identifikátor backendu ve správci referencí - pokud se za

běhu aplikace nezmění OpenGL kontext, měla by tato hodnota zůstat pro jednu instanci `GLRenderingBackend` stále stejná, bez ohledu na změny objektu GL4.

Při programování přímo s API OpenGL je často zvykem pečlivě šetřit změnami stavů rasterizéru jako funkce hloubkového testu nebo aktuální shader program, abychom snížili počet systémových volání. Vykreslovací backend knihovny URender se však o tyto úspory stará automaticky, sám sleduje poslední použitou hodnotu stavu a vyhne se systémovým voláním, pokud je jakákoli příchozí hodnota stejná. Není tak nutné sledovat hodnoty stavů rasterizéru na straně implementora.

Oproti state machine rozhraní OpenGL je abstraktní API třídy `RenderingBackend` navrženo jako soběstačná volání nezávislá na stavu. Všechny funkce, které pracují s referencí objektu třídy `UObjHandle` se tak starají o správu volání typu `glBindBuffer` v zákulisí backendu a nevyžadují tak další zásahy do stavu ze strany implementora. Výjimkou jsou funkce `framebufferBind(UObjHandle fb)`, resp. `framebufferResetScreen()`, které nastavují aktuální cílový framebuffer pro následující vykreslovací operace a vyžadují tak explicitní připojení.

Manuálně lze třídu `UObjHandle` použít i pro jiné proměnné než reference objektů - výchozí vykreslovací engine URender v ní ukládá například poslední použité rozměry framebufferu. Přiřazování hodnot by mělo proběhnout zpravidla pro každý backend pouze jednou, zavoláním metody `initialize(RenderingBackend backend, int value)`, avšak několikanásobná inicializace není zakázána. Pokud se z nějakého důvodu inicializace nezdaří, lze tak indikovat nastavením rezervované hodnoty `-1`, na základě níž volání metody `isValid(RenderingBackend backend)` vrátí `false`. Doporučuje se též využívat metody `getAndResetForceUpload(RenderingBackend backend)` v každé vykreslovací rutině i na již inicializovaných a načtených objektech - usnadníte si tak správu aktualizací objektů ze strany klienta vyvolaných procedurou `reset()`.

3.4.2 Sdílené principy high-level objektů

Většina high-level objektů jako `UShader` či `UTexture` se chová jako abstrakce nad grafickým prostředkem ve videopaměti, dostupná ostatním prostředkům pomocí *lokálního jména* - libovolného řetězce unikátního v rámci nadřazeného kontextu. Objekty lze v rámci kolekcí vyhledávat dle jména pomocí funkce `UGfxObject.find(Collection<T> collection, String name)`. Každý objekt základní třídy `UGfxEngineObject` nebo `UGfxScenegraphObject` navíc disponuje metodou `getType()`, která vrátí výčtový typ `UGfxEngineObjectType`, resp. `UGfxScenegraphObjectType` reprezentující třídu objektu pro bezpečný převod typů bez nutnosti používání operátoru `instanceof`.

Pro synchronizaci stavu klienta a dat ve videopaměti slouží metoda `setup`, která může vyžadovat pro každou třídu jiné speciální parametry popsané v dokumentaci Javadoc. Zavolání této metody zajistí plnou dostupnost daného objektu při vykreslování. Účinky metody `setup` lze obrátit pomocí metody `delete(RenderingBackend backend)`, která data uložená ve videopaměti smaže. Používání high-level objektů tak zpravidla nevyžaduje přímá volání do vykreslovacího backendu.

3.4.3 Shadery a shader programy

Shadery jsou v knihovně URender spravované třídou `UShader`. Lze je vytvořit pomocí konstruktoru `UShader(String name, UShaderType type, String source)` ze zdrojového kódu ve formátu prostého textu.

Shadery jsou dále komponentami shader programů reprezentovaných třídou `UShaderProgram`. Tato třída se mimo linkování shaderů stará i o správu atribut a uniforem programu. Podobně jako u stavů rasterizéru není nutné externě sledovat lokace a hodnoty atribut/uniforem - metoda `setUniform(UUniform u, RenderingBackend rnd)` se v případě uniforem postará o oboje, pro atributy si metoda `getAttributeLocation(RenderingBackend rnd, String name)` pamatuje lokace dříve vyžádaných symbolů. Shader program lze sestavit jako referenci na lokální jména podléhajících shaderů konstruktorem `UShaderProgram(String name, String vshName, String fshName)` a používat voláním metody `setup(RenderingBackend rnd, List<UShader> shaders)`, kde parametr `shaders` reprezentuje seznam shaderů, ve kterých má program vyhledávat podle lokálních jmen.

Aktuální shader program, který bude použit pro všechny následovné vykreslené objekty, lze nastavit zavoláním metody `use(RenderingBackend rnd)` na objektu třídy `UShaderProgram`.

3.4.4 Materiály a parametry shaderů

Třída `UMaterial` sama o sobě je poněkud nevšední - s nástupem programovatelných grafických procesorů totiž koncept „materiálu“ co do hardwaru prakticky neexistuje. Třída tak reprezentuje pouze instrukce pro použití shaderu a sadu parametrů a referencí textur. Materiál lze sestavit pomocí třídy `UMaterialBuilder` a aplikovat na shader program metodou `configureShader(UShaderProgram shader, RenderingBackend rnd, List<UTexture> textures)`, kde parametr `textures` je kolekce objektů typu `UTexture` prohledávaná pro potřebné prostředky referencí textur.

Parametry shaderů lze předávat jako instance třídy `UUniform`, resp. jejích podtříd pro konkrétní datové typy. V rámci materiálu pro uchovávání parametrů slouží pole `shaderParams`, na úrovni balíčku `scenegraph` lze též přidávat sdílené uniformy jako transformační matice nebo počítadlo času pro všechny objekty ve scéně pomocí metody `addGlobalUniform(UUniform uniform)` třídy `UScene`.

3.4.5 Textury

Třídu `UTexture` sama o sobě není možné instancovat, jedná se totiž o základní třídu pro jednotlivé druhy textur - `UTexture2D` pro standardní 2D textury a `UTexture2DCube` pro cubemapy. 1D a 3D textury zatím nejsou v high-level abstrakci podporovány. Oba typy tříd 2D textur disponují builder třídou pro konstrukci a sdílenou metodou `setup` pro nahrání do videopaměti. Málo známou, ale velmi užitečnou funkcí textur je mixování kanálů podporované třídou `UTextureSwizzleMask`, které dovoluje přepojení hodnot duplicitních kanálů nebo nastavení konstantních hodnot 0 či 1 pro celý barevný kanál, čímž při vysokých rozlišeních dosahuje drastického uspoření videopaměti např. u textur bez průhlednosti nebo jiných textur s méně než čtyřmi použitými kanály.

URender aktuálně podporuje 11 formátů textur, z nichž většině je zaručena funkčnost ve všech implementacích OpenGL. Detaily těchto formátů jsou dostupné v dokumentaci třídy `UTextureFormat`.

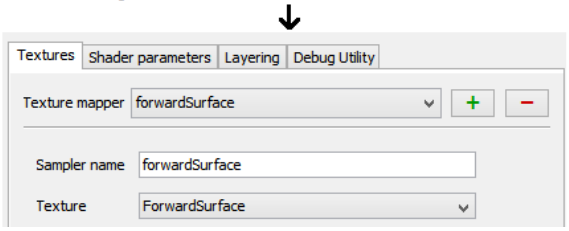
3.4.6 Geometrie

Geometrie je v knihovně URender součástí třídy `UMesh` jako nezpracovaný index buffer a vertex buffer. Konstrukce třídy by se tudíž ve většině případů měla přenechat třídám balíčku `g3dio`. Geometrii lze jednoduše vykreslit metodou `draw(RenderingBackend rnd, UShaderProgram program)`, avšak volání této metody předpokládá, že uživatel předem nastavil proměnné shaderu jako transformační matice, světla, textury apod. - tato funkce pouze nastaví atributy programu a zavolá funkci backendu pro vykreslení vertex bufferu. Referenční implementací správného toku vykreslování jest v tomto případě balíček `scenegraph`, konkrétně pak třída `URenderQueueMeshState`.

3.4.7 Podpora balíčku SceneGraph

Třídy scenegraphu URender používají namísto API třídy RenderingBackend vlastní abstraktní jádro UGfxRenderer, která je před použitím třeba rozšířit o implementace metod specifických pro shader model a rozložení framebufferů. Především je potřeba připravit si dva hlavní framebuffery - jeden pro odložené a druhý pro přímé stínování. Každý framebuffer je v základě zafixovaná sada pojmenovaných vykreslovacích cílů - lze jej tedy vytvořit pomocí konstruktoru `UFramebuffer(URenderTarget... renderTargets)`. Důležité je mít na paměti, že obsah framebufferu bude v pozdějším stádiu vykreslování nutné namapovat jako texturu na celoobrazovkový obdélník, tudíž je potřeba připravit si pro tento účel zároveň jednoduchý 3D model, jehož namapovaná textura (nebo v případě kompozitoru G-Bufferu více textur) v materiálu je pojmenovaná stejně jako vykreslovací cíl (obr. 3.15). Při nastavení materiálů se pak URender sám postará o přepsání placeholder textur daty přímo z framebufferu.

```
private final URenderTarget rtForward = new URenderTarget(
    0,
    "ForwardSurface",
    UFramebufferAttachment.COLOR,
    UTextureFormat.RGBA8
);
private final UFramebuffer framebufferForward = new UFramebuffer(
    rtForward,
    rtSharedDepth
); //shared depth buffer for both deferred and forward shading
```



Obrázek 3.15: Nastavení vykreslovacích cílů

Také je potřeba každému vykreslovacímu cíli přiřadit index - tento by se měl shodovat s indexem v rámci kvalifikátoru rozvržení ve fragment shaderu. Referenční implementací v tomto případě může být třída `DemoRenderEngine` a shader `shaders/Turbo/GBufferShaderTurbo.fsh`.

Další důležitou abstraktní třídou je rozhraní `ULightAdapter`, které slouží pro převod tříd `ULight` na uniformy pro použití ve fragment shaderu. Kompletní implementace tohoto rozhraní založená na osvětlovacím modelu Phong je přítomna ve třídě `TurboLightManager` a shaderu `shaders/Turbo/Turbo.fsh`.

Vykreslovací engine také může obsahovat sadu shader parametrů, které jsou sdílené pro všechny modely včetně kompozitorů G-Bufferu. Například v případě třídy `DemoRenderEngine` se jedná o transformační matice a vektor pozice kamery. Seznam všech takovýchto parametrů lze vykreslovacímu jádru předat přepsáním abstraktní metody `getSystemUniforms()`.

Nedůležitějšími abstraktními metodami jsou zbývající funkce `drawScene(UScene scene)` a `blitScreen()`. Tyto funkce nejsou součástí základní třídy, neboť využívají implementačně specifických framebufferů a prostředků. Funkce `blitScreen()` by se měla postarat o přepnutí na framebuffer obrazovky a vykreslení celoobrazovkového obdélníku přímého framebufferu. Funkce `drawScene(UScene scene)` je již o něco složitější, její součástí je totiž i implementace dočasného vícevrstvého G-Bufferu a průhlednosti. Každá takováto funkce by měla na začátku zavolat metodu `beginScene(scene)` pro vykreslovanou scénu a následně si nechat vypočítat vykreslovací frontu metodou `scene.calcRenderQueue()`. Poté může dle potřeby buď předem nebo až během vykreslování pro všechny objekty `DrawSources` ve vykreslovací frontě zavolat metodu `setup(UGfxRenderer rnd`, která zajistí připravenost prostředků ve videopaměti. Následuje implementačně specifický proces pro vykreslení jednotlivých vrstev scény - funkční implementace dočasného vícevrstvého G-Bufferu na základě návrhu z této práce je použitá ve třídě `DemoRenderEngine`.

3.4.8 Načítání prostředků do scény

Podporu načítání grafických prostředků ve formátu uGFX nebo Wavefront OBJ zajišťuje balíček G3DIO, resp. třídy `OBJModelLoader` a `UGfxResource` v konjunkci s rozšířeními formátu v balíčku `scenegraph.io`. Prostředky ve formátu uGFX lze načíst do instance třídy `USceneNode` pomocí adaptéru `USceneNodeGfxResourceAdapter` následujícím způsobem:

```
UGfxResource.loadResourceFile(
    new File(fileName),
    UScenegraphGfxResourceLoader.getInstance(),
    new USceneNodeGfxResourceAdapter(dest)
);
```

V tomto případě je `fileName` cesta ke zdrojovému souboru a `dest` odkaz na objekt `USceneNode`, do jehož seznamu prostředků se načte obsah kontejneru.

3.4.9 Kompozice a transformace ve scéně

Základními členy scény jsou instance třídy `USceneNode`, které podporují SRT transformace relativní k nadřazeným členům scény. Instanci `USceneNode` lze přidat jako kořenového člena scény metodou `addChild(USceneNode child)` třídy `UScene`, nebo jako člena podřazeného jinému přidáním do seznamu podřazených členů, ve výchozí konfiguraci dostupného zavoláním metody `getChildren()` na poli `parentRelation` nadřazeného člena.

Transformace i vztahy mezi členy scény jsou definovány abstraktními rozhraními `UNodeTransform` a `UParentRelation`. Výchozí implementace `UDefaultNodeTransform` a `UDefaultParentRelation` by měly pro většinu užití postačit, avšak v případě potřeby lze jejich metody přepsat podle instrukcí v dokumentaci Javadoc.

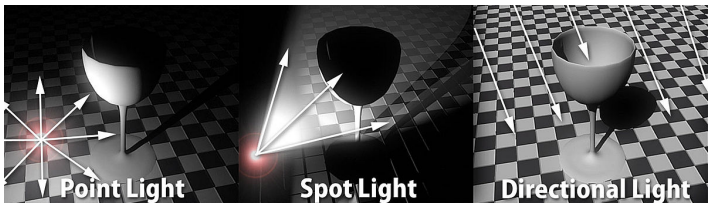
Další důležitou součástí kompozice scény je kamera. Scenegraph URender poskytuje čtyři základní typy kamery, které by měly bohatě vyhovět většině požadavků.

`UCameraOrtho` je kamera s ortogonální projekcí, translací a rotací v pořadí ZYX.
`UCameraViewpoint` je kamera s perspektivní projekcí, translací kamery a následnou rotací v bodě translace v pořadí ZYX.
`UCameraLookAt` je kamera s perspektivní projekcí, která vytváří transformaci sledující definovaný cíl z definovaného bodu s orientací definovanou směrovým vektorem.
`UCameraLookAtOrbit` je zjednodušená verze kamery `UCameraLookAt`, která sleduje definovaný cíl z bodu vypočítaného z YXZ rotace a definované translace od tohoto bodu.

Kameru lze použít ve scéně přiřazením reference veřejnému poli `camera` objektu `UScene`.

3.4.10 Osvětlení scény

Třída `UScene` obsahuje veřejnou kolekci `lights`, do které lze přidávat objekty podtříd typu `ULight` pro reprezentaci světel ve scéně.



Obrázek 3.16: Typy světel [Wiko9]

URender podporuje 3 základní typy světel - směrová, bodová a kuželová (obr. 3.16). Směrové světlo osvětluje objekty v každém bodě rovnoměrně a bez atenuace, zatímco intenzita bodových a kuželových světel postupně opadá se vzdáleností na základě koeficientů specifických pro jednotlivá světla. Kuželová světla navíc omezují maximální úhel mezi směrem zdroje světla a osvětleným bodem, podobně jako stínítko na lampě. Pro světla momentálně neexistuje výchozí serializér formátu uGFX - v demonstračním programu se definují manuálně a pomocí utility RTLI.

3.4.11 Rozšiřování formátu uGFX

Formát kontejnerů uGFX je navržen pro snadné rozšiřování o nové serializéry a deserializéry implementorem. Základem přidání podpory pro čtení a zápis nové třídy je implementace rozhraní `IGfxResourceSerializer<R>`. Zde je potřeba nejdříve zvolit unikátní čtyřznakový identifikátor pro všechny serializované instance třídy, který slouží pro identifikaci typu. Tento identifikátor by měl být vrácen přepsanou metodou `getTagIdent()` a neměl by se v žádném případě měnit.

Zároveň je pro správné rozeznání serializéru při zápisu nutné přepsat metodu `accepts(Object o)` tak, aby vrátila `true` pouze (a právě tehdy) když je serializér schopný plně zapsat daný objekt - kód bude pravděpodobně ve formátu `o instanceof <třída>`, avšak může mít i zvláštní případy pro rozeznávání podtříd.

Následně se již můžete vrhnout na implementaci samotných metod `serialize` a `deserialize` - mělo by se jednat o takřka symetrické metody, tj. to, co se zapíše, se opačnými voláními přečte. V

metodě `deserialize` je důležité věnovat pozornost parametru `IGfxResourceConsumer consumer` - po skončení čtení objektu by měl deserializér zavolat metodu `consumer.loadObject(o)`, kde `o` je deserializovaný objekt. Tento systém tak potenciálně umožňuje deserializaci několika pod-objektů pro vlastní implementaci třídy `IGfxResourceConsumer` např. pro streamování vertex bufferů do videopaměti, zatímco se zbytek geometrie načítá apod.

Serializér je nakonec ještě nutné zaregistrovat implementací rozhraní `IGfxResourceLoader`, resp. metod `getResourceSerializers` a `getEnumSerializers`, které vrátí instance všech serializérů dostupných pro daný serializační proces. Instanci (pravděpodobně singleton) tohoto rozhraní pak lze předat statickým funkcím třídy `UGfxResource` při deserializaci/serializaci.

Formát uGFX navíc disponuje zvláštní funkcionalitou pro konzistentní serializaci výčtových typů. Pro každý serializovatelný výčtový typ by mělo být implementováno rozhraní `IGfxEnumSerializer<E>` (pro většinu případů postačí výchozí implementace `AbstractGfxEnumSerializer<E>`) tak, aby zajišťovalo funkční čtení a zápis výčtových typů pomocí metod `readEnum` a `writeEnum` datových proudů uGFX mezi různými verzemi tříd. Výchozí implementace pro API a engine URender využívají lookup tabulek - k inspiraci tak mohou posloužit třídy `GfxAPIEnumSerializers` a `GfxEngineEnumSerializers`. Nové serializéry výčtových typů se registrují podobně jako serializéry prostředků pomocí vlastní implementace rozhraní `IGfxResourceLoader`.

Závěr

V závěru práce bych se nejprve vrátil k motivaci jejího konceptu. Původní záměr za několikavrstvým odloženým stínováním vyústil ze zájmu rozšířit škálu benefitů této stínovací techniky do scén využívajících průhlednost, neboť aktuální implementace povětšinou vykreslují průhledné objekty pomocí tradičních přímých postupů. Na základě výsledků práce však vyvstává poněkud filozofická otázka - zda to ve finále stojí za to.

Přestože se mi podařilo úspěšně implementovat navržený koncept tak, že splňuje většinu definovaných požadavků, zátěžové testy ukázaly, že přímé stínování je dosud velmi kompetentní technika, která v méně složitých scénách dokonce předčí odložené stínování, co se výkonu týče. Též je nutné si uvědomit, že žádná jedna průhledná vrstva nebude (nebo by spíše neměla) obsahovat více překrývajících se objektů, což efektivně anuluje jeden z hlavních benefitů odloženého stínování při vykreslování jiných vrstev než té základní, neprůhledné. Stále nám ještě zbývají benefity plynoucí ze skutečnosti, že máme předem vypočítaný G-Buffer, pro víceprůchodové vykreslování nebo screen-space post-processing, a mimo to jsme schopni pro všechny modely využívat sjednocený shader model, což nejenže zjednoduší používání, ale i omezí počet systémových volání při přepínání kontextů. I přesto jsem nucen konstatovat, že jsem si ze začátku představoval znatelnější náskok před přímým stínováním, než ukázaly zátěžové testy, avšak ve výsledku to naštěstí nevadí.

Technika odloženého stínování pro mne byla ve všech ohledech nová i přes mé předchozí zkušenosti v oboru počítačové grafiky. Tato práce pro mne byla v jistém směru tak trochu výsledkem dlouhodobějšího zájmu - jednalo se zčásti o experiment, o kterém jsem přemýšlel již od chvíle, co jsem se na diskuzním fóru rozebírajícím katastrofický grafický výkon některých titulů na herní konzoli Nintendo Switch dozvěděl o existenci a principu odloženého stínování. Zároveň jsem si konečně našel čas hlouběji pochopit výpočty použité pro real-time osvětlování a přiblížil se tak o krůček blíže další úrovni osvětlovacích modelů, konkrétně PBR, které jsem v době psaní této práce dosud nebyl s to implementovat.

Myslím, že vykreslovací engine URender má ještě spoustu prostoru pro nové či optimálnější funkce a rád bych někdy prodiskutoval svůj výstup s někým znalejším tohoto oboru, naskytne-li se mi příležitost. Do té doby pro mne bude, byť prozatím jen jako subjekt maturitní práce, stále představovat velmi zajímavý úvod do pokročilých technik světa počítačové grafiky.

Seznam použité literatury

[Alf] Alfonse. *Shader*. [online; cit. 2022-03-20]. URL: https://www.khronos.org/opengl/wiki/Shader.

[Eve] Cass Everitt. *Order-independent transparency - nvidia*. [online; cit. 2022-03-20]. URL: https://developer.download.nvidia.com/assets/gamedev/docs/OrderIndependentTransp
pdf.

[HH04] Shawn Hargreaves a Mark Harris. *Deferred shading*. [online; cit. 2022-03-20]. 2004. URL: https://http.download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf.

[KI10] John Kessenich a Intel. [online; cit. 2022-03-20]. Čvc. 2010. URL: https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.00.pdf.

[LL19] Jon Leech a Benj Lipchak. [online; cit. 2022-03-20]. Lis. 2019. URL: https://www.khronos.org/registry/OpenGL/specs/es/3.0/es_spec_3.0.pdf.

[Mag12] Christian Magnerfelt. *Transparency with Deferred Shading*. [online; cit. 2022-03-20]. 2012. URL: https://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand12/Group1Marten/final/final_TransparencyWithDeferredShading.pdf.

[NVI] NVIDIA. *Antialiased deferred rendering*. [online; cit. 2022-03-20]. URL: https://docs.nvidia.com/gameworks/content/gameworkslibrary/graphicssamples/d3d_samples/antialiaseddeferredrendering.htm.

[Ove] Alexander Overvoorde. *Shader modules*. [online; cit. 2022-03-20]. URL: https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Shader_modules.

[SA10] Mark Segal a Kurt Akeley. [online; cit. 2022-03-20]. Bře. 2010. URL: https://www.khronos.org/registry/OpenGL/specs/gl/glspec40.core.pdf.

[Vri] Joey de Vries. *Frustum culling*. [online; cit. 2022-03-20]. URL: https://learnopengl.com/Guest-Articles/2021/Scene/Frustum-Culling.

[Wik09] Wikipedia, the free encyclopedia/Tcpp. *File:Six Lights.jpg*. [online; cit. 2022-03-20]. 2009. URL: https://commons.wikimedia.org/wiki/File:SixLights.jpg.

Seznam obrázků