VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Grafický engin pro tvorbu Android aplikací

Graphical engine for Android application development

2014 Roman Horňák

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Roman Horňák**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Grafický engin pro tvorbu Android aplikací
Graphical Engine for Android Application Development

Zásady pro vypracování:

Práce se zabývá vývojem Java Android knihovny, která bude sloužit k tvorbě aplikací založených na vykreslování 2D a 3D objektů pomocí OpenGL ES 2.0. Součástí knihovny bude render pracující s VBO, řešící manipulaci s objekty. Knihovna bude umožňovat vytvářet předdefinované 2D objekty pro tvorbu menu a ovládacích prvků.

1. Implementace v jazyce Java, Android SDK a OpenGL ES 2.0.
2. Načítání textur, práce s audio soubory, shadery, operace s kamerou.
3. Zobrazení mesh objektů, parser mesh data souborů.
4. Skeletální animace objektů, parser skelatal data souborů.
5. Řešení kolizí mezi objekty.
6. Implementace ukázkové aplikace.

Seznam doporučené odborné literatury:

[1] Reto Meier, Professional Android 4 Application Development, Wrox, 2012, ISBN-13: 978-1118102275
[2] Cay S. Horstmann, Core Java(TM), Volume I--Fundamentals, Prentice Hall, 2007, ISBN-13: 978-0132354769
[3] Aaftab Munshi, Dan Ginsburg, OpenGL ES 2.0 Programming Guide, Addison-Wesley, 2008, ISBN-13: 978-0321502797

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Mgr. Ing. Michal Krumnikl**

Datum zadání: 01.09.2013

Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7.5.2014 .....

.....

Tímto bych rád poděkoval mému konzultantovi Michalu Krumnikovi, který mě poskytl cenné rady a postřehy při řešení Bakalářské práci.

Abstrakt

Bakalářská práce je zaměřena na problematiku vývoje knihovny určené k tvorbě graficky náročných aplikací pro Android platformu. Dále řeší využití hardwarové akcelerace při vykreslování scény. Animace a transformace mesh objektů. Práci s audio soubory, texturami a shadery. Nastavení a ovládání kamery, či možnost využití fyzikálního enginu. Hlavním cílem práce je tedy vytvoření komplexní knihovny poskytující prostředí pro rychlý a snadný vývoj aplikací.

Klíčová slova: Android, OpenGL ES, Java, knihovna, API

Abstract

Bachelor thesis is focused on issue of development a library to creating high graphical applications for Android platform. Then solves usage of hardware accelelaration. Animation and transformation of mesh objects. Work with audio files, textures and shaders. Configuration and camera controls or the possibility of using physics engine. The main purpose of this thesis is creation of comprehensive library that provides interface for fast and easy development of aplications.

Keywords: Android, OpenGL ES, Java, library, API

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface
EGL	–	Enterprise Generation Language
GC	–	Garbage Collector
GLU	–	OpenGL Utility Library
GLSL	–	OpenGL Shading Language
JNI	–	Java Native Interface
MVP	–	Model View Projection
OpenGL	–	Open Graphics Library
OpenGL ES	–	OpenGL for Embedded Systems
SDK	–	Software Development Kit
VBO	–	Vertex Buffer Object

Seznam obrázků

Seznam výpisů zdrojového kódu

Obsah

1 Úvod

Možnost vypracování kvalitní Android knihovny, zaměřené na vývoj grafických aplikací, jako bakalařské práce, byla z hlediska budoucího praktického využití velkou motivací. Převážně práce se systémem OpenGL ES, jako rozhraním pro komunikaci mezi aplikací a video zařízením pro dosažení hardwarové akcelerace, byla inspirujícím aspektem bakalářské práce.
První kapitoly se zabývají náhledem na vyvýjenou knihovnu, její schopnosti, strukturu a prostředí. Zmíněno je i základní nastavení zařízení a možnosti přizpůsobení systému.
Následující kapitola popisuje práci s obrázky jako texturami v OpenGL. Zjištění omezujících kritérií daného zařízení. Možnosti nastavení vlastností reprezentace textur a teoretický náhled interpolace textur pro hladké zobrazení.
Při práci s OpenGL ES 2.0 (případně 3.x) je nutné použít vlastní GLSL shadery pro výslednou rasterizaci 3D scény. Kapitola věnovaná tomuto problému se zabývá kompilací zdrojových kódů shaderů a popisem základních předdefinovaných shaderů.
Nedílnou součástí je i práce s kamerovým systémem. Popis vlastností a možnosti přizpůsobení či operace s kamerou jsou důležitým aspektem pro následné nastavení kamery a získání výsledného dojmu z pohledu na scénu.
Dále je zde zahrnuta práce s polygony, mesh objekty. Popis struktury nově vytvořeného souboru pro ukládání mesh dat, jež lze exportovat z 3D modelovacího programu Blender. Shrunutí výhod mezi systémovými a video buffery, kde jsou mesh data uchována a jejich implementace. Řešení matic umožňujících základní transformace jako je translace, rotace a změna měřítka či billboarding pro zamezení nevhodného natočení objektu.
Následuje popis implementace skeletální animace umožňující deformaci polygonu. Výpočet transformačních matic pomocí kostí skeletu a následná aplikace na vrcholy polygonu. Také skeletální animace lze vytvořit a exportovat v programu Blender s následným importem do aplikace.
Zmíněna je i práce s audio soubory a nakládáním s jejich zdroji při přehrávání. Také řešení kolizí mezi objekty, případně možnost využití některého z fyzikálních enginů třetích stran.
Důležitou součástí knihovny je samostatnost při nakládání s různými zdroji, kdy je nutné naslouchat životnímu cyklu Android aplikace pro korektní správu zvukových stop, mesh dat, textur a jiných OpenGL zdrojů.

2 Představení knihovny

Podmětem pro vývoj nové knihovny byla vize vytvořit kvalitní a zároveň srozumitelné rozhraní určené k programování graficky náročných aplikací pro zařízení s operačním systémem Android. Pro samotné programování byl zvolen OOP jazyk Java z důvodu jeho všeobecné znalosti a popularity u široké veřejnosti. Knihovna poskytuje API pro rychlý a jednoduchý vývoj zobrazování 2D či 3D prostředí aplikace.
Pro rasterizaci jednotlivých objektů na výsledné scéně v reálném čase (prakticky 30 a více krát za sekundu) je nutná hardwarová akcelerace, která je podporována s použitím OpenGL ES rozhraní jež komunikuje přímo s grafickým čipem daného zařízení. Především využití VBO nabízí nezbytně podstatnou část výkonu, kdy se data uchovávají raději ve video paměti zařízení nežli v systémové paměti, a proto může být objekt vykreslen přímo z grafického čipu.
Dále je zde zahrnuta podpora vytváření OpenGL ES kontextu a automatická manipulace při ztrátě tohoto kontextu. Veškeré textury, shadery a ostatní OpenGL zdroje jsou při návratu k aplikaci obnoveny. Automaticky je postaráno i o samotné vytváření textur, shaderů a VBO kdy je nutné tyto data inicializovat pomocí OpenGL kontextu, tedy pomocí vlákna ve kterém OpenGL pracuje. Operace spojené s vytvářením jednotlivých OpenGL struktur jsou vždy přeposlány do správného vlákna a jsou provedeny v nejbližším možném kroku. Shader program je generován z vertex a fragment GLSL kódu a automaticky vytváří ukazatele na uniform a attribute proměnné.
Mesh data jednoduchých objektů lze generovat přímo pomocí statických metod. Složitější objekty lze exportovat prostřednictvím Python addonu pro 3D modelovací program Blender a následně importovat struktury do knihovny a bufferů. Addon dále poskytuje export animací v podobě deformací aplikovaných na armaturu objektu s přidělením vrcholů k jednotlivým kostem skeletu. Implementace taktéž zahrnuje plně upravitelné komponenty pro tvorbu UI prostředí.
Knihovna dále zahrnuje tvorbu jak orthografické tak i perspektivní kamery, které podporují základní transformace jako je translace, rotace a zoom.
Při práci s audio soubory a zvukovými efekty je zde zavedena podpora automatické manipulace při ztrátě kontextu aplikace. Veškeré přehrávání podporovaných audio souborů je obnoveno nebo pozastaveno dle aktuálního stavu aplikace.
Prostředí umožňuje práci se soubory na libovolném úložišti. Načítání souborů z interního úložiště aplikace, assets a resources složky, SD karty zařízení, případně z webového úložiště či FTP serveru. Ukládat data je možné na interní úložiště aplikace, SD kartu a FTP server.
Zahrnuty jsou i základní matematické funkce pro testování průniku různých strukur nebo jejich vzdálenosti. Pro simulaci fyzikálního prostředí a kolizí v 2D světě je použit kvalitní a optimalizovaný engin Box2D.
Knihovna umožňuje vyvíjet jak 2D tak i 3D tituly založené na OpenGL ES pro získání nezbytného výkonu a dosažení hladkých animací. Jednoduchost a srozumitelnost kódu zaručuje rychlou cestu k tvorbě kvalitních aplikací jak pro zkušené, tak i pro méně zkušené vývojáře. Implementace umožňuje variabilní nastavení dle preferencí vývojáře přičemž však zachovává velkou míru výkonosti celého systému.

2.1 Obdobné projekty

Na trhu již existuje mnoho profesionálních produktů, které řeší problematiku vytváření aplikací pro mobilní zařízení a nové stále přibívají. Mnohé z nich jsou však za nemalé poplatky. Produkty jež jsou zdarma jsou většinou příliš obecné a tím složitější na pochopení, přičemž je kladen velký důraz na znalosti uživatele ohledně jednotlivých postupů při vývoji o optimalizaci aplikace. Dalším problémem těchto knihoven je jejich aktuálnost, kdy mnohé přestávají být v průběhu času podporovány a neumožňují tak uživatelům plně využít možnosti novějších technologií. Pro srovnání byly vybrány především OpenSource projekty, které se charakteristikou podobají vyvíjené knihovně.

2.1.1 AndEngine

Tato knihovna je určena pro tvorbu výhradně 2D prostředí. Jedná se o velmi populární vývojový nástroj právě pro 2D hry s podporou načítání Tile Map a Box2D jako fyzikálního a kolizního enginu. Původně byla knihovna vytvořena pro GLES 1.x rozhraní, později však bylo implementováno GLES 2.0 rozhraní s možností tvorby vlastních shaderů a akcelerace pomocí VBO. Bohužel autor AndEngine toto prostředí již dále nevyvíjí a veškerá podpora o oprava chyb zůstala na komunitě podporující tento engin. Zdrojové kódy ovšem zůstaly nadále volně dostupné a mnoho vývojářů na této knihovně staví svoje projekty. Dalším problémem této knihovny je chybějící dokumentace.

2.1.2 libGDX

Jedná se o prostředí, které bylo původně vyvinuto pro tvorbu 2D her určených na Android zařízení. Postupem času a s velkou podporou veřejnosti se knihovna stala multiplatformní. Také tato knihovna byla původně vyvíjena pro GLES 1.x rozhraní s pozdějším přechodem k GLES 2.0, především kvůli výkonnosti a možnostem tohoto API. LibGDX umožňuje pracovat s oběma GLES rozhraními a zároveň emuluje metodu přímého render módu pro GLES 2.0. Později přibyla podpora pro vykreslování 3D grafiky a použití perspektivní kamery. Knihovna dále umožňuje tvorbu texture atlasů, načítání tile map, vytváření 2D UI na bázi scene-graphu, billboarding, načítání Wavefront OBJ a MD5 formátů obsahujících popis mesh objektů. Zahrnuty jsou i operace s vektory a maticemi. Pro simulaci fyziky 2D světa umožňuje používat Box2D engin skrze JNI wrapper. Tento wrapper je taktéž použit pro 3D Bullet Physics knihovnu umožňující simulovat kolize a fyzikální vlastnosti reálného světa. Jedná se o jednu z nejpopulárnějších knihoven, jež má obrovskou podporu u široké veřejnosti i vývojářů, a proto je neustále vyvíjena. Poskytuje tak uživateli široké možnosti pro tvorbu aplikací. LibGDX se skládá z mnoha samostatných knihoven řešících dílčí problémy. Také obecnost knihovny z důvodu multiplatformosti je velká a může tak zpomalit rychlost vývoje nové aplikace, avšak port aplikace do iOS, Android či HTML5 je naopak velmi rychlý.

2.1.3 jPCT-AE

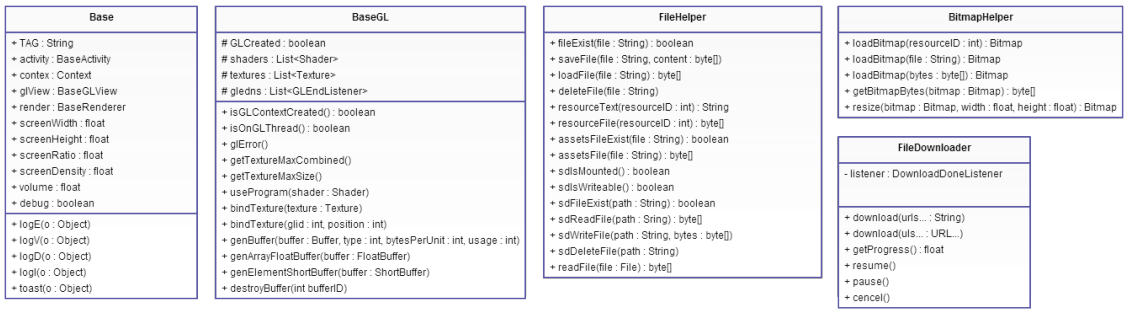
Knihovna jPCT byla původně vyvinuta v jazyce Java, především pro 3D desktopové aplikace. Jedná se o odlehčenou verzi tohoto systému určenou pro Android zařízení. Knihovna pracuje jak s OpenGL ES 1.x tak i s verzí 2.0. Je zde zahrnuta podpora skeletálních animací a načítání bežných typů souborů jako je OBJ, MD2, ASC a serializované soubory. Systém dále nabízí několik předdefinovaných shaderů a základní sadu primitivních objektů. Zahrnuta je i podpora osvětlení scény a texturování či komprese textur. Detekce kolizí kruhových a eliptických objektů. Dále je zde možnost automatického generování normál a tangent vektorů. Knihovna disponuje kvalitní dokumentací a je stále podporována. Popularitu si u veřejnosti našla jak v desktopové, tak i v android verzi.

2.2 Vyvíjená knihovna

Hlavní motivací pro vývoj nové knihovny byla vize vytvořit kvalitní, ale zároveň jednoduchý nástroj pro rychlý a snadný postup při tvorbě Adroid aplikací. Převážně se knihovna snaží odlišit v né příliš velké obecnosti a složitosti při nastavení jednotlivých struktur a celého systému. Nabízí tedy jak přímé, konkrétní struktury, tak i obecné postupy pro rychlé přizpůsobení obsahu.

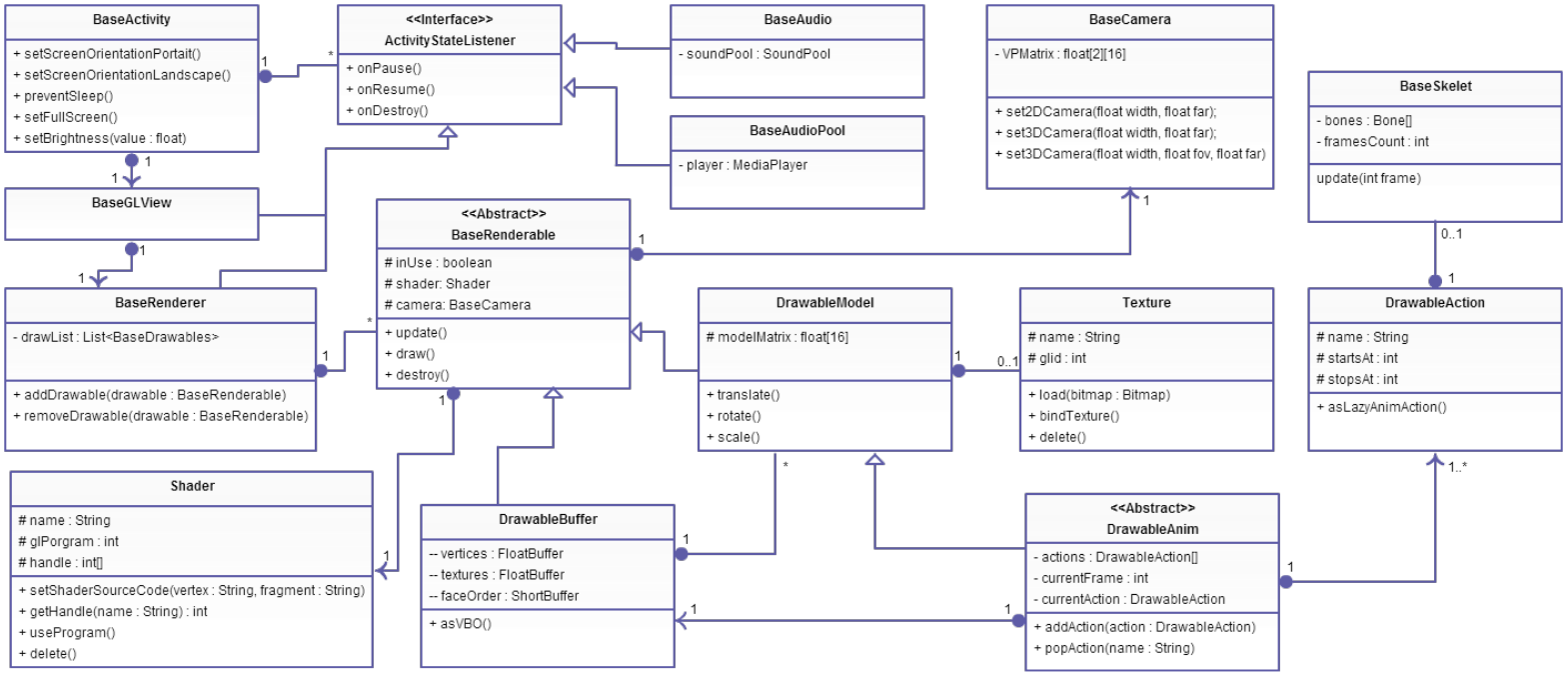
2.2.1 Struktura knihovny

Knihovna se postupem vývoje stala komplexním a rozsáhlým systémem pro vývoj aplikací. Avšak základní struktura je centralizována do několika tříd, s nimiž je možné plně využít sílu systému. Také je zde několik pomocných tříd, které uchovávají reference či pomáhají při tvorbě struktur.



Obrázek 1: Struktura vybraných pomocných tříd.

Diagram na následující stránce zobrazuje nosné třídy a funkce knihovny.



Obrázek 2: Class diagram nosných tříd a funkcí knihovny.

2.3 Prostředí knihovny

Projekt je vytvořen jako Android Java knihovna, kterou lze použít ve formě předkompilovaného $.jar$ souboru nebo přímo celý balíček knihovny s možností editace zdrojových souborů. Struktura balíčku knihovny je specifikována jako $Gradle$ prostředí, a proto je možné projekt importovat do mnoha různých vývojových nástrojů. Minimální verze Android SDK je stanovena na verzi 9 (2.3 API). GLES 2.0 bylo implementováno již ve verzi 8, avšak v této verzi SDK chyběly funkce nezbytné pro implementaci VBO.

2.4 Prostředí aplikace

Aplikace je založena na Android aktivitě, jež vytváří kontext pro spojení a komunikaci se systémem, spravování zdrojů balíčku, sdílení voleb, přístup do databáze či informace o samotné aplikaci. Jakým způsobem bude aplikace prezentována uživateli, lze specifikovat pomocí AndroidManifest.xml souboru, který je povinný pro každou aplikaci a obsahuje informace jako:

- jméno balíčku a verze
- minimální verzi SDK a verzi, na jakém SDK byla aplikace vyvíjena
- k jakým informacím, modulům bude mít aplikace přístup (např. Internet, SD karta, GPS, Wifi, Prodej in-app produktů)
- umístění ikony a názvu aplikace
- vlastnosti a orientace displeje
- hlavní a vedlejší aktivity
- a mnohé další nastavení

Takto specifikované vlastnosti jsou nastaveny již během závádění aplikace. Mnohé z nich však lze ovlivnit i po spuštění aplikace. Jedná se především o operace spojené s nastavením displeje zařízení. Tyto operace je potřeba provádět pod správným kontextem k dané aktivitě a aplikaci, proto byly jednotlivé metody implementovány ve třídě $BaseActivity$.

Automatické rotaci displeje při překlopení zařízení lze předejít nastavením fixní orientace.

```
public void setScreenOrientationPortrait(){
  setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_PORTRAIT);
}

public void setScreenOrientationLandscape(){
  setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);
}
```

Výpis 1: Nastavení orientace displeje.

Při tvorbě grafických aplikací je většinou požadováno, aby byly přes celou obrazovku a informační panel v horní části displeje či název aplikace byly skryty.

```
public void setFullScreen(){
  requestWindowFeature(Window.FEATURE_NO_TITLE);
  getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
      WindowManager.LayoutParams.FLAG_FULLSCREEN);
}
```

Výpis 2: Nastavení plochy přes celý displej.

Android dále umožňuje nastavení jasu obrazovky přímo v aplikaci tak, aby se uživatel nemusel vracet až do nastavení celého systému.

```
public void setBrightness(float reqScreenBrightness){
  final Window window = getWindow();
  final WindowManager.LayoutParams windowLayoutParams = window.getAttributes();
  windowLayoutParams.screenBrightness = reqScreenBrightness;
  window.setAttributes(windowLayoutParams);
}
```

Výpis 3: Změna intenzity jasu obrazovky.

Při delší neaktivitě uživatele zařízení automaticky ztmaví, případně uplně vypne obrazovku. K této neaktivitě může dojít například při přehrávání videa či určité sekvence, kdy není od uživatele vyžadována žádná akce k ovlivnění běhu programu.

```
public void preventSleep(){
  getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
}
```

Výpis 4: Předejítí vypnutí nebo ztmavení obrazovky.

Důležitou součástí je taky zjištění jakou verzi GLES aplikace používá. V systému je tato informace uložena jako hexadecimální číslo. Pokud by byla verze nižší než 0x200000 (OpenGL ES 2.0), dojde k automatickému ukončení již při zavádění aplikace.

```
final ConfigurationInfo configurationInfo = ((ActivityManager) getSystemService(Context.
    ACTIVITY_SERVICE)).getDeviceConfigurationInfo();
if (configurationInfo.reqGlEsVersion < 0x200000){
  return;
}
```

Výpis 5: Kontrola podporované verze OpenGL ES.

2.4.1 Chybová hlášení

Nedílnou součástí při vývoji nové aplikace je možnost odchytávání chyb či výpisů jak ve vlastním kódu, tak i chybová hlášení z OpenGL. Zároveň by však tato hlášení neměla být publikována ve finální verzi aplikace, aby neobtěžovala uživatele či zpomalovala chod aplikace. Proto je vhodné obalit jednotlivé 'logy' možností debug módu. Debug mód taktéž aktivuje kontrolu a výpis chyb při jednotlivých krocích zavádění EGL a GL kontextu. K tomuto účelu slouží pomocná třída $Base$.

```
Base.debug = true;

public static void logE(Object o){
    if (debug) Log.e(TAG, o.toString());
}

public static void logE(String tag, Object o){
    if (debug) Log.e(tag, o.toString());
}
```

Výpis 6: Využití debug módu

2.5 Prostředí OpenGL ES

Samotné OpenGL je sada příkazů ovlivňující operace grafického hardwaru. Pokud zařízení poskytuje pouze adresovatelný framebuffer, OpenGL je většinou téměř zcela implementováno na CPU. Častěji však grafický hardware poskytuje různé stupně grafické akcelerace, od vykreslování dvou-dimenzionálních linek a polygonů až po sofistikované procesy schopné transformovat geometrii složitých těles. Nejdříve je nutné otevřít okno do framebufferu, ve kterém bude program vykreslován. Po přidělení GL kontextu a jeho přičlenění k oknu je možné plně využívat příkazy OpenGL systému. Samotná správa displeje a vytvoření GL kontextu je již implementována v Android API pomocí EGL rozhraní. Také dvojtý buffering je již v implementaci zahrnut. Android však nabízí cestu k ovlivnění vlastností, se kterými bude GL kontext vytvořen pro EGL displej. Jde především o možnost nastavit verzi OpenGL ES, hloubku jednotlivých barev, alfa kanálu, velikost depth a stencil bufferu, možnost využít multisampling a další nastavení. Jednotlivé hodnoty lze ovlivnit statickou cestou ve třídě $BaseEGLConfig$.

```
public EGLConfig chooseConfig(EGL10 egl, EGLDisplay display) {
        int [] numConfig = new int[1];

        int [] configSpec = {
                EGL10.EGL_RED_SIZE, 5,
                EGL10.EGL_GREEN_SIZE, 6,
                EGL10.EGL_BLUE_SIZE, 5,
                EGL10.EGL_DEPTH_SIZE, 16,
                EGL10.EGL_RENDERABLE_TYPE, 4, // EGL OpenGL ES 2.0 bit
                EGL10.EGL_SAMPLE_BUFFERS, 1, // enable sampling
                EGL10.EGL_SAMPLES, 2, // number of samples
                EGL10.EGL_NONE      // end of specification
        };

        // check if specified config is correct on given device
        egl.eglChooseConfig(display, configSpec, null, 0, numConfig);
        .
        .
}
```

Výpis 7: Ukázka vytvoření EGL specifikace umožnujícího multisampling

Problém nastává při použití této specifikace u zařízení s grafickým čipem Nvidia Tegra, kde je nutné použít jiné bitové příznaky pro antialiasing. Dle specifikace výrobce je zapotřebí nahradit statickou proměnnou EGL SAMPLE BUFFERS a EGL SAMPLES. Avšak nově vytvořený 'Coverage' anti-aliasing přináší jiný algoritmus do OpenGL, který dramaticky zvyšuje kvalitu multisamplingu bez ovlivnění robustnosti či značného nárustu požadavků na paměť.

```
final int EGL_COVERAGE_SAMPLE_BUFFERS_NV = 0x30E0;
final int EGL_COVERAGE_SAMPLES_NV = 0x30E1;

int [] configSpec = {
        EGL10.EGL_RED_SIZE, 5,
        EGL10.EGL_GREEN_SIZE, 6,
        EGL10.EGL_BLUE_SIZE, 5,
        EGL10.EGL_DEPTH_SIZE, 16,
        EGL10.EGL_RENDERABLE_TYPE, 4,
        EGL_COVERAGE_SAMPLE_BUFFERS_NV, 1,
        EGL_COVERAGE_SAMPLES_NV, 5,
        EGL10.EGL_NONE
};
```

Výpis 8: EGL specifikace pro zařízení s čipem Nvidia Tegra

Pokud by se nepodařilo ani jedenu z konfigurací vytvořit, nebude nejspíše zařízení podporovat multisampling nebo zvolený formát. V tomto případě je nutné odpustit od požadované specifikace a vytvořit korektní speficikaci, případně přenechat tuto implementaci na samotnému Android systému. Knihovna však automaticky volí EGL konfiguraci pro multisampling dle hardwaru daného zařízení (vytvoří běžný/Coverage antialiasing, případně konfig bez multisamplingu).

2.5.1 OpenGL chybová hlášení

OpenGL pracuje na principu client-server, a proto samovolně jednotlivé chyby neposílá do konzole zařízení. Tyto chyby je nutné ručně odchytávat při jednotlivých operacích, protože OpenGL udržuje v paměti pouze poslední známou chybu. Chybová hlášení jsou v OpenGL reprezentována číselným kódem, pomocí kterého lze zjistit možnou chybu.

```
public static void glError(String tag){

    int errCode = GLES20.glGetError();
    if (errCode != GLES20.GL_NO_ERROR){
        String err = GLU.gluErrorString(errCode);
        Base.logE(tag, errCode +" "+ err);
    }
}
```

Výpis 9: Zobrazení OpenGL chybového hlášení

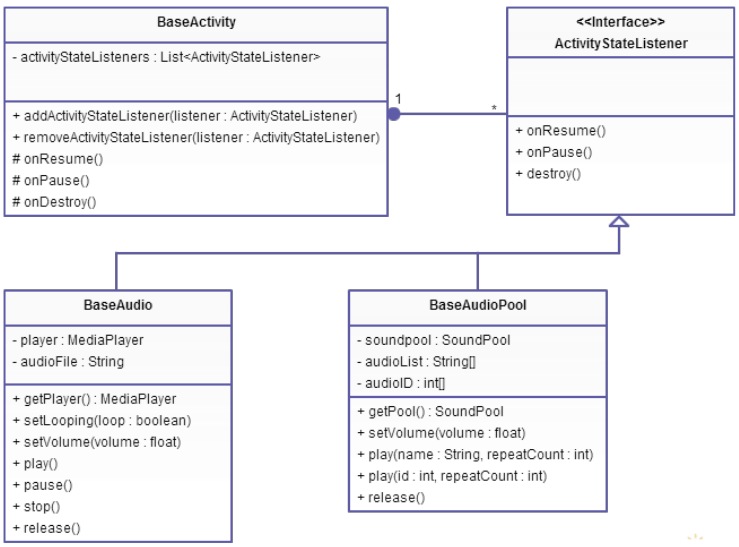
2.6 Práce s audio soubory

Android umožňuje pracovat se všemi běžnými typy audio soborů, jedná se především o soubory typu $.wav$, $.mp3$, $.ogg$, $.flac$, $.3gp$, $.mp4$ a mnoho dalších. Před začátkem přehrávání je vhodné zjistit uživatelské nastavení hlasitosti zařízení. Na tuto hodnotu hlasitosti bude později nastavena audio pasáž.

```
AudioManager audioManager = (AudioManager) context.getSystemService(Context.
    AUDIO_SERVICE);
float actualVolume = (float) audioManager.getStreamVolume(AudioManager.
    STREAM_MUSIC);
float maxVolume = (float) audioManager.getStreamMaxVolume(AudioManager.
    STREAM_MUSIC);
volume = actualVolume / maxVolume;
```

Výpis 10: Zjištění nastavené hodnoty hlasitosti na zařízení.

Všeobecně při tvorbě programů lze audio rozdělit na dvě základní podmnožiny. Hudba na pozadí aplikace a krátkometrážní zvukové efekty při určité akci. Obě tyto možnosti jsou v Androidu podporovány a logicky odděleny. Třída $BaseAudio$ slouží pro přehrávání hudby na pozadí, k čemuž využívá služeb Android třídy $MediaPlayer$. Zdrojové soubory lze tradičně načíst ze všech možných zdrojů zařízení. O sdružování a přehrávání zvukových efektů se stará třída $BaseAudioPool$, která implementuje Android třídu $SoundPool$. Pomocí této třídy je možné přehrát i několik zvuků zároveň. Zdrojové soubory lze načíst buď přiřazením složky, ze které budou načteny veškeré podporované audio formáty, nebo zadáním cest k jednotlivým souborům.



Obrázek 3: Class diagram popisující třídy pro práci s audio soubory.

Při vytváření nových instancí těchto tříd dochází k automatické registraci posluchačů

stavu aktivity. Tudíž není nutná žádná další akce ze strany uživatele knihovny pro správné nakládání se zdroji hudby. Zdroje jsou pozastaveny nebo uvolněny při opuštění aplikace, případně znovu vytvořeny a přehrávání je navázano při návratu.
Pro kontrolu zda je audio formát podporován byla vytvořena statická metoda $AudioFormat.isSupported(String format)$, která jednoduše kontroluje konconvku souboru, případně i verzi Android API, od kterého je formát podporován.

3 Textury

Textury představují bitmapové nebo rastrové obrázky, které jsou nahrány do OpenGL a poté aplikovány (mapovány) na plochu obrazce nebo polygonu.

3.1 Bitmapy

Android nabízí metody pomocí kterých lze bitmap obrázky načítat, ukládat či měnit rozměry. Systém umožňuje pracovat se všemi běžnými typy souborů (JPEG, PNG, BMP, GIF). Defaultně Android mění měřítko načítaných bitmap v závislosti na rozlišení displeje a podle případného typu umístění souboru v *resources* složce. Při použití bitmap pro GL texturování však tato praktika není vhodná, a proto je vždy automatická změna měřítka zakázána. Knihovna bitmapy vytváří až ve chvíli kdy je příkaz vykonán na GL vlákně, tak aby bylo možné okamžitě bitmapu recyklovat a tím předejít problémům s nedostatkem paměti.

```
final BitmapFactory.Options options = new BitmapFactory.Options();
options.inScaled = false;

//load bitmap from resources file
public static Bitmap loadBitmap(int resourceId){

        return BitmapFactory.decodeResource(Base.resources, resourceId, options);
}

//load bitmap from assets file
public static Bitmap loadBitmap(String file){

        Bitmap bitmap = null;

        try {
            InputStream is = Base.assets.open(file, AssetManager.ACCESS_STREAMING);
            bitmap = BitmapFactory.decodeStream(is, null, options);
            .
            .
            .

        return bitmap;
}

//create bitmap from array of bytes (SD Card, URL, FTP)
public static Bitmap loadBitmap(byte[] bytes){

        return BitmapFactory.decodeStream(new ByteArrayInputStream(bytes), null, options);
}
```

Výpis 11: Načtení bitmapy z různých zdrojů.

V případě použití bitmap s velkým rozlišením na menší obrazovce, než pro jakou byli původně vytvořeny, lze změnit jejich velikost tak, aby se v GL paměti nemusely uchovávat zbytečně robustní data.

```
public static Bitmap resize(Bitmap bitmap, float newWidth, float newHeight) {

    int width = bitmap.getWidth();
    int height = bitmap.getHeight();

    float scaleWidth = newWidth / (float)width;
    float scaleHeight = newHeight / (float)height;

    Matrix matrix = new Matrix();
    matrix.postScale(scaleWidth, scaleHeight);

    Bitmap resizedBitmap = Bitmap.createBitmap(bitmap, 0, 0, width, height, matrix, false);
    bitmap.recycle();

    return resizedBitmap;
}
```

Výpis 12: Změna velikosti bitmapy

Po finálním vytvoření textury a načtení do OpenGL je originální bitmapa vždy recyklována, což je důležitý krok pro uvolnění paměti. Bitmapy obsahují data, které přetrvávají v nativní paměti a GC může trvat více cyklů než tyto data původní bitmapy vymaže, pokud není recyklována explicitně. To znamená, že může dojít k vyčerpání přidělené paměti při neprovedení této operace, i když na bitmapu nebude dále držena žádná reference.

3.2 Prerekvizity

Před samotným generováním textur z bitmap je důležité znát možnosti a omezení zařízení. Tyto vlastnosti lze získat pomocí statických metod třídy $BaseGL$.
Maximální velikost textury je typicky 2048 * 2048 pixelů, ale na určitých zařízení může být tato hodnota jiná.

```
public static int getTextureMaxSize(){

        int [] maxSize = new int[1];
        GLES20.glGetIntegerv(GLES20.GL_MAX_TEXTURE_SIZE, maxSize, 0);
        return maxSize[0];
}
```

Výpis 13: Získání maximální velikosti GL textury pro dané zařízení

V OpenGL lze používat/kombinovat několik textur zároveň(např. pro bump mapping). Na většině zařízení lze kombinovat až 8+ textur, jiné zařízení však můžou podporovat méně textur.

```
public static int getTextureMaxCount(){

        int [] maxCount = new int[1];
        GLES20.glGetIntegerv(GLES20.GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS,
              maxCount, 0);
        return maxCount[0];
}
```

Výpis 14: Získání maximálho počtu kombinací textur

3.3 Vytvoření textururovací jednotky

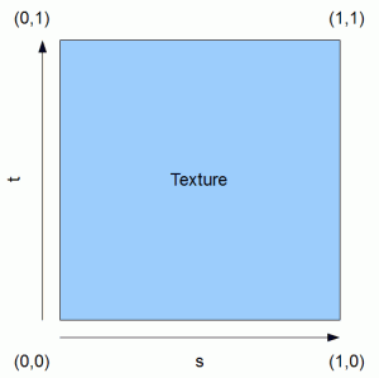
Textury z bitmap obrázků je možné vytvářet pouze pomocí GL kontextu, tedy pouze ve vlákně ve kterém OpenGL pracuje. Veškeré problémy spojené s vytvářením textur ve správném vlákně řeší třída $Texture$. Třída vždy kontroluje v jakém vlákně se uživatel pokouší texturu vytvořit a případně operace spojené s generováním přepošle na GL vlákno.

```
final int [] textureHandle = new int[1];
GLES20.glGenTextures(1, textureHandle, 0);

GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, textureHandle[0]);
```

Výpis 15: Generování texturovací jednotky a nastavení na pozici aktuální OpenGL textury.

3.4 Vlastnosti textury

V OpenGL mají textury specifický koordinační systém s horizontální osou S a vertikalní osou T. Jeden bod se nazývá $texel$.



Obrázek 4: OpenGL koordinační systém textur

Zdroj: http://www.learnopengles.com

To jak budou UV mapovací souřadnice reprezentovány na textuře lze ovlivnit pomocí nastavení parametru zalomení na jednotlivých osách. Pro každou osu lze nastavit jednu ze dvou vlastností:

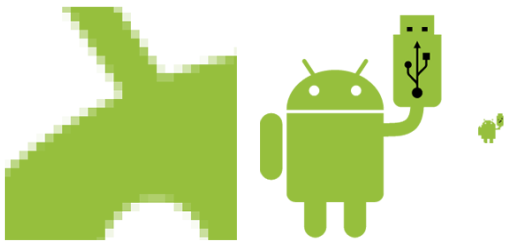
- CLAMP - Uzavření texturovacích souřadnic do oblasti [0, 1].
- REPEAT - Systém ignoruje celočíselnou část souřadnice, přičemž používá pouze druhou část. To má za následek opakující se vzor. Texturovací souřadnice s hodnotou 2.5 způsobí, že textura bude 2.5x přeložena a poté renderována.

```
GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_WRAP_S,
    GLES20.GL_CLAMP_TO_EDGE);

GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_WRAP_T,
    GLES20.GL_CLAMP_TO_EDGE);
```

Výpis 16: Nastavení CLAMP zalomení textury.

Při vykreslování scény v OpenGL jsou texturované objekty vykreslovány v různých vzdálenostech od bodu pohledu. Textury jsou aplikovány na trojúhelníky a vykresleny na obrazovce, tedy tyto textury můžou být vykresleny v různé velikostech a orientaci. Před samotným vložením bitmapy do texturovací jednotky je tedy vhodné nastavit filtry a parametry pro danou texturu. Nastavení filtrů textury ovlivní jak bude zobrazen texel na pixel obrazovky. Rozlišují se tři případy:

- 1 texel mapy na více než 1 pixel - Magnificent filter.
- 1 texel mapy odpovídá 1 pixelu - Žádný filter není aplikován.
- 1 texel mapy na méně než 1 pixel - Minificent filter.



Obrázek 5: Magnificentní, normální a minificentní rozměr.

Pro magnificent filter lze nastavit jednu ze dvou vlastností:

- NEAREST - Určuje hodnotu prvku textury, která je nejbližší k pixelu jež bude texturován. Ostré přechody.
- LINEAR - Vrací průměr čtyř nejbližších texelů, které jsou nejblíže k pixelu. Hladký přechod pomocí bilineární interpolace.

Pro minificent filter lze nastavit jednu z následujících vlastností:

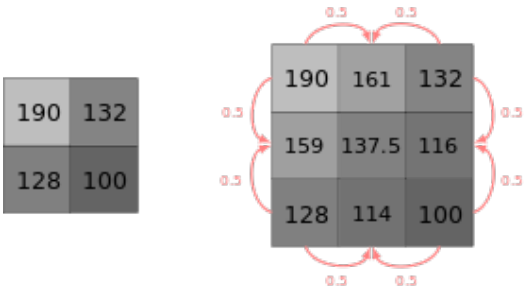
- NEAREST a LINEAR - Tyto filtry mají stejné vlastnosti jak pro magnificent tak pro minificent filtrování.
- NEAREST MIPMAP NEAREST - Je vybrána mipmapa, která je velikostně nejlépe vhodná pro daný pixel a poté je aplikován NEAREST filter.
- LINEAR MIPMAP NEAREST - Definice je obdobná s předchozí možností. V druhé části filtrace je však aplikován LINEAR filter. V tomto případě je na vybranou mipmapu aplikována bilineární interpolace.
- NEAREST MIPMAP LINEAR - Jsou vybrány dvě mipmapy, které nejvíce odpovídají velikosti pixelu jež bude texturován a poté je na každou z nich aplikován NEAREST filter. Finální hodnota je určena průměrem hodnot z obou mipmap.
- LINEAR MIPMAP LINEAR - Definice je obdobná s předchozí možností. V druhé části filtrace je však aplikován LINEAR filter. Využití trilineární interpolace.

```
GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_MIN_FILTER,
    GLES20.GL_LINEAR_MIPMAP_NEAREST);

GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D, GLES20.GL_TEXTURE_MAG_FILTER,
    GLES20.GL_LINEAR);
```

Výpis 17: Nastavení LINEAR MIPMAP NEAREST a LINEAR filtrů

3.4.1 Bilineární interpolace

Kvalita zobrazení může být docela dramaticky ovlivněna použitím bilineární interpolace. Místo přiřazení hodnot skupiny pixelů na stejné nejbližší texel hodnoty, lze tyto hodnoty interpolovat mezi sousedními čtyřmi texely. Každý pixel tím bude vyhlazen.



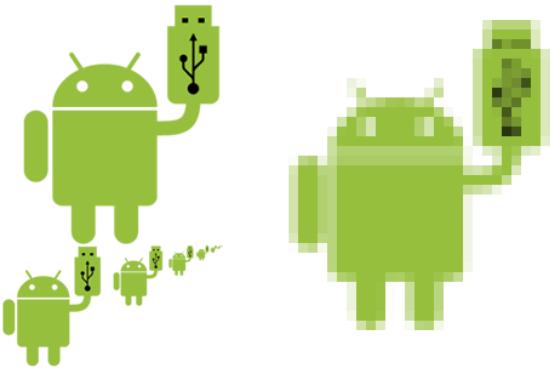
Obrázek 6: Bilineární interpolace mezi sousedními texely

Zdroj: http://www.wikipedia.com

Obrázek 7: Výsledek magnificent a minificent bilineární interpolace.

3.4.2 Mipmapy

Pokud je objekt přesunut daleko, je nutné texturu přefiltrovat, aby mohla být aplikována na menší objekt. Problémem při filtrování textury na menší rozměry může být následné blikání či kmitání. Řešením problému je *mipmapping*. Jedná se o proces při kterém je vygenerováno několik textur s menším rozlišením. Mipmapping dovoluje OpenGL použít odpovídající úroveň detailů textury při vykreslování, nežli smrštěnou verzi originální textury. Každá úroveň mipmapy je vytvořena v poloviční rozměrech předchozí mipmapy a poté je aplikován filtr.



Obrázek 8: Generování mipmap. Bilineární interpolace s použitím mipmap.

3.4.3 Trilineární interpolace

Při použití mipmap s bilineárním filtrem, někdy může být viděn znatelný skok nebo linie na zobrazené scéně, kde OpenGL přeplo mezi různými úrověmi mipmap textury. Trilineární filtrování řeší tento problém dalším interpolováním mezi různými úrovněmi mipmap, tedy k získání finálního pixelu bude interpolováno 8 texelů.

3.5 Přiřazení bitmapy textuře

Android nabízí velmi užitečnou utilitu jak nahrát bitmapu přímo do OpenGL. Touto cestou lze vytvářet textury ze všech typů souborů jež Android podporuje jako bitmapy. Po načtení dat do OpenGL by se vždy měla bitmapa recyklovat, aby nedošlo k přeplnění paměti. Bitmap data jsou nahrány do právě aktivní OpenGL textury.

```
GLUtils.texImage2D(GLES20.GL_TEXTURE_2D, 0, bitmap, 0);

bitmap.recycle();
```

Výpis 18: Načtení bitmapy do OpenGL s následnou recyklací.

Ne vždy je vhodné neustále vytvářet nové textury a tím přehlcovat paměť. Případně texturu smazat a vygenerovat novou s jiným ID. Místo toho lze pouze změnit bitmapu na pozici již vytvořené OpenGL textury.

3.6 Smazání textury

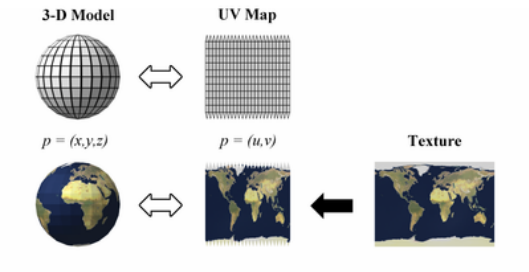
Nezbytnou součástí práce s texturami je jejich mazání z OpenGL a tím uvolnění paměti. Je důležité mazat textury, jak při ukončení aplikace, tak i v případě, že textura nebude dále zapotřebí.

```
public static void deleteTexture(int textureID){

        GLES20.glDeleteTextures(1, new int[]{textureID}, 0);
}

public static void deleteTextures(int ... textureIDs){

        if (textureIDs != null) {
            GLES20.glDeleteTextures(textureIDs.length, textureIDs, 0);
        }
}
```

Výpis 19: Smazání textury / pole textur

3.7 UV Mapování

Jedná se o proces kdy je 2D obrázek reprezentován na 3D model. Písmena U a V označují osy 2D textury. UV texturování umožňuje polygonu, který tvoří 3D objekt, být zobrazen s barvami textury. UV souřadnice jsou aplikovány na plošky, ne na vrcholy. To znamená, že vertex shader může mít různé UV souřadnice pro každou z ploch, tedy i třilehlé plochy můžou být rozděleny a umístěny na různé pozice texturové mapy. Proces UV mapovaní zahrnuje přiřazení pixelů obrázku na plošku mapovaného polygonu. Běžně se jedná o kopírování trojúhelníkového tvaru obrázku mapy a vložení ho na trojúhelníkovou plošku objektu.



Obrázek 9: Postup UV mapování textury na 3D model.

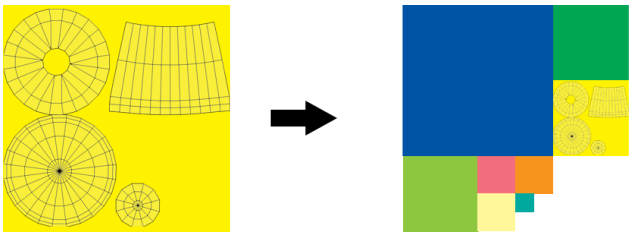
Zdroj: http://www.wikipedia.com

3.8 Atlas textur

Jedná se o jeden velký oprázek který obsahuje několik menších pod-obrázků. Z hlediska paměti a časové náročnosti zavádění každé textury do OpenGL zvlášť, je vhodnější vytvořit jednu velkou texturu. Samostatné sub-textury lze vykreslovat modifikací UV souřadnic textury. Ne vždy je možné mít dopředu vygenerované atlasy textur, proto byla vytvořena třída $TextureAtlas$ pomocí které lze vytvářet atlasy dynamicky. Při každém přidání bitmapy do atlasu je kontrolována celková předpokládaná velikost, která nesmí překročit omezení na celkovou velikost textury na daném zařízení. Bitmapy jsou seřazeny podle velikosti a poté pomocí algoritmu umísťovány do atlasu, tak aby výsledná bitmapa byla ve tvaru čtverce. Souřadnice jednotlivých sub-textur jsou generovány společně s atlasem.

3.8.1 Informace o textuře

Pomocí třídy $TextureInfo$ lze získat informace potřebné pro UV maping textury na objekt, které se po přidání bitmapy do atlasu změnily. Samozřejmostí je výpočet texturovacích souřadnic pro předem vygenerované atlasy, zde se informace o velikostech udávají v pixelech nebo počtem řádků a sloupců. Dále jsou zde zahrnuty metody řešící texturovací sub-souřadnice pro různé typy objektů (kruh, obdelník), nebo samostatně generovatelné informace pro různě velké sub-textury.



Obrázek 10: Textura s UV souřadnicemi v atlasu textur.

4 Shadery

Používání shaderů je nedílnou součástí pro vykreslování objektů pomocí OpenGL ES 2.0+. Shadery používají programovací jazyk $GLSL$ a veškeré operace provádějí přímo na grafické kartě. Stejně jako u textur, lze nové shadery vytvářet pouze pomocí GL kontextu, tedy pouze ve vlákně ve kterém OpenGL pracuje. Veškeré problémy spojené s vytvářením textur řeší třída $Shader$. V OpenGL ES rozlišujeme dva typy shaderů:

- Vertex shader: Jedná se o program, který se provede na každém vrcholu geometrie objektu. Mezi nejčastější operace patří transfromace vrcholu v prostoru.
- Fragment shader: Program je prováděn na každém pixelu rasterizované scény, vytváří tedy již 2D obraz. Mezi nejčastější operace patří aplikace textur, stínování či jiné modifikace výsledné barvy pixelu.

Kombinací vertex a fragment shaderu vzniká shaderovací program.

Deklarace proměných, které zaručují spojení mezi shadery, OpenGL a aplikací:

- uniform: Proměná zůstává během zpracování objektu stejná. Typicky matice.
- attribute: Jednotlivá data objektu. Pozice vrcholů, texturovací souřadnice, barva.
- varying: Atribut pro komunikaci mezi vertex a fragment shaderem.

Při kompilaci programu pomocí třídy $Shader$ je nutné dodržovat zavedenou $uniform$ a $attribute$ konveci u názvu proměných. Při nedodržení této konvence nebudou automaticky generovány ukazatele na jednotlivé proměné, i když shaderovací program bude úspěšně vytvořen.

- uniform proměná musí začínat písmenem u.
- attribute proměná musí začínat písmenem a.

V případě nedodržení této konvence nebudou automaticky svázány ukazatele, které jsou potřebné pro komunikaci mezi shadery, OpenGL a aplikací.

4.1 Kompilace shaderu

O samotnou kompilaci shaderů se stará třída $ShaderHelper$. Zdrojové kódy jednotlivých shaderů jsou do OpenGL posílány přímo ve formátu $String$.
Následující výpis ukazuje vytvoření vertex shader jednotky a vložení zdrojového kódu s následnou kompilací a kontrolou zda nedošlo k chybě při kompilaci.

```
public static int compileShader(int shaderType, String shaderSourceCode){
    int shaderHandle = GLES20.glCreateShader(shaderType);
    GLES20.glShaderSource(shaderHandle, shaderSource);
    GLES20.glCompileShader(shaderHandle);
```

```
    final int [] compileStatus = new int[1];
    GLES20.glGetShaderiv(shaderHandle, GLES20.GL_COMPILE_STATUS, compileStatus, 0);

    if (compileStatus[0] == 0) {
        GLES20.glDeleteShader(shaderHandle);
    .
    .
    .
    return shaderHandle;
}
```

Výpis 20: Vytvoření a kompilace vertex či fragment shaderu.

Výsledný shaderovací program vždy tvoří jeden Vertex shader a jeden Fragment shader. Společně s kompilací programu dochází ke svázání GLSL atributů s OpenGL. Opět je kontrolováno korektní svázání jednotlivých shaderů v programu a jeho korektnost.

```
public static int createAndLinkProgram(int vertexShaderHandle, int fragmentShaderHandle,
    String[] attributes){
    int programHandle = GLES20.glCreateProgram();
    GLES20.glAttachShader(programHandle, vertexShaderHandle);
    GLES20.glAttachShader(programHandle, fragmentShaderHandle);

    if ( attributes != null){
        final int size = attributes.length;
        for (int i = 0; i < size; i++){
            GLES20.glBindAttribLocation(programHandle, i, attributes[i]) ;
        }
    }

    GLES20.glLinkProgram(programHandle);

    final int [] linkStatus = new int[1];
    GLES20.glGetProgramiv(programHandle, GLES20.GL_LINK_STATUS, linkStatus, 0);

    if (linkStatus[0] == 0) {
        GLES20.glDeleteProgram(programHandle);
    .
    .
    .
    return programHandle;
}
```

Výpis 21: Ukázka vytvoření shaderovacího programu.

Pokud při kompilaci jednoho z shaderů dojde k chybě, nedojde k vytvoření shaderu a chybové hlášení OpenGL se zobrazí v konzoli.
Poslední operací pro svázání shaderu s aplikací je získání ukazatelů na jednotlivé uniform a attribute proměné.

```
    int matrixHandle = GLES20.glGetUniformLocation(programHandle, "u_MVPMatrix");
    int verticeCoordHandle = GLES20.glGetAttribLocation(programHandle, "a_Position");
```

Výpis 22: Ukázka získání ukazatelů na unifrom a attribute proměné.

4.2 Předdefinované shadery

Knihovna obsahuje několik základních shaderů, které jsou automaticky kompilovány při vytvoření Rendereru. Shadery jsou určeny především pro mobilní zařízení, a proto zde bylo dbáno na jednoduchost a rychlost provádění programů.
U všech shaderů je použita uniformní proměnná $u\ MVPMatrix$ přenášející multiplikaci matic modelu, pohledu a projekce kamery. Tato matice je poté násobena pozicí vrcholu, $a\ Position$. Tím se určí konečné souřadnice pro daný vrchol, $gl\ Position$, na který bude posléze aplikována barva. Proměnné $uniform$ a $attribute$ jsou získány uživatelským vstupem a proměnné $varying$ jsou dále poslány fragment shaderu. Výslednou barvu pixelu představuje hodnota $gl\ FragColor$.

4.2.1 Barevný shader

Každému vrcholu je přiřazena právě jedna barva ve formátu RGBA.

```
uniform mat4 u_MVPMatrix;

attribute vec4 a_Position;
attribute vec4 a_Color;

varying vec4 v_Color;

void main()
{
    v_Color = a_Color;
    gl_Position = u_MVPMatrix * a_Position;
}
```

Výpis 23: Barevný vertex shader

Fragment shader pouze přiřadí získanou barvu konkrétnímu pixelu.

```
varying vec4 v_Color;

void main()
{
    gl_FragColor = v_Color;
}
```

Výpis 24: Barevný fragment shader

4.2.2 Texturovací shader

Každému vrcholu jsou přiřazeny texturovací souřadnice, které jsou posléze odeslány fragment programu pro zpracování.

```
uniform mat4 u_MVPMatrix;

attribute vec4 a_Position;
attribute vec2 a_TexCoordinate;
```

```
varying vec2 v_TexCoordinate;

void main()
{
    v_TexCoordinate = a_TexCoordinate;
    gl_Position  = u_MVPMatrix * a_Position;
}
```

Výpis 25: Texturovací vertex shader

Fragment shader využívá uniformní texturovací sampler, jež aplikuje právě aktivní texturu. Výsledná barva pixelu je určena mapováním s, t souřadnic na texturu.

```
uniform sampler2D u_Texture;

varying vec2 v_TexCoordinate;

void main()
{
    gl_FragColor = texture2D(u_Texture, v_TexCoordinate);
}
```

Výpis 26: Texturovací fragment shader

4.2.3 Bumpmap shader

Oproti obyčejnému texturovacímu shaderu zde přibyla uniformní proměná přenášející polohu světla, která je posléze normalizována a poslána fragment shaderu k dalšímu zpracování.

```
uniform mat4 u_MVPMatrix;
uniform vec3 u_Light;

attribute  vec4 a_Position;
attribute  vec2 a_TexCoordinate;

varying vec2 v_TexCoordinate;
varying vec3 v_LightPos;

void main()
{
    v_TexCoordinate = a_TexCoordinate;
    v_LightPos = normalize(u_Light);
    gl_Position  = u_MVPMatrix * a_Position;
}
```

Výpis 27: Bumpmap vertex shader

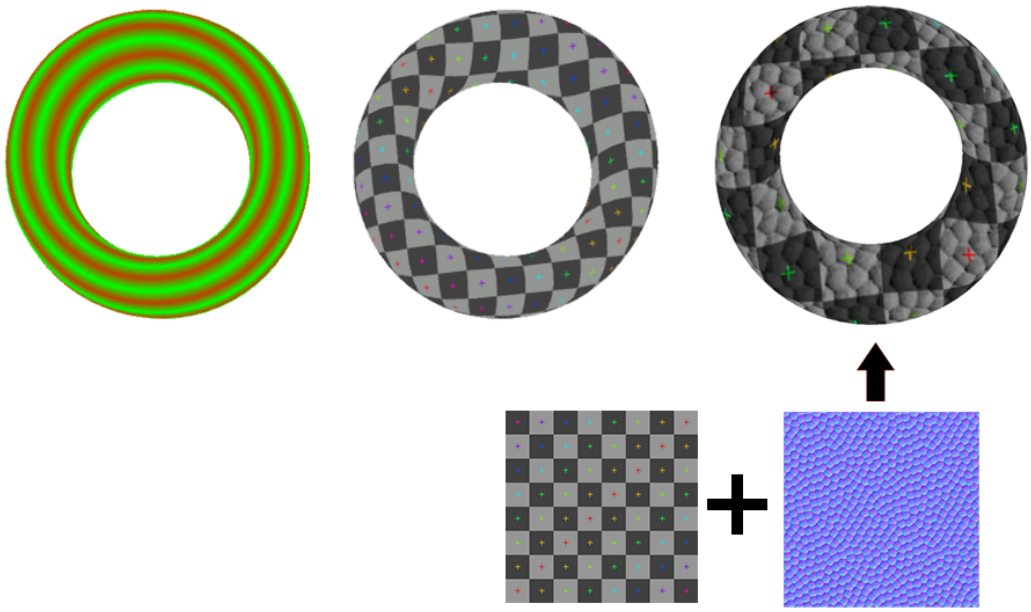
Dále je zde druhý uniformní sampler držící normálovou mapu objektu. Ukládání normál do textur neumožňuje záporný směr vektoru, proto po extrakci normály z textury je nutné převést vektor z prostoru [0, 1] do prostoru [-1, 1] tak, aby výsledná normála směřovala

libovolným směrem. Normálu je nutné poté normalizovat. Výsledný faktor modifikující ztmavení či zesvětlení barvy pixelu je určen z pozice světla a vypočtené normály. Po aplikaci faktoru na difuzní složku textury je získána výsledná barva.

```
uniform sampler2D u_Texture;
uniform sampler2D u_Normals;

varying vec2 v_TexCoordinate;
varying vec3 v_LightPos;

void main()
{
    vec3 normal = normalize(2.0 * texture2D(u_Normals, v_TexCoordinate.st).rgb - 1.0);
    float factor = max(dot(normal, v_LightPos), 0.0);
    vec4 diffuse = texture2D(u_Texture, v_TexCoordinate);
    vec3 color = factor * diffuse.rgb;
    gl_FragColor = vec4(color, diffuse.a);
}
```

Výpis 28: Bumpmap fragment shader

Uživatel knihovny není nijak vázána na použití stávajících shaderů a může si definovat vlastní, přičemž mu knihovna pomůže s jejich kompilací a svázáním attributů potřebných ke komunikaci mezi shadery, OpenGL a aplikací.
Následující ukázka zobrazuje vykreslení objektu při použití dosavadních shaderů.



Obrázek 11: Barevný, texturovací a bumpmap shader.

5 Maticové operace

Transformace objektů jsou v OpenGL řešeny pomocí matic.V těchto transformačních maticích je použit homogení souřadnicový systém, čili souřadnice jednoho bodu jsou x, y, z, w, přičemž souřadnice w je vždy nastavena na hodnotu 1. OpenGL ES pracuje se sloupcově orientovanými 4 x 4 maticemi. Matice jsou uloženy ve $float$ poly. O práci s maticemi se stará třída $BaseMatrix$, případně implementace Android třídy $Matrix$.

Pole v maticovém formátu OpenGL ES:

$$\begin{pmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{pmatrix}$$

5.1 Translace

Proměnná x, y, z označuje osu po které bude operace translace provedena.

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rozšířením této matice je translace, jež bere v potaz skutečnost, že na objekt již byli aplikovány určité transformace. Tím zde odpadá nustnost časově náročnější multiplikace původní matice a matice translace.

$$\begin{pmatrix} a_0 & a_4 & a_8 & a_{12} + x * a_0 + y * a_4 + z * a_8 \\ a_1 & a_5 & a_9 & a_{13} + x * a_1 + y * a_5 + z * a_9 \\ a_2 & a_6 & a_{10} & a_{14} + x * a_2 + y * a_6 + z * a_{10} \\ a_3 & a_7 & a_{11} & a_{15} \end{pmatrix}$$

5.2 Rotace

Před vytvořením matice rotace je nutné úhel zadaný ve stupních převést do radiánů a určit jeho sinus a cosinus hodnotu. Výsledná matice je tedy určena dle hodnot sinus, cosinus úhlu a osou ve které má být rotace provedena:

$$\begin{pmatrix} & & & 0 \\ & R & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rozšíření matice rotace umožňující vynechat multiplikaci, avšak tato implementace zaručuje rotaci pouze v jedné ose a musí být aplikována před operací změny měřítka:

$$\begin{pmatrix} & & & a_{12} \\ & R & & a_{13} \\ & & & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{pmatrix}$$

5.3 Změna měřítka

Proměné x, y, z označují osu po které bude operace změny měřítka provedena.

$$\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rozšíření matice měřítka umožňující vynechat časově náročnější multiplikaci matic:

$$\begin{pmatrix} a_0 * x & a_4 * y & a_8 * z & a_{12} \\ a_1 * x & a_5 * y & a_9 * z & a_{13} \\ a_2 * x & a_6 * y & a_{10} * z & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{pmatrix}$$

5.4 Billboarding

Obecně se jedná o techniku kdy je objekt orientován na určitý cíl, většinou kameru. Tato technika se používá především v částicových systémech, aby nedošlo k nevhodnému natočení částice ke kameře. Billboardingu lze dosáhnout pomocí prostého smazání rotací objektu, případně natočením dle rotace kamery. Dojde však i ke zrušení transformace měřítka, a proto je nutné aplikovat tuto transformaci až po zavedení billboardingu.

$$\begin{pmatrix} 1 & 0 & 0 & a_{12} \\ 0 & 1 & 0 & a_{13} \\ 0 & 0 & 1 & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{pmatrix} || \begin{pmatrix} & & & a_{12} \\ & Rcam & & a_{13} \\ & & & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{pmatrix}$$

Cilindrický billboarding, přičemž rotace na jedné z os je ponechána a druhé dvě jsou anulovány.

$$\begin{pmatrix} a_0 & 0 & 0 & a_{12} \\ a_1 & 1 & 0 & a_{13} \\ a_2 & 0 & 1 & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{pmatrix} || \begin{pmatrix} 1 & a_4 & 0 & a_{12} \\ 0 & a_5 & 0 & a_{13} \\ 0 & a_6 & 1 & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{pmatrix} || \begin{pmatrix} 1 & 0 & a_8 & a_{12} \\ 0 & 1 & a_9 & a_{13} \\ 0 & 0 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{pmatrix}$$

Při použití billboardingu na model, jehož transformace jsou řešeny způsobem kroku, je nutné matici modelu po aplikaci transformací vždy uložit do mezi-matice a až poté aplikovat billboarding. Naopak v dalším kroku je potřeba opět kopírovat původní matici modelu zpět z mezi-matice. Použití matice pro změnu měřítka je možné až po billboardingu. Poté již lze vypočítat finální MVP matici, která bude předána shaderovacímu programu zbavená rotačních transformací modelu.

```
BaseMatrix.copy(tempMatrix, modelMatrix);

BaseMatrix.translate(modelMatrix, stepX, stepY, stepZ);
BaseMatrix.rotate(modelMatrix, rotX, rotY, rotZ);

BaseMatrix.copy(modelMatrix, tempMatrix);

BaseMatrix.billboard(modelMatrix); // or billboard(modelMatrix, camera);

BaseMatrix.scale(ratioX, ratioY, ratioZ);

BaseMatrix.multiplyMM(MVPMatrix, camera.mVPMatrix, modelMatrix);
```

Výpis 29: Postup aplikace billboardingu

Takto vytvořený billbording je ve své podstatě podvod, i když velice populární a používaný způsob. Místo měnění matice modelu a nutnosti dalších operací je možné manuálně transformovat jednotlivé vrcholy objektu, podle transponované matice pohledu. Tento způsob je však efektivní pouze u objektů složených z několika vrcholů. Ve vytvořené knihovně však tato možnost není implementována, především z hlediska využití bufferingu částicového systému, kdy jsou zároveň použity jedny data pro všechny částice. V této chvíli by se stal zmiňovaný způsob billboardingu méně efektivní a nežádoucí z hlediska nemožnosti použití u složitějších polygonů.

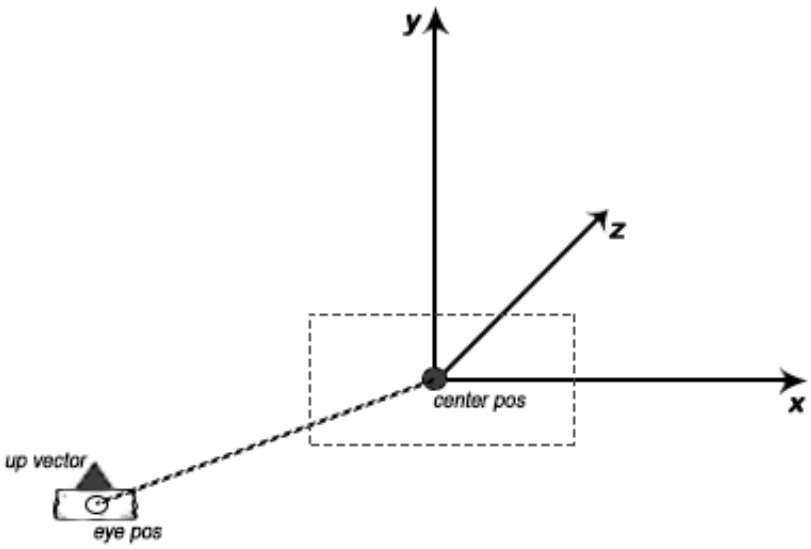
6 Kamera

Správné nastavení kamery je nedílnou součástí výsledného dojmu z pohledu na scénu. O veškerou práci s kamerovým systémem se stará třída $BaseCamera$, kde je možné využít již stávající předdefinované kamery, nebo použít vlastní nastavení. Kamerový systém se skládá z matice pohledu (view) a matice projekce (projection). Výsledná matice kamery je tvořena multiplikací pohledové a projekční matice. Jednotlivé matice jsou implementovány dle GLU speficikace. Matice jsou ve formátu 4 x 4 a uloženy pomocí float pole.

6.1 Pohled kamery

Matice pohledu kamery je definována třemi ukazateli. Umístěním ve scéně, bodem zájmu a orientací kamery. Každý z těchto parametrů obsahuje tři proměné.

- $eyeX, eyeY, eyeZ$ - specifikace pozice ve scéně
- $centerX, centerY, centerZ$ - bod zájmu kamery
- upX, upY, upZ - směr vektoru orientace



Obrázek 12: Pohled kamery.

Transformace kamery lze aplikovat přímo změnou nastavení matice pohledu, případně multiplikací transformační a pohledové matice.

6.2 Projekce

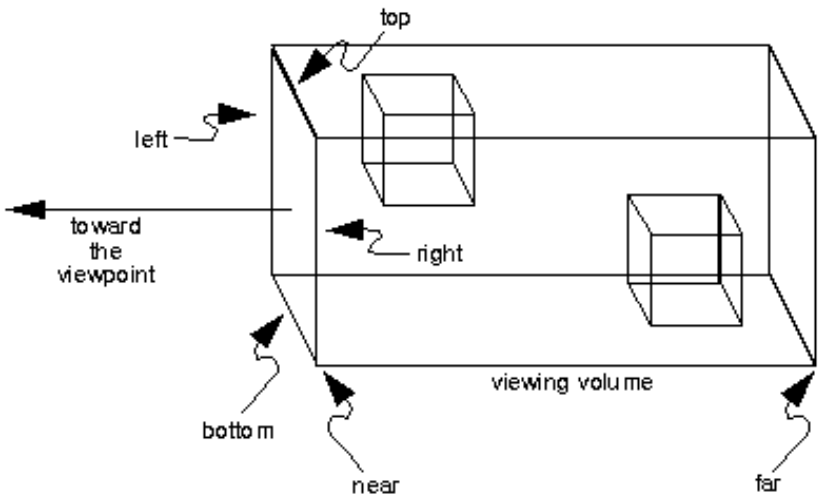
Projekční matice definuje jak bude bod transformován z 3D prostoru na 2D obrazovku. Vhodně zvolená projekce dodává scéně patřičný význam.
Projekce je vždy ohraničena šesti stěnami:

- levou, pravou, spodní, horní, bližší a vzdálenější stěnou.

Veškeré objekty které se nacházejí mezi stěnami budou zobrazeny. Standartně se volí výška scény ku poměru šířky scény nebo naopak. Při nedodržení této zásady dochází k deformaci scény a objekty nejsou zobrazovány korektně(avšak i to může být záměr projekce).

6.2.1 Ortografická projekce

Jedná se v podstatě o 2D zobrazení scény, bez ohledu na to kde se objekt na scéně nachází. Pokud jsou ve scéně umístěny dva identické objekdy vedle sebe, ale s různou vzdáleností od kamery, budou se stále jevit jako identické.
Hodnoty far a $near$ musí být kladné. Boční stěna $left$ se nesmí rovnat parametru $right$ a taktéž $bottom$ se nesmí rovnat hodnotě top.



Obrázek 13: Ortografická projekce.

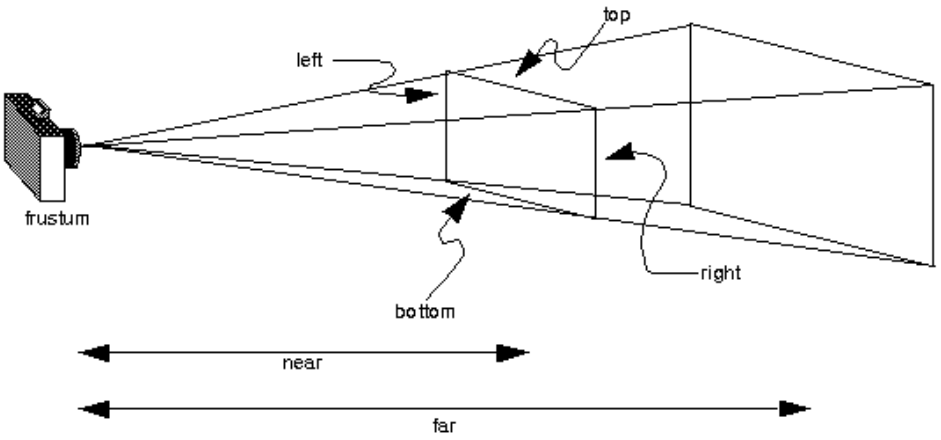
Zdroj: http://njoubert.com

6.2.2 Perspektivní projekce

Matice perspektivní projekce se snaží o realistické zobrazení 3D scény na 2D obrazovce tak, jakoby scéna ležela v reálném světě. Pokud jsou ve scéně umístěny dva identické

objekdy vedle sebe, ale s různou vzdáleností od kamery, vzdálenější objekt se bude jevit menší nežli ten bližší, tak jako v reálném světě.

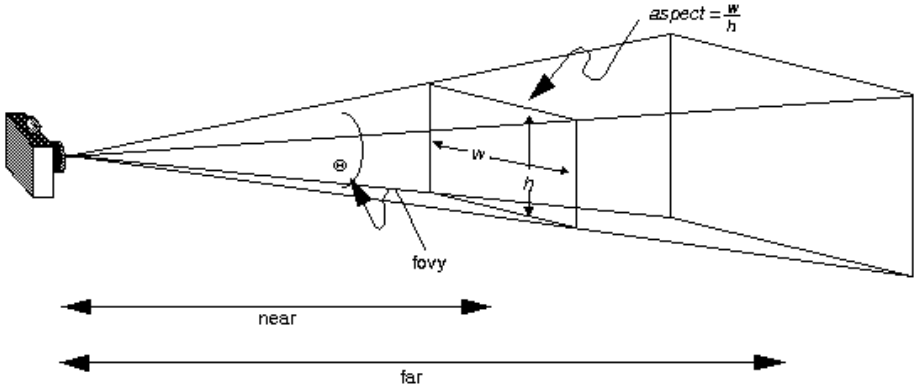
6.2.2.1 Projekce určená jehlanem Hodnoty far a $near$ musí být kladné. Boční stěna $left$ se nesmí rovnat parametru $right$ a taktéž $bottom$ se nesmí rovnat hodnotě top.



Obrázek 14: Perspektivní projekce zadaná jehlanem.

Zdroj: http://njoubert.com

6.2.2.2 Projekce určená polem pohledu Parametr $fovy$ určuje úhel pole pohledu ve stupních. Hodnoty far a $near$ musí být kladné.



Obrázek 15: Perspektivní projekce určená polem pohledu.

Zdroj: http://njoubert.com

6.3 Práce s kamerou

Knihovna umožňuje generovat kamerové systémy dle různých speficikací, především zadáním šířky zorného pole a hodnotami ohraničení bližší a vzdálenější stěny. Ostatní potřebné atributy jsou automaticky dopočítány na základě zadaných hodnot. Rozhraní samozřejmě umožňuje vytvářet vlastní specifikace jednotlivých matic. Při automatickém generování kamery je dopočítáno několik důležitých parametrů. Jedná se především o parametry typu:

- Poměr šířky a výšky zorného pole kamery vůči rozměrům displeje zařízení. Tento poměr je důležitý např. při transformaci pozice doteku obrazvky do prostředí aplikace.
- Velikost jednoho pixelu v poměru velikosti displeje a zorného pole kamery.

Výslednou kameru lze generovat pomocí statických konstruktorů nebo metod k tomu určených.

6.3.1 Transformace kamery

Kamera podporuje všechny standartní transformační operace jako je translace, rotace a zoom. Veškeré transformace jsou aplikovány do transformační matice a poté multiplikovány s pohledovou maticí. Přičemž lze využít transformační matice zmíněné v sekci 5 (Maticové operace). Pro simulaci zoom transformace je využita operace změny měřítka.

7 Mesh objekty

Polygon mesh je kolekce vrcholů, hran a ploch, které dohromady definují tvar objektu. Knihovna pracuje výhradně s polygony, jejichž plochy jsou tvořeny trojúhelníky. Za tímto účelem bylo vytvořeno, v jazyce Python, rozšíření do 3D modelovacího programu Blender, které řeší právě export 3D modelů a skeletálních akcí. Import dat do knihovny je umožněn třídou $BeoParser$.

7.1 Formát mesh data souboru

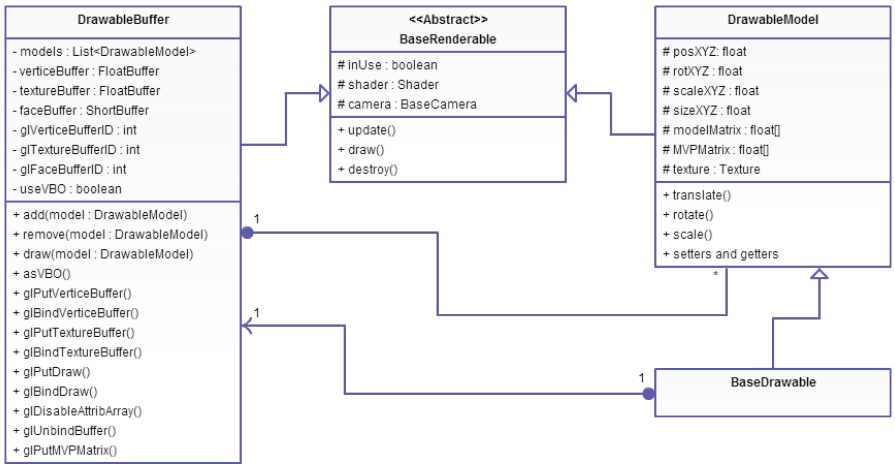
Nový formát *.beo* byl zaveden kvůli nedostačujícím kvalitám již tradičního $Wavefront(.obj)$ formátu. Načítání dat z *.obj* souboru trvá mnohem déle, kvůli nutnosti jednotlivé data přeskládávat či duplikovat vrcholy pro správné texturování. Kdežto v *.beo* souboru jsou všechna data již předpřipravena a není nutná žádná další operace ke korektnímu načtení objektu.

```
version 0.256
ob Name
cpv 3
bbox 2.0 2.0 2.0
v 60 { −1.0 1.0 1.0 1.0 1.0 1.0  .....  }
t 40 { 0.0 1.0 1.0 1.0  .....  }
f 36 { 0 1 2 1 2 3  .....  }
```

Výpis 30: Struktura mesh souboru

- $version$ - verze export skriptu.
- *ob* - název objektu.
- $bbox$ - ohraničující rozměry objektu (v ose x, y, z).
- v - vrcholy objektu (v tomto případě tři čísla tvoří jeden vrchol).
- t - texturovací souřadnice pro UV mapování (vždy dvě čísla na jeden vrchol).
- f - indexy vrcholů jednotlivých trojúhelníků polygonu (vždy tři čísla pro jednu plošku).

Při srovnání *.obj* a vyvýjeného mesh souboru bylo docíleno stejné nebo i menší velikosti souboru a zároveň i značně menšího času při načítání strukury dat. Při testech načítání různě velkých souborů bylo vždy dosaženo přibližně 6-10x rychlejších časů parsování dat. Tato vylepšení jsou znatelná převážně při načítání dat v průběhu běhu aplikace, kdy je nutné dbát na celkovou hladkost chodu.



Obrázek 16: Diagram popisující strukturu tříd pro zobrazení objektu.

7.2 Vlastnosti objektu

Veškeré informace o objektu a jaké parametry tomuto modelu náleží jsou schrnuty ve třídě $DrawableModel$. Nejdůležitější vlastností třídy je řešení aplikace transformačních matic na model a následná multiplikace matice modelu, pohledu a projekce ve výslednou $MVPMatici$, která se použije v daném shaderovacím programu. Samozřejmostí jsou proměnné držící aktuální velikost modelu, pozici, rotaci a metody odpovídající těmto atributům. Při odstranění modelu z rendereru dochází k jeho zničení a likvidaci referencí.

7.3 Buffer

Render pracuje jak se systémovými buffery tak i s VBO, a proto je nutné před zahájením komunikace s OpenGL veškerá data mesh objektu nahrát do odpovídajících typů bufferů. Pomocí třídy $DrawableBuffer$ je možné vykreslovat jeden nebo i více modelů bez nutnosti opětovného zasílání jednotlivých bufferů obsahujících mesh data do OpenGL. Jednotlivé data lze přechovávat v systémové paměti nebo přímo ve video paměti zařízení. Informace o vrcholech a texturovací souřadnice jsou ukládány do FloatBufferu, pořadí elementů je uloženo v ShortBufferu. Pomocí třídy $Buffers$ je možné vytvářet všechny typy systémových bufferů.

```
public static FloatBuffer floatBuffer(float[] floatArray) {
    FloatBuffer buffer;
    buffer = ByteBuffer.allocateDirect(floatArray.length * BYTESPERFLOAT)
            .order(ByteOrder.nativeOrder()).asFloatBuffer();
    buffer.put(floatArray).position(0);
    return buffer;
}
```

Výpis 31: Ukáza vytvoření FloatBufferu

Pomocí takto vytvořených bufferů, jež jsou uloženy v systémové paměti, lze vykreslit polygon s podporou OpenGL ES a daným GLSL shader programem. Data je však nutné při každém překreslení opět poslat do video paměti.

```
// set position of the buffer to 0, verticeCoordHandle - position shader pointer, 3 -
    coords per vertice, not-normalized, sekvencion, buffer
verticeBuffer.position(0);
GLES20.glVertexAttribPointer(verticeCoordHandle, 3, GLES20.GL_FLOAT, false, 0,
    verticeBuffer);
GLES20.glEnableVertexAttribArray(verticeCoordHandle);

textureBuffer.position(0);
GLES20.glVertexAttribPointer(textureCoordHandle, 2, GLES20.GL_FLOAT, false, 0,
    textureBuffer);
GLES20.glEnableVertexAttribArray(textureCoordHandle);

faceBuffer.position(0);
GLES20.glDrawElements(GLES20.GL_TRIANGLES, faceBuffer.capacity(), GLES20.
    GL_UNSIGNED_SHORT, faceBuffer);

GLES20.glDisableVertexAttribArray(verticeCoordHandle);
GLES20.glDisableVertexAttribArray(textureCoordHandle);
```

Výpis 32: Vykreslení polygonu ze systémové paměti.

Nejdříve je nutné nastavit pozici bufferu na počátek pole. V dalším kroku je buffer poslán přímo do video zařízení. Zde je zahrnut ukazatel z shaderovacího programu, počet čísel na jeden element, typ bufferu, zda je potřeba data normalizovat, po kolika bytech má být pole sekvencováno (0 znamená defaultní sekvencování) a odpovídající buffer. Poté se ještě dané pole aktivuje pro shader program. Tímto způsobem jsou přenášeny vrcholy polygonu, texturovací souřadnice, případně i normály či jiná pole dat.
Samotné vykreslení je specifikováno zobrazovacím módem, počtem elementů, typem a referencí na buffer, který obsahuje pořadí vrcholů/ploch. Poté jsou jednotlivé pole deaktivovány pro další použití.

7.3.1 VBO

Jedná se o možnost načíst data topologie objektu přímo do video paměti zařízení. Data tedy nemusí být v každém vykreslovacím kroku znovu posílány ze systémové paměti. VBO tím nabízí značnout výhodu ve výkonu, protože data přetrvávají ve video paměti a můžou z ní být přímo zpracovány.

```
public static int genBuffer(Buffer buffer, int bufferType, int bytesPerUnit, int usage){

    final int out[] = new int[1];

    buffer.position(0); // reset position for sure
    GLES20.glGenBuffers(1, out, 0); // generate new buffer
    GLES20.glBindBuffer(bufferType, out[0]); // bind buffer as actual buffer
```

```
        GLES20.glBufferData(bufferType, buffer.capacity() * bytesPerUnit, buffer, usage); // write
            data into buffer
        GLES20.glBindBuffer(bufferType, 0); // unbind buffer for future use

        return out[0]; // returns buffer id for future bindings
    }

    // creates GL buffer for vertices
    int glVerticeBufferID = genBuffer(verticeFloatBuffer, GLES20.GL_ARRAY_BUFFER, Buffers.
        BYTESPERFLOAT, GLES20.GL_STATIC_DRAW);

    // creates GL buffer for face order
    int glFaceBufferID = genBuffer(faceShortBuffer, GLES20.GL_ELEMENT_ARRAY_BUFFER,
        Buffers.BYTESPERSHORT, GLES20.GL_STATIC_DRAW);
```

Výpis 33: Vytvoření statického bufferu do video paměti.

Pomocí takto vytvořených bufferů, jež jsou uloženy ve video paměti, lze vykreslit polygon bez nutnosti dalších přesunů dat mezi systémovou a video pamětí. Buffer je možné dále vytvořit jako dynamický (výměna dat) nebo streamovaný (výměna dat při čtení).

```
        GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, glVerticeBufferID);
        GLES20.glVertexAttribPointer(verticeCoordHandle, 3, GLES20.GL_FLOAT, false, 0, 0);
        GLES20.glEnableVertexAttribArray(verticeCoordHandle);

        GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, glTextureBufferID);
        GLES20.glVertexAttribPointer(textureCoordHandle, 2, GLES20.GL_FLOAT, false, 0, 0);
        GLES20.glEnableVertexAttribArray(textureCoordHandle);

        GLES20.glBindBuffer(GLES20.GL_ELEMENT_ARRAY_BUFFER, glFaceBufferID);
        GLES20.glDrawElements(GLES20.GL_TRINAGLES, faceCount, GLES20.
            GL_UNSIGNED_SHORT, 0);

        GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, 0);
        GLES20.glBindBuffer(GLES20.GL_ELEMENT_ARRAY_BUFFER, 0);
        GLES20.glDisableVertexAttribArray(verticeCoordHandle);
        GLES20.glDisableVertexAttribArray(textureCoordHandle);
```

Výpis 34: Vykreslení polygonu z vido paměti.

Způsob použí VBO je velmi podobný předchozí ukázce, kdy byl polygon vykreslen ze systémové paměti. Avšak jednotlivé buffery se už do video paměti nemusí posílat, ale stačí je pouze přiřadit k jednotlivým atributům shader programu. Po vykreslení polygonu jsou opět jak buffery tak i ukazatele shader programu deaktivovány.

8 Skeletální animace

Od začátku vývoje knihovny bylo prioritou zobrazovat 3D objekty a 3D svět do kterého bezpochyby patří i pohyb. Jako způsob, který tento pohyb bude zajišťovat byla vybrána skeletální animace. Výsledný objekt je reprezentován mesh daty a soustavou kostí s transformačními operacemi.

8.1 Formát skeletal data souboru

Při výběru vhodného formátu pro ukládání a načítání animací byl z hlediska již vytvořené implementace *.beo* souborů do knihovny, jako nástroje pro získávání mesh dat, opět vybrán a rozšířen tento script pracující v jazyce Python v programu Blender. Rozšíření skriptu se týkalo hlavně možnosti rozdělit jednotlivé vrcholy mesh objektu do skupin tak, aby odpovídali vždy některé z kostí, která bude s daným vrcholem pracovat a aplikovat na něj transformace. Informace o skeletální animaci je možné exportovat přímo i s mesh daty objektu, nebo v případě, že je na objekt aplikováno více akcí, může uživatel zvolit export pouze vybrané akce, bez mesh dat.

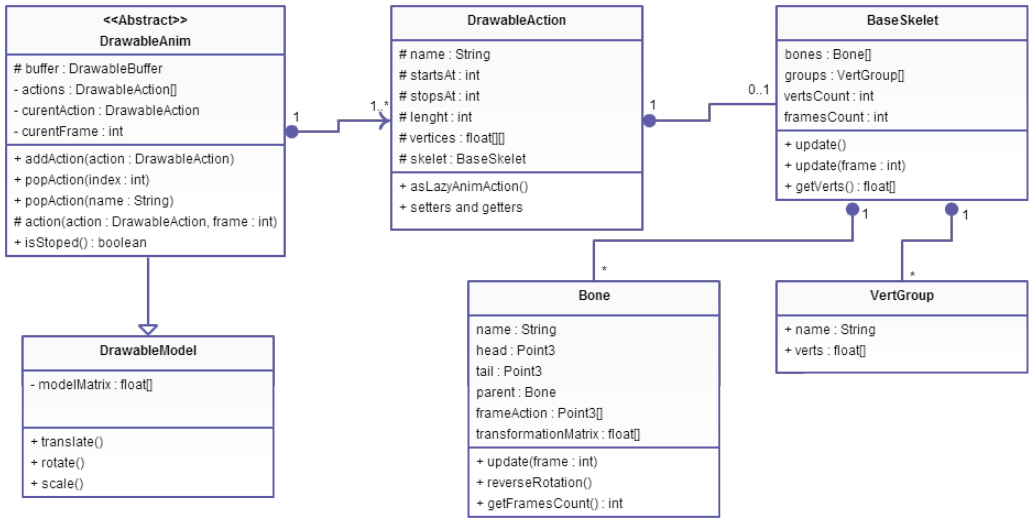
Struktura souboru:

```
version 0.256
ob AnimSampe
a 2 groups
cpv 3
bbox 2.0 2.0 2.0
v 60 {
Bone1 30 −1.0 1.0 1.0 1.0 1.0 1.0  .....
Bone2 30 −1.0 −1.0 1.0 1.0 −1.0 1.0 .....
}
t 40 { 0.0 1.0 1.0 1.0  .....  }
f 36 { 0 1 2 1 2 3  .....  }
skelet 3 {
Root −1 { 0.0 0.0 0.0 0.0 0.5 0.0 }
Bone1 0 { 0.0 0.5 0.0 −1.0 1.0 0.0 }
Bone2 1 { −1.0 1.0 0.0 1.0 0.0 0.0 }
}
frames 2 {
0.0 0.0 45.0 0.0  .....
0.0 0.0 90.0 0.0  .....
}
```

Výpis 35: Struktura skeletal mesh souboru

Oproti již známému mesh souboru se liší nový soubor pouze v reprezentaci vrcholů. Nyní jsou rozděleny do skupin podle toho, ke které kosti patří. Texturovací souřadnice jsou pouze přeskládány, aby odpovídali pozicím vrcholů rozdělených do skupin. Indexy ploch zůstaly stejné. Přičemž počet kostí může být větší než počet skupin do kterých jsou vrcholy rozděleny. Dále je zde zahrnuta struktura kostry (skelet) kdy prvním elementem je název kosti, následuje index rodičovské kosti (kořenová kost je označena indexem -1)

a složená závorka obsahuje souřadnice obou kloubů. Následující položkou jsou samotné kroky akce (frames), kdy pro každou kost jsou zde zahrnuty transformační operace ve všech krocích animace.



Obrázek 17: Diagram popisující strukturu tříd řešící animace.

8.2 Kosti

Jednotlivé kosti jsou vedeny samostatně ve třídě *Bone*. Celý skelet poté tvoří soustava kostí. Kost má své jméno podle kterého je přiřazena ke skupině vrcholů, referenci na předchozí kost, na kterou je napojena a souřadnice určující dva klouby. Dále kost obsahuje seznam transformací a transformační matici. Každá kost přebírá transformační matici od rodičovské kosti a na kopii této matice aplikuje vlastní transformace.

```
Point3 trans = new Point3();
Point3.copy(head, trans);
if (parent != null){
    BaseMatrix.copy(parent.transformMatrix, transformMatrix);
    Point3.sub(trans, parent.head);
} else {
    BaseMatrix.setIdentity(transformMatrix);
}

Point3 rot = frameAction[frame];

BaseMatrix.translate(transformMatrix, trans.x, trans.y, trans.z);
BaseMatrix.rotate(transformMatrix, rot.x, rot.y, rot.z);
```

Výpis 36: Aplikace transformace na kost skeletu.

8.3 Aplikace transformací na objekt

Soustava kostí spolu se skupinami vrcholů jsou umístěny ve třídě $BaseSkelet$, kde probíhá aplikace transformací kostí na mesh vrcholy. Aplikace transformační matice se provádí na každý vrchol zvlášť, dle skupiny do které patří.

```
for (Bone bone : bones) {
    VertGroup group = VertGroup.getGroup(bone.name, groups);
    if (group != null) {
        Point3 h = bone.head;
        float [] m = bone.transformMatrix;
        float [] v = group.verts;
        int index = VertGroup.startIndex(group, groups);
        for (int j = 0; j < v.length; ) {
            Point3 p = BaseMatrix.multiplyMV(m, v[j++], v[j++], v[j++]);
            vertice[index++] = p.x;
            vertice[index++] = p.y;
            vertice[index++] = p.z;
        }
    }
}
```

Výpis 37: Aplikace transformací na vrcholy polygonu.

Výstupem tohoto cyklu je float pole $vertice$, kde jsou uloženy modifikované souřadnice vrcholů.

8.4 Akce

Každý mesh objekt může mít jednu nebo i více skeletálních akcí. Pro ukládání akcí slouží třída $DrawableAction$. V této třídě lze skeletální data s transformacemi buť pouze uchovávat, nebo lze i předpřipravit všechny kroky animace do polí vrcholů. Dále je zde možné specifikovat od kterého okna má animace začínat, případně ve kterém okně má skončit.

8.5 Animace objektu

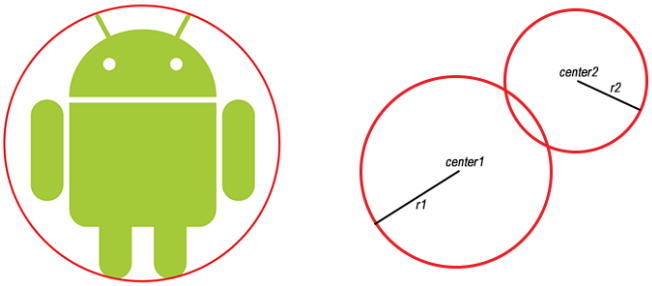
Pro zobrazení animace slouží abstraktí třída $DrawableAnim$, ze které dále dědí třídy $SkeletAnimDrawable$ (objekt založený na skeletální animaci, kdy se v každém kroku modifikují vrcholy polygonu) a $LazyAnimDrawable$ (všechny kroky animace jsou dopředu vypočítany a uloženy ve float polích). Dvojtý buffering je zajištěn pomocí bufferů, které se postupně střídají v zapisování a čtení (aktualizace dat probíhá na první buffer a data pro vykreslení se čtou z druhého bufferu, v dalším kroku se role bufferů vymění).

9 Řešení kolizí mezi objekty

Detekce kolizí, průniky dvou a více objektů, je důležitou součástí především při tvorbě video her. V knihovně jsou implementovány základní funkce pro detekci kolizí mezi objekty. Jednotlivé metody řeší pouze jestli došlo k průniku ohraničení objektu či nikoliv. Pro využití sofistikovaných případů kolizí a aplikaci fyzikálních vlastností na objekt je zde možnost využít fyzikální enginy třetích stran.

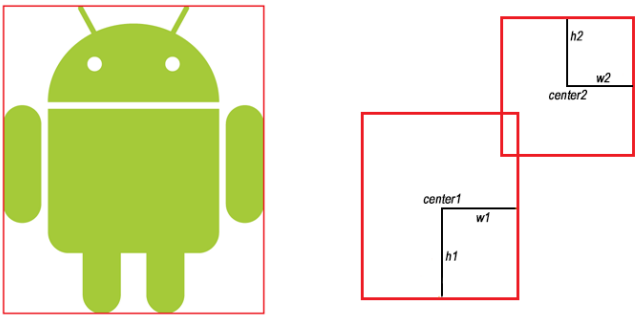
9.1 Kolize ve 2D rovině

Prvním a nejprostším způsobem jak můžou objekty mezi sebou kolidovat, je pomocí kruhového ohraničení. Objekt je ohraničen kruhem definující středový bod a poloměr. Test kolize s jiným ohraničeným objektem probíhá porovnáním vzdálenosti středových bodů a součtem poloměrů.



Obrázek 18: Kruhové ohraničení objektu.

Druhé řešení problému je reprezentace objektu jako obdelníku přilehlého k ose. Obdelník je definován středem a polovinou šířky, výšky. Test kolize s jiným ohraničeným objektem probíhá porovnáním vzdálenosti na jednotlivých osách a součtem polovin sířek a výšek obdelníků.

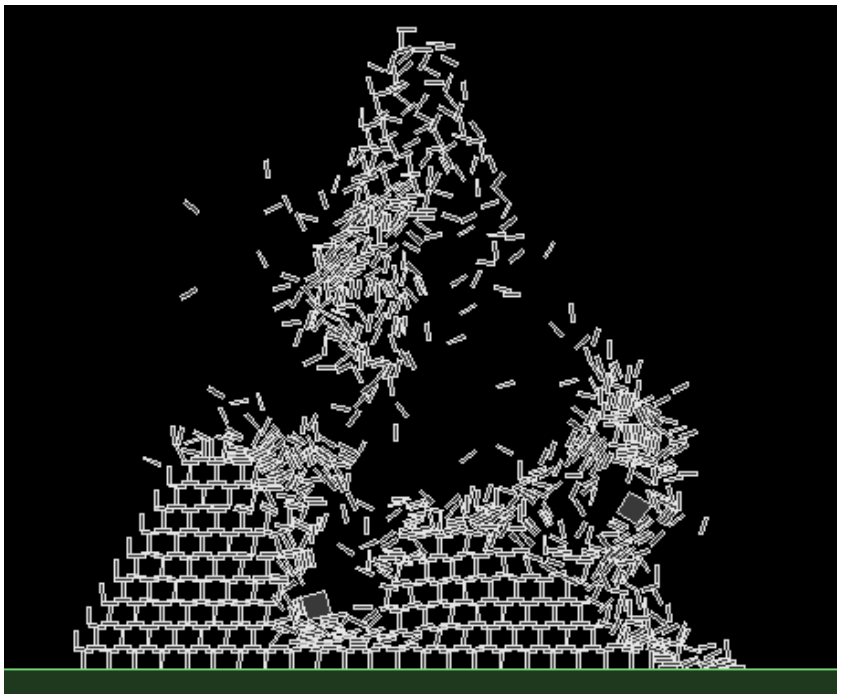


Obrázek 19: Obdélníkové ohraničení objektu.

9.1.1 Box2D

Jedná se o volně dostupný fyzikální engin vyvíjen programátorem Erinem Catto a publikován pod licencí *zlib*. První verze tohoto enginu vyšla již v roce 2007 a systém je podporován dodnes. Jedná se tedy o velmi stabilní a spolehlivý produkt. Box2D představuje simulaci rigidních těles. K simulaci využívá objekty složené z konvexních i nekonvexních polygonů, kruhů a tvarů složených stěnami. Jednotlivé objekty lze sdružovat různými druhy spojů, nebo na ně aplikovat síly. Engin také aplikuje gravitační či frikční sílu. Pro povrch objektu lze dále definovat jeho hrubost či elasticitu, ze které lze vypočítat odrazovou sílu při kolizi. Systém dále nabízí kontinuální detekci kolizí, která je nezbytná především pro příliš rychlé objekty či velký objem struktur. Box2D intermě při výpočtech používá metrický systém k dosažení simulace fyzikálních vlastností reálného světa.

Knihovna implementuje důležité postupy při práci s tímto enginem tak, aby systém fungoval dle očekávání. Dochází zde například k synchronizaci aktualizace Box2D světa s vytvářením, ničením jednotlivých těl objektů či spojů. Dále je zde možnost *debug* módu, kdy jsou těla objektů reprezentována drátěnými modely. Zahrnuto je i přímé vytváření primitivních objektů jako je kruh, obdelník a různé polygony. Knihovna především pomáhá a zjednodušuje postupy pří práci s tímto rozsáhlým systémem.

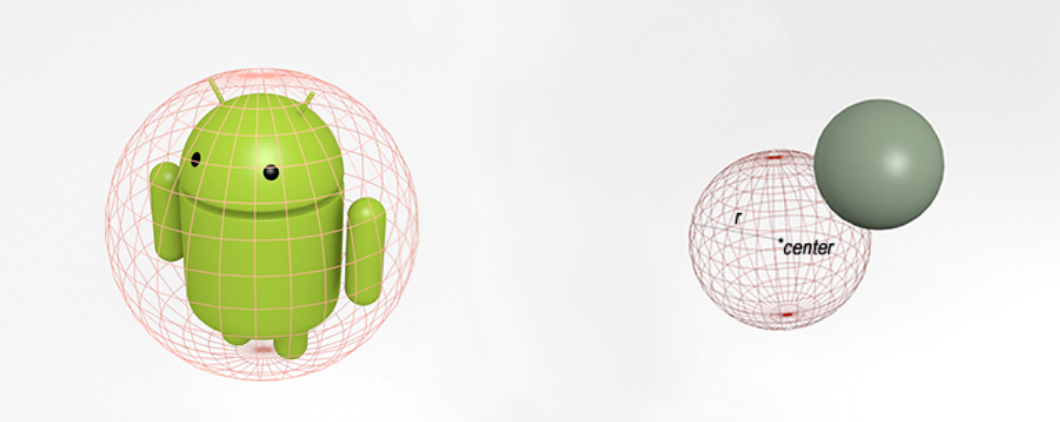


Obrázek 20: Ukázka Box2D simulace.

Zdroj: http://www.jbox2d.org

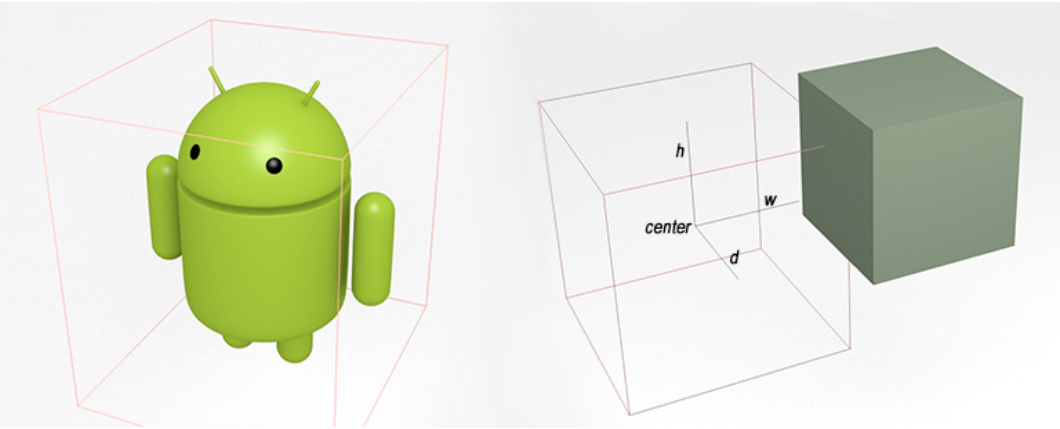
9.2 Kolize v 3D prostoru

Tak jako v 2D rovině lze použít kruhově(kulově) ohraničené objekty pro test kolizí. Objekt je uzavřen do koule definující středový bod a poloměr. Test kolize s jiným ohraničeným objektem probíhá porovnáním vzdálenosti středových bodů a součtem poloměrů.



Obrázek 21: Objekt ohraničený koulí.

Model lze dále reprezentovat jako krychli přilehlou k ose. Krychle je definována středovým bodem a polovinou šířky, výšky a hloubky. Test kolize s jiným ohraničeným objektem poté probíhá porovnáním vzdálenosti a součtem paremetrů v každé ose.



Obrázek 22: Objekt ohraničený krychlí.

9.2.1 Bullet physics

Jedná se o volně dostupný fyzikální engin vyvíjen programátorem Erwinem Coumans a publikován pod lincencí *zlib*. Systém má dlouholetou tradici a dodnes je podporován

a vyvíjen. Jedná se tedy o velmi stabilní a spolehlivý produkt. Systém je integrován do mnoha úspěšných 3D modelovacích programů a herních enginů. Taktéž je podporován tvůrci filmových efektů. Bullet umožňuje simulaci jak rigidních, tak deformovatelných 3D těles. K simulaci kolizí mezi objekty využívá všechny standartní tělasa jako je koule, krychle, válec, konvexní i nekonvexní objekty a mesh data. Bullet knihovna dále umožňuje oddělit sekci věnovanou detekci kolizí od zbytku SDK tak, aby byla tato část samostatně použitelná. Aplikace gravitačních a jiných sil je ve fyzikálním módu samozřejmostí. Bullet interně nejlépe pracuje při zadání jednotek v metrickém systému, pro dosažení fyzikálních vlastností reálného světa.

Do knihovny není zatím tento systém přímo integrován, především kvůli jeho obsáhlosti a variabilitě. Pro použití této knihovny je však možné využít některý z volně dostupných JNI Wrapperů, umožňujících volat nativní C/C++ metody z prostředí Java.

Obrázek 23: Ukázka Bullet physics simulace.

Zdroj: http://www.mashpedia.com/Kapla

10 Ukázková aplikace

Součástí bakalářské práce je i vypracovaná ukázkové aplikace, prezentující jednotlivé fragmenty knihovny. V menu aplikace lze najít:

- *Primitives* - Objekty generované knihovnou a základní UI prvky. Plus transformace a zobrazení doteků na obrazovce.
- *Shaders* - Ukázka dosavadních shaderů aplikovaných na polygon.
- *Model3D* - Načtení mesh souborů. Využití bufferingu na pole objektů. Aplikace transformací na objekt.
- *Camera* - Vizuální rozdíl mezi orthografickou a perspektivní kamerou. Jednoduché transformace kamer.
- *Animation* - Načtení mesh souborů pro skeletální a předpřipravenou animaci. Dále načtení akcí skeletu a jejich následné provádění.
- *Particles* - Možnosti generování jednoduchých částicových systémů.
- *Box2D* - Ukázka vybraných funkcí tohoto enginu a jejich integrace do knihovny. Kolize objektů, aplikace fyzikálních sil a různé vlastnosti povrchů.
- *Audio* - Načtení a ovládání různých zvukových stop.
- *Scene* - Komplexní prostředí obsahující více než minutovou pasáž v podobě průchodu scénou.
- *Game* - Implementace hry prezentující možnosti knihovny.

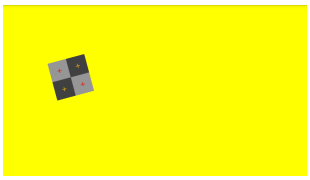
Obrázek 24: Fragmenty ukázkové aplikace.

10.1 Ukázkový projekt

Následující zdrojový kód demonstruje vytvoření Android aplikace, základní struktury a jejich použití. Výstupem tohoto kódu je rotující čtverec, který se pohybuje po křivce.

```
public class MainActivity extends BaseActivity {

    @Override
    protected void onCreate() {
        Base.debug = true;
        setFullScreen();
        setScreenOrientationLandscape();

        setView(new Render());
    }

    class Render extends BaseRender{

        BaseDrawable rect;
        BezierCurve curve;

        public Render(){
            camera = BaseCamera.ortho(20);
            rect  = new BaseDrawable(DrawableData.RECTANGLE(2.5f, 2.5f));
            rect.prepareDrawable(camera);
            curve = new BezierCurve(new Point2(0, -camera.getSemiHeight()), .....);
            addDrawable(rect);
        }

        @Override
        protected void onCreate() {
            setClearColor(Colorf.YELLOW);
        }

        @Override
        protected void onUpdate() {
            rect.translate(curve.next2());
            rect.rotateZ(rect.getRotZ()+7.5f);
            if(curve.isDone()) curve.reverse();
        }
    }
```

Výpis 38: Základní struktura nové aplikace.



Obrázek 25: Zobrazení výstupu aplikace.

11 Závěr

V práci je podrobně popsána problematika praktické implementace jednotlivých fragmentů knihovny a popis některých již zabudovaných funkcí jak OpenGL ES API, tak i Android SDK a práce s nimi.
Od jiných podobných produktů se vyvýjená knihovna snaží odlišit převážně v né příliš velké obecnosti a složitosti při běžné práci s jednotlivými elementy. Pomáhá řešit i základní nastavení Android zařízení a OpenGL ES systému.
I když se knihovna postupem vývoje stala komplexním a rozsáhlým rozhraním, základní struktura je centralizována do několika tříd s nimiž je možné plně využít sílu a přizpůsobivost této knihovny.
Velký důraz byl kladen i na výslednou strukturu a srozumitelnost knihovny, aby s ní mohl pracovat i méně zkušený programátor. S tím související jednoduchost a nenáročnost při vytváření struktur a jiných zdrojů zaručuje rychlou cestu k tvorbě kvalitních aplikací.
Výsledné rozhraní umožňuje uživateli plně využít sílu hardwarové akcelerace rasterizace scény pomocí OpenGL ES, bez nutnosti hlubší znalosti tohoto systému či vytváření a správou zdrojů se kterými pracuje.
Také implementační vlastnosti knihovny automaticky pomáhají uživateli se správou dat a zdrojů tak, aby tyto důležité akce byly vždy provedeny ve správnou chvíli životního cyklu aplikace.
Knihovna bude i nadále vyvýjena a vylepšována pro praktické nasazení při tvorbě Android aplikací.

12 Reference

[1] Google Android, *Dokumentace Android SDK.* [Online] `http://www.developer.android.com`

[2] Mark Segal, Kurt Akeley, *The OpenGL Graphics System : A Specification.* [Online] `http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf`

[3] Jon Leech, *Khronos Native Platform Graphics Interface.* [Online] `http://www.khronos.org/registry/egl/specs/eglspec.1.5.pdf`

[4] Robert J. Simpson, *The OpenGL ES Shading Language.* [Online] `http://www.khronos.org/files/opengles_shading_language.pdf`

[5] Niels Joubert, *Viewing and Camera Control in OpenGL.* [Online] `http://www.njoubert.com/teaching/cs184_fa08/section/sec09_camera.pdf`

[6] Reto Meier, *Professional Android 4 Application Development.* Wrox, 2012, ISBN-13: 978-1118102275

[7] FLIPCODE, *Advenced OpenGL Texture Mapping.* [Online] `http://www.flipcode.com/archives/Advanced_OpenGL_Texture_Mapping.shtml`

[8] Caz S. Hrstmann, Core Java(TM), *Volume I–Fundamentals.* Prentice Hall, 2007, ISBN-13: 978-0132354769

[9] Aaftab Munshi, Dan Ginsburg, *OpenGL ES 2.0 Programming Guide.* Addison-Wesley, 2008, ISBN-13: 978-0321502797

A CD

Obsah CD přílohy:

- Bakalářská práce ve formátu PDF
- Zdrojové soubory knihovny
- Zdrojové soubory ukázkové aplikace a hry
- Kompilovaný $.apk$ soubor ukázkové aplikace
- Ukázkový projekt