

# Na tomto místě bude oficiální zadání vaší práce

- Toto zadání je podepsané děkanem a vedoucím katedry,
- musíte si ho vyzvednout na studijním oddělení Katedry počítačů na Karlově náměstí,
- v jedné odevzdáné práci bude originál tohoto zadání (originál zůstává po obhajobě na katedře),
- ve druhé bude na stejném místě neověřená kopie tohoto dokumentu (tato se vám vrátí po obhajobě).

[3] [5] [1] [2]



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



Diplomová práce

**Efektivní zobrazování rozsáhlých nočních scén na mobilních zařízeních**

*Bc. Luboš Vonásek*

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

Studijní program: Otevřená informatika, Navazující magisterský

Obor: Počítačová grafika a interakce

6. dubna 2014



## Poděkování

Rád bych poděkoval panu Jiřímu Bittnerovi za vedení této diplomové práce, za obrovské množství užitečných rad a za připomínky, které mi pomohly zvýšit kvalitu práce. Dále bych rád poděkoval učitelům z katedry počítačové grafiky a interakce za kvalitní výuku a za mnoho předaných znalostí, které jsem v této práci využil.



## Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 12. 5. 2014

.....



# Abstract

TODO Translation of Czech abstract into English.

# Abstrakt

TODO Abstrakt práce by měl velmi stručně vystihovat její podstatu. Tedy čím se práce zabývá a co je jejím výsledkem/přínosem.  
Očekávají se cca 1 – 2 odstavce, maximálně půl stránky.



# Obsah

<b>1</b>	<b>Úvod</b>	<b>1</b>
<b>2</b>	<b>Specifikace práce</b>	<b>3</b>
2.1	Požadavky na funkčnost . . . . .	3
2.1.1	Mapy osvětlení . . . . .	4
2.1.2	Statické osvětlení s dynamickými objekty ve scéně . . . . .	5
2.2	Cílová platforma Android . . . . .	7
2.3	Existující implementace . . . . .	8
2.3.1	Projekty zabývající se vykreslování noční scény . . . . .	8
2.3.1.1	Proceduralní modelování města a osvětlení nočního města . .	8
2.3.1.2	Imperfektní stínové mapy pro výpočet nepřímého osvětlení .	9
2.3.2	Závodní simulátory s noční scénou pro platformu Android . . . . .	9
2.3.2.1	Asphalt Urban od Gameloftu . . . . .	9
2.3.2.2	Need for Speed Most Wanted od EA Games a Firemonkeys .	10
2.3.2.3	GT Racing 2 od Gameloftu . . . . .	10
2.3.2.4	Sports Car Challenge od Fishlabs . . . . .	10
2.3.2.5	Track Racing od vývojáře Soulkey . . . . .	10
<b>3</b>	<b>Teoretická část</b>	<b>13</b>
3.1	Osvětlovací model . . . . .	13
3.2	Výpočet osvětlení . . . . .	15
3.3	Stínování . . . . .	16
3.4	Míchání textur . . . . .	17
3.5	Výpočet stínů v reálném čase . . . . .	18
3.6	Předpočtené osvětlení . . . . .	21
3.6.1	Vytvoření texturovacích souřadnic . . . . .	21
3.6.2	Oprava chyby sousedících trojúhelníků . . . . .	23
3.6.3	Generování map osvětlení . . . . .	23
3.6.4	Spekulární složka osvětlovací modelu . . . . .	24
3.7	Odlesky povrchů . . . . .	24
<b>4</b>	<b>Realizace</b>	<b>27</b>
4.1	Simulátor . . . . .	27
4.2	Realizace na platformě Android . . . . .	29
4.2.1	Problematika Android API . . . . .	29

4.2.2	Využití vláken . . . . .	31
4.2.3	Práce se soubory . . . . .	31
4.3	Předpočtené osvětlení . . . . .	32
4.3.1	Vytváření map osvětlení . . . . .	33
4.3.1.1	Generování texturovacích souřadnic pro mapy osvětlení . . . . .	34
4.3.1.2	Generování map osvětlení . . . . .	35
4.3.2	Příprava dat pro dynamickou aktualizaci map osvětlení . . . . .	37
4.3.3	Statické osvětlení a dynamické objekty . . . . .	38
4.3.4	Světla vozidel . . . . .	39
4.4	Screen-space přístup . . . . .	40
4.4.1	Rozměr textur na OpenGL ES 2.0 . . . . .	41
4.4.2	Motion-blur . . . . .	41
4.4.3	Odlesky . . . . .	42
<b>5</b>	<b>Testování</b>	<b>45</b>
5.1	Generování map osvětlení . . . . .	46
5.2	Simulátor . . . . .	47
<b>6</b>	<b>Závěr</b>	<b>51</b>
<b>Literatura</b>		<b>53</b>
<b>A</b>	<b>Seznam použitých zkratek</b>	<b>55</b>
<b>B</b>	<b>Obsah přiloženého CD</b>	<b>57</b>
<b>C</b>	<b>Uživatelská příručka</b>	<b>59</b>
C.1	Instalace simulátoru . . . . .	59
C.2	Používání simulátoru . . . . .	60
C.3	Generování map osvětlení . . . . .	60

# Seznam obrázků

2.1	Ukázka mapy osvětlení na jednoduchém 3D modelu . . . . .	4
2.2	Ukázka aktualizace map osvětlení na jednoduchém 3D modelu, žlutou barvou je znázorněna část záplaty, která nenesí informaci o změně osvětlení . . . . .	5
2.3	Komponenty platformy Android, komponenty vyznačené modrou barvou jsou napsané v jazyce Java, žlutou barvou je virtuální stroj, který umožňuje běh Java aplikací, zelenou barvou jsou C/C++ knihovny a červenou je Linuxové jádro . . . . .	7
2.4	Ukázka nočního renderingu . . . . .	8
2.5	Ukázka výsledků techniky imperfektních stínových map pro efektivní výpočet nepřímého osvětlení [6] . . . . .	9
2.6	Ukázka závodních simulátorů s noční scénou pro platformu Android, první řada zleva: Asphalt 7, Asphalt 8, Need for Speed Most Wanted, druhá řada zleva: GT Racing 2, Sports Car Challenge, Track Racing . . . . .	11
3.1	Phongův osvětlovací model . . . . .	13
3.2	Vektory k výpočtu difúzního osvětlení, $\vec{L}$ je normalizovaný směr světla, $\vec{N}$ je normalizovaná normála povrchu, $\alpha$ je úhel, který tyto vektory svírají, $\vec{E}$ je normalizovaný směr pohledu kamery (eye vektor) a $\vec{P}$ je bod na povrchu tělesa . . . . .	14
3.3	Vektory k výpočtu spekulárního osvětlení, $\vec{E}$ je normalizovaný směr pohledu kamery (eye vektor), $\vec{R}$ je normalizovaný směr odrazu světla, $\beta$ je úhel, který tyto vektory svírají a $\vec{P}$ je bod na povrchu tělesa . . . . .	14
3.4	Vektory k výpočtu efektivity reflektoru, $\vec{D}$ je normalizovaný směr světla, $\vec{V}$ je normalizovaný směr světla na kraji světelného kužeče, $\alpha$ je úhel, který tyto vektory svírají, $\vec{L}_p$ je pozice zdroje světla a $\vec{P}$ je bod na povrchu tělesa . . . . .	15
3.5	Ukázka jednotlivých typů stínování . . . . .	16
3.6	Pomocný obrázek k výpočtu barycentrické interpolace . . . . .	17
3.7	Ukázka problému při vykreslení polopruhledného materiálu před neprůhledným . . . . .	18
3.8	Pomocný obrázek k výpočtu zastínění pomocí stínových map . . . . .	18
3.9	Ukázka scény (vlevo) a její stínové mapy (vpravo) . . . . .	19
3.10	Rozdíl mezi ostrými stíny (vlevo) a měkkými stíny (vpravo) . . . . .	20
3.11	Ukázka vyhodnocení zastínění pomocí PCF 3x3. Vlevo jsou data ze stínové mapy, uprostřed vyhodnocení zastínění jednotlivých bodů (0 je viditelný bod, 1 je zastíněný bod) a vpravo výsledné zastínění . . . . .	20
3.12	Ukázka výsledků filtrování PCF . . . . .	20
3.13	Výsledek řešení problému batohu se spárovanými trojúhelníky . . . . .	21

3.14 Změna délky přepony trojúhelníka pro vytvoření páru dvou trojúhelníků, vlevo původní nepravoúhlý trojúhelník, vpravo trojúhelník s novou délkou přepony . . . . .	22
3.15 Problém kolizních texelů dvou trojúhelníků . . . . .	23
3.16 Využití cube mapování na stínové mapy u bodových osvětlení, vlevo dva náhledy scény, vpravo stínová mapa . . . . .	24
3.17 Diagram odrazu vodní hladiny [4] . . . . .	25
4.1 UML diagram projektu, horní část je abstraktní, dolní část implementuje jednotlivá rozhraní a načítá data do paměti, spustitelné soubory jsou označeny tečkou . . . . .	28
4.2 UML diagram spustitelného klienta pro Android, kde libopen4speed.so je zkompilovaná C++ knihovna projektu . . . . .	30
4.3 Scéna s aplikovanými mapami osvětlení lamp . . . . .	32
4.4 Ukázka difúzní textury modelu (vlevo) a osvětlovací textury (vpravo) . . . . .	33
4.5 Ukázka naplnění kD stromu pomocí metody Packing Lightmaps . . . . .	34
4.6 Ukázka hledání průsečíku paprsku v datové struktuře quadtree . . . . .	36
4.7 Ukázka převodu záplaty osvětlovací mapy do vektorové podoby, vlevo původní mapa osvětlení, uprostřed první iterace algoritmu a vpravo další iterace . . . . .	38
4.8 Vržený stín vozidla . . . . .	39
4.9 Ukázka světel pomocí polopruhledných kuželů . . . . .	40
4.10 Vykreslovací řetězec simulátoru . . . . .	40
4.11 Odlesky na povrchu vozovky . . . . .	42
4.12 Ukázka výsledných odlesků na vozidle, vlevo bez odlesků, vpravo s odlesky . . . . .	43
5.1 Závislost rychlosti vykreslování scény na zařízení a poměru rozlišení oproti nativnímu . . . . .	47
5.2 Závislost rychlosti vykreslování scény na platformě a rozlišení textur map osvětlení při vizuálních detailech normal . . . . .	48

# Seznam tabulek

5.1	Parametry zařízení, na kterých jsem projekt testoval . . . . .	45
5.2	Vykreslování 106 bodových zdrojů světla pomocí OpenGL a pomocí raycastingu	46
5.3	Vykreslování několika plošných zdrojů světel navzorkovaných na 202 bodových světel, optimalizace spočívá v testování paprsků vždy mezi dvěma trojúhelníky a detekování, zda bod na trojúhelník dosvítí . . . . .	46
5.4	Vykreslování všech plošných zdrojů světel navzorkovaných na 372854 bodových světel, optimalizace spočívá v testování paprsků vždy mezi dvěma trojúhelníky a detekování, zda bod na trojúhelník dosvítí . . . . .	46
5.5	Závislost rychlosti vykreslování scény na zařízení a poměru rozlišení oproti nativnímu(v závorce je uvedena konfigurace v nastavení simulátoru). Z každého měření je uvedená minimální a maximální hodnota . . . . .	47
5.6	Závislost rychlosti vykreslování scény na platformě a rozlišení textur map osvětlení při vizuálních detailech normal(poměr 0.6x ku nativnímu rozlišení). Z každého měření je uvedená minimální a maximální hodnota . . . . .	48
5.7	Závislost využití paměti na rozlišení textur map osvětlení. Testováno na Acer TravelMate P253-e (laptop) . . . . .	48



# Kapitola 1

## Úvod

V minulosti se výrobci počítačového hardwaru primárně soustředili na dosažení nejvyššího výpočetního výkonu na trhu. Během posledních let v tomto konkurenčním boji nastala změna. Už tolik nedochází ke zvyšování výkonu hardwaru, výrobci se soustředí spíše na miniaturizaci. S miniaturizací hardwaru souvisí stoupající výpočetní výkon mobilních zařízení.

Mobilní telefony a tablety jsou zatím v porovnání s výkonem osobních počítačů slabší, ale to za několik let nemusí platit. Chytré mobilní telefony postupně přebírají funkce jiných zařízení. Některé mobilní telefony lze použít jako dálkový ovladač k televizi, platební kartu, autonavigaci a další.

Trend udělat z mobilního telefonu univerzální zařízení zasahuje i do oblasti počítačových her. To souvisí s odvětvím počítačové grafiky. Výrobci her se předhánějí, kdo vytvoří nejrealističtější grafiku ve svých hrách. Tím nepřímo přispívají k rozvoji počítačové grafiky.

Vývoj her pro mobilní zařízení je v porovnání s vývojem her pro osobní počítače rozdílný z několika hledisek. Používá se rozdílné ovládání, mobilní zařízení je menší a je zde různý výpočetní výkon. V mé práci se zabývám problematikou výpočetního výkonu z hlediska grafického výpočtu.

Obecně platí, že během vykreslování 3D scény se některé výpočty cyklicky opakují. V práci většinu těchto výpočtů předpočítávám a tím získám ve výsledné implementaci daleko lepší výsledky, než kdybych předpočítání nepoužil.



# Kapitola 2

## Specifikace práce

Cílem práce bylo realizovat efektivní vykreslování rozsáhlých nočních scén na platformě Android za použití grafické knihovny OpenGL. Rozhodl jsem se práci realizovat v podobě závodního simulátoru, který obsahuje několik pohybujících se vozidel a blikajících světel. Tato podoba realizace plně využívá veškeré grafické techniky, které jsem implementoval a zároveň umožní lépe ohodnotit použitelnost projektu, než kdybych zvolil podobu technického dema.

Důraz byl kladen na co nejvyšší optimalizaci vykreslování. Kromě hlavní grafické techniky map osvětlení (anglicky lightmap) bylo použito několik dalších technik řešící jednotlivé problémy, které vznikají při použití map osvětlení. Problémy nastávají hlavně při použití dynamických objektů nebo při změně osvětlení. Technika map osvětlení řeší pouze statickou scénu a v případě použití dynamické scény je potřeba metodu značně rozšířit.

### 2.1 Požadavky na funkčnost

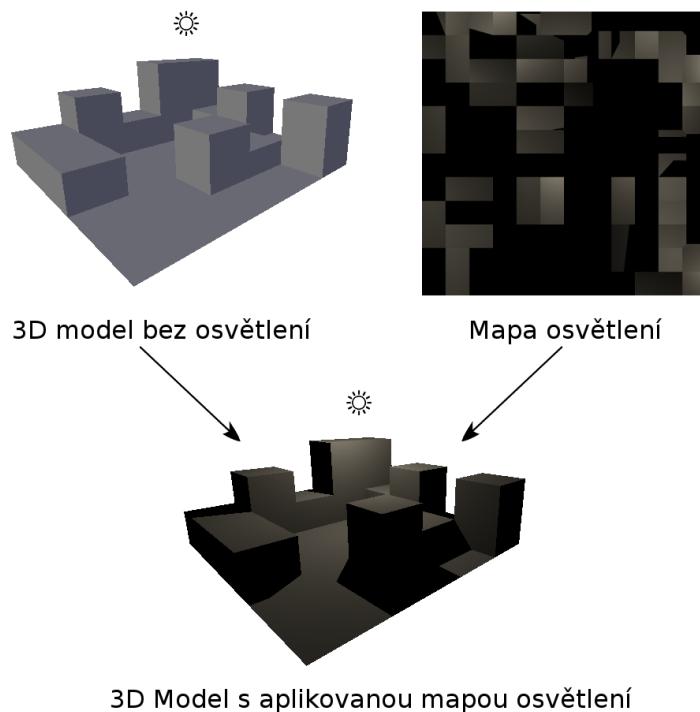
Zádání předpokládá, že projekt bude rozdelen do dvou samostatných částí. První částí bude předzpracování, ve kterém se budou vytvářet mapy osvětlení. Druhou částí projektu bude závodní simulátor, který mapy osvětlení využije. Fázi předzpracování by teoreticky bylo možné provádět i na platformě Android. Vzhledem k tomu, že klasické počítače dosahují vyššího výkonu než mobilní zařízení, bude daleko efektivnější provést předzpracování na počítači.

Simulátor bude využívat vlastní formát, který bude umožňovat mít v sobě jak souřadnice standardních textur, tak i souřadnice v mapách osvětlení. Formát bude zachovávat veškeré informace o modelu. I když je třeba výsledný simulátor nevyužije, bude dobré tyto informace ponechat pro případ dalšího vývoje. Formát bude schopen uložit některé další informace o materiálu a shaderu.

Plocha map osvětlení bude co možně nejvíce využita z důvodu očekávané malé paměti vymezeny pro velké textury. Aby bylo možné určit nevhodnější rozlišení těchto textur, bude umožněno škálování map osvětlení.

### 2.1.1 Mapy osvětlení

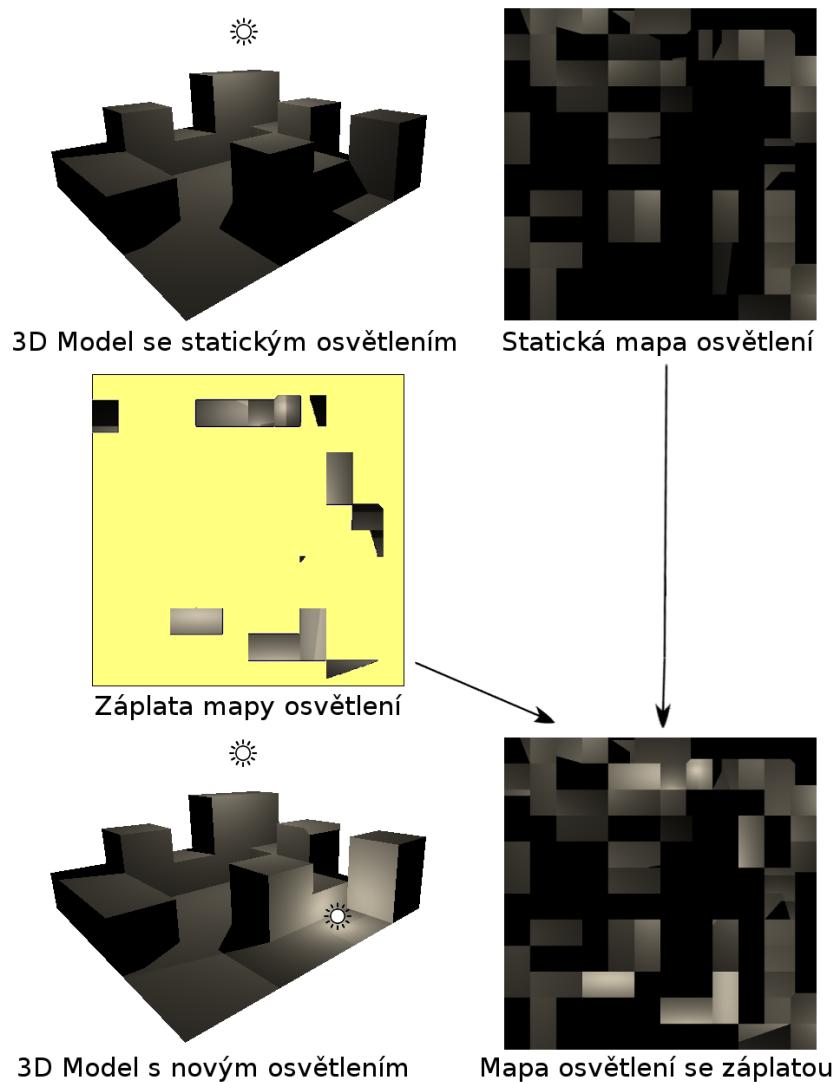
Mapy osvětlení jsou textury 3D modelů, které obsahují informace o difúzním osvětlení. Difúzní osvětlení je závislé pouze na poloze světla, směru světla a povrchu tělesa, lze jej tedy pro statickou část modelu předpočítat.



Obrázek 2.1: Ukázka mapy osvětlení na jednoduchém 3D modelu

Aby bylo možné mapy osvětlení dynamicky aktualizovat, je potřeba mít do map rychlý zapisovací přístup. Rychlé zapisování do textur se v OpenGL realizuje pomocí Frame Buffer Objektu (FBO). Pro aktualizaci musíme mít předpočítanou tzv. záplatu. Záplatou se rozumí menší textura, která se přidá do současné osvětlovací mapy. Přidáním záplaty lze tedy přidat do scény další světlo, které máme předpočítané. Záplata jde z FBO opět odebrat (provede se odečet stejné záplaty).

Protože uchovávání záplat v paměti by bylo příliš paměťově náročné, jsou uloženy ve vektorové podobě přímo na GPU pomocí Vertex Buffer Objektu (VBO).



Obrázek 2.2: Ukázka aktualizace map osvětlení na jednoduchém 3D modelu, žlutou barvou je znázorněna část záplaty, která nenese informaci o změně osvětlení

### 2.1.2 Statické osvětlení s dynamickými objekty ve scéně

Při použití dynamických objektů ve scéně se statickém osvětlením vznikají dva základní problémy. Prvním problém je, že dynamické objekty mají jiné osvětlení než statická scéna a to vypadá velice nepřirozeně. Druhým problémem je, že pokud dynamický objekt zastíní zdroj světla, tak nevznikne stín na statických objektech.

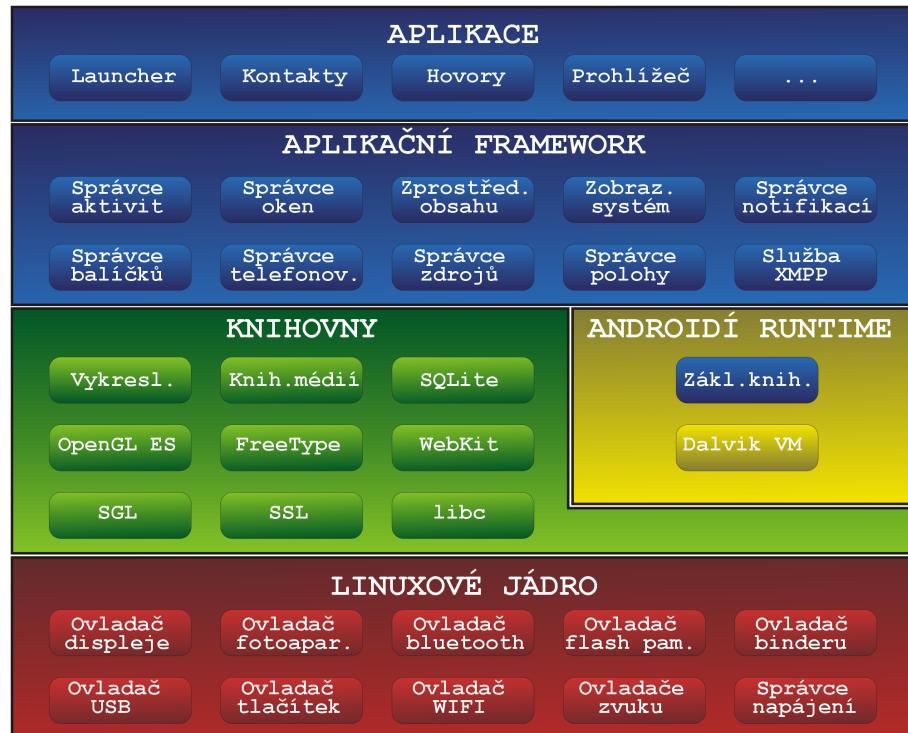
Oba problémy lze vyřešit jen částečně. V prvním případě lze přečíst informaci o intenzitě osvětlení z nejbližšího bodu na statickém objektu a tuto intenzitu aplikovat na dynamický objekt. V druhém případě při vykreslování jednotlivých pixelů lze zjistit, zda se v okolí

vykreslovaného bodu nenachází dynamický objekt a tím approximačně zjistit, zda byl vykreslovaný bod zastíněn.

## 2.2 Cílová platforma Android

Platforma Android je postavena na Linuxovém jádře a využívá knihovny, které bývají součástí unixových systémů. Od klasického desktopového Linuxu se odlišuje hlavně v tom, že jako hlavní programovací jazyk využívá Javu. Pomocí Javy je naprogramováno celé Android API, které zprostředkovává téměř veškeré služby systému.

Na Androidu je téměř nemožné spustit C/C++ program přímo. Standardně se to řeší tak, že se vytvoří z programu knihovna, kterou spustí kód napsaný v Javě. Pak je nutné veškeré vstupy/výstupy obsluhovat v jazyce Java a to velice komplikuje portování programů z desktopového Linuxu na Android. Výjimku tvoří OpenGL, které umožňuje vykreslovat na displej přímo z C/C++ kódu.



Obrázek 2.3: Komponenty platformy Android, komponenty vyznačené modrou barvou jsou napsané v jazyce Java, žlutou barvou je virtuální stroj, který umožňuje běh Java aplikací, zelenou barvou jsou C/C++ knihovny a červenou je Linuxové jádro

Pro platformu Android existuje nepřeberné množství aplikací a toho začínají využívat ostatní platformy. V současné době platformy BlackBerry a Jolla umožňují spouštět Android aplikace, dále existuje platforma NokiaX, která je postavena přímo na operačním systému Android.

## 2.3 Existující implementace

Rešerši existujících implementací jsem zaměřil na dvě různé kategorie. První kategorie jsou implementace pro PC, kde je nejčastěji řešeno nepřímé osvětlení, které by na mobilních zařízeních zatím použít nešlo. Druhá kategorie jsou závodní simulátory s noční jízdou pro platformu Android. V této kategorii bývají nejčastěji použity techniky jako v této práci.

### 2.3.1 Projekty zabývající se vykreslování noční scény

V této sekci jsem vybral co nejrealističtější implementace, ke kterým existuje nějaký článek o tom, jak bylo výsledku dosaženo. Existuje sice několik implementací, které vypadají ještě více realističtěji. To jsou ale většinou komerční produkty, které nesdílejí ostatním, jakým způsobem byla implementace realizována.

#### 2.3.1.1 Proceduralní modelování města a osvětlení nočního města

Tato implementace je z roku 1999 a zabývá se proceduálním modelováním města (konkrétně Bostonu) a následně jej vykresluje pomocí sledování paprsku za využití algoritmu Monte Carlo. Implementace pochopitelně neběží v reálném čase. Výsledek je při detailním záběru po grafické stránce velice realistický. Vytknul bych jen, že reflektory vozidla osvítí pouze blízké okolí. Při záběru na město z výšky výsledek nebudí tolik realistický dojem jako při detailním záběru.



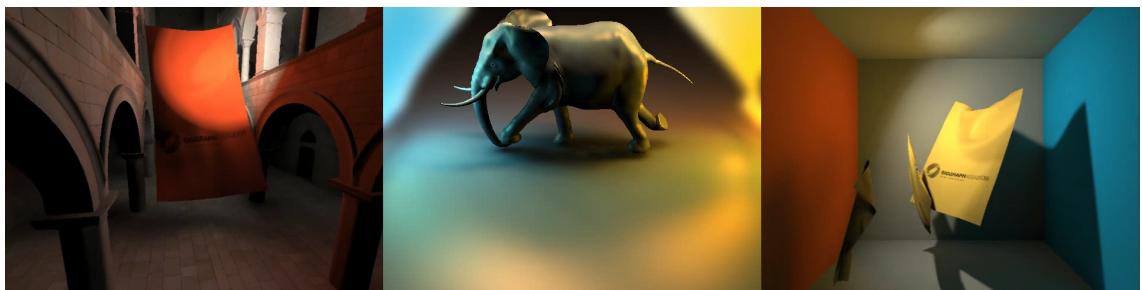
Obrázek 2.4: Ukázka nočního renderingu

### 2.3.1.2 Imperfektní stínové mapy pro výpočet nepřímého osvětlení

Tato implementace, představená roku 2008, se nezabývá přímo noční scénou, zabývá se nepřímým osvětlením, které s noční scénou souvisí. Implementace běží v reálném čase a její výsledky jsou srovnatelné se snímky, které se vykreslují několik hodin.

V této implementaci se nevyužívá předpočtených dat. Využívá se zde virtuálních bodových světel (VPL), která simuluje odrazy světla. Z každého VPL se vytvoří stínová mapa o malém rozlišení a při vykreslování výsledné scény se vyhodnocuje viditelnost ze všech VPL.

Implementace podporuje více typů zdrojů světla, včetně plošných. Nemá problémy s barevným světlem ani s kaustiky.



Obrázek 2.5: Ukázka výsledků techniky imperfektních stínových map pro efektivní výpočet nepřímého osvětlení [6]

### 2.3.2 Závodní simulátory s noční scénou pro platformu Android

V této sekci jsem vyhledával i implementace, u kterých není uvedeno, jak bylo výsledků dosaženo. Je to z důvodu malého počtu implementací odpovídající dané kategorii.

#### 2.3.2.1 Asphalt Urban od Gameloftu

Asphalt Urban je série mobilních závodních simulátorů, jejichž první díl byl publikován spolu s herním smartphonem Nokia N-Gage. Jedná se o úspěšnou sérii, která přitahuje hráče všech možných platform.

Zmíním 7.díl ze série Asphalt Urban, který osobně považuji za nejúspěšnější díl. Osvětlení z pouličních lamp je součástí textur, reflektor vozidla je tvořen zřejmě pomocí projektivní textury. Povrch zrcadlově odráží 3D objekty. Po bližším zkoumání lze zjistit, že některé tyto odražené objekty jsou odlišné. Domnívám se, že zde byl použit duplicitní objekt.

Další díl ze série Asphalt Urban už odlesky řeší lépe. Odráží se hlavně světla a odlesk je lépe přizpůsoben povrchu. Vzniká zde dojem mokré silnice. Dochází i k rozmazání brzdových světel. Osvětlení od lamp je opět řešeno texturou. Nepříjemnou změnou je absence reflektoru vozidla a přidání rušivého chvění kamery během jízdy.

### 2.3.2.2 Need for Speed Most Wanted od EA Games a Firemonkeys

Simulátor, který je spíše známý z PC, na mobilním trhu takový úspěch nemá. Od PC verze je velice odlišný, ale rozhodně se nejedná o nějakou lacinou napodobeninu. Jedná se o první závodní simulátor pro mobilní zařízení, který disponuje modelem ničení vozidla (je možné vozidlo poškrábat, rozbít mu okna apod.).

Oproti Asphalt 8 má navíc efekt Depth-of-field. To znamená, že vzdálené modely jsou rozmazené a zabarveny do barvy pozadí. Tento efekt vytváří příjemný mlhovitý dojem. Jinak bych řekl, že tyto dva simulátory jsou si sobě velice podobné. Na stejném enginu funguje i známý simulátor Real Racing 3, který ale noční jízdu nemá.

### 2.3.2.3 GT Racing 2 od Gameloftu

Úspěšným závodním simulátorem je v současné době GT Racing 2, je to hlavně díky jeho zpracování. Jako jediný z uvedených simulátorů má při noční jízdě nízké ambientní osvětlení a scéna je osvětlena hlavně reflektorem vozidla.

Nejsem si jist, zda osvětlení lampami je přímo součástí textur nebo jestli jsou zde použity mapy osvětlení.

### 2.3.2.4 Sports Car Challenge od Fishlabs

Asi nejdetailnější model vozidla má právě Sports Car Challenge. Vývojář spolupracuje s předními výrobci automobilů a to se odrazilo právě na modelech vozidel. Modely jsou realistické a to včetně interiérů.

Projekt trpí slabší kompatibilitou s mobilními zařízeními, i když náročnost enginu je nízká. Co se týká noční jízdy, je realizována stejnými technikami jako denní jízda. Jsou zde použity pouze tmavé textury a halo efekt pouličních lamp.

### 2.3.2.5 Track Racing od vývojáře Soulkey

I když tento projekt není závodní simulátor s noční jízdou, zmiňuji ho hlavně kvůli ovládání. Jedná se o neoficiální remake hry Trackmania známé z PC. Projekt běží na Unity3D enginu a je dostupný na mnoha platformách, nedá se ale stáhnout přímo z Marketu či Storu.

U výše uvedených simulátorů vozidlo automaticky zrychlují a u některých i samo brzdí. Hráč nemá tedy nad vozidlem takovou kontrolu. V případě Track Racing má hráč nad vozidlem plnou kontrolu a zážitek ze hry se dá srovnat se zážitkem z původní PC hry.



Obrázek 2.6: Ukázka závodních simulátorů s noční scénou pro platformu Android, první řada zleva: Asphalt 7, Asphalt 8, Need for Speed Most Wanted, druhá řáda zleva: GT Racing 2, Sports Car Challenge, Track Racing



# Kapitola 3

## Teoretická část

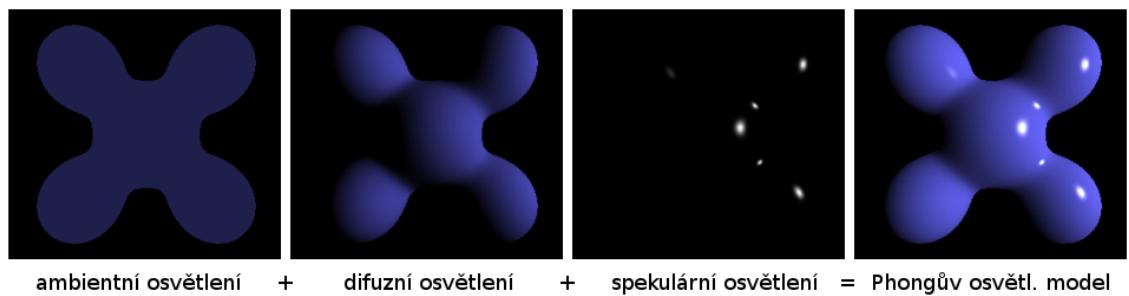
### 3.1 Osvětlovací model

Existuje několik osvětlovacích modelů, asi nejpoužívanější lokální osvětlovací model je Phongův. Phongův model se skládá ze tří typů osvětlení.

- ambientní osvětlení - nahrazuje nepřímé osvětlení konstantní hodnotou osvětlení
- difúzní osvětlení - osvětlení nezávislé na pohledu kamery, odpovídá ideálně matnému povrchu
- spekulární osvětlení - osvětlení závislé na pohledu kamery, odpovídá ideálně odrazivému povrchu

Jak je patrné z obrázku 3.1, výsledná intenzita osvětlení povrchu  $I$  je součtem ambientní složky ( $I_a$ ), difúzní složky ( $I_d$ ) a spekulární složky ( $I_s$ ).

$$I = I_a + I_d + I_s$$

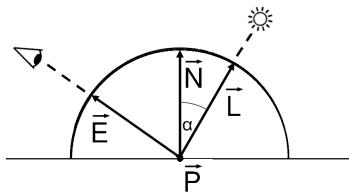


Obrázek 3.1: Phongův osvětlovací model

Všesměrové osvětlení je ve Phongovo osvětlovacím modelu konstantní pro daný objekt a říká se jí ambientní osvětlení  $I_a$ . Výsledná intenzita se spočte jako součin barvy ambientního světla  $C_a$ , koeficientem ambientního odrazu  $k_a$  a barvou povrchu  $C_d$ , která je shodná i pro difúzní složku.

$$I_a = C_a \cdot k_a \cdot C_d$$

Difúzní složka odpovídá ideálně matnému (Lambertovskému) povrchu a závisí na úhlu  $\alpha$  mezi vektoru  $\vec{L}$  a  $\vec{N}$ .



Obrázek 3.2: Vektor k výpočtu difúzního osvětlení,  $\vec{L}$  je normalizovaný směr světla,  $\vec{N}$  je normalizovaná normála povrchu,  $\alpha$  je úhel, který tyto vektoru svírají,  $\vec{E}$  je normalizovaný směr pohledu kamery (eye vektor) a  $\vec{P}$  je bod na povrchu tělesa

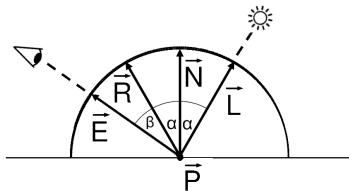
Intenzitu difúzní složky lze vyjádřit následovně.

$$I_d = C_l \cdot k_d \cdot C_d \cdot \cos(\alpha)$$

$C_l$  je barva světla,  $k_d$  koeficient difúzního odrazu,  $C_d$  barva povrchu a pro úhel  $\alpha$  platí, že  $\cos(\alpha) = \vec{L} \cdot \vec{N}$ . Po dosazení tedy získáme výsledný výraz.

$$I_d = C_l \cdot k_d \cdot C_d \cdot (\vec{L} \cdot \vec{N})$$

Spekulární (zrcadlová) složka odpovídá ideálně odrazivému tělesu a závisí na úhlu  $\beta$  mezi vektoru  $\vec{E}$  a  $\vec{R}$ .



Obrázek 3.3: Vektor k výpočtu spekulárního osvětlení,  $\vec{E}$  je normalizovaný směr pohledu kamery (eye vektor),  $\vec{R}$  je normalizovaný směr odrazu světla,  $\beta$  je úhel, který tyto vektoru svírají a  $\vec{P}$  je bod na povrchu tělesa

Intenzitu spekulární složky lze vyjádřit následovně.

$$I_s = C_l \cdot k_s \cdot C_s \cdot \cos^h(\beta)$$

$C_l$  je barva světla,  $k_s$  koeficient spekulárního odrazu,  $C_s$  barva lesklého povrchu (většinou bývá bílá),  $h$  je ostrost odrazu a pro úhel  $\beta$  platí, že  $\cos(\beta) = \vec{E} \cdot \vec{R}$ . Po dosazení tedy získáme výsledný výraz.

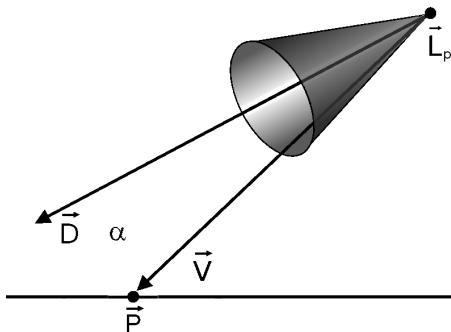
$$I_s = C_l \cdot k_s \cdot C_s \cdot (\vec{E} \cdot \vec{R})^h$$

## 3.2 Výpočet osvětlení

Pro výpočet osvětlení konkrétního bodu na obrazovce je potřeba nejdříve spočítat příspěvky jednotlivých světel pro tento bod. Příspěvky jednotlivých světel spočteme jako součin efektivity reflektoru pro daný vertex  $E_r$ , faktoru útlumu  $F_a$  a součtu složek Phongova osvětlovacího modelu.

$$\text{light}_i = E_r \cdot F_a \cdot (I_a + I_d + I_s)$$

Efektivita reflektoru  $E_r$  je parametr určující intenzitu osvětlení zdroje světla typu reflektor. Příkladem takového zdroje světla je například baterka.



Obrázek 3.4: Vektory k výpočtu efektivity reflektoru,  $\vec{D}$  je normalizovaný směr světla,  $\vec{V}$  je normalizovaný směr světla na kraji světelného kuželetu,  $\alpha$  je úhel, který tyto vektory svírají,  $\vec{L}_p$  je pozice zdroje světla a  $\vec{P}$  je bod na povrchu tělesa

Hodnota efektivity reflektoru se spočte podle následujících podmínek.

Pokud světlo není reflektor, poté  $E_r = 1$ .

Pokud bod leží mimo světelný kužel, tedy  $(\vec{V} \cdot \vec{D}) < \cos(\alpha)$ , poté  $E_r = 0$ .

Pro ostatní případy  $E_r = \max(0, \cos(\vec{V} \cdot \vec{D}))^{k_{se}}$ , kde  $k_{se}$  je parametr spot efektu.

Faktor útlumu  $F_a$  řeší, jak moc má být intenzita světla utlumena s rostoucí vzdáleností  $d$  od zdroje světla. Konstantní útlum určuje parametr  $k_c$ , lineární útlum parametr  $k_l$  a kvadratický útlum parametr  $k_q$ .

$$F_a = \frac{1}{k_c + k_l \cdot d + k_q \cdot d^2}$$

Výsledná barva bodu na obrazovce se spočítá jako součet barvy emisního světla  $C_e$ , barvy globálního ambientního světla  $C_{ag}$  a sumy příspěvků jednotlivých světel.

$$color = C_e + C_{ag} + \sum light_i$$

### 3.3 Stínování

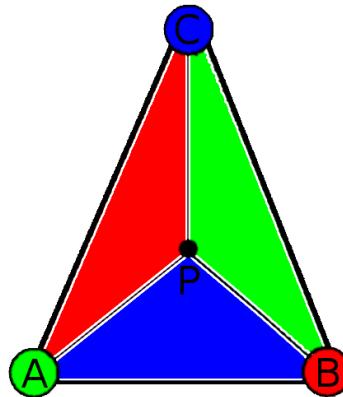
Pojem stínování v počítačové grafice znamená postup vybarvování trojúhelníků. Existují tři typy stínování.

- Konstantní stínování - celý polygon se vybarví jednou barvou dle normály povrchu. Používá se ve scénách, ve kterých se nacházejí pouze směrové zdroje světla.
- Gouraudovo stínování - spočítá se osvětlení ve vrcholech a následně se interpolují hodnoty mezi vrcholy. Toto stínování je velmi rychlé. Problém nastane, pokud světlo osvětuje střed polygonu a zároveň neosvětuje nebo jen částečně osvětuje vrchol. Osvětlení je po té ve vrcholech velmi nízké a nemůže tedy vzniknout korektně osvětlený obraz.
- Phongovo stínování - spočítá se osvětlení pro každý pixel polygonu na obrazovce. Dnes je to asi nejpoužívanější stínování. V OpenGL se toto stínování provádí pomocí shaderů. Vstupem do vertexového shaderu je pozice vrcholu a jeho normála. Ve vertex shaderu se provede transformace do pohledu kamery a určí se, že pozice vrcholu a normála se mají interpolovat. Po rasterizaci získáme interpolované hodnoty jako vstup ve fragment shaderu a zde provedeme výpočet podle výše uvedeného osvětlovacího modelu.



Obrázek 3.5: Ukázka jednotlivých typů stínování

Interpolaci provádí OpenGL automaticky. Jeden ze způsobů jak provést interpolaci ručně je provést rasterizaci (tím získat všechny body uvnitř trojúhelníka) a poté provést barycentrickou interpolaci.



Obrázek 3.6: Pomocný obrázek k výpočtu barycentrické interpolace

Nejdříve potřebujeme spočítat obsah jednotlivých trojúhelníků, abychom získali váhu hodnot v jednotlivých vrcholech.

$$S_{PBC} = \frac{|\vec{PB} \times \vec{PC}|}{2}; S_{APC} = \frac{|\vec{AP} \times \vec{AC}|}{2}; S_{ABP} = \frac{|\vec{AB} \times \vec{AP}|}{2}; S_{ABC} = \frac{|\vec{AB} \times \vec{AC}|}{2}$$

Pokud si označím hodnotu u bodu  $A$  jako  $a$ , u bodu  $B$  jako  $b$  a u bodu  $C$  jako  $c$ , získáme interpolovanou hodnotu  $p$  v bodě  $P$  následovně.

$$p = a \cdot \frac{S_{PBC}}{S_{ABC}} + b \cdot \frac{S_{APC}}{S_{ABC}} + c \cdot \frac{S_{ABP}}{S_{ABC}}$$

## 3.4 Míchání textur

Míchání textur se v OpenGL provádí pomocí blendingu. Tato technika míchá cílový obraz (ten již vykreslený ve FBO) a zdrojový obraz, tedy texturu namapovanou na nějaký model.

Vstupem pro blending funkci je zdrojová barva  $C_s$ , cílová barva  $C_d$  a parametry  $B_s$  a  $B_d$ , které určují pro každý kanál, jak se má hodnota míchat. Výstupem funkce je nová cílová barva  $C_d$ .

$$C_d = C_s \cdot B_s + C_d \cdot B_d$$

Příklad použití blendingu je technika poloprůhledného materiálu. Nejdříve se vykreslí neprůhledná část scény (tím se vykreslí i hloubkový buffer). Poté se vypne zápis do hloubkového bufferu a vykreslí se geometrie s částečnou průhledností.



Obrázek 3.7: Ukázka problému při vykreslení polopruhledného materiálu před neprůhledným

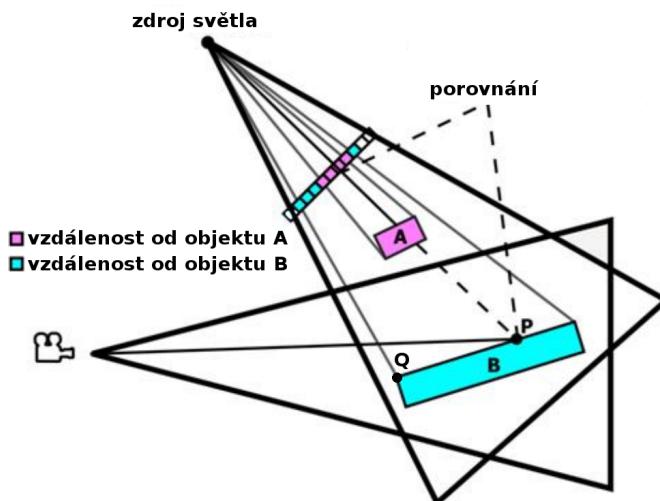
### 3.5 Výpočet stínů v reálném čase

Stíny v počítačové grafice napomáhají k vnímání souvislostí ve scéně. Pokud ve scéně stíny nepoužijeme, nebudeme například schopni poznat, zda se těleso dotýká povrchu nebo se vznáší (výjimku tvoří pohled, při kterém vzniká viditelná mezera mezi objektem a povrchem).

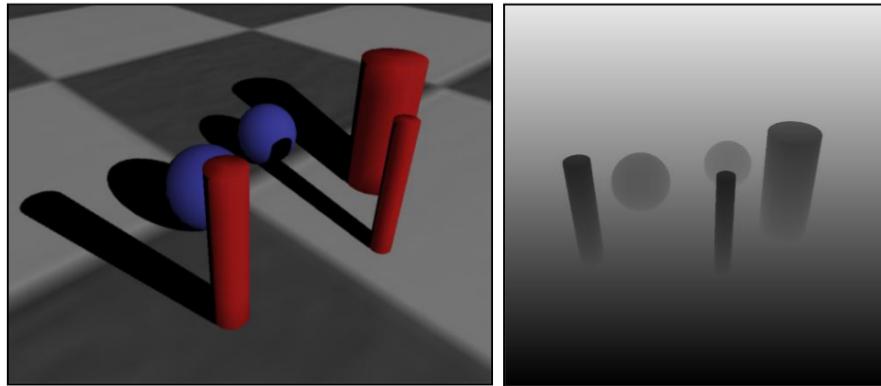
Abychom mohli stín spočítat, musíme znát informace o okolí aktuálně vykreslovaného primitiva. To je v OpenGL problém, protože při vykreslování primitiv nemáme standardně žádné informace o okolí. Problém se řeší pomocí tzv. stínových map.

Stínová mapa je textura, do které se vykreslí scéna z pohledu zdroje světla. Do stínové mapy se nekreslí barevná informace, kreslí se do ní vzdálenost od zdroje světla (na obrázku 3.9 je tato vzdálenost namapovaná na odstíny šedi).

Princip stínových map je naznačen na obrázku 3.8. Při vyhodnocování bodu P se spočítá vzdálenost bodu od zdroje světla a porovná se s hodnotou uloženou ve stínové mapě. V případě bodu P je vzdálenost vyšší než hodnota ve stínové mapě a bod je vyhodnocen jako zastíněný. V případě bodu Q je vzdálenost stejná jako ve stínové mapě (s numerickou tolerancí) a bod je tedy viditelný.



Obrázek 3.8: Pomocný obrázek k výpočtu zastínění pomocí stínových map



Obrázek 3.9: Ukázka scény (vlevo) a její stínové mapy (vpravo)

Transformace ze světových souřadnic do souřadnic projekce je vyjádřen pomocí součinu modelové, pohledové a projekční matice (anglicky model, view, projection matrix, používá se zkratka MVP). Transformaci kamery si označím jako  $MVP_{camera}$  a transformaci pro pohled zdroje světla jako  $MVP_{light}$ . Vrchol primitiva si označím jako  $\vec{v}$ , spočteme vektor  $\vec{p}$ , jehož složky  $x, y$  určují pozici hodnoty ve stínové mapě a složka  $z$  hodnotu texelu. Pokud bychom měli hodnotu ve stínové mapě přepisovat, zapíšeme vždy nižší hodnotu (nižší hodnota je bod bližší zdroji světla).

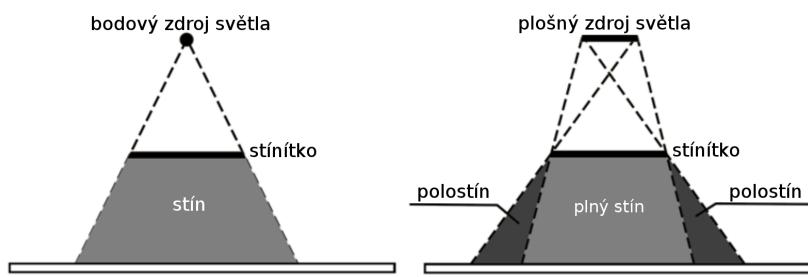
$$\vec{p} = MVP_{light} \cdot \begin{pmatrix} \vec{v} \\ 1 \end{pmatrix}$$

Pomocí tohoto postupu získáme stínovou mapu, kterou využijeme při vykreslování scény z pohledu kamery. Vypočteme  $\vec{p}$  stejně jako při výpočtu stínové mapy a získáme vzdálenost  $d$  ze stínové mapy  $SM$ .

$$d = SM[\vec{p}_x, \vec{p}_y]$$

Nakonec zjistíme, jak se tyto hodnoty liší pomocí podmínky  $|\vec{p}_z - d| < \epsilon$ , kde  $\epsilon$  je malé číslo filtrující numerickou chybu. Pokud je podmínka splněna, je bod osvětlený daným zdrojem světla.

Výše uvedeným postupem jsou řešeny ostré stíny pro bodové zdroje světla, případně světelné reflektory. Dále je potřeba řešit měkké stíny, které vznikají z plošných zdrojů světla. Rozdíl mezi ostrými a měkkými stíny je znázorněn na obrázku 3.10.



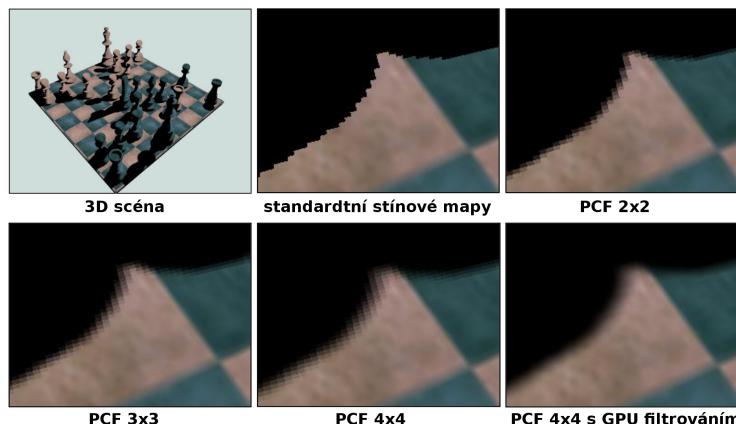
Obrázek 3.10: Rozdíl mezi ostrými stíny (vlevo) a měkkými stíny (vpravo)

Výpočet plošných zdrojů světla v reálném čase je stále výpočetně příliš náročný. Pro zobrazení měkkých stínů je možné použít filtrování PCF (Percentage Closer Filtering), které umožní vytvořit měkký stín pro bodový zdroj světla.

Filtrování PCF je snadno implementovatelné, vrací dobré výsledky, ale může být náročnější na výpočetní výkon. Princip filtrování PCF je, že při výpočtu zastínění se ze stínové mapy vyhodnocují i okolní body a tím vzniká přechod mezi zastíněnými a nezastíněnými pixely na obrazovce.

<table border="1"> <tr><td>15</td><td>14</td><td>12</td></tr> <tr><td>13</td><td>9</td><td>7</td></tr> <tr><td>12</td><td>8</td><td>6</td></tr> </table>	15	14	12	13	9	7	12	8	6	porovnání s hloubkou 10	<table border="1"> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> </table>	0	0	0	0	1	1	0	1	1	5x0, 4x1 zastínění bodu 44%
15	14	12																			
13	9	7																			
12	8	6																			
0	0	0																			
0	1	1																			
0	1	1																			

Obrázek 3.11: Ukázka vyhodnocení zastínění pomocí PCF 3x3. Vlevo jsou data ze stínové mapy, uprostřed vyhodnocení zastínění jednotlivých bodů (0 je viditelný bod, 1 je zastíněný bod) a vpravo výsledné zastínění



Obrázek 3.12: Ukázka výsledků filtrování PCF

## 3.6 Předpočtené osvětlení

Předpočtené osvětlení je z hlediska vykreslování poměrně jednoduchá technika. Na 3D model se aplikuje další textura či textury, které mají vlastní texturovací souřadnice a nesou v sobě informaci o osvětlení jednotlivých trojúhelníků. Jak již bylo uvedeno, těmto texturám se říká mapy osvětlení a dalo by se říct, že se jedná o techniku, která rozšiřuje techniku předpočtených stínů.

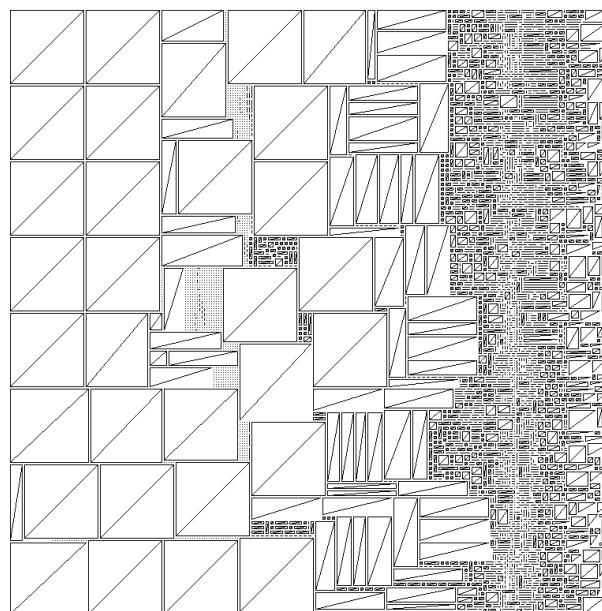
Z hlediska generování těchto textur je tato technika náročnější než výsledné vykreslování, skládá se z několika kroků. V prvním kroku je třeba vytvořit texturovací souřadnice pro jednotlivé trojúhelníky.

### 3.6.1 Vytvoření texturovacích souřadnic

Pro mapy osvětlení není vhodné použít standardní texturovací souřadnice. V osvětlovacích mapách je potřeba zajistit, aby každý texel mapy odpovídal právě jednomu bodu v prostoru. Pokud by tomu tak nebylo, stalo by se, že při osvětlení jednoho bodu se rozzáří i jiný bod než ten osvětlený.

Obsah povrchu 3D modelu v ideálním případě odpovídá obsahu ploch map osvětlení. Abychom se tomuto případu přiblížili, musíme plochu map co nejvíce využít. Maximálním využitím 2D plochy se zabývá problém batohu, který patří do kategorie problémů NP-hard (obtížný nedeterministický problém řešitelný v polynomiálním čase).

Běžně se problém batohu zabývá pouze osově zarovnanými obálkovými tělesy (AABB). V této práci tento problém rozšiřuji o práci s trojúhelníky. Řešení spočívá ve spojování trojúhelníků do tvaru obdélníku, aby bylo možné problém řešit jako běžný problém batohu.



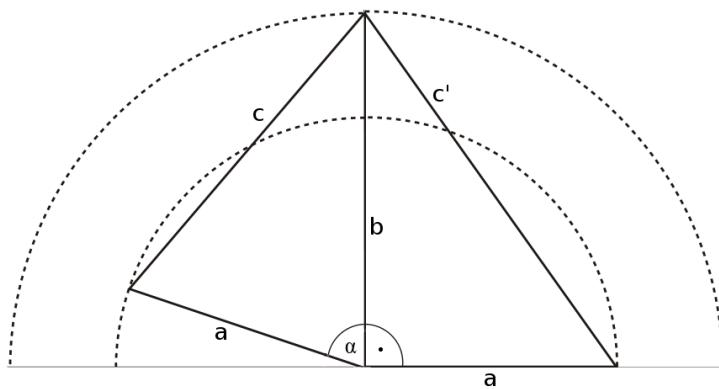
Obrázek 3.13: Výsledek řešení problému batohu se spárovanými trojúhelníky

Problém batohu lze řešit pomocí ILP (Integer Linear programming). Problém lze přeformulovat pomocí kolizí dvou AABB. Pokud si všechny levé horní body AABB označím souřadnicemi  $[minx, miny]$  a pravé dolní souřadnicemi  $[maxx, maxy]$ , platí poté následující výraz, pomocí kterého lze sestrojit ILP řešení (toto řešení neuvažuje otáčení AABB).

$$\forall_{i,j}, i \neq j : (minx_i < maxx_j) \cap (miny_i < maxy_j) \cap (maxx_i > minx_j) \cap (maxy_i > miny_j)$$

Abychom mohli problém řešit tímto způsobem, musíme mít nejdříve AABB místo trojúhelníků. Vytvoření AABB se zde provádí pomocí párování trojúhelníků.

V prvním kroku se spočítá Eulerovská délka jednotlivých hran, nejdelší hranu označíme jako  $c$  (to je přepona). Pokud chceme vytvořit plně využité AABB musíme mít všechny trojúhelníky pravoúhlé, z toho důvodu původní  $c$  zahodíme a vytvoříme pravoúhlý trojúhelník o délce stran  $a, b$ . Pro novou přeponu platí  $c' = \sqrt{a^2 + b^2}$ .

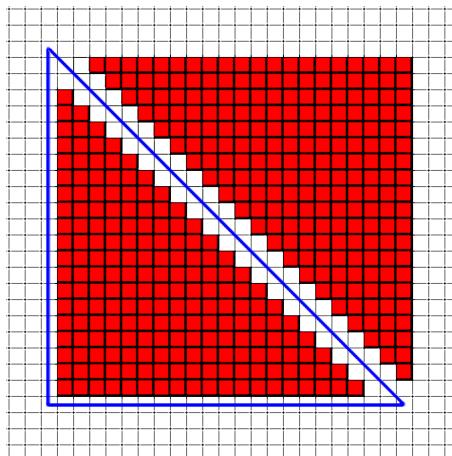


Obrázek 3.14: Změna délky přepony trojúhelníka pro vytvoření páru dvou trojúhelníků, vlevo původní nepravoúhlý trojúhelník, vpravo trojúhelník s novou délkou přepony

V dalším kroku hledáme párové trojúhelníky se stejnými délky stran  $a, b$  (s určitou tolerancí), párové se spojí přeponou k sobě. Pro nepárové trojúhelníky vytvoříme AABB o velikosti  $a, b$  a tím máme vyřešený problém párování trojúhelníků.

### 3.6.2 Oprava chyby sousedících trojúhelníků

Při použití výše uvedeného postupu vzniká problém při vykreslování, protože není u texelů kolem přepon jednoznačné, ke kterému trojúhelníku texel patří a proto vznikají ve výsledné scéně artefakty. Stává se to, protože sousedící trojúhelníky v mapě osvětlení spolu nesousedí ve scéně.



Obrázek 3.15: Problém kolizních texelů dvou trojúhelníků

Provádí se zde posun texturovacích souřadnic směrem dovnitř trojúhelníku, aby k tomuto problému nedocházelo. Posouvají se i osově zarované hrany, protože při použití lineárního filtrování textur by docházelo k obdobným artefaktům jako v případě přepon.

### 3.6.3 Generování map osvětlení

Existují dvě základní metody generování map osvětlení, pomocí vrhání či sledování paprsku a pomocí rasterizace. V této kapitole popisují rasterizační metodu (tedy pomocí OpenGL). Lze využít stínových map stejným způsobem, jako se používají k výpočtu zastínění v reálném čase. Rozdíl je pouze v transformování vrcholů a ve výpočtu barvy (nechceme vykreslovat texturu do světelnyh map, vedlo by to ke ztrátě kvality textur).

Nejdříve si vložíme ke každému vrcholu do VBO jeho souřadnice v osvětlovací mapě. Transformujeme vrchol do souřadnic mapy osvětlení. To se provede přiřazením souřadnic  $u, v$  jako pozici ve FBO (s pozicí vrcholu se počítá pouze při výpočtu osvětlení).

Výslednou barvu texelu  $C_t$  vypočteme z difúzního osvětlení pomocí níže uvedeného vzorce, kde  $E_r$  je efektivita reflektoru,  $F_a$  faktor útlumu,  $C_l$  barva světla,  $k_d$  koeficient difúzního odrazu,  $\vec{L}$  je normalizovaný směr světla a  $\vec{N}$  je normalizovaná normála povrchu. Více informací k výpočtu v kapitole 3.1.

$$C_t = E_r \cdot F_a \cdot C_l \cdot k_d \cdot (\vec{L} \cdot \vec{N})$$

Pokud bychom chtěli použít i bodové zdroje světla, musíme mít stínovou mapu pro všechny možné směry světelých paprsků. K tomu lze využít tzv. cube mapování. Cube mapování je mapování scény na krychli, při kterém se na každou stěnu vykreslí pohled v jednom směru s perspektivním úhlem  $90^\circ$ . V případě využití cube mapování na stínové mapy, se pro každou stranu provádí výpočet jakoby se jednalo o samostatné světlo (efektivita reflektoru  $E_r = 1$ ).



Obrázek 3.16: Využití cube mapování na stínové mapy u bodových osvětlení, vlevo dva náhledy scény, vpravo stínová mapa

### 3.6.4 Spekulární složka osvětlovací modelu

Spekulární složka osvětlovacího modelu je závislá na pohledu kamery a nelze jí tedy předpočítat a také nelze počítat v reálném čase spekulární složky pro desítky světel. Tento problém se dá řešit vypnutím vzdálených světel. Světla se seřadí podle vzdálenosti od kamery a provede se započítání  $n$  nejbližších (konstantu  $n$  je vhodné mít co nejnižší, protože spekulární složka je poměrně náročná na výpočet).

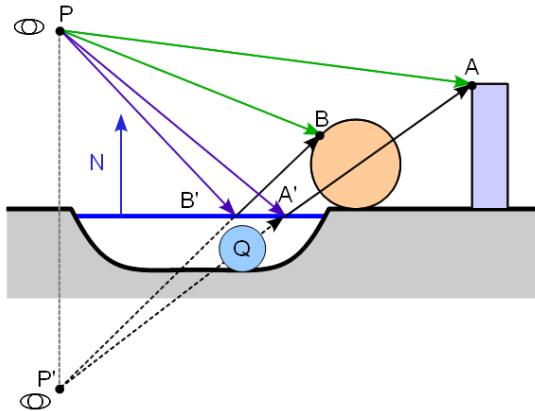
Nutno podotknout, že tento přístup neřeší viditelnost světla. Stejný přístup lze použít i pro difúzní osvětlení, ale předpočtené osvětlení je efektivnější.

## 3.7 Odlesky povrchů

Odlesky dodávají scéně lepší vzhled, dodávají dojem buď kovového nebo mokrého povrchu. V rasterizačním řetězci jsou odlesky drobě problematické, protože grafická karta vždy zpracovává pouze aktuální geometrii a nemá informace o okolí.

Standardně se odlesk realizuje dalším průchodem, ve kterém se vykreslí okolí do textury a tato textura se poté aplikuje na model s odleskem. U menších objektů, jako je například auto, se vykreslí okolí pomocí cube mapování do FBO. Výsledná textura se následně namapuje na objekt a tím se získá dojem lesklého povrchu.

V případě rovných povrchů je nutné použít jiný přístup, scéna se vykreslí do textury z pohledu pod povrchem, který je na obrázku 3.17 označen jako  $P'$ . Pohled snímá objekty jakoby z povrchu a při vykreslování na obrazovku se spočítá pozice daného pixelu povrchu ve vykreslené textuře a z té se aplikuje barva pixelu.



Obrázek 3.17: Diagram odrazu vodní hladiny [4]

Pohled  $P$  je MVP matici, pro získání pohledu  $P'$  potřebujeme vypočítat reflexní matici  $R$ , která převrátí pohled podle normály povrchu  $\vec{N}$  a parametru  $d$  spočítaného z rovnice roviny.

$$d = -(xN_x + yN_y + zN_z)$$

$$R = \begin{pmatrix} 1 - 2 \cdot \vec{N}_x^2 & -2 \cdot \vec{N}_x \cdot \vec{N}_y & -2 \cdot \vec{N}_x \cdot \vec{N}_z & -2 \cdot \vec{N}_x \cdot d \\ -2 \cdot \vec{N}_y \cdot \vec{N}_x & 1 - 2 \cdot \vec{N}_y^2 & -2 \cdot \vec{N}_y \cdot \vec{N}_z & -2 \cdot \vec{N}_y \cdot d \\ -2 \cdot \vec{N}_z \cdot \vec{N}_x & -2 \cdot \vec{N}_z \cdot \vec{N}_y & 1 - 2 \cdot \vec{N}_z^2 & -2 \cdot \vec{N}_z \cdot d \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Vynásobením matice  $R$  s maticí  $P$  získáme reflekční matici, která otáčí obraz o  $180^\circ$ , abychom získali korektní pohled, musíme provést ještě rotaci kolem osy Z.

$$P' = R \cdot P \cdot \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Při vykreslování reflexního pohledu je nutné provést ořez scény pod povrchem a nevykreslovat odvrácené trojúhelníky. Pro finální vykreslování si označím reflexní mapu jako  $RM$  a vykreslovaný vrchol jako  $\vec{v}$ . Barva odlesku  $color_r$  se spočítá dle níže uvedeného vzorce.

$$\vec{p} = P' \cdot \begin{pmatrix} \vec{v} \\ 1 \end{pmatrix}$$

$$color_r = RM[\vec{p}_x, \vec{p}_y]$$



# Kapitola 4

## Realizace

Realizace projektu byla provedena ve třech fázích. Nejdříve byla vytvořena verze pro PC bez předpočteného osvětlení, dále bylo provedeno portování projektu na Android a v poslední fázi jsem se zabýval předpočteným osvětlením a problémy s ním spojené.

### 4.1 Simulátor

Simulátor je navržen tak, aby fungoval na X11-Linuxu a Androidu. Znamená to výběr multiplatformních knihoven nebo ve výjimečných případech rozdílných implementací pro jednotlivé platformy. K vykreslování jsem vybral OpenGL ES 2.0 (jedná se o odlehčené OpenGL 3.0) a GLSL core verzi 1.0 (to zaručuje maximální možnou kompatibilitu).

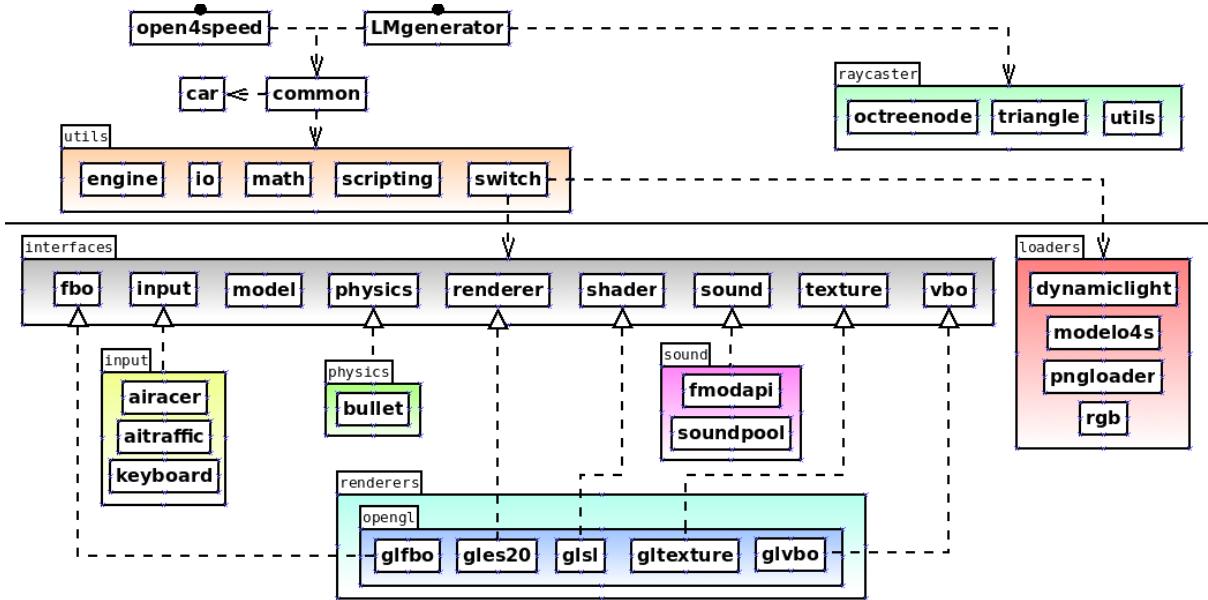
Fyzikální vlastnosti vozidel jsou realizovány pomocí enginu Bullet Physics (tentot engine používá mnoho známých firem zabývajících se vývojem mobilních her). Zvuk je realizován na mobilních zařízeních pomocí Android API a na Linuxu pomocí knihovny FMOD API. Správce oken je na Androidu třeba doprogramovat, aby fungovalo dotykové ovládání a po-zastavení simulátoru (např. při hovoru). Na Linuxu je použitá knihovna freeglut.

Samotný simulátor je rozdělen do dvou oddělených částí. První část je abstraktní a řeší logickou část projektu. Abstrakce je realizována pomocí rozhraní, které definují spo- lečné metody abstraktních objektů. Druhá část implementuje jednotlivá rozhraní a řeší také načítání dat ze souborů.

```
1  /// load models
2  skin = getModel(getConfigStr("skin_model", attributes), false);
3  wheel = getModel(getConfigStr("wheel_model", attributes), false);
4
5  /// set car wheels position
6  wheelX = getConfig("wheel_x", attributes);
7  wheelY = getConfig("wheel_y", attributes);
8  wheelZ1 = getConfig("wheel_back", attributes);
9  wheelZ2 = getConfig("wheel_front", attributes);
```

Ukázka 4.1: Načtení 3D modelu vozidla pomocí abstraktního přístupu

Důvodem oddělení částí je, že implementace jednotlivých rozhraní může být v budoucnu snadno nahrazena anebo mohou být použité různé implementace pro různé platformy. Příkladem různých implementací rozhraní pro různé platformy může být například renderer. V simulátoru je využita knihovna OpenGL, která například na platformě Windows Phone není dostupná a bylo by potřeba na této platformě použít Direct3D.



Obrázek 4.1: UML diagram projektu, horní část je abstraktní, dolní část implementuje jednotlivá rozhraní a načítá data do paměti, spustitelné soubory jsou označeny tečkou

Jak je vidět na obrázku 4.1, simulátor je rozdělen do dalších dílčích částí. Část `raycaster` a třída `LMgenerator` realizují předzpracování. Třída `open4speed` se zabývá správou okna a spouštěním dílčích součástí. `Common` uchovává veškeré objekty a proměnné, je to z důvodu, že jinak by v třídách `open4speed` a `LMgenerator` musel být obsah struktury `common` duplicitně.

Třída `car` je v první řadě struktura uchovávající veškeré proměnné realizující vozidlo, dále zajišťuje propojení mezi fyzikálním enginem, vykreslováním, zvukem, vstupem umělé inteligence a uživateli.

Část `utils` je balík příkazů tvořící simulátor.

- `engine` implementuje GUI a 3D scénu
- `io` obstarává práci se soubory
- `math` poskytuje sadu matematických příkazů pro práci s 3D objekty
- `scripting` interpretuje skriptování GUI
- `switch` obstarává objekty jednotlivých rozhraní a naplňuje je daty

Jednotlivá rozhraní mají různé funkce. Rozhraní `fbo`, `renderer`, `shader`, `texture` a `vbo` vytváří abstraktní přístup k OpenGL, rozhraní `input` je rozhraní pro ovládání vozidla (implementací může být buď klávesnice anebo umělá inteligence), rozhraní `physics` reprezentuje fyzikální engine a rozhraní `sound` umožňuje přehrávat zvuky. Implementace rozhraní `sound` jsou příkladem použití různých implementací na různých platformách.

## 4.2 Realizace na platformě Android

Platforma Android je postavená na Linuxovém jádře a využívá mnohé komponenty z projektu GNU, nicméně celé Android API je v jazyku Java za využití virtuálního stroje Dalvik. Pomocí Android NDK(native development kit) je možné zkompilovat C/C++ kód jako knihovnu, kterou je možné zavolat z jazyku Java. Toto je prováděno pomocí JNI(Java native interface).

Snažší by bylo naprogramovat simulátor přímo v Javě, ale tím bych ztratil možnost portovat simulátor na další platformy, které virtuální stroj Dalvik nemají. Dále by byl problém s použitím fyzikálního enginu, který je napsaný v jazyce C++. Musel bych napsat interface pro fyzikální engine, který by mi umožnil komunikaci mezi C++ a Java kódem.

### 4.2.1 Problematika Android API

Java native interface je rozhraní, které umožňuje spustit C++ kód z jazyku Java a obráceně. Největší problém je, že neexistuje použitelný port freeglut knihovny pro Android. Udělal jsem tedy JNI rozhraní, které odpovídá handlerům knihovny freeglut. Tedy nahrazuji veškeré funkce této knihovny svou implementací. Knihovna implementuje správu oken, vstup klávesnice a myši a volání idle a display funkcí. Klávesnice a myš byla pochopitelně plně nahrazena dotykovým ovládáním, ale pokud do Android zařízení připojíme klávesnici nebo myš, bude plně funkční. Funkčnost myši zařizuje sám systém, klávesnice stačila namapovat.

Aby bylo možné volat C++ metody přímo z Javy musí být pojmenovány ve formátu `Java_package_třída_metoda`. C++ knihovna bude tedy možné spustit pouze z jedné Java třídy.

```
1 void Java_com_cvut_o4s_O4SJN1_nativeInit(JNIEnv* e) { main(0, 0); }
2 void Java_com_cvut_o4s_O4SJN1_nativeKey(JNIEnv* e, jint i) { key(i); }
3 void Java_com_cvut_o4s_O4SJN1_nativeDisplay(JNIEnv* e) { display(); }
```

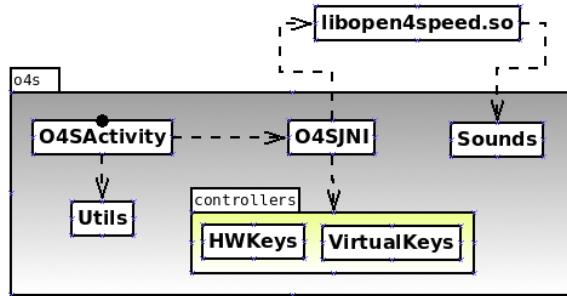
Ukázka 4.2: Volání metod pomocí JNI v C++ pro nahrazení freeglut knihovny

Dále je potřeba C++ knihovnu načíst pomocí volání `System.loadLibrary` a metody deklarovat v dané Java třídě pomocí klíčového slova `native`.

```
1 static native void nativeInit();
2 static native void nativeKey(int i);
3 static native void nativeDisplay();
```

Ukázka 4.3: Ukázka deklarování C++ metod v Java kódu

Dalším problémem je volání Android API z C++ kódu (tzn. volání Java kódu z C++). Příkladem je přehrávání zvuků pomocí Android API, ovládání zvuku je nutno řídit z C++, ale API je přímo přístupné pouze z Javy. Na obrázku 4.2 je znázorněn případ třídy Sounds. Problém se řeší také pomocí JNI, pouze volání jsou prováděny v opačném směru.



Obrázek 4.2: UML diagram spustitelného klienta pro Android, kde libopen4speed.so je zkomplikovaná C++ knihovna projektu

```

1 void soundpool::play(int index) {
2     jclass clazz = instance->FindClass("com/cvut/o4s/Sounds");
3     //get method "void soundPlay(int, int)"
4     jmethodID method = instance->GetStaticMethodID(clazz, "soundPlay", "(II)V");
5     instance->CallStaticVoidMethod(clazz, method, index, (int)looping);
6 }
```

Ukázka 4.4: Volání Java metody pro přehrání zvuku z C++ kódu

V Javě se poté musí ještě rozlišovat přehrávání zvuků od přehrávání hudby. Hudba bývá řádově náročnější na paměť a nemá smysl jí celou držet v paměti, když na ni není potřeba aplikovat další zvukové efekty.

```

1 public static void soundPlay(int index, int loop) {
2     //instance is a music
3     if (list.get(index).id == TYPE_MUSIC) {
4         music.setLooping(loop == 1);
5         music.start();
6     }
7
8     //audio clip
9     else {
10        Sound s = list.get(index);
11        snd.play(s.id, s.volume, s.volume, 1, loop, s.rate);
12    }
13 }
```

Ukázka 4.5: Přehrání zvuku v Javě pomocí Android API

### 4.2.2 Využití vláken

Většina zařízení, které dnes už spadají do střední třídy, je vybavena aspoň dvěma procesory, je proto vhodné rozdělit činnost do více vláken, aby bylo dosaženo maximálního výkonu. Třída GLSurfaceView, která umožňuje vykreslovat OpenGL scénu z C++ umožňuje volat OpenGL příkazy pouze z jednoho vlákna. Z tohoto důvodu jsem rozdělil kód na grafické a negrafické vlákno.

V menu simulátoru se používá pouze grafické vlákno, negrafické vlákno je primárně použito pro fyzikální výpočty, které zařízení vytěžují nejvíce. Vlákna jsou synchronizovaná, aby nedocházelo k použití transformace vozidla z předešlého snímku (to by bylo velmi rušivé).

Dále provádím regulaci rychlosti simulátoru v závislosti na výkonu hardwaru. Je potřeba, aby obě vlákna prováděla výpočet co nejvíce paralelně, pokud by negrafické vlákno příliš dlouho nedodávalo novou transformaci vozidla, docházelo by k trhanému vykreslování (některý snímek by se vykresloval víckrát a to by se projevilo tak, že by se snímková frekvence zdála být na krátký okamžik poloviční). Pokud k tomuto případu dojde, je vhodné snížit frekvenci volání obou vláken a tím snížit rychlosť simulátoru.

Problém je, že nelze snižovat rychlosť simulátoru do nekonečna, proto je zde nastavena minimální hodnota, na kterou se může simulátor zpomalit. Je zde i horní limit, aby nedocházelo k zrychlené jízdě a tedy i ke zvýšené spotřebě energie.

### 4.2.3 Práce se soubory

V Androidu se aplikace publikují v APK balíčku. Jedná se o ZIP soubor, který má nějakou danou strukturu. V Java kódu je možné na veškeré soubory uvnitř balíčku přistupovat, v C++ tato možnost není. Někteří vývojáři toto řeší, že po prvním spuštění si aplikace stáhne ze serveru přídavná data a ty si uloží například na paměťovou kartu. Lze také rozbalit soubory z APK balíčku a s těmi pak v C++ pracovat. To se moc nepoužívá, protože jsou potom data v zařízení dvakrát.

Další možnou variantou je přistupovat k APK balíčku jako k ZIP souboru. Knihovna libZIP umožňuje zpracovávat data ze ZIP souboru jako kdyby byla uložena normálně na disku nebo na paměťové kartě. To má výhodu, že data jsou komprimovaná a zároveň přístupná.

<pre> 1 void gets(char* line , zip_file* f) { 2     for (int i=0; i&lt;MAX_LENGTH; i++) { 3         zip_fread(f, character , 1); 4         line[i] = character[0]; 5         if (line[i] == '\n') { 6             line[i + 1] = '\000'; 7             return; 8         } 9     } 10 }</pre>	<pre> void gets(char* line , FILE* f) {     for (int i=0; i&lt;MAX_LENGTH; i++) {         fread(character , 1, 1, f);         line[i] = character[0];         if (line[i] == '\n') {             line[i + 1] = '\000';             return;         }     } }</pre>
--	--

Ukázka 4.6: Čtení řádky z APK balíčku (vlevo) a ze souboru (vpravo)

Problém může nastat při použití externích knihoven, které čtou ze souborů. To je případ knihovny libPNG, která načítá ze souborů PNG rastrová data. Tato knihovna umožňuje použití vlastní čtecí funkce a tím umožňuje číst data přímo z APK, kdyby tomu tak nebylo, nemohli bychom touto knihovnou načítat data přímo z APK balíčku.

```

1 void png_read(png_structp png_ptr, png_bytep data, png_size_t length) {
2 #ifdef USE_APK_PACKAGE
3     zip_fread(zipFile, data, length);
4 #else
5     fread(data, length, 1, file);
6 #endif
7 }
8
9 Texture* loadPNG(const char* filename) {
10 #ifdef USE_APK_PACKAGE
11     zipFile = zip_fopen(APKArchive, prefix(filename), 0);
12 #else
13     file = fopen(prefix(filename), "rb");
14 #endif
15     png_set_read_fn(png_ptr, NULL, png_read);
16     ...
17 }
```

Ukázka 4.7: Použití vlastní čtecí funkce při použití knihovny libPNG

### 4.3 Předpočtené osvětlení

Předpočtené osvětlení umožňuje velmi rychle zobrazovat stínování a stíny na statických objektech. Problémem je, že je potřeba aplikovat tento efekt i na dynamických objektech. Úroveň osvětlení lze přečíst z nejbližšího statického objektu, tedy za předpokladu, že budeme mít tuto hodnotu někde k dispozici. Tuto hodnotu lze mít uloženou v alfa kanálu aktuálního snímku a jen jí přečíst před tím, než se vykreslí dynamické objekty.



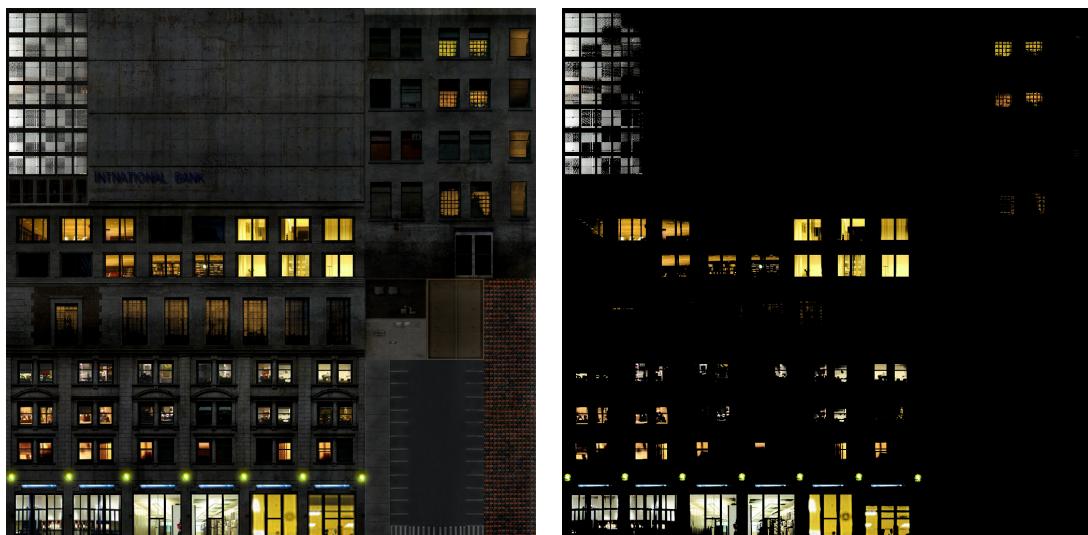
Obrázek 4.3: Scéna s aplikovanými mapami osvětlení lamp

#### 4.3.1 Vytváření map osvětlení

Vytváření map osvětlení je realizováno dvěma přístupy, první je rasterizační (pomocí OpenGL) a druhý je vrhání paprsku. Pro rasterizační přístup jsou k dispozici dva druhy světel: bodové světlo a reflektor. Tyto světla jsou definovaná přímo v 3D modelovacím software. Dosah světel je omezen na 300metrů (je to z důvodu chyb, které vznikají u osvětlení na velkou vzdálenost). Světla pochopitelně nepoužívají spekulární složku, protože tato složka je závislá na pohledu kamery a tudíž jí nelze předpočítat.

I maximální možné rozlišení map osvětlení pro použitou scénu není dostatečné a stíny jsou kostičkovány. Z tohoto důvodu se používají dva druhy filtrování. Prvním je rozmazání map osvětlení, které se provádí přímo po generování mapy osvětlení a druhým je lineární filtrování, které se aplikuje až při aplikování mapy osvětlení. Tato kombinace filtrování značně zvýší výslednou kvalitu stínů.

V přístupu využívající vrhání paprsku jsou navíc k dispozici plošné zdroje světla, které jsou definovány pomocí osvětlovacích textur, což jsou textury, u kterých jsou vykresleny pouze texely vyzařující světlo. Osvětlovací textury mají stejné texturovací souřadnice jako difúzní textury 3D modelu. Plošný zdroj světla se realizuje tak, že každý barevný texel v osvětlovací textuře vygeneruje bodový zdroj světla. Množina vzniklých bodových světel pak tvoří plošný zdroj světla.

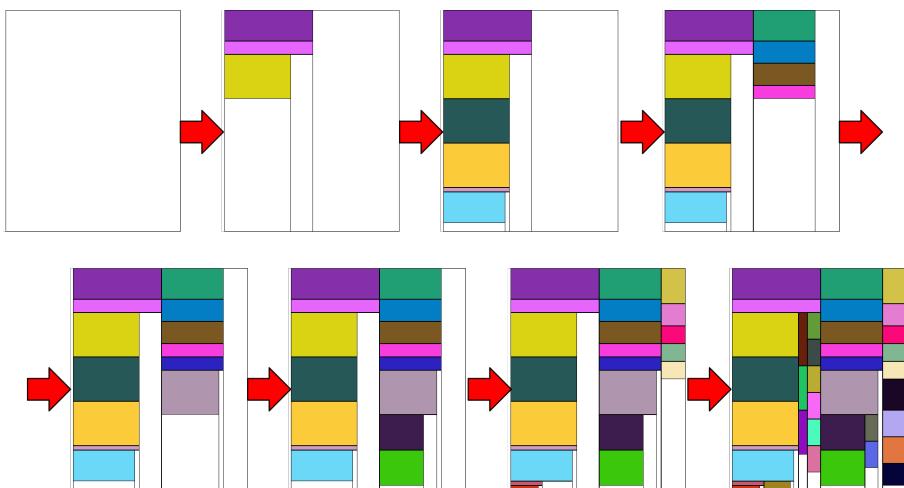


Obrázek 4.4: Ukázka difúzní textury modelu (vlevo) a osvětlovací textury (vpravo)

#### 4.3.1.1 Generování texturovacích souřadnic pro mapy osvětlení

V teoretické části byl popsán postup generování texturovacích souřadnic pomocí ILP. V praxi toto řešení nelze použít, protože jeho výpočetní složitost  $f(m, n) \in O(m^2 \cdot n^2)$ ,  $m$  je počet trojúhelníků a  $n$  počet texelů v lightmapě. Pomocí ILP se tedy dají řešit pouze velice malé instance problému. V této práci problém řeší pomocí metody Packing Lightmaps [7], což je metoda řešící problém batohu za použití datové struktury kD stromu, která dělí  $k$ -rozměrný objekt půlením.

V případě Packing Lightmaps dělíme rovinu horizontálním nebo vertikálním řezem, kD strom je v podstatě binární strom, jehož kořen reprezentuje  $k$ -rozměrný objekt (v našem případě mapu osvětlení). Vnitřní uzly jsou dělící, určují podle které osy se řez povede a je v nich zapsána pozice řezu. Listy stromu reprezentují objekty uložené ve struktuře, v této práci AABB - tedy dva párové trojúhelníky (viz kapitola 3.6.1).



Obrázek 4.5: Ukázka naplnění kD stromu pomocí metody Packing Lightmaps

Metoda Packing Lightmaps nejdříve seřadí AABB podle jejich plochy a postupně je vkládá do kD stromu. Při každém vložení se nejdříve hledají neobsazené listy, zda neexistuje takový, který by odpovídal rozměrům AABB. Pokud takový neexistuje pokusí se algoritmus nalézt takový neobsazený list, kterému odpovídá aspoň jeden rozměr. V případě úspěšného hledání se AABB vloží do listu, v ostatních případech se najde větší list, provede se jeho rozdělení a následně se AABB do jednoho z potomků vloží.

```

1 public boolean addAABB(AABB aabb) {
2     // check if current node is empty and has enough space for AABB
3     if (isEmpty() && (aabb.width <= width) && (aabb.height <= height)) {
4         // fits exactly
5         if ((aabb.width == width) && (aabb.height == height)) {
6             child1 = new KDNode(aabb);
7             return true;
8         }
9         // subdivision
10        else {
11            child1 = new KDNode(aabb.width, height);
12            if (child1.addAABB(aabb))
13                return true;
14        }
15    }
16    // try to insert into first child
17    else if (child1.addAABB(aabb))
18        return true;
19
20    // create second child
21    else if (child2 == null) {
22        // horizontal subdivision
23        if ((aabb.width <= width) && (aabb.height <= getChild2Height())) {
24            child2 = new KDNode(width, height - aabb.height);
25            if (child2.addAABB(aabb))
26                return true;
27        }
28        // vertical subdivision
29        if ((aabb.width <= getChild2Width()) && (aabb.height <= height)) {
30            child2 = new KDNode(width - child1.width, height);
31            if (child2.addAABB(aabb))
32                return true;
33        }
34    }
35    // try to insert into second child
36    else if (child2.addAABB(aabb))
37        return true;
38    return false;
39}

```

Ukázka 4.8: Vkládání AABB do kD stromu za využití rekurze

#### 4.3.1.2 Generování map osvětlení

Generování map osvětlení znamená v podstatě vykreslení celé scény do souřadnic map osvětlení (tím se vykreslí celá scéna, aniž by docházelo k zakrývání objektů). Generování je v práci řešeno dvěma různými přístupy. Prvním přístupem je rasterizační (pomocí OpenGL) a druhým je vrhání paprsku. V obou případech se provede výpočet difúzního osvětlení stejným způsobem jako se provádí při běžném vykreslování scény.

Aby mapa osvětlení obsahovala stíny, musí se provést test viditelnosti jednotlivých bodů pro jednotlivá světla. V rasterizačním přístupu se toto provádí pomocí stínových map (viz. kapitola 3.6.3).

V druhém přístupu se pro každý zdroj světla vrhají paprsky do všech bodů v mapě osvětlení (paprsky mají definováno, kde končí). Pokud paprsek protne mezi počátkem a destinací nějaký trojúhelník, nepřipočte se pro bod v osvětlovací mapě příspěvek z daného zdroje světla.

Pokud bychom testovali pro každý paprsek existenci průsečíku s nějakým trojúhelníkem ve scéně, bylo by generování příliš pomalé. Z toho důvodu jsem použil akcelerační datovou strukturu octree. Octree je datovou stromovou datovou strukturou, která dělí rovnoramenně prostor vždy na osm stejných podprostorů. V mé realizaci je rozměr každého uzlu určený pomocí AABB a obsahuje seznam trojúhelníků, který tomuto uzlu náleží.

Stavba stromu octree se provádí odshora dolů, to znamená, že nejdříve se všechny trojúhelníky vloží do kořenového uzlu a po té se provede jejich přesun do nižších uzlů, pokud jsou splněna následující kritéria.

- dochází k průniku AABB trojúhelníka a AABB aktuálního uzlu
- počet trojúhelníků v aktuálním uzlu je vyšší než 20 (v takovém případě je neefektivní strom dále větvit)

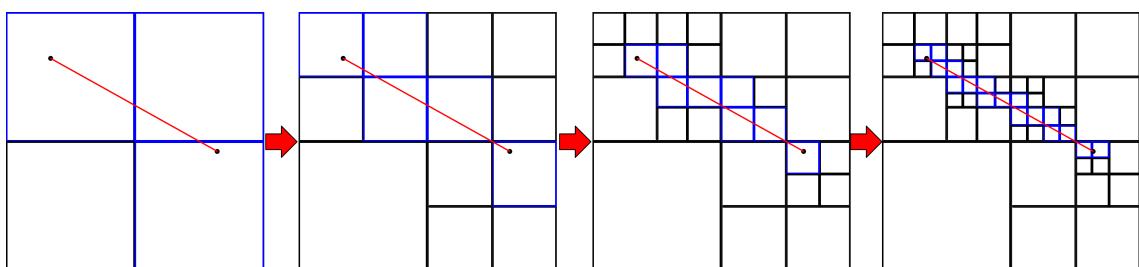
```

1 bool TestAABBAABB(AABB* a, AABB* b) {
2     if (a->max[0] < b->min[0] || a->min[0] > b->max[0]) return false;
3     if (a->max[1] < b->min[1] || a->min[1] > b->max[1]) return false;
4     if (a->max[2] < b->min[2] || a->min[2] > b->max[2]) return false;
5     return true;
6 }
```

Ukázka 4.9: Průnik dvou AABB ve 3D

Po sestavení stromu máme tedy rozdělené trojúhelníky do segmentů, které jsou definovány pomocí AABB. Abychom nalezli průsečík paprsku s nějakým trojúhelníkem ve scéně, musíme strom procházet. Nejdříve provedeme test průniku paprsku s AABB potomků kořene. U každého potomka, u kterého byl test pozitivní, provedeme test jeho potomků. Tímto způsobem pokračujeme dokud se nedostaneme na list. V každém uzlu, u kterého byl test pozitivní, provedeme také test průsečíku se všemi trojúhelníky, které má v seznamu. Pokud test jakéhokoliv trojúhelníku je pozitivní, vrátíme test paprsku jako pozitivní.

Ukázka rekurzivního hledání v octree je na obrázku 4.6 (na obrázku je pouze quadtree, což je obdoba octree ve 2D).



Obrázek 4.6: Ukázka hledání průsečíku paprsku v datové struktuře quadtree

Aby hledání ve stromu bylo ještě rychlejší, lze použít ještě několik optimalizací. Může se například ukládat  $n$  posledních průsečíků a testovat, jestli další paprsek nemá stejný průsečík se stejným trojúhelníkem jako předešlý paprsek. Tato optimalizace má smysl pouze pro malé  $n$ . V mému projektu jsem nastavil  $n = 2$  a testuji paprsky, které mají cílový trojúhelník stejný, vždy za sebou.

Dále je vhodné přeskakovat testy stejných trojúhelníků. Podle zvoleného kritéria dělení stromu při jeho stavbě může dojít k tomu, že některé trojúhelníky jsou uloženy ve stromu víckrát (pokud jejich AABB proniká s více uzly stejné hloubky). Řešit to lze jednoduše. Stačí si u trojúhelníku vždy pamatovat ID paprsku, s kterým byl proveden poslední test průsečíku. Díky tomu lze zjistit, jestli některý paprsek není testován víckrát a případně tento test přeskočit.

Další možnost urychlení generování map osvětlení je přeskakování testování paprsků, u kterých je délka paprsku delší než vzdálenost, na kterou dosvítí zdroj světla. Tuto vzdálenost lze spočítat pomocí vzorce pro útlum (viz. kapitola 3.2). Minimální přípustnou hodnotu si označím  $\epsilon$  a maximálně délku paprsku jako  $d$ .

$$F_a = \frac{1}{k_c + k_l \cdot d + k_q \cdot d^2}, F_a = \epsilon$$

Po dosazení a otočení rovnice získáme výraz

$$k_c + k_l \cdot d + k_q \cdot d^2 = \frac{1}{\epsilon}$$

Kvadratickou funkci spočteme pomocí diskriminantu

$$D = k_l^2 - 4 \cdot k_q \cdot (k_c - \frac{1}{\epsilon})$$

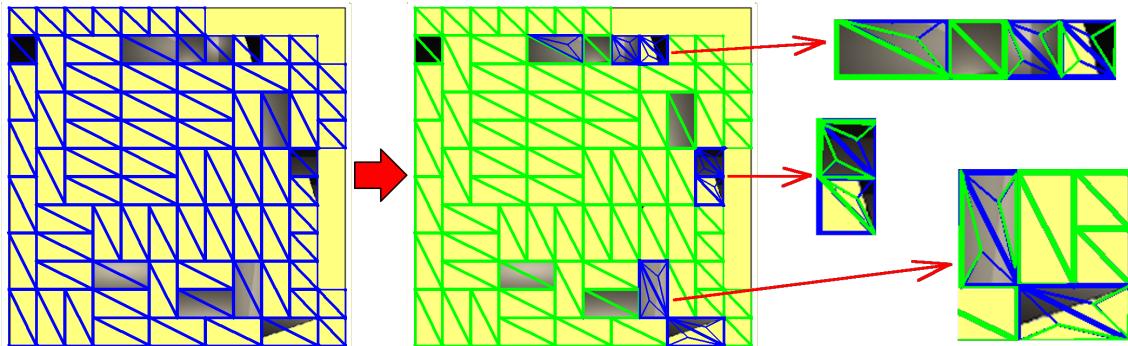
$$d = \frac{-k_l \pm \sqrt{D}}{2 \cdot (k_c - \frac{1}{\epsilon})}$$

### 4.3.2 Příprava dat pro dynamickou aktualizaci map osvětlení

Data pro dynamickou aktualizaci map osvětlení se nazývají záplaty. Záplata je v podstatě také mapa osvětlení, ale pouze pro jeden zdroj světla. Problém je, že tyto záplaty by byly příliš paměťově náročné, proto používám vektorovou podobu těchto dat.

Data záplaty jsou při běhu simulátoru uloženy jako trojúhelníky ve VBO (tedy na GPU). Každý vrchol má dva atributy a to pozici v mapě osvětlení a barvu světla vynásobenou intenzitou. Lze tedy snadno a rychle mapy osvětlení aktualizovat, pokud máme záplaty v této podobě uloženy.

Abychom získali záplaty v této podobě musíme nejdříve vykreslit pro každý dynamický zdroj světla vykreslit mapu osvětlení a poté tyto data zvektorizovat. Pro zvektorizování lze využít informaci o souřadnicích jednotlivých trojúhelníků v mapě osvětlení. V prvním kroku se přečtu informace o osvětlení ve vrcholech trojúhelníků. Poté se provede interpolace těchto hodnot (vždy pouze v rámci jednoho trojúhelníku) a zkонтroluje se, zda hodnota s nějakou tolerancí odpovídá. Pokud ano, přidá se geometrie trojúhelníku do vektorových dat záplaty. Pokud ne, rozdělí se trojúhelník podle jeho středu a provádí se znova stejný postup.



Obrázek 4.7: Ukázka převodu záplaty osvětlovací mapy do vektorové podoby, vlevo původní mapa osvětlení, uprostřed první iterace algoritmu a vpravo další iterace

#### 4.3.3 Statické osvětlení a dynamické objekty

Pokud máme ve scéně předpočtené stíny, je vhodné tyto statické stíny aplikovat i na dynamické objekty. V případě vozidla lze přečíst intenzitu osvětlení z některých bodů na silnici pod vozidlem. Intenzitu osvětlení lze jednoduše ukládat do alfa kanálu textury, ve které se nachází aktuální scéna.

Výpočet pozice bodu na obrazovce se provádí na CPU. Fyzikální engine vrací pozici středu vozidla  $\vec{p}$ , tu lze vynásobit maticí  $MVP$  a po perspektivním dělením získáme pozici vozidla na obrazovce označenou jako  $sp$ .

$$\begin{aligned} s &= MVP \cdot \vec{p} \\ sp &= s/s_w \end{aligned}$$

Pozice je v rozmezí  $< -1, 1 >$ . Pro čtení z textury je třeba pozici přepočítat do rozmezí  $< 0, 1 >$ .

$$sp' = sp \cdot 0.5 + 0.5$$

Tím jsme získali pozici vozidla, pokud chceme pozici pod vozidlem, musíme vypočítat pozici o něco niž. Úroveň posunu označíme jako  $a$ . Velikost posunu je dále závislá na vzdálenosti vozidla od kamery. Vzdálenost je uložena v proměnné  $s_z$ .

$$\begin{aligned} sp''_x &= sp'_x \\ sp''_y &= sp'_y - a/s_z \end{aligned}$$

Tím jsme získali pozici texelu nesoucí informaci o intenzitě osvětlení. Vzniká zde ale problém s barevným světlem. Jsou zde dvě možnosti řešení. Prvním je přidání další textury, ve kterém by byla barevná intenzita světla (tím by došlo ke zvýšení náročnosti na hardware a ke zvýšení paměťové náročnosti). Druhým možným řešením je omezit barevná světla do nějaké sytosti (pokud se slabě zabarvené světlo aplikuje jako bílé, není to totiž rušivé).

Dalším problémem je, že dynamické objekty (tedy vozidla) by měly zastínit statický model. Tento problém lze řešit opět pomocí screen-space přístupu. Ve screen-space textuře si můžeme označit texely, na kterých je vykreslené vozidlo. V době vykreslování vozovky se vždy díváme o několik texelů výše, než je aktuální vykreslovaný texel, a pokud je texel označen, ztmavíme aktuálně vykreslovaný fragment. Důležitou poznámkou je, že texely kol vozidla se neoznačují (tím by vznikal chybný stín).

Stín je kvalitou obdobný jako projektivní stín. Bylo by možné zjišťovat pozice světel, aby vznikl měkký stín, ale tím by se simulátor značně zpomalil. Při rychlém otáčení lze zpozorovat chybu, která vzniká kvůli použitému screen-space přístupu. V noční scéně to moc není vidět, nicméně ve světlých scénách by to mohlo být značně rušivé.

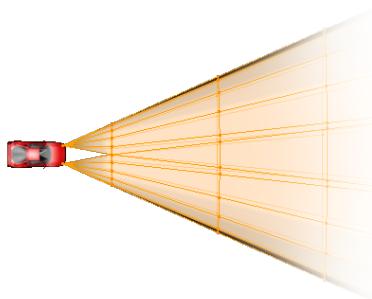


Obrázek 4.8: Vržený stín vozidla

#### 4.3.4 Světla vozidel

Dle předběžných měření jsem zjistil, že na mobilních zařízeních lze za běhu počítat jedno až dvě světla (bez řešení viditelnosti světla). U vozidla ovládané hráčem není potřeba řešit viditelnost světla, prakticky zde nedochází k situacím, že by světlo osvětlovalo neviditelné objekty (je to z důvodu těsné vazby světla hráčova vozidla a kamery).

U ostatních vozidel je třeba osvětlení zjednodušit, aby to mobilní hardware zvládal. Zvolil jsem metodu vykreslování polopruhledných kuželů do scény, k tomu je použita OpenGL funkce blending, která umožňuje kreslit objekty s částečnou průhledností. Je zde počítán útlum světla stejně jako u standardních světel.



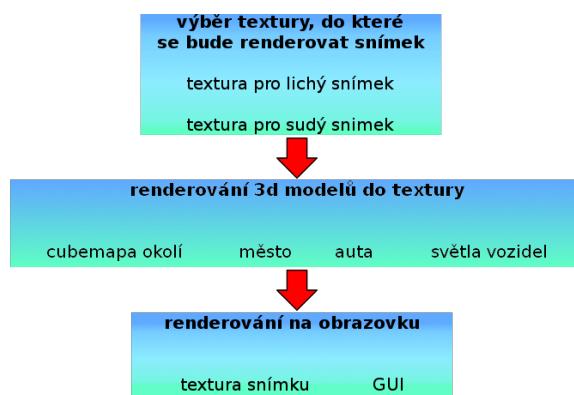
Obrázek 4.9: Ukázka světel pomocí polopruhledných kuželů

## 4.4 Screen-space přístup

Při běhu jakéhokoliv simulátoru je největší zátěž mobilního hardware způsobena vykreslováním 3D scény, proto je nutné tuto část co nejvíce zefektivnit. Pro mnoho efektů se používají takzvané víceprůchodové přístupy (scéna se vykreslí víckrát z různých pohledů nebo pomocí různých shaderů), ale na mobilním zařízení je zatím z hlediska výkonu možné použít pouze jednopruhodový přístup.

Abych částečně nahradil ve svém simulátoru víceprůchodový přístup, použil jsem screen-space přístup (to je obecně jakákoliv technika využívající data z pohledu kamery, jinými slovy máme pouze výsledný snímek a na něm provádime další operace). V praxi se používá hlavně pro zrcadlové plochy. Potřeboval jsem přistupovat do aktuálně vykreslovaného snímku. To je ale problém, protože tento snímek není kompletní a kdybych vykresloval přímo do něj a zároveň z něj četl způsobilo by to chyby ve výsledném obraze.

Řešením je vykreslování střídavě do dvou různých textur. To znamená, že při lichém snímku kreslím do textury 1 a čtu z textury 2. Při sudém snímku je tomu obráceně. Tímto způsobem získám přístup do vykreslené scény, sice s drobným zpožděním, ale při dostatečně rychlém vykreslování si toho uživatel nevšimne (za předpokladu, že textura bude použita na efekty jako jsou odlesky nebo stíny).



Obrázek 4.10: Vykreslovací řetězec simulátoru

#### 4.4.1 Rozměr textur na OpenGL ES 2.0

Verze OpenGL ES 2.0 umožňuje používat obdélníkové textury pouze omezeně. Textury tedy musí být čtvercové a jejich strana musí být o velikosti mocniny 2. Pro vykreslování do textury, která bude výsledně umístěna na obrazovku je tedy nutné zvolit vyšší rozlišení. Např. u tabletu Google Nexus 7 2012 s rozlišením 1280x720 se zvolí textura 2048x1024. To znamená, že by se ve výsledku vykreslilo víc než 2x více pixelů než je potřeba.

Není ale nutné vykreslovat do celé textury, pomocí příkazu *glViewport* se dá nastavit část textury, do které má OpenGL vykreslovat a nedojde tedy k plýtvání výkonem.

Dále je možné zvolit nižší rozlišení viewportu než je rozlišení displeje. Mobilní zařízení obvykle disponují obrovskou hustotou pixelů a pokud zařízení nebude stíhat vykreslovat scénu v plném rozlišení, lze vykreslovat v rozlišení menším. U počítačových 3D her se také setkáváme s možností změnit rozlišení, tam řeší změnu přímo hardware displeje nebo jeho ovladače.

#### 4.4.2 Motion-blur

Jedním z velmi rychlých efektů ve screen-space je motion-blur. Jedná se o efekt rozmáznutí obrazu rychlým pohybem. Motion-blur se používá ve většině závodních simulátorů, rozmáznutí dodává dojem rychlosti.

```

1 uniform sampler2D color_texture , pFrm;
2 uniform float res , speed;
3 varying vec2 v_Coords;
4
5 void main() {
6     //apply color texture
7     gl_FragColor = texture2D(color_texture , v_Coords);
8
9     //apply motion blur
10    gl_FragColor *= (1.0 - speed);
11    gl_FragColor += speed * texture2D(pFrm, gl_FragCoord.xy * res);
12 }
```

Ukázka 4.10: Motion blur fragment shader dle rychlosti vozidla

*Speed* je rychlosť vozidla od 0 do 1, *pFrm* je textura předchozího snímku a *res* je 1 / plné rozlišení textury (pro jednoduchosť je použitý čtvercový rozměr textury).

Tento kód je použit přímo při renderování modelů, nikoliv při postprocesu. Výhodou je, že rozmáznutí se projeví po několika snímků za sebou pomocí jednoho čtení textury navíc. Nevýhodou je, že tento kód musí být vložen do každého shaderu vykreslující 3D model.

#### 4.4.3 Odlesky

Dále je možné ve screen-space řešit odlesky. Existuje obecný (složitý) postup jak tyto odlesky vypočítat, který by mobilní hardware nezvládl. Problém jsem rozdělil podle typu ploch. Vzhledem k tomu, že v simulátoru dochází pouze k rotaci pohledu podle osy y (rotace yaw), je možné řešit pro odlesky pro různé tvary povrchů zvláště.

Abychom vytvořili odlesk povrchu vozovky optimálně, potřebovali bychom informaci o hloubce scény. To by bylo opět výpočetně náročné a proto jsem zvolil jednodušší postup. Odlesk funguje pouze pro jeden úhel pohledu mezi vozovkou a směrovým vektorem kamery.

Na úrovni přední řezné roviny kamery je horizont povrchu přesně v polovině obrazovky. Pokud budeme tedy mít normálu povrchu směřující kolmo nahoru, jsme schopni realizovat odlesk pomocí převrácení aktuální vykreslovací textury. Pro souřadnice aktuálně vykreslovaného fragmentu  $[f_x, f_y, f_z]$  vypočteme souřadnice odlesku  $r$  převrácením y hodnoty. Tím získáme odlesk na úrovni přední řezné roviny.

$$r = [f_x, 1 - f_y]$$

S rostoucí vzdáleností od přední řezné roviny se lineárně posouvá horizont směrem vzhůru. Nalezením konstanty  $a$  získáme souřadnice odlesku  $r'$  pro kolmý povrch limitovaný úhlem pohledu kamery.

$$r' = [f_x, 1 - f_y + f_z \cdot a]$$

Dále je potřeba podporovat i nekolmé povrchy. To se provede nalezením další konstanty  $b$ , která bude měnit lineární posun odlesku podle normály povrchu  $\vec{n}$ . Tím získáme souřadnice odlesku  $r''$  limitované pouze úhlem pohledu kamery.

$$r'' = [f_x, 1 - f_y + f_z \cdot a + f_z \cdot (1 - \vec{n}_y) \cdot b]$$



Obrázek 4.11: Odlesky na povrchu vozovky

Pro můj zvolený pohled jsem zvolil konstanty  $a = 0.1; b = 2$ . Odlesk vizuálně odpovídá a vykresluje se velice rychle. Problém je při nízké snímkovací frekvenci, kdy dochází k posunu odlesku (jak je vidět na obrázku 4.11). Posun není nijak výrazný, ale pokud se na něj soustředíme, vidíme jej. Dále při mísení efektů odlesků s motion-blurem dochází k rozmažání odlesku více než původního obrazu. Motion-blur je pro odlesky dvojnásobný, protože se aplikuje jak na původní objekt, tak na zrcadlový povrch.

U vozidel by se optimálně měl odlesk řešit pomocí víceprůchodového algoritmu za využití cube mapování. Vzhledem k tomu, že potřebuji maximální možný výkon a Android zařízení mají typicky malý displej, použil jsem jednodušší variantu.

Souřadnice odlesku  $r$  u vozidel se provádí pouze podle normály. Pro výpočet pozice odraženého obrazu se vezme střed vozidla  $s$  v souřadnicích kamery a přičte se k němu normála  $\vec{n}$  vynásobena určitým faktorem  $f$  (to funguje pouze u vozidla, které uživatel ovládá).

$$r = [s_x + \vec{n}_x \cdot f, s_y + \vec{n}_y \cdot f]$$



Obrázek 4.12: Ukázka výsledných odlesků na vozidle, vlevo bez odlesků, vpravo s odlesky



# Kapitola 5

## Testování

Testování probíhalo na smartphonu, tabletu a laptopu, které jsou blíže popsány v tabulce 5.1. Zařízení byla během testování plně nabita, byla připojena ke zdroji napájení a byly vypnuty volby úspory energie. Veškeré testy byly prováděny v režimu „free ride“ a za stejných podmínek.

	Samsung Galaxy S3 mini (mobil)	Google Nexus 7 2012 (tablet)	Acer TravelMate P253-e (laptop)
Procesor	NovaThor U8500	ARM Cortex-A9	Intel 1005M
Počet jader	2	4+1*	2
Frekvence CPU	1.0GHz	1.3GHz	2.4GHz
Operační paměť	1GB	1GB	4GB
Grafická karta	ARM Mali-400 MP	Nvidia Tegra 3	Intel HD Graphics
Rozlišení displeje	800x480	1280x800	1366x768
Operační systém	Android 4.1	Android 4.4	Kubuntu 13.10 32bit

\*tablet má extra procesor pro nižší spotřebu během nečinnosti

Tabulka 5.1: Parametry zařízení, na kterých jsem projekt testoval

## 5.1 Generování map osvětlení

### Bodové zdroje světla

Nastavení	Čas vykreslování
OpenGL bez PCF	2.25min
OpenGL s PCF8x8	19.267min
raycasting	28.176min

Tabulka 5.2: Vykreslování 106 bodových zdrojů světla pomocí OpenGL a pomocí raycastingu

### Plošné zdroje světla

Nastavení	Čas vykreslování
kvadratický útlum 0.2, optimalizováno	10.276s
kvadratický útlum 0.1, optimalizováno	19.483s
kvadratický útlum 0.05, optimalizováno	36.875s
kvadratický útlum 0.05, neoptimalizováno	359.796s
lineární útlum 0.05, neoptimalizováno	960.546s
lineární útlum 0.05, spot effekt 0.5, neoptimalizováno	12874.661s

Tabulka 5.3: Vykreslování několika plošných zdrojů světel navzorkovaných na 202 bodových světel, optimalizace spočívá v testování paprsků vždy mezi dvěma trojúhelníky a detekování, zda bod na trojúhelník dosvítí

Nastavení	Čas vykreslování
kvadratický útlum 0.1, optimalizováno	4.925h
kvadratický útlum 0.05, optimalizováno	9.578h

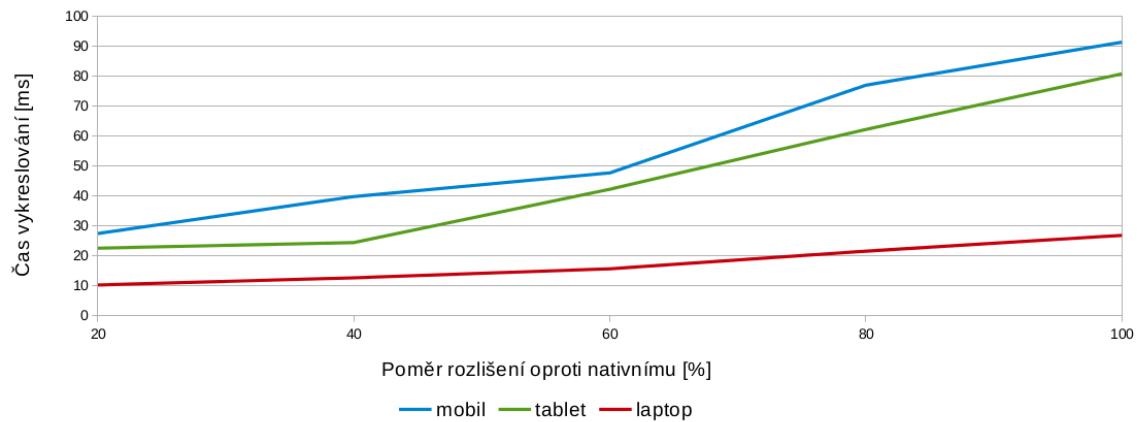
Tabulka 5.4: Vykreslování všech plošných zdrojů světel navzorkovaných na 372854 bodových světel, optimalizace spočívá v testování paprsků vždy mezi dvěma trojúhelníky a detekování, zda bod na trojúhelník dosvítí

## 5.2 Simulátor

V prvním testu zjišťuji, jak je vykreslování rychlé při různém počtu vykreslovaných fragmentů. Jedná se v podstatě o snížení rozlišení displeje k zvýšení výkonu.

	Samsung Galaxy S3 mini (mobil)	Google Nexus 7 2012 (tablet)	Acer TravelMate P253-e (laptop)
0.2x (poor)	27.3ms - 29.9ms	22.4ms - 28.4ms	10.1ms - 10.6ms
0.4x (low)	39.7ms - 42.7ms	24.3ms - 33.2ms	12.5ms - 13.0ms
0.6x (normal)	47.6ms - 59.9ms	42.1ms - 45.4ms	15.5ms - 15.9ms
0.8x (high)	76.9ms - 85.1ms	62.1ms - 64.4ms	21.4ms - 21.7ms
1.0x (ultra)	91.3ms - 97.9ms	80.7ms - 81.8ms	26.7ms - 27.0ms

Tabulka 5.5: Závislost rychlosti vykreslování scény na zařízení a poměru rozlišení oproti nativnímu (v závorce je uvedena konfigurace v nastavení simulátoru). Z každého měření je uvedená minimální a maximální hodnota



Obrázek 5.1: Závislost rychlosti vykreslování scény na zařízení a poměru rozlišení oproti nativnímu

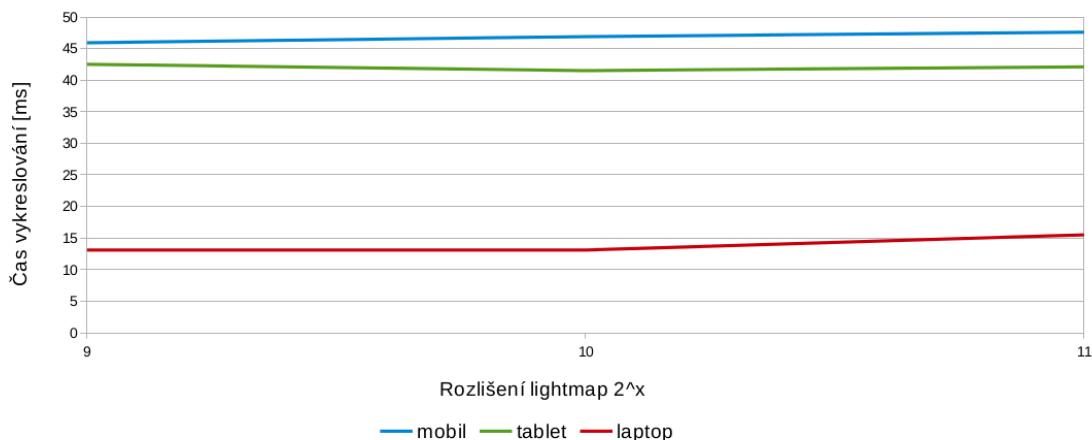
Z výsledku je patrně vidět výkonový rozdíl mezi laptopem proti tabletu a mobilu. Laptop je přibližně 3x až 4x rychlejší při použití srovnatelného rozlišení, což je menší rozdíl než jsem očekával.

Dále jsem testoval závislost rozlišení textury mapy osvětlení a rychlosti vykreslování. Naměřené údaje jsou v tabulce 5.3.

Z měření vyplývá relativně malá závislost na rozlišení textury. Pokud pominu nepřesnost v měření, jedná se o rozdíl přibližně 2ms u všech zařízení.

	Samsung Galaxy S3 mini (mobil)	Google Nexus 7 2012 (tablet)	Acer TravelMate P253-e (laptop)
512x512	45.9ms - 49.7ms	42.5ms- 44.4ms	13.1ms - 15.6ms
1024x1024	46.9ms - 48.6ms	41.5ms - 45.4ms	13.1ms - 13.3ms
2048x2048	47.6ms - 59.9ms	42.1ms - 45.4ms	15.5ms - 15.9ms

Tabulka 5.6: Závislost rychlosti vykreslování scény na platformě a rozlišení textur map osvětlení při vizuálních detailech normal(poměr 0.6x ku nativnímu rozlišení). Z každého měření je uvedená minimální a maximální hodnota



Obrázek 5.2: Závislost rychlosti vykreslování scény na platformě a rozlišení textur map osvětlení při vizuálních detailech normal

V dalším testu jsem se zaměřil na paměťovou náročnost. Tento test bohužel nebylo možné provést na platformě Android, ale zřejmě bych došel ke stejnemu výsledku pro obě platformy.

	Paměť	Sdílená paměť
512x512	29780kB	35424kB
1024x1024	29776kB	56928kB
2048x2048	29776kB	159328kB

Tabulka 5.7: Závislost využití paměti na rozlišení textur map osvětlení. Testováno na Acer TravelMate P253-e (laptop)

Nakonec jsem otestoval kompatibilitu s dalšími zařízeními, v testu uspěl tablet Google Nexus 7 2013, Samsung Galaxy S4 a smartTV set-top-boxu eGreat U8. V případě set-top-boxu bylo nutné ještě implementovat ovládání hardwarovými tlačítky. Další zařízení nebyla testována. Uvedená zařízení byla testována pouze na kompatibilitu, měl jsem je zapůjčená pouze na několik desítek minut.



# Kapitola 6

## Závěr

Vzniklý simulátor sice není po grafické stránce schopen konkurovat komerčním simulátorům na platformě Android. Nicméně disponuje několika grafickými technikami, které mnoho komerčních simulátorů postrádá. Předzpracované osvětlení dodává scéně lepší vzhled a vedlo se dokonce částečně aplikovat předzpracované osvětlení i na dynamické objekty.

Zadání bylo splněno, pouze publikávající lampy nebyly realizovány ideálně. Jsou realizovány pomocí střídání více map osvětlení, což je paměťově zbytečně náročné a není zde možnost nastavení více stavů světla. Práce je poměrně rozsáhlá a nestihl jsem zhasínající světla realizovat efektivněji.

Práci do budoucna hodlám rozšířit o generování map osvětlení pomocí vrhání paprsku. Výsledek provedený pomocí rasterizace je sice uspokojivý, ale pomocí vrhání paprsku by šly realizovat i plošné zdroje světla a tím by byl výsledný efekt daleko kvalitnější. Aby scéna nebyla příliš tmavá, bylo nutné nastavit poměrně velkou hodnotu pro ambientní složku osvětlovacího modelu.



# Literatura

- [1] Samuel R. Buss. *3-D Computer Graphics: A Mathematical Introduction with OpenGL*. 2003.
- [2] Christer Ericson. *Real Time Collision Detection*. 2004.
- [3] Richard S. Wright Jr. a Nicholas Haemel Graham Sellers. *OpenGL SuperBible*. 2010.
- [4] Lauris Kaplinski. Reflective water with glsl. [khayyam.kaplinski.com/2011/09](http://khayyam.kaplinski.com/2011/09), 2011.
- [5] Dave Astle Kevin Hawkins. *OpenGL Game Programming*. 2001.
- [6] Tobias Ritschel, Thorsten Grosch, Min H. Kim, Hans-Peter Seidel, Carsten Dachsbacher, and Jan Kautz. Imperfect Shadow Maps for Efficient Computation of Indirect Illumination. *ACM Trans. Graph. (Proc. of SIGGRAPH ASIA 2008)*, 27(5), 2008.
- [7] Jim Scott. Packing lightmaps. [www.blackpawn.com/texts/lightmaps](http://www.blackpawn.com/texts/lightmaps), 2002.



## Příloha A

### Seznam použitých zkratek

- 3D (Three-dimensional) - trojrozměrné
- API (Application Interface) - rozhraní aplikace neboli šablona, která říká, jaké má mít aplikace metody
- GLSL (openGL Shading Language) - jazyk pracující s grafickou kartou k výpočtu stínování
- GPS (Global Positioning System) - systém pro určení polohy kdekoliv na světě
- UML (Unified Modeling Language) - vizuální jazyk pro modelování softwaru



## Příloha B

### Obsah přiloženého CD

Položka	Popis
bin/objConverter.jar	konvertor modelů
bin/open4speed.zip	spustitelná verze simulátoru
doc/latex	editovatelná dokumentace simulátoru
doc/text.pdf	dokumentace simulátoru
src/objConverter	zdrojové kódy konvertoru modelů
src/open4speed	zdrojové kódy simulátoru
support	knihovny potřebné ke zkompilování simulátoru
install.txt	návod k instalaci simulátoru
readme.txt	informace potřebné k používání simulátoru



## Příloha C

# Uživatelská příručka

### C.1 Instalace simulátoru

#### Linux

Před komplikováním je potřeba mít nainstalované následující balíčky:  
libbullet-dev freeglut3-dev libpng-dev qmake make

Pokud máte 64bitový systém je potřeba nahradit v jni/open4speed.pro řádek  
.fmodapi/libfmodex-4.44.08.so  
řádkem  
.fmodapi/libfmodex64-4.44.08.so

Komplikování se spustí zadáním těchto příkazů do terminálu:

```
cd jni  
qmake open4speed.pro  
make
```

Spuštění se provede příkazem  
.open4speed

#### Android

Pro Android je k dispozici spustitelný binární soubor Open4speed.apk.  
Pro komplikování je potřeba mít nainstalované následující software:  
Android Studio, Android NDK(v případě potřeby komplikovat C++ kód)

Komplikování C++ kódu:  
cd jni  
<cesta k Android NDK>/ndk-build

Pro zbytek stačí otevřít projekt v Android Studiu a klepnout na Run

## C.2 Používání simulátoru

### Menu

- Start race - herní mód, je zde vytyčen okruh, závodí se s vozidly ovládané umělou inteligencí
- Free ride - volná jízda po městě
- Options - přejde do nabídky nastavení
- Quit game - ukončí simulátor

### Ovládání simulátoru

- šipky nahoru/dolu - plyn/brzda
- šipky vlevo/vpravo - zatáčení
- klávesa esc/zpět - pauza, menu pro restart závodu, ukončení závodu

## C.3 Generování map osvětlení

TODO