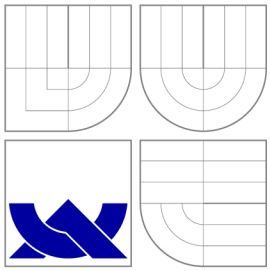
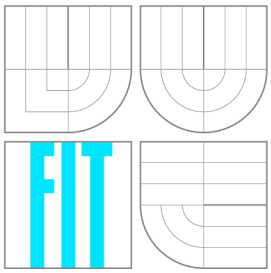


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

BENCHMARK PRO ZAŘÍZENÍ S PODPOROU OPENGL ES 3.0

BENCHMARK FOR OPENGL ES 3.0 DEVICES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ KIMER

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. LUKÁŠ POLOK

BRNO 2014

Abstrakt

Tato práce se zabývá tvorbou aplikace pro měření výkonnosti grafických akcelerátorů podporujících standard OpenGL ES 3.0, a to formou realistického zobrazování 3D scén v reálném čase. Nejprve je popsána historie a nové funkce grafické knihovny OpenGL ES 3.0. Poté jsou stručně rozebrány vybrané algoritmy realistického zobrazování 3D scén v reálném čase, které lze díky novým funkcím v knihovně implementovat. Dále je popsán návrh testovací aplikace, a to včetně online databáze výsledků obsahující podrobné informace o zařízeních. Následuje popis implementace na platformách Android a Windows, včetně popisu testování na mobilních zařízeních po publikování aplikace v obchodu Google Play. Závěrem jsou zhodnoceny dosažené výsledky a je zmíněno možné budoucí pokračování projektu.

Abstract

This thesis deals with the development of benchmark application for the OpenGL ES 3.0 devices using the realistic real-time rendering of 3D scenes. The first part covers the history and new features of the OpenGL ES 3.0 graphics library. Next part briefly describes selected algorithms for the realistic real-time rendering of 3D scenes which can be implemented using the new features of the discussed library. The design of benchmark application is covered next, including the design of online result database containing detailed device specifications. The last part covers implementation on Android and Windows platforms and the testing on mobile devices after publishing the application on Google Play. Finally, the results and possibilites of further development are discussed.

Klíčová slova

měření výkonnosti, mobilní zařízení, grafický akcelerátor, realistické zobrazování v reálném čase, deferred shading, screen space ambient occlusion, geometry instancing, Android

Keywords

performance measurement, mobile device, graphics processing unit, realistic real-time rendering, deferred shading, screen space ambient occlusion, geometry instancing, Android

Citace

Kimer, T. *Benchmark pro zařízení s podporou OpenGL ES 3.0*. Diplomová práce. Brno: FIT VUT v Brně, 2014.

Benchmark pro zařízení s podporou OpenGL ES 3.0

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Lukáše Poloka a že jsem uvedl všechny literární zdroje a publikace, ze kterých jsem čerpal.

.....
Tomáš Kimer
28. května 2014

Poděkování

Na tomto místě bych rád poděkoval vedoucímu práce Ing. Lukáši Polokovi za odborné vedení a cenné rady, kterými přispěl k vypracování této práce. Poděkování patří i rodičům za jejich podporu při studiu.

Obsah

Kapitola 1

Úvod

V posledních letech jsme svědky prudkého rozvoje mobilních zařízení, jejichž grafický výkon roste vysokým tempem. Hlavním motorem tohoto pokroku, stejně jako u osobních počítačů, jsou především počítačové hry, jejichž průmysl je svou velikostí již srovnatelný s průmyslem filmovým. Aplikace, které před pár lety plynule fungovaly pouze na velmi výkonných počítačích, jsou nyní přenášeny na mobilní zařízení. S tímto trendem souvisí také rozvoj mobilních grafických knihoven, které výkon grafických akcelerátorů zpřístupňují programátorům. Poslední verze nejpoužívanější z těchto knihoven, OpenGL ES, přinesla ve svém novém standardu množství nových funkcí, kterými se zase o kus přiblížila své plnohodnotné verzi z osobních počítačů, a díky nimž lze mobilní grafiku posunout zase o kus dál.

V této práci je řešeno měření výkonnosti zařízení podporujících jednu z posledních verzí tohoto standardu, OpenGL ES 3.0, a to formou realistického zobrazování 3D scén v reálném čase. Díky tomu je možné zjistit, jaké nové grafické efekty lze na nejnovějších zařízeních podporujících tento standard implementovat, která zařízení zvládají implementaci nových funkcí z tohoto standardu nejlépe, a to vše s cílem prezentovat tyto informace uživatelům a grafickým vývojářům pomocí podrobné databáze výsledků obsahující detailní informace o grafickém subsystému jednotlivých zařízení.

Práce je rozdělena do čtyř hlavních částí. V první části (kapitola 2) je popsána knihovna OpenGL ES 3.0, zejména její historie a nové funkce. V kapitole 3 jsou stručně popsány algoritmy pro realistické zobrazování 3D scén v reálném čase relevantní k využití na mobilních zařízeních pomocí popsané knihovny. Následuje popis návrhu testovací aplikace, včetně analýzy existujících aplikaci a návrhu webové databáze výsledků (kapitola 4). Poslední část (kapitola 5) popisuje implementaci a testování výsledné aplikace na systémech Android a Windows po zveřejnění v obchodu Google Play. Závěrem jsou zhodnoceny dosažené výsledky a je zmíněno možné budoucí pokračování projektu.

Kapitola 2

Knihovna OpenGL ES 3.0

OpenGL ES (*OpenGL for Embedded Systems*) je multiplatformní aplikační rozhraní (API) pro zobrazování akcelerované 2D a 3D počítačové grafiky na vestavěných systémech, jako jsou chytré telefony, tablety, herní konzole apod. Je založeno na původní plnohodnotné verzi OpenGL z PC platformy, kterého je (až na některé vyjímky) podmnožinou – aplikace vyvinuté pro ES verzi lze tedy jednoduše přenášet na odpovídající desktopovou verzi, opačně to nemusí být možné [15].

Mobilní GPU ve vestavěných systémech se v některých ohledech liší od GPU desktopových (proto se používá speciální verze API). Velkou roli hraje především spotřeba, zahřívání a plocha na čipu. GPU je také zpravidla implementováno společně s procesorem a dalšími komponentami na jedné ploše čipu – tzv. *system on a chip* (SoC).

2.1 Historie

Do této chvíle bylo vyvinuto několik verzí knihovny. Nejnovější v praxi používaná verze je verze 3.0 [15], které se především věnuje tato práce. Za dobu psaní tohoto textu již ale byla přestavena nová specifikace, která je zmíněna na konci kapitoly.

Verze 1.0

První verze knihovny byla vytvořena podle specifikace OpenGL 1.3., z níž bylo ale mnoho funkcí odstraněno. Podpora byla například pouze pro vykreslování primitiv polygonů a čtveřic (*quad*), nebyla podpora bitmapových operací, pokročilejších vykreslovacích módů, zobrazovacích seznamů (*display list*), či zásobníkových operací (funkce `glPush*` a `glPop*`).

Přítomny byly tzv. *common* a *common lite* profily [15]. První zmíněný podporuje jak výpočty v pevné řádové čárce, tak i plovoucí, druhý jen v pevné. Toto bylo zavedeno kvůli větší podpoře různých vestavěných zařízení, protože mnohá neobsahovala FPU (*floating point unit*).

Tato verze už není dále podporována a byla nahrazena verzí 1.1.

Verze 1.1

Tato aktualizace, vytvořena podle verze OpenGL 1.5, rozšířila předchozí funkčnost mimo jiné o podporu tzv. *buffer objektů* – *Vertex Buffer Object* (VBO) pro ukládání dat v paměti grafické karty, automatické generování *mipmap* v hardware, rozšířenou podporu texturování (minimální podpora alespoň dvou multi-textur a možnosti kombinovat textury pro efekty

jako *bump-mapping* a *per-pixel* osvětlení), dále podporu animací pomocí *vertex skinningu*, uživatelsky definovaných ořezávacích rovin (*clip planes*), či podporu pro vykreslování bodů namísto pouze čtveřic (pro efektivní a realistické částicové efekty). Tato verze, stejně jako předchozí, obsahuje common a common lite profily a je plně zpětně kompatibilní s verzí předchozí [15].

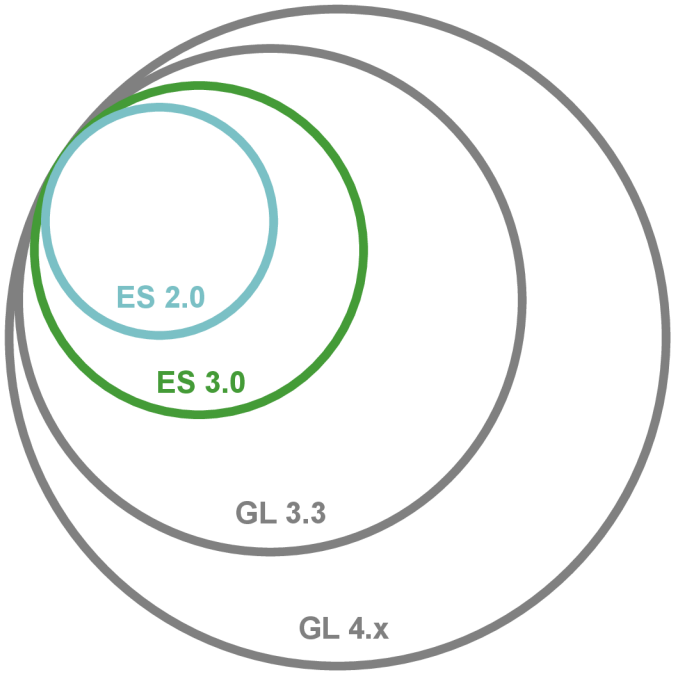
Verze 2.0

Tato aktuálně nejrozšířenější verze knihovny, představena roku 2007, byla vytvořena podle specifikace OpenGL 2.0 a přinesla programovatelný vykreslovací řetězec a s ním související podporu *vertex* a *fragment shaderů* psaných v jazyce GLSL (s mírnými modifikacemi pro vestavěná zařízení). Také byly přidána podpora pro *frame buffer objekty* (FBO).

Tato verze není plně zpětně kompatibilní s verzemi předchozími (ani s OpenGL 2.0), jelikož byly odstraněny některé neprogramovatelné (*fixed-function*) části vykreslovacího řetězce (lze je ale nahradit odpovídajícími shader programy) [15].

2.2 Nové funkce ve verzi 3.0

Specifikace OpenGL ES 3.0 byla zveřejněna 6. srpna 2012 [16]. Tato verze je nadmnožinou předchozí verze 2.0 a zároveň podmnožinou desktopových verzí OpenGL 3.3 a 4.x (viz obr. 2.1).



Obrázek 2.1: OpenGL ES 3.0 v kontextu ostatních verzí.

Z nových funkcí přinesla větší standardizaci a s ní související kompatibilitu, dále novinky ve vykreslovacím řetězci, texturování a shaderech.

2.2.1 Standardizace a kompatibilita

Jednou z hlavních novinek je standardizace funkcí, které dříve byly přístupny pouze jako nepovinná rozšíření. Díky tomu je omezena rozdílnost implementací na různých platformách, čímž je výrazně ulehčeno psaní přenositelných aplikací.

Tato standardizace se týká hlavně texturování – ETC2/EAC komprese textur je nyní jako povinná součást standardu, což eliminuje potřebu více různých sad textur pro různé platformy (pro různá mobilní GPU nebude třeba různých formátů textur jako u předchozích verzí). Dále rozsáhlá sada vyžadovaných formátů textur a vykreslovacích pamětí (*render buffer*) s explicitně definovanou velikostí (např. minimální velikost 2D textury je nyní 2048 místo 64, jak tomu bylo v OpenGL ES 2.0) [5].

2.2.2 Vykreslovací řetězec

Ve vykreslovacím řetězci přibyla mimo jiné podpora následujících technologií [5]:

Vícenásobné cíle pro vykreslování (*multiple render targets*, zkratka MRT) umožňují efektivní implementaci efektů jako je *deferred shading*, *screen space ambient occlusion*, *bloom* apod. v reálném čase (viz kap. 3). Podpora je minimálně čtyř a více cílů, a to včetně podpory *multisample antialiasingu* (MSAA).

Geometry instancing (také *instanced rendering*) umožňuje efektivně vykreslovat velké množství stejných objektů, a to každý s jinými parametry – např. stromy, trávu, listí, mnoho domů pro zobrazení rozsáhlých měst a podobně [18]. Ukázku lze vidět na obrázku 2.2.

Transform feedback umožňuje výstupy z vertex (či geometry na desktop OpenGL) shaderů ukládat do buffer objektů a následně s nimi dále pracovat, bez nutnosti jejich přesunu z grafické paměti (vše se odehrává pouze na GPU). Toto lze využít např. pro efektivní částicové systémy, jejichž částice jsou aktualizovány pouze pomocí shader programů [18].

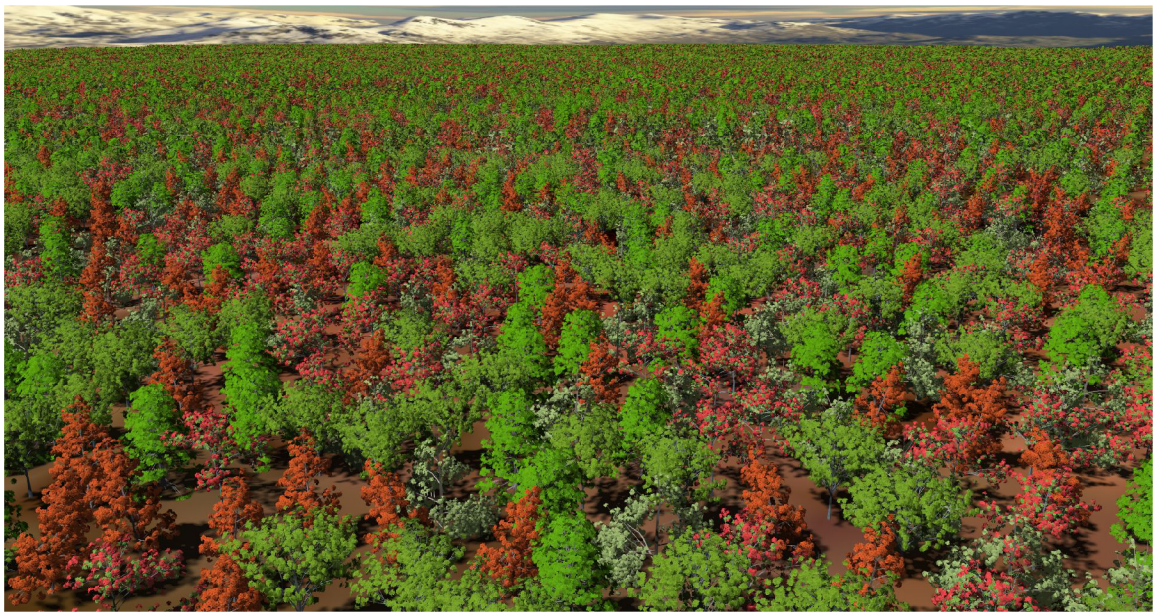
Occlusion queries přináší efektivnější vykreslování díky odstraňování zakrytých (a tudíž neviditelných) objektů z vykreslovacího řetězce hned v rané fázi jejich zpracování.

2.2.3 Texturování

Nových funkcí v texturování přibylo mnoho. Již výše zmíněná ETC2/EAC standardizovaná komprese textur – nové kodeky umožňují vyšší kvalitu při zachování stejné bitové šířky (oproti předchozí, nepovinné verzi ETC1), podporu alfa kanálu (celý anebo jednobitový) a podporu jedno (R) či dvou (RG) kanálových textur (EAC). Komprese má cca o 0.8 dB lepší kvalitu než u formátu S3TC/DXTC (jehož licence navíc není zdarma). ETC2 je zpětně kompatibilní s ETC1, navíc je možné ukládat obě verze do jednoho souboru, který bude díky sdílení bloků menší než obě textury uložené zvlášť [14]. Komprese textur přináší výhody ve vyšším výkonu díky lepšímu využití vyrovnávací paměti či v redukci spotřeby paměti na GPU.

Kromě komprese textur byla dále přidána podpora pro [5]:

- textury s plovoucí řádovou čárkou (16 i 32 bitů) – tzn. možnost implementace HDR renderingu (viz kap. 3.3), či obecných výpočtů;



Obrázek 2.2: Ukázka metody geometry instancing pro vykreslení rozsáhlého lesa[1].

- hloubkové (*depth*) textury – např. pro implementaci stínových map včetně podpory PCF filtrování;
- sRGB textury pro gamma korektní vykreslování;
- pole 2D textur, které lze využít např. pro texturové animace;
- 3D textury pro podporu *volume renderingu*;
- *seamless cubemaps* pro plynulé filtrování v okrajích;
- celočíselné (integer) textury a další formáty textur, jako např. textury se sdíleným exponentem pro efektivnější implementaci HDR renderingu, aj.;
- *non-power-of-2* (NPOT) textury, které nemusí mít velikost o násobcích dvou;
- *level of detail* (LOD) pro streaming mipmap, např. přes síť;
- *swizzles* pro možnost změny pořadí kanálů textury (R, G, B a A) při přístupu v shaderu;
- *immutable* (neměnné) textury, díky kterým nemusí ovladač dělat kontroly konzistence v době vykreslování.

2.2.4 Shadery

Nová verze GLSL ES 3.0 jazyka přidává mimo jiné níže zmíněné nové prvky. Za zmínku stojí, že podpora *geometry* shaderů stále přidána nebyla [5].

- Plná podpora celočíselných a 32-bitových operací v plovoucí řádové čárce.

[1] http://www.siggraph.org/asia2011/technical-sketches-detail?id=100-65&session=sketches

- Podpora ne-čtvercových matic.
- Uniformní bloky proměnných v shaderech a *uniform buffer objekty* (UBO).
- *Flat/smooth* interpolátory (v OpenGL ES 2.0 byla vždy použita lineární interpolace).
- *Layout* kvalifikátory pro vstupy vertex a výstupy fragment shaderů (odpadá nutnost volat funkci `glGetAttribLocation`).
- Povinný kompilátor shaderů a možnost ukládat slinkované shadery do binárního formátu pro zkrácení načítací doby aplikace.
- Nyní je ve vertex shaderu přístupná informace o indexu zpracovávaného vertexu (`gl_vertexID`) a také číslo instance (`gl_InstanceID`) při použití instanced renderingu.
- Fragment shader může nyní explicitně kontrolovat hloubku aktuálně zpracovávaného fragmentu.
- Nové vestavěné funkce pro podporu nových funkcí v texturování, matematických operací apod.
- Volnější limity – není omezena délka instrukcí, je plná podpora cyklů, skoků a indexování polí.

2.2.5 Ostatní

Dále stojí ještě za zmínku přidaná podpora *vertex array objektů* (VAO), *sampler objektů* (pro samostatné ukládání parametrů textur), a *sync objektů/fences* pro synchronizaci mezi CPU a GPU (např. pro vícevláknové vykreslování).

2.3 Podpora zařízení

Podpora pro OpenGL ES ve verzi 3.0 byla uvedena v následujících verzích mobilních operačních systémů:

- Android od verze 4.3, která byla představena 24. června 2013 [3],
- iOS od verze 7 a
- BlackBerry OS od verze 10.2.

Z mobilních GPU mají podporu mimojiné následující grafické akcelerátory:

- Adreno 320, 330 a 420 (Android), např. v SoC Snapdragon 600, 800 a 805 (výrobce Qualcomm),
- PowerVR Series6 a výše (iOS), např. v SoC Apple A7 (výrobce Imagination) [19],
- Mali série T6xx a výše (Android).

Nvidia ve svém posledním mobilním SoC Tegra 4 plnohodnotnou podporu verze 3.0 nemá, ale mnoho nových funkcí je podporováno v rámci rozšíření.

2.4 Výhled do budoucna

Vývoj na poli mobilní grafiky jde stále velmi rychle dopředu. Již za dobu psaní této práce bylo představeno několik dalších novinek.

2.4.1 OpenGL ES 3.1

17. března 2014 již byla uvedena nová verze OpenGL ES s číslovkou 3.1 (dříve označení Next). Specifikace obsahuje novinky jako výpočetní (*compute*) shadery (včetně *atomics* a *image load/store*), oddělené shader objekty, nepřímé vykreslovací (*indirect-draw*) příkazy, dále *multisample* a *stencil* textury a další rozšíření jazyka GLSL ES. Toto vše za zpětné kompatibility s OpenGL ES 2.0 a 3.0. API ale stále nebude podporovat geometry shadery a teselaci [17].

2.4.2 Tegra K1

Tegra K1 (dříve Project Logan), představena na CES 2014 firmou Nvidia, přinese později v roce 2014 plnohodnotné verze DirectX 11 a OpenGL 4.4 na mobilní zařízení. To znamená i podporu teselace či rozhraní CUDA, a to vše za spotřeby nižší než 5 wattů [8].

Kapitola 3

Realistické zobrazování 3D scén v reálném čase

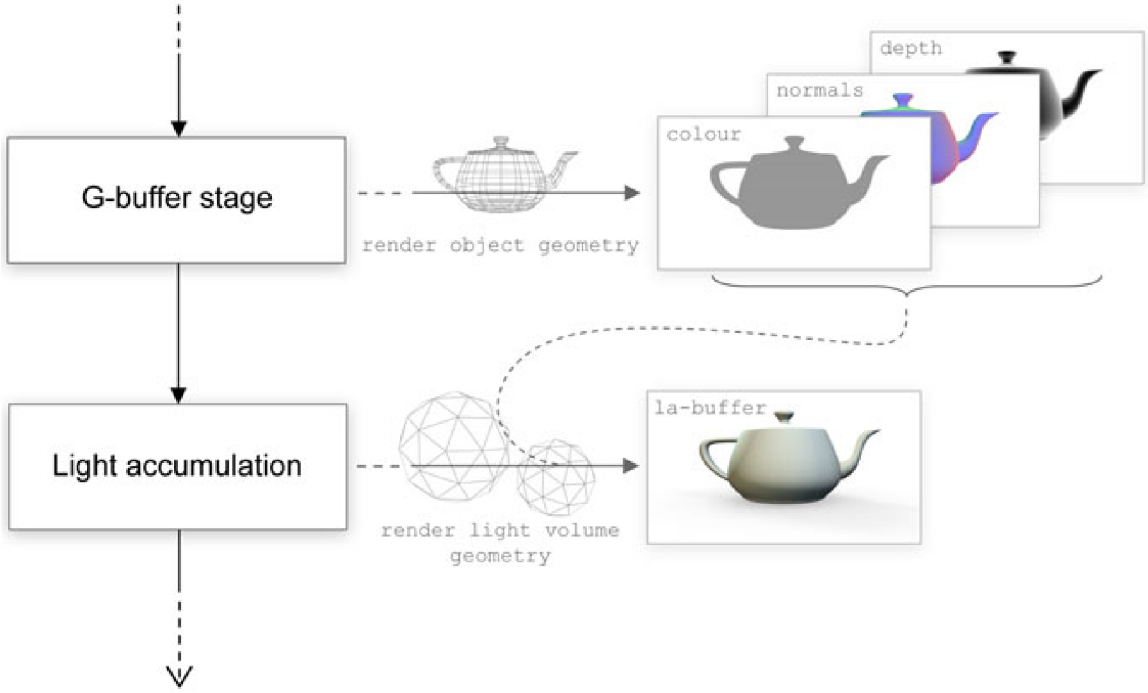
Pro realistické zobrazování 3D scén bylo postupem času vyvinuto mnoho metod a postupů, řešících různé aspekty tohoto problému. Většina metod ve svém základu vychází ze základních přístupů založených na simulaci fyzikálních vlastností světla a materiálů. Pro zobrazování v reálném čase je ale nutno přistoupit na mnoho kompromisů či využití empiricky odvozených metod, protože fyzikálně přesné simulace (např. metody založené na sledování paprsku) jsou stále velmi náročné na výkon.

V této kapitole jsou uvedeny vybrané pokročilejší metody řešení výpočtu osvětlení v reálném čase. Popis základních metod, lokálních/globálních osvětlovacích modelů aj. lze nalézt například v knize Real-Time Rendering [1]. V kapitole se předpokládá využití rasterizace.

3.1 Deferred shading

Deferred shading (lze přeložit jako *odložené stínování*) je tzv. *screen space* technika pro efektivní výpočet osvětlení ve scéně s mnoha světelnými zdroji. Výpočet osvětlení se provádí až po vykreslení všech objektů jako 2D post proces (pří vykreslování scény se ukládají potřebné informace do textur/bufferů za použití vícenásobných vykreslovacích cílů). Výpočet osvětlení (používá se tradičních lokálních osvětlovacích modelů) se tedy nemusí provádět pro každý objekt a světelný zdroj zvlášť jako u dopředného vykreslování, takže složitost výpočtu nestoupá s počtem světel ve scéně. Nevýhodou jsou naopak nároky na grafickou paměť (rostou s rozlišením), protože je nutno ukládat mnoho informací (normály, informace o materiálech apod.). Další nevýhodou je v základu nepodporovaný alfa blending, protože při vykreslení do textur jsou ztraceny potřebné informace [11].

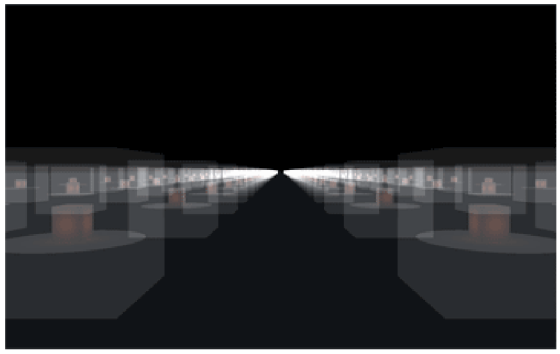
Schéma metody lze vidět na obrázku 3.1. V prvním průchodu je vykreslena geometrie scény a jsou ukládány hodnoty barvy (albedo), normál a hloubky do textur pomocí vícenásobných vykreslovacích cílů (tzv. geometry buffer, resp. G-buffer). V druhém průchodu je pro každé světlo vykresleno obalové těleso (aby se omezilo volání fragment shaderu pouze na požadovanou oblast), pro které je spočítáno osvětlení na základě lokálního osvětlovacího modelu pomocí informací přečtených z textur vygenerovaných v prvním průchodu. Výsledek je poté akumulován do výstupní textury (používá se alfa blending).



Obrázek 3.1: Schéma dvouprůchodového vykreslování metodou deferred shading[1].

Ukázku výsledné implementace lze vidět na obrázku 3.2a (2175 bodových světel). Obrázek 3.2b zachycuje stejnou scénu se zobrazenými obalovými tělesy světel.

(a) Scéna s 2175 bodovými světly.



(b) Stejná scéna se zobrazením obalového tělesa každého světla.

Obrázek 3.2: Ukázka osvětlení metodou deferred shading[2].

[1] http://www.john-chapman.net/content.php?id=13

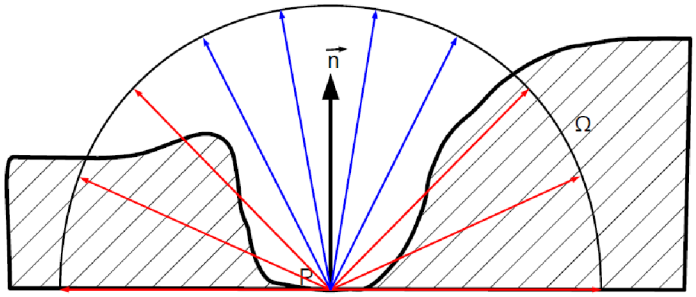
[2] http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/image-space-lighting-r2644

3.2 Screen space ambient occlusion

Ambient occlusion je aproximace globálního osvětlení umožňující simulaci jemných stínů, které lze vidět například v rozích místností či v úzkých místech mezi objekty. Obecně, čím více je bod na povrchu zakryt okolní geometrií, tím hůř k němu může pronikat světlo, a tedy o to více je zastíněn. Tato metoda může výrazně zlepšit vizuální realismus scény, a to s nižší výpočetní náročností než u plnohodnotného globálního osvětlení.

Základní idea metody je počítat pro každý bod povrchu faktor zastínění (*occlusion factor*) a ten zakomponovat do osvětlovacího modelu, obvykle úpravou ambientní složky osvětlení, a to způsobem, že čím je tento faktor zastínění větší, tím je intenzita světla v daném bodě nižší [4].

Výpočet faktoru zastínění ale může být pro aplikace v reálném čase drahý – typicky se provádí vysíláním mnoha paprsků do okolí bodu ve směru polokoule orientované podle normály a hledání průsečíků s okolní geometrií. Čím více paprsků okolní geometrii protne, tím více je bod zastíněn (viz obr. 3.3).



Obrázek 3.3: Princip metody Ambient occlusion.

Tento přístup lze implementovat metodou sledování paprsku, což je ale pro aplikace běžící v reálném čase příliš výpočetně náročné.

Varianta metody vhodná pro aplikace v reálném čase se nazývá *Screen space ambient occlusion* (SSAO). Tato metoda byla poprvé použita ve hře Crysis (2007). Základní princip spočívá ve využití hloubkové informace scény, uložené po jejím vykreslení, jako aproximaci její geometrie. Následný výpočet faktoru zastínění pak probíhá ve screen space. To znamená, že celý proces může být počítán kompletně na grafické kartě (ve fragment shaderu) a je nezávislý na složitosti scény (pouze na jejím rozlišení – podobně jako u metody deferred shading). Díky tomu lze tento algoritmus úspěšně provádět v reálném čase [4].

Místo vysílání paprsků do okolí ve směru polokoule metoda ve hře Crysis používala náhodné vzorky vrhané do paměti hloubky (depth buffer) v okolí koule okolo zpracovávaného bodu (viz obr. 3.4a). Pokud je hloubka vzorku nižší něž hloubka bodu, pak je vzorek uvnitř geometrie a je zvýšen faktor zastínění.

Kvalita takto získaného výsledku je úměrná počtu vrhaných vzorků. Stejně tak ale výpočetní náročnost algoritmu. Řešením je využít menší počet vzorků s následným rozmazáním výsledku (např. Gaussovým jádrem).

Protože je prováděno vrhání vzorků v okolí koule, tak i rovné, nezastíněné stěny budou vypadat šedě (viz obr. 3.5a), protože přibližně polovina vzorků vždy padne dovnitř okolní geometrie (viz obr. 3.4a).

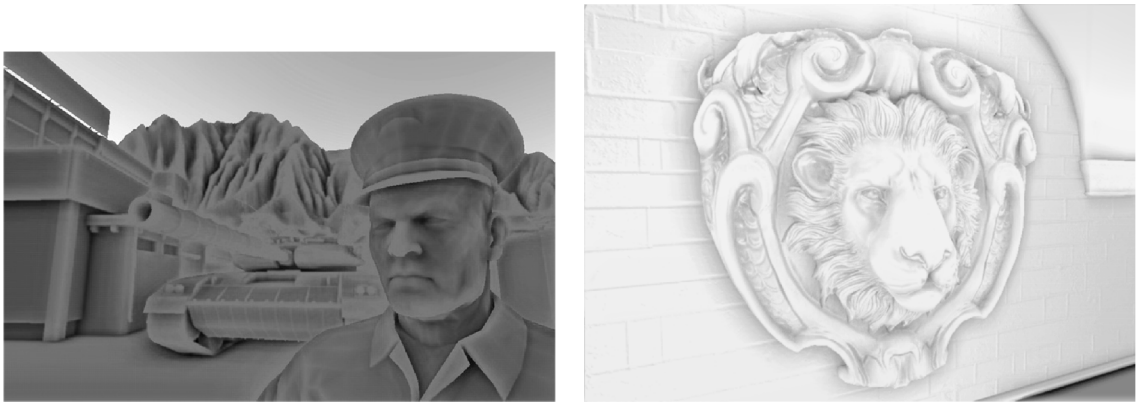
Lepší výsledky tedy dává vrhat body opět do okolí polokoule orientované podle normály, jako u původní metody (viz obr. 3.4b). Což znamená, že algoritmus musí mít přístup kromě

hloubky také k hodnotám normál v každém bodě. Při vykreslení scény je tedy nutno uložit do textur hodnoty hloubky a normál za použití vícenásobných vykreslovacích cílů, podobně jako u metody deferred shading [4]. Výsledek této metody lze pak vidět na obrázku 3.5b.

(a) Vzorky vrhané do okolí koule (Crysis).

(b) Vzorky vrhané do okolí polokoule orientované podle normály.

Obrázek 3.4: Rozdílné přístupy k okolí při vrhání vzorků v depth bufferu [4].



(a) Výsledek metody použité ve hře Crysis. Scéna trpí zabarvením do šeda [20].

(b) Výsledek metody s použitím polokoule orientované podle normály. Nezastíněná místa jsou správně bílá[3].

Obrázek 3.5: Ukázka výstupu metody SSAO před kombinací se scénou. Lze vidět jemné stíny v záhybech modelů.

3.3 Zobrazování s vysokým dynamickým rozsahem

Zobrazování s vysokým dynamickým rozsahem (*high dynamic range rendering*, zkratka HDRR či HDR rendering) umožňuje lepší vnímání reality přes vyšší dynamický rozsah – zachovává detaily, které mohly být ztraceny díky omezenému kontrastnímu poměru ve scéně (lze tedy například zobrazovat velmi světlá i tmavá místa ve scéně zároveň, a to při zachování všech detailů) [1].

HDR rendering pracuje s *floating-point* hodnotami na pixel (např. rozsah až 1 : 65 536 při použití 16-bit float, oproti tradičnímu rozsahu 256 : 1), aby bylo možno pracovat s vyšším kontrastním poměrem (je tedy nutno vykreslovat do floating-point textur, čímž rostou nároky na grafickou paměť).

[3] http://www.leadwerks.com/werkspace/gallery/image/890-ssao-shot-3

Takto vytvořený obraz je poté nutno převést zpátky do nižšího rozsahu (LDR), jelikož tradiční displeje takový rozsah neumí zobrazit. Tento převod lze dosáhnout pomocí *tone mappingu*.

3.3.1 Tone mapping

Existuje mnoho tone mapping operátorů, které dávají rozdílné výsledky. Můžou být jednoduché a efektivní pro využití v reálném čase, anebo i velmi komplexní, např. pro úpravu fotografií, kde délka výpočtu nehraje až tak roli. Operátory mohou obsahovat různé parametry, jako je například požadovaná hodnota expozice výsledného obrazu.

Obrázek 3.6 srovnává rozdíl mezi lineárním převodem do nižšího rozsahu a globálním tone mapping operátorem představeným Reinhard et al. [13]. Lze vidět, že s využitím operátoru jsou zachovány veškeré detaily ve scéně.

(a) Lineární převod do nižšího rozsahu. Informace v jasných místech jsou ztraceny.

(b) Použití tone mapping operátoru. Informace jsou zachovány.

Obrázek 3.6: Srovnání lineárního převodu do nižšího rozsahu a globálního tone mapping operátoru [13].

Jako ukázku velmi jednoduchého tone mapping operátoru lze uvést operátor 3.1, který mapuje z vyššího rozsahu $[0, \infty)$ na zobrazitelný rozsah $[0, 1)$.

$$L_d(x, y) = \frac{L(x, y)}{1 + L(x, y)} \tag{3.1}$$

Operátor má tendenci snížit hodnoty s vysokým jasem směrem k hodnotě 1, zatímco hodnoty s nízkým jasem nechává prakticky beze změny [1].

3.3.2 Bloom

Bloom (či také *glow*) efekt se obvykle používá spolu s HDR renderingem. Napodobuje artefakt reálných kamer, kdy se obraz velmi jasných (zářících) regionů rozpíjí do stran, což je způsobeno nedokonalostí jejich optiky.

Tento efekt se dá napodobit rozmazáním (jako výše u SSAO) světlých regionů (extrahovaných např. pomocí prahu) ještě před převodem do nižšího rozsahu a následným zkombinováním se scénou. Porovnání tradičního a HDR zobrazování spolu s bloom efektem lze vidět na obrázku 3.7.

Obrázek 3.7: Porovnání tradičního (vlevo) a HDR zobrazování ve hře Far Cry (2004). Na pravém obrázku lze vidět živější barvy při zachování všech detailů i v tmavých místech scény. Zároveň lze pozorovat bloom efekt vytvářený vstupujícím světlem. Scéna vypadá přirozeněji než její LDR varianta zachycena na levém obrázku[4].

3.4 Stínové mapy a stínová tělesa

Jelikož rasterizace při použití lokálních osvětlovacích modelů stíny přirozeně nepodporuje (oproti globálním metodám, jako je např. sledování paprsku), je třeba je řešit speciálními technikami. Základními přístupy k řešení stínů jsou metody stínových map a stínových těles.

3.4.1 Stínové mapy

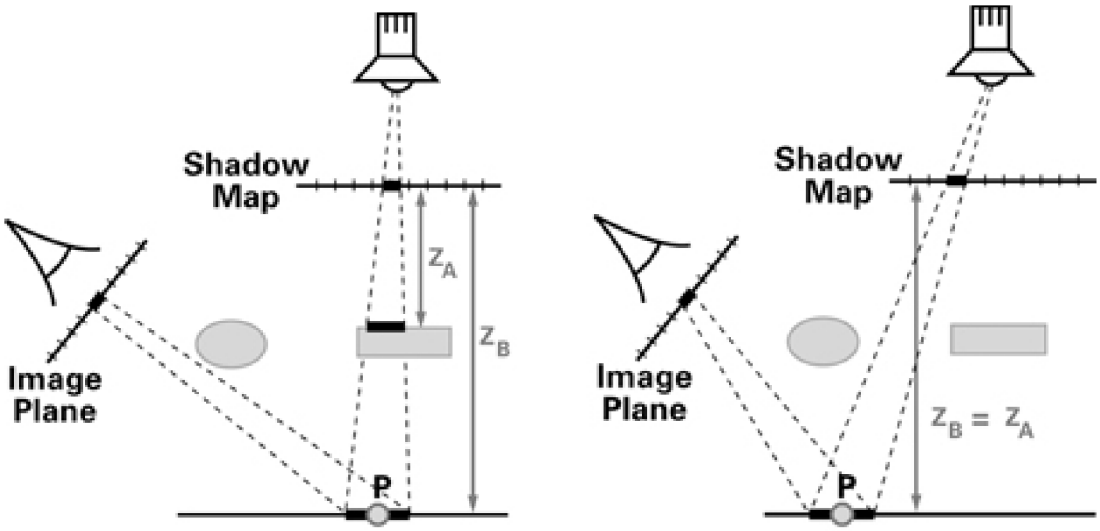
Stínové mapy (*shadow maps/mapping*) [21] je metoda vytváření dynamických stínů způsobených bodovými a směrovými světelnými zdroji, jejíž princip spočívá v testování pixelů, zda jsou viditelné z pohledu světla.

Scéna je nejprve vykreslena pro každé světlo z jeho pozice pohledu a informace jsou uloženy do hloubkové mapy (stínová mapa – obvykle textura), která reprezentuje vzdálenost světelného zdroje od plochy povrchu objektu ve scéně. Poté je scéna vykreslena z pohledu kamery, při čemž jsou porovnávány hodnoty hloubky ve stínové mapě a aktuálním fragmentu. Pokud je hloubka ve stínové mapě menší než hloubka fragmentu, fragment leží ve stínu (viz obr. 3.8).

Porovnávání hloubek lze provádět v hardware, a proto je tato metoda velmi rychlá. Výhodou je také podpora transparentních objektů a měkkých stínů. Nevýhodou je alias hran výsledných stínů související s omezeným rozlišením hloubkových map (textur) – je nutno filtrovat, například pomocí metody PCF (*percentage closer filtering*) [12] s následným bilineárním filtrem (viz obr. 3.9). Pro podporu všesměrových světel jsou nutné další optimalizace metody.

Metoda stínových map je pro své výhody velmi často používaná. Její nevýhody lze řešit mnoha různými modifikacemi [1].

[4] http://titan.cs.ukzn.ac.za/opengl/opengl-d7/2009notes/HighDynamicRangeLighting.pdf
[5] http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter09.html
[6] http://developer.nvidia.com/sites/default/files/akamai/gamedev/docs/2006-GDC-Variance-Shadow-Maps.pdf



(a) $Z_B > Z_A$, bod P je ve stínu.

(b) $Z_B = Z_A$, bod P je osvětlen.

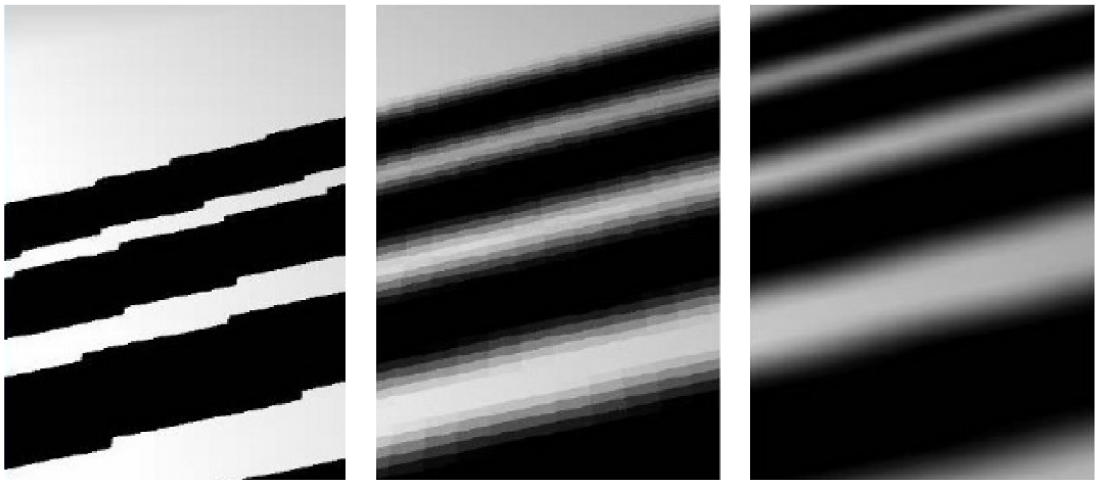
Obrázek 3.8: Schéma porovnávání u metody stínových map. Na levém obrázku je bod P ve stínu, protože hloubka bodu Z_B je větší než hloubka Z_A uložená ve stínové mapě. Oproti tomu na pravém obrázku má bod P stejnou hloubku jako hodnota ve stínové mapě, což znamená, že mezi bodem P a světelným zdrojem není žádný objekt, a tudíž bod není ve stínu[5].

3.4.2 Stínová tělesa

Stínová tělesa (*shadow volumes*) je metoda vytváření dynamických stínů, která rozděluje virtuální svět na dvě oblasti – ty, které jsou zastíněné pomocí stínové geometrie, a ty, které ne. Pro praktickou implementaci této metody se používá tzv. paměť šablony (*stencil buffer*) [1].

Stíny vytvořené touto metodou jsou ostré a sedí přesně na pixel. Metoda ale není moc rozšířená kvůli nárokům na výkon (průchod pro každé světlo, vytváření stínových těles – mění se v čase) a nemožnosti dosáhnout měkkých stínů. Nejznámější využití této metody bylo ve hře Doom 3 (obrázek 3.11).

[7] http://downloads.guru3d.com/Variance-Shadow-Maps-Demo-DX10-download-1628.html
[8] http://www.pcgameshardware.de/screenshots/1280x1024/2003/05/trites.jpg



(a) Původní stínová mapa.

(b) Aplikován PCF filtr o velikosti 5x5.

(c) PCF filtr 5x5 včetně bilineární filtrace. Výsledkem je měkký stín.

Obrázek 3.9: Ukázka aliasu výsledných stínů u metody stínových map a jeho možné řešení filtrováním metodou PCF[6].

(a) Při velmi nízkém rozlišení lze vidět alias stínové mapy, i když je použito filtrování.

(b) Stínová mapa bez aliasu při použití vyššího rozlišení.

Obrázek 3.10: Ukázka závislosti kvality stínových map na jejich rozlišení v praxi[7].

Obrázek 3.11: Ukázka ostrých stínů za použití stínových těles ve hře Doom 3 (2004)[8].

Kapitola 4

Návrh aplikace pro měření výkonu

Návrh aplikace pro měření výkonu se skládá nejprve z analýzy již existujících mobilních aplikací. Na základě této analýzy a předchozích dvou kapitol jsou poté navrženy vlastní testy. Následně je navržen systém pro měření výkonu a nakonec návrh databáze pro prezentaci výsledků měření a podrobných detailů o zařízeních uživatelům.

4.1 Existující aplikace

Aplikace pro měření výkonu mobilních zařízení lze rozdělit na komplexní benchmarky, testující výkon všech subsystémů najednou (CPU, GPU, RAM, I/O apod.), a na benchmarky zaměřené pouze na některé oblasti (např. pouze GPU, či výkon při specifických operacích – např. HTML5, GUI apod.). Dále existují technologická dema, které nejsou primárně určena pro měření výkonu, ale spíše ukazují technologické možnosti zařízení (především u 3D grafiky). Tato práce je zaměřena především na grafické benchmarky a některá technologická dema.

Testovací aplikace jsou často multiplatformní, kdy buď podporují více či všechny mobilní platformy, či navíc podporují i platformu PC.

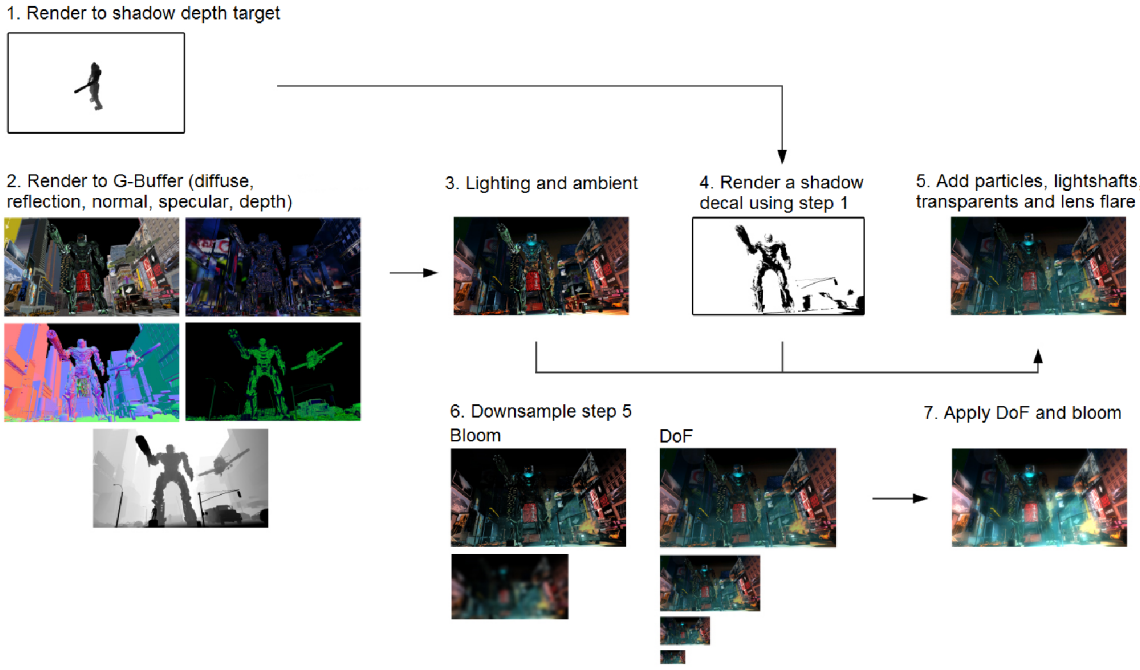
4.1.1 Grafické benchmarky

Grafické benchmarky jsou zaměřeny především na výkon v 3D grafice v reálném čase. Častou jsou zkombinovány také s výpočtem fyziky. Do existujících benchmarků se podpora OpenGL ES 3.0 postupně přidávala v posledním roce.

GFXBench (*Kishonti Informatics*[1]) je multiplatformní grafický benchmark, který uvedl podporu OpenGL ES 3.0 od verze 3.0 (23. ledna 2014) testem s názvem Manhattan. Tento test využívá vícenásobné vykreslovací cíle pro deferred shading s více jak šedesáti světly. Test dále obsahuje geometry instancing, odrazy, bloom a další efekty. Jeho vykreslovací řetězec je zachycen na obrázku 4.1. Jako ukazatel skóre využívá hodnotu počtu vykreslených snímků za daný reálný čas animace testu [2].

AnTuTu Benchmark (*AnTuTu Hong Kong*[2]) je komplexní benchmark pro Android, který je velmi rozšířený. Jeho součástí je samostatná aplikace pro testování pouze grafického výkonu – **AnTuTu 3D Rating Benchmark**, která existuje ve verzích pro OpenGL

[1] http://gfxbench.com
[2] http://antutu.com/Ranking.shtml



Obrázek 4.1: Vykreslovací řetězec testu Manhattan z benchmarku GFXBench 3.0 [2].

ES 2.0 i 3.0. Tato verze navíc, oproti základnímu benchmarku, podporuje i operační systémy iOS a Windows Phone.

Basemark X (*Rightware*[3]) je multiplatformní grafický benchmark. Podpora OpenGL ES 3.0 je dostupná od verze Basemark ES 3.0, která zatím ale nebyla zpřístupněna veřejnosti. Skládá se ze dvou testů, které jsou vykreslovány ve fixním časovém rámci (nezávislém na rozdílovém času), takže každý běh vykreslí vždy identickou sérii snímků. Skóre je počítáno na základě času potřebného pro vykreslení každého ze snímků.

3DMark (*Futuremark*[4]) je multiplatformní grafický a fyzikální benchmark s dlouhou historií na osobních počítačích, který na mobilních zařízeních zatím nepodporuje standard ES 3.0. Již však ale byla demonstrována upravená scéna 3DMark Cloud Gate, která byla doteď dostupná pouze na osobních počítačích (DirectX 11), běžící na akcelerátoru PowerVR Series6 Rogue společnosti Imagination s využitím OpenGL ES 3.0. Scéna využívá deferred lighting, HDR rendering s bloom filtrem, částicový systém pomocí transform feedback, stínové mapy za použití světelných těles a další [6]. Veřejnosti bude zpřístupněna v budoucnu na operačních systémech iOS a Android.

Mobile GPU Mark (*Silicon Studio*[5]) je multiplatformní (Android/iOS) grafický benchmark, který kromě základního benchmarku obsahuje také interaktivní demo mód. Podpora OpenGL ES 3.0 není součástí, je dostupná pouze jako technologické demo v rámci aplikace YEBIS OpenGL ES 3.0 Tech Demo[6].

[3] http://www.rightware.com/benchmarking-software/product-catalog
[4] http://www.futuremark.com/benchmarks/3dmark
[5] http://www.siliconstudio.co.jp/en/contents/benchmark
[6] http://play.google.com/store/apps/details?id=jp.co.siliconstudio.YEBISDemo

OpenGL ES 3.0 benchmark (*Maniac Software*[7]) – pouze pro úplnost, obsahuje jen jednu testovací scénu ze základních assetů z Unity (*RobotLab*). Tato aplikace jako jedna z prvních obsahovala podporu OpenGL ES 3.0 (existuje také totožná verze pro ES 2.0).

Většina aplikací obsahuje nějakou formu online databáze výsledků, ale v žádné nejsou obsaženy podrobné informace o vlastnostech OpenGL ES 3.0, např. o dostupných rozšířeních či implementačně závislých limitech.

4.1.2 Technologická dema

Technologická dema především ukazují technologické možnosti 3D grafiky na mobilních zařízeních. Nejsou primárně určena k měření výkonu, i když mohou také obsahovat jednoduché benchmarky, či o výkonu vypovídají plynulostí běhu.

Obrázek 4.2: Ukázka z technologického dema Unity: The Chase [3].

The Chase (*Unity Technologies*[8]) je technologické demo postavené na herním engine Unity, které bylo předvedeno spolu s představením nové verze Androidu 4.3 s podporou OpenGL ES 3.0 (obrázek 4.2). Předvedlo technologie jako *subsurface scattering*, *physically based shading* a další. Demo mělo být následně zveřejněno v obchodu Google Play, zatím se tak ale nestalo.

Epic Citadel (*Epic Games, Inc.*[9]) je multiplatformní interaktivní technologické demo postavené na Unreal Engine 3. Aplikace obsahuje i jednoduchý benchmark. Na mobilní

[7] http://play.google.com/store/apps/details?id=com.Maniac.UBenchEnhanced

[8] Unity: The Chase – Pushing the Limits of Modern Mobile GPU (http://www.realtimerendering.com/blog/stuff-from-siggraph-2013/).

[9] http://play.google.com/store/apps/details?id=com.epicgames.EpicCitadel

zařízení již byla přenesena i nová verze Unreal Engine 4 s podporou OpenGL ES 3.0, která byla demonstrována na telefonu Nexus 5 [9]. Demo ale zatím nebylo zpřístupněno veřejnosti.

4.2 Platforma

Jako hlavní platforma aplikace byl zvolen operační systém Android od verze 4.3 (SDK verze 18 a výše). Jelikož emulátor systému Android na PC nepodporuje OpenGL ES 3.0 a vývoj přímo na zařízení by byl zdlouhavý, další z požadavků je umožnit spuštění aplikace nativně přímo na vývojovém PC.

Toto je možné za pomocí využití nativního kódu v jazyce C++ pomocí *Android Native Development Kit* (NDK), který pro komunikaci s Java částí používá *Java Native Interface* (JNI). Jelikož OpenGL ES 3.0 je podmnožinou OpenGL 3.3 (viz kap. 2.2), je možné vytvořit na desktopu tento kontext (*core profile*) a použít na obou platformách stejný kód [7]. Různý kód bude pouze pro platformě závislé funkce, jako je vytváření okna (aktivity), do kterého se bude vykreslovat, OpenGL kontextu, správa vstupních zařízení pro testování apod., což se dá řešit podmíněným překladem (direktiva `ifdef`).

4.3 Testovací scéna

Cílem práce je vytvořit aplikaci měřící výkon za využití vybraných algoritmů realistického zobrazování 3D scén v reálném čase, jelikož toto využití je na mobilních zařízeních díky hrám jedno z nejčastějších. Aby byl průběh testování zajímavý i pro uživatele, je třeba výsledky vykreslování prezentovat ihned na displeji zařízení.

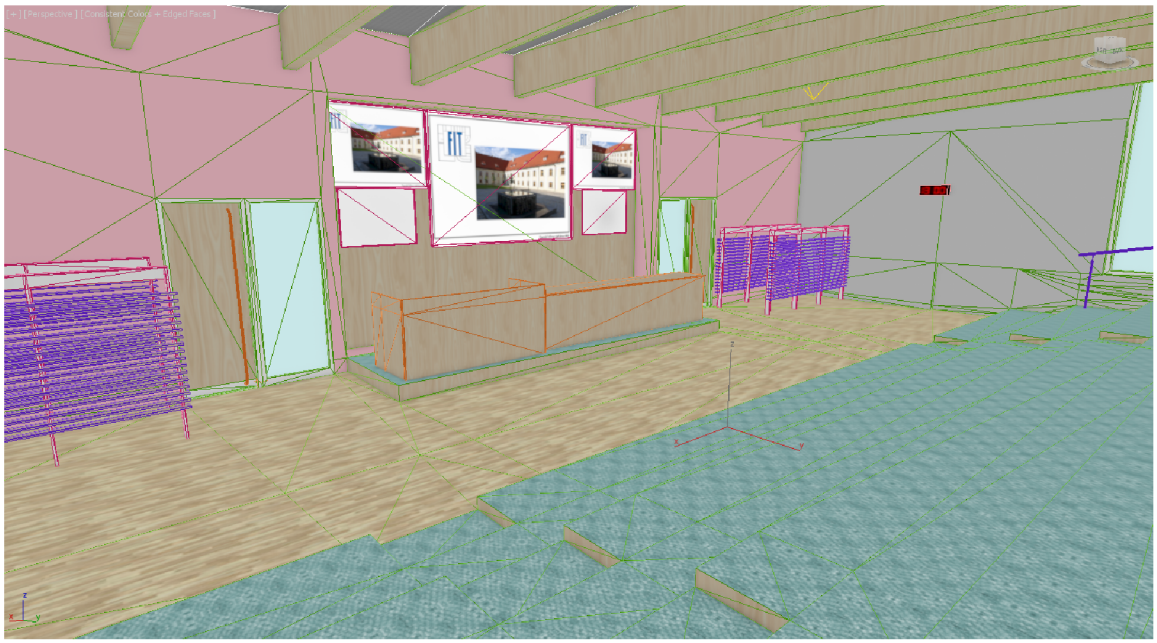
4.3.1 Výběr algoritmů

Základem benchmarku tedy bude jeden komplexní test s využitím nových funkcí (a s nimi souvisejících efektů) z OpenGL ES 3.0. Pro test byly s nejvyšší prioritou zvoleny algoritmy deferred shading a screen space ambient occlusion (viz kap. 3) díky hlavní novince v OpenGL ES 3.0, což jsou vícenásobné vykreslovací cíle (viz kap. 2), které jsou pro jejich implementaci nezbytné. Poté geometry instancing a využití komprese textur. Tyto algoritmy testují jak schopnost vykreslovat velké množství geometrie (instancing), tak fillrate a prostupnost grafické paměti (screen space techniky). Další algoritmy budou zváženy až po implementaci na základě toho, jak na tom budou mobilní zařízení s výkonem. Bylo by pak přidáno využití stínů (metoda stínových map), HDR rendering s bloom efektem, a případně částicové efekty pomocí metody transform feedback.

4.3.2 Modely a textury

Jako testovací scéna bude využit model přednáškové učebny (obrázek 4.3), u které se využití výše zmíněného instancingu nabízí pro židle (obr. 4.4) a stoly (obr. 4.5)[10]. Modely budou vyexportovány do formátu Autodesk Collada (.dae) a načteny v aplikaci pomocí vhodné knihovny, spolu s texturami komprimovanými ETC2 kompresí do formátu Khronos Texture (KTX).

[10] Za poskytnutí modelů děkuji bratrům Václavu a Davidu Šabatovým.



Obrázek 4.3: Testovací scéna zobrazená v aplikaci 3ds Max 2015 (bez stolů a židlí).

Test se bude skládat z keyframe animace průletu kamery po učebně, kdy budou zabírány implementované efekty. Související systém měření výkonu je probrán v následující části.

4.4 Systém měření výkonu

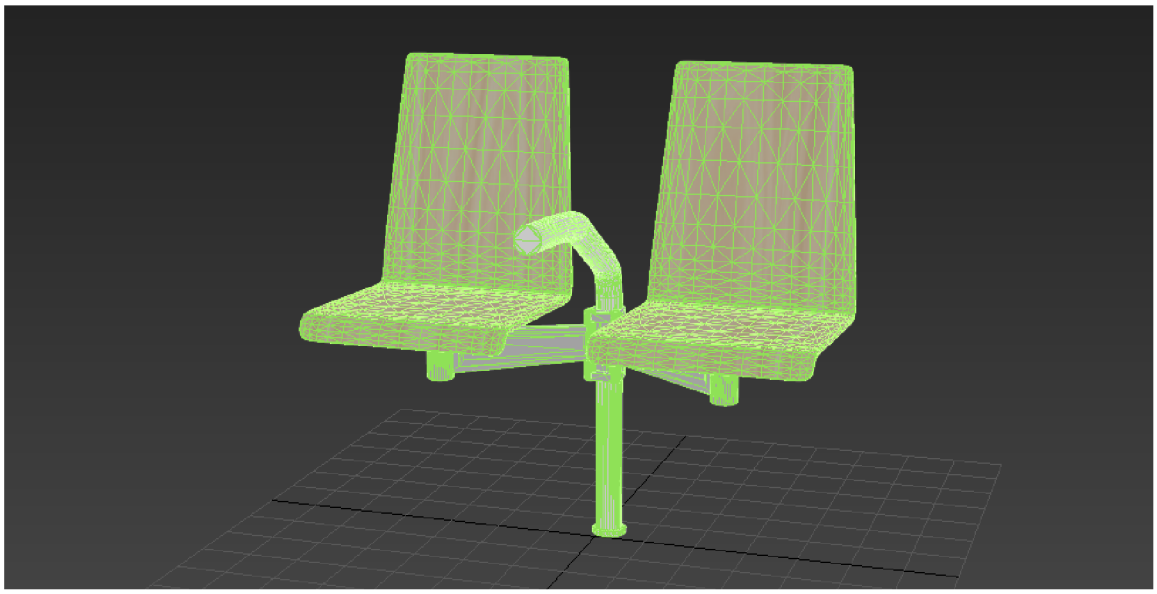
Jako měřítko výkonu je možno použít dvě veličiny – počet snímků za sekundu (*frames per second*, FPS), či čas mezi vykreslením po sobě jdoucích snímků (rozdílový čas). Tyto veličiny jsou na sobě závislé: FPS lze vypočíst jako $\frac{1}{x}$, kde x je rozdílový čas mezi snímky v sekundách. Pro vývojáře je lepší ukazatel výkonu rozdílový čas, protože je lineární povahy, oproti nelineárně se měnící hodnotě FPS.

Z těchto hodnot lze po zobrazení animace v testovací scéně (tzn. po průletu kamery) vypočíst průměr, který bude udávat, jak moc plynule aplikace běžela (je ale nutno brát v úvahu odchylky od průměru, protože časté větší odchylky nejsou žádoucí – buď jednotlivě, anebo např. pomocí hodnoty směrodatné odchylky). Čím vyšší FPS a nižší rozdílový čas mezi snímky, tím větší výkon. Z těchto hodnot lze pak vypočíst skóre, které bude udávat výkonnost zařízení a bude možné na jeho základě různá zařízení porovnávat.

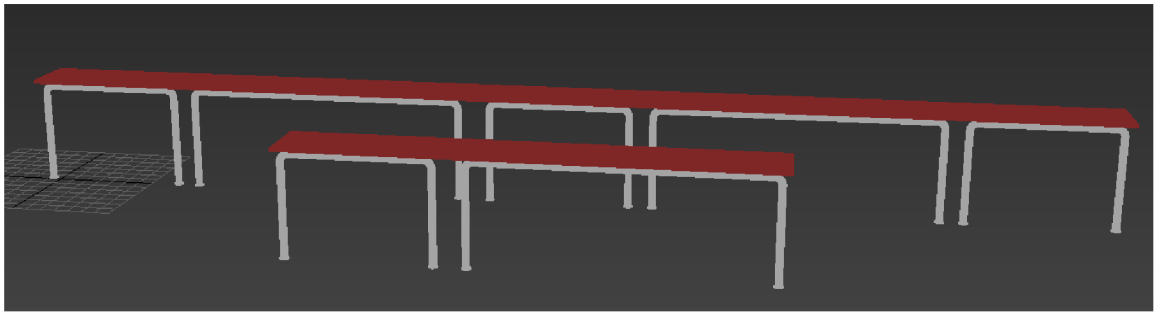
4.4.1 Nezávislost měření

Pro výpočet takového nezávislého skóre je ale nutno vzít v úvahu další vlivy – rozlišení vykreslované scény musí být na všech testovaných zařízeních stejné. Toto se dá řešit pomocí vykreslení mimo obrazovku (*offscreen rendering*) a následném up-scale/down-scale do nativního rozlišení displeje a zobrazení uživateli.

Dalším vlivem je vertikální synchronizace (*vsync*). Mobilní zařízení ji mají zpravidla zapnutou, tudíž je rychlost vykreslování omezena maximální obnovovací frekvencí displeje (obvykle 60 Hz, což odpovídá 60 FPS). Nejjednodušším řešením je vertikální synchronizaci softwarově vypnout, což zpravidla umožňuje OpenGL kontext.



Obrázek 4.4: Model židlí testovací scény zobrazen v aplikaci 3ds Max 2015.



Obrázek 4.5: Modely stolů testovací scény zobrazeny v aplikaci 3ds Max 2015.

Navrhovaná aplikace bude obsahovat keyframe animaci kamery, která bude probíhat ve fixním celkovém reálném čase. Jako základní ukazatel skóre bude počet vykreslených snímků za tento čas. Ostatní veličiny (průměrný počet snímků za sekundu a rozdílový čas, a směrodatné odchylky) budou uživateli prezentovány, ale nebudou ve skóre zohledňovány (podobně jako u aplikace GFXBench 3.0).

4.5 Uživatelské rozhraní

Uživatelské rozhraní bude implementováno pouze pro systém Android v jazyce Java. Aplikace se bude skládat z dvou aktivit – hlavní s uživatelským rozhraním, a benchmark nativní aktivity, která spustí benchmark z jazyka C++ a po skončení vrátí do hlavní aktivity jeho výsledky.

Hlavní aktivita bude obsahovat tři přepínatelné záložky implementované pomocí fragmentů (hrubý návrh na obrázku 4.6):

- záložka **Test** – možnost spustit benchmark, zobrazit jeho výsledky a tlačítko pro jejich upload do webové databáze (viz kap. 4.6),

- záložka **Informace o zařízení** (*Device info*) – detailní informace o zařízení,
- záložka **Pořadí** (*Ranking*) – pořadí z webové databáze.

Obrázek 4.6: Navrhované uživatelské rozhraní (zdroj: Eclipse ADT).

Záložky půjde přepínat buď dotknutím na jejich hlavičku s popisem, anebo přetažením prstu do strany (*swipe*).

4.6 Databáze výsledků

Pro možnost porovnávání výsledků jednotlivých zařízení je vhodné založit webovou službu, do které bude moci benchmark aplikace po měření odeslat výsledné skóre spolu s detailními informacemi o zařízení, včetně podrobných detailů o OpenGL (zejména implementačně závislé limity), jelikož ostatní benchmarky ve svých databázích tyto pokročilé informace nenabízí, a pro grafické vývojáře by taková databáze mohla být užitečná.

4.6.1 Webová služba

Jelikož je v plánu, aby byla aplikace vždy zdarma, tak také webový hosting musí být zdarma. Jako vhodný kandidát se jeví *Google App Engine*, který je zdarma do určité měsíční kvóty[11], která by pro tuto aplikaci měla dostačovat. Služby se pro něj dají programovat v jazycích Python, Java, PHP a Go. Adresa výsledné aplikace je poté ve formátu `http://<name>.appspot.com`, kde `<name>` je název aplikace. Data lze ukládat do NoSQL databáze *Datastore* a výsledky z Java aplikace na Androidu lze odesílat ve formátu JSON, jelikož jazyky nativně obsahují knihovny pro jeho zpracování.

4.6.2 Požadované vlastnosti

Po přístupu na stránku se zobrazí žebříček s body seřazený od nejvyššího skóre spolu s informacemi o modelu zařízení a názvu grafického akcelerátoru. Každá položka půjde rozkliknout pro zobrazení detailnějších informací:

- základní informace (jméno uživatele, datum a čas, model zařízení, název akcelerátoru),
- výsledky benchmarku (skóre, průměrný, nejlepší, nejhorší čas a počet snímků za sekundu, spolu se směrodatnými odchylkami),

[11] http://cloud.google.com/products/app-engine

- základní informace o OpenGL včetně informací o kontextu,
- informace o displeji, procesoru a paměti zařízení,
- informace o hardware zařízení (výrobce, model, značka apod.),
- informace o operačním systému Android (verze, kernel apod.),
- implementačně závislé limity OpenGL spolu se seznamem dostupných rozšíření.

Na informace o zařízení se lze dotazovat přímo systému Android, informace o vykreslovacím subsystému pak poskytne přímo OpenGL (funkce `glGet*` s vhodnými konstantami) a OpenGL kontext implementovaný na Androidu knihovnou EGL.

Tento žebříček bude také zpřístupněn v mobilní aplikaci za pomocí vestavěné komponenty webového prohlížeče a využití responzivního web designu (včetně upravených barev).

Kapitola 5

Implementace a testování

Jak bylo zmíněno v návrhu, byl požadavek implementovat aplikaci na systém Android, a zároveň umožnit její spuštění kvůli rychlému vývoji na PC platformě Windows. Popis této implementace, spolu s popisem implementace webové databáze a testováním na mobilních zařízení, je obsažen v této kapitole.

5.1 Použité technologie

Pro vývoj na platformu Android bylo použito vývojové prostředí Eclipse spolu s Android Developer Tools (ADT)[1] a Android Native Development Kit (NDK)[2]. V budoucnu je možno převést projekt do nového vývojového prostření Android Studio[3], které je ale zatím stále v testovací fázi (*early access preview*), a proto nebylo pro projekt použito. Pro nativní část aplikace byl použit jazyk C++, a to jeho nejnovější standard C++11. Jádro aplikace bylo implementováno ve vývojovém prostředí Visual Studio 2013. Pro implementaci serverové části, běžící na Google App Engine, byl použit jazyk Python 2.7 ve vývojovém prostředí PyCharm[4]. Veškerý vývoj probíhal pod operačním systémem Windows 8.1.

5.1.1 Knihovny

Pro matematické operace byla použita knihovna *OpenGL Mathematics* (GLM)[5], která je implementována pouze v hlavičkových souborech, takže zakomponování do projektu je velmi jednoduché. Knihovna dále dodržuje konvence jazyka GLSL, což ji činí jednoduchou na naučení a používání.

Pro načítání scény z formátu Collada (.dae), exportovaného programem 3ds Max 2014, byla použita multiplatformní knihovna Assimp[6]. Knihovna dokáže načítat velké množství různých formátů, a to včetně světel, materiálů, animací apod. Navíc umí při načítání modelů provést mnoho optimalizací (redukce počtu vrcholů apod.), což je vhodné pro aplikace běžící v reálném čase.

Pro kompresi textur do formátu ETC2 byl využit nástroj PVRTexTool z balíku PowerVR

[1] http://developer.android.com/tools/index.html
[2] http://developer.android.com/tools/sdk/ndk/index.html
[3] http://developer.android.com/sdk/installing/studio.html
[4] http://www.jetbrains.com/pycharm
[5] http://glm.g-truc.net
[6] http://assimp.sourceforge.net

SDK[7] společnosti Imagination. Jako formát souboru byl použit formát KTX[8] od Khronos Group, a to včetně knihovny libktx pro jeho načtení v aplikaci.

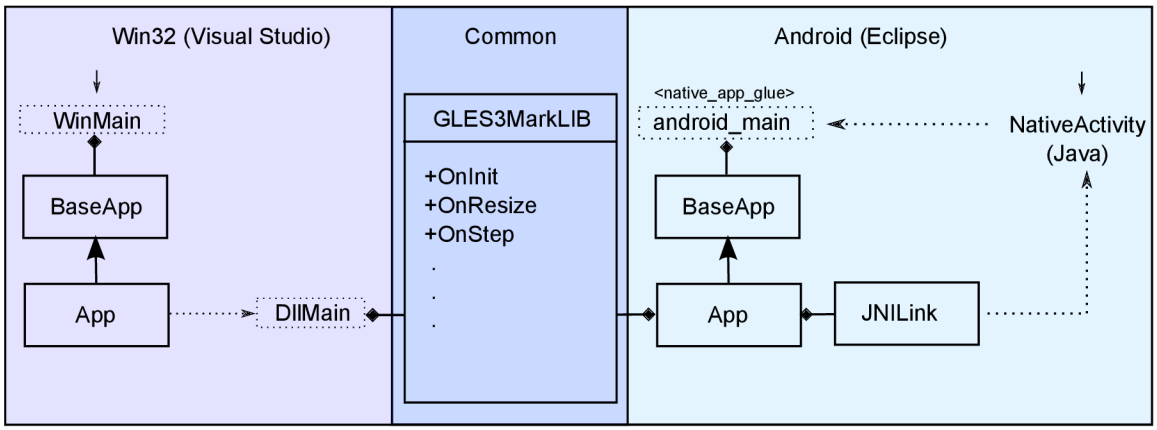
Pro vytvoření OpenGL ES kontextu na platformě Android byla využita knihovna EGL[9], která je součástí Android NDK. Pro vytvoření kontextu na platformě Windows byla použita knihovna OpenGL Extension Wrangler Library (GLEW)[10].

Serializace dat, která jsou předávána z nativní části aplikace do Java části pomocí JNI (informace o výsledku testu a podrobné informace o OpenGL), a které jsou později odesílány spolu s ostatními na server do databáze, je pro formát JSON prováděna knihovnou jsoncons[11].

Konečně, na straně serveru byla kromě základního frameworku webapp2 využita pro vytváření výsledných HTML dokumentů šablonovací knihovna Jinja2[12], která je přímo součástí Google App Engine.

5.2 Architektura aplikace

Architekturu aplikace, běžící na platformách Android a Win32 (desktop), zachycuje obrázek 5.1. Pro každou platformu je vytvořen zvláštní projekt, oba ale využívají společný C++ kód – Android část skrze NDK, Windows část přímo přes vlastní DLL rozhraní. Platformě závislé konstrukce, nacházející se ve společné části, jsou rozlišovány pomocí definovaných maker `ANDROID` a `_WIN32`.



Obrázek 5.1: Architektura aplikace pro platformy Win32 (desktop) a Android.

5.2.1 Platforma Android

Android část aplikace obsahuje uživatelské rozhraní v jazyce Java (viz kap. 5.2.3), z kterého je volána nativní část aplikace pomocí tzv. nativní aktivity – třída `BenchmarkActivity` dědící z vestavěné třídy `NativeActivity`.

[7] http://community.imgtec.com/developers/powervr/graphics-sdk
[8] http://www.khronos.org/opengles/sdk/tools/KTX
[9] http://www.khronos.org/egl
[10] http://glew.sourceforge.net
[11] http://sourceforge.net/projects/jsoncons
[12] http://jinja.pocoo.org

Nativní aktivita byla do systému Android přidána jako pomocná třída pro aplikace, které jsou kompletně v nativním kódu (zejména pro portování her apod.). Při jejím spuštění ze začne vykonávat nativní kód aplikace v jazyce C++. Pro správu vstupních událostí a životního cyklu aktivity obsahuje NDK jednoduchou C knihovnu `native_app_glue`[13], která běží ve vlastním vlákně (není tedy třeba řešit spouštění aplikace mimo GUI vlákno – při delších výpočtech nehrozí ukončení aplikace) a pro komunikaci s Java částí aplikace pomocí JNI obsahuje vlastní smyčku zpráv (*message loop*) včetně mechanismů na vybírání událostí ze vstupní fronty apod. Dědění z nativní aktivity je využito proto (normálně aplikace může běžet i bez jediného řádku Java kódu), aby bylo možno zpět do Java části aplikace předávat data (výsledky měření) s využitím JNI (zapouzdřuje třída `JNILink`).

Jako druhá možnost se nabízelo využít obyčejnou aktivitu s voláním vlastně vytvořených metod pomocí JNI (klíčové slovo `native` u deklarace metody) a využitím `GLSurfaceView`[14] pro správu OpenGL kontextu. Toto řešení nebylo použito kvůli nižší kontrole nad OpenGL kontextem (nemožnost vypnout vertikální synchronizaci apod.) a nutnosti vytvářet zbytečně mnoho Java kódu pro správu každé požadované události.

Pro vytvoření OpenGL kontextu tedy byla využita knihovna EGL na straně nativního kódu. Vytváření kontextu je zobecněno abstraktní bázovou třídou `RenderContext`, na jejímž základě je vytvářen také kontext na platformě Windows.

5.2.2 Platforma Windows

Na platformě Windows bylo pro vytvoření okna a správu vstupů implementováno vlastní rozhraní pomocí Win32 API. Jako jednodušší přístup by se mohla jevit multiplatformní knihovna Simple DirectMedia Layer (SDL)[15], ta ale nebyla použita kvůli větší kontrole nad životním cyklem aplikace a vytvářením OpenGL kontextu.

OpenGL od verze 3.0 přineslo ideu tzv. zastarávání funkčnosti (*deprecation*). Některé funkce mohou být takto označeny a poté v pozdějších verzích úplně odstraněny (např. vykreslování pomocí funkcí `glBegin`/`glEnd` bylo označeno deprecated ve verzi 3.0 a odstraněno ve verzi 3.1). Avšak mnoho implementací OpenGL zachovává tyto staré funkce kvůli zpětné přenositelnosti [7].

Od OpenGL verze 3.2 byl představen nový mechanismus, kdy si může uživatel při vytváření kontextu vybrat tzv. profil – buď zpětně kompatibilní (*compatibility profile*), anebo tzv. *core profile*, kdy je veškerá deprecated funkčnost odstraněna a API je tudíž čistší. Tento core profil ve verzi 3.3 je praktický identický s verzí OpenGL ES 3.0, a proto byl v tomto projektu na platformě Windows použit [7].

Pro vytvoření OpenGL 3.3 core profile kontextu bylo využito rozhraní WGL, které je součástí WinAPI a je velmi podobné již zmíněnému rozhraní EGL z platformy Android (a tudíž zobecnění vytváření kontextů třídou `RenderContext` nebylo obtížné). Dále byla využita knihovna GLEW pro správu OpenGL rozšíření.

Pro vytvoření konkrétní verze kontextu jsou nutná rozšíření `WGL_ARB_pixel_format`[16] a `WGL_ARB_create_context`[17]. Pro zpřístupnění těchto rozšíření je nutno nejprve vytvořit dočasné okno, inicializovat OpenGL pomocí základního kontextu[18], získat ukazatele na tyto

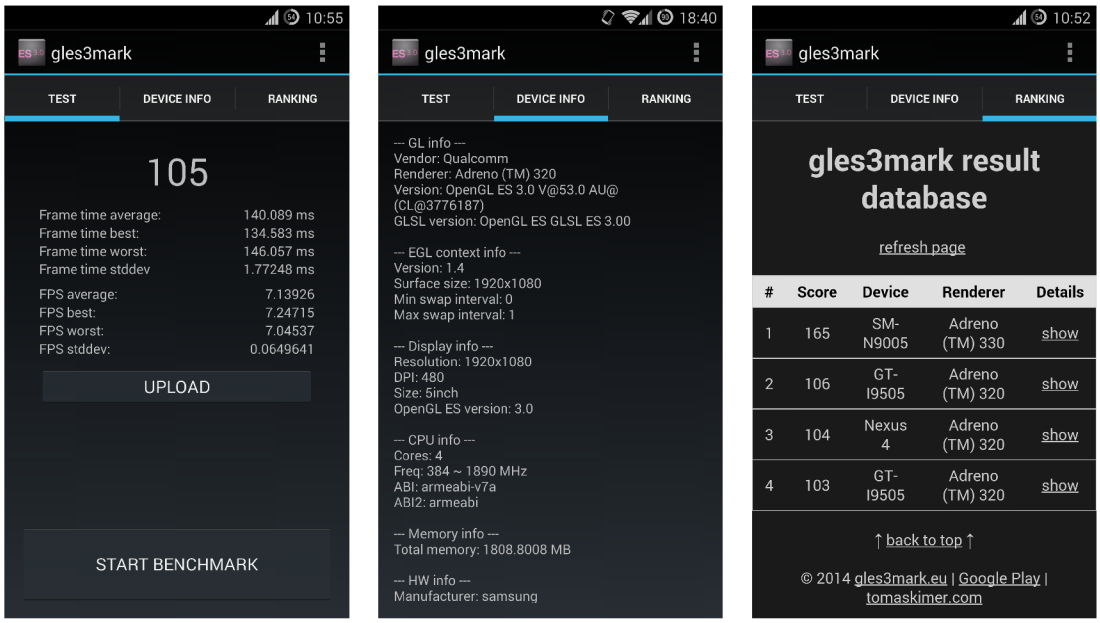
[13] http://developer.android.com/reference/android/app/NativeActivity.html
[14] http://developer.android.com/guide/topics/graphics/opengl.html
[15] http://www.libsdl.org
[16] http://www.opengl.org/registry/specs/ARB/wgl_pixel_format.txt
[17] http://www.opengl.org/registry/specs/ARB/wgl_create_context.txt
[18] http://www.opengl.org/wiki/Creating_an_OpenGL_Context_(WGL)

funkce (knihovna GLEW), poté dočasné okno zrušit (jak je jednou formát nastaven, nelze ho už změnit), a vytvořit kontext už s konkrétními požadovanými parametry znovu [7].

Pro doplnění, existuje ještě možnost vytvořit EGL kontext také přímo na platformě Windows pomocí emulace, a to např. pomocí balíků Qualcomm Adreno SDK, PowerVR Insider SDK, či ARM Mali OpenGL ES 3.0 Emulator [5].

5.2.3 Uživatelské rozhraní

Uživatelské rozhraní bylo implementováno pouze na systém Android v Java části aplikace, a to na základě šablony z Android ADT, kterou nabízí přímo Eclipse při vytváření nového projektu – byla zvolena prázdná aktivita s navigací typu *Action Bar Tabs (with ViewPager)*. Navigace využívá tři fragmenty, mezi kterými lze přepínat pomocí komponenty `ViewPager`. Výsledné rozhraní je zachyceno na obrázku 5.2.



(a) Záložka Test.

(b) Záložka Informace o zařízení.

(c) Záložka Pořadí.

Obrázek 5.2: Implementované uživatelské rozhraní.

První záložka (obr. 5.2a) obsahuje informace o výsledku testu (skóre velkým písmem) a tlačítka pro spuštění benchmarku (spustí výše zmíněnou nativní aktivitu) a odeslání výsledků do databáze (viz kap. 5.4). V druhé záložce (obr. 5.2b) jsou vypsány podrobné informace o zařízení. Pro zjišťování těchto informací byli využity vestavěné třídy `Build`[19] a `System`[20]. Poslední záložka (obr. 5.2c) obsahuje komponentu `WebView`, která zobrazuje webovou databázi výsledků.

5.2.4 Vývojářské rozhraní

Vývojářské rozhraní bylo vyvinuto pro obě platformy (pouze v C++ části aplikace) pro jednodušší testování při vývoji – v publikované verzi není obsaženo (není zahrnuto do

[19] http://developer.android.com/reference/android/os/Build.html
[20] http://developer.android.com/reference/java/lang/System.html

překladu při definování makra `ANDROID_PRODUCTION`).

Rozhraní se skládá s možnosti volně pohybovat kamerou po scéně (tzn. přepínat mezi animací a volným pohybem), předběžně ukončit benchmark s předáním výsledku, a na platformě Windows také různě pohybovat světly či vypisovat ladící informace na základě vstupu z klávesnice.

Společná nativní knihovna pro tyto případy dědí z rozhraní `IInputListener`, které poskytuje jednotlivé vstupní metody jak pro klávesnici, tak multidotykové ovládání včetně podpory myši. Tyto metody jsou pak volány z obou platforem – `native_app_glue` lze předat ukazatel na callback funkci zpracovávající vstupní události (`onInputEvent`).

5.3 Testovací scéna a měření výkonu

Testovací scéna byla implementována uvnitř třídy `Scene`. V inicializaci (metoda `OnInit`) jsou nejprve načteny všechny používané modely (každý mesh do samostatného VAO) a materiály včetně textur. Poté jsou načteny a přeloženy všechny shader programy.

Pro načítání souborů byly vytvořeny třídy `AssetFile` a `AssetManager`, protože na obou platformách funguje práce se soubory jinak. Ve Windows lze použít tradiční práce se streamy, avšak na Androidu je třeba využít speciálně poskytované funkce systémem, protože data jsou komprimována metodou zip do balíku apk. Uvedené třídy tuto rozdílnou implementaci odstiňují. Veškerá data jsou tedy nejprve načtena do paměti, a až poté předána knihovnám pro zpracování.

Po načtení dat je vytvořen jeden offscreen framebuffer, kterému jsou jako attachmenty přiřazeny textury o fixním HD rozlišení 1280x720 (G-buffer). Textury nejsou používány vždy všechny najednou – jejich použití je zapínáno podle potřeby funkcí `glDrawBuffers`. Tento přístup je levnější než připojování a odpojování textur v každém vykreslovaném snímku (nemusí se kontrolovat kompletnost framebufferu). Je garantováno, že OpenGL ES 3.0 podporuje vždy minimálně čtyři barevné attachmenty [5] (na tyto limity se lze dotazovat OpenGL, viz kap. 5.4). Použité formáty textur jsou znázorněny v tabulce 5.1.

Název	**Formát**	**Vnitřní formát**	**Datový typ**	**Attachment**
Depth	DEPTH_COMP	DEPTH_COMP24	U_INT	DEPTH_ATT
Albedo	RGBA	RGBA	U_BYTE	COLOR_ATT0
Normal	RGB	RGB16F	FLOAT	COLOR_ATT1
Final	RGBA	RGBA8	U_BYTE	COLOR_ATT2
SSAO	RED	RED	U_BYTE	COLOR_ATT3

Tabulka 5.1: Formáty textur v geometry bufferu (názvy konstant zkráceny).

Vykreslení scény probíhá v celkem čtyřech průchodech:

1. Vykreslení geometrie scény do textur pomocí MRT – uložení hloubky, albeda a normál ve *view-space* (shader `geometrypass`).

2. Výpočet SSAO termu na základě textur z předchozího bodu a uložení výsledku opět do textury. Dále uložení intenzity ambientního světla do textury, která bude použita jako vykreslovací cíl v následujícím kroku (shader `ssaopass`).

3. Výpočet odloženého osvětlení na základě textur z kroku 1 a zkombinování s SSAO termem a ambientním světlem z kroku 2, a uložení výsledku do finální textury (shader `lightpass`).

4. Zobrazení textury z předchozího kroku přes celou obrazovku (tzn. její případné roztažení/zmenšení do rozměrů okna obrazovky, pokud rozlišení textury neodpovídá rozměrům okna) (shader `screenquad`).

Detaily jsou popsány v následujících podkapitolách.

5.3.1 Využití ETC2 komprese textur

Jak už bylo zmíněno v kapitole 5.1.1, pro načítání komprimovaných textur byla použita knihovna libktx. Knihovna při načítaní zjistí vnitřní formát textury a automaticky provede načtení do OpenGL (zavolá mimo jiné funkci `glTexImage*` či `glCompressedTexImage*` se správnými parametry). Následně lze textuře nastavit filtrování, a to podle toho, zda již byla načtena včetně mipmap, anebo bez (výstupní parametr `isMipmapped`). Textury v aplikaci jsou načítány včetně mipmap vygenerovaných aplikaci PVRTexTool (formát KTX podporuje jejich ukládání). Knihovna také dokáže textury softwarově dekomprimovat, pokud OpenGL kontext komprimovaný formát nepodporuje. Toto ale není v aplikaci použito, jelikož použitý ETC2 formát je v OpenGL ES 3.0 standardizován.

5.3.2 Geometry instancing

V testovací scéně, která znázorňuje přednáškovou učebnu, je mnoho stejných objektů – židlí a stolů. Aby tyto objekty nemusely v mnoha kopiích zabírat grafickou paměť (kdyby byly přímo součástí načítané scény), je možno zvlášť načíst vždy jen jednu jejich kopii a následně je v cyklu vykreslovat na daných pozicích.

Tento přístup je ale neefektivní, jelikož stoupá počet vykreslovacích příkazů, které jsou drahé. Proto je pro tyto objekty využit geometry instancing, který dokáže mnoho stejných objektů vykreslit vždy pouze jediným vykreslovacím příkazem s různými parametry, jako je transformační matice či např. barva [5].

Vykreslení je provedeno příkazem `glDrawElementsInstanced`, který oproti původní funkci `glDrawElements` bere navíc jako poslední parametr počet instancí pro vykreslení.

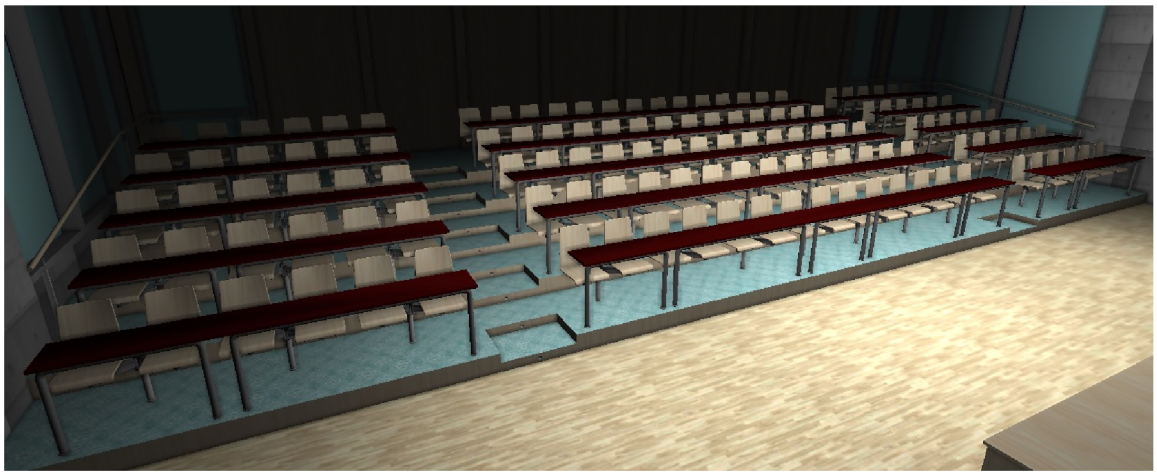
Přístup k instančním datům lze ve vertex shaderu provádět dvěma způsoby:

- použití vestavěné proměnné `gl_InstanceID` jako index do bufferu (např. do pole uniformních proměnných),

- uložení dat do VBO a zavolání funkce `glVertexAttribDivisor`, pomocí které lze nastavit, aby se data četla vždy pro každou instanci zvlášť, a ne klasicky pro každý vrchol vykreslovaného objektu.

V aplikaci byl použit druhý přístup, kdy jsou transformační matice židlí a stolů nejprve vytvořeny statickou třídou `InstanceDataBuilder` a následně načteny do VBO metodou `InitInstanceData` třídy `ModelRenderer`. Vykreslování je poté prováděno metodou `Render Instanced` té samé třídy. Implementovaný instancing všech lavic a židlí v učebně zachycuje obrázek 5.3.

Při implementaci na systému Android nastaly problémy s předáváním transformační matice do vertex shaderu. Kód z obrázku 5.4 hlásil při linkování chybu *multiple attribute*



Obrázek 5.3: Výsledný instancing všech židlí a stolů ve scéně.

attempt to bind at same location a musel být nahrazen kódem z obrázku 5.5. Tato oprava ale činí zápis zbytečně pracný a nepřehledný. Chyba je pravděpodobně v ovladači akcelerátoru Adreno 320, jelikož podle standardu je původní zápis správný (kód byl zkontrolován oficiálním referenčním kompilátorem glslang[21]).

```
layout (location=3) in mat4 instanceModelMat;
```

Obrázek 5.4: Kód způsobující chybu při linkování shaderu na platformě Android.

```
layout (location=3) in vec4 instanceModelMat0;
layout (location=4) in vec4 instanceModelMat1;
layout (location=5) in vec4 instanceModelMat2;
layout (location=6) in vec4 instanceModelMat3;

void main()
{
    instanceModelMat = mat4(instanceModelMat0,
                            instanceModelMat1,
                            instanceModelMat2,
                            instanceModelMat3);
}
```

Obrázek 5.5: Ekvivalent kódu z obrázku 5.4, nyní již však linkování proběhne úspěšně.

5.3.3 Screen space ambient occlusion

Metoda SSAO byla implementována pomocí metody vzorků vrhaných polokoulí orientované podle normály (viz kap. 3.2). Byl využit přístup, u kterého není nutno rozmazávat výsledek

[21] http://www.khronos.org/opengles/sdk/tools/Reference-Compiler

průchody navíc [20].

Na začátku fragment shaderu s názvem `ssaopass` je nejprve získána hodnota normály z textury a je zrekonstruována view-space pozice fragmentu na základě jeho hloubky z depth textury, inverzní projekční matice a aktuální texturovací souřadnice (funkce `reconstruct Position` uvedená na obrázku 5.5 – pro rekonstrukci pozice z hloubky existuje mnoho různých metod [10]). Poté je proveden výpočet SSAO termu na této pozici a na základě normály funkcí `computeSSAO`.

```
vec3 reconstructPosition(in vec2 coord, in float depth, in mat4 invProj)
{
    vec4 viewPos = invProj * vec4(coord * 2.0 - 1.0,
                                  depth * 2.0 - 1.0,
                                  1.0);
    viewPos /= viewPos.w;
    return viewPos.xyz;
}
```

Obrázek 5.6: Rekonstrukce view-space pozice fragmentu na základě texturovací souřadnice, hloubky a inverzní projekční matice [20].

Funkce využívá celkem šestnáct náhodných vzorků (vektory `vec2` v rozsahu [-1, 1]) uložených v konstantním poli `poissonSamples` vygenerovaných metodou *Poisson Disk* (podobně jako u Monte Carlo metod) [20]. Na základě každého ze vzorků je vygenerován náhodný soused fragmentu v jeho okolí o zadaném poloměru (proměnná `radius`, která je závislá na rozlišení). Pro každý takový fragment je pak spočtena jeho view-space pozice (`samplePos`), na jejím základě vzdálenost od pozice originálního fragmentu (`sampleDist`) a poté úhel, který svírá normála originálního fragmentu s normalizovaným směrovým vektorem mezi aktuálním a sousedním fragmentem (`sampleDir`).

Na základě této vzdálenosti a úhlu je pak možné vypočíst finální hodnotu zastínění fragmentu. Plynulý přechod hodnoty zastínění na základě vzdálenosti od fragmentu zajistí funkce `smoothstep`. Úhel vypočtený pomocí skalárního součinu pak udává, zda bod leží v polokouli orientované podle normály, a pokud ano, tak jak moc je zastíněn.

Takto vypočtená hodnota zastínění fragmentu je poté akumulována v proměnné `occlusion`. Tato hodnota je nakonec vydělena celkovým počtem vzorků a odečtena od hodnoty 1, aby výsledný term vytvářel s větším zastíněním tmavší barvu. Term je poté uložen do jednokanálové textury `ssaoTex` (attachment 3) pro následné použití při výpočtu osvětlení. Také je zapsána do textury `finalTex` složka ambientního osvětlení, která je opět ovlivněna vypočteným SSAO termem. Výsledný SSAO term před kombinací se scénou lze vidět na obrázku 5.7.

V prvotní verzi implementace byla pro nastavování poloměru na základě rozlišení použita ve fragment shaderu funkce `textureSize(depthTex, 0)`, která má vrátit rozlišení textury (datový typ `ivec2`). Pokud ale bylo provedeno čtení z takto vrácené hodnoty, aplikace se na systému Android ukončila. Pravděpodobně opět chyba v ovladači akcelerátoru Adreno 320.

Obrázek 5.7: Výsledná implementace SSAO před kombinací se scénou. V levém dolním rohu je zobrazená původní scéna (albedo), v pravém view-space normály.

5.3.4 Deferred shading

Pro implementaci osvětlení metodou deferred shading bylo použito 120 bodových světel o malém poloměru (viz obr. 5.8) a tři světla o poloměru větším pro osvětlení přední části učebny (viz finální obrázek 5.9). Zároveň bylo pro přímo neosvětlené části scény využito ambientní osvětlení.

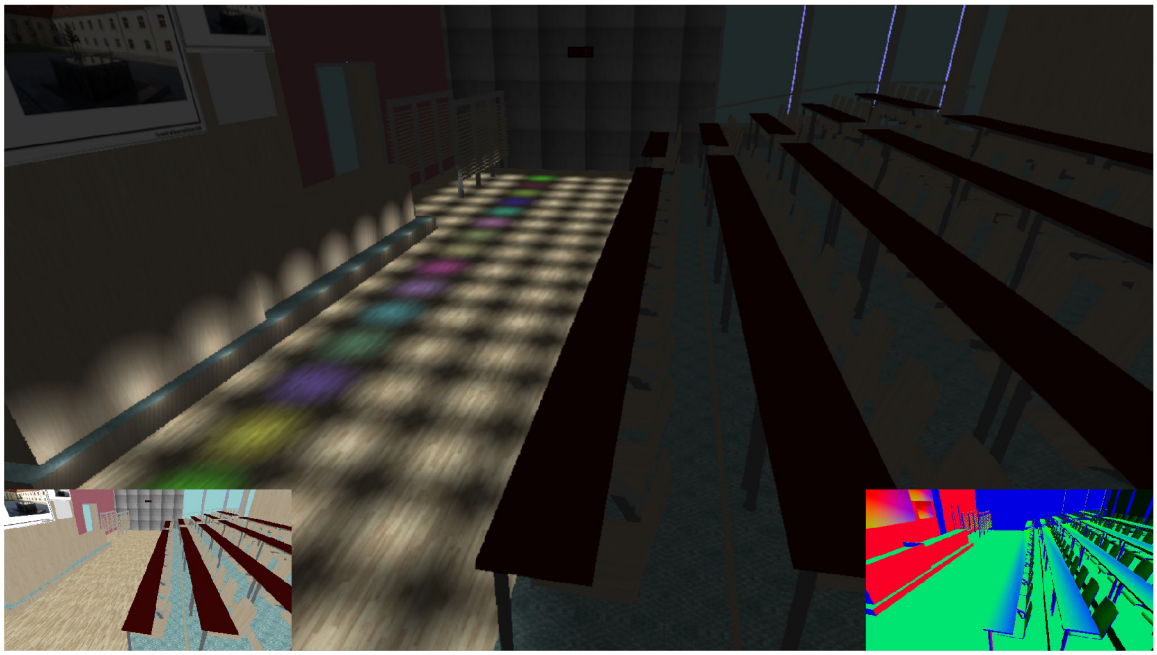
Při inicializaci scény jsou nejprve vytvořena všechna světla s konkrétními pozicemi, velikostí a barvou (třída `Light`), a jsou uloženy do pole. Následně je ještě inicializován mesh reprezentující hranice světla (tvar krychle).

Po vykreslení geometrie scény s uložením hloubky, normál a barvy do textur a po výpočtu SSAO termu následuje průchod pro výpočet osvětlení. Nejprve je vypnut zápis do paměti hloubky, z které je pouze čteno (metoda porovnávání `GL_GEQUAL`), poté je nastavena opačná strana redukování odvrácených polygonů (`glCullFace` s pamaterem `GL_FRONT`) než při vykreslování geometrie, aby se vykreslovalo i když se bude kamera nacházet uvnitř krychle reprezentující světlo. Následně je zapnut blending s funkcí $1 * src + 1 * dst$ (funkce `glBlendFunc` s oběma parametry hodnoty `GL_ONE`) a pro každé světlo je vykreslen mesh tvaru krychle shaderem `lightpass`.

V shaderu je nejprve rekonstruována pozice fragmentu z hloubky stejnou metodou, jako u výpočtu SSAO (viz kód 5.6) a poté je vypočteno osvětlení na aktuální pozici pomocí Lambertova osvětlovacího modelu (pouze difůzní světlo). Přírůstek vypočteného osvětlení je zkombinován s SSAO termem (kvůli názornějšímu zobrazení efektu, správně by osvětlená místa být zastíněna neměla) a zapsán do textury `finalTex` (attachment 2), do které už bylo při předchozím průchodu zapsáno ambientní osvětlení (barvy jsou přičítány blendingem).

Pro vykreslování modelů krychlí reprezentující bodová světla je vhodné využít také metodu geometry instancingu, aby se redukoval počet vykreslovacích příkazů.

Po tomto průchodu je již pouze zobrazena finální textura uživateli roztažená přes celou obrazovku.



Obrázek 5.8: Test implementace osvětlení metodou deferred shading spolu s ambientním světlem (120 bodových světel).

5.3.5 Systém měření

Při inicializaci je nejprve v OpenGL kontextu vypnuta vertikální synchronizace (funkce `eglSwapInterval` a Windows ekvivalent `wglSwapIntervalEXT`), aby nebyl maximální počet vykreslených snímků za sekundu limitován obnovovací frekvencí displeje.

Následně je scéna co nejvyšší možnou rychlostí vykreslována s aktuální pozicí kamery (metoda `OnStep`). Kamera je animována pomocí keyframe animace, a to za fixního celkového reálného času – dva stejné průlety kamery učebnou, výběr aktuálního klíčového snímku je prováděn na základě rozdílového času mezi aktuálním a předchozím snímkem (tzv. delta čas). Měření času je prováděno třídou `Time`, která využívá platformě nezávislou implementaci knihovny STL jazyka C++11 (`std::chrono`).

Jak už bylo zmíněno výše, vykreslování scény probíhá ve fixním rozlišení 1280x720, a až nakonec je výsledná textura roztažena do rozměrů okna. Tímto jsou zaručeny stejné podmínky pro každé zařízení, ať má jakékoli rozlišení displeje. Konečné roztažení textury do jakékoli velikosti okna už je totiž výkonově, při srovnání s předchozími výpočty, prakticky zanedbatelné.

V každém snímku jsou měřeny statistiky třídou `BenchmarkStatistics`:

- průměrný, nejnižší a nejvyšší rozdílový čas,
- průměrná, nejvyšší a nejnižší hodnota FPS,
- směrodatné odchylky hodnot rozdílového času a FPS,
- celkové skóre – počet vykreslených snímků za dobu animace.

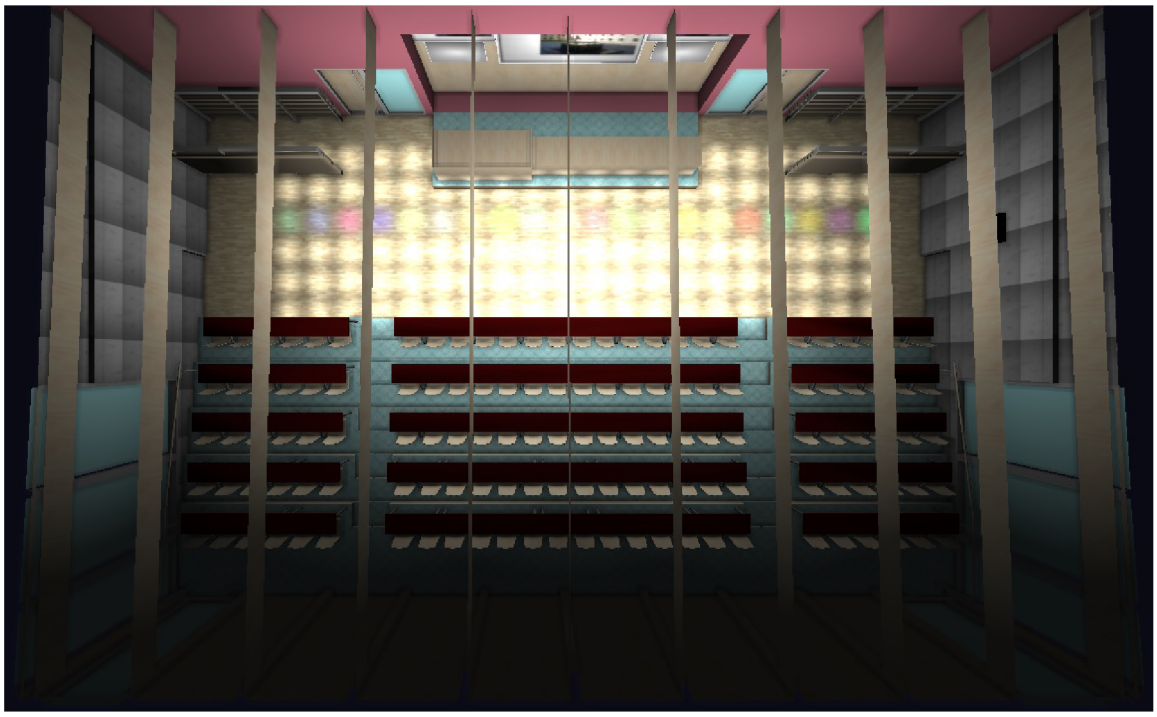
Měření i animace začne probíhat jednu sekundu od inicializace scény, aby se vyrušily různé výkyvy, které mohou nastat čerstvě po spuštění aplikace.

Při implementaci na systému Android nastalo několik problémů. Zejména při vypnutí vertikální synchronizace se stávalo, že se aplikace náhodně zasekávala, a to na dobu vždy kolem třinácti sekund, a po jejich uplynutí pokračovala dále. Debugováním bylo zjištěno, že aplikaci zablokuje vždy volání funkce `eglSwapBuffers`. Problém se přestal projevovat, když byla aplikace po, anebo před voláním této funkce uměle zpožděna o cca 40 ms (funkce `sleep`), což ale není vhodné řešení. Dalším řešením bylo v každém snímku volat funkci `glFinish`, která vynutí synchronizaci mezi CPU a GPU a zablokuje vykonávání programu, dokud všechny příkazy na straně OpenGL nejsou provedeny. Toto řešení také není optimální, jelikož snižuje rychlost aplikace. Nakonec bylo ale použito, protože žádné jiné nebylo nalezeno.

Dalším, a pravděpodobně souvisejícím, problémem bylo, že při nízkých FPS (cca 15 a níže) se začalo stávat, že vykreslované snímky zvláštně problikávaly (vždy se jakoby zobrazil na chvíli snímek předchozí, i když už byl zobrazený nový). Tento problém opět vyřešily výše zmíněné pauzy či volání `glFinish`. Jiné řešení se opět nepovedlo nalézt, a proto je na systému Android funkce `glFinish` volána vždy na konci každého snímku těsně před `eglSwapBuffers`.

Po implementaci všech výše zmíněných algoritmů a následným prvotním otestováním bylo zjištěno, že i když na PC platformě je čas snímku průměrně jen 4 ms (přes tři roky stará grafická karta střední třídy GeForce GTX 560 Ti), mobilní zařízení mají se scénou výkonové problémy (časy až kolem 150 ms). Proto už další algoritmy nebyly do scény přidávány. Finální testovací scéna je tedy znázorněna na obrázích 5.9 a 5.10.

Obrázek 5.9: Finální scéna.



Obrázek 5.10: Kompletní pohled na finální scénu při pohledu shora.

5.4 Databáze výsledků

Jak už bylo uvedeno výše, pro databázi výsledků byla využita služba Google App Engine za použití jazyka Python 2.7 na adrese `http://gles3mark.appspot.com`. Informace jsou na server odesílány ve formátu JSON. Tato data jsou složena z několika dílčích JSON objektů, které jsou získány na různých místech aplikace a před odesláním jsou spojeny do jednoho výsledného objektu, jehož úplná struktura je uvedena v příloze B.

5.4.1 Kolekce informací na straně klienta

Hlavní aktivita hned po startu aplikace získá data o zařízení pomocí tříd `Build` a `System`. Z dat je vytvořen objekt třídy `JSONObject` a data jsou zároveň prezentována uživateli v záložce Informace o zařízení.

Při řádném dokončení benchmarku je na straně C++ aplikace vytvořen JSON soubor (knihovna jsoncons) s výsledky testu, podrobnými informacemi o OpenGL (třída `GLQuery`) a informacemi o EGL kontextu. O zapsání všech těchto informací se stará třída `JSONStatsBuilder`, která nakonec vrátí výsledná data formátu JSON v datovém typu `std::string`. Tyto data jsou posléze poslány pomocí třídy `JNILink` do Java části aplikace těsně před ukončením nativní aktivity (metoda `FinishMe`, ve které je voláno `setResult` s přijatým řetězcem a nakonec metoda `finish` pro ukončení aktivity).

Java část aplikace (hlavní aktivita) tyto data přijme v metodě `onActivityResult`, rozparsuje je pomocí třídy JSONObject, zobrazí je uživateli v záložce Informace o zařízení (připojí je ke stávajícím datům o zařízení), a uloží si je pro možnost budoucího odeslání na server (tlačítko Upload se stane aktivním).

Po stisku tlačítka Upload je nejprve uživatel dotázán na jméno, které může volitelně

zadat. Po potvrzení dialogu tlačítkem OK je složen výsledný JSON objekt, který je následně odeslán na server asynchronní metodou `UploadTask`, která vytvoří HTTP požadavek na adresu serveru s hlavičkou `application/json`.

5.4.2 Strana serveru

Pro ukládání výsledků na straně serveru byla vytvořena NoSQL databáze Datastore. Strukturu jedné položky v databázi znázorňuje tabulka 5.2.

Název	Datový typ	Popis
score	Integer	hodnota skóre (řazeno sestupně)
device	String	typ zařízení
renderer	String	název GPU zařízení
uploader	String	jméno uživatele
date	DateTime	čas a datum přídání záznamu
benchInfo	Text (JSON)	výsledky benchmarku
glInfo	Text (JSON)	informace o OpenGL
glContextInfo	Text (JSON)	informace o OpenGL kontextu
deviceInfo	Text (JSON)	informace o zařízení

Tabulka 5.2: Struktura záznamu v Datastore databázi.

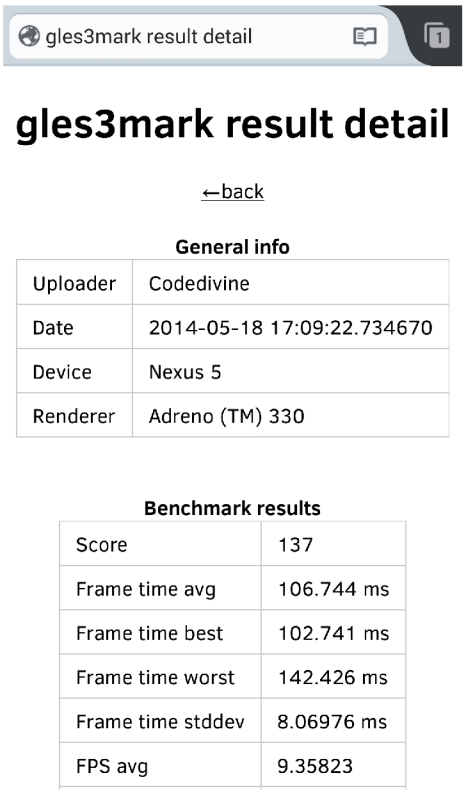
Webovou aplikaci založenou na frameworku webapp2 obsluhují dva hlavní tzv. *handlery*, které zpracovávající HTTP požadavky.

MainHandler spravuje přístup na hlavní stránku webové aplikace (cesta `/`). Při požadavku get načte z databáze položky a vykreslí stránku s jejich seznamem pomocí šablonovacího systému jinja2 (metoda `render_html` za použití šablony `main.html`). Při požadavku post je přijatý JSON string načtěn pomocí metody `json.loads` a jednotlivé položky uloženy do databáze (viz tab. 5.2). Pokud uložení proběhlo v pořádku, je zpět odeslána potvrzující zpráva.

DetailHandler zobrazuje detaily položek v databázi (cesta `/detail`). V parametru požadavku get je předán unikátní identifikátor položky uložené v databázi (je generován automaticky při jejím vkládání), na základě kterého jsou z databáze načteny detaily o této položce (metoda `get_by_id` a šablona `detail.html`). V budoucnu by bylo vhodné pro načítaní položek využit Memcache[22], aby se při větší navštěvnosti četnějšími přístupy do databáze zbytečně nevyčerpávala kvóta a služba mohla být stále zdarma[23].

Oba handlery při požadavku get používají css styl na základě identifikátoru prohlížeče (*user agent*) – Android aplikace nastavuje tuto hodnotu na `gles3mark_android_app`, která když je na straně serveru detekována, tak je použit css styl s černým pozadím a bílým písmem, aby stránka zapadala do tmavého designu aplikace. Srovnání lze vidět na obrázku 5.11.

[22] http://developers.google.com/appengine/docs/python/memcache/usingmemcache
[23] Viz http://www.jfgeyelin.com/2012/04/memcaching-right-stuff.html.



(a) Světlý styl pro zobrazení v tradičních prohlížečích.

(b) Tmavý styl pro zobrazení v aplikaci.

Obrázek 5.11: Rozdílné styly webové stránky aplikace.

5.5 Testování na mobilních zařízeních

Aplikace byla vyvíjena a průběžně testována na mobilním telefonu Samsung Galaxy S4 GT-I9505 (grafický akcelerátor Adreno 320) s operačním systémem Android 4.4.2. Veškeré problémy zmíněné v předchozích kapitolách tedy byly odhaleny na tomto zařízení.

Aplikace byla vydána v obchodu Google Play[24], kde byla stažena a nainstalována přibližně týden od vydání celkem šestnácti unikátními uživateli. Do databáze výsledků bylo odesláno celkem 11 záznamů. Pořadí zachycuje tabulka 5.3. Kompletní detail výsledku s nejvyšším skóre je uveden v příloze A.

Podle informací z vývojářské konzole Google Play aplikaci podporuje 400 zařízení z celkových 6091 (čísla průběžně rostou, jak jsou stále registrovány nové modely). Zjišťování podpory je na základě souboru `AndroidManifest.xml`, ve kterém je vyžadována podpora Androidu minimálně ve verzi 4.3 a podpora OpenGL ES verze 3.0.

Do aplikace byla implementována služba Google Analytics[25], na základě které bylo zjištěno, že uživatelé aplikace byli mimo České republiky také například ze Spojených států, Kanady, Španělska či Velké Británie. Průměrná doba strávená v aplikaci byla čtyři a půl minuty. Za veškeré testování aplikace nebyl zjištěn žádný pád či zastavení (ANR).

Podle databáze výsledků se průměrné časy na snímek pohybovaly od 88 ms do 150 ms,

[24] http://play.google.com/store/apps/details?id=com.tomaskimer.gles3mark
[25] http://www.google.com/analytics

Pořadí	Skóre	Zařízení	GPU
1	165	SM-N9005	Adreno (TM) 330
2	137	Nexus 5	Adreno (TM) 330
3	136	Nexus 5	Adreno (TM) 330
4	133	Nexus 5	Adreno (TM) 330
5	113	Nexus 7	Adreno (TM) 320
6	112	GT-I9505	Adreno (TM) 320
7	106	GT-I9505	Adreno (TM) 320
8	104	Nexus 4	Adreno (TM) 320
9	103	GT-I9505	Adreno (TM) 320
10	98	Nexus 7	Adreno (TM) 320
11	97	Nexus 7	Adreno (TM) 320

Tabulka 5.3: Pořadí v databázi výsledků.

což pro srovnání stále značně výkonově zaostává např. nad přes tři roky starou grafickou kartou střední třídy GeForce GTX 560 Ti z PC platformy (jak už bylo uvedeno v kapitole 5.3.5), kde byl čas na snímek v průměru 4 ms (3525 bodů v testu). Směrodatné odchylky měření byly také občas relativně velké (až 12 ms), na což by teoreticky mohla mít vliv funkce `glFinish` volaná v každém snímku (opět viz kap. 5.3.5), protože GPU pak musí čekat na synchronizaci s CPU, díky čemuž nemůže pracovat naplno a mohou tedy vznikat prostoje, protože GPU není na tento způsob práce optimalizováno [5].

Kapitola 6

Závěr

V této práci byla nejprve popsána nová verze grafické knihovny OpenGL ES 3.0, a to zejména její nově přidané funkce. Dále byly stručně popsány související algoritmy pro realistické zobrazování 3D scén v reálném čase. Následně byla navržena aplikace pro měření výkonu mobilních zařízení s podporou diskutované knihovny, a to za využití zejména jejich nových funkcí pro implementaci realistického zobrazování 3D scén v reálném čase.

Navržená aplikace byla implementována pro systém Android za využití zejména algoritmů deferred shading, screen space ambient occlusion, geometry instancing a komprese textur. Aplikace byla zveřejněna v obchodu Google Play pod názvem gles3mark[1]. Zároveň byla vytvořena webová databáze[2] sloužící jako nástroj pro porovnávání výsledků a detailních specifikací zařízení. Aplikaci je také možno spustit na PC platformě Windows pod OpenGL 3.3 core profile kontextem, avšak bez možnosti odesílat výsledky do databáze.

Aplikace byla otestována na několika zařízeních a do této chvíle bylo do databáze odesláno kolem jedné desítky výsledků (pět unikátních zařízení[3], pouze zástupci mobilních akcelerátorů Adreno 320 a Adreno 330). Výsledky byly v průměru od 88 ms do 150 ms na snímek (165 – 97 bodů v testu), což pro srovnání stále značně výkonově zaostává např. nad přes tři roky starou grafickou kartou střední třídy GeForce GTX 560 Ti z PC platformy, kde byl čas na snímek v průměru 4 ms (3525 bodů v testu). Při testování bylo také zjištěno, že nová funkčnost zatím ještě není v ovladačích implementována bezchybně, když nastaly problémy například u instancingu (viz kap. 5.3.2) či funkce `eglSwapBuffers` (kap. 5.3.5).

6.1 Možnosti budoucího vývoje

Možností pokračování projektu se nabízí celá řada. Zejména podpora dalších platforem, vytvoření atraktivnější testovací scény (i včetně animací), implementace více efektů, případně samostatných testů – částicový systém s využitím transform feedback, stínové mapy, HDR rendering, a mnoho dalších. Velkou výzvou budou optimalizace, aby aplikace na mobilních zařízeních běžela rychleji (např. pomocí Adreno profileru) a dovolila přidání zmíněných efektů. Dále stojí za zvážení podrobnější statistiky testování (např. graf znázorňující historii hodnot rozdílového času všech snímků celého testu, časy provádění jednotlivých efektů apod.).

[1] http://play.google.com/store/apps/details?id=com.tomaskimer.gles3mark
[2] http://gles3mark.appspot.com
[3] Google Nexus 4, 5 a 7 (2013), Samsung Galaxy S4 GT-I9505 a Samsung Galaxy Note 3 SM-N9005.

Literatura

[1] Akenine-Möller, T., Haines, E. a Hoffman, N. *Real-Time Rendering*. 3. vyd. Wellesley: A K Peters/CRC Press, 2008. 1045 s. ISBN 978-1-4398-6529-3.

[2] Black, D. *GFXBench 3.0: A Fresh Look At Mobile Benchmarking* [online]. February 21, 2014 [cit. 11. března 2014]. Dostupné na: <`http://www.tomshardware.com/reviews/gfxbench-3-graphics-performance,3743.html`>.

[3] Burke, D. a Cohen, G. *Introducing Android 4.3, a sweeter Jelly Bean* [online]. July 24, 2013 [cit. 17. prosince 2013]. Dostupné na: <`http://officialandroid.blogspot.cz/2013/07/introducing-android-43-sweeter-jelly.html`>.

[4] Chapman, J. *SSAO Tutorial* [online]. 15/01/2013 [cit. 11. dubna 2014]. Dostupné na: <`http://john-chapman-graphics.blogspot.cz/2013/01/ssao-tutorial.html`>.

[5] Ginsburg, D. a Purnomo, B. *OpenGL ES 3.0 Programming Guide*. 2. vyd. Boston: Addison-Wesley, 2014. 560 s. ISBN 978-0-321-93388-1.

[6] Imagination Technologies. *Imagination first to show new 3DMark OpenGL ES 3.0 Benchmark from Futuremark* [online]. 26 February, 2014 [cit. 14. března 2014]. Dostupné na: <`http://www.imgtec.com/News/detail.asp?ID=846`>.

[7] Kosarevsky, S. a Latypov, V. *Android NDK Game Development Cookbook*. Birmingham: Packt Publishing, 2013. 320 s. ISBN 978-1-78216-778-5.

[8] Kravitz, N. *The 192-Core Super Chip, Tegra K1, Brings Serious Gaming Chops to Mobile* [online]. January 6, 2014 [cit. 9. ledna 2014]. Dostupné na: <`http://blogs.nvidia.com/blog/2014/01/06/the-192-core-super-chip-tegra-k1-brings-serious-gaming-chops-to-mobile`>.

[9] Merrick, J. *Snapdragon processors behind superior graphics and mobile gaming technology at GDC* [online]. March 19, 2014 [cit. 5. května 2014]. Dostupné na: <`http://www.qualcomm.com/snapdragon/blog/2014/03/19/snapdragon-processors-behind-superior-graphics-and-mobile-gaming-technology`>.

[10] Pettineo, M. *Scintillating Snippets: Reconstructing Position From Depth* [online]. March 10, 2009 [cit. 5. března 2014]. Dostupné na: <`http://mynameismjp.wordpress.com/2009/03/10/reconstructing-position-from-depth`>.

[11] Phar, M. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Boston: Addison-Wesley, 2005. 814 s. ISBN 978-0-321-33559-3.

[12] Reeves, W. T., Salesin, D. H. a Cook, R. L. Rendering antialiased shadows with depth maps. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques,* SIGGRAPH '87. New York, NY, USA: ACM, 1987. S. 283–291.

[13] Reinhard, E., Stark, M., Shirley, P. et al. Photographic tone reproduction for digital images. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques,* SIGGRAPH '02. New York, NY, USA: ACM, 2002. S. 267–276.

[14] Strom, J. *ETC2-Package* [online]. August 2012 [cit. 11. ledna 2014]. Dostupné na: <`http://www.khronos.org/assets/uploads/developers/library/2012-siggraph-opengl-es-bof/Ericsson-ETC2-SIGGRAPH_Aug12.pdf`>.

[15] The Khronos Group. *OpenGL ES: The Standard for Embedded Accelerated 3D Graphics* [online]. [cit. 19. prosince 2013]. Dostupné na: <`http://www.khronos.org/opengles`>.

[16] The Khronos Group. *Khronos Releases OpenGL ES 3.0 Specification to Bring Mobile 3D Graphics to the Next Level* [online]. August 6th, 2012 [cit. 15. prosince 2013]. Dostupné na: <`https://www.khronos.org/news/press/khronos-releases-opengl-es-3.0-specification`>.

[17] The Khronos Group. *Khronos Releases OpenGL ES 3.1 Specification* [online]. March 17, 2014 [cit. 10. května 2014]. Dostupné na: <`https://www.khronos.org/news/press/khronos-releases-opengl-es-3.1-specification`>.

[18] Voica, A. *Introducing the brand new OpenGL ES 3.0* [online]. August 20th, 2012 [cit. 20. prosince 2013]. Dostupné na: <`http://withimagination.imgtec.com/index.php/news/introducing-the-brand-new-opengl-es-3-0`>.

[19] Voica, A. *PowerVR Series6 passes OpenGL ES 3.0 conformance with flying colours* [online]. February 20th, 2013 [cit. 20. prosince 2013]. Dostupné na: <`http://withimagination.imgtec.com/index.php/powervr/powervr-series6-cores-pass-opengl-es-3-0-conformance-with-flying-colours`>.

[20] Weinzierl-Heigl, C. *Hemispherical Screen-Space Ambient Occlusion (SSAO) for Deferred Renderers using OpenGL/GLSL* [online]. 2012-12-31 [cit. 10. dubna 2014]. Dostupné na: <`http://blog.evoserv.at/index.php/2012/12/hemispherical-screen-space-ambient-occlusion-ssao-for-deferred-renderers-using-openglglsl`>.

[21] Williams, L. Casting curved shadows on curved surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques,* SIGGRAPH '78. New York, NY, USA: ACM, 1978. S. 270–274.

Seznam příloh

Příloha A Detail výsledku z databáze.

Příloha B Formát dat odesílaných do databáze.

Příloha C Plakát ve formátu A2 prezentující výsledky práce.

Příloha D Datový nosič se zdrojovými kódy, binárními soubory a manuálem.

Příloha A

Detail výsledku z databáze

Uploader	maurean's note3
Date	2014-05-17 12:47:44
Device	SM-N9005
Renderer	Adreno (TM) 330

Tabulka A.1: Základní informace.

Score	165
Frame time avg	88.9836 ms
Frame time best	87.541 ms
Frame time worst	111.408 ms
Frame time stddev	2.85397 ms
FPS avg	11.2443
FPS best	11.3911
FPS worst	10.893
FPS stddev	0.157641

Tabulka A.2: Výsledky měření.

Vendor	Qualcomm
Renderer	Adreno (TM) 330
Version	OpenGL ES 3.0 V@66.0 AU@ (CL@)
GLSL version	OpenGL ES GLSL ES 3.00

Tabulka A.3: Informace o OpenGL.

Version	1.4
Surface size	1920x1080
Min swap interval	0
Max swap interval	1

Tabulka A.4: Informace o EGL kontextu.

Resolution	1920x1080
DPI	480
Size	5.7-inch

Tabulka A.5: Informace o displeji.

Cores	4
Frequency	300 2265 MHz
Instruction set (ABI)	armeabi-v7a
Instruction set 2 (ABI2)	armeabi
Total memory	2779.293 MB

Tabulka A.6: Informace o procesoru a paměti.

Manufacturer	samsung
Brand	samsung
Model	SM-N9005
Device	hlte
Hardware	qcom
Board	MSM8974
Product	hltexx

Tabulka A.7: Informace o hardware zařízení.

Android version	4.4.2
Android SDK version	19
OS	Linux 3.4.0-636608 armv7l
Bootloader	N9005XXUENC2
Java specification	Dalvik Core Library 0.9
VM specification	Dalvik Virtual Machine Specification 0.9
VM implementation	Dalvik 1.6.0
Build ID	KOT49H.N9005XXUENC2
Fingerprint	samsung/hltexx/hlte:4.4.2/KOT49H/N9005XXUENC2:user/ release-keys
Kernel	Linux version 3.4.0-636608 (dpi@SWDD5619) (gcc version 4.7 (GCC)) #1 SMP PREEMPT Tue Mar 4 13:42:20 KST 2014

Tabulka A.8: Informace o operačním systému zařízení.

ALIASED LINE WIDTH RANGE	1, 8
ALIASED POINT SIZE RANGE	1, 1023
MAX 3D TEXTURE SIZE	1024
MAX ARRAY TEXTURE LAYERS	256
MAX COLOR ATTACHMENTS	4
MAX COMBINED FRAGMENT UNIFORM COMPONENTS	197504
MAX COMBINED TEXTURE IMAGE UNITS	32
MAX COMBINED UNIFORM BLOCKS	24
MAX COMBINED VERTEX UNIFORM COMPONENTS	197632
MAX CUBE MAP TEXTURE SIZE	4096
MAX DRAW BUFFERS	4
MAX ELEMENT INDEX	2147483647
MAX FRAGMENT INPUT COMPONENTS	71
MAX FRAGMENT UNIFORM BLOCKS	12
MAX FRAGMENT UNIFORM COMPONENTS	896
MAX FRAGMENT UNIFORM VECTORS	224
MAX PROGRAM TEXEL OFFSET	7
MAX RENDERBUFFER SIZE	4096
MAX SAMPLES	4
MAX SERVER WAIT TIMEOUT	1000000000
MAX TEXTURE IMAGE UNITS	16
MAX TEXTURE LOD BIAS	15.9844
MAX TEXTURE SIZE	4096
MAX TRANSFORM FEEDBACK INTERLEAVED COMPONENTS	64
MAX TRANSFORM FEEDBACK SEPARATE ATTRIBS	4
MAX TRANSFORM FEEDBACK SEPARATE COMPONENTS	4
MAX UNIFORM BLOCK SIZE	65536
MAX UNIFORM BUFFER BINDINGS	24
MAX VARYING COMPONENTS	64
MAX VARYING VECTORS	16
MAX VERTEX ATTRIBS	16
MAX VERTEX OUTPUT COMPONENTS	69
MAX VERTEX TEXTURE IMAGE UNITS	16
MAX VERTEX UNIFORM BLOCKS	12
MAX VERTEX UNIFORM COMPONENTS	102
MAX VERTEX UNIFORM VECTORS	256
MAX VIEWPORT DIMS	4096x4096
MIN PROGRAM TEXEL OFFSET	-8
NUM COMPRESSED TEXTURE FORMATS	54
SAMPLES	0
SUBPIXEL BITS	4
UNIFORM BUFFER OFFSET ALIGNMENT	4

Tabulka A.9: Implementačně závislé limity OpenGL.

GL_AMD_compressed_ATC_texture	GL_OES_texture_float
GL_AMD_performance_monitor	GL_OES_texture_half_float
GL_AMD_program_binary_Z400	GL_OES_texture_half_float_linear
GL_EXT_debug_label	GL_OES_texture_npot
GL_EXT_debug_marker	GL_OES_vertex_half_float
GL_EXT_discard_framebuffer	GL_OES_vertex_type_10_10_10_2
GL_EXT_robustness	GL_OES_vertex_array_object
GL_EXT_texture_format_BGRA8888	GL_QCOM_alpha_test
GL_EXT_texture_type_2_10_10_10_REV	GL_QCOM_binning_control
GL_NV_fence	GL_QCOM_driver_control
GL_OES_compressed_ETC1_RGB8_texture	GL_QCOM_perfmon_global_mode
GL_OES_depth_texture	GL_QCOM_extended_get
GL_OES_depth24	GL_QCOM_extended_get2
GL_OES_EGL_image	GL_QCOM_tiled_rendering
GL_OES_EGL_image_external	GL_QCOM_writeonly_rendering
GL_OES_element_index_uint	GL_EXT_sRGB
GL_OES_fbo_render_mipmap	GL_EXT_sRGB_write_control
GL_OES_fragment_precision_high	GL_EXT_texture_sRGB_decode
GL_OES_get_program_binary	GL_EXT_texture_filter_anisotropic
GL_OES_packed_depth_stencil	GL_EXT_multisampled_render_to_texture
GL_OES_depth_texture_cube_map	GL_EXT_color_buffer_float
GL_OES_rgb8_rgba8	GL_EXT_color_buffer_half_float
GL_OES_standard_derivatives	GL_EXT_disjoint_timer_query
GL_OES_texture_3D	

Tabulka A.10: Dostupná OpenGL rozšíření.

Příloha B

Formát dat odesílaných do databáze

Data jsou odesílána na server ve formátu JSON, a to v následující struktuře[1] (konkrétní hodnoty jsou pro ilustraci převzaty z přílohy A):

```
{
    "Uploader": "maurean's note3",
    "BenchInfo": {
        "score": 165,
        "SPFavg": "88.9836",
        "SPFbest": "87.541",
        "SPFworst": "111.408",
        "SPFstddev": "2.85397",
        "FPSavg": "11.2443",
        "FPSbest": "11.3911",
        "FPSworst": "10.893",
        "FPSstddev": "0.157641"
    },
    "GLInfo": {
        "Vendor": "Qualcomm",
        "Renderer": "Adreno (TM) 330",
        "Version": "OpenGL ES 3.0 V@66.0 AU@ (CL@)",
        "GLSL version":"OpenGL ES GLSL ES 3.00",
        "Limits": {
            "ALIASED LINE WIDTH RANGE": "1, 8",
            ...
        },
        "Extensions": [
            "GL_AMD_compressed_ATC_texture",
            ...
        ]
    },
    "GLContextInfo": {
        "Version": "1.4",
```

[1] Položky `BenchInfo`, `GLInfo` a `GLContextInfo` jsou vytvářeny v C++ části aplikace, zbytek v Java části.

```
        "Surface size": "1920x1080",
        "Min swap interval": 0,
        "Max swap interval": 1
    },
    "DeviceInfo": {
        "Display": {
            "Resolution": "1920x1080",
            "DPI": 480,
            "Size": "5.7-inch",
            "OpenGL ES version": "3.0"
        },
        "CPU": {
            "Cores": 4,
            "Freq": "300 ~ 2265 MHz",
            "ABI": "armeabi-v7a",
            "ABI2": "armeabi"
        },
        "Mem": {
            "Total memory": "2779.293 MB"
        }
        "HW": {
            "Manufacturer": "samsung",
            "Brand": "samsung",
            "Model": "SM-N9005",
            "Device": "hlte",
            "Hardware": "qcom",
            "Board": "MSM8974",
            "Product": "hltexx",
            "Serial": "9e064fb5"
        },
        "OS": {
            "Android version": "4.4.2",
            "Android SDK version": 19,
            "Android version incremental": "N9005XXUENC2 REL",
            "OS": "Linux 3.4.0-636608 armv7l",
            "Bootloader": "N9005XXUENC2",
            "Java specification": "Dalvik Core Library 0.9",
            "VM specification": "Dalvik Virtual Machine Specification 0.9",
            "VM implementation": "Dalvik 1.6.0",
            "Build ID": "KOT49H.N9005XXUENC2",
            "Fingerprint": "samsung/hltexx/hlte:4.4.2/KOT49H/N9005XXUENC2: ...",
            "Kernel": "Linux version 3.4.0-636608 (dpi@SWDD5619) ...",
            "Id": "KOT49H",
            "Type": "user",
            "Tags": "release-keys"
        }
    }
}
```