Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informatiky a kvantitativních metod

Možnosti vývoje a použití 3D aplikací pro mobilní zařízení na platformě Android

Diplomová práce

Autor: Bc. Tomáš Marek
Studijní obor: Ai2

Vedoucí práce: doc. Ing. Ondřej Krejcar, Ph.D.

Hradec Králové Duben 2015

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury.

vlastnoruční podpis

V Hradci Králové dne 24.4.2015 Bc. Tomáš Marek

Poděkování:

Děkuji doc. Ing. Ondřeji Krejcarovi, Ph.D. za metodické vedení diplomové práce, pomoc a odborný dohled.

Anotace

Hlavním tématem této diplomové práce je implementace a popis různorodých grafických efektů s využitím OpenGL ES 2.0. V prvních kapitolách je představen jednoduchý grafický engine, který slouží pro realizaci jednotlivých 3D scén. Další kapitoly se věnují efektům skeletální animace, osvětlení scény, stínů, vykreslení terénu a vegetace, vodní hladiny, postprocessingu, simulace tkaniny, simulace srsti a augmentové reality na mobilních zařízeních. Každá kapitola obsahuje ukázku implementace a fotodokumentaci. Ke konci práce je provedeno testování vybraných implementací. Na závěr je zhodnoceno celkové provedení práce a schopnosti zařízení. Zároveň je uveden seznam doporučení pro optimalizaci mobilních 3D aplikací.

Annotation

Possibilities for development and use of 3D applications for mobile devices on the Android platform

The main topic of this thesis is implementation and description of various graphic effects using OpenGL ES 2.0. The first chapters present a simple graphical engine, which is used for the realization of individual 3D scenes. Other chapters deal with the effects of skeletal animation, scene lighting, shadows, rendering the terrain and vegetation, water, postprocessing, cloth simulation, fur simulation and augmented reality on mobile devices. Each chapter contains a sample implementation and documentation. At the end of the work is carried out by testing selected implementations. Finally, it assesses the overall performance and capabilities of the device. It is also shown a list of recommendations for optimizing mobile 3D applications.

Obsah

Seznam obrázků

Seznam tabulek

1 Teoretický úvod

Oblíbeným a všudypřítomným trendem poslední doby jsou multifunkční přenosná zařízení. Velkému množství uživatelů naprosto nahrazují domácí počítač a slouží pro zábavu i spojení se světem. Nové přístroje jsou neustále ve vývoji a staré jsou rychle nahrazovány. Tento překotný vývoj dospěl do stádia, kdy možnosti těchto zařízení přinášejí využití do nezměrného množství oborů. Výkonem se však pouze vzdáleně přibližují jiným systémům.

Právě kvůli rozdílům ve výkonu vznikají také rozdíly ve schopnosti zpracovávat určité informace a pro některé aplikace mohou být mobilní zařízení dokonce nevhodná. Příkladem takového případu mohou být aplikace, jejichž součástí je zobrazování 3D grafiky. Při zmínce o grafice mnohé čtenáře mohou napadnout převážně různé druhy her a zábavy. Tato kategorie je velmi rozšířená a přináší podstatný zisk, ale možnosti 3D zobrazování zahrnují mnohem více. Může se jednat o aplikace augmentové reality pro doplnění informací do okolí uživatele při navigaci, zobrazení lékařských dat nebo mobilní moduly větších systémů.

V následujícím textu by měl čtenář dostat odpověď na otázku, kde jsou slabá místa při použití 3D grafiky na mobilních zařízeních a jaké jsou momentální a budoucí možnosti těchto přístrojů. Z tohoto důvodu bude vytvořen jednoduchý grafický engine, na kterém proběhne implementace různých efektů převážně pro zobrazení realistické scény. Vzhledem k širokému záběru informací je práce členěna na kapitoly obsahující popis i implementaci.

První kapitoly seznamují s důležitými pojmy a popisují vznik prostředí pro různé vizualizace. Zahrnut je základní návrh enginu v jazyce Java a popis použitého grafického rozhraní. Tento postup, na rozdíl od použití hotového řešení, přináší bližší kontakt se způsoby 3D vykreslování. Popsané metody a závěry však mohou být platné i u převzatých enginů a 2D grafiky, i když na její použití není práce zaměřena.

Po úvodu do práce s grafikou přichází část věnovaná formátům pro 3D modely. Součástí této kapitoly je i představení dat potřebných k provedení

animací. Ty jako nepostradatelný způsob zobrazení dynamiky scény jsou obsahem navazující kapitoly. Čtenář je důkladně seznámen s momentálně nejpoužívanější a nejefektivnější formou animovaní – skeletální animací.

Práce volně pokračuje v kapitolách postupně popisujících efekty vhodné pro vytvoření komplexní grafické scény, která se vyrovná aktuálním implementacím na volně dostupných i komerčních enginech. Počáteční kapitoly se věnují různým typům osvětlení a pokračují k metodám úpravy povrchu pomocí textur. Následuje implementace dynamických stínů a krajiny. Společně s krajinou jsou představeny i metody pro zobrazení oblohy a mlhy. Důležitým prvkem otevřené scény je vodní hladina se schopností odrážet obraz okolních předmětů. Text se přibližuje i částicovým systémům a představuje způsob vykreslení vegetace založený na billboardingu a inspirovaný hojně používaným komerčním řešením.

Poslední kapitoly se věnují úpravě výsledného obrazu po provedení 3D operací, augmentové realitě a simulacím některých dalších zajímavých efektů. Konečnou částí je přehled základních možností optimalizace postavených na zjištěných výsledcích. Závěrem je provedeno testování vytvořených řešení na skutečných zařízeních.

1.1 Cíle práce

Jádrem práce je implementace a popis různorodých vizuálních efektů na mobilních zařízeních a subjektivní posouzení schopností těchto zařízení. Jednotlivé efekty budou představeny pomocí samostatných scén se zobrazením rychlosti překreslování v jednotkách FPS. Pro závěrečné testování budou scény zatěžovat GPU zařízení různým způsobem. Z provedené implementace budou vyvozeny závěry a doporučení pro optimalizaci grafických aplikací. Součástí práce je i poučení čtenáře o způsobech zobrazování rozmanitých částí 3D scény.

Výsledná aplikace bude sdružovat jednotlivé spustitelné ukázky v přehledné formě. Pro udržení struktury a zjednodušení je vedlejším cílem implementace primitivního grafického enginu, který bude sloužit jako další podklad pro závěrečné shrnutí. Starostí enginu budou převážně základní prvky pro načítání a renderování grafického obsahu, které zahrnují zpracování geometrie, textur a shaderů.

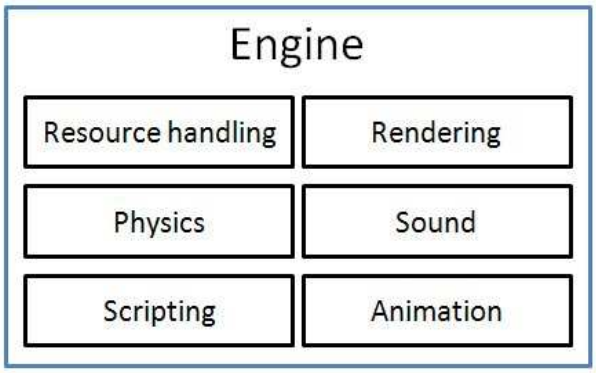
2 3D grafika na mobilních zařízeních

Androidí zařízení používají k práci s 2D a 3D grafikou programovací rozhraní (API) OpenGL ES (Open Graphics Library for Embedded Systems), které je podporováno i přímo výrobci procesorů. Toto rozhraní slouží k základnímu zpracování vektorové grafiky a vyskytuje se v několika verzích. Nejrozšířenějším zástupcem je OpenGL ES 2.0 podporovaný od Androidu 2.2 Froyo [6] a bude použito pro tuto práci. Tato verze umožňuje, na rozdíl od předchozí 1.0, kontrolovat průběh vykreslování. Některá zařízení mohou od Androidu 4.3 podporovat i novější OpenGL ES 3.0 (3.1 pro Android 5.0), které je zpětně kompatibilní a přináší několik vylepšení. Různí výrobci GPU mají drobné odlišnosti v implementaci OpenGL ES API. Jedná se především o podporu vlastností OpenGL nevyskytujících se pro OpenGL ES, případně drobné odlišnosti ve zpracování programovacího jazyka pro shadery (OpenGL ES Shading Language). Hlavní značky GPU jsou:

- ARM Mali,
- Qualcomm Adreno,
- Imagination Power Vr,
- Nvidia Tegra.

Na OpenGL ES API často navazují vyšší rozhraní pro práci s grafikou – enginy. Grafický engine přináší mnohá zjednodušení a automatizaci, ale zároveň některé nevýhody. Může se jednat o nutnost platit za použití a zisk, problém při provádění úprav, doba pro naučení a závislost na vybraném řešení nebo nedostatečný výkon v určitých konkrétních případech. Naprogramování komplexního grafického enginu je ale složitou a pro jednotlivce velmi problémovou činností. Při vytváření grafické aplikace na mobilním zařízení i PC však často vznikají metody a nástroje, které ve výsledku primitivní engine tvoří. Tento postup je vhodný pro seznámení s možnostmi počítačové grafiky a uvědomění si slabých míst pro budoucí schopnost využít plného potenciálu 2D a 3D vykreslování ať už s vlastním, nebo převzatým enginem.

Obecně přináší grafický engine balíky pro práci se samotným vykreslováním (rendering), přidává předem připravené efekty, usnadňuje načítání dat, umožňuje simulace fyzikálních jevů, pracuje se zvukem a mnoho dalšího. Na obrázku číslo 1 je znázornění schopností grafického enginu.



Obrázek 1: Obsah grafického enginu

2.1 Možnosti 3D vizualizací na mobilních platformách

Obecně jsou 2D a 3D enginy nástroje pro efektivní zobrazování a práci s grafikou za účelem zobrazení obsahu uživateli. Enginy mohou obsahovat i další nástroje, např. pro práci s částicemi nebo návrhem scény. Tato práce se věnuje základním částem enginu pro vykreslování a těmi jsou resource handling, rendering a částečně animation.

Resources (zdroje) představují data načítaná a zpracovávaná enginem. Jedná se o modely objektů, jejich textury, zvuky, ale i shadery, animace, scripty a další. Modely jsou obrazem prvků reálného světa nebo mysli designéra a jsou vytvářeny ve speciálních modelovacích nástrojích jako je Blender nebo Autodesk 3ds Max. Modely mohou být také nazývány mesh data a jsou souhrnem všech informací potřebných k vykreslení komplexní scény. Pro přenos dat existují různé druhy formátů [1], např. OBJ, COLLADA, 3DS a další.

Důležitou programovou součástí vykreslování jsou shadery. Jedná se o krátké programy, které ovlivňují průběh renderování. Rozdělují se na vertex a fragment shadery [9] podle jejich zaměření. Veškerá vykreslovaná data jimi procházejí. Vrcholy (vertexy) vykreslovaných objektů jsou transformovány pomocí vertex shaderu. Data pro tyto operace je možné dodat do shaderů v průběhu vykreslování pomocí uniform proměnné. Na rozdíl od uniform, která obsahuje stejná data pro celý model, jsou vlastnosti modelu (pozice, normála, barva atd. pro každý vrchol) předávány pomocí attribute proměnné. Zde se jedná o matice potřebné ke správnému natočení a deformaci scény. Výsledek z vertex shaderu je dále zpracováván a použit ve fragment shaderu, který slouží pro provádění operací na úrovni jednotlivých pixelů. Předávaná data se v shaderech označují jako varying.

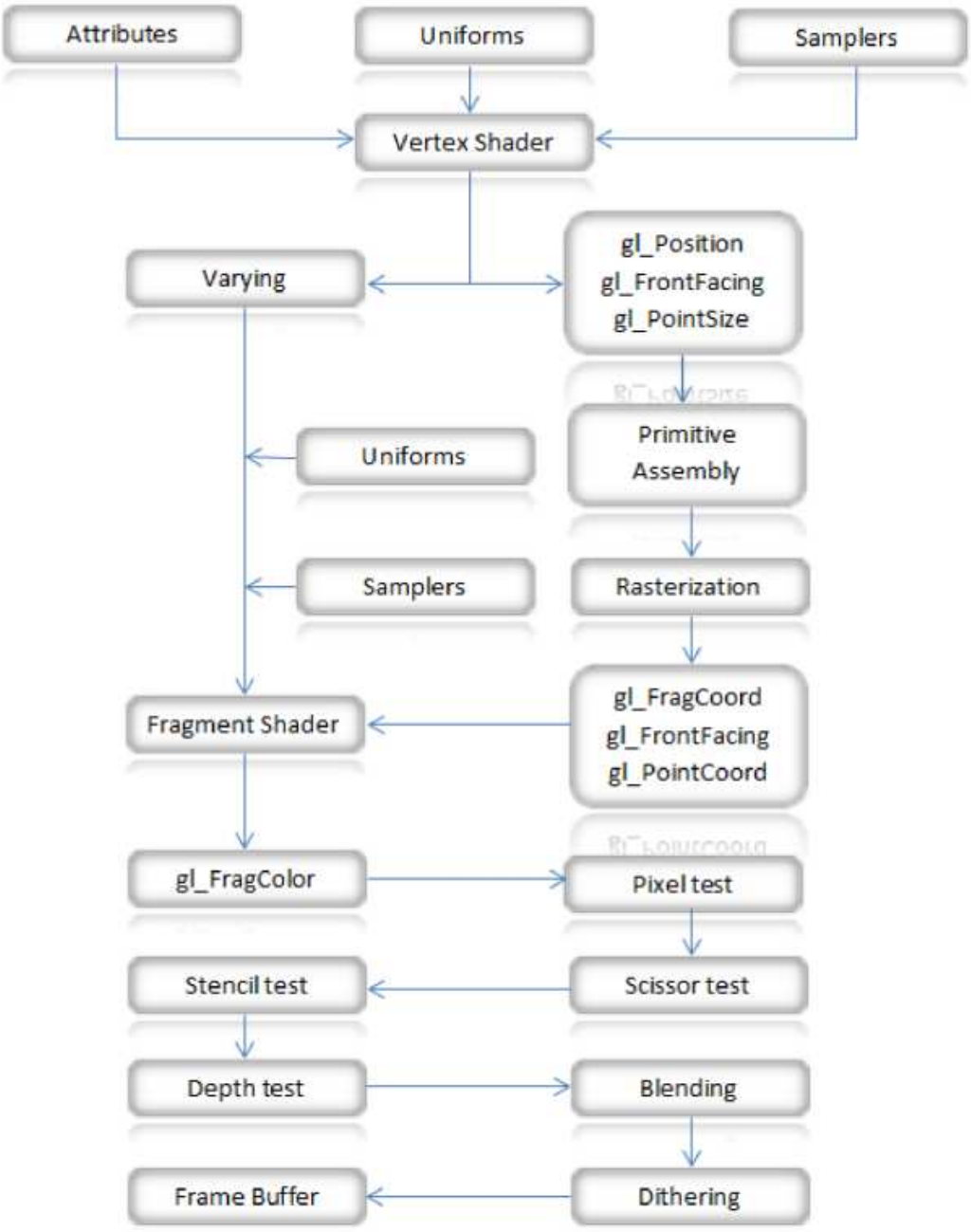
Zobrazování grafického obsahu se téměř neobejde bez použití textur (v shaderu jako proměnná sampler). Textura je 2D obraz, většinou definující barvu pomocí RGB (způsob míchání barev), který je mapován na povrch tělesa. Textury se používají k širokému množství efektů, například pro definování zvrásnění povrchu, pro uložení stínů nebo světel, pro odlesky nebo rovnou zrcadlové odrazy.

Pouze výjimečně je možné mapovat texturu na objekt v poměru jednu ku jedné. OpenGL v ostatních případech využívá nastavení pro texture filtering, kterým lze definovat způsob získání dat z textury. Různá nastavení přináší různé výsledky a mají rozdílný dopad na rychlost vykreslování.

Optimalizace načítaných dat znamená minimalizaci jejich velikosti, případně předběžnou přípravu pro zjednodušení následného renderování. Shadery společně se složitými modely a texturami představují hlavní zátěž enginu. S rostoucí scénou mohou přibývat další náročné výpočty pro fyziku, kolize, animace a další. Vlastní rendering je možné popsat jako přenos dat do grafické karty a jejich průchod grafickým řetězcem, jehož součástí jsou právě shadery. Ke snížení náročnosti je vhodné se zamýšlet už nad datovými typy, dále nad použitím komprimovaných textur, texture atlasů, optimalizací modelových dat, způsobu vykreslování atd. Způsobem vykreslování je míněno zpracování dat při posílání na grafickou kartu. Jedná se o použití volání glDrawArrays pro neindexovaná data a glDrawElements pro indexovaná [9]. Indexy pro data vedou ke snížení duplicity a většinou ke zvýšení rychlosti vykreslování. Parametrem mode lze upravit typ grafických primitiv, která budou renderována. Cílem je znovu urychlení vykreslování a snížení velikosti dat. Grafická primitiva jsou body (POINTS), úsečky (LINES), trojúhelníky (TRIANGLES), speciální druhy spojení trojúhelníků (TRIANGLE_STRIP, TRIANGLE_FAN) a další. Ještě před voláním pro vykreslení je nutné data načíst a poslat na grafickou kartu. Pro různé typy scén existují různé způsoby předávání dat. Základem jsou vertex arrays object (VAO), kdy jsou data uložena v RAM paměti. Lepším řešením je pak využití vertex buffer object (VBO) a uložení vykreslovaných dat ve video paměti.

Detailní pohled na celý grafický řetězec u OpenGL ES 2.0 je na obrázku číslo 2. Uniform, sampler a attribute proměnné jsou předány do vertex shaderu, kde dochází k per vertex operacím. Z vertex shaderu může být volitelná výstupní proměnná varying pro předání dat do fragment shaderu. Gl_Position a další jsou vestavěné výstupní proměnné, kde právě gl_Position je nutné specifikovat. Gl_PointSize je používáno při vykreslování vertexů jako bodů a gl_FronFacing je automaticky generovaná proměnná s informací o natočení vertexů. Vrcholy jsou

odeslány k provedení primitive assembly, tedy k vytvoření grafických primitiv a případnému ořezání pohledovým objemem. Po rasterizaci (převedení na 2D obraz) je spuštěn fragment shader jehož výsledkem je barva pixelu. Jako poslední jsou spuštěny testy pro určení viditelnosti pixelu, míchání barev při průhlednosti a po úspěšném projití těmito testy je pixel uložen do framebufferu a vykreslen na obrazovku.



Obrázek 2: Grafický řetězec OpenGL ES 2.0 [6]

3 Návrh SW rámce pro 3D vizualizace na mobilních zařízeních

Pro práci s grafikou je potřeba OpenGL ES a v Androidu je nutné ho inicializovat pomocí speciální třídy GLSurfaceView. Tato třída zajišťuje konfiguraci displaye, renderování ve speciálním vlákně a pomáhá řídit životní cyklus aktivity [10]. Vykreslování je pak vykonáváno na části displaye nazývané surface nebo viewport. Toto view je následně přiděleno aktivitě jako aktuální content view [11]. U GLSurfaceView je také možné ovládat typ vykreslování (setRenderMode) a nastavit tak plynulé vykreslování, nebo renderovat na vyžádání. Aktivita s GlSurfaceView vypadá následovně [10].

```
public class FirstOpenGLProjectActivity extends Activity {
private GLSurfaceView glSurfaceView;
@Override
     public void onCreate(Bundle savedInstanceState) {
           super.onCreate(savedInstanceState);
//inicializace GLSurfaceView
           glSurfaceView = new GLSurfaceView(this);
//kontrola podpory OpenGL ES 2.0 nebo vyšší
           final ActivityManager activityManager =
                 (ActivityManager)
                 getSystemService(Context.ACTIVITY_SERVICE);
           final ConfigurationInfo configurationInfo =
                 activityManager.getDeviceConfigurationInfo();
           finalboolean supportsEs2 =
                 configurationInfo.reqGlEsVersion >= 0x20000;
           if (supportsEs2) {
//Nastavení GLSurfaceView pro použití s OpenGL ES 2.0
                 glSurfaceView.setEGLContextClientVersion(2);
//přiřazení rendereru
                 glSurfaceView.setRenderer(new Renderer1());
           } else {
                 Log.i(TAG, "ES 2.0 neni podporováno");
           }
     }
```

```
//přiřazení GLSurfaceView aktivitě
     setContentView(glSurfaceView);
           @Override
           protected void onResume()
           {
                 super.onResume();
                 glSurfaceView.onResume();
           }
           @Override
           protected void onPause()
           {
                 super.onPause();
                 glSurfaceView.onPause();
           }
}
```

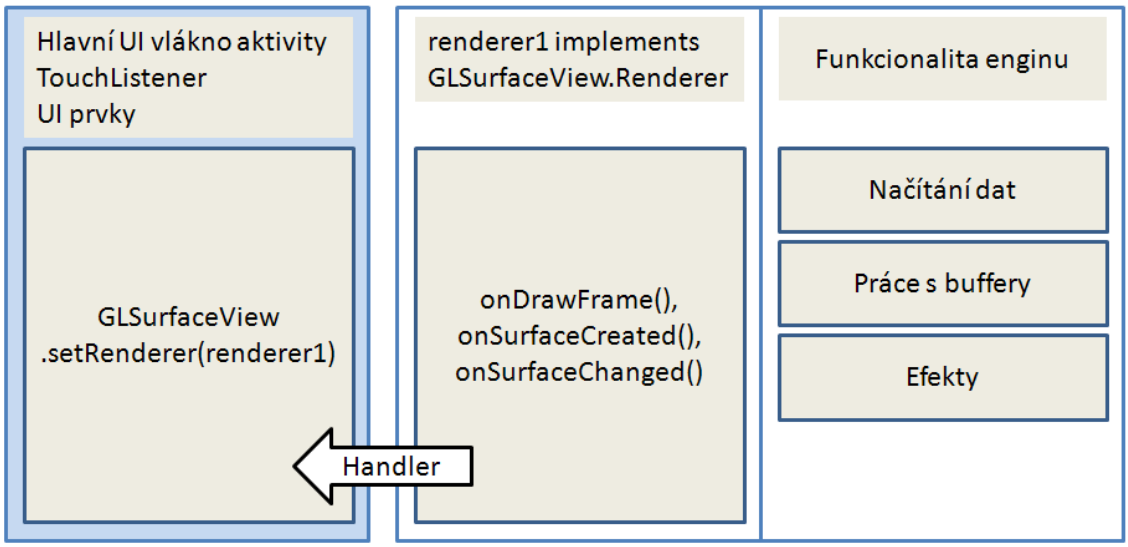
Nyní je potřeba vytvořit třídu, která bude obstarávat samotné kreslení do OpenGL surface view. Taková třída implementuje GLSurfaceView.Renderer a obsahuje tři základní metody:

- **OnSurfaceCreated** – metoda volaná při vytvoření surface. Zde je možné provádět načítání dat a vytváření bufferů.
- **OnDrawFrame** – voláno vždy, když je nutné něco vykreslit. Zde probíhá volání glDraw.
- **OnSurfaceChanged** – voláno po vytvoření surface a vždy když je změněna velikost obrazu například při otočení displaye.

Základní aktivita s GLSurfaceView a renderer tvoří základ budoucího enginu. Dalším prvkem je zajištění komunikace mezi hlavním uživatelským vláknem (aktivitou) a OpenGL vláknem na pozadí. Z UI vlákna je možné přímo pracovat s metodami rendereru a předávat tak informace o uživatelském vstupu. Základní takovou informací je pohyb prstu po obrazovce, který lze zjistit pomocí OnTouchListeneru v UI vlákně, nebo v GLSurfaceView. Pokud je ale nutné zajistit například zobrazení FPS (rychlost překreslování – Frame Per Second), jedná se komunikaci z vlákna v pozadí do hlavního vlákna a je nutné použít Handler (způsob komunikace mezi vlákny Androidu). Hodnota FPS může být jednoduše

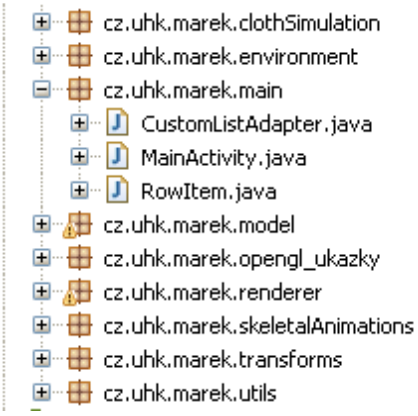
zobrazena v popisku aktivity nebo v připojeném view, kde je také možno nadefinovat další UI prvky.

Po zajištění komunikace s hlavním UI vláknem je konečně připraven základ pro realizaci tříd a balíčků, které budou tvořit samotnou funkcionalitu enginu. Jedná se o pomocné třídy pro práci s buffery a maticemi, třídy starající se o animace, shadery, načítání dat a veškeré budoucí efekty. Konceptuální návrh je zobrazen na obrázku číslo 3.



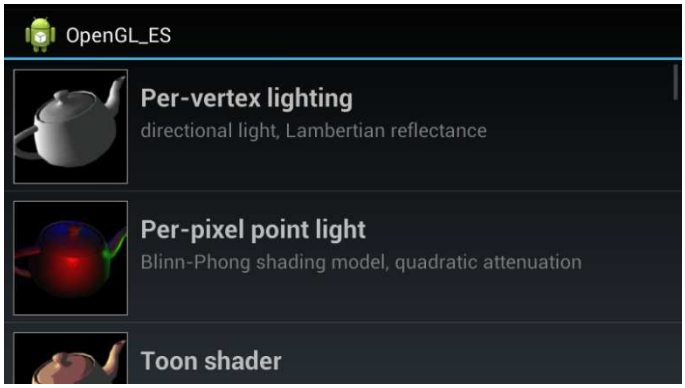
Obrázek 3: Schéma navrhovaného enginu

Main aktivita aplikace, viz obrázek 4, je uložena v balíčku main. Tato aktivita se stará o zobrazení hlavní nabídky, která bude obsahovat seznam všech aktivit pro jednotlivé připravované scény a zajišťovat jejich spuštění po tapnutí na položku.



Obrázek 4: Struktura balíčků pro aplikaci

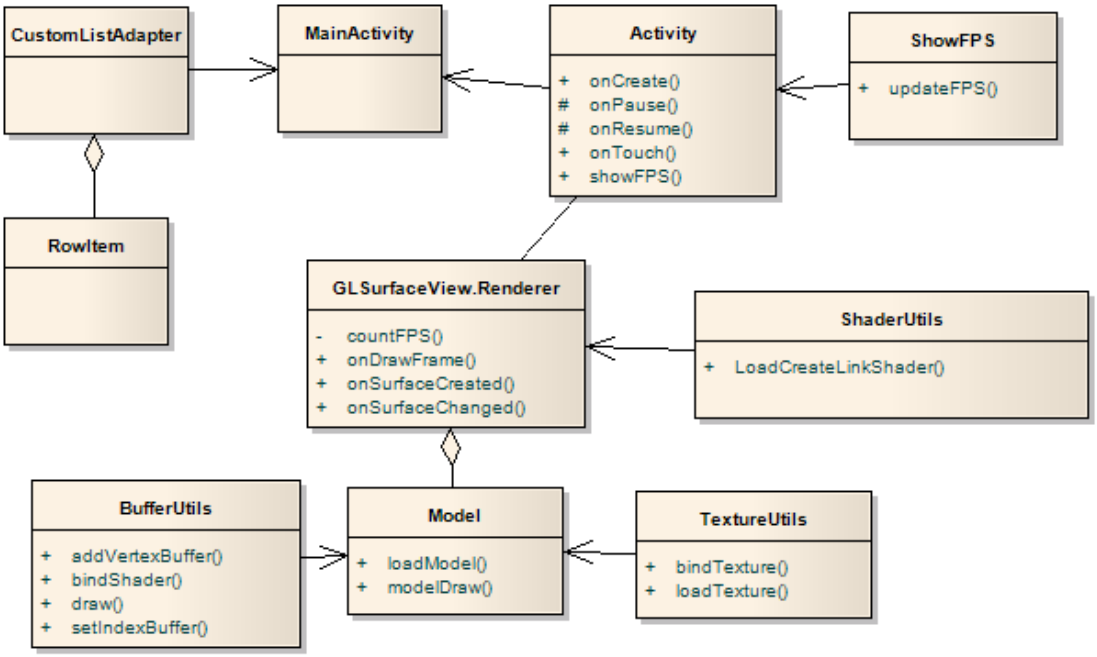
Pro zobrazení jednotlivých odkazů na spuštění scény je použito ListView, které vykresluje scrollovací seznam položek. Každá položka bude obsahovat náhledový obrázek scény, hlavní popis a případné další poznámky k ukázce a je reprezentována třídou RowItem, viz obrázek 5.



Obrázek 5: Ukázka hlavní aktivity s ListView

Tato data jsou předána do ListView pomocí třídy CustomListAdapter, který je rozšířením Androidího BaseAdapter. CustomListAdapter se pak postará o získání a přidělení patřičných dat pro každý řádek ListView.

Řádky ListView pak odkazují na aktivitu s GlSurfaceView. Všechny tyto aktivity jsou uloženy v balíčku opengl_ukazky. Balíček renderer pak obsahuje třídy pro vykreslování přiřazené jednotlivým aktivitám z opengl_ukazky. Balíčky skeletalAnimations a clothSimulation budou obsahovat potřebnou obsluhu pro plánované implementace těchto efektů. Balíček environment je připraven pro zajištění dat efektům zobrazujícím prvky scény, jako je například příprava rovné podkladové plochy. V balíčku model budou implementovány třídy pro reprezentaci geometrie objektů, které budou využívat základní třídy pro vytváření bufferů, přípravu shaderů a textur z balíčku utils. Zde jsou také využity některé třídy pro pomocné výpočty z předmětu PGRF2. Poslední package transforms obsahuje pomocné třídy pro matematické výpočty s maticemi, vrcholy a vektory. Zjednodušený pohled na základ aplikace poskytuje UML diagram na obrázku 6.



Obrázek 6: UML diagram aplikace

4 Implementace navrženého řešení

Na připraveném podkladu lze začít s implementací jednotlivých efektů. Prvním krokem je zobrazení a případně animace modelu, který je uložen v některém ze specifických formátů. Následně tato kapitola popíše spektrum efektů, které postupně rozšíří prosté zobrazení modelu o další vlastnosti a efekty reálného světa.

4.1 Formáty pro ukládání 3D objektů

Důležitou součástí grafického enginu je schopnost načítat modely (meshe). Tyto modely představují zobrazovaný obsah, se kterým je možné pomocí enginu dále manipulovat. Tento druh dat je běžně vytvářen pomocí specializovaných programů určených pro 3D modelování například AutoDesk Maya, AutoDesk 3ds Max, LightWave nebo Blender. Součástí modelů je velké množství informací zaručujících jejich správné vykreslení a animování. Základem je geometrie (tvar objektu) definovaná sadou vrcholů (vertexů) o třech souřadnicích. Vertexy jsou následně spojeny do trojúhelníků (v některých programech čtverců) tvořících povrch modelů. Vytvoření těchto základních ploch (polygonů, faces) je provedeno s použitím indexů, které vrcholy spojují. Modely, které často představují objekty reálného světa, případně jsou realizací designérovi představivosti, nesou také informaci o barvě povrchu (textuře). Kromě samotného texturovacího souboru je nutné, aby model obsahoval také informace sloužící k namapování textury (přiřazení 2D bodu v textuře k vrcholu) na objekt (texturovací souřadnice). Pro aplikaci různých druhů efektů na model, jsou velmi často potřeba další informace. Například normálový vektor určující natočení ploch modelu v prostoru, který je nejčastěji využíván pro výpočet osvětlení. Tato data teoreticky stačí pro zobrazení základní 3D scény a jsou tím nejzákladnějším, co by měl grafický formát pro přenos modelů obsahovat.

Formátů [1] je možné nalézt velké množství a mohou se lišit v množství přenášených informací, způsobu zápisu dat nebo formátu dat (binární, XML – Extensible Markup Language, text). Vzhledem k různým potřebám grafických programů jsou formáty vytvářeny i speciálně pro konkrétní aplikace.

Jako příklad jsou uvedeny některé nejznámější formáty v tabulce číslo 1. Formáty Wavefront obj a Collada budou detailně popsány v následujících kapitolách.

Tabulka 1: Formáty pro přenos 3D modelů [1]

Zkratka	*Název*	*Typ*
blend	Blender file format	Binary, xml
3ds	AutoDesk 3ds Max	binary
md3	Quake 3 Arena Model File	binary
dae	AutoDesk Collada	xml
max	3Ds Max format	binary
obj	Wavefront	text
md5	Doom 3 FPS model format	text
ply	Stanford polygon file format	text, binary

4.1.1 Wavefront OBJ

Wavefront OBJ [2] je textový ASCII formát (American Standard Code for Information Interchange) pro ukládání a přenos základních grafických modelů. Je nejznámějším formátem a je podporován prakticky každým programem pro práci s 3D grafikou. Jeho výhodou je jednoduchá a člověku srozumitelná forma. Podporuje zápis geometrie modelu pomocí vrcholů a jejich spojení do linek nebo polygonů. OBJ je možné využít i pro zápis křivek a ploch. Kromě vrcholů lze ukládat i texturovací souřadnice a normály. Pro informace o vlastnostech povrchu jako je textura, barva a vlastnosti osvětlení se používá formát MTL (Material Library File) [3]. Celý model se tak skládá ze dvou souborů, jedním s příponou obj a druhým s příponou mtl. Struktura OBJ souboru pro základní model je představena v následující ukázce.

```
mtllib Rabbit.mtl
v -0.085634 0.269700 0.112950
v -0.060920 0.287949 0.130558
v 0.043634 0.279194 0.076287
vt 0.958496 0.634766
vt 0.944336 0.670898
vt 0.932129 0.659668
vn -0.900693 -0.407910 0.149388
vn -0.136418 -0.985717 -0.098422
vn -0.929319 -0.142888 -0.340434
usemtl Rabbit
f 6/1/1 7/2/2 8/3/3
f 8/3/3 9/4/4 10/5/5
f 10/5/5 9/4/4 11/6/6
```

První řádek začínající mtllib je referencí na externí materiálový MTL soubor. Řádky s parametrem v na začátku definují vertexy ve tvaru „v x y z (w)“, kde hodnota w je volitelná a využívá se především při ukládání křivek. V ukázce je uvedeno pouze několik řádků každého typu, běžný model má průměrně tisíce vrcholů a tedy i odpovídajících záznamů. Další řádky s označením vt udávají texturovací souřadnice a vn identifikuje normálové vektory. Druhá část souboru začíná usetml, který udává název materiálu v MTL souboru. Pomocí usemtl jsou

také rozděleny jednotlivé části modelu, kde každá část může mít jiný materiál a vykresluje se samostatně. Spojení těchto dat zajišťují řádky s f na začátku. Každá sada tří čísel oddělená mezerou představuje vrchol s přidělenými texturovacími souřadnicemi a normálou. Jednotlivá čísla jsou pořadové indexy odpovídajících typů dat. Tři sady za f vytváří polygon, v tomto případě trojúhelník. Hodnota usemtl označuje název materiálu v MTL souboru:

```
newmtl Rabbit                    #název materiálu
Ns 96.078431                     #váha pro spektaculární světlo
Ka 0.000000 0.000000 0.000000    #ambientní světlo v RGB
Kd 0.640000 0.640000 0.640000    #difůzní světlo v RGB
Ks 0.000000 0.000000 0.000000    #spectaculární světlo v RBG
d 1.000000                       #průhlednost
illum 1                          #osvětlovací model
map_Kd Rabbit_D.png              #difůzní textura
```

Za prvním řádkem s názvem materiálu jsou v MTL souboru uvedeny jednotlivé vlastnosti. Význam řádků je popsán za znakem pro komentář #. Výpočet osvětlení a význam příslušných názvů je vysvětlen v kapitole 4.3. V ukázkách jsou vynechány některé méně důležité vlastnosti. Při použití modelů se ve většině případů pracuje pouze s několika parametry a nastavením scény jako celku. Proto může každý materiál obsahovat jiné množství řádků, určené podle požadavků vývojáře.

Jako základ pro zobrazení modelu může být OBJ formát dostačující, ale jeho hlavní nevýhoda spočívá v neschopnosti pracovat s animacemi. Možnost dodat enginu data pro animování je důležitou vlastností formátu i na mobilních zařízeních. Existuje mnoho formátů podporujících animace, ale pro svou čitelnost je v následující kapitole popsán Collaborative Design Activity (COLLADA).

4.1.2 Collada

Collaborative Design Activity (COLLADA) [5] je formát pro 3D objekty. Collada používá XML schéma a je tak lehce čitelný. Soubory s koncovkou .dae podporují přenos geometrie, transformací, shaderů, materiálů, osvětlení, animačních dat nebo dat pro rag doll fyziku (druh animace). Collada formát není určen pouze pro jeden model, ale dokáže uložit celou scénu se všemi modely a nastaveními. Programy pro 3D modelování tak často umožňují ukládat pouze část z možných dat a práci s collada formátem realizují pomocí pluginů. Vzhledem k složitější struktuře s mnoha možnostmi je vhodné se při načítání modelu zaměřit pouze na potřebná data, která budou enginem využita [4]. Základní rozložení dokumentu se skládá z uzlů obalujících hlavní typy informací:

```
<COLLADA>
  <asset>
  <library_animations>
  <library_lights>
  <library_images>
  <library_materials>
  <library_effects>
  <library_geometries>
  <library_controllers>
  <library_visual_scenes>
  <scene>
</COLLADA>
```

Pro zobrazení a rozpohybování modelu je potřeba několik knihoven (uzlů). Jak vyplývá z názvu, geometrie modelu je ukryta pod uzlem <library_geometries>. Způsob uložení dat je následující:

```
<library_geometries>
  <geometry id="shape" name="shape ">
    <mesh>
      <source id="shape-positions" name="position">
        <float_array id="shape-positions-array" count="11736">
        …</float_array>
        <technique_common>
```

```
            <accessor source="#shape-positions-array"
            count="3912" stride="3">
              <param name="X" type="float"/>
              <param name="Y" type="float"/>
              <param name="Z" type="float"/>
            </accessor>
          </technique_common>
        </source>
        <source>…</source>
      </mesh>
    </geometry>
  </library_geometries>
```

Geometrie jednoho modelu se nachází v uzlu <mesh>, ten dále obsahuje více uzlů <source>, které obsahují informace o vertexech, texturovacích souřadnicích, normálách atd. Každý <source> uzel má id, pomocí kterého bude následně nalezen při formování geometrie. Vlastní data jsou v uzlu <float_array> a v případě textových informací je použit uzel <name_array>. Parametr count u <float_array> značí počet prvků v uzlu, zde se jedná o 11 736 jednotlivých čísel. Tato čísla vytvářejí při vhodném spojení sadu prvků, kterou pomáhá vytvořit uzel <technique_common>. Parametr stride obsažený v uzlu <accessor> říká, kolik floatů z <float_array> je potřeba k vytvoření jednoho prvku. Dále je možné se dočíst i informace o počtu těchto prvků - vertexů (3912) a názvu jednotlivých částí prvku. Jaký je význam dat obsažených v <source> vysvětluje další část uzlu <mesh>.

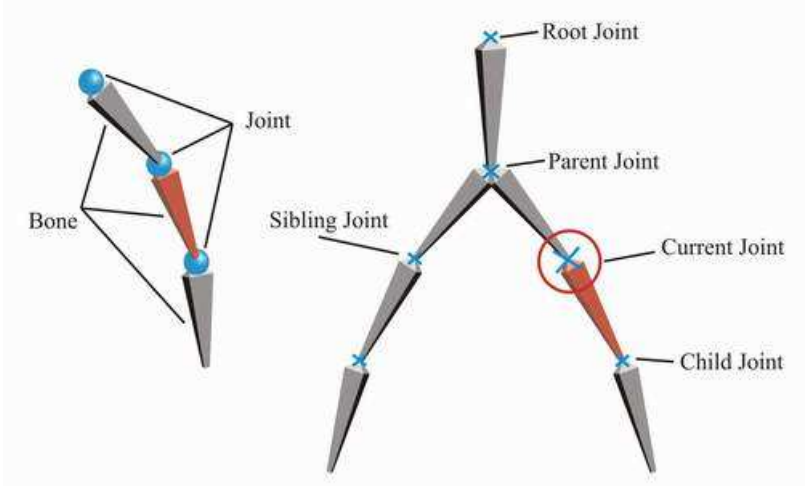
```
<vertices id="shape-vertices">
  <input semantic="POSITION" source="#shape-positions"/>
</vertices>
<triangles material="default" count="7820">
  <input semantic="VERTEX" source="#shape-vertices"
  offset="0"/>
  <input semantic="NORMAL" source="#shape-normals" offset="1"/>
  <input semantic="TEXCOORD" source="#shape-map-channel1"
  offset="2" set="1"/>
  <p>0 0 5 3 3 1 2 2 8 …</p>
</triangles>
```

Pomocí uzlu <triangles> lze sestavit finální model, protože obsahuje reference na veškerá potřebná data v přechozích uzlech. Parametr semantic v <input> je názvem dat a source odkazuje na id některého z dříve uvedených uzlů <source>. Pouze u vertexů je výjimka, kdy source odkazuje na uzel <vertices> a ten poté odkazuje na skutečná data. Zvláštní uzel <p> obsažený v rodičovském <triangles> jsou indexy do patřičného <float_array>. Pro normály je nutné přečíst každý druhý záznam <p>, identifikovaný pomocí parametru offset, a odkázat se do správného <float_array> pro tři čísla tvořící normálový vektor. Protože formát podporuje různý způsob ukládání dat, je nutné tyto prvky sledovat. Uzel <triangles> se vyskytuje pouze, pokud jsou při ukládání jako polygony zvoleny trojúhelníky. Také texturovací souřadnice mohou obsahovat data v lehce odlišném formátu.

Podobným způsobem lze dohledat další informace o modelu uložené v XML formátu. Například parametr material u <triangles> odkazuje do uzlu <library_materials>, kde jsou popsány vlastnosti daného povrchu.

Jak bylo zmíněno v úvodu k této kapitole, cílem je model nejen zobrazit, ale také rozpohybovat. Konkrétní způsob animace modelu je popsán v kapitole 4.2, před animací je ale nutné nejdříve získat potřebná data [4]. Ta jsou uložena v uzlech <library_controllers>, <library_visual_scenes> a <library_animations>. V aktuální implementaci byla použita metoda skeletální animace. Toto pojmenování souvisí se způsobem a potřebnými daty k její realizaci. K základnímu modelu je připojena takzvaná kostra (skeleton, armatura) a na tuto kostru jsou aplikovány transformace. Následně dochází ke skinningu, neboli připevnění původního modelu k rozpohybovanému skeletu. Animační data jsou tedy tvořena skeletonem, daty pro skinning a v tomto případě sadou transformačních matic, které kostrou pohybují.

Běžně je kostra složena z kostí spojených klouby a nejinak je tomu i u kostry pro animování, pouze klouby se nazývají jointy. Z pohledu implementace jsou potom kosti (bones) a jointy prakticky totožné. Na obrázku 7 je znázornění jointů, kostí a jejich hierarchie.



Obrázek 7:Rozdělení kostry pro skeletální animaci [4]

Pro získání kostry potřebujeme přečíst posloupnost jointů v <library_visual_scenes>. Informace o tom, který joint je rodičem jiného, je zásadní pro správné provedení animace. Každý joint je uzel <node> s parametrem type = „JOINT“. Při čtení je, jak již bylo naznačeno, důležité zachovat hierarchii jointů. Dalším kritickým prvkem pro práci s jointy je id, které umožní propojení s daty pro skinning. Uzly <node>(jointy) obsahují také matici v uzlu <matrix>, která slouží pro nastavení polohy kostry, pokud není aplikována animace (bind pose, resting pose). Struktura dat se může lišit, pokud model obsahuje více koster nebo jointy, ke kterým nebude připevněn model.

Skeletona je nyní nutné obalit modelem a provést tak skinning. Ke každému jointu jsou přiřazeny vertexy modelu a transformace jointu následně ovlivní i každý z přidělených vertexů. Jeden vrchol je často přiřazen k více než jednomu jointu. Bez tohoto propojení by animovaný model působil tvrdě a uměle. Váha (wieght) s jakou bude konkrétní joint vertex ovlivňovat je společně s dalšími skinning daty uložena v uzlu <library_controllers>. Tento uzel obsahuje, pro jeden model s jednou kostrou, uzly <controller> a <skin>, ve kterých jsou potřebné skinning informace. Prvním uzlem je <bind_shape_matrix>, která slouží pro nastavení vzájemné polohy modelu a kostry a většinou bývá nastavena jako identity matice (ekvivalent násobení jednou). Následují tři uzly <source> nesoucí id jointů, váhy a inverse bind shape matice. Poslední uzly <joints>

a <vertex_weights> slouží pro propojení všech těchto dat pomocí indexů podobně jako uzel <p> u geometrie.

```
<bind_shape_matrix>1 0 0 …</bind_shape_matrix>
<source>informace o id jointů</source>
<source>inverse bind shape matrices pro každý joint</source>
<source>weights</source>
<joints>propojení jointů a inverse bind shape matic</joints>
<vertex_weights>propojení vah, vertexů a
jointů</vertex_weights>
```

Inverse bind shape matrix slouží pro odstranění transformací bind matrix a správné provedení animace. Nyní jsou přečteny informace o geometrii modelu a každý vrchol má informaci o tom, které jointy ho budou ovlivňovat a jak moc. Samotná animace, matice pohybující kostrou, je uložena v <library_animations>. Tento uzel znovu obsahuje tři <source> uzly a následně <sampler> a <channel> uzly k propojení dat. Informace, které se získají, jsou sada transformačních matic pro každý joint, čas animace a způsob interpolace mezi maticemi. Čas animace je časový údaj, kdy od času nula má být použita daná animační matice daného jointu.

Problémem při čtení collada formátu je, že může obsahovat data v různých tvarech a další drobné odlišnosti způsobené exportéry grafických programů. Vhodným řešením může být použití knihovny, která navíc zajistí načítání dalších formátů a data podává ve stále stejné struktuře. Velmi často používanou knihovnou je například Assimp.

4.1.3 Proprietární formáty

Při snaze vybrat a použít některý formát v enginu se objevuje několik problémů. Prvním je rychlost načítání dat, kdy je často data nutné parsovat (určit gramatickou strukturu prvků) a následně upravit pro potřeby grafiky. Mnoho formátů také obsahuje velké množství nepotřebných údajů a používá relativně složitou strukturu k jejich záznamu. Častým řešením bývá vytvoření vlastního formátu pro daný engine, který je efektivnější. Zároveň s novým formátem je nutné vytvořit separátní aplikaci pro čtení běžných formátů a vytvoření nového souboru.

Výhodou takové aplikace je možnost optimalizace modelů. Například odstranění duplicit v datech, přepočet dat pro triangle strip nebo přiřazení vlastních materiálů. Výběr materiálu je možný i v některých grafických programech, ale obzvlášť na mobilních zařízeních je nutné vytvořit optimalizované materiály. Tím se rozumí osvětlovací a jiné efekty použité na povrch modelu za účelem simulace určitého typu reálného nebo jiného povrchu, např. kovu nebo dřeva. Dále je možné upravovat animace a veškeré další vlastnosti modelu včetně textur. Na zpracování dat v samostatné aplikaci je dostatek času, na rozdíl od spouštění finálního produktu, kdy není vhodné nechat uživatele dlouho čekat.

Příkladem je využití Java InputStream a OutputStream pro uložení dat v binární podobě. Nejjednodušším řešením může být uložení instance objektu, který obsahuje celý model kromě textur.

4.2 Animování 3D objektů

Cílem animace je uvést statický model do pohybu. To je možné několika způsoby. Může se jednat o pohybování modelem jako celkem, ale složitějším krokem je pohybovat různými částmi modelu a simulovat tak plynulý pohyb. Nejzákladnějším druhem je key frame animace. Tu je možné si představit jako rozpohybování loutek ve filmu, kdy pro každý záběr je nutné nastavit polohu loutky. Každá napozicovaná loutka se tak v 3D grafice stává samostatným modelem, který je nutné načíst a zobrazit ve správný čas. Ve 2D je podobný způsob nazýván sprite animace. Sada napozicovaných modelů je zde vyměněna za sadu obrázků a jejich postupným zobrazováním je vytvořen efekt plynulého pohybu. Tento způsob je ale nevýhodný kvůli objemu potřebných dat a nemožnosti další práce s animacemi, jako je například blending (smíchání dvou a více animací). Vhodnější metodou tak je skeletální animace nastíněná v kapitole 4.1.2.

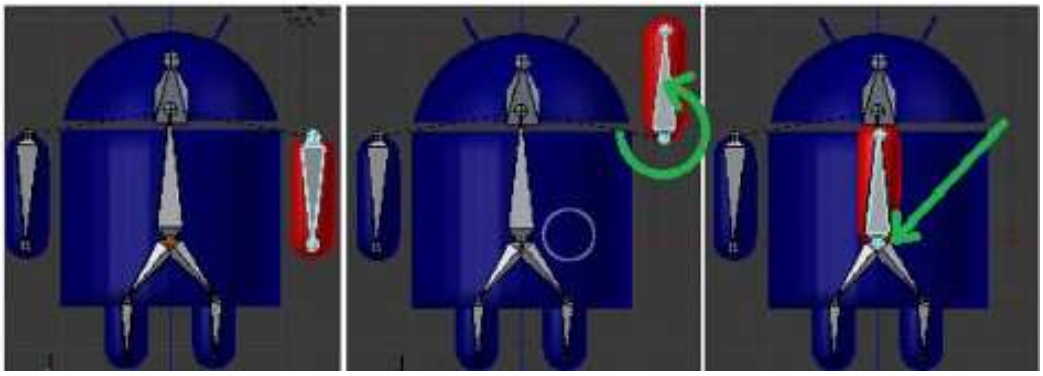
Základním prvkem skeletální animace [6] je kostra a její části (jointy), ke kterým jsou přiřazeny sady animačních matic. Tyto matice tvoří samotný pohyb, nemusí se ale vždy jednat zrovna o matice. K práci s animačními daty, které určují posun a hlavně rotaci, lze použít i quaterniony (reálná čísla se speciálními operacemi sčítání a násobení) nebo b-splainy (aproximační křivka).

V kapitole 4.1.2 byla z collada formátu získána všechna data potřebná k zobrazení modelu, tedy vertexy a k nim přiřazené informace o textuře a normálách, stejně tak jako spojení vertexů do polygonů. Pro animace jsou k dispozici následující data:

- **kostra** (skeleton, armature) zaznamenaná jako hierarchie „kloubů“ jointů,
- **bind shape matrix** – matice pro nastavení vzájemné polohy kostry a modelu, většinou indentity matice – mají stejný počátek,
- **animační matice** s časem pro každý joint,
- **bind pose**(resting pose) maticepro každý joint – póza, ve které byl model připevněn ke kostře,
- **inverse bind pose** matice pro každý joint – pro odstranění bind pose při aplikaci animační matice,
- **spojení jointů a vertexů** s informací o **váze** s jakou joint vertex ovlivní.

Nový model je standardně v grafickém programu vytvářen v modelovací póze. Taková póza umožňuje pohled na veškeré detaily modelu. Například lidská ruka je lépe viditelná, pokud není zaťata v pěst. Stejně tak i model postavy je vytvářen ve tvaru písmene T. V takové chvíli je k modelu připevněna kostra. Této metodě se říká rigging a výsledkem je bind póza. Znázornění kostry a modelu programem Blender je zobrazeno na obrázku číslo 8. Rest (bind) póza ale může být uložena i v jiné než modelovací póze. Každý joint kostry je reprezentován maticí, kterou je právě bind pose matice. Při skinningu je tedy při použití pouze bind pose matic vykreslen model v bind pose. Pro zobrazení modelu v jedné z póz určených animačními maticemi je nutné tuto základní bind pózu vyrušit a k tomu slouží inverse bind pose matice.

Pro pochopení animace je vhodné popsat průběh transformací modelu. Na obrázku 8 je znázorněn vliv inverse bind pose matice na modelu Androida.



Obrázek 8: Vliv inverse bind pose matice na joint a vertexy [6]

Joint (bone) a s ním i veškeré vertexy jsou na obrázku číslo 8 přesunuty z bind pose do počátku pomocí inverse bind pose matice (červená část). Nyní je možné aplikovat na tuto kost vlastní animaci, tedy jednu z animačních matic vybranou podle času. Po transformaci je potřeba dostat bone zpět na její původní pozici. Zde přichází složitější část, protože každá kost je ovlivněna svou rodičovskou kostí a musí reagovat na tyto předchozí transformace. To je zajištěno pomocí skládání transformací a zaznamenané hierarchie jointů. Vynásobením dvou matic definující transformaci v prostoru vznikne kombinace těchto transformací. Je nutné dávat pozor na pořadí násobení, které u matic není komutativní. Pokud zmíněná fakta převedeme na rovnici, ze které lze dostat finální animovaný vrchol, bude vypadat takto:

```
Animated vertex = vertex * bind shape matrix * inverse bind
pose matrix * world matrix
```

World matrix je tedy matice, která vznikla vynásobením world matice rodičovského jointu s animační maticí aktuálního jointu a zajistí přesun aktuálního jointu na správnou pozici. To znamená, že transformace se vztahují k modelu a ne relativně k rodičovskému jointu. Bind shape matrix lze vynechat, pokud je počátek modelu i kostry stejný (bin shape matice je identity matice). Následuje ukázka pseudokódu, pomocí kterého ze základních dat dostaneme sadu matic potřebných pro další krok - skinning. Jako první je nutné zjistit, v jakém čase se animace nachází a vybrat správná data pro další transformaci modelu.

```
float keyFrame = t / animationIncrementTime;
for(int i = 0; i <počet jointů; i++){
  if (keyFrame není celé číslo){
         animationMatrix = interpolate (
         joint(i).getAnimationMatrix(round(keyFrame +
         0.5)),joint(i).getAnimationMatrix(round(keyFrame -
         0.5)), hodnotaInterpolace);
  }else{
    animationMatrix = joint(i).getAnimationMatrix(keyFrame);
  }
```

KeyFrame je index do pole animačních matic, který získáme z aktuálního času běhu animace (t) a časových přírůstků (animationIncrementTime), ve kterých byly animační matice zaznamenány. Pokud máme novou animační matici každých 0.3 s a aktuální čas běhu animace je 1.2 s, pak budeme potřebovat čtvrtou animační matici aktuálního jointu. Problém je v případě, že čas animace není dělitelný přírůstkem. V takovém případě se animace nachází mezi dvěma animačními maticemi. Pokud se použije vhodná matice, až když čas animace překročí dobu pro použití dané matice, bude animace trhaná. Zde, na rozdíl od přímého key frame animování, lze využít jedné z výhod skeletální animace – interpolaci. Dvě matice lze jednoduše lineárně interpolovat a získat tak kombinaci obou matic, to vede k finální plynulé animaci. Interpolace probíhá na základě hodnoty (hodnotaInterpolace), která udává váhu, jakou bude každá matice na výsledek mít. Aktuální matice pro transformaci jointu při animaci (animationMatrix) nemusí vždy existovat. V takovém případě je možné ji nahradit bind pose maticí daného jointu, protože není animován a je tedy ve své výchozí poloze. Sadu matic, kterými stačí vynásobit patřičné vrcholy modelu pro získání animace (skinningMatrices), dostaneme takto [7]:

```
if( joint(i).hasParent() ){
  joint(i).worldMatrix = animationMatrix *
                         joint(i).parent.worldMatrix();
}else{
  joint(i).worldMatrix = animationMatrix;
}
skinningMatrices(i) = joint(i).invBindMatrix() *
                         joint(i).worldMatrix();
}
```

Pokud má joint rodiče (nejedná se o kořen), bude jeho animační matice vynásobena s world maticí rodiče a následně proběhne násobení s inverse bind pose matrix. Po dokončení tohoto forcyklu získáme matice pro každou kost, ke kterým stačí přiřadit správný vertex a dokončit animaci. Tyto operace je možné provádět stále na procesoru zařízení, pak se jedná o CPU animaci, nebo na grafickém čipu při použití vertex shaderu (GPU animace). Vzhledem k tomu, že GPU je více uzpůsobeno k operacím s mnoha prvky (vrcholy), je rychlejší a vhodnější použít tuto metodu. Při CPU animaci je získán každý výsledný vrchol na CPU a je tak nutné updatovat buffery při vykreslování. Tento způsob je více přiblížen v kapitole 4.12 Cloth simulation.

Finální část animování (skinning) proběhne ve vertex shaderu, kde se využijí skinning matrices představující konečnou polohu jointů. Další potřebné informace mohou být přenášeny společně s vertexy. Jedná se hlavně o propojení s jointy a váhy těchto propojení. Jednou z možných realizací je přenos dvou číselných polí, kdy v jednom jsou obsaženy indexy ke skinning maticím a ve druhém váhy. Shader pracující s těmito daty vypadá následovně:

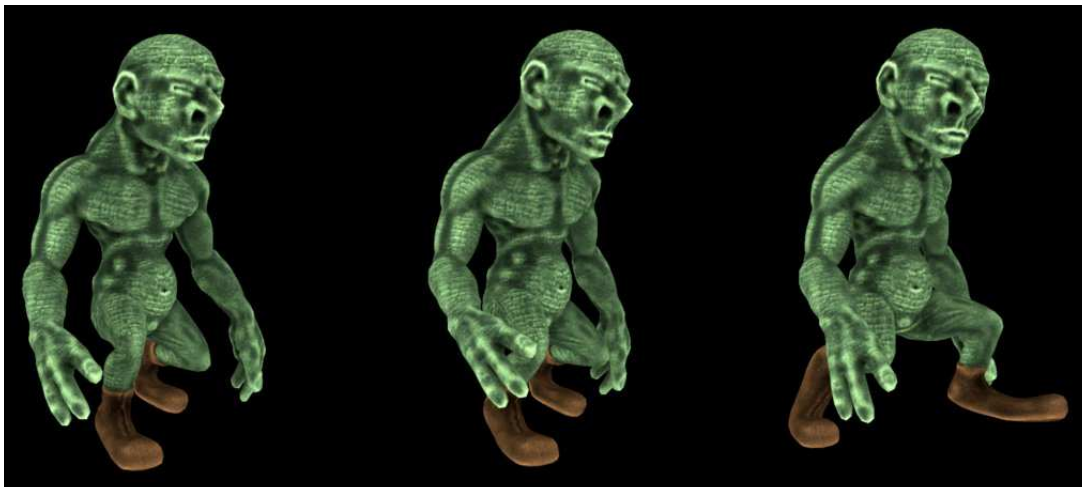
```
attribute mediump vec3 inPosition;
attribute lowp vec4 weights;
attribute mediump vec4 jointsId;

uniform mediump mat4 t_projection;
uniform mediump mat4 t_modelview;
uniform mediump mat4 bones[42];
```

```
void main() {
   mat4 animationMatrix =
    (weights[0] * bones[int(jointsId[0])]) +
    (weights[1] * bones[int(jointsId[1])]) +
    (weights[2] * bones[int(jointsId[2])]) +
    (weights[3] * bones[int(jointsId[3])]);

gl_Position = t_projection* t_modelview
              *animationMatrix*vec4(inPosition, 1.0);
}
```

Jako atributy (proměnné pro každý jednotlivý vertex) jsou přenášena data o spojení s jointy a váhy. V OpenGL ES lze využít datový typ vec4, ve kterém lze přenést čtyři hodnoty. V některých případech, například u CPU animací, je ale jeden vertex přiřazen k více než čtyřem jointům. Zde je nutné data přepočítat a pro všechny takové případy rovnoměrně rozdělit přesahující propojení k prvním čtyřem. Součet vah by měl v každém případě dát jedna. Ořezání počtu spojení ve většině případů není znát, protože se často jedná pouze o drobná doplnění pohybu. Obzvláště na mobilních zařízeních je pak ořezání vhodné i z hlediska výkonu. Naopak chybějící spojení jsou nahrazena libovolnou hodnotou a váha je nastavena na nulu, takže nebude mít na výsledek žádný vliv.

Takové úpravy, pokud jsou nutné, lze provádět v dedikované aplikaci. V knize Android Game Development: From shaders to skeletal animation [6] je celý proces načítání dat zabudován do finální aplikace a podle autora trvá několik vteřin. Důraz byl očividně kladen na vysvětlení problematiky a ne na rychlost. Použití proprietárního formátu a samostatné aplikace pro zpracování a optimalizaci dat modelu přineslo výrazné zlepšení, kdy načtení složitějšího modelu a animace se pohybuje kolem jedné vteřiny. Zároveň byly provedeny úpravy ve výpočtech, kdy lze předem připravit skinning matice pro celý model najednou a pro vertex shader nechat pouze samotný skinning proces, čímž dojde k jeho výraznému zkrácení. Zároveň je tento způsob vhodnější pro budoucí práci s animacemi.

Další část uvedeného vertex shaderu obsahuje uniform proměnné přenášející jednotná data pro všechny vrcholy měnící se při každém překreslení. Jako první jsou zapsány běžné matice pro perspektivní transformaci a vhodné napozicování celého modelu ve scéně. Pro animace je důležitý další řádek, který je zápisem proměnné přenášející pole matic - bones. Tato proměnná obsahuje právě dříve vypočtené skinning matice. Nyní je možné vytvořit matici, která přesune aktuální vrchol do cílové pozice. Ta vznikne vynásobením příslušné skinning matice získané pomocí indexu a váhy, se kterou má vrchol ovlivnit. Po sečtení všech čtyř záznamů lze finální animační matici zařadit do běžné transformace vrcholu v gl_Position. Otexturovaný animovaný model je k prohlédnutí na obrázku číslo 9.



Obrázek 9: Ukázka animovaného modelu
Zdroj: Model vytvořil Christoph Rienaecker

Nyní je engine schopen spustit jednoduchou skeletální animaci. Možných rozšíření je velké množství. Jednou z velice oblíbených metod je blending animací, kdy dvě animace jsou sloučeny do jedné nové. Příkladem může být chůze postavy a skrčení, kdy kombinací vznikne plížení. Problém nastává, pokud je cílem spojit dvě samostatné animace bez míchání, jako je mávání rukou a chůze. Tomuto problému se věnuje animation layering. Další vylepšení může spočívat v animacích, které reagují na své okolí. Jedná se o chůzi, kdy se chodidla postavy perfektně umisťují na schody, nebo o uchopení předmětu. Těchto efektů lze dosáhnout za použití inverse kinematics. Tato metoda dopočítává pozice jointů podle jednoho zvoleného jointu a využívá se také v robotice. Zajímavě pojal animace David Rosen

na Games Developer Conferenci 2014 [8], kde popsal různé druhy interpolace mezi klíčovými polohami modelu a simulace fyzikálních jevů jako zrychlení a zachování hybnosti. Ukázal tak, že je možné použít pouze několik základních poloh modelu a veškerý pohyb dopočítat. U těchto metod se dá mluvit o procedurálních animacích. S procedurálními animacemi se pojí další pojem a tím je ragdoll fyzika, která se nejčastěji používá pro simulaci umírající postavy. Jednotlivé jointy se v tu chvíli vzájemně ovlivňují a postava se zhroutí k zemi. Podobných metod lze nalézt větší množství, například v kapitole 4.12 je popsána verlet integration.Animace jsou tak důležitou součástí fyzikálních enginů a velkou měrou přispívají k věrohodnosti zobrazované scény.

4.3 Osvětlení scény

Světlo je důležitou součástí scény a umožňuje rozpoznat trojrozměrnou strukturu světa. Způsobem jakým se světlo odráží od povrchu tělesa lze také popsat materiálové vlastnosti. V reálném světě existují různé druhy světelných zdrojů [12]. Některé, většinou nekonečně vzdálené nebo nekonečně malé, poskytují téměř rovnoběžné paprsky. Implementace takového světla je relativně jednoduchá a slouží především k simulaci slunečního svitu, jehož paprsky se liší pouze o 0,005 stupňů. Tento druh světla se nazývá directional light (směrové světlo) a přichází rovnoměrně z jednoho směru. Oproti directional light jsou zdroje zářící do všech směrů – point light (bodová světla). Bodová světla se používají pro vykreslení většiny ostatních zdrojů (pouliční osvětlení, lampy, oheň), i když v reálném světě dokonalé bodové světlo neexistuje. Záření je často vhodné určitým způsobem regulovat. Světlo z žárovky nesvítí do nekonečna a navíc může být omezeno krytem lampy. Úbytek energie světla je souhrnně pojmenován attenuation. Zastíněné světlo baterky lze nalézt pod pojmem spot light. Způsobů výpočtu osvětlení je více. Nejrealističtějším je ray tracing, který bere v úvahu každý, nebo téměř každý, paprsek ve scéně. Takový způsob je ale velmi náročný na výkon a nepoužívá se pro real-time rendering, kdy je nutné každý obraz překreslit mnohokrát za vteřinu. Většina používaných osvětlovacích modelů je založena na tzv. BRDF (Bidirectional Reflectance Distribution Function) funkci, která udává pravděpodobnost odrazu paprsku světla daným směrem. Jednotlivé modely si vystačí pouze s částmi celého vzorce a definují odrazivost (reflectance) povrchu. Odrazivost lze dělit na difúzní (ideální všesměrový odraz) a spekulární (ideálně zrcadlový odraz). Další používanou složkou je ambient light, která jednoduše simuluje globální osvětlení a zvýrazňuje místa, kam se přímé světlo nedostane.

4.3.1 Lambertian reflectance

Tento osvětlovací model je jeden z nejjednodušších a popisuje ideální difúzní odraz pomocí Lambertova zákona [13]. Hlavní část vertex shaderu pro výpočet Lambertova difúzního odrazu:

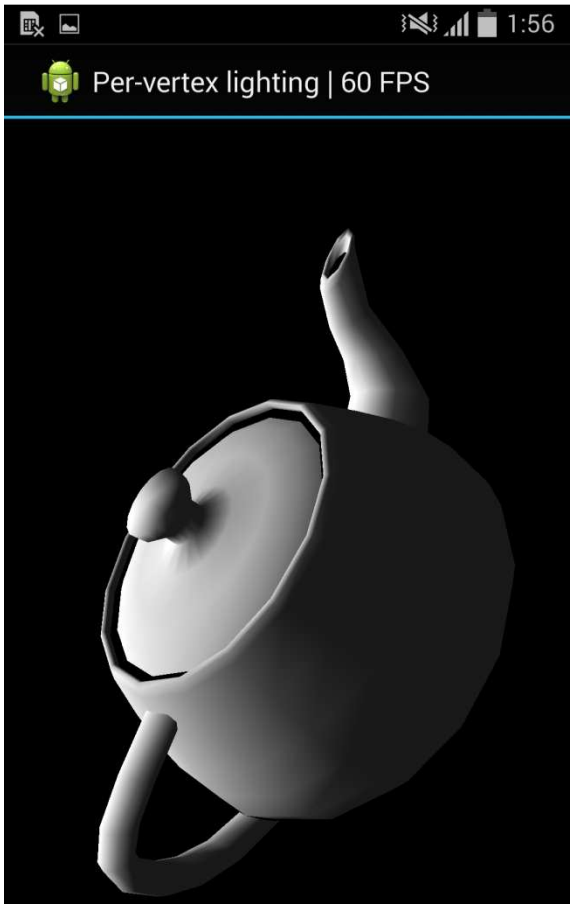
```
lowp vec3 normal = normalize(modelView *
                   vec4(inNormal,0.0)).xyz;
lowp vec3 lightDirection = normalize(lightPos);
lowp float ndotl = max(dot(normal,lightDirection),0.0);
lightColor = ndotl * diffLightColor;
```

Lambertův zákon říká, že úroveň odrazu je přímo úměrná cosinu úhlu mezi normálou a směrem světla (lightDirection). Aby byl povrch osvětlen, musí být úhel 0 až 90 stupňů. Nejdříve získáme normálu (normal) v eye space vynásobením původní normály (inNormal) s modelView maticí. Správně zvolený prostor pro výpočty může usnadnit některé operace. Eye space je možné si představit jako kameru umístěnou v počátku a veškeré objekty jsou umístěny vůči kameře (počátku). Normalizací se zajistí, že délka vektoru bude vhodná pro další výpočty. Normála je třísložkový vektor a modelView je matice 4x4, proto je nutné přidat k normále jednu hodnotu. Při přesunu vertexu do eye space je tato hodnota jedna, ale u normály je nula. Důvodem je, že chceme měnit pouze orientaci normály a to zajistí horní levá 3x3 matice z modelView. Tento postup je možný pouze, pokud nedochází k deformacím objektu, jinak je nutné vypočítat speciální matici určenou k přesunu normály do eye space.

Dot produkt mezi normálou a lightDirection vrátí hodnotu -1 až 1. Záporné hodnoty znamenají úhel větší než 90 stupňů a úroveň světla je v takovém případě nulová, protože plocha je příliš odkloněna od zdroje světla. Výsledek (ndotl) je tedy vhodné oříznout na kladné hodnoty. Po vynásobení ndotl a barvy (diffLightColor) je získáno finální osvětlení, které může být ve fragment shaderu přímo zapsáno do výstupní proměnné gl_FragColor. Pro implementaci ambient light stačí přičíst barvu dalšího světla. Může se jednat o slabou šedou barvu.

Tímto shaderem bylo vytvořeno směrové (directional) pohledově nezávislé osvětlení a povrch, který je dokonale difúzní. Další důležitou poznámkou je, že

světlo je počítáno ve vertex shaderu. Hodnota světla je tak získána pouze pro každý vrchol a bude interpolována při přechodu do fragment shaderu. To může přinášet určité nedokonalosti v osvětlení povrchu, ale výpočet je méně náročný. Vertex shader má významně nižší počet spuštění než fragment shader. Obrázek číslo 10 zobrazuje známý model čajové konvice z Utahu s vypočteným per-vertex Lambert difuze light.



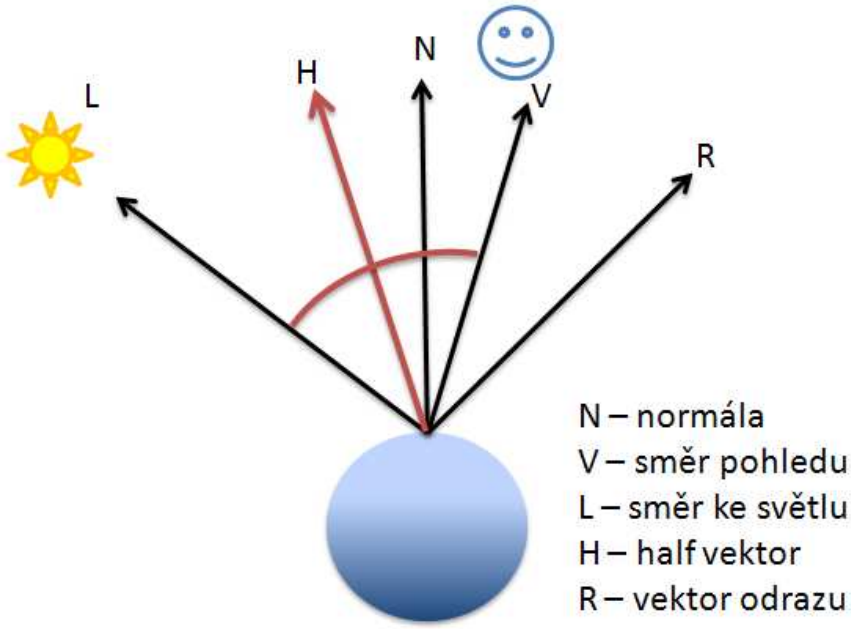
Obrázek 10: Znázornění per-vertex Lambert light na čajové konvici

4.3.2 Blinn–Phong osvětlovací model

Blinn-Phong osvětlovací model je velmi oblíbený způsob, jak přidat spekulární složku a zobrazit tak odlesk. Výpočet odlesku závisí na tzv. half vektoru, který představil Jim Blinn jako modifikaci Phongova modelu. Blinn-phong model je výpočetně méně náročný a to zejména při použití se směrovým světlem. Následující shadery budou však pro ukázku počítat bodové světlo včetně útlumu.

```
//vertex shader
//pozice vertexu v eye space
vec3 position = (modelview * vec4(inPosition,1.0)).xyz;
//normála v eye space
normal = (modelview * vec4(inNormal,0.0)).xyz;
//směr pohledu
viewDir = (- position);
//směr ke světlu
lightDirection = (lightPos - position);

//výpočet útlumu se vzdáleností
float distance = length(lightDirection);
attenuation = (1.0 / (1.0 + (0.01 * distance * distance)));
```

První hodnoty jsou pozice vertexu a normálový vektor v eye space. Směr pohledu (viewDir) je vektor mířící k pozorovateli (kameře). Vzhledem k tomu, že kamera je v počátku, je možné použít pouze zápornou pozoci vrcholu. Směr ke světlu (lightDirection) je dán jako pozice světla (lightPos) mínus pozice vertexu. Výpočet útlumu je založen na zákonu převrácených čtverců (inverse square law), který říká, že energie klesá s druhou mocninou vzdálenosti od zdroje světla. Tento počítaný útlum je dále možné upravit pomocí koeficientu pro konkrétní zdroj. Tyto předpočítané údaje jsou poslány do fragment shaderu, kde dojde k finálnímu výpočtu. Jednotlivé vektory jsou znázorněny na obrázku 11.

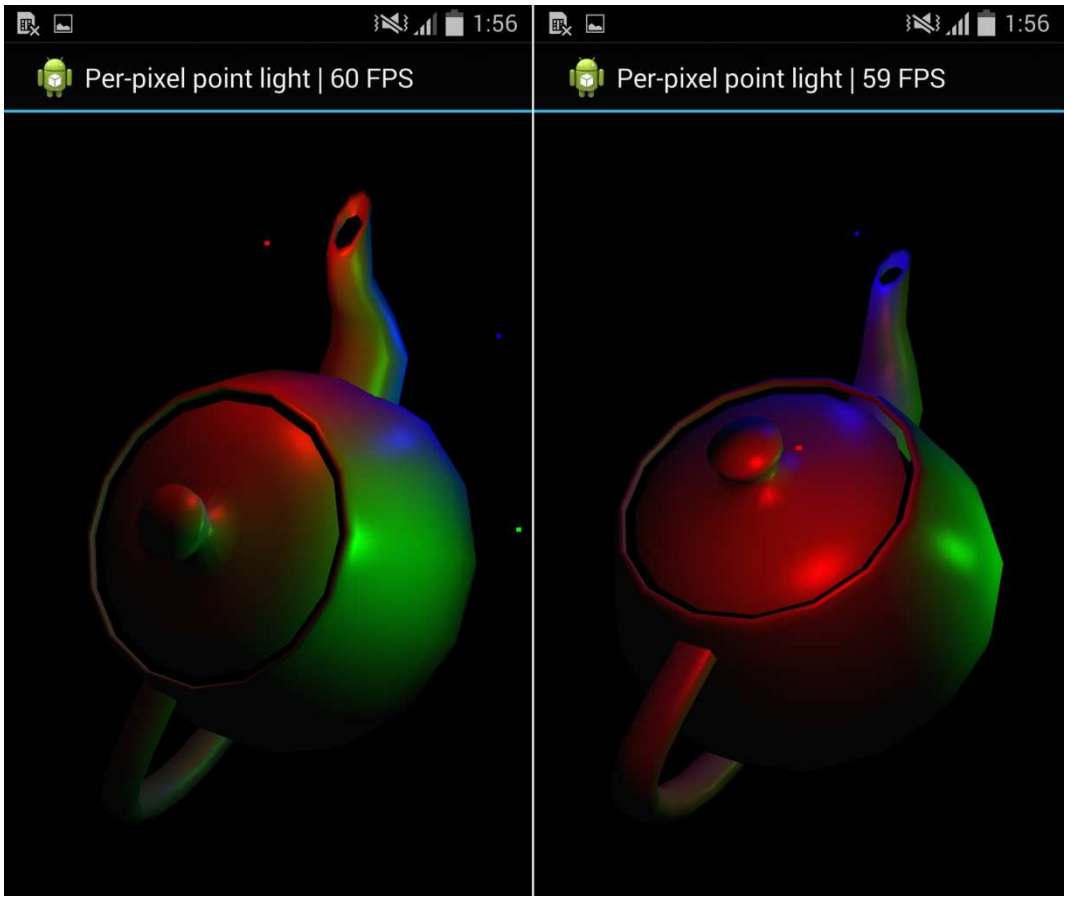


Obrázek 11: Vektory pro Blinn-Phong osvětlovací model

Pro čistý Phongův model by byl potřebný R (reflection) vektor, který se počítá jako R = 2(N dot L)N – L. Koeficient odrazu je pak závislý na úhlu mezi vektory R a V. Blinnův half vektor je jednodušší pro výpočet H = (L + V)/2 a výsledný koeficient je závislý na H a normále N.

```
//fragment shader
lowp vec3 n = normalize(normal);
lowp vec3 V = normalize(viewDir);
lowp vec3 l = normalize(lightDirection);
//Lambert difuze
ndotl = max(dot(n,l),0.0);
//half vektor
H = normalize(l + V);
//koeficient odrazu
ndoth = pow(max(dot(n,H),0.0),60.0);

//finální barva
gl_FragColor = (diffColor * ndotl * attenuation) + (specColor *
ndoth * attenuation) + ambientColor;
```

Umocnění koeficientu odrazu spekulárním exponentem (60) řídí sílu odrazivosti povrchu a udávají velikost a ostrost odrazu. Počet světel pro jeden objekt je omezen pouze výpočetní silou. Na obrázku 12 jsou tři zdroje světla. Taková implementace vyžaduje použití forcyklu a některé operace (například přístup do pole pomocí hodnoty z forcyklu) mohou dělat problémy při kompilaci shaderu. Větší množství světel (například lampy na ulici) je náročné na výkon a je vhodné využít některou z dalších metod výpočtu světla. Například deffered shading je metoda, kdy jsou nejdříve zaznamenána potřebná data do textur a následně je výpočet proveden ve screen – space (pouze na výsledném obrazu). Mobilní zařízení však zatím neposkytují vhodné nástroje (např. ukládání více textur jedním průchodem přes shadery) pro tento druh osvětlení.



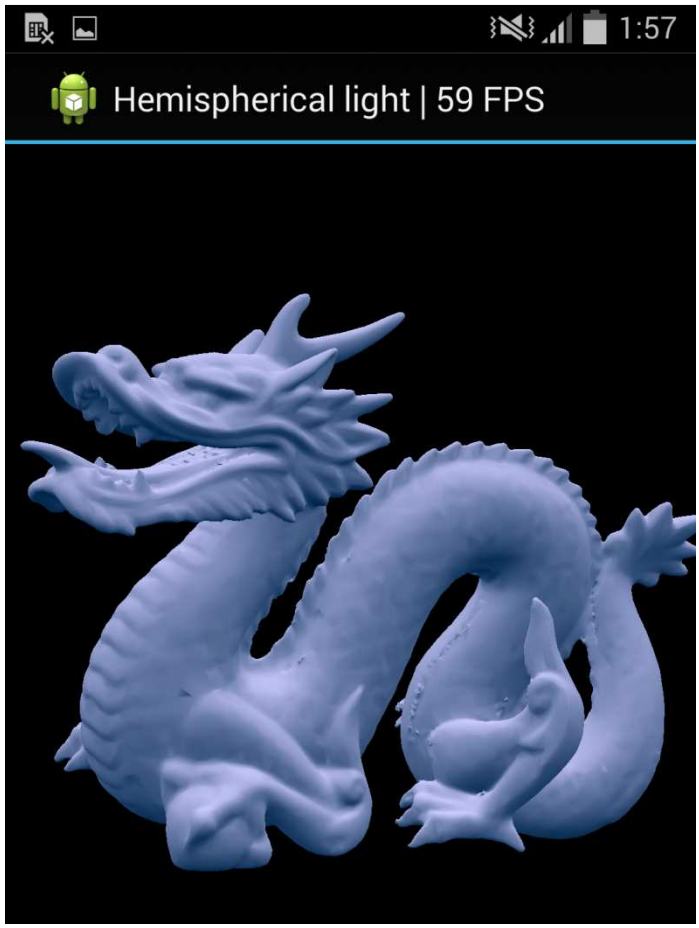
Obrázek 12: Blinn-Phong osvětlovací model se znázorněním pozic světel

4.3.3 Hemispherical light

S využitím předchozích znalostí lze vytvářet nové druhy osvětlení s novými vlastnostmi. Jedním z jednoduchých rozšíření je hemisphericke světlo [14]. Touto metodou je možné osvětlit povrch ze všech stran a použít ji tak například pro zobrazování modelů nebo jako část ambientního světla.

```
lowp vec3 n = normalize(normal);
lowp vec3 l = normalize(lightDirection);
ndotl = dot(n,l);
//přepočet z intervalu -1,1 na 0,1
intensity = (ndotl * 0.5) + 0.5;
//síla osvětlení
float i = 1.0;
//smíchání barev podle získané intenzity
lowp vec3 ambient = mix(groundColor,skyColor,intensity) * i;
```

V případě otevřené scény v krajině je možné za lightDirection dosadit vektor k obloze a s odpovídajícími barvami tak zvýšit realističnost celkového osvětlení. Objekt pak částečně „odráží“ barvu země pod sebou a oblohy nad sebou. Toho je dosaženo smícháním barev pomocí intenzity odvozené od difúzního osvětlení. Na obrázku číslo 13 je model Stanfordského draka osvětlený výše popsaným způsobem.



Obrázek 13: Zobrazení modelu draka s hemispherical light

4.3.4 Minaert shading

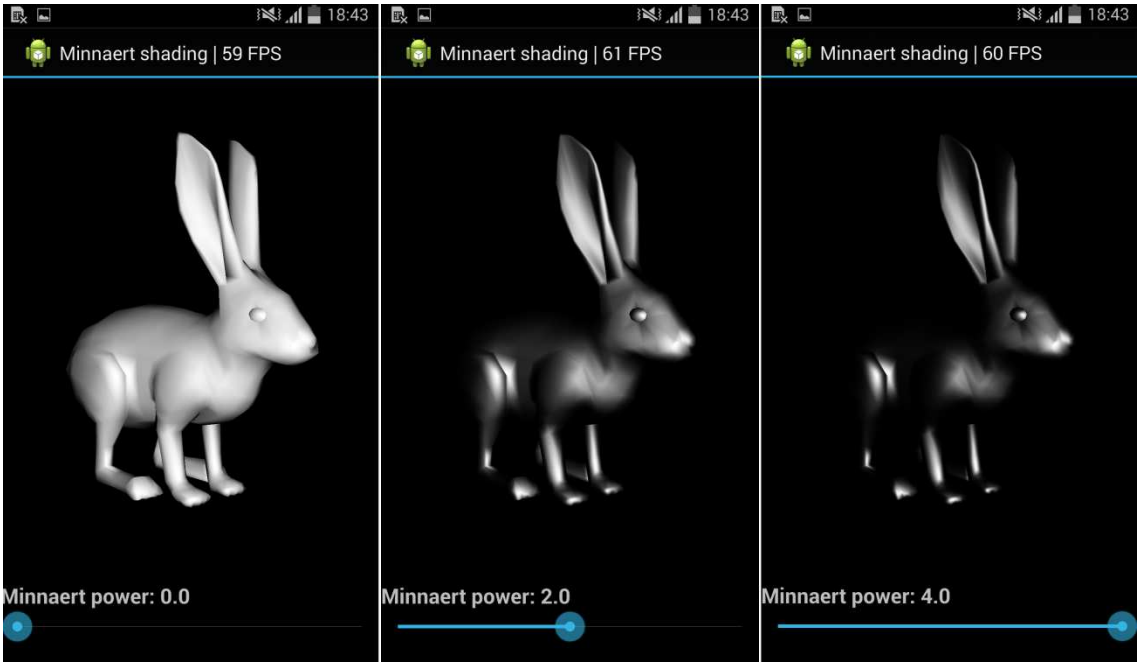
Lambertův osvětlovací model neodpovídá všem možným difúzním povrchům. Některé plochy neodrážejí světlo do všech směrů, jak Lambertův model předpokládá, ale mají určitou míru retroreflexe. To znamená, že odrážejí určité množství světla zpět k jeho zdroji. Může se jednat o drsné povrchy jako je hlína nebo některé druhy látek. Jednou ze speciálních metod pro simulaci takových povrchů je Minnaert shading [12]. Tento model je vhodný například pro simulaci sametu, protože povrch bude nejsvětlejší při směru pohledu rovnoběžném se směrem ke světlu a plochy odvrácené od světla budou naopak rychleji tmavnout. Z toho vyplývá, že model je závislý na směru pohledu a směru ke světlu. Tyto parametry lze vypočítat ve vertex shaderu:

```
vec3 position = vec3(modelview * vec4(inPosition,1.0)).xyz;
//normálový vektor
normal = (t_modelview * vec4(inNormal,0.0)).xyz;
//směr ke světlu
lightDir = (lightPos - position);
//směr pohledu
viewDir = (-position);
```

Po normalizaci těchto tří složek ve fragment shaderu dochází k výpočtu samotného osvětlení:

```
float NdotV = dot(normal,viewDir);
float NdotL = dot(normal,lightDir);
float lambertian = max(NdotL,0.0);
gl_FragColor = diffColor*pow(NdotV * NdotL,power)*lambertian;
```

Pomocí koeficientu power lze měnit sílu efektu, který upravuje Lambertův model. Při nastavení na nulu je osvětlení shodné s Lambertovým osvětlením. Implementace tohoto modelu s různým nastavením koeficientu je viditelná na obrázku číslo 14. Toto osvětlení bylo použito i v ukázce 4.12 Cloth simulation.



Obrázek 14: Ukázka osvětlovacího modelu Minnaert
Zdroj: Model byl získán na http://tf3dm.com

Pro realizaci sametového povrchu lze spojit tento typ osvětlení s normal mappingem, který je představen v kapitole 4.4, pro silnější efekt zvlněné látky.

4.3.5 Cook–Torrance shading

Stejně jako Lambertův model nepopisuje všechny možnosti difúzního osvětlení, tak Blinn-Phong model nestačí pro všechny možnosti odlesků. Blíže realitě je Cook-Torrance model [12], který simuluje povrch jako soustavu velmi malých ploch (mikrofacets). Tyto plochy mohou odrážet světlo do mnoha směrů, nebo ho dokonce blokovat. Pokud plochy blokují světlo na cestě k jiným plochám, je používán termín shadowing. Efekt, kdy mikrofacets blokují odražené světlo od okolí, se nazývá masking. Model zahrnuje i Fresnelovy rovnice pro řízení odrazu světla v různých úhlech a je vhodný pro simulace kovových povrchů. Základní rovnice vypadá následovně:

$$I = I_i * \left[\frac{F * D * G}{(N \cdot L) * (N \cdot V)}\right]$$

Parametry ve jmenovateli jsou již známé NdotL (skalární součin normály a směru světla) a NdotV (skalární součin normály a směru pohledu). Důležité jsou parametry v čitateli, které definují jednotlivé světelné efekty. F je zástupce pro Fresnel efekt, D pro roughness (hrubost) a G je geometrický faktor.

Geometrický faktor (G) je odpovědný za shadowing a masking. Pomáhá určit množství spekulárního odrazu v určitém směru. Na rozdíl od ostatních faktorů neobsahuje parametry pro definici povrchu. Jeho rozepsaný tvar je:

$$G = min\left\{1, \frac{2(N \cdot H)(N \cdot V)}{(V \cdot H)}, \frac{2(N \cdot H)(N \cdot L)}{(V \cdot H)}\right\}$$

Parametr H označuje již dříve zmiňovaný half vektor. První část funkce min je případ, kdy je světlo odraženo bez přerušení. Druhá část je pro masking a poslední pro shadowing.

Fresnel faktor (F) určuje množství světla odraženého každou plochou (mikrofacet) a definuje kovový vzhled povrchu. Původní Fresnelova rovnice je příliš výpočetně náročná, proto se používá aproximace:

$$F = R + (1 - R) * \left(1 - (N \cdot L)\right)^5$$

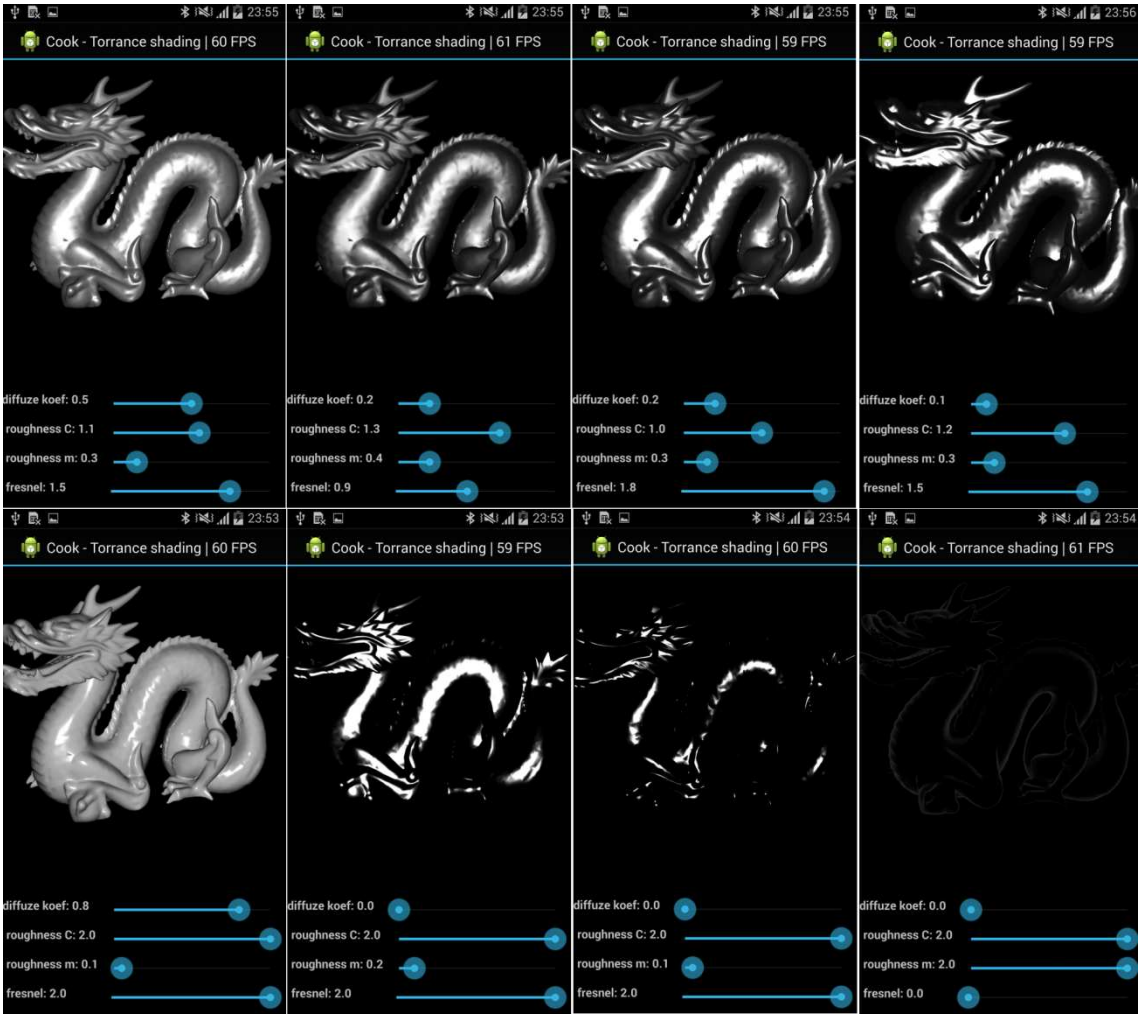
Nový parametr R slouží pro řízení Fresnelova faktoru. Faktor hrubosti (D) popisuje plochy stejně natočených mikrofacets pomocí Beckmannovy distribuční funkce. V shaderu je použita jednodušší Blinnova distribuční funkce.

$$D = \frac{1}{m^2(N \cdot H)^4} e^{\left(\frac{(N \cdot H)^2 - 1}{m^2 (N \cdot H)^2}\right)}$$

Kombinací těchto tří faktorů vzniká konečné osvětlení. Implementace ve fragment shaderu je následující:

```
float NdotH = dot(N, H);
float alpha = acos(NdotH);
//hrubost povrchu, C a m jsou volitelné parametry
float D = C * exp(-(pow(alpha / m, 2.0)));

float NdotV = dot(N, V);
float VdotH = dot(H, V);
float NdotL = dot(L, N);
float G1=2.0*NdotH * NdotV / VdotH;
float G2=2.0*NdotH * NdotL / VdotH;
//geometrický faktor
float G = min(1.0, max(0.0, min(G1, G2)));

//fresnel aproximace
//na obrázku 12 je parametr R zadáván jako fresnel
float F = R +(1.0 - R) * pow(1.0 - NdotL, 5.0);

float spec = (F * D * G) /(NdotL * NdotV);
//x slouží pro propojení s difůzním osvětlením (difuze koef)
gl_FragColor = max(NdotL,0.0)*diffColor*(x + spec*(1.0 - x));
```

Řízením těchto tří faktorů a kombinací s difúzním osvětlením lze dosáhnout zobrazení různých kovů, ale i keramiky nebo plastu. Další možností jak zvýšit realističnost kovového povrchu je přidání zrcadlového odrazu okolního prostředí. Toto téma se nachází v kapitole 4.6 Skybox, environment mapping. Možnosti aktuální implementace Cook-Torrance modelu jsou zobrazeny na obrázku 15.



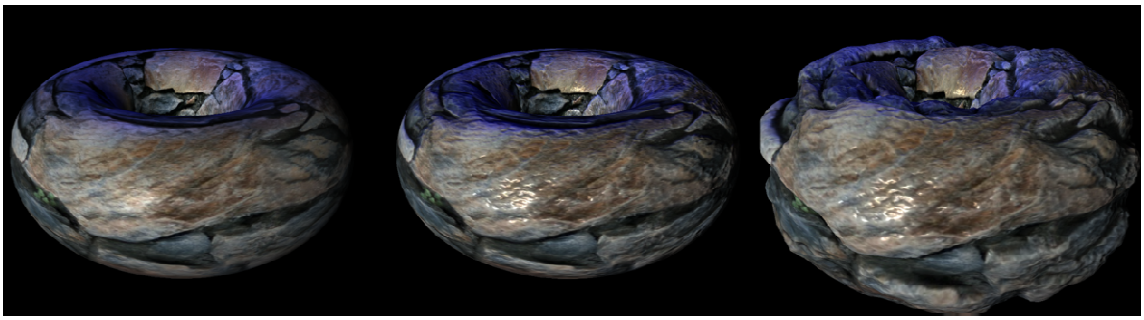
Obrázek 15: Cook-Torrance model s různým nastavením faktorů

Dalším zajímavým faktorem ovlivňujícím osvětlení je anizotropie. Dosavadní modely počítají s izotropním povrchem, který se vyznačuje stejnou reakcí při rotaci kolem normály. Některé materiály ovšem odrážejí světlo jiným způsobem. Jedná se například o broušený kov nebo vlasy. I lidská kůže se vyznačuje určitou mírou anizotropie. K reálnému vzezření objektů mohou pomoci i efekty jako subsurface scattering. Touto metodou se řeší síření světla pod povrchem a používá se pro simulaci kůže nebo mramoru. Možnosti rozšíření způsobu osvětlování je mnoho a dnešní mobilní zařízení jsou dostatečně výkonná pro jejich realizaci. Stále však platí, že složité shadery ubírají určitou část výkonu.

4.4 Normal mapping

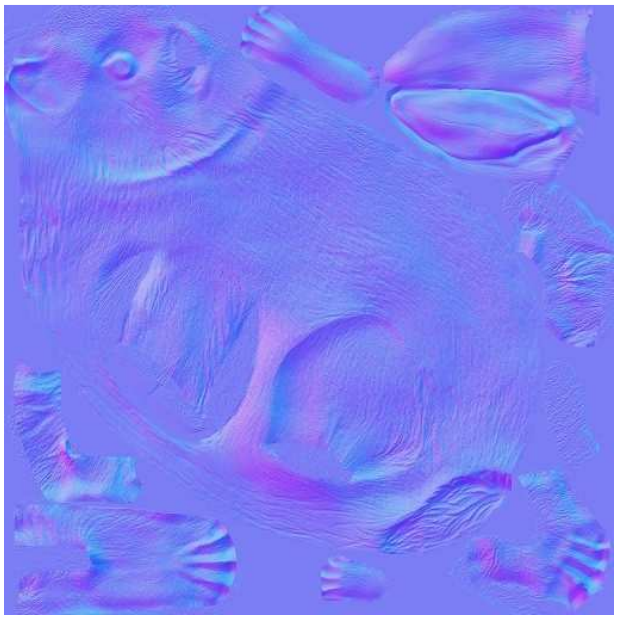
Přímou souvislost se světlem mají metody, které upravují jeho chování na povrchu. Pro tyto efekty je nutné použít speciální texturu, která obsahuje potřebné informace [15]. Nejstarším efektem je bump mapping, který používá texturu v odstínech šedi [16]. Taková textura (heightmap) nese informaci o výšce povrchu a je možné ji použít k upravení normálového vektoru pro simulaci detailních nerovností. Hlavní výhodou takových metod je použití se zjednodušenou geometrií, které pak dodávají potřebný detail. Bump mapping je však nahrazován dokonalejším normal mappingem. Tento efekt používá barevnou texturu a dokáže na rozdíl od bump mappingu definovat směr natočení normály. Ještě pokročilejší je paralax mapping (nebo také virtual displacement mapping), který přidává posuny textury při pohybu pro výraznější efekt a kombinuje heightmapu i normal mapu. Nevýhodou je, že v některých úhlech pohledu je znatelná původní neovlivněná geometrie modelu.

Na rozdíl od těchto „fake“ efektů je displacement mapping jedním ze způsobů jak skutečně upravit vzhled povrchu. Využívá heightmapu k reálnému posunu vrcholů a společně s normal mappingem může produkovat velmi detailní a realistické změny. Obzvlášť na mobilních zařízeních je však nevýhodou nutnost použití velkého množství vertexů. To znamená, že model musí být načten ve větším rozlišení, aby mohl být vytvořen reálný detail. Tato metoda může být užitečná pro změnu modelu. Například postupná změna částí modelu (výrůstky, poškození), kdy je detail uložen v textuře a není nutné použít mnoho samostatných modelů. Častým využitím je také modelování terénu popsané v kapitole 4.7. Normal mapping a displacement mapping je ukázán v PC implementaci na obrázku číslo 16, kde jsou postupně přidávány efekty. První zleva je pouze textura s osvětlením, uprostřed je přidán normal mapping a vpravo je přidán displacement mapping.



Obrázek 16: Ukázka efektů úprav povrchu

Nejpoužívanějším a nejvhodnějším efektem je normal mapping. Poskytuje výrazné zlepšení detailu a zároveň nepožaduje vysoké základní rozlišení modelu. První požadavek pro realizaci normal mappingu je model s normálou a tangent vektorem. Tangent je vektor kolmý na normálu a nejlepší řešení je získat ho přímo z modelovacího software spolu s modelem. Dále je nutné vytvořit texturu s definicí normál. Taková textura je často generována z detailnějšího modelu, ze kterého přebere nerovnosti povrchu. Tyto nerovnosti a detaily se po aplikaci objeví na zjednodušeném modelu s méně polygony. Obrázek 17 ukazuje, jak vypadá běžná normal mapa. Povrch modelu je rozložen a pro nanesení zpět na model při texturování slouží texturovací souřadnice (u, v). U normal mapy je normála uložena v běžném RGB formátu, kdy každý kanál obsahuje informaci o jedné z os (x, y, z). Normálový vektor je ale nutné definovat vzhledem k nějakému souřadnicovému systému. Často se používá takový systém, kdy osy obrázku (u, v) se zarovnají s osami systému, vůči kterému je definována normála. To vysvětluje proč je obrázek většinou v modré barvě. Modrá barva definovaná B (blue) kanálem je přiřazena k ose „z“ směřující pryč od povrchu. Ostatní kanály pak upravují směr normály a vytvářejí nerovnosti. Tento prostor výpočtů se nazývá tangent space. Výpočet samotného osvětlení pak probíhá právě v tomto prostoru a je nutné do něj převést ostatní důležité informace.



Obrázek 17: Normal mapa pro model králíka

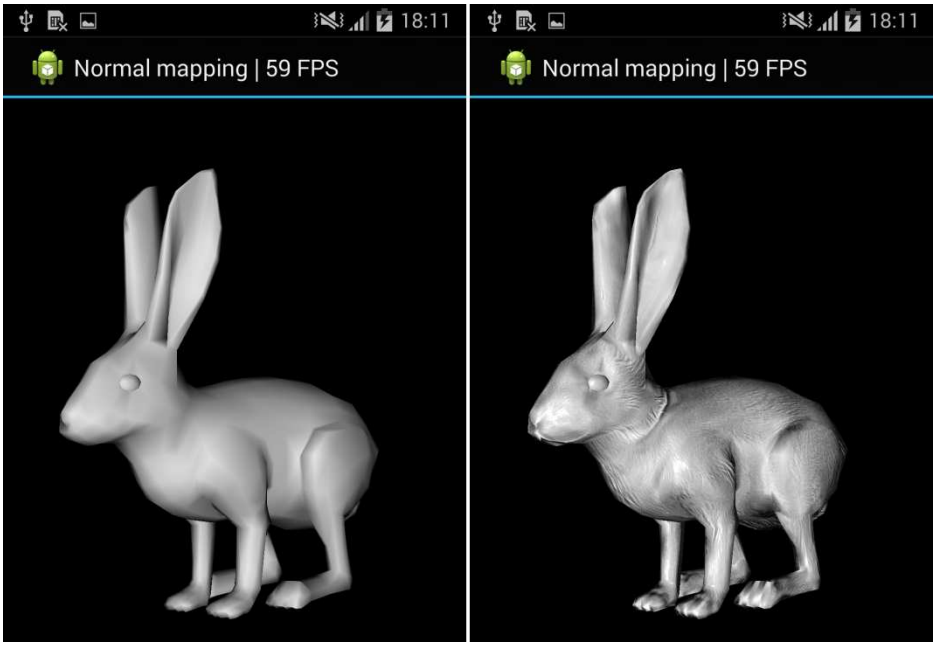
Ve vertex shaderu jsou informace potřebné k osvětlení přeneseny do z eye (kamera) space do tangent space pomocí normály, tangentu a bitangentu. Bitangent je další vektor, který je kolmý na oba předchozí a lze ho získat pomocí vektorového součinu normály a tangentu.

```
vec3 normal = normalize(modelview * vec4(inNormal,0.0)).xyz;
vec3 tangent = normalize(modelview * vec4(inTangent,0.0)).xyz;
vec3 bitangent = cross(tangent,normal);

vec3 pos = vec3(modelview * vec4(inPosition,1.0)).xyz;
vec3 ldir = normalize(lightPos - pos);

//transformace light direction z eye space do tangent space
lightDir_ts.x = dot(ldir,tangent);
lightDir_ts.y = dot(ldir,bitangent);
lightDir_ts.z = dot(ldir,normal);

//position z eye space do tangent space
position.x = dot(pos,tangent);
position.y = dot(pos,bitangent);
position.z = dot(pos,normal);
position = -normalize(position); //směr pohledu
```

S informacemi v tangent space stačí už jen získat normálový vektor (uložen v tangent space) a provést výpočet osvětlení ve fragment shaderu.

```
//převod RGB na normálu
vec3 normal = texture2D(normalTex,texCoord).xyz * 2.0 - 1.0;
//osvětlení
float ndotl = max(dot(lightDir_ts, normal),0.0);
```

Výsledný efekt je viditelný na následujícím obrázku číslo 18.



Obrázek 18: Ukázka normal mappingu, vlevo pro srovnání bez efektu

4.5 Shadow mapping

Stín je důležitou složkou reálného světa a umožňuje vnímat velikost a vzdálenost předmětů. Aplikovat stín lze dvěma způsoby. První je zanesení stínu přímo do textury při modelování objektů a scény. Výhodou je, že lze dosáhnout dokonalého efektu bez nutnosti jakéhokoliv výpočtu. Od 3D scény se však většinou vyžaduje aplikace pohybu objektů a s tím statické stíny neudrží krok. Použít pak lze druhou náročnější metodu – dynamické stíny.

Stín vzniká, pokud se mezi světlem a plochou nachází jiný objekt. Vytvořit stín je tedy nutné z pohledu světla a následně ho použít při běžném vykreslení. Zde přichází na řadu realizace víceprůchodových algoritmů. V mnoha případech je nutné vyrenderovat scénu vícekrát pro jeden finální efekt složený z výsledků těchto průchodů. Shadow mapping je typickým případem dvou průchodového algoritmu. Při prvním průchodu je scéna vykreslena z pohledu zdroje světla. Výsledek je uložen do textury, která se použije při druhém průchodu. Místo běžného barevného obrázku se však používá depth mapa, která zaznamenává hloubku objektů ve scéně nejblíže pozorovateli. Díky tomu lze při vykreslování stínu porovnávat, jestli aktuální pixel je za pixelem v shadow textuře (depth mapě). Pokud ano, je vykreslen stín ztmavením barvy pixelu [15].

Texturu z prvního průchodu je ovšem nutné nějakým způsobem uložit. K tomu slouží rozšíření OpenGL pro off-screen rendering Frame Buffer Object (FBO). Víceprůchodové algoritmy lze dále využít například pro vykreslení více obrazů jedné scény z různých úhlů (kamerový systém), realizaci odrazu na vodní hladině (kapitola 4.9) nebo post processing efekty a filtry (kapitola 4.11).

Nejprve je vytvořen FBO pro první průchod při načítání aplikace. FBO má několik základních částí. Nejdůležitější je color a depth attachement, které mohou sloužit pro uložení textury z fragment shaderu. Pro použití FBO je nutné ho nabindovat a nastavit velikost obrazu. Pseudokód pro renderer je následující:

```
//bindování dříve vytvořeného FBO a nastavení viewportu pro
//přesměrování vykreslování do tohoto FBO
GLES20.glBindFramebuffer(GLES20.GL_FRAMEBUFFER, frameBufferID);
GLES20.glViewport(0, 0, shadowWidth, shadowHeight);
```

```
//vyčištění předchozí depth mapy a nastavení ořezávání
//přivrácených ploch
GLES20.glClear( GLES20.GL_DEPTH_BUFFER_BIT );
//důležité jsou odvrácené plochy vytvářející samotný stín
GLES20.glCullFace( GLES20.GL_FRONT );

//vykreslení modelu se shaderem pro vytvoření depth mapy
GLES20.glUseProgram(shaderProgram);
GLES20.glUniformMatrix4fv(proj, 1, false, lightMatrix, 0);
GLES20.glUniformMatrix4fv(mv, 1, false, modelMatrix, 0);
model.modelDraw(GL10.GL_TRIANGLES, shaderProgram);
```

Light matrix je view projection matice, která definuje pohled ze směru světla. Tato část kódu má za úkol vytvořit stín. Druhá část (druhý průchod) stín vykresluje společně s objekty.

```
//nastavení vykreslování na obrazovku
GLES20.glBindFramebuffer( GLES20.GL_FRAMEBUFFER, 0);
GLES20.glCullFace(GLES20.GL_BACK);

//čístění a nastavení view portu na normální velikost
GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT |
GLES20.GL_DEPTH_BUFFER_BIT);
GLES20.glViewport(0, 0, mWidth, mHeight);

//vykreslení plochy pod modelem, na kterou je vrhán stín
GLES20.glUseProgram(shaderProgram2);

//připojení textury vytvořené prvním průchodem
GLES20.glActiveTexture(GLES20.GL_TEXTURE0);
GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, depthBuffer[0]);
GLES20.glUniform1i(GLES20.glGetUniformLocation(shaderProgram2,
"DepthTexture"), 0);

//matice pro další výpočty
GLES20.glUniform3fv(lpos,1,lightPos,0);
GLES20.glUniformMatrix4fv(n_mvp, 1, false, proj, 0);
GLES20.glUniformMatrix4fv(n_mv, 1, false, modelView, 0);
```

```
GLES20.glUniformMatrix4fv(mvp_l, 1, false, lightMatrixBias, 0);

Model2.modelDraw(GL10.GL_TRIANGLE_STRIP, shaderProgram2);

//vykreslení modelu Suzzane
GLES20.glUseProgram(shaderProgram3);

//matice pro další výpočty
GLES20.glUniform3fv(lpos2,1,lightPos,0);
GLES20.glUniformMatrix4fv(n_mvp2, 1, false, proj, 0);
GLES20.glUniformMatrix4fv(n_mv2, 1, false, modelView, 0);

model.modelDraw(GL10.GL_TRIANGLES, shaderProgram3);
```

Vertex shader pro vytvoření stínu (depth / shadow mapy) je primitivní. Model Suzzane je vykreslen z pohledu světla.

```
mediump vec3 pos = (modelMatrix * vec4(inPosition, 1.0)).xyz;
gl_Position = lightMatrix * vec4(pos, 1.0);
```

Fragment shader obsahuje pouze prázdnou metodu main(). Vertex shader pro druhý průchod:

```
vec3 position = (modelview * vec4(inPosition,1.0)).xyz;
vec3 normal = (modelview * vec4(inNormal,0.0)).xyz;
vec3 lightDirection = (lightPos - position);
vec4 shadowMapCoord = (lightMatrixBias * vec4(inPosition,1.0));
gl_Position = projection * vec4(position, 1.0);
```

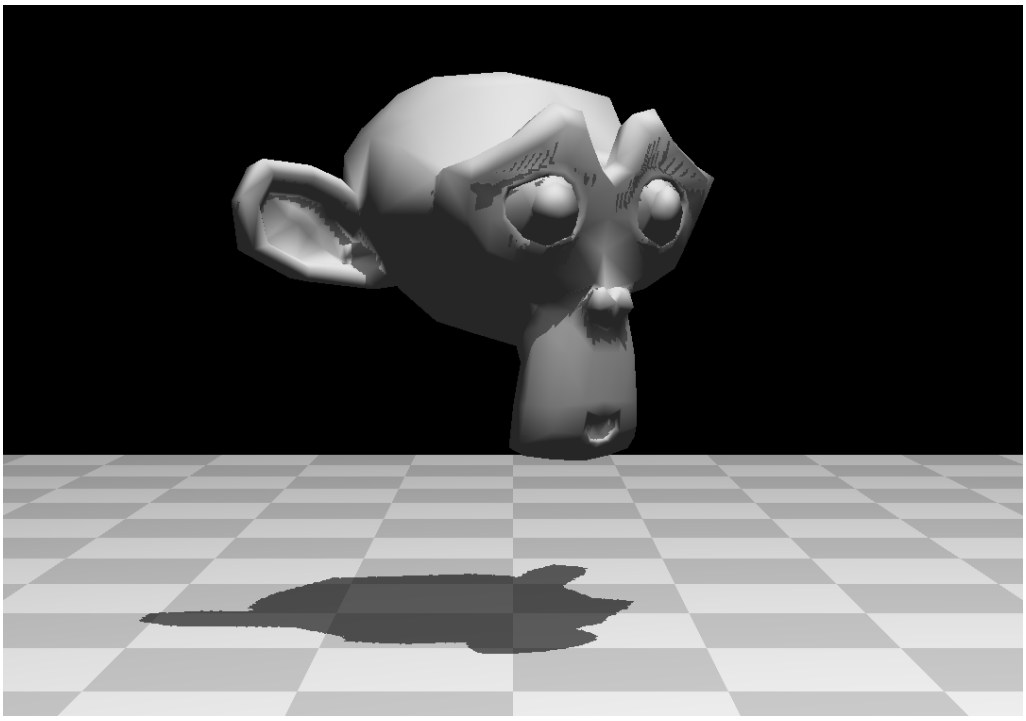
Jediný zvláštní řádek v tomto shaderu je pro výpočet shadowMapCoord. Výsledkem tohoto řádku je pozice vrcholu z pohledu světla. Jedná se o původní view projection matici světla přednásobenou s bias maticí. Bias matice slouží pro přesunutí z object space do screen (image) space. To je nutné kvůli rozdílným souřadnicím obrazovky a textury. Výsledné souřadnice pak lze použít přímo pro čtení dat z depth (shadow) mapy ve fragment shaderu.

```
//hloubka aktuálního pixelu ve scéně
float depth = shadowMapCoord.z/shadowMapCoord.w;
//hloubka pixelu nejblíže světlu
```

```
float depthLight = texture2DProj(DepthTexture,
                                 shadowMapCoord).z;
float shadow = depth <= depthLight ? 1.0 : 0.4;
gl_FragColor =color * ndotl * shadow;
```

Proměnná depth obsahuje vzdálenost (hloubku) aktuálního pixelu od zdroje světla. Do proměnné depthLight je uložena hloubka pixelu z textury. V textuře je uložen pixel nejblíže ke světlu. Jako poslední stačí porovnat, zda je aktuální pixel za hodnotou z shadow textury a má se vykreslit stín. Na obrázku 19 je model Suzzane z programu Blender vrhající stín na plochu pod ním. Navíc byl aplikován stín i na samotný model pro znázornění některých problémů, které shadow mapping přináší.

Prvním problémem je shadow acne. To je způsobeno nedostatečnou přesností informací ukládaných v shadow textuře. Typickým projevem je šrafování ploch, které nemají být zastíněné. Dalším problémem je nepřesné mapování stínové textury na renderovaný obraz. Pokud není rozlišení jedna ku jedné, vzniká schodovitý okraj stínů. Při snaze řešit tyto problémy může dojít i k posunu stínu (Petter Panning).



Obrázek 19: Ukázka shadow mapping s modelem Suzzane

Z mnoha vylepšení lze uvést front face culling pro redukci shadow acne na osvětlených plochách nebo nastavení far a near ořezávacích rovin co nejblíže k objektům. Dále se používá přičtení malé konstanty (bias) k hloubce pixelu.

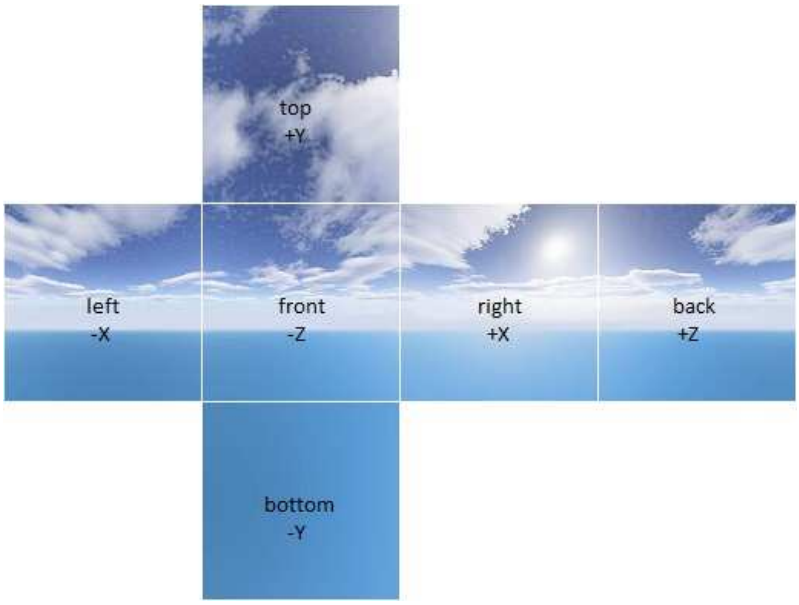
Pro zjemnění okrajů stínů existuje více metod. Nejjednodušší je PCF (Percentage Closer Filtering), která zjistí zastínění pixelů v okolí aktuálního a určí tak momentální hodnotu stínu. Tato metoda vyžaduje velké množství přístupů do textury a je výkonnostně náročná. Zajímavější může být VSM (Variance Shadow Maps) umožňující využít vestavěného OpenGL filtrování textur, ale je nutné ukládat upravené hodnoty o hloubce. Zvýšit rozlišení stínů lze použitím cascaded shadow mappingu. Vzhledem k perspektivní transformaci scény je možné vytvořit více shadow textur pokrývajících různě velké plochy a rozmístit je ve scéně. Stíny blíže pozorovateli tak mohou být kvalitnější než ty na druhém konci scény.

Vytvořit realisticky vypadající real-time stíny je obecně obtížně vyřešitelný problém. Vhodné je kombinovat více metod společně s použitím statických stínů. I v nejjednodušším případě je nutné vykreslit v prvním průchodu geometrii pro vytvoření stínů a to vede k nutnému snížení maximálního množství objektů ve finální scéně.

4.6 Skybox, environment mapping

Ve všech předchozích příkladech byl zobrazen model na černém pozadí. Ve skutečném světě je však v pozadí objektů obloha, moře, budovy nebo hory. Pro vytvoření efektu rozsáhlého světa s pozadím lze využít skybox. Stejně jako atmosféra obaluje planetu je i skybox obalem scény. Je možné použít tvar koule, polokoule, nebo krychle se speciální texturou. Pozorovatel umístěný uprostřed skyboxu pak má dojem skutečného prostředí.

Textura pro takový panoramatický efekt při použití s krychlí se nazývá cube map. Krychle musí být větší než celá scéna, nebo se musí pohybovat společně s pozorovatelem. Zároveň je nutné vykreslit skybox za všemi ostatními objekty. Cube map je načítána jako šest samostatných textur, kdy každá má zorný úhel 90 stupňů a navazují na sebe [10]. Příklad takové textury je na obrázku 20.



Obrázek 20: Vzhled cube map textury [17]

Nevýhodou je, že veškeré části skyboxu se zdají být stále ve stejné vzdálenosti bez ohledu na pohyb pozorovatele. Také se jedná o statickou texturu, takže nelze počítat například s pohybem mraků. Rozlišení textur navíc musí být často relativně vysoké a stojí tak mnoho paměti. Vylepšením může být kombinace skyboxu se speciální 3D scénou zobrazovanou také za veškerými modely.

Při načítání textur do OpenGL je nutné je přiřadit k jednotlivým stranám krychle.

```
GLES20.glBindTexture(GLES20.GL_TEXTURE_CUBE_MAP,
textureObjectIds[0]);
GLUtils.texImage2D(GLES20.GL_TEXTURE_CUBE_MAP_NEGATIVE_X, 0,
cubeBitmaps[0], 0);
GLUtils.texImage2D(GLES20.GL_TEXTURE_CUBE_MAP_POSITIVE_X, 0,
cubeBitmaps[1], 0);
GLUtils.texImage2D(GLES20.GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, 0,
cubeBitmaps[2], 0);
GLUtils.texImage2D(GLES20.GL_TEXTURE_CUBE_MAP_POSITIVE_Y, 0,
cubeBitmaps[3], 0);
GLUtils.texImage2D(GLES20.GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, 0,
cubeBitmaps[4], 0);
GLUtils.texImage2D(GLES20.GL_TEXTURE_CUBE_MAP_POSITIVE_Z, 0,
cubeBitmaps[5], 0);
```

Kvůli tomuto přiřazování je nutné textury ve správném pořadí i načítat. Pořadí je left, right, bottom, top, front a back. Po dokončení vytváření skybox textury je možné vrátit id (textureObjectIds) a dále zpracovat obraz v shaderech.

```
//vertex shader
v_Position = inPosition;
v_Position.z = -v_Position.z;
//doplnění pozice o nulu, protože je potřeba pouze rotace
gl_Position = modelViewProjection * vec4(inPosition, 0.0);
gl_Position = gl_Position.xyww;

//fragment shader
gl_FragColor = textureCube(cubeMap, vPosition);
```

Proměnná v_Position nese informaci pro nalezení správného pixelu ve správné textuře. Souřadnice „z“ se otáčí pro správné natočení skyboxu protože ten je počítán v left-handed coordinate space. Poslední řádek vertex shaderu zajistí, že skybox bude vykreslen za všechny ostatní modely, protože „z“ souřadnice bude maximální možná. Ve fragment shaderu je použit třísložkový vektor, který slouží

k vržení paprsku do stěny krychle a nalezení správného pixelu. Tato funkcionalita je zajištěna pmocí textureCube().

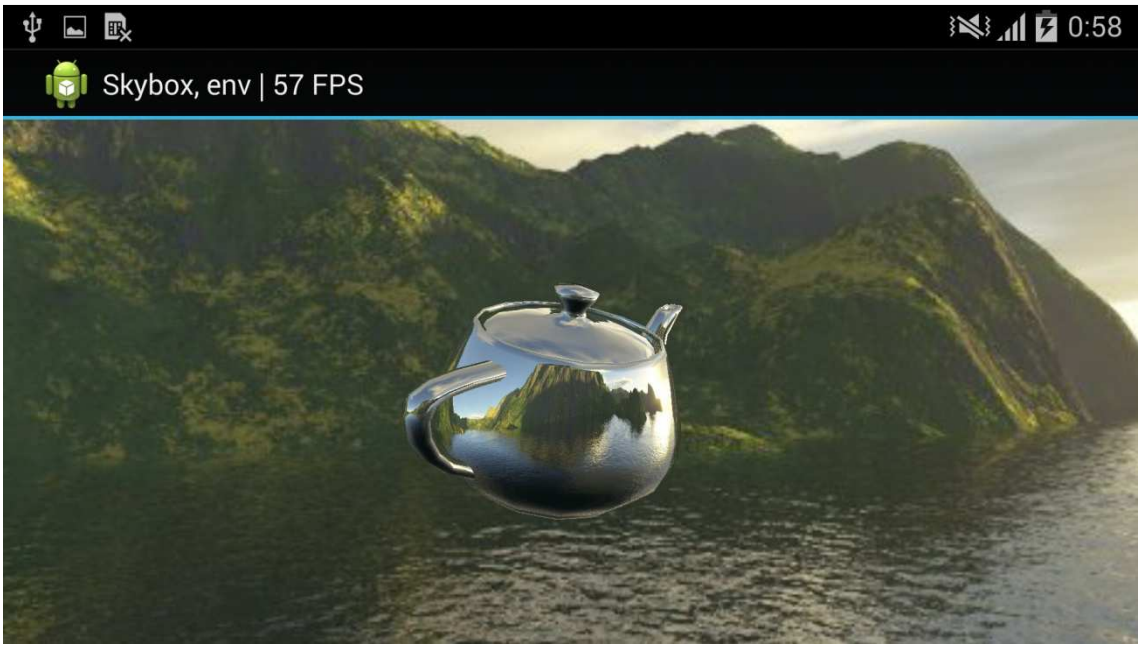
Další využití cube map je pro vytvoření zrcadlového všesměrového odrazu (reflection), nebo pro lom vlnění (refraction) pro průhledná tělesa [18]. Reflection vektor není stejný jako na obrázku číslo 11 v kapitole 4.3.2 Blinn-Phong osvětlovací model, kde je to vektor odrazu světla. Zde je tento vektor označením pro odraz paprsku z kamery. Při výpočtu odrazu slouží jako vektor do textureCube() místo původní pozice. Metodě používající reflection se říká environment mapping.

```
//vertex shader
vec3 position = vec3(model * vec4(inPosition,1.0)).xyz;
normal = (model * vec4(inNormal,0.0)).xyz;
//eye (incident) vektor, opak vektoru pohledu
eye = (position -eyePosition);
gl_Position = projection * view * vec4(position, 1.0);
```

Ve vertex shaderu jsou normála a eye vektor počítány ve world space kvůli správnému čtení z cube map. Při výpočtu v eye space by bylo nutné použít inverzní view matici na reflection vektor pro jeho přesun do world space.

```
//fragment shader
lowp vec3 n = normalize(normal);
lowp vec3 e = normalize(eye);
//reflection vektor
vec3 R = (reflect(e, n));
R.z = -R.z;
//čtení z cube map pomocí reflection vektoru
lowp vec4 envColor = textureCube(cubeMap,R);
gl_FragColor = envColor;
```

Reflection vektor představuje paprsek z kamery odražený od povrchu modelu. Tento paprsek je následně použit pro přečtení dat v textuře. Skybox a environment mapping je ukázán na obrázku 21.



Obrázek 21: Zobrazení environment mapping na konvici a skyboxu

Pro realistický zrcadlový odraz v kompletní scéně běžná cube map bohužel nestačí. Proto je nutné jednotlivé textury vytvořit pomocí šesti průchodů s nastaveným pohledem do patřičných stran, které budou uloženy do textur pro aktuální cube map a umožní vznik dynamic environment mapping. Tímto způsobem je možné vytvářet i cube shadow map pro vykreslení stínu při použití s point light.

Skybox lze dále obohatit o prvky zvyšující realističnost scény. Například rovná plocha v horní části s pohyblivou texturou mraků nebo jednoduché uživatelem nedosažitelné 3D objekty zajišťující pocit hloubky při pohybu.

4.7 Vykreslení terénu

Předchozí kapitola popsala způsob vytvoření pozadí pro otevřenou scénu. Takový svět je však stále prázdný a objekty levitují v prostoru. Je proto vhodné přidat nějaký podklad. Může se jednat o převážně rovnou plochu pro město, nebo zvlněný terén pro krajinu. V kapitole 4.4 Normal mapping byla zmíněna height map textura nesoucí informace o výšce. Jedním z častých použití takové textury je vytváření terénu.

Prvním krokem je příprava sítě vrcholů s rovnoměrným rozestupem, kterou lze vytvořit pomocí dvou vnořených forcyklů. Následně je nutné vygenerovat indexy pro spojení vrcholů. Takto lze dostat libovolně detailní rovnou plochu. Způsoby jak jí deformovat pomocí height mapy jsou dva. První je deformace ve vertex shaderu, kde dekódováním hodnoty z textury lze posunout každý vrchol na požadovanou pozici. Způsob velmi podobný displacement mappingu v OpenGL ES 2.0 je bohužel nepoužitelný, protože tato verze OpenGL neumožňuje čtení z textury ve vertex shaderu.

Texturu je však možné použít i na procesoru a předpřipravit tak model terénu [10]. Příklad height textury je na obrázku 22.

Obrázek 22: Ukázka height map textury

Každý pixel reprezentuje výšku terénu. Černá barva je pro minimální výšku a bílá pro maximální. Model terénu je vytvořen pomocí uvedeného pseudokódu.

```
//načtení height textury
Bitmap bitmap = BitmapFactory.decodeResource(
                context.getResources(),HeightMap);
width = bitmap.getWidth();
height = bitmap.getHeight();
if (width * height > 65536) {
     throw new RuntimeException("Heightmap is too large for the
     index buffer.");
}
```

Je nutné zkontrolovat velikost textury, protože pro indexy se v OpenGL ES používá datový typ unsigned short (pouze někteří výrobci podporují int) a ten má rozsah 0 až 65 535. Tímto je dána maximální velikost jedné mapy a všech modelů vůbec. Následuje příprava jednotlivých pixelů z bitmapy do jednorozměrného intového pole.

//získání pixelů

```
final int[] pixels = new int[width * height];
bitmap.getPixels(pixels, 0, width, 0, 0, width, height);
//bitmapa již není potřeba
bitmap.recycle();
//každý vrchol má tři souřadnice
final float[] heightmapVertices =
                          new float[width * height * 3];
//generování vrcholů
int offset = 0;
for (int row = 0; row < height; row++) {
  for (int col = 0; col < width; col++) {
     final float xPosition = ((float)col / (float)(width - 1)) -
                          0.5f;
     //přepočet z pixelu na vrchol
     final float yPosition =(float)Color.red( pixels[(row *
                          height) + col]) / (float)255;

     final float zPosition = ((float)row / (float)(height - 1))
                          - 0.5f;
     heightmapVertices[offset++] = xPosition;
     heightmapVertices[offset++] = yPosition;
```

```
      heightmapVertices[offset++] = zPosition;
    }
  }
  //vygenerování indexů pro spojení vrcholů a normálových vektorů
  createIndexData();
  createNormalData();
  createBuffers(heightmapVertices, indices, normals);
```

Po získání pole pixelů jsou generovány vrcholy. Model bude mít rozsah -0.5 až 0.5 na osách x a z. Na ose y je pozice upravena hodnotou odpovídajícího pixelu.Height textura je v odstínech šedi, tedy každý kanál má stejnou hodnotu 0 až 255. Lze tedy libovolný kanál vzít a po vydělení 255 dostat výšku 0 až 1.

Nyní je připraven model terénu a je nutné ho přijatelně pokrýt texturou. Běžná krajina je často složena z různého povrchu (tráva, skály, písek, hlína). To je možné simulovat pomocí více opakujících se textur, které se použijí na základě nějakého pravidla. Tím může být výška a sklon terénu. Tak lze definovat přechody z nízkých oblastí s pískem do výše položených travnatých plání až ke skalnatým stěnám. Fragment shader využívající sklon terénu může vypadat následovně.

```
//příprava barev z textur
vec4 grassColor = texture2D(grass, position.xz);
vec4 rockColor = texture2D(rock, position.xz);
//získání hodnoty sklonu terénu
mediump float a = smoothstep(0.8,0.9, max(dot(
                  normal, vec3(0.0,1.0,0.0)),0.0));
//namixování barev podle sklonu
mediump vec4 finalColor = mix(rockColor,grassColor,a);
```

Funkce smoothstep vrací hodnotu od 0 do 1 podle třetího parametru. Ten lze získat podobně jako u difúzního osvětlení. Tedy úhel mezi normálou povrchu a up vektorem (vektor směřující vzhůru). První dva parametry jsou hranice pro třetí parametr. Pokud tedy sklon terénu pod 0.8 je vrácena 0. Pro sklon 0.85 je vrácena hodnota 0.5 a pro hodnoty nad 0.9 je vrácena 1. Funkce mix pak podle finální hodnoty namíchá barvy textur dohromady. Ve výsledku je pro sklon do 0.8 použita textura skal, pro sklon nad 0.9 textura trávy a mezi hodnotami dojde

k pozvolnému přechodu textur. Výsledek je na obrázku 24 s aplikovaným efektem mlhy popsaným v následující kapitole.

Velikost omezená rozsahem shortu může být nedostačující, nejen proto se často používá vykreslení terénu podle height textury ve více částech. Vykreslením více částí jednoho modelu se řeší nejen omezení velikosti pro indexy, ale také použití různých materiálů u modelů. Další výhodou pro krajinu je možnost rozdělení do menších částí a jejich vykreslení pouze v případě potřeby. Tedy použití metody (Frustum Culling). Na jednotlivé části je možné aplikovat i level of detail (LOD) a tím vykreslit části vzdálenější od pozorovatele v nižším rozlišení.

Nevýhodou výškové mapy je nemožnost definovat dutiny a jeskyně. Takové objekty se společně s dalšími modely ostrých skal a kamenů vkládají do krajiny až dodatečně.

4.8 Použití mlhy pro zdůraznění hloubky scény

Mlha je jednoduchý a velmi často používaný efekt, kdy tělesa přebírají se vzdáleností barvu této mlhy. V běžném světě jsou vzdálená tělesa skryta za jinými tělesy, za horizontem nebo v mlze. Vzhledem k omezenému výkonu zařízení nelze vykreslovat veškeré objekty obsáhlé scény. Mlha je jednou z možností jak vytvořit pozvolné ukončení prostoru. Mnoho objektů naprosto skrytých mlhou nemusí být vůbec vykresleno a mohou se v případě potřeby z mlhy postupně vynořovat. Dalším efektem je zdůraznění vzdálenosti objektů.

Existují tři základní způsoby výpočtu mlhy [19]. Prvním nejjednodušším způsobem je použití lineární interpolace.

```
fogFactor = 1.0-clamp((fogEnd-fogCoord)/(fogEnd-fogStart),0.0,
            1.0);
```

Fog faktor je číslo, které udává úroveň mlhy mezi zadanými fog start a fog end. Pomocí funkce clamp je zajištěno, že hodnoty mimo rozsah budou ořezány. Fog coord označuje hloubku aktuálního pixelu, kterou získáme následovně.

```
fogCoord = abs(eyePos.z/eyePos.w);
```

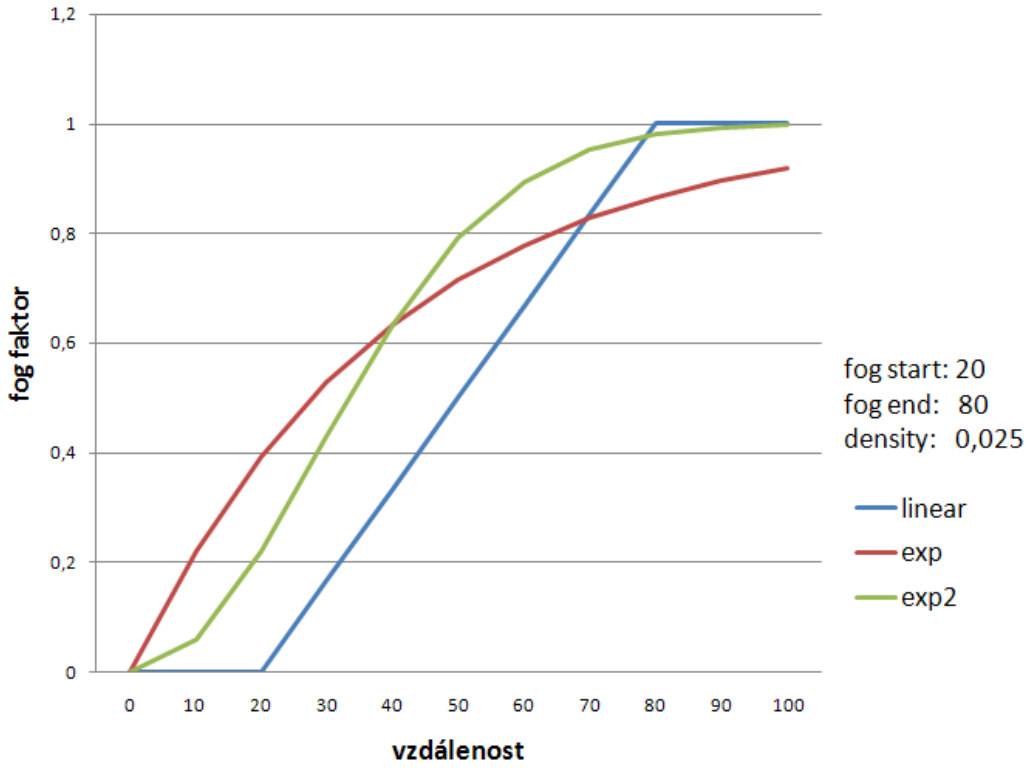
Takto je možné získat vzdálenost aktuálního pixelu od pozorovatele, protože efekt mlhy je na této informaci závislý. Pozice eyePos (pozice vertexu v eye space) je získána vynásobením pozice vrcholu a model view matice. Dalším způsobem výpočtu je použití exponenciální funkce.

```
fogFactor = 1.0-clamp(exp(-fogDensity*fogCoord),0.0,1.0);
```

Novým parametrem je zde fog density udávající rychlost s jakou efekt nastupuje. Třetí možností je přidání umocnění na druhou, čímž se funkce znovu pozmění.

```
fogFactor = 1.0-clamp(exp(-pow(fogDensity*fogCoord,2.0)),0.0,
            1.0);
```

Na obrázku 23 je znázornění fog faktoru těchto metod pomocí grafu, ze kterého vyplývá způsob růstu efektu mlhy.

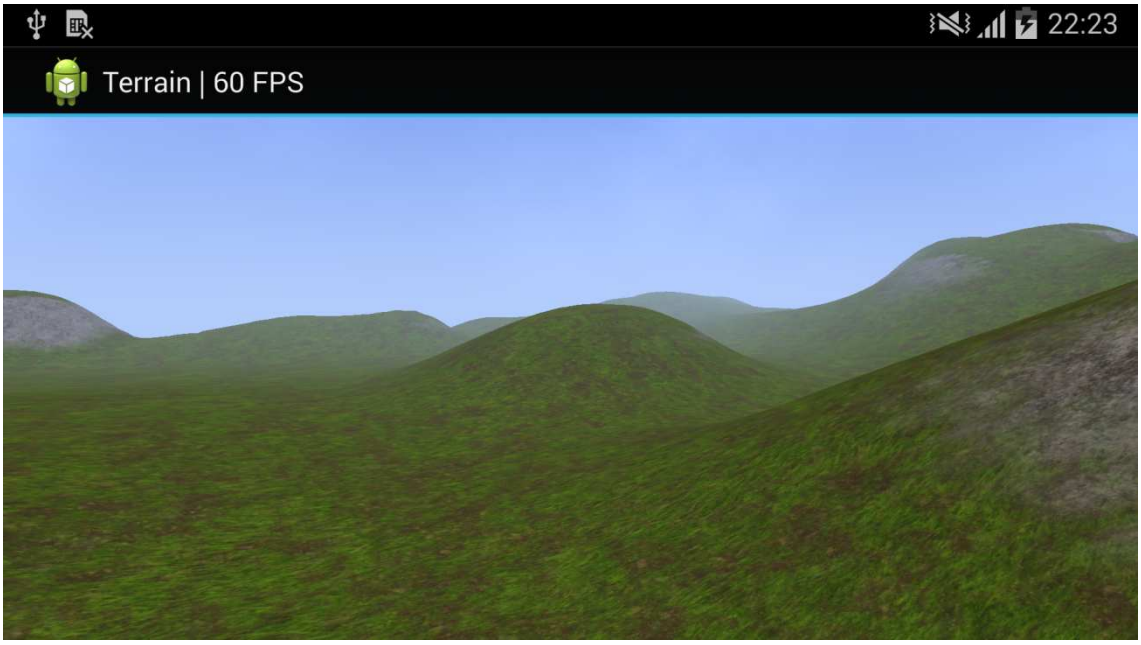


Obrázek 23: Graf metod pro výpočet mlhy

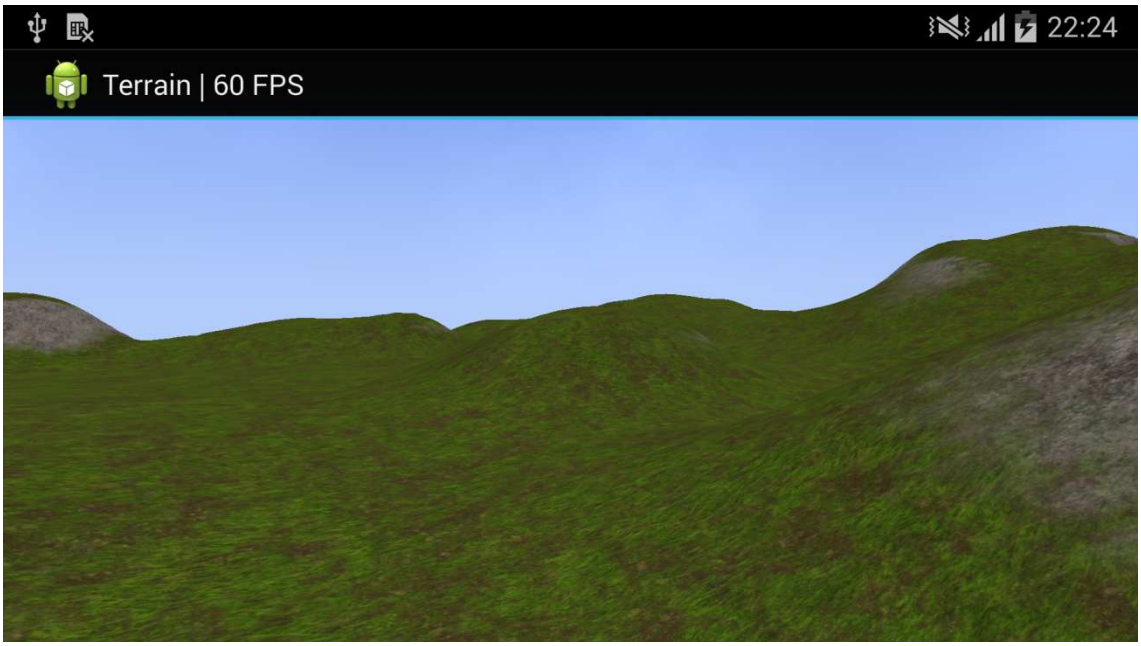
Posledním krokem je použití fog faktoru pro smíchání barvy mlhy a barvy objektu. Barva mlhy by měla splývat s pozadím (skyboxem) pro čistý efekt.

```
gl_FragColor=mix(finalColor, fogColor, fogFaktor);
```

Obrázek 24 ukazuje jaký efekt má mlha ve scéně. Pro porovnání je na obrázku 25 stejná krajina bez vypočtené mlhy. Výpočet mlhy je možné dále upravovat a dosáhnout tak například mlhy obarvené zdrojem světla z jednoho směru nebo mlhu závislou na výšce, ve které se pixel nachází. Efekt mlhy valící se po povrchu je vytvářen již odlišným způsobem. Jsou použity poloprůhledné jednoduché tvary, které se neustále natáčejí směrem k pozorovateli (billboarding). Tímto způsobem je možné vytvářet efekty založené na částicových systémech (oheň, déšť, sníh, vegetace, kouř, exploze atd.).



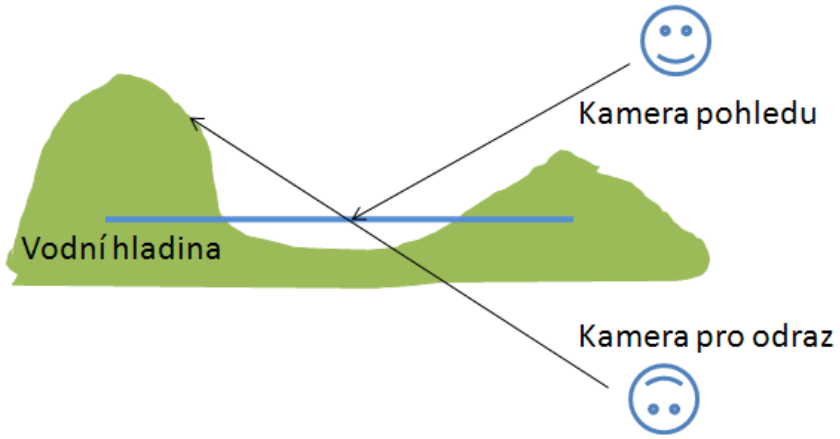
Obrázek 24: Zobrazení krajiny, oblohy a mlhy metodou exp2



Obrázek 25: Krajina bez mlhy

4.9 Vodní hladina

Důležitou součástí otevřené krajiny jsou plochy pokryté vodou. Ty se vyznačují určitou mírou odrazu okolního prostředí, průhlednosti, barvy a vlněním povrchu. Tento pohyb vodní hladiny zároveň upravuje směr odraženého paprsku a vytváří tak vlnění celého odraženého obrazu. Metody pro takové efekty lze najít v dynamice kapalin. Často se jedná o fyzikální simulace za použití částicových systémů pro popis chování kapalin a plynů. Ovšem pro reálné použití je nutné dbát na výkon cílového zařízení a proto je vhodnější použít pouze přibližná řešení. Hlavní myšlenkou za vodním povrchem je použití pouze čtyř vrcholů tvořících základní plochu, normal mappingu pro vytvoření vln a textury nesoucí odraz okolního prostředí [20]. Z toho vyplývá, že je potřeba speciální průchod, který vytvoří texturu s odrazem. Při tomto vykreslování je nutné umístit kameru na pozici, ze které je viditelný potřebný odraz, viz obrázek 26.



Obrázek 26: Návrh realizace odrazu pro vodní hladinu

Tuto druhou pozici kamery lze získat následovně:

```
//pozice kamery pro odraz
camPosition.z = 2.0f * waterHeight - camPosition.z;

//nový směr pohledu a up vektor
camTarget.z = 2.0f * waterHeight - camTarget.z;
vec3 forwardVector = camTarget - camPosition;
vec3 sideVector = camera.GetRigthVector();
vec3 reflectionCamUp = sideVector.cross(forwardVector);
createLookAt(camPosition, camTarget, reflectionCamUp);
```

První řádek zajistí, že pozice kamery pro odraz bude ve stejné vzdálenosti od vodní hladiny (water height) jako kamera pohledu. Dále je nutné zařídit, že samotný směr pohledu (místo kam se uživatel dívá) bude zrcadlen, tedy obrácen na ose z. Poslední potřebnou součástí pro novou pozici kamery je up vektor, který se získá z předchozích údajů. Bez nového up vektoru nebude výsledný pohled zrcadlovým odrazem a nebude otočen o 180 stupňů. Otočit je však možné i samotnou výslednou texturu obrácením druhé souřadnice v. Posledním problémem jsou objekty umístěné pod vodní hladinou. Tyto objekty se nevykreslují, pokud nezasahují nad vodní hladinu. Objekty částečně nad hladinou je nutné oříznout a odstranit ponořenou část. Toho je možné docílit použitím funkce discardve fragment shaderu pro hodnoty pod „z" souřadnicí hladiny při vykreslování těchto objektů. Lepším řešením je použití ořezávacích ploch (clip plane), které ale OpenGL ES nepodporuje. Po získání textury s odrazem lze vykreslit plochu představující vodní hladinu.

```
//fragment shader
vec2 texCoordNormal0 = position.xz * tile;
texCoordNormal0 += time ;
vec2 texCoordNormal1 = position.xz * tile;
texCoordNormal1.s -= time;
texCoordNormal1.t += time;

//výpočet souřadnic do textury s odrazem podle normal textury
vec3 normal0 = texture2D(waterWave,texCoordNormal0).rgb*2.0-
1.0;
vec3 normal1 = texture2D(waterWave,texCoordNormal1).rgb*2.0-
1.0;
vec3 normal =normalize(normal0 + normal1);

//získání souřadnic do textury s odrazem
vec2 texCoordReflection = vec2(gl_FragCoord.x /
scrWidth,gl_FragCoord.y / scrHeight);

//získání konečné barvy pomocí upravených souřadnic
vec2 texCoordFinal = texCoordReflection.xy + noise * normal.xy;
vec4 fcolor = texture2D(waterReflect, texCoordFinal);
```

```
//Smíchání s barvou vody
gl_FragColor = (0.55*waterColor + (1.0-0.55)*fcolor);
```

Jako první jsou získány souřadnice do textury, která slouží pro realizaci vln. Ty jsou pouze roztažením nebo stlačením určitých míst textury s odrazem. Parametr tile je použit pro nastavení velikosti vln. Souřadnice k druhé textuře získané v předchozím průchodu se získají přepočtem rozsahu obrazovky na 0 až 1. Jako poslední lze získat upravený (zvlněný) obraz a dále s ním pracovat. Jednoduchým zlepšením je přidání barvy vody. Realizace dalších efektů již může přinést snížení výkonu. Stejně jako je zvlněn odraz je možné vytvořit zvlněnou verzi objektů pod vodní hladinou (refraction). Pak je ale nutné přidat další průchod pro vykreslení textury s těmito objekty. Jednodušším vhodným řešením je refrakci vynechat a použít průhlednost, která může být případně nastavena v závislosti na úhlu pohledu. Další úpravou souřadnic do normal textury jde také docílit nasměrování vln a tím například efektu řeky tekoucí korytem. Konečná scéna je zobrazena na obrázku číslo 27.

Obrázek 27: Scéna s vodní hladinou

4.10 Billboarding pro vykreslení vegetace

Po předchozí kapitole již zavlaženou krajinu mohou začít okupovat další objekty a modely. Kromě konkrétního obsahu je dalším důležitým prvkem vegetace. Většina stromů a rostlin se vyznačuje značně složitou strukturou a není reálné modelovat každý list zvlášť. Pro filmy a hry se často používá knihovna SpeedTree, která zajišťuje ztvárnění, modelování a vykreslování různých druhů stromů na vysoké úrovni. Knihovna bohužel není dostupná pro mobilní zařízení a na jiné platformy je pouze v placené verzi.

Je tedy nutné najít jiný jednoduchý způsob jak vykreslit stromy. Jednou možností je použití statického modelu. Problém však může nastat, pokud by měl být model animován. Kromě nároků na výkon zařízení může být samotné vytvoření animace náročné. Dalším prvkem jsou listy a větve, které se vykreslují jako 2D plochy pokryté texturou s alfa kanálem (zajištění průhlednosti textury mimo samotné listy). Aby strom vypadal dobře ze všech stran, je potřebné velké množství takových ploch zkřížených přes sebe. Několik překřížených ploch je vhodné použít pro drobnou vegetaci, jako je tráva, ale u větších objektů není efekt dostačující. Jedním z řešení může být využití částicových systémů. Ty se skládají z mnoha 2D dílků, které se neustále natáčejí směrem k pozorovateli a mohou mít definovaný další pohyb. Jako inspirace využití takového systému pro detailnější model stromu poslouží starší implementace SpeedTree knihovny ze hry The Elder Scrolls IV: Oblivion viditelná na obrázku 28.

Obrázek 28: Referenční obrázek stromu ze hry TES IV: Oblivion

Model stromu se skládá ze dvou částí. První je kmen a větve, druhá část je pak složena z množství 2D ploch s texturou listů a drobných větviček natáčejících se směrem k pozorovateli. Metoda pro tyto rotující plochy se nazývá billboarding a při použití pro vykreslení stromů snižuje množství potřebných ploch s listy a umožňuje jednoduchou animaci.

Jednou z možností jak získat čtverce natočené vždy ke kameře je použití GL_POINTS. U této jednoduché metody [15][10] stačí definovat souřadnice středů a vykreslit s vhodnou texturou. Velikost čtverce je nutné definovat pomocí vertex shaderu voláním gl_PointSize a pomocí speciálního výpočtu lze určit i perspektivní velikost. Zadaná point size je vždy konečná velikost v pixelech a je nutné ji upravovat podobně jako v případě úbytku osvětlení.

```
//vzdálenost bodu od pozorovatele
float d = length(viewSpacePosition);
//úbytek se vzdáleností
gl_PointSize = 100.0 / sqrt(attenuation.x + (attenuation.y * d)
               + (attenuation.z * d * d ));
```

V mnoha případech jsou nutné i další výpočty pro zajištění správné velikosti vůči scéně. Použití GL_POINTS má ale více problémových vlastností. Na některých

zařízeních je maximální velikost gl_PointSize příliš malá. To znamená, že při přibližování kamery přestane v určitou chvíli narůstat velikost plochy a to může značně narušit konečný dojem. Dalším problémem je automatické odstranění celého čtverce ze scény (point clipping), pokud jeho střed překročí okraj pohledového objemu. I přes nastavenou velikost je čtverec stále považován za jednoduchý bod. Kvůli těmto problémům jsou částicové systémy postavené na GL_POINTS vhodné pouze pro efekty využívající drobné částečky (sníh, déšť).

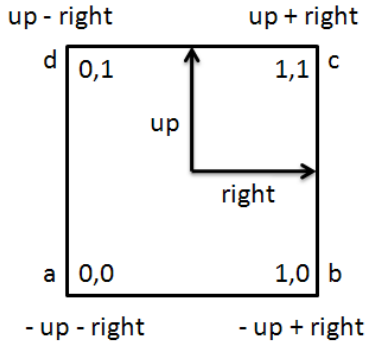
Pro částice, které zabírají větší plochu obrazovky (kouř, oheň, listy), je možné vytvořit billboarding ručně. Billboarding [21] slouží obecně pro natočení jakéhokoliv objektu směrem ke kameře. Pro realizaci větví s listy jako na obrázku 28 je vhodné použít 2D plochu s odpovídající texturou. Nejdříve je nutné určit tvar požadované plochy. Nejjednodušším tvarem s nejmenším počtem vrcholů je trojúhelník. Běžná textura pro částice má většinou definovanou průhlednost pro vytvoření finálního tvaru (sněhová vločka, list). Při mapování na trojúhelník se sice ušetří několik vrcholů, ale fragment shaderem zbytečně zpracovávaná plocha bude značná. Proto je vhodné využít tvar částice, který blíže odpovídá požadovanému konečnému tvaru při zobrazování. Pro nejlepší výsledek se hledá správný poměr mezi počtem vrcholů a velikostí plochy. Pro jednoduchost je v aktuální implementaci zvolen tvar čtverce.

Druhým krokem je vykreslit čtverec tak, aby byl vždy natočen ke kameře. Toho lze dosáhnout získáním right a up vektorů z inverzní 3x3 horní levé submatice z view matice. Tyto vektory definují směr nahoru a doprava vůči kameře.

$$\text{viewMatrix} = \begin{pmatrix} \boldsymbol{A_{11}} & \boldsymbol{A_{21}} & \boldsymbol{A_{31}} & A_{41} \\ \boldsymbol{A_{12}} & \boldsymbol{A_{22}} & \boldsymbol{A_{32}} & A_{42} \\ \boldsymbol{A_{13}} & \boldsymbol{A_{23}} & \boldsymbol{A_{33}} & A_{43} \\ A_{14} & A_{24} & A_{34} & A_{44} \end{pmatrix}$$

Zvýrazněná submatice definuje rotaci a je ortogonální. Pro takovou ortogonální matici platí, že její inverzní matice je rovna transponované. Potřebné vektory tedy lze extrahovat přímo jako první řádek submatice pro right vektor a druhý pro up vektor. Poslední řádek je front vektor, který může být v mnoha případech také užitečný.

Pomocí těchto vektorů lze určit pozice vrcholů každého billboardu. Jako základ slouží čtyři vrcholy a jejich rozmístění probíhá na základě texturovacích souřadnic. Základní pozice každého vrcholu je stejná a může být zvolena jako střed budoucí plochy. Veškeré potřebné informace (vrcholy, uv souřadnice do textury, vektory) jsou znázorněny na obrázku 29.



Obrázek 29: Znázornění dat pro billboarding

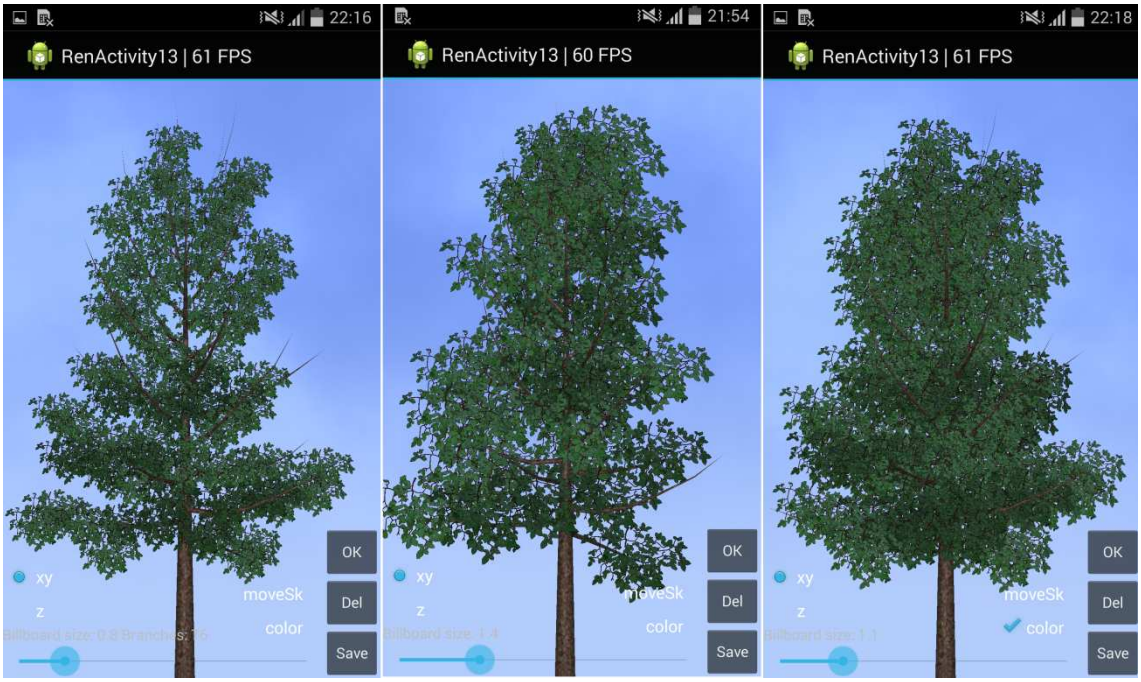
Pomocí vektorů a středu plochy lze vypočítat jednotlivé vrcholy.

```
a = center - (right + up) * size;
b = center + (right - up) * size;
c = center + (right + up) * size;
d = center - (right - up) * size;
```

Parametrem size se ovlivňuje velikost posunu a tím i celého billboardu a lze jím vektory vynásobit již na procesoru. Ve vertex shaderu je možné výpočet upravit pomocí texturovacích souřadnic následovně.

```
v = posCenter+ (tex.x * 2.0 - 1.0) * right + (1.0 - tex.y *
2.0) * up;
```

V některých případech je třeba otáčet plochou pouze kolem up vektoru. Může se jednat například o vykreslení LOD (level of detail) obrazů stromů v dálce. Samotný up vektor je pak neměnný a shodný s osou y. Po stanovení up vektoru je nutné zajistit, aby na něj byl right vektor kolmý pomocí Gram-Schmidt ortogonalizace. Obrázek 30 ukazuje kmen a různě velké billboard větve v pomocné aplikaci pro vytváření a ukládání stromu.



Obrázek 30: Realizace modelu stromu

Aplikace používá tmavší barvu pro některé větve a definuje pohyb kolem front vektoru kamery pro simulaci větru. Tento pohyb je aplikován jako rotační matice na původní view matici, ze které jsou následně extrahovány potřebné vektory. Potřebný počet billboardů se pro efekt podobný referenčnímu obrázku 28 pohybuje kolem sta. Předloha oproti implementaci používá dvě textury, protože použití jedné vede k rozeznatelné struktuře na okrajích modelu. Další vylepšení může být zaměřeno na výpočet osvětlení případně prosvítání světla skrz listy (subsurface scattering).

4.11 Postprocessing

Ve 2D a 3D grafice je vytvářeno mnoho efektů, které ke své realizaci potřebují informace o okolí. Při běžném vykreslování ale takové informace nejsou dostupné. Například ve fragment shaderu nelze přistupovat k pixelům v okolí momentálně zpracovávaného. Řešením je použití víceprůchodových algoritmů a nejdříve scénu vykreslit do textury [15]. Tato textura je namapována na čtverec, který perfektně vyplňuje celý obraz, a libovolně upravena. Způsob využití víceprůchodových algoritmů byl představen v kapitole 4.5. Potřebný čtverec je definován přímo v normalizovaných souřadnicích. Ve 3D vznikají normalizované souřadnice vydělením pozice vrcholu čtvrtou souřadnicí w (rozsah každé souřadnice je pak od -1 do 1). Souřadnice x a y jsou následně mapovány na 2D obrazovku. Potřebné vrcholy je tedy možné definovat pouze ve 2D [22]:

```
-1, -1,
1, -1,
-1,  1,
1,  1
```

Vertex shader tohoto postprocessing průchodu nepotřebuje žádné matice. Pouze definuje pozice vrcholů tak, že přímo odpovídají velikosti obrazovky a dopočítává souřadnice do textury:

```
texCoord = (inPosition + 1.0) / 2.0;
gl_Position = vec4(inPosition,0.0, 1.0);
```

Konkrétní efekty vznikají ve fragment shaderu, který má nyní přístup k informacím kdekoliv v textuře. Obraz je možné například deformovat úpravou souřadnic do textury jako v kapitole 4.9 Vodní hladina. Ze základních efektů [23] lze jednoduše aplikovat například následující:

```
//inverze barev
gl_FragColor = vec4(1.0,1.0,1.0,1.0) - texture2D(postProcTex,
texCoord);

//odstíny šedi
vec4 color = texture2D(postProcTex, texCoord);
```

```
float value = (color.r + color.g + color.b) / 3.0;
gl_FragColor = vec4(value,value,value,1.0);
//odstíny šedi odpovídající citlivosti lidského oka
vec4 color = texture(screenTexture, TexCoords);
floatvalue = 0.2126*color.r+0.7152*color.g+0.0722 *color.b;
gl_FragColor = vec4(value,value,value,1.0);

//jednoduchý blur efekt při aktivovaném mipmappingu
gl_FragColor = texture2D(postProcTex, texCoord, 3.0);
```

Kromě těchto základních efektů je možné aplikovat různá rozostření nebo zvýraznění hran. Tyto efekty jsou založeny na konvolučních maticích, které definují, co se má stát s pixely v určité oblasti. Pixely jsou násobeny určitým číslem a jejich hodnoty sečteny. Součet hodnot jádra je jedna, jinak je výsledná barva tmavší nebo světlejší.

```
//gaussian kernel blur
//offset definuje vzdálenost vybíraných pixelů od aktuálního
float offset = 1.0 / 350.0;
//konvoluční matice
float kernel[9] = float[9](
    1.0 / 16.0, 2.0 / 16.0, 1.0 / 16.0,
    2.0 / 16.0, 4.0 / 16.0, 2.0 / 16.0,
    1.0 / 16.0, 2.0 / 16.0, 1.0 / 16.0
);
//aktuální pixel (0,0) a 8 sousedních
vec2 offsets[9] = vec2[9](
    vec2(-offset, offset),
    vec2(0.0f,    offset),
    vec2(offset,  offset),
    vec2(-offset, 0.0f),
    vec2(0.0f,    0.0f)
    vec2(offset,  0.0f),
    vec2(-offset, -offset),
    vec2(0.0f,    -offset),
    vec2(offset,  -offset)
);
vec3 color;
for(int i = 0; i < 9; i++)
```

```
{
//ve forcyklu je vybrán pixel a vynásoben odpovídající hodnotou
    color += vec3(texture2D(postProcTex, texCoord.st +
             offsets[i])).xyz * kernel[i];
}
gl_FragColor = vec4(color,1.0);
```

Konvoluční matice bývají většinou ve formátu 3x3. V některých případech je to však málo. Například aktuálně aplikované rozostření je pouze mírné a v některých částech obrazu nedostatečné. Řešením je zvětšit konvoluční matici. Množství přístupů k textuře ovšem roste kvadraticky $O(N^2)$. Na mobilních zařízeních je vhodné udržovat čtení textury v minimální míře, protože výrazně spotřebovává výkon. Konečným řešením je pak přidání jednoho průchodu a rozdělení konvoluční matice na dva vektory. Každý vektor slouží pro rozmazání v jednom směru a každý průchod tak má složitost pouze O(N). Tento víceprůchodový blur filter se používá pro výraznější rozmazání a pro efekty jako je depth of field (rozmazání podle hloubky scény) nebo bloom (dodatečná záře kolem objektů). Použití uvedených jednoduchých filtrů je zobrazeno na obrázku 31.

Obrázek 31: Ukázka postprocessing efektů

4.11.1 Volumetric light scattering

Jedním ze složitějších a zajímavějších efektů tvořených postprocessingem je volumetric light scattering (god rays). Pokud je v atmosféře dostatek částic rozptylujících světlo (vodní pára, prach, kouř), jsou viditelné samotné paprsky světla. Tento efekt je obzvlášť znatelný, pokud světlu v cestě stojí jiné objekty [24].

Efekt je založen na akumulaci intenzity barvy od aktuálního pixelu směrem ke zdroji světla. K základní barvě pixelu je přičtena hodnota podle následujícího vzorce:

$$\mathrm{L} = explosure * \sum_{i=0}^{n}\left(decay^{i} * weight * sample[i]\right)$$

Každý pixel je pak součtem intenzit z předpřipravené textury (sample[i]), které jsou ovlivněny parametrem decay, který snižuje sílu paprsku směrem od zdroje světla, a parametrem weight ovlivňujícím intenzitu každého vzorku. Parametr explosure pak určuje celkovou sílu paprsku.

Potřebné vzorky jsou uloženy v textuře, která vzniká ve speciálním průchodu. Pro správné určení intenzity paprsku je nutné vykreslit zdroj světla a následně veškeré objekty mezi tímto světlem a pozorovatelem bez textur a osvětlení. Druhý průchod obsahuje běžné vykreslení scény a v posledním dojde k samotnému výpočtu paprsků ve sceen space.

```
//fragment shader volumetric light scattering (poslední
//průchod)
float density = 0.5;//délka paprsku
float exposure = 0.3; //celková světlost
float illuminationDecay = 0.25;//počátečni světlost
float decay = 0.94;//úbytek světlosti se vzdáleností

//výpočet kroku pro následování paprsku
vec2 deltaTextCoord = vec2( texCoord.st - lightScreenPos.st );
vec2 texCoo = texCoord.st;
deltaTextCoord *= 1.0 /  float(NUM_SAMPLES) * density;

//vzorky intenzity
vec4 intenColor = texture2D(postProcDepTex, texCoord);
```

```
//barva pixelu ve scéně
vec4 color = texture2D(sceneTex, texCoord);
vec4 sample;
     for(int i=0; i < NUM_SAMPLES ; i++)
     {
           //sčítání intenzity vzorků od pixelu ke světlu
           texCoo -= deltaTextCoord;
           sample = texture2D(postProcDepTex, texCoo);
           sample *= illuminationDecay;
           intenColor += sample;
           illuminationDecay *= decay;
     }
gl_FragColor = color + (intenColor *exposure);
```

S rostoucí vzdáleností od zdroje světla získá potřebnou intenzitu, určenou tvarem a polohou zdroje, stále méně vzorků. Velkým problémem je počet potřebných vzorků a tím množství přístupů do textury. Pro kvalitní dlouhé a nepřerušované paprsky je vhodné použít kolem 200 vzorků. Takové množství nepřipadá na mobilním zařízení v úvahu. Jako použitelné se ukázalo 20 až 50 vzorků. Bohužel i toto množství razantně snižuje rychlost vykreslování. Výsledný efekt pro 30 vzorků je na obrázku 32.

Obrázek 32: Implementace volumetric light scattering pro 30 vzorků

Paprsky jsou bohužel závislé na tom, že zdroj světla je umístěn v aktuálním obraze. Tento problém lze vyřešit použitím jiné metody založené na znovupoužití shadow map. Počet přístupů do textury jde snížit a použít blur efekty pro smazání viditelných kroků. Pro aktuální zařízení však představuje i několik málo čtení textury problém. Dalším jednoduchým, ale nedokonalým způsobem realizace světelných paprsků, například z oken v budovách, je vykreslení pomocí poloprůhledného modelu.

4.12 Cloth simulation

Systém částic popsaný v předchozí kapitole byl z hlediska jejich chování primitivní. Každá částice byla zobrazena jako billboard se základními vlastnostmi, jako je pozice, textura a případně rotace. Prací s těmito atributy lze vytvořit například kouř, oheň nebo vodní páry. Pokud se však systém rozšíří o další vlastnosti, je možné simulovat zcela nové prvky reálného světa. Velmi zajímavým obsahem scény založeným na částicových systémech může být pohybující se látka.

Takovou látku si lze představit jako 2D grid ovlivňovaný různými silami. Každý z vrcholů uspořádaných v síti je pak samostatnou částicí, na kterou může působit určitá síla (kolize, gravitace, vítr) [25]. Jejich vzájemné vztahy jsou vyjádřeny pomocí spojů (constraints, springs), které je drží v původním tvaru.

Prvním krokem je aplikace síly a její převod na pozici částice. Na začátku je přítomna síla působící v určitém směru. Pomocí Newtonova druhého zákona (zákon síly) a hmotnosti lze vypočítat zrychlení tělesa (částice).

```
void addForce(Vec3 f)
{
     acceleration += f/mass;
}
```

Ze zrychlení lze získat pozici pomocí numerické integrace. Výpočet nové pozice ze staré za použití zrychlení se nazývá verlet integration.

```
Vec3 temp = pos;
pos = pos + (pos-old_pos) + acceleration*TIME_STEPSIZE
old_pos = temp;
acceleration=Vec3(0,0,0);
```

Po přidání síly se bude částice neustále pohybovat ve zvoleném směru. Pomocí parametru TIME_STEPSIZE je určena velikost kroku a lze ji spojit s rychlostí vykreslování pro zajištění stále stejné rychlosti pohybu na různě výkonných zařízeních. Rychlost pohybu a uražená vzdálenost částice ovšem v běžném prostředí klesá kvůli tření vzduchu. Toto tlumení (damping) je aplikováno násobením číslem od 0 do 1 následovně:

```
pos = pos + (pos-old_pos)*(1.0-DAMPING) +
      acceleration*TIME_STEPSIZE
```

Nyní je nutné vytvořit podmínky pro spojení částic takovým způsobem, aby měly snahu zůstat ve svém původním tvaru (gridu). Každé dvě částice mají mezi sebou základní vzdálenost, kdy je látka v klidovém stavu (rest distance). Při verlet integration dochází ke změně této vzdálenosti a je třeba nějakým způsobem zajistit snahu vrátit částice do původního stavu. Tento proces se nazývá constraint satisfaction.

```
public void satisfyConstraint(){
// vektor z částice 1 (p1)dočástice 2 (p2)
Vec3D p1_to_p2 = p2.getPos().minus(p1.getPos());

// vzdálenost mezi p1 a p2
float current_distance = (float) p1_to_p2.length();

// vektor, kterým lze posunout p1 na základní vzdálenost od p2
Vec3D correctionVector = p1_to_p2.mul((1 -
                         rest_distance/current_distance));

// každá částice se může posunout o polovinu tohoto vektoru
Vec3D correctionVectorHalf = correctionVector.mul(0.5);

// posunutí p1 k p2
p1.offsetPos(correctionVectorHalf);

// posunutí p2 k p1
P2.offsetPos(correctionVectorHalf.mul(-1));
     }
```

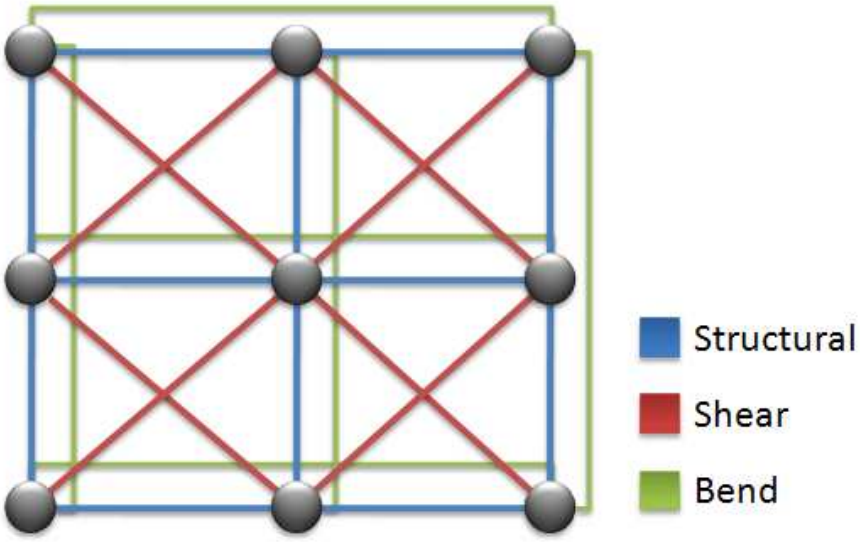
Faktor, kterým se násobí vektor od p1 do p2 a slouží k posunu p1 směrem zpět k p2 na základní vzdálenost, je:

```
(1 – rest_distance/current distance)*0,5
```

Tento faktor vyjadřuje, o kolik procent je nutné změnit vzdálenost mezi p1 a p2, aby se částice vrátili do své původní polohy. Aby se celý objekt udržel ve svém

tvaru, jsou vytvořeny tři druhy spojení (constraints, springs) mezi částicemi. Ilustrace těchto vláken je na obrázku 33.

První jsou strukturální vlákna, která udržují látku v základním tvaru. Jsou tvořena spojením sousedních částic (vrcholů). Bohužel tato spojení nestačí k realistickému chování látky. Je vhodné proto přidat „shear" vlákna, která zajišťují lepší držení tvaru tkaniny. Poslední typ vláken (bend) brání částicím vzdáleným ob jednoho souseda v přiblížení a úplnému přeložení látky.



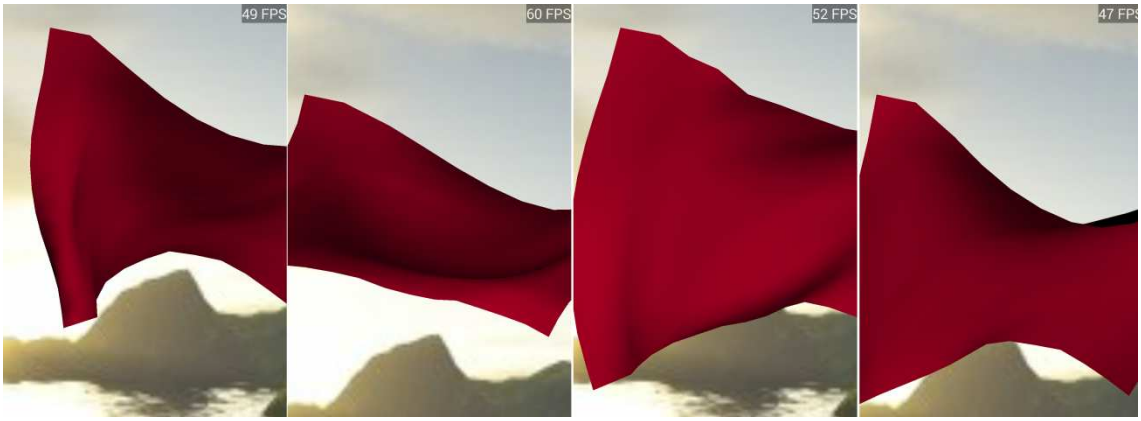
Obrázek 33: Druhy spojení pro cloth simulation

Nevýhodou těchto spojení je, že každá částice je propojena více vlákny. To znamená, že každé spojení se neustále snaží upravit umístění částice a pohyb tak není plynulý. Používaným řešením je spustit constraint satisfaction vícekrát pro jedno vykreslení. Vzhledem k operacím s vektory roste zatížení procesoru a simulace detailních látek tak může být problematická. Pro další využití je vhodné uvažovat nad některými optimalizacemi. Jednoduchým vylepšením je odstranění „bend" vláken pro zavěšenou látku, která se nedostává příliš často do situace, kde by tyto vlákna měli významný vliv. Dále využití jazyka C++, který je označován za rychlejší a případné snížení počtu částic. Efektních výsledků dosahuje také pouze 2D implementace tkaniny.

Posledním krokem je vykreslení částic (vrcholů) s vypočítanými normálovými souřadnicemi. Souřadnice vrcholů a normály se neustále mění. Pro

neměnná data je vhodné poslat je grafickému čipu a následně vykreslovat (GL_STATIC_DRAW). V tomto případě je ale vhodné data posílat opakovaně pro každé vykreslení a využít tzv. GL_DYNAMIC_DRAW. Jedinou neměnnou částí dat jsou indexy a souřadnice do textury.

Na částice lze aplikovat různé síly v různých směrech. Výsledek implementace 3D cloth simulation s gravitací a větrem je na obrázku 34.



Obrázek 34: Zobrazení tkaniny ve větru

V případě, že cloth simulation stojí příliš výpočetní síly, je možné použít zjednodušený pohyb pomocí funkce ve vertex shaderu v závislosti na čase. Tento způsob přináší nedokonalý, ale levný výsledek. Využití lze nalézt také v simulaci vln na vodní hladině.

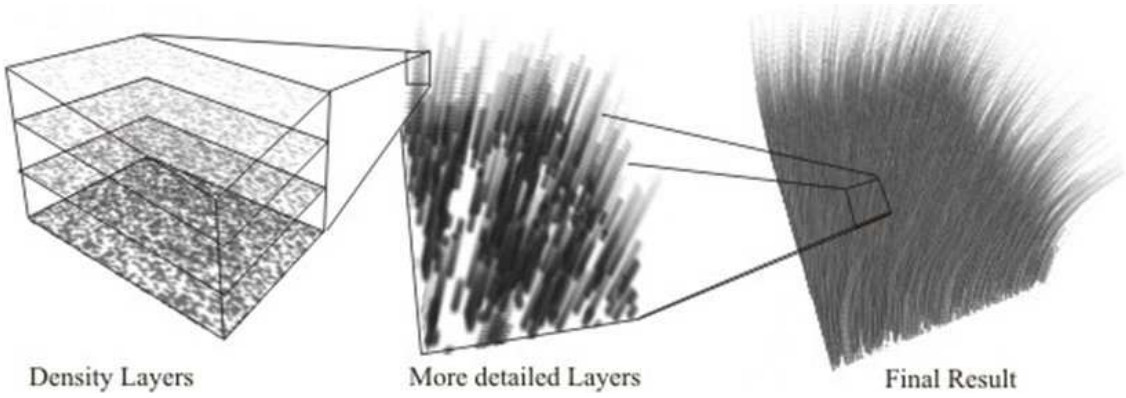
4.13 Fur simulation

Objekty reálného světa mohou mít velmi odlišnou strukturu. Mnoho povrchů lze simulovat pomocí osvětlení a normal mappingu. Problém však nastane u povrchů, které se skládají z velkého množství drobných částí. Příkladem takového povrchu je srst. Modelovat každý chlup samostatně, nebo použít 2D plochy s trsy chlupů není efektivní. Srst ovšem pokrývá daný objekt podobně jako textura. Pokud se tedy zobrazí vhodná textura ve vrstvách, je možné simulovat různé vláknité povrchy, jako je srst nebo tráva. Tato metoda se nazývá shell texturing [26].

Základem je textura, která obsahuje body s alfa kanálem na hodnotě 1, zatímco okolí má hodnotu 0. Na každé vrstvě pak budou zobrazeny pouze tyto body. Pro vytvoření vrstvy se vykreslí daný model, který ve vertex shaderu expanduje ve směru normály do vzdálenosti podle pořadí vrstvy.

```
pos = vec3(modelview * vec4(inPosition + (inNormal *
       furLength),1.0)).xyz;
```

Parametr furLength je číslo od nuly do maximální velikosti vláken určené podle počtu vrstev. Jemnějšího a reálnějšího efektu je možné dosáhnout větším množstvím jednotlivých vrstev. Znázornění tohoto postupu je na obrázku 35.



Obrázek 35: Koncept shell texturingu [26]

Jednou z výhod této metody je možnost dalších úprav a vylepšení. Příkladem může být sklon vláken směrem k zemi zapříčiněný gravitační silou.

```
//Definice směru síly
vec3 vGravity = vec4(0.0,-0.2,0.0);
//exponenciální parametr sklonu podle pořadí vrstvy
float k = pow(furLayer,3.0);
//aplikace sklonu na vlákna
pos+= vec3(vGravity*k);
```

Dále je možné měnit výšku jednotlivých vláken, směr síly pro sklon nebo přidat různé efekty stínování. K větší realističnosti může pomoct také použití textur s různou hustotou bodů, pro simulaci řídnoucí srsti směrem od objektu. Na obrázku 36 je implementace shell texturing s deseti vrstvami. Velikost vláken je možné upravit pomocí změny texturovacích souřadnic. Problémem může být nutnost několikanásobného opakování geometrie. Řešení přináší až nejnovější zařízení s Androidem 5.0 a OpenGL 3.1, která mají k dispozici geometry shader. Tento nástroj umožňuje levnější opakování geometrie a tím snadnější implementaci nejen shell texturingu.

Obrázek 36: Implementace shell texturing

4.14 Augmentová realita

Kromě plně grafických aplikací existuje možnost obohatit a řídit vykreslovanou scénu pomocí senzorů daného zařízení. Taková aplikace se skládá ze dvou druhů dat. První jsou reálné informace ze senzorů a kamery. Nejčastější je zobrazení vrstvy s obrazem z kamery pomocí rozhraní android.hardware.Camera. Další vrstvu pak tvoří druhý typ dat, kterými je realita „vylepšena“. Jedná se o libovolné objekty vykreslované nad první vrstvou.

4.14.1 Senzor based augmented reality

Pouhé vykreslení objektů přes obraz ovšem není příliš užitečné. Pro implementaci aplikací například pro zobrazování informací o světě po najetí kamerou na správné místo je nutné řídit transformace scény. Pro správnou změnu polohy je pak nutné získat matice z dat poskytovaných senzory [27]. Použitelné senzory jsou:

- **Accelerometer** slouží pro zachycení zrychlení zařízení,
- **Magnetometer** detekuje směr a sílu magnetického pole,
- **Gyroscope** měří rotaci zařízení,
- **GPS** poskytuje informace o poloze.

Pro základní aplikaci augmentové reality stačí použít accelerometer a magnetometer. Ukázková aktivita implementuje SenzorEventListener, stará se o použití kamery a spojení s rendererem [28]. Proces spojení obrazu z kamery a OpenGL v metodě onCreate je následující:

```
private GLSurfaceViewmGLSurfaceView;
private AugCamera cameraView;
private Renderer ren;

//vytvoření opengl view
mGLSurfaceView = new GLSurfaceView(this);
mGLSurfaceView.setEGLContextClientVersion(2);

//nastavení instance opengl rendereru
```

```
ren = new Renderer(this);

//nastavení průhlednosti pro opengl view
mGLSurfaceView.setEGLConfigChooser( 8, 8, 8, 8, 16, 0 );
mGLSurfaceView.getHolder().setFormat( PixelFormat.TRANSLUCENT);

//přiřazení rendereru k opengl view
mGLSurfaceView.setRenderer(ren);

//vytvoření camera surface view
cameraView = new AugCamera( this );

//nastavení content view na camera view
setContentView(cameraView, new LayoutParams(
    LayoutParams. MATCH_PARENT, LayoutParams.MATCH_PARENT) );
//přidání opengl view
addContentView( mGLSurfaceView, new LayoutParams(
    LayoutParams. MATCH_PARENT, LayoutParams.MATCH_PARENT) );
```

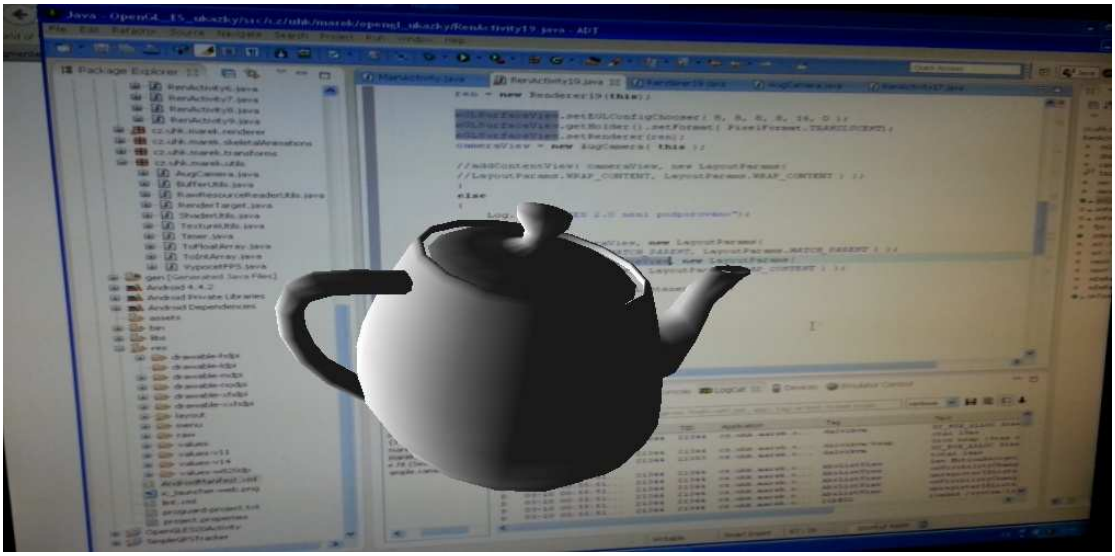
Dalším krokem je využití senzorů implementací SensorEventListener pro získání rotační matice, která bude aplikována na objekty vykreslované rendererem.

```
publicvoid onSensorChanged(SensorEvent evt) {
    int type=evt.sensor.getType();
    //získání a očištění dat z magnetometru
    if (type == Sensor.TYPE_MAGNETIC_FIELD) {
          geomag = lowPass( evt.values, geomag );
    //získání a očištění dat z akcelerometru
    } elseif (type == Sensor.TYPE_ACCELEROMETER) {
          gravity = lowPass( evt.values, gravity );
    }
    //získání rotační matice zařízení
    if ((type==Sensor.TYPE_MAGNETIC_FIELD) ||
    (type==Sensor.TYPE_ACCELEROMETER)) {
        rotationMatrix = newfloat [16];
        SensorManager.getRotationMatrix(rotationMatrix, null,
        gravity, geomag);
        //změna os pro landscape polohu zařízení
```

```
            SensorManager.remapCoordinateSystem(
                  rotationMatrix,
                  SensorManager.AXIS_Y,
                  SensorManager.AXIS_MINUS_X,
                  rotationMatrix );
      }
      //nastavení matice do rendereru
      ren.setRotationMatrix(rotationMatrix);
}
float ALPHA = 0.5f;
//metoda pro očištění dat ze senzorů od šumu
protectedfloat[] lowPass( float[] input, float[] output ) {
      if ( output == null ) return input;

            for ( int i=0; i<input.length; i++ ) {
                  output[i] = (float) ((input[i] * ALPHA) +
                  (output[i] * (1.0 - ALPHA)));
            }
      return output;
}
```

Očištěná data ze senzorů slouží pro získání matice pomocí getRotationMatrix(). Tuto matici lze následně použít jako view matici v OpenGL. Při použití portrait orientace zařízení lze matici použít přímo, protože OpenGL i senzory používají stejný systém souřadnic. Při landscape orientaci je ale systém souřadnic senzorů brán s ohledem na vlastní zařízení a OpenGL souřadnicový systém podle polohy zařízení. Je tedy nutné upravit některé osy v rotační matici pro odpovídající OpenGL reprezentaci pomocí remapCoordinateSystem().

Vykreslovaný objekt nyní působí, jako by se vznášel na určitém místě v prostoru. Na obrázku 37 je model konvice před monitorem. Tuto základní implementaci lze dále rozšířit o data z GPS, přesnější přípravu rotační matice a úpravu chování kamery při změně orientace.

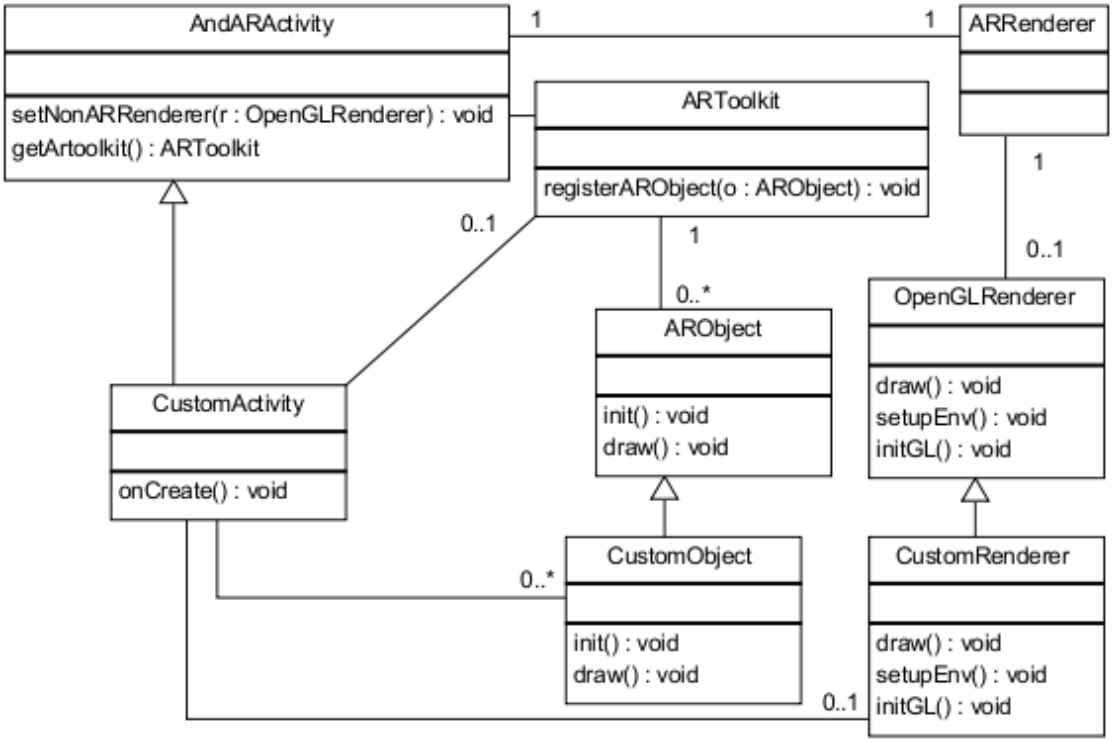


Obrázek 37: Realita augmentovaná konvicí

4.14.2 Marker based augmented reality

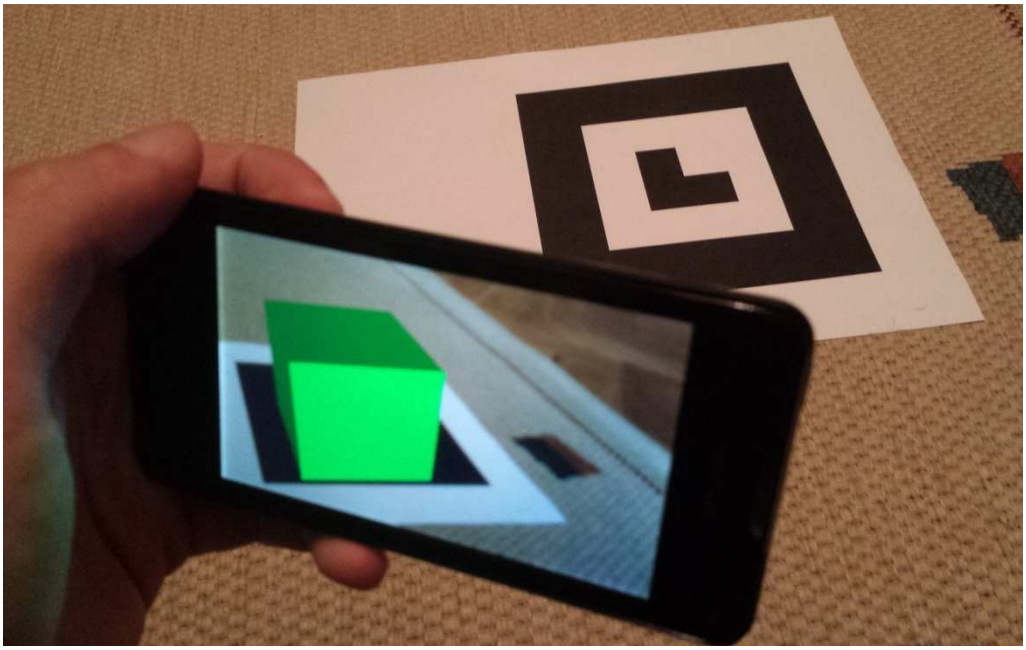
Další možností použití augmentové reality je založeno na vykreslení objektu na předdefinovanou značku (marker) umístěnou v prostoru. Marker je nejčastěji jednoduchý černobílý 2D obraz. Pro získání rotační matice dochází k zpracování obrazu z kamery pomocí metod počítačového vidění, které lze aplikovat s použitím OpenCV knihovny.

Pro vypořádání se s identifikací markeru a získání potřebných matic zajišťujících správné umístění objektu, lze použít řadu různých nástrojů. Jedním z nich je AndAR framework [29], který využívá ARToolkit knihovnu pro identifikaci markeru a získání potřebných matic. Java API AndAR se postará o vykreslení obrazu z kamery jako textury v OpenGL ES a zajistí volání metody, kde je možné definovat vlastní objekty. Základem použití frameworku jsou abstraktní třídy AndARActivity a ARObject. Pomocí rozšíření AndARActivity dochází k získání instance ARToolkit a registraci ARObject, kde dochází k vykreslování. Třída rozšiřující ARObject musí implementovat metodu draw(), která je volána za účelem vykreslení objektu na zadaný marker. Zobrazení obrazu z kamery probíhá automaticky. Na obrázku 38 je ukázán diagram tříd AndAR. Třídy rozšiřující základní abstraktní třídy jsou označeny slovem „Custom“. Použít je možné i renderer bez vztahu k augmentové realitě pomocí CustomRenderer.



Obrázek 38: Diagram tříd pro AndAR aplikace [29]

Na vytvořené aplikace se vztahuje GNU GPL licence a pro případné komerční využití je nutné kontaktovat tvůrce knihovny ARToolkit ARToolworks, Inc. Ukázka použití AndAR je na obrázku 39.



Obrázek 39: Ukázka marker based augmented reality

5 Testování vyvinutého řešení pro různé případy použití

Vybrané aplikované scény byly testovány na hlavní ukazatel výkonu, kterým je rychlost vykreslování. Zvoleny jsou scény, které se vzájemně z hlediska zátěže na zařízení co nejvíce liší. Ukazatel rychlosti renderování je zobrazen v každé scéně pomocí jednotky FPS (frame per second). Cílem testování je posouzení schopností aktuálních zařízení v zobrazování složitějších grafických scén, případně vyvození závěrů nad provedenou implementací.

FPS poskytuje informaci o rychlosti překreslování a relativní minimum pro zachování plynulosti pohybu ve scéně je 25 až 30 FPS. Tato hodnota je ovšem velmi zavádějící pro měření samotného výkonu. Například zhoršení o 10 FPS má jiný význam jedná-li se o zhoršení ze 100 FPS nebo z 20 FPS. Frame per second je přepočítaná hodnota z rychlosti jednoho vykreslení scény. Hodnota 100 FPS tedy znamená, že jedno vykreslení trvá 0,01 s. Tabulka číslo 2 ukazuje rozdíl mezi zhoršením o 10 FPS z různých výchozích FPS. Důsledkem je, že ke změně o 10 FPS je potřebná různá zátěž. U měření výkonu budou z těchto důvodů uváděny také údaje o rychlosti vykreslování v sekundách. Je vhodné uvést, že zařízení s Androidem mají pevně stanovenou obnovovací frekvenci na 60 FPS. To znamená, že vyšších hodnot není možné dosáhnout.

Tabulka 2: Porovnání rychlosti překreslování v jednotkách FPS oproti rychlosti překreslení jednoho obrazu v sekundách

Původní rychlost překreslení [FPS]	*Zhoršení rychlosti překreslení [FPS]*	*Původní rychlost překreslení [s]*	*Rychlost překreslení po zhoršení [s]*	*Rozdíl / zhoršení v rychlosti překreslení [s]*
100	10	0,01	0,0111	0,001
20	10	0,05	0,1	0,05

Testy byly provedeny na zařízeních uvedených v tabulce 3. Kromě základních parametrů je vhodné zahrnout i informace o rozlišení, které může mít nemalý vliv na výsledný výkon v grafických aplikacích, protože zařízení s vyšším rozlišením musejí zpracovat větší množství pixelů. Rozlišení zařízení obsahuje tabulka číslo 4.

Tabulka 3: Použitá testovací zařízení

Zařízení	*SoC*	*CPU*	*GPU*	*RAM*
Samsung Galaxy S4	Qualcomm Snapdragon 600 APQ8064T	Krait 300, 1900 MHz, Cores: 4	Qualcomm Adreno 320, 400 MHz, Cores: 4	2 GB, 600 MHz
Samsung Galaxy S3	Samsung Exynos 4 Quad 4412	ARM Cortex-A9, 1400 MHz,Cores: 4	ARM Mali-400 MP4, 440 MHz,Cores: 4	1 GB, 400 MHz
Huawei Ascend y300	Qualcomm Snapdragon S4 Play MSM8225	ARM Cortex-A5, 1000 MHz, Cores: 2	Qualcomm Adreno 203	512 MB
Zopo zp990	MediaTek MT6592	ARM Cortex-A7, 1700 MHz, Cores: 8	ARM Mali-450 MP4, 700 MHz, Cores: 4	2 GB, 666 MHz

Tabulka 4: Rozlišení obrazovky testovacích zařízení

Zařízení	*Width [pix]*	*Height [pix]*
Samsung Galaxy S4	1920	1080
Samsung Galaxy S3	1280	720
Huawei Ascend y300	800	480
Zopo zp990	1920	1080

5.1 Testování HW nároků

První testovanou scénou je vodní hladina z kapitoly 4.9. Scéna obsahuje krajinu, dva modely stromu, mlhu, skybox, částečně průhlednou vodní hladinu s odrazem okolí a je zachycena na obrázku 27.

Další scénou je nejnáročnější volumetric light scattering z kapitoly 4.11.1, kterou tvoří tři modely stromu, krajina, skybox a postprocessing efekt slunečních paprsků, viz obrázek 32.

Jako třetí je testována fur simulation z kapitoly 4.13, která desetkrát vykresluje model Suzanne s potřebnou průhledností a skybox, jako na obrázku 36.

Poslední scénu tvoří krajina, na které je umístěno 99 stromů popsaných v kapitole 4.10. Tato kompozice testuje hlavně množství vrcholů, které je možné na mobilních zařízeních použít. Testována je navíc verze s implementací metody frustum culling, která slouží k identifikaci objektů mimo pohledový objem (frustrum) definovaný view maticí. Tyto objekty následně nejsou vykresleny a tím se snižují nároky na výkon. Ukázková scéna obsahuje krajinu, která tvoří základ pro vykreslované objekty. Krajina je sestavena ze 4096 vrcholů a je obalena běžným skyboxem. Tato část scény není ovlivněna ořezáním pomocí pohledového objemu. Vliv použité metody je testován na modelu stromu o 1648 vrcholech. Součástí tohoto modelu je částicový systém pro vykreslení detailu větví s listy. Každému z 99 stromů umístěných ve scéně je přiřazena obálka ve tvaru koule. Testování oproti pohledovému objemu probíhá s pomocí tzv. obálky, kterou tvoří jednoduchý tvar obalující vlastní model. Koule byla zvolena jako nejjednodušší a nejméně náročný tvar pro testování vůči frustumu. Pro lepší znázornění jsou obálky menší než samotné modely. Pozice obálek je vizualizována pomocí krychle s hranou o velikosti průměru koule. Obrázky z vizualizace použití této metody jsou v příloze č. 2.

Výsledky testování jsou uvedeny v tabulce 5 pro všechna zařízení v jednotkách FPS i rychlost jednoho vykreslení v sekundách. Každá hodnota je průměrem za tisíc jednotlivých výsledků konkrétních překreslení v landscape módu pro jednu scénu.

Tabulka 5: Výsledky testování vybraných scén

Zařízení	*Vodní hladina [FPS]/[s]*	*Volumetric light scattering [FPS]/[s]*	*Fur simulation [FPS]/[s]*	*Stromy [FPS]/[s]*	*Stromy s frustum culling [FPS]/[s]*
Samsung Galaxy S4	58/0,017	15/0,067	59/0,017	25/0,04	57/0,018
Samsung Galaxy S3	44/0,023	22/0,045	59/0,017	35/0,028	50/0,02
Huawei Ascend y300	26/0,038	13/0,077	23/0,043	12/0,083	21/0,048
Zopo zp990	42/0,024	16/0,063	59/0,017	29/0,034	43/0,023

5.2 Testování řešení pro využitelnost v oblasti rozšířené reality

Jako první byla testována implementace senzor based augmentové reality. Použita byla všechna zařízení kromě Huawei y300, který nemá potřebné senzory. Výsledky jsou zaznamenány v tabulce číslo 6. Samotné testování je tvořeno pomocí dvou scén. První zobrazuje jednoduchý model konvice. Druhý test zkouší potenciální zhoršení výkonu na již dříve testované scéně stromů s frustum culling, kde byl skybox nahrazen obrazem z kamery.

Tabulka 6: Testování senzor based augmentové reality

Zařízení	*Scéna s konvicí [FPS]/[s]*	*Stromy s frustum culling [FPS]/[s]*
Samsung Galaxy S4	60/0,0167	56/0,0178
Samsung Galaxy S3	59/0,017	51/0,019
Zopo zp990	59/0,017	41/0,024

Z testů vychází, že vliv zobrazení obrazu z kamery a použití senzorů je prakticky neznatelný. Vykreslování není ovlivněno a po nutném zdokonalení je možné tuto metodu používat pro libovolné aplikace. Zdokonalením je myšleno zpřesnění práce se senzory a kamerou. Získaná matice je i přes snahu očistit senzorová data od šumu značně nestálá a vytváří trhavý pohyb vykreslovaného tělesa. Dále použití kamery je pouze na nejzákladnější úrovni a například obraz se nepřizpůsobuje natočení přístroje.

Druhé testování se týká použití frameworku AndAR a zúčastnila se ho všechna zařízení. Znovu jsou použity dvě scény. První pouze s jednou krychlí pro otestování rychlosti identifikace markeru a druhá s jedním tisícem krychlí. Naměřená data jsou uvedena v tabulce číslo 7.

Tabulka 7: Testování marker based augmentové reality

Zařízení	*1 krychle [FPS]/[s]*	*1000 krychlí [FPS]/[s]*
Samsung Galaxy S4	12,8/0,078	4,5/0,222
Samsung Galaxy S3	8.2/0,122	4,4/0,227
Huawei Ascend y300	29,3/0,034	4,39/0,227
Zopo zp990	11.7/0,085	4,32/0,23

Výsledky pro scénu s tisícem krychlí jsou pro všechna zařízení téměř totožné. Na výsledcích pro scénu s jednou krychlí je patrný lepší výsledek pro zařízení s podstatně nižším výkonem. Příčinou může být rozlišení zpracovávaného obrazu, které AndAR framework zvolil u tohoto zařízení nižší než u ostatních přístrojů. Rychlost vykreslování se v naprosté většině případů pohybuje ve velmi nízkých hodnotách. Tento framework postrádá podporu i dokumentaci a jeho úprava tak může být komplikovanější. Výhodou je jednoduchost řešení oproti jiným používaným nástrojům, které často vyžadují instalaci speciálních programů a enginů.

6 Diskuze výsledků

Základní otázkou této práce je, jaké efekty a detaily jsou mobilní zařízení schopna vykreslit. Celkový výkon je samozřejmě značně nižší než u PC nebo novějších konzolí. Schopnost zobrazit detail scény reprezentovaný složitostí modelů se liší mezi jednotlivými výkonnostními a cenovými třídami zařízení. Počet vrcholů, které je možné bez problémů použít, se pohybuje v desítkách tisíc. Pro zlepšení vizuálního dojmu lze použít řadu efektů, například normal mapping. Přesto je množství geometrie relativně nízké pro vykreslení kompletní detailní scény a je nutné použít mnohé kompromisy a optimalizace. Tento problém se však razantně snižuje a následující roky přinesou procesory schopné vykreslovat statisíce a víc vrcholů [30].

Pro per-pixel efekty platí, že složité shadery použité na plochy zabírající většinu obrazovky jsou značně neefektivní. Na srovnatelné úrovni jsou efekty využívající průhlednost. Vodní hladina se ve výsledcích udržela ve většině případů nad hranicí 30 FPS. Výkon však značně kolísá podle plochy, kterou zrovna voda na obrazovce zabírá. Jako velký problém se ukázalo vícenásobné čtení z textury (volumetric light scattering), které je často používáno ve složitějších efektech (například rozmazání obrazu nebo rozostření stínů).

Nejpoužívanějším způsobem pro ušetření výkonu na mobilních zařízeních jsou light mapy. Jedná se o textury, ve kterých jsou uloženy informace o osvětlení a stínech, není tedy nutné tyto efekty dynamicky vypočítávat. Nevýhodou je nemožnost měnit vlastnosti, jako je umístění zdroje světla a objektů ve scéně.

Na základě testování a dalších zdrojů [31] lze sestavit seznam zásad pro aplikace využívající OpenGL ES na mobilních zařízeních:

- Použít co nejnižší počet drawCall (volání k vykreslení geometrie).
- Využívat texture atlas (více textur spojených do jedné).
- Komprimovat textury.
- Vyměnit real time osvětlení za light mapy, vždy když je to možné.
- Vyvarovat se složitým shaderům a drahým funkcím (pow, exp, log, atd.).
- Využít v shaderech nejnižší možný precision formát (lowp, mediump, highp).
- Používat vhodné formáty atributů (např. vertexů a normál) při vytváření bufferů pro snížení velikosti přenášených dat a ve video paměti.
- Omezit postprocesing efekty a průhlednost.
- Udržet minimální množství shader průchodů.
- Kontrolovat vzájemné překreslování objektů a případně využít seřazení modelů front-to-back ve směru od kamery (back-to-front pro průhledné objekty).
- Využívat metody pro optimalizace (frustum culling, snížení rozlišení, occlusion culling, level of detail atd.)
- Implementovat více vláken pro oddělení logiky aplikace, fyziky, vykreslování apod.
- Používat back-face culling, mipmapping a VBO (Vertex Buffer Objects).

Při testování byly zaznamenány drobné rozdíly ve zpracování určitých konstrukcí v shaderech a v podání barev různých displayů. V některých výjimečných případech byl i vizuální rozdíl u stejného efektu pro různé výrobce GPU. Nejznatelněji se projevuje u délky paprsků pro volumetric light scattering.

Dnešní mobilní zařízení jsou schopna zpracovat relativně komplexní grafický obsah. Posouzení, zda je výkon dostatečný pro požadavky konkrétní aplikace, je nakonec vždy na vývojáři. Nicméně nové a výkonnější procesory včetně grafických čipů jsou pravidelně představovány každý rok a to umožňuje neustále posouvat uživatelský zážitek ze zobrazovaného obsahu. Představu o úrovni

vizualizace grafické scény si může čtenář udělat na základě provedeného testování a důkladné obrazové dokumentace.

Při implementaci efektů byl průběžně vytvářen primitivní engine, který usnadňuje další práci. Mnoho částí enginu vznikalo pouze jako nutná podpora a chybí tak určitá konkrétní struktura a plán. Dále jsou mnohé třídy implementovány v minimalistické podobě a nezahrnují některé důležité prvky, jako je například využití komprese textur. Vzniklo také několik pomocných nástrojů, které kromě vlastní funkcionality přinášejí hlavně poučení pro budoucí implementace. Hlavním takovým vedlejším nástrojem je aplikace pro převod modelů z formátu COLLADA do binárního tvaru, který značně urychluje načítání v samotném enginu.

Na začátku přípravy práce bylo vzhledem k dostupnému testovacímu zařízení zvoleno OpenGL ES 2.0. Tato verze je dnes (2015) již nahrazována OpenGL ES ve verzi 3.0 a s nejnovějším Androidem 5.0 i verzí 3.1. Důsledkem je, že práce nepočítá s některými důležitými nástroji, jako je například instancing pro opakující se geometrii.

Samotná struktura práce, která obsahuje velké množství kapitol, přináší některé nevýhody. Každá kapitola popisuje pouze základní implementaci zvoleného efektu, i když právě pokročilejší verze by mohla přinést zajímavé výsledky. Tento styl však vychází z cílů, které spočívají ve vyzkoušení schopností mobilních zařízení pracovat s 3D grafikou a realizace různých efektů.

Vývoj kvalitního enginu je však složitým počinem, který trvá měsíce a roky. Ukázková aplikace zobrazuje množství efektů, které se aktuálně na mobilních zařízeních začínají prosazovat a některé další originální. Po provedení optimalizačních a strukturálních úprav by byl vyvinutý engine plnohodnotným nástrojem pro tvorbu 3D aplikací.

V práci bylo představeno, popsáno a implementováno množství různorodých efektů aplikovatelných v 3D grafice se zaměřením na mobilní zařízení. Úvodní kapitoly seznámily čtenáře se základním postupem při přípravě enginu pro podporu následujících implementací. Stručně byly popsány formáty

a způsoby přenosu geometrie z modelovacích programů. Výsledný engine zpracovává předpřipravená data v binárním formátu a dosahuje tak výrazně rychlejšího načítání. Tato funkcionalita nebyla původně zamýšlena a vznikla jako součást další kapitoly, která se věnuje skeletálním animacím. U animací byl představen základní způsob přípravy matic kostry modelu na CPU a následný skinning provedený na GPU.

Kapitola 4.3 pokračuje popisem různých druhů osvětlení a představuje práci s shadery. V této kapitole byl kromě Blinn-Phong a Minnaert osvětlení implementován i Cook-Torrance shading efekt převážně pro simulace kovových povrchů s možností nastavení parametrů za běhu aplikace. Navazující kapitola rozšiřuje způsob výpočtu osvětlení o úpravu jeho chování na povrchu a vytváření drobných nerovností pro zvýšení detailu modelu pomocí normal mappingu. Součástí světla v reálném světě je i stín, jehož základní implementace byla představena společně s víceprůchodovými algoritmy další kapitolou. Práce pokračuje popisem vykreslování pozadí scény pomocí skyboxu a souvisejícím environment mappingem.

Několik dalších kapitol je věnováno zobrazování krajiny. Byl popsán základní způsob vytvoření a vybarvení terénu pomocí výškové mapy a byly popsány i způsoby zvýraznění vzdálenosti a hloubky scény pomocí mlhy. Následně byla krajina doplněna o vodní hladinu včetně odrazu okolí a tato scéna se stala součástí závěrečného testování. Do krajiny byla také doplněna vegetace v originálním ztvárnění stromů pomocí billboardingu. Scéna s necelým stem těchto stromů se také objevila v testu i s doplňkovou implementací optimalizační metody frustum culling. Finální scéna je dále upravována pomocí popsaných efektů postprocessingu. Součástí této kapitoly je i testovaný volumetric light scattering pro zobrazení slunečních paprsků.

Ke konci práce se čtenář dozví o jednoduché metodě pro realistické zobrazení srsti a simulaci pohybující se tkaniny. Poslední obsahovou kapitolou je pak práce rozšířena o změnu pozadí na obraz z kamery a řízení pohybu objektů pomocí senzorů. Tím je vytvořen základ pro senzor based augmented reality

a v navazující podkapitole je přidáno i využití frameworku AndAR pro marker based augmented reality.

Z testování vyplývá, že všechna zařízení jsou schopna zobrazit dostatečné množství dat pro vykreslení obsáhlých scén. Pokročilé a náročnější efekty jsou ovšem problémem, kvůli kterému je realistické zobrazování obtížně použitelné. Zařízení Samsung S3, S4 a Zopo zp990 jsou srovnatelná v rychlosti vykreslování. Při nutnosti sestavit pořadí by na prvním místě byl Samsung S4, na druhém Samsung S3 a jako třetí Zopo zp990. Překvapením mohou být některé výsledky ve prospěch Samsungu S3 oproti novější S4. Zopo zp990 si nevedlo tak dobře jako zařízení Samsung i přes novější grafický čip (Mali-450) oproti Samsungu S3 (Mali-400). Výrazně za ostatními testovanými zařízeními se umístil zástupce nejnižší třídy Huawei Ascend y300.

7 Závěr

Hlavní cíle práce, uvedené v kapitole 1.1, jsou naplněny pomocí různých vizuálních efektů implementovaných v osmnácti samostatných scénách a popsaných ve čtrnácti kapitolách. Každá scéna splňuje požadavek na zobrazení rychlosti vykreslování v jednotkách FPS a je připravena ke spuštění v přehledném seznamu s náhledovým obrázkem a názvem. Diplomovou práci zakončuje testování některých vybraných 3D scén s popisem metodologie a použitých zařízení. Zadaná implementace primitivního grafického enginu byla realizována a dovedena k funkčnímu základu pro jednotlivé scény. Další cíle jsou naplněny shrnutím závěrů z předchozího testu i celé implementace a představením seznamu doporučení pro vytváření grafických aplikací s OpenGL ES na mobilních zařízeních. Na základě výsledků práce prezentovaných v kapitole 5 (Testování vyvinutého řešení pro různé případy použití) a dále diskutovaných v kapitole 6 (Diskuze výsledků), je možné říci, že se definované cíle podařilo plně naplnit.

Rozšířit práci lze mnoha směry. Základem navazující implementace by měla být úprava struktury a rozšíření možností aktuálního enginu. Dále pak vylepšení práce s modely a skeletální animace. Vhodným pokračováním se může stát také pokročilejší implementace stále oblíbenější augmentové reality.

8 Seznam použité literatury

[1] MCHENRY, Kenton a Peter BAJCSY. NATIONAL CENTER FOR SUPERCOMPUTING APPLICATIONS UNIVERSITY OF ILLINOIS AT UR BANA-CHAMPAIGN, Urbana, IL. *An Overview of 3D Data Content, File Formats and Viewers* [online]. 2008. Dostupné z: http://www.archives.gov/applied-research/ncsa/8-an-overview-of-3d-data-content-file-formats-and-viewers.pdf.

[2] Wavefront OBJ File Format Summary. *FileFormat.info* [online]. [cit. 2015-01-21]. Dostupné z: http://www.fileformat.info/format/wavefrontobj/egff.htm

[3] Alias/WaveFront Material (.mtl) File Format. *FileFormat.info* [online]. [cit. 2015-01-21]. Dostupné z: http://www.fileformat.info/format/material/

[4] Step by Step Skeletal Animation in C++ and OpenGL, Using COLLADA Part 1. *Wazim.com* [online]. 2010 [cit. 2015-01-22]. Dostupné z: http://www.wazim.com/Collada_Tutorial_1.htm

[5] KHRONOS GROUP. *COLLADA.org* [online]. 2015 [cit. 2015-01-22]. Dostupné z: http://www.collada.org/

[6] MONDRAGÓN, Luis Jiovanni Ramírez. *Android Game Development: From Shaders to Skeletal Animation* [online]. 2014 [cit. 2015-01-23]. ISBN 978-607-00-7733-3. Dostupné z: http://www.play.google.com

[7] Step by Step Skeletal Animation in C++ and OpenGL, Using COLLADA Part 2. *Wazim.com* [online]. 2010 [cit. 2015-01-22]. Dostupné z: http://www.wazim.com/Collada_Tutorial_2.htm

[8] GDC Vault - Animation Bootcamp: An Indie Approach to Procedural Animation. *GDC Vault* [online]. [cit. 2015-01-24]. Dostupné z: http://www.gdcvault.com/play/1020583/Animation-Bootcamp-An-Indie-Approach

[9] KHRONOS GROUP. *OpenGL ES Common Profile Specification Version 2.0.25 (Full Specification)* [online]. 2010. vyd. [cit. 2014-11-14]. Dostupné z: https://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf

[10] BROTHALER, Kevin. *OpenGL ES 2 for Android: A Quick-Start Guide*. United States of America: The Pragmatic Bookshelf, 2013. ISBN 13:978-1-937785-34-5

[11] Article 11 - Beginner's Guide to Android Animation/Graphics. GRASSHOPPER.IICS. *CodeProject* [online]. 2014 [cit. 2015-01-28]. Dostupné z: http://www.codeproject.com/Articles/822412/Article-Beginners-Guide-to-Android-Animation-Gr

[12] DEMPSKI, Kelly a Emmanuel VIALE. *Advanced lighting and materials with shaders*. Plano, Tex.: Wordware Pub., c2005, xx, 340 p. ISBN 15-562-2292-0

[13] *Learn OpenGL ES | Learn how to develop mobile graphics using OpenGL ES 2* [online]. 2014 [cit. 2015-01-30]. Dostupné z: http://www.learnopengles.com/

[14] Tutorial 19, Hemispheric ambient light | digitalerr0r. *XNA Shader Programming* [online]. 2009 [cit. 2015-01-30]. Dostupné z: https://digitalerr0r.wordpress.com/2009/05/09/xna-shader-programming-tutorial-19-hemispheric-ambient-light/

[15] FOINO, Marucchi. *Game and graphics programming for ios and android with opengl es 2.0*. 1st ed. Indianapolis, IN: Wiley Pub., Inc., 2012, p. cm. ISBN 11-199-7591-3.

[16] RUSSEL, Eddie. Know the Difference: Bump, Normal and Displacement Maps. In: *Digital-tutors: a pluralsight company* [online]. 2014 [cit. 2015-02-03]. Dostupné z: http://blog.digitaltutors.com/bump-normal-and-displacement-maps/

[17] 3D C/C++ tutorials - OpenGL 2.1 - GLSL cube mapping. *3D C/C++ tutorials* [online]. 2015 [cit. 2015-02-06]. Dostupné z: http://www.3dcpptutorials.sk/index.php?id=24

[18] Cube Maps: Sky Boxes and Environment Mapping.*Anton's OpenGL 4 Tutorials* [online]. 2014 [cit. 2015-02-07]. Dostupné z: http://antongerdelan.net/opengl/cubemaps.html

[19] Fog Outside. BUBNAR, Michal. *Megabyte Softworks: C++, OpenGL, Algorithms* [online]. 2009 [cit. 2015-02-11]. Dostupné z: http://www.mbsoftworks.sk/index.php?page=tutorials&series=1&tutorial=15

[20] Tutorial 6: Water Reflection. *Http://www.virtual-vision.net* [online]. 2014 [cit. 2015-02-14]. Dostupné z: http://www.virtual-vision.net/19.html

[21] Billboarding Tutorial: Cheating - Faster but not so easy. FERNANDES, António. *Lighthouse3d.com Mostly about 3D graphics* [online]. 2014 [cit. 2015-02-16]. Dostupné z: http://www.lighthouse3d.com/opengl/billboarding/index.php?billCheat2

[22] OpenGL Programming/Post-Processing. WIKIMEDIA FOUNDATION. *Wikibooks* [online]. 2015 [cit. 2015-02-19]. Dostupné z: http://en.wikibooks.org/wiki/OpenGL Programming/Post-Processing

[23] Framebuffers. DE VRIES, Joey. *Learn OpenGL, extensive tutorial resource for learning Modern OpenGL* [online]. 2014 [cit. 2015-02-19]. Dostupné z: http://www.learnopengl.com/#!Advanced-OpenGL/Framebuffers

[24] GPU Gems 3 - Chapter 13. Volumetric Light Scattering as a Post-Process. NVIDIA CORPORATION. *NVIDIA Developer Zone* [online]. 2007 [cit. 2015-02-21]. Dostupné z: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch13.html

[25] MOSEGAARD, Jesper. Mosegaards Cloth Simulation Coding Tutorial. *Computer Graphics Lab* [online]. 2009 [cit. 2015-02-22]. Dostupné z: http://cg.alexandra.dk/?p=147

[26] Fur Effects - Teddies, Cats, Hair .... *XBDEV.NET* [online]. 2014 [cit. 2015-03-05]. Dostupné z: http://www.xbdev.net/directx3dx/specialX/Fur/

[27] Android Augmented Reality Tutorial – Part 1: Device Orientation. *Techoutbreak* [online]. 2011 [cit. 2015-03-10]. Dostupné z: http://techoutbreak.com/2011/android-ar-tutorial-part-1-device-orientation/

[28] Android: Render OpenGL on top of camera preview. *Digitalbreed* [online]. 2009 [cit. 2015-03-10]. Dostupné z: http://digitalbreed.com/2009/android-render-opengl-on-top-of-camera-preview/comment-page-1

[29] DOMHAN, Tobias. Andar: AndAR - Android Augmented Reality. *Code.google.com* [online]. 2010 [cit. 2015-03-12]. Dostupné z: https://code.google.com/p/andar/wiki/HowToBuildApplicationsBasedOnAndAR

[30] Soft Kitty: An overview of the latest OpenGL ES 3.0 technical demo from Imagination. IMAGINATION TECHNOLOGIES. *Imagination* [online]. 2014 [cit. 2015-03-15]. Dostupné z: http://blog.imgtec.com/powervr/soft-kitty-overview-latest-opengl-es-3-0-tech-demo-imagination

[31] Optimizations. UNITY TECHNOLOGIES. *Unity - Game engine, tools and multiplatform* [online]. 2014 [cit. 2015-03-15]. Dostupné z: http://docs.unity3d.com/Manual/MobileOptimisation.html

9 Přílohy

1) Extended abstract
2) Obrázky z implementace frustum culling
3) Zadání práce
4) Přiložený apk soubor a kódy aplikace

Příloha č. 1 Extended abstract

Possibilities for development and use of 3D applications on the Android platform

Tomáš Marek
Faculty of Informatics and Management
University of Hradec Kralove,
Hradec Kralove, Czech Republic
tomas.marek.5@uhk.cz

***Abstract* — Computer graphics in combination with mobile devices finds many applications in the fields of entertainment, education and displaying data. The amount of information that can be shown to the user largely depends on the optimization of the graphic chain during application development. For the development of such a solution is needed to become familiar with the possibilities and differences of the platform. Appropriate way can be implementation of the graphics engine and sample scenes that show a basic level of current devices.**

Keywords- mobile devices; engine; computer graphics;

I. INTRODUCTION

Mobile devices increasingly substitute desktop or notebook to many users. Their constantly increasing power enables developing of application environment in which is possible to find tools for solving many tasks. These devices are also constantly available and ready for use. Wide availability and performance are increasing the demand for smarter applications in various fields.

One possibility is using computer graphics with interface Open Graphics Library for Embedded Systems (OpenGL ES) for Android and a new Metal API (application programming interface) introduced with IOS8 [1]. The range of application is very wide. For example, effects for recording video and photos as depth of field effect [2] or 3D talking avatar [3] presented at the APSIPA conference [4]. Significantly most profitable and most popular categories across applications are games, however, 2D and 3D graphics finds use in other popular categories such as education or transportation. Generally graphic can be used for professional programs modules, viewing data or augment reality. Even cheap mobile device may display large medical data by volumetric rendering [5]. With this trend there are new mobile versions of the large graphics engines such as Unity, Unreal engine or CryENGINE. For developers and studios arises question how to start creating graphical content.

Freely available engines have several disadvantages that need to be considered. The first is the overall quality and speed of updates. The developer has no control over the engine and must rely that errors will be corrected in time. Some modules can also be written inappropriately for specific solutions. The big advantage is verified cross-platform solution. Another problem is the learning rate. Engines come with their own development tools and their full mastery may take as long as creating a new engine. In many cases is better to use smaller custom solutions.

II. PROBLEM DEFINITION

What can graphics engine contain is described by figure 1.

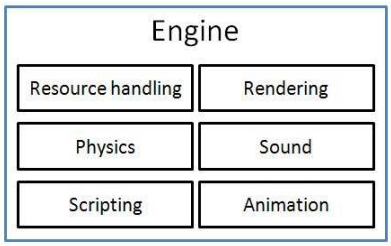


Figure 1. Basic graphics engine scheme

Generally, 2D and 3D engines tools for effective work with graphics for the purposes of display content to the user. Engines may also have other tools, such as particle system or scenegraph. This article outlines the essential parts of the engine for rendering and resource handling.

Resources represent data loaded and processed by engine. These are the object models, their textures, sounds, but also shaders, animations, scripts and more. The models are the elements of the real world or designer mind and are created in special modeling tools such as Blender or Autodesk 3ds Max. Models can also be called a mesh data and are the sum of all the information needed to render the scene geometry. For transmission models between programs and engines there are different kinds of formats, for example. OBJ [6], COLLADA [7], 3DS [8] and others.

An important part of the program are shaders. These are short programs that affect the course of rendering. They are divided into vertex and fragment shaders [9] according to their focus. All rendering data passes through them. Vertices of rendered objects are transformed by the vertex shader. Data for these operations can be supplied to the shader during rendering using uniform variables. In contrast to uniform, which contains the same data for a model, the model properties (position, normal, color etc. for each vertex) are transmitted via the attribute variables. This involves matrices required for proper rotation and deformation of the scene. The result of the vertex shader is further processed and used in the fragment shader, which is used for conducting operations at the level of individual pixels. The transferred data between shaders are referred to as varying variable.

Displaying graphical content is almost not possible without the use of textures. Texture is a 2D image, generally defining a color using the RGB (color mixing method), which is mapped to the surface of the body. Textures are used for a wide variety of effects, such as to define a roughened surface, for storing shadow or highlight, for glitter or directly specular reflections. Only rarely is possible to map a texture to an object in the ratio of one to one. OpenGL in other cases use the settings for texture filtering, which can define a method of obtaining data from the texture. Various settings have different results and different impacts on the rendering speed.

Optimization of data means minimizing their size or preliminary preparation to simplify subsequent rendering. Shaders together with complex models and textures are the main engine load. With increasing complexity of the scene may contain more complex calculations for physics, collisions, animation and more. Rendering can be described as the transfer of data to the graphics card and their passage through the graphic chain, which includes shaders. To reduce the complexity is worthwhile to consider over data types, the use of compressed textures, texture atlases, optimization model data, the method of rendering etc. Rendering meant processing while data sending on the graphics card. It is a use of glDrawArrays call for not indexed data and glDrawElements for indexed data [10]. An index for data leads to reduction duplication and mostly to increase rendering speed. By parameter mode you can set the the type of graphics primitives to be rendered. The aim is to speed up rendering and reduce the data size. Graphics primitives are points, lines, triangles, special kinds connected triangles (TRIANGLE_STRIP, TRIANGLE_FAN) and others. Before calling the renderer is required to load and send data to the graphics card. For various scenes there are different ways of transmission data. The basis is vertex arrays object (VAO). The data are then stored in RAM memory. A better solution is to then use vertex buffer object (VBO) and store the data in video memory.

III. NEW SOLUTION

Android devices to work with 2D and 3D graphics use programming interface (API), OpenGL ES (Open Graphics Embedded Systems Libraryfor), which is supported directly by processors. This interface is used for basic processing vector graphics and is available in several versions. The most widespread OpenGL ES 2.0 is supported from Android 2.2 Froyo.

Future engine will use a modular structure. The basis packages will display geometry, load textures and shaders. OpenGL is used in Android through GLSurfaceView class. This view shows the final render of OpenGL thread and is assigned to the main activity that caters to run the application and possibly user input.

OpenGL thread is represented by renderer class that uses the other modules. The result of drawing is stored in the graphics cards framebuffer (image output device). Objects are composed of graphics primitives, which are the basis for rendering OpenGL and are set when calling glDraw. To define the basic shape is needed to set vertices (vertices) where each vertex is a point in space. With each vertex are passed its properties and connection of these points, which define the entire model. Figure number 2 shows a simplified graphical chain process.

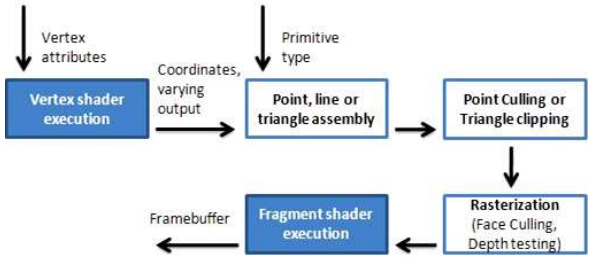


Figure 2. Graphics pipeline

The result of vertex shader, which is running for each vertex, are final vertex coordinates after the transformation and eventual varying variables. The next step, except for automatic translation for the following processes, is creation of graphics primitives and cropping to view volume. Following is rasterization (converting to 2D image), starting fragment shader and save the value to framebuffer. In case that the triangle is partially offscreen, some of its vertices are removed and to maintain the shape new points are connected to the edge the view volume. When processing fragment (color, depth, varying data) is sometimes necessary to re-read the previous record from framebuffer. For example, in dealing with transparency.

On the provided base can be implemented variety of graphic scenes and effects that help test the capabilities of current devices and embedded engine. Any such scene uses the base class renderer and helps extend engine with new features. Many effects are created mainly using shaders. In addition to the basic classes for working with geometry and data is also implemented a package for loading models. These are the vertices organized into specific shapes and optionally containing additional information (texture, normal vectors etc.). Selected scenes can be used for testing the devices and the engine.

Water level scene

The first realized scene is terrain with trees models and water level. The main idea behind the water surface is to use only four peaks forming the base area, normal mapping (change the behavior of light on the surface of the object to create a small bumps) to create waves and texture carrying reflection of the environment. The scene is shown in figure 3. To create the reflection is necessary to render geometry once more and use the resulting image in the second rendering.

Figure 3. Water level scene

Volumetric light scattering scene

This effect is based on posprocessing. The finished scene is further adjusted to the level of a 2D image. Volumetric light scattering effect is formed by light scattering in the environment with a high content of dust particles, water vapor atc. The effect is realized by the accumulation of the light intensity towards the source and requires a large amount of texture reading. To create separate beams of light is necessary to add one pass forming special texture. It is therefore needed to render the scene, another pass for preparing special texture and ultimately by the final pass create effect shown in figure 4.

Figure 4. Volumetric light scattering scene

Fur simulation scene

Fur simulation is a simple effect based on geometry repetition. By repeating are formed layers which, in conjunction with a semi-transparent noise texture create effect of fiber, as shown in figure 5.

Figure 5. Fur simulation scene

Forest scene

The last scene form the landscape where is located 99 trees. This composition primarily tests the number of peaks that can be used on mobile devices. Testing is moreover version implementation frustum culling, which is used to identify objects outside the viewing frustum defined by the view matrix. These objects are not rendered, thus reducing power requirements. Sample scene contains a landscape which forms the basis for drawing objects. The landscape is composed of 4096 vertices and is wrapped by skybox. This part of the scene is not affected by cropping using the view volume. The influence of the applied method is tested on a tree

model consisting of 1648 vertices. Figure 6 shows forest scene and and bounding volumes for each tree as a simplified model for frustum culling.

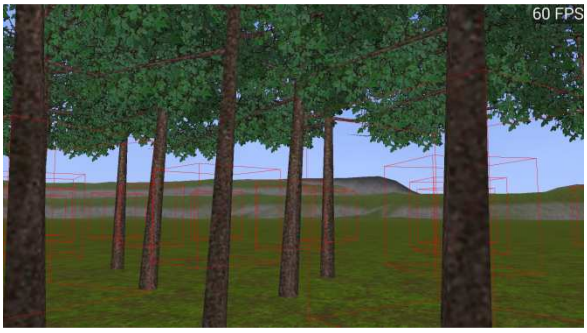


Figure 6. Forest scene with Bounding Volumes

IV. IMPLEMENTATION

For working with graphics is needed OpenGL ES and in Android is necessary to initialize it using a special class GLSurfaceView. This class provides displays configuration, rendering in special thread and helps manage the life cycle of activity [9]. Rendering is then performed on the part of display called surface or viewport. This view is subsequently allocated to activity as the current content view [11]. With GLSurfaceView is also possible to control the type of rendering (setRenderMode) and set the continuous rendering or rendering on request. Now you need to create a class that will cater the actual drawing to OpenGL surface view. This class implements GLSurfaceView.Renderer and contains three basic methods:

- **OnSurfaceCreated** – this method is called when you create a surface. Here it is possible to do data loading and creating buffers.
- **OnDrawFrame** – called whenever it is necessary to render something. Here is a call glDraw.
- **OnSurfaceChanged** – called for the surface formation and whenever the image is resized, for example, when you rotate the display.

Basic activity with GLSurfaceView and renderer form the basis for future engine. Another element is to ensure communication between the user thread (activity) and OpenGL thread in the background. The UI thread can directly work with renderer methods and transmit information about the user's input. Such basic information is finger move across the screen, which can be determined by OnTouchListener in the UI thread, or GLSurfaceView. But if it is necessary to ensure, for example, displaying FPS (redraw speed - Frame Per Second), it is a communication from a background thread to the main thread, and you must use a Handler (communication between the fibers Android). FPS value can be easily displayed in the activity label or in connected view, where it is also possible to define additional UI elements. After securing the communication with the main UI thread basis is finally ready for the implementation classes and packages that will form the actual functionality of the engine. Those are utility classes for working with buffers and matrices, classes caring for animation, shaders, data loading and any future effects. Conceptual design is shown in figure 7.

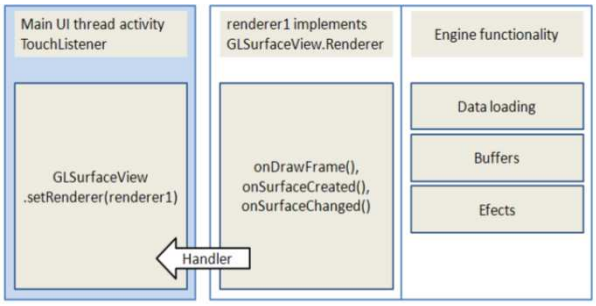


Figure 7. Engine scheme

Application main activity (shown in figure 8) is stored in the main package. This activity is responsible for the main menu, which will contain a list of all the activities planned for each scene and ensure that they run after the tap.

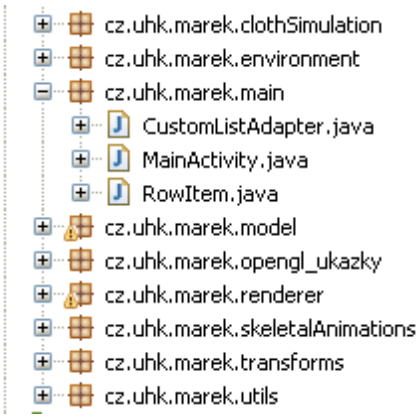


Figure 8. Package structure

To display each start the scene link is used ListView, which renders scrolling list items. Each item will include a preview scene image, the main description and any additional notes to the sample and is representative of a class rowItem, see in Figure 9.

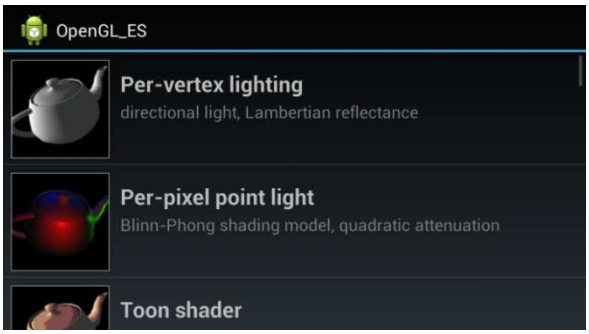


Figure 9. Main activity with list view

This data is passed to a ListView using CustomListAdapter class, which is an extension of BaseAdapter. CustomListAdapter will then take care of acquisition and allocation of appropriate data for each row in ListView. Rows of the ListView then refer to the activity of GlSurfaceView. All these activities are stored in the package opengl_ukazky. Renderer package then contains classes for rendering assigned to each of the activities from opengl_ukazky. Packages skeletalAnimations and clothSimulation will contain the necessary operation for the planned implementation of these effects. Package environment is prepared for securing data for effects showing scene elements, such as the preparation of the underlying flat surface. Within the package model will be implemented class to represent geometry objects, which will use the base class for creating buffers, shaders and textures preparation of utils package. Last package transforms contains utility classes for mathematical computations with matrices, vectors and vertices.

V. TESTING OF DEVELOPED APPLICATION

Selected scenes were tested on the main performance indicator, which is the rendering speed. Rendering speed indicator is displayed in each scene using unit FPS (frame per second). The goal of testing is to assess the capabilities of current devices in displaying complex graphics scenes or draw conclusions above implementations. FPS provides rate information redrawing and relative minimum to maintain a smooth movement in the scene is 25 to 30 FPS. This value is very misleading to measure the actual performance. For example, deterioration of 10 FPS has a different meaning in the case of deterioration from 100 FPS or 20 FPS. Frame per second is the scaled value from the speed of rendering a scene. The consequence is that to change of 10 FPS is needed various loads. For these reasons are also given information about the rendering speed in seconds. It is worth noting that Android devices have a fixed frame rate to 60 FPS. Tests were performed on devices listed in table 1.

Device	*SoC*	*CPU*	*GPU*	*RAM*
Samsung S4	Qualcomm Snapdragon 600 APQ8064T	Krait 300, 1900 MHz, Cores: 4	Qualcomm Adreno 320, 400 MHz, Cores: 4	2 GB, 600 MHz
Samsung S3	Samsung Exynos 4 Quad 4412	ARM Cortex-A9, 1400 MHz,Cores: 4	ARM Mali-400 MP4, 440 MHz,Cores: 4	1 GB, 400 MHz
Huawei y300	Qualcomm Snapdragon S4 Play MSM8225	ARM Cortex-A5, 1000 MHz, Cores: 2	Qualcomm Adreno 203	512 MB
Zopo zp990	MediaTek MT6592	ARM Cortex-A7, 1700 MHz, Cores: 8	ARM Mali-450 MP4, 700 MHz, Cores: 4	2 GB, 666 MHz

Table 1. Devices for testing

The test results are listed in table 2 for all the devices in units of the FPS and the speed of one rendering in seconds. Each value is the average of a thousand individual results specific to redraw in the landscape mode for one scene.

Device	*Water level [FPS]/[s]*	*Volumetric light scattering [FPS]/[s]*	*Fur simulation [FPS]/[s]*	*Forest scene [FPS]/[s]*	*Forest scene with frustum culling [FPS]/[s]*
Samsung S4	58/0,017	15/0,067	59/0,017	25/0,04	57/0,018
Samsung S3	44/0,023	22/0,045	59/0,017	35/0,028	50/0,02
Huawei y300	26/0,038	13/0,077	23/0,043	12/0,083	21/0,048
Zopo zp990	42/0,024	16/0,063	59/0,017	29/0,034	43/0,023

Table 2. The results of testing

The overall performance is obviously significantly lower than the PC or the newer consoles. The ability to show detail scenes represented model complexity varies between performance and price classes of devices. The number of vertices that can be used without problems ranging in tens of thousands. To improve the visual impression can be used a variety of effects, such as normal mapping. Still the amount of geometry is relatively low for rendering complete detailed scenes and it is necessary to use many compromises and optimization. However this

problem dramatically decreases and the following years will bring processors capable of rendering hundreds of thousands and more vertices. For per-pixel effects applies that complex shaders used in areas occupying most of the screen are largely ineffective. At a comparable level are effects using transparency. The water level in the results maintained in most cases above the 30 FPS. Performance varies considerably according to the area that water just occupies on the screen. As a big problem was revealed multiple reading from textures (volumetric light scattering) which is often used in more complex effects (such as image blur blur or shadow).

VI. CONCLUSIONS

On the implementation of the basic graphics engine has been tested five different scenes. Created engin is a source of experience and can be the basis for any graphics applications using OpenGL ES on Android platform. At the same time were tested capabilities of modern mobile devices. An appropriate use of the identified properties can create advanced graphics applications. Due to faster development, less content and easier publishing are mobile applications compared to PC programs a significant opportunity to achieve profits.

VII. REFERENCES

[1] IOS8. APPLE INC. *Apple* [online]. [cit. 2014-12-18]. Dostupné z: https://www.apple.com/cz/ios/

[2] Wang,QS., Yu. Z, Rasmussen, C, Yu, JY.: Stereo vision-based depth of field rendering on a mobile device, In JOURNAL OF ELECTRONIC IMAGING, Volume 23, Issue 2, 2014, DOI: 10.1117/1.JEI.23.2.023009

[3] Lin, Hj., Jia.J, Wu, XJ, Cai, LH.: Stereo TalkingAndroid: An Interactive, Multimodal and Real-time Talking Avatar Application on Mobile Phones, In 2013 ASIA-PACIFIC SIGNAL AND INFORMATION PROCESSING ASSOCIATION ANNUAL SUMMIT AND CONFERENCE (APSIPA), 2013, DOI: 10.1109/APSIPA.2013.6694211

[4] Asia-Pacific Signal and Information Processing Association [online]. 2014 [cit. 2014-12-18]. Dostupné z: http://www.apsipa.org/

[5] Hachaj, T.: Real time exploration and management of large medical volumetric datasets on small mobile devices-Evaluation of remote volume rendering approach, In INTERNATIONAL JOURNAL OF INFORMATION MANAGEMENT, Volume 34, pp. 336-343, 2014, DOI: 10.1016/j.ijinfomgt.2013.11.005

[6] Wavefront OBJ File Format Summary. *FileFormat.info* [online]. 2014 [cit. 2014-12-18]. Dostupné z: http://www.fileformat.info/format/wavefrontobj/egff.htm

[7] KHRONOS GROUP. *Collada* [online]. 2014 [cit. 2014-12-18]. Dostupné z: https://collada.org/

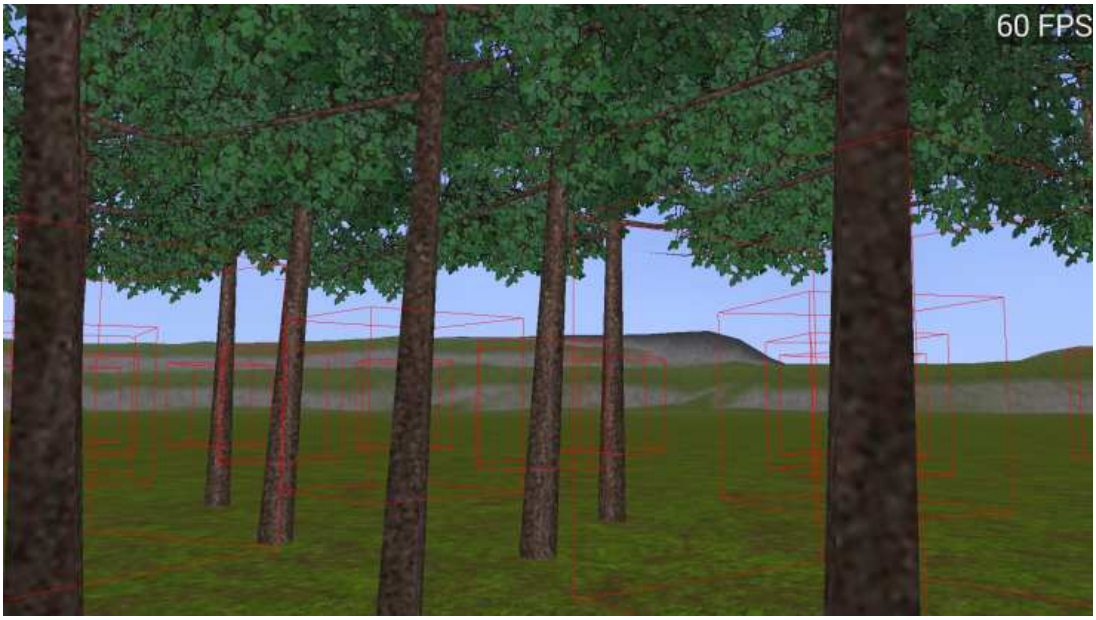
[8] Autodesk 3D Studio File Format Summary. *FileFormat.info* [online]. 2014 [cit. 2014-12-18]. Dostupné z: http://www.fileformat.info/format/3ds/egff.htm

[9] BROTHALER, Kevin. *OpenGL ES 2 for Android: A Quick-Start Guide*. Raleigh: The Pragmatic Programmers, 2013. ISBN 978-193-7785-345.

[10] KHRONOS GROUP. *OpenGL ES Common Profile Specification Version 2.0.25 (Full Specification)* [online]. 2010. vyd. [cit. 2014-11-14]. Dostupné z: https://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf

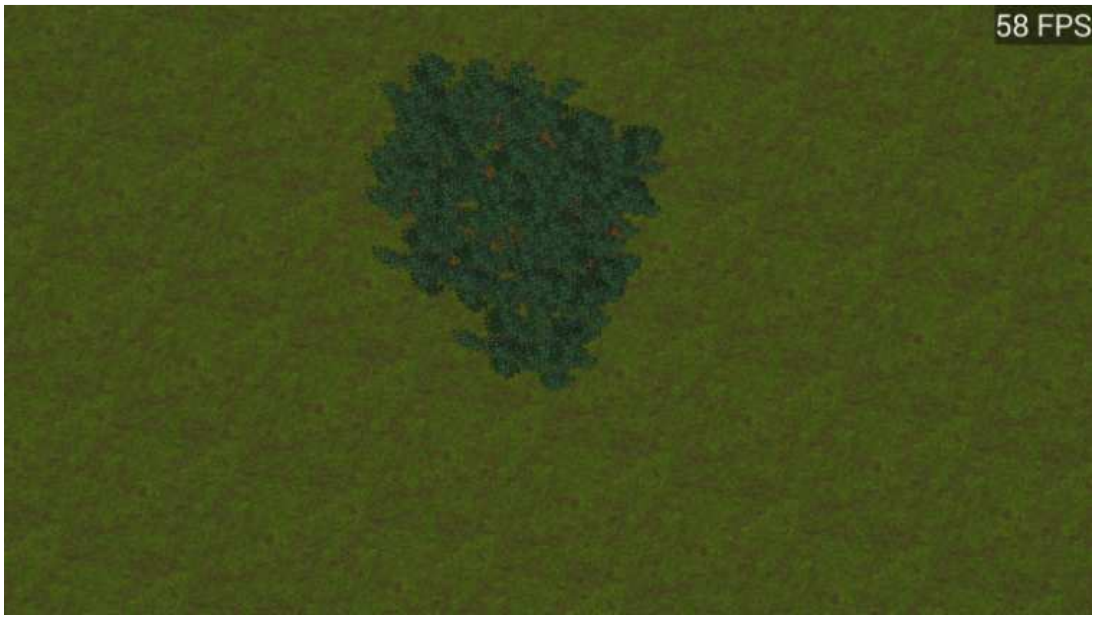
[11] Article 11 - Beginner's Guide to Android Animation/Graphics. GRASSHOPPER.IICS. *CodeProject* [online]. 2014 [cit. 2015-01-28]. Dostupné z: http://www.codeproject.com/Articles/822412/Article-Beginners-Guide-to-Android-Animation-Gr

Příloha č. 2 Obrázky z implementace frustum culling



Scéna z pohledu pozorovatele se znázorněním umístění a velikosti testovacích obálek

Stejná scéna při pohledu ze shora bez frustum culling



Scéna pri pohledu shora s aplikovanou metodou frustum culling

UNIVERZITA HRADEC KRÁLOVÉ
Fakulta informatiky a managementu
Rokitanského 62, 500 03 Hradec Králové, tel: 493 331 111, fax: 493 332 235

Zadání k závěrečné práci

Jméno a příjmení studenta: Tomáš Marek

Obor studia: Aplikovaná informatika (2)
Jméno a příjmení vedoucího práce: Ondřej Krejcar

Název práce:
Možnosti vývoje a použití 3D aplikací pro mobilní zařízení na platformě Android

Název práce v AJ:
Possibilities for development and use of 3D applications for mobile devices on the Android platform

Podtitul práce:

Podtitul práce v AJ:

Cíl práce: Analýza možností 3D grafiky na mobilních platformách, identifikace slabých míst a návrh efektivního řešení. Popis případných budoucích rozšíření a možných optimalizací.

Osnova práce:

1. Teoretický úvod
2. 3D grafika na mobilních zařízeních
    1. Možnosti 3D vizualizací na mobilních platformách
3. Návrh SW rámce pro 3D vizualizace na mobilních zařízeních
4. Implementace navrženého řešení
    1. Formáty pro ukládání 3D objektů
    2. Animování 3D objektů
    3. Osvětlení scény
    4. Normal mapping
    5. Shadow mapping
    6. Skybox, environment mapping
    7. Vykreslení terénu
    8. Použití mlhy pro zdůraznění hloubky scény
    9. Vodní hladina
    10. Billboarding pro vykreslení vegetace
    11. Postprocessing
    12. Cloth simulation
    13. Fur simulation
5. Testování vyvinutého řešení pro různé případy použití
    1. Testování HW nároků
    2. Testování řešení pro využitelnost v oblasti rozšířené reality
6. Diskuze výsledků
7. Závěr
8. Literatura
9. Extended abstract

Projednáno dne:

Podpis studenta Podpis vedoucího práce