VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

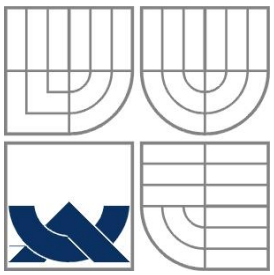
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

# 3D ZÁVODNÍ HRA PRO PLATFORMU ANDROID

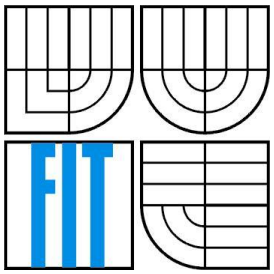
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Ing. MARTIN ŠEVČÍK
AUTHOR

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

# 3D ZÁVODNÍ HRA PRO PLATFORMU ANDROID

3D RACING GAME FOR ANDROID PLATFORM

DIPLOMOVÁ PRÁCE
MASTER‘S THESIS

AUTOR PRÁCE
AUTHOR

Ing. MARTIN ŠEVČÍK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ALEŠ LÁNÍK

BRNO 2013

# Abstrakt

Táto diplomová práce se zabývá možnostmi využití open source 3D herního enginu Java Monkey Engine (jME) při vývoji aplikací pro platformu Android. Obsahuje teoretické poznatky k architektuře jMonkey Engine a platformě Android. Dále popisuje použité techniky a externí knihovny při řešení problému interakce jME a Androidu a při práci s objekty v 3D scéně a jejich způsob implementace.



# Abstract

This master's thesis deals with the possibilities of using open source 3D game engine Java Monkey Engine (jME) in developing applications for the Android platform. It includes theoretical knowledge for the jMonkey Engine architecture and the Android platform. In the following section, the thesis describes used techniques and external libraries in solving issues of interaction between jME and Android platform and of working with objects in the 3D scene and their way of implementation.



# Klíčová slova

Java Monkey Engine, jME, jMonkey Engine, Android, NiftyGUI, jBullet, Open source, 3D závodní hra pro Android, 3D Android hra, 3D scéna, 3D modely, 3D Studio Max, 3DS Max.

# Keywords

Java Monkey Engine, jME, jMonkey Engine, Android, NiftyGUI, jBullet, Open source, 3D racing game for Android, 3D Android game, 3D scene, 3D models, 3D Studio Max, 3DS Max.


# Citace

ŠEVČÍK, M. *3D závodní hra pro platformu Android.* Diplomová práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2013.

# 3D závodní hra pro platformu Android

## Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Aleše Láníka
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Martin Ševčík
6. máj 2013

## Poděkování

Chcel by som poďakovať môjmu vedúcemu Ing. Alešovi Lánikovi za odborné a podnetné vedenie, poskytnuté zariadenie s platformou Android Nexus 7, ktoré slúžilo po celú dobu vývoja pre testovanie aplikácie a taktiež za ústretovosť pri výbere zadania. Rovnako sa chcem poďakovať mojej rodine, priateľom a všetkým, ktorí ma pri vytváraní tejto diplomovej práce podporovali. Ďakujem!

# Obsah

# 1 Úvod

V posledných rokoch možno badať trend neustále sa zvyšujúceho počtu ľudí používajúcich moderné multifunkčné elektronické prístroje, medzi ktoré v neposlednej rade patria chytré telefóny, označované tiež ako smartfóny, ako aj tablety. Ich obľúbenosť medzi používateľmi stále rastie, a preto sa čoraz viac kladie dôraz na možnosti ich využitia v bežnom živote. Vďaka neustále sa rozširujúcim funkciám a pribúdajúcim aplikáciám sa z týchto zariadení vyvinuli nielen veľmi sofistikovaní pracovní pomocníci, ale hlavne pre mnohých používateľov aj zábavné a multimediálne centrá. A práve pre týchto herných nadšencov sa smartfóny a tablety stávajú novými hernými konzolami súčasnej éry. Mobilná platforma Android dovoľuje prakticky každému vytvárať aplikácie a rozširovať tak funkcie a možnosti využitia týchto moderných zariadení.

Pri tvorbe počítačovej hry je pre väčšinu vývojárov jedným z hlavných rozhodujúcich faktorov čas, za ktorý sú schopný priniesť svoj produkt na trh. Preto, ak chcú uspieť v silnej konkurencii, musia vyvíjať hry pomerne rýchlo a kvalitne a pokiaľ možno s čo najmenšími nákladmi na vývoj. Jedným z riešení je práve použitie open source herného enginu Java Monkey Engine (jME), ktorý umožňuje pomerne rýchlo a jednoducho vytvárať trojrozmerné herné aplikácie pre platformu Android. Vďaka možnosti použitia externých 3D modelov je možné dosiahnuť vytvorenie dokonalých a po vizuálnej stránke prepracovaných hier v pomerne krátkom čase.

Preto táto práca prináša demonštráciu využitia hernej knižnice jMonkey Engine pri vývoji aplikácií pre Android, a to na praktickom príklade vytvorenia trojrozmernej hry.

Úvodná kapitola č. 2 popisuje úvod do prostredia Java Monkey Engine, jeho históriu, architektúru, graf scény, objekt kamery, či prácu zo zdrojmi v prostredí jME. Kapitola č.3 sa venuje architektúre hernej knižnice Lightweight Java Game Library a jej výhodám pri vytváraní počítačových hier. Kapitola č.4 stručne pojednáva o histórii programu Autodesk 3D Studio Max, spôsobe vytvorenia polygónu a transformáciách objektov. Kapitola č.5 sa venuje platforme Android z pohľadu histórie, architektúry a jeho výhod pri vytváraní počítačových hier. Kapitola č. 6 sa venuje návrhu hernej aplikácie a ponúka zhrnutie poznatkov, prečo je pre vývojárov výhodné vytvárať herné aplikácie pomocou jMonkey Engine pre platformu Android a prečo je toto prepojenie témou tejto diplomovej práce. Kapitola č. 7 popisuje konkrétny postup pri implementácii aplikácie, jej štruktúru, vlastnosti jednotlivých elementov a ich dôležité prvky, integráciu fyziky, zvukového systému, užívateľského rozhrania, multi-dotykového ovládania, efektov a ďalších prvkov. Záverečná kapitola č.8 ponúka zhrnutie limitácií jME pri vývoji hry pre platformu Android a otestovanie jej výkonnosti.

Výsledkom tejto práce je vytvorená 3D počítačová hra pomocou herného enginu jME 3, ktorá je spustiteľná na zariadeniach s operačným systémom Android.

# 2 Java Monkey Engine

Táto kapitola sa venuje v krátkosti histórii herného enginu Java Monkey Engine (jME, jMonkey Engine) a popisuje prehľad jeho architektúry, objektu kamery, grafu scény, prácu so zdrojmi v prostredí jME a vývojové prostredie jMonkey Engine SDK. Kapitola bola čerpaná z [1, 2, 3].

## 2.1 Prehľad jME

jMonkey Engine je open source herný engine vydaný pod novou BSD licenciou [4] a bol vyvinutý pre vývojárov, ktorí chcú vytvárať 3D hry za pomoci posledne dostupných moderných technologických štandardov. jME Framework je naprogramovaný v čistej Jave a využíva LWJGL ako svoj predvolený vykresľovací modul. Zahrňuje plnú podporu pre OpenGL 2 až OpenGL 4. Jeho hlavnými výhodami sú, že je široko dostupný a dovoľuje vývojárom pomerne rýchle nasadenie svojho produktu od jeho vytvorenia až po jeho jednoduchú prispôsobiteľnosť pre rôzne platformy, či už sú to desktopové riešenia, webové applety alebo mobilné telefóny a tablety využívajúce Android operačný systém.

## 2.2 História

Java Monkey Engine bol vytvorený roku 2003 autorom menom Mark Powell ako vedľajší projekt, na ktorom sa chcel presvedčiť, či je možné vytvoriť pokročilé grafické API v jazyku Java. V ranných začiatkoch vývoja sa nechal inšpirovať C++ knihou *3D Game Engine Design* od Davida Eberlyho. Na konci toho istého roka sa k vývojárskemu tímu pripojil Joshua Slack a stal sa jeho neoddeliteľnou súčasťou. Títo dvaja hlavní predstavitelia spolu s ďalšími členmi vývojárskeho tímu vyvíjali jMonkey Engine od základnej verzie až po verziu 2.0 a podarilo sa im založiť komunitu, ktorá existuje dodnes a naďalej sa rozvíja.

Od augusta 2008 prešiel projekt vývoja jME viacerými zmenami. Počiatočný vývojový tím bol nahradený novými členmi. Joshua Slack oznámil odstúpenie od aktívneho vývoja a novým vedúcim programátorom ako aj jediným architektom a vývojárom novej verzie jME 3.0 sa stal Kirill Vainer. Prvá skúšobná verzia jME 3.0 na začiatku roku 2009 spôsobila v komunite veľký rozruch. Nakoniec sa jej valná väčšina zhodla na tom, že táto nová verzia bude oficiálnym nástupcom jME 2.0. Jadro tímu sa v súčasnosti skladá z ôsmich angažovaných jednotlivcov. V máji roku 2010 bola vydaná prvá alfa verzia **jMonkey Engine 3.0** a o pár mesiacov na to bolo zverejnené IDE (Integrated Development Environment) pre vývoj aplikácii v jME **jMonkey Engine SDK** (Software Development Kit). V súčasnosti je pre vývojárov k dispozícií stabilná verzia **jMonkey Engine 3 SDK RC2** [5].

## 2.3 Architektúra jME

Java Monkey Engine bol navrhnutý ako vysokorýchlostný real-time grafický engine. Je implementovaný v programovacom jazyku Java. S pokrokmi vo vývoji hardvéru, počítačových grafických kariet a taktiež s pokročilým vývojom programovacieho jazyka Java sa ukázalo, že je nevyhnutné a potrebné vytvoriť pre Javu hernú knižnicu. A jME je práve touto hernou knižnicou, ktorá spĺňa mnohé z potrieb herných vývojárov práve poskytovaním vysoko výkonného vykresľovacieho systému grafu scény. Tento grafický engine využíva najnovšie funkcie OpenGL, ktoré umožňujú naplno využiť možnosti zobrazovania a vykresľovania najmodernejších grafických kariet za predpokladu jednoduchého používania zo strany herných vývojárov.

jME sa môže pochváliť s veľmi prijateľným intuitívnym ovládaním pre užívateľa, čo zvyšuje silu tohto programovacieho jazyka. Ďalšou výhodou je, že dokázal spojiť intuitívne rozhranie s pokročilými technikami moderného grafického programovania. Jeho modulárna architektúra zaisťuje možnosť rýchleho prispôsobenia sa novým vylepšeniam z oblasti grafického hardwaru, a to aj vďaka tomu, že OpenGL neustále zlepšuje svoju prispôsobivosť a zdokonaľuje svoje vykresľovacie schopnosti v súčinnosti s najnovšími grafickými kartami. Preto je jME pomerne rýchlo schopné využívať tieto nové vylepšenia a vďaka svojej architektúre sa daným zmenám prispôsobiť takmer okamžite.

Jadrom jME systému je graf scény. Graf scény je dátová štruktúra, ktorá obsluhuje dáta celého virtuálneho sveta. Vzťahy medzi dátami v hre napríklad geometria, zvuk, fyzikálne zákony a podobne sú uchovávané v stromovej štruktúre s listovými uzlami reprezentujúcimi základné prvky hry. To sú práve elementy, ktoré sú zvyčajne vykresľované do scény alebo prehrávané prostredníctvom zvukovej karty.

Organizácia grafu scény je veľmi dôležitá a väčšinou závislá od druhu a typu aplikácie. Keďže graf scény je väčšinou reprezentovaný veľkým množstvom dát, je tu možnosť tieto dáta rozdeliť na menšie, ľahšie ovládateľné a dostupné elementy. Obyčajne sú tieto elementy zoskupované a poprepájané vzájomnými vzťahmi, veľmi často priestorovým rozmiestnením. Toto zoskupenie umožňuje odstrániť veľké sekcie dát hry, ktoré nie je potrebné vykresľovať na aktuálnej scéne. Napríklad ak nie je potrebné, aby sa zobrazovala určitá sekcia virtuálneho sveta, získame tým rýchlejšie spracovávanie dát, pretože zaberieme menej času CPU a GPU potrebného pri ich spracovaní a vďaka tomu je rýchlosť hry zlepšená.

Kým graf scény je jadro grafických prvkov v jME, aplikačné nástroje poskytujú rýchlu možnosť vytvorenia grafického obsahu a spustenie hlavnej slučky hry. Vytvorenie herného okna, vstupného systému, kamerového a herného systému obsluhuje volanie jednej, či dvoch metód. To umožňuje programátorom stráviť viac času prácou na ich vlastnej aplikácií, než nad zdĺhavým implementovaním metód potrebných k spusteniu aplikácie. Zobrazovací a vykresľovací systém umožňuje oddelene komunikovať s grafickou kartou, a to z dôvodu, aby bolo možné spracovať veľké

množstvo dát prostredníctvom vykresľovacieho systému, ktorý je pre užívateľa neviditeľný počas celého behu aplikácie. Hlavné slučky hry obstarávajú možnosť jednoduchého rozšírenia a zaručujú jednoduchý vývoj aplikácie.

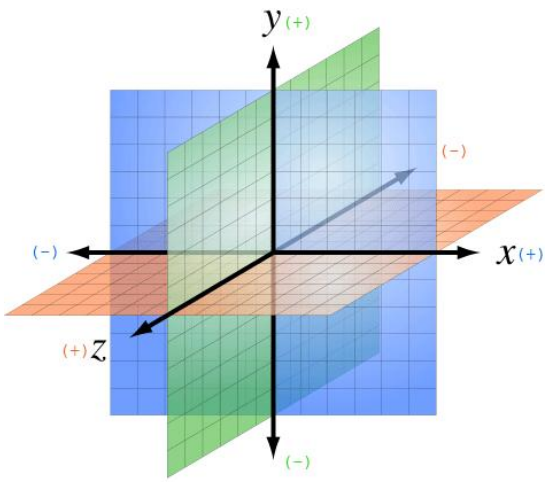
Skutočné vykresľovanie grafu scény je skryté pred užívateľom. To má za výhodu možnosť prepínania medzi rozdielnymi vykresľovacími systémami bez prepisovania jediného riadku kódu aplikácie. Ak aplikácia pre spustenie používa knižnicu LWJGL (Lightweight Java Game Library) od Puppy Games, nie je problém prepnúť na druhú knižnicu JOGL (Java OpenGL) od spoločnosti Sun Microsystems. Je možné, že sa vyskytnú nejaké menšie rozdiely vo výslednom zobrazení, ale nie je potrebná prestavba kódu celého projektu. Avšak zameranie vývojárov sa postupne prenieslo na pridávanie nových vylepšení iba do jedného vykresľovacieho systému, čo malo za následok zastaranosť toho druhého a tým pádom aj jeho neatraktívnosť. Práve z tohto dôvodu je v súčasnosti podporovaný iba jeden vykresľovací systém a ním je práve LWJGL [6].

Každý nástroj jME systému bol postavený na základe jednoduchých stavebných blokov. Takže všetky grafické elementy dedia z triedy `Spatial`, čo predstavuje abstraktnú dátová štruktúru, ktorá uchováva informácie o transformácii objektu (transláciu, rotáciu, veľkosť a podobne). Všetky vstupy a im pridružené akcie dedia z triedy `com.jme3.input`. Tým je zaistená konzistencia, vďaka ktorej používanie pokročilejších metód a technológií bude podstatne jednoduchšie a pomocou nástrojov obsiahnutých v jME je vytvorenie aplikácie rýchle a efektívne [1].

### 2.3.1 Koordinačný systém

jMonkey Engine používa trojrozmerný pravouhlý koordinačný systém rovnako ako OpenGL. Pozostáva z:

- počiatočného bodu v priestore. Tento bod má vždy súradnice (0, 0, 0)
- tieto tri súradnicové osy sú na seba navzájom kolmé a stretávajú sa v počiatočnom bode
    - X-ová os je „vpravo/vľavo“.
    - Y-ová os je „hore/dole“.
    - Z-ová os je „vzadu/vpredu“.

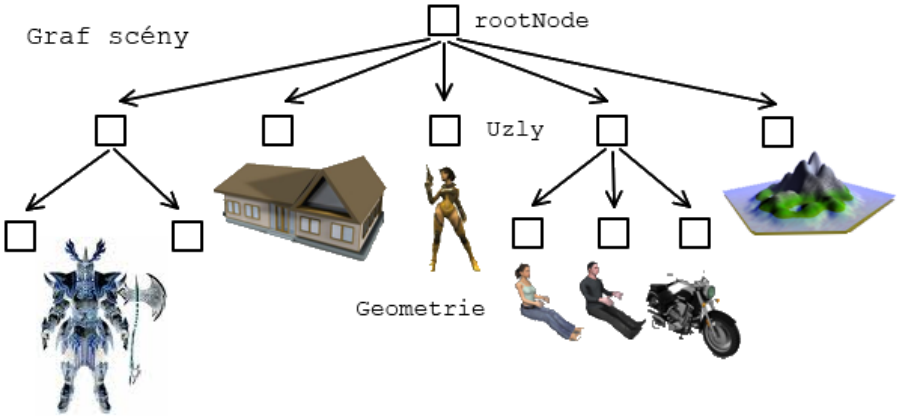



**Obrázok 2.1: Koordinačný systém v jME (Zdroj [7])**

Každý bod v 3D priestore je definovaný svojimi (X, Y, Z) súradnicami. Dátový typ pre vektory je z triedy `com.jme3.math.Vector3f`. Prednastavené umiestnenie kamery je nastavené na (0.0f, 0.0f, 10.0f) a pozerá sa v smere (0.0f, 0.0f, -1.0f). To znamená že uhol pohľadu kamery je umiestnený na pozitívnej strane Z-ovej osy a smeruje k počiatočnému bodu pod miernym sklonom smerom nadol. Merateľnou jednotkou je `world unit (wu)`. Typicky 1 `wu` je považovaný za jeden meter a všetky veľkosti, vektory a body sú relatívne k tomuto koordinačnému systému.

## 2.4 Graf scény

Graf scény je dátová štruktúra, ktorá obsahuje hierarchické zoskupenie uzlov podľa priestorového umiestnenia. Koreňový uzol `rootNode` je hlavný element v grafe scény. Všetky ostatné elementy sú pripojované pod neho. Graf scény je tvorený dvoma typmi uzlov, a to vnútornými a listovými. Vnútorné uzly sa nazývajú uzlami a listové uzly sa nazývajú geometrie. Uzly môžu obsahovať potomkov (iný uzol alebo geometriu), zatiaľ čo geometria potomkov obsahovať nemôže. Listové uzly stromu reprezentujú dáta, ktoré majú byť spracované zatiaľ čo vnútorné uzly stromu pomáhajú tieto dáta spravovať.




**Obrázok 2.2: Príklad grafu scény (Zdroj [7])**

Existujú štyri hlavné dôvody, prečo je takáto organizácia scény výhodná pri tvorbe hier:

- Dáta v hre (objekty, prostredie a ďalšie) sú zvyčajné veľké. Sú normálne usporiadané v priestore podľa ich polohy, takže tieto dáta môžu byť jednoducho zoskupené v priestore v 3D svete. Keďže svet je organizovaný týmto spôsobom, je prirodzené, že umiestnenie napríklad uzlu predstavujúceho svetlo ovplyvní ďalšie elementy pod ním. Toto je zaobstarané automaticky vďaka grafu scény.
- Takáto hierarchická štruktúra poskytuje referenčný bod pre priestorové umiestnenie jednotlivých objektov prostredníctvom koreňového uzla. Koreňový uzol obsahuje všetky objekty k nemu pripojené. Vďaka tomu je možné jednoducho odstrániť irelevantné časti grafu scény.
- Veľa elementov vo virtuálnom svete je reprezentovaných takouto stromovou štruktúrou, a preto je ľahké zobraziť ich štruktúru. Pozícia a rotácie uzlov sú dedené smerom nadol. To znamená, že ak nadradený uzol zaznamená pohyb alebo rotáciu, tak aj jeho podriadené uzly dedia danú operáciu, a tá sa prejaví aj na ich polohe.
- Je nenáročné vyjadriť graf scény v iných nástrojoch. To napríklad umožňuje scéne, aby bola vytvorená v grafickom editore a prostredníctvom XML alebo iného nástroja bola jednoducho importovaná do jME.

Každý uzol ako vnútorný tak aj listový obsahuje dôležité informácie o sebe samom: transformácia, ohraničujúci zväzok, vykresľovací stav a kontroléry.

**Transformácia** (Transformation) definuje orientáciu, polohu a veľkosť uzla a vzťahuje sa aj na potomkov, takže rotácia rodiča ovplyvní aj všetkých jeho potomkov.

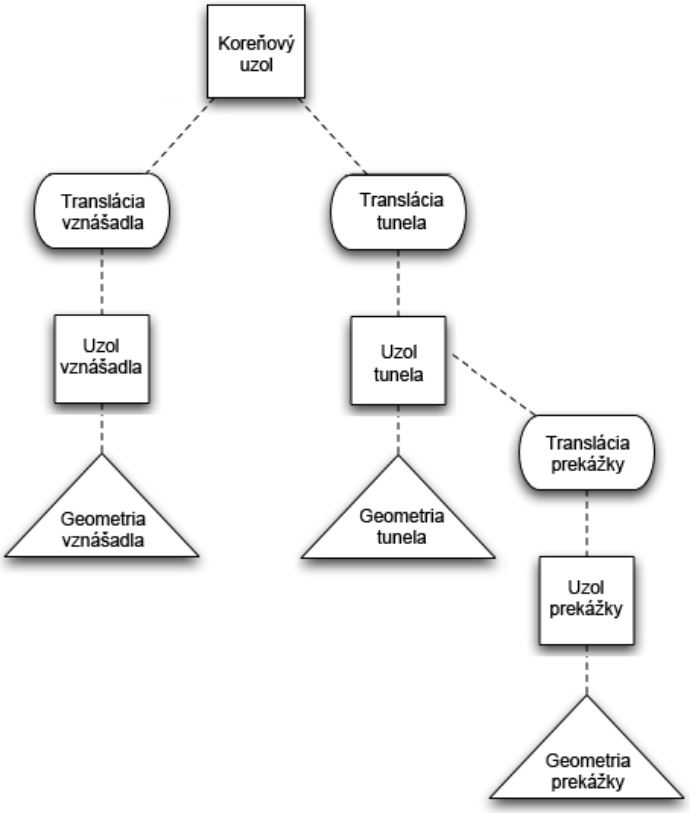
**Ohraničujúci zväzok** (Bounding Volume) definuje oblasť, ktorá minimálne obsahuje veľkosť príslušného uzla a jeho vrcholov. Preto zväzok na úrovni geometrie obsahuje všetky jeho vrcholy. Avšak zväzok na úrovni uzla pokrýva oblasť aj všetkých jeho potomkov.

**Stav vykresľovania** (Render State) definuje ako budú jednotlivé elementy vykresľované do scény. To ďalej ovplyvní aj ich potomkov. Toto zaručuje minimálne prepínanie stavu, teda ak uzol obsahuje tisíce rovnakých geometrií, tento uzol môže obsahovať jeden vykresľovací stav pre textúru, ktorý nastaví rovnakú textúru pre všetkých tisíc geometrií.

**Kontroléry** (Controllers) slúžia na definovanie modifikácii uzlov za časovú jednotku. To zvyčajne zahŕňa animácie modelu, fyzikálne vlastnosti (hodenie loptičkou) a podobne. Ovládače samé o sebe nemôžu ovplyvniť svojich potomkov avšak môžu mať efekt na uzol [1, 7].

Tento príklad ukazuje možné navrhnutie organizácie hry vznášadla v tuneli. Existujú tri hlavné typy geometrie: tunel, vznášadlo a prekážka. Tunelu by malo byť umožnené, aby bol voľne umiestnený

v priestore, a preto je pripojený ku koreňovému uzlu. Pre zachovanie voľného pohybu vznášadla v priestore je vznášadlo taktiež ako tunel pripojené ku koreňovému uzlu. Prekážka sa môže nachádzať iba vo vymedzenom priestore, a to práve vo vnútri tunela. A z dôvodu, aby sa pri zmene polohy tunela zaručil zároveň aj pohyb prekážky, je uzol pripojený k uzlu tunela.




**Obrázok 2.3: Graf scény- príklad hry vznášadla v tuneli (Zdroj: vlastný)**

### 2.4.1 Osvetlenie

Osvetlenie je dôležitou témou v 3D grafike. Svetlá v scéne majú vážny dopad na scénu. Umožňujú, aby sa javila ako vysoko atmosférická alebo skoro fotograficky reálna, alebo naopak ako plochá a umelá. Existujú dva základné problémy s nastavovaním osvetlenia pre scénu. Prvý spočíva v pochopení ako je osvetlenie prepočítavané, ako tiež pochopenie mechaniky nastavovania svetelných parametrov a písanie potrebnej programovej časti. Druhý problém je estetický a zahŕňa umelecké rozhodovanie týkajúce sa toho, ako má scéna vyzerať. V jME 3 je možné pomerne jednoducho pripojiť do scény niekoľko typov svetiel za použitia metódy `rootNode.addLight()`. Všetky objekty, ktoré majú byť v scéne viditeľné vyžadujú svetelný zdroj. Dostupné svetelné zdroje v jME 3 sú:

- `PointLight`
- `DirectionalLight`
- `AmbientLight`
- `SpotLight`

Niektorým z nich sa dá dodatočne nastaviť intenzita, rádius a pozícia. Pri svetle `PointLight` nastavujeme pozíciu a svietivosť do všetkých smerov. S väčšou vzdialenosťou od svetla sa jeho intenzita znižuje. `DirectionalLight` nemá v scéne pozíciu, ale iba smer vrhania svetla. Vrhá nekonečne dlhé paralelné svetelné lúče. Dokáže vrhať tieň a je typicky používaný na simuláciu slnka. `AmbientLight` ovplyvňuje jas scény globálne, pretože na všetky objekty vrhá rovnaké svetlo. Nemá smer ani pozíciu a nevrhá tieň. `SpotLight` imituje baterku, ktorá vrhá zreteľný lúč. Má smer, pozíciu a taktiež vzdialenosť a dva uhly, ktoré vymedzujú osvietený priestor. Všetky objekty, ktoré sú mimo tento vymedzený priestor svetelného lúču, nie sú svetlom ovplyvnené. Pre vrhanie tieňov v scéne je použitý `ShadowRenderer`. Uzly v scéne dokážu tiene vrhať a prijímať. Chovanie tieňov sa dá pre každý uzol v scéne nastaviť osobitne, pretože nerozumné používanie vrhania tieňov môže mať na plynulosť aplikácie značný dopad [8].

### 2.4.2 Zvukový systém

jME 3 podporuje dva typy zvukových súborov, a to Ogg Vorbis audio (`.ogg`) a PCM Wave (`.wav`) formáty. Existujú tu dve možnosti ako v scéne obstarať zvuk. Krátke audio súbory sú celé načítané do pamäte. Dlhé súbory môžu byť streamované priamo z disku. Každému súboru sa dá nastaviť hlasitosť, loop, maximálny dosah zvuku pre počutie a možnosť 3D priestorového efektu. Novinkou je možnosť prehratia inštancie zvukového uzla, hlavne keď je potrebné prehrať ten istý zvuk viackrát v scéne zároveň a bez ovplyvnenia predošlých inštancií. Prednastaveným poslucháčom v scéne je statický uzol, pri použití priestorového zvuku existuje možnosť pripojiť poslucháča k objektom v scéne. Zväčša sa to využíva s pripojením ku kamere [9].

## 2.5 jME Kamera

jME využíva triedu `com.jme3.rendered.Camera`, ktorá obsahuje niekoľko kľúčových nastavení. Tieto nastavenia majú za následok ako bude výsledná renderovaná scéna vyzerať. Teda koľko a čo bude z virtuálneho sveta vykreslené a zobrazené pre koncového užívateľa.

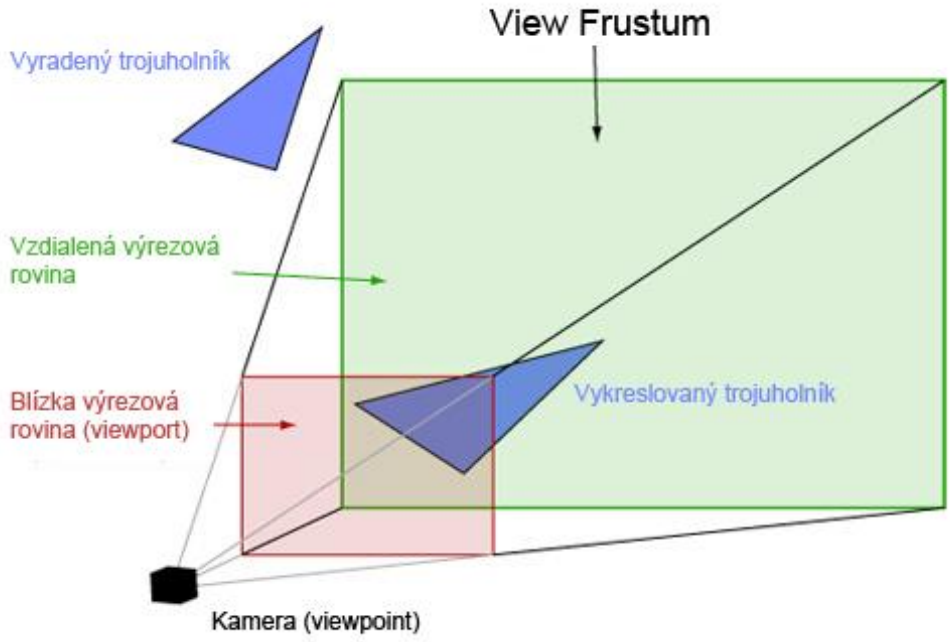
Kamerou môže byť zaobchádzané rovnako ako s akýmkoľvek iným objektom v scéne, ktorý môže byť pripojený do scény ako uzol alebo ako jednoduchý objekt. To si však vyžaduje použitie objektu `cam`.

### 2.5.1 Možnosti kamery

Základným objektom kamery v jME je objekt `cam`. Je to uzol, do ktorého sa pripája vytvorená kamera. To umožňuje využiť objekt kamery ako súčasť scény. Prínosom je, že je následne možné pripojiť kameru na akýkoľvek objekt v danej scéne a jej ďalšie použitie je pomerne jednoduché a zautomatizované. Preto je väčšinou zaobchádzané s kamerou ako s elementom scény. To umožní jej

pripojenie v akejkoľvek výške stromu grafu scény, čo otvára veľa možností využitia. Napríklad pripojenie kamery na lietadlo, kde nie je potrebné sa obávať pohybu lietadla, pretože kamera sa bude pohybovať automaticky s ním. Preto lietadlo nikdy nezmizne zo zorného uhlu kamery, a teda aj užívateľa. Taktiež je možné pripojiť uzly na kameru, čím sa dá vytvoriť pohľad z prvej osoby.

Umiestnenie a nastavenia kamery definuje **View Frustum**, čo je logický popis celkového objemu priestoru, ktorý je viditeľný pre užívateľa.




**Obrázok 2.4: View Frustum (Zdroj: vlastný)**

Takto novo vytvorený vrchol je v podstate obdĺžnik, ktorý reprezentuje obrazovku užívateľa. To umožňuje rozšírenie zorného poľa. Všetko v rámci pyramídy je vykresľované, zatiaľ čo ostatné objekty nie je potrebné vykresľovať. Práve vďaka tejto možností je jME schopné zvládnuť zložité scény zložené z mnohých objektov pri zachovaní vysokého počtu snímok za sekundu (FPS).

Nastavenia kamery budú teda definovať, kde je View Frustum umiestnené, jeho orientáciu v rámci virtuálneho sveta a vlastnosti jeho bočných stien. Všetky tieto nastavenia teda dávajú možnosť vytvoriť si pohľad na svet taký, ako sám autor požaduje, či už náhľad z veľkej výšky so širokým zorným uhlom alebo tunelové videnie pri umiestnení kamery čo najnižšie. A práve tieto možnosti vyzdvihujú schopnosti jMonkey Engine [1].

## 2.6 Práca s modelmi a zdrojmi v jME

Slovom model je v jME myslená perzistentná a obnoviteľná reprezentácia objektu v 3D virtuálnom svete. Veľkým prínosom toho, že daný model nie je potrebné pracne vytvárať v jME rôznymi základnými útvarmi a dohliadať na ich výslednú harmóniu, ale stačí jednoducho vytvoriť model v externom 3D grafickom editore je ten, že takýmto spôsobom je možné dosiahnuť kvalitné grafické

efekty a taktiež vytvoriť dokonalé prepracované modely po vizuálnej stránke, či už pre aplikáciu alebo hru, čo určite vyzdvihuje a uľahčuje prácu pri vývoji týchto zložitých programov. Prínos je naozaj obrovský, pretože je pomerne jednoduchšie vytvoriť objekty pre aplikáciu v externom modelovacom programe ako napríklad Blender, Maya, 3D Studio Max, než ich zdĺhavo manuálne programovať kúsok po kúsku v jazyku Java.

jMonkeyEngine 3.0 SDK obsahuje vstavaný modelový importér, ktorý dovoľuje prezerať a importovať podporované modely do aplikácie. Model je konvertovaný do `.j3o` súboru a všetky ďalšie potrebné súbory sú skopírované do projektového priečinku obsahujúceho zdroje. Binárny formát `.j3o` je pre platformu dobre optimalizovaný, vďaka čomu je beh aplikácie oveľa plynulejší aj pri použití prepracovaných 3D modelov. jME 3 SDK dokáže načítať a konvertovať široké spektrum najznámejších 3D formátov [10, 11]:

- `.3ds` – formát programu Autodesk 3D Studio Max.
- `.Ase` – ASCII Scene Exporter, jedná sa o textovú podobu 3DsMax formátu [12].
- `.Md2` – modely z hry Quake2.
- `.Md3` – modely z hry Quake3.
- `.Md5` – modely z hry Doom3.
- `.Ms3d` – MilkShape modely.
- `.Obj` – Wavefront 3D objekty.
- `.X3d` – ISO štandard XML súborový formát pre reprezentáciu 3D grafiky.

Pre načítanie týchto modelov do prostredia jME slúžia samostatné triedy `XxxToJme`, kde Xxx je označenie formátu, ktoré dokážu tieto triedy prekonvertovať do formátu, ktorý využíva jME. Tieto triedy sú obsiahnuté v knižnici jME 3 v balíku `jme3tools.converters.model`, a preto nie je potrebné ich zdĺhavé vytváranie. Niektoré z týchto modelov ako napríklad md5, ktorý je jeden z najviac pokročilých 3D formátov, plne podporujú kostrovú animáciu a textúrovanie, čo určite pridá na prepracovanosti a vizuálnej hodnote modelu.

Tieto formáty môžu zvyčajne obsahovať viacero vlastností. Ukladajú v sebe napríklad 3D informácie ohľadom nastavenia kamery, svetla, textúry alebo systémového nastavenia programu.

### 2.6.1 Asset Manager

jMonkey Engine 3.0 má v sebe zabudovaného správcu prostriedkov `AssetManager`, ktorý pomáha udržovať použité zdroje v aplikácii dobre organizované. Medzi zdroje radíme multimediálne súbory, 3D modely, materiály, textúry, scény, vlastné shadery, zvukové súbory a vlastné písma. Jeho využitie je myslené ako súborový systém pre aplikáciu, ktorý je nezávislý na aktuálnej vývojovej platforme.

Medzi jeho hlavné výhody patria [13]:

- Cesty k súborom ostávajú stále rovnaké a sú nezávislé na platforme, na ktorej aktuálne hra beží, či už je to Windows, Mac, Linux alebo Android.
- Asset Manager automaticky optimalizuje správne spravovanie OpenGL objektov, napríklad zabraňuje viacnásobnému nahrávaniu rovnakých textúr do grafickej karty v prípade, že tieto textúry používa viacero objektov.
- Prednastavený buildovací skript automaticky zabalí všetky potrebné zdroje do spustiteľného súboru. Pokročilý užívatelia si môžu písať svoje vlastné buildovacie skripty a registrovať si vlastné cesty k jednotlivým súborom.

Asset Manager teda predstavuje zdrojovú cestu a prístup ku všetkým prostriedkom v aplikácii. Príklad použitia:

```
Spatial ninja = AssetManager.loadModel("Models/Ninja/Ninja.j3o");
rootNode.attachChild(ninja);
```

## 2.7 jMonkey Engine SDK

Od verzie jME 3.0 je možné využívať pre vývoj aplikácií vlastné SDK. Základ tohto SDK je v známom projekte NetBeans. Oproti NetBeans sú tu však pridané niektoré ďalšie funkcie typické práve pre programovanie 3D grafiky [14]:

- Importovanie, zobrazovanie a konvertovanie modelov.
- Real-time ladenie aplikácie, je možné hýbať uzlami a efektmi za behu aplikácie.
- Prináša možnosť uloženia celej scény, napríklad do formátu `.XML`.
- Scene Composer - pridávanie jednotlivých prvkov do scény z prednastaveného zoznamu elementov, podobne ako v iných 3D programoch.
- Scene Explorer – prináša štrukturálny prehľad aktuálne editovanej scény a umožňuje jej upravovanie.
- Obsahuje vlastné editory pre špeciálne typy súborov. Materiálový editor, editor terénu, editor modelov a podobne.

# 3 Lightweight Java Game Library

Táto kapitola stručne popisuje architektúru knižnice Lightweight Java Game Library (LWJGL) a jeho výhody pri programovaní hier. Informácie sú čerpané z [6].

## 3.1 Prehľad LWJGL

LWJGL je riešením pre programovanie kvalitných hier v jazyku Java ako pre profesionálnych, tak aj pre začínajúcich Java programátorov. LWJGL poskytuje vývojárom prístup k vysoko-výkonným medzi-platformovým knižniciam ako napríklad **OpenGL** (Open Graphics Library) a **OpenAL** (Open Audio Library), ktoré umožňujú tvorbu najmodernejších 3D hier s priestorovým 3D zvukom. Navyše LWJGL dokáže zariadiť prístup k ovládačom ako gamepad, volant, či joystick, čím sa ovládanie hry môže stať zábavnejšie, jednoduchšie a vierohodnejšie.

LWJGL nebol vytvorený za účelom jednoduchého programovania hier, ale primárne za účelom sprístupniť technológiu, ktorá umožní vývojárom používať prostriedky, ktoré sú inak nedostupné alebo len veľmi slabo implementovateľné na existujúcej platforme Javy. Predpokladá sa, že prostredníctvom dlhodobejšieho vývoja a možnosťami rozšírenia bude knižnica LWJGL základom pre programovanie komplexnejších herných knižníc a herných enginov. LWJGL je dostupná pod BSD licenciou, čo znamená že je open source a jeho používanie nie je nijako spoplatnené. Pomocou LWJGL je možné dosiahnuť:

- Rýchlosť
- Jednoduchosť
- Všeobecnosť
- Šetrnosť
- Bezpečnosť
- Robustnosť
- Minimalizmus

### 3.1.1 Rýchlosť

Hlavným cieľom LWJGL bolo zvýšiť rýchlosť vykresľovania Javy. Preto sa napríklad vyhodili metódy určené pre efektívne programovanie v C, ktoré nemajú zmysel pre Javu, napríklad metóda `glColor3fv`.

Knižnica umožňuje hlásiť výnimku, ak na danom prístroji nie je dostupná hardwarová akcelerácia, pretože nemá zmysel spúšťať aplikáciu s príliš nízkym FPS.

### 3.1.2 Jednoduchosť

LWJGL potrebuje byť jednoduchá, aby ju mohlo využívať široké spektrum vývojárov. Preto na jednej strane nie je ťažké sa rýchlo a efektívne oboznámiť s možnosťami tejto knižnice aj pre začiatočníkov a použiť nadobudnuté poznatky v praxi. Na strane druhej však ponúka profesionálnym vývojárom veľké možnosti ako svoje skúsenosti premietnuť do tvorby aplikácie.

Autori v nej pracujú s dvomi paradigmami. Prvým, ktorý skutočne po všetkých stránkach vyhovuje pre OpenGL a druhým, ktorý zasahuje presne cielené platformy obsiahnutých od najmenších prístrojov ako napríklad PDA až po desktopové riešenia. Preto bola z hernej knižnice LWJGL odstránená veľká škála nepotrebných konštrukcií, ktoré herní programátori vôbec nevyužívali a nepotrebovali k vývoju aplikácií. A pretože konzistencia je lepšia ako komplexnosť je v knižnici namiesto pomalých polí používaný zásobník, vďaka čomu bola z knižnice vypustená možnosť volania jednej a tej istej metódy viacerými spôsobmi, a tým predchádzať zahlcovaniu knižnice.

### 3.1.3 Všeobecnosť

Knižnica LWJGL je navrhnutá, aby mohla spoľahlivo fungovať na veľkom množstve zariadení od malých ako napríklad telefóny až po multiprocesorové vykresľovacie servery. Síce v súčasnosti žiadne telefóny alebo konzoly nemajú dostatočne rýchle JVM (Java Virtual Machine) [15] a 3D akceleráciu, avšak LWJGL je vyvíjaná s ohľadom na tieto budúce možnosti. Aj preto je v nej implementovaná podpora **OpenGL ES** ( OpenGL Embeded System). Čo je štandard pre akcelerované 3D grafické vstavané systémy. To dosiahla aj vďaka tomu, že jej binárna distribúcia má menej ako 1MB a pritom dokáže zaobstarať 3D zvuk, grafiku, vstupy a výstupy. To určite ocenia menšie zariadenia, ktoré nemajú veľký pamäťový priestor.

### 3.1.4 Šetrnosť

Malé a šetrné sa rovná:

- Jednoduché. Čím je menej spôsobov ako niečo spraviť, tým je to jednoduchšie sa naučiť.
- Nechybové. Zdrojový kód obsahuje minimálne množstvo chýb.
- Stiahnuteľné. LWJGL zaberá skutočne iba málo priestoru, čo mu umožňuje byť stiahnuteľný každou aplikáciou, ktorá ho dokáže používať.
- J2ME. Java 2 Micro Edition je súčasťou programovacieho jazyka Java spoločnosti Sun Microsystems. Táto platforma je určená k programovaniu spotrebnej elektroniky počnúc mobilnými telefónmi, rôznymi PDA až po špecifické moduly [16].

### 3.1.5 Bezpečnosť

Vývojári do tejto knižnice zaviedli zvýšenú bezpečnosť.

- Začal sa používať zásobník (buffer) namiesto ukazovateľov.
- Postupne sú pridávané ďalšie metódy na kontrolu, ktoré majú za účel zabezpečiť, aby nedošlo k pretečeniu zásobníka, a tým zaručiť zabránenie nežiaducich útokov cez zásobník.

### 3.1.6 Robustnosť

Keďže spoľahlivý systém je oveľa užitočnejší a cennejší ako systém, ktorý dbá iba na rýchlosť, boli aplikácie vytvorené pomocou LWJGL podrobené mnohým testom pre získanie skutočných dát prostredníctvom benchmarkov [6]. A na základe týchto testov výkonnosti boli assert makrá z knižnice odstránené a nahradené oveľa jednoduchšími podmienkami **if{}** alebo **try{} catch{}** sekciami. Navyše boli všetky kontroly GL chýb presunuté z natívneho kódu do Java kódu, čím sa zaistilo, že nie je naďalej potrebná oddelená DLL knižnica pre potrebu debugovacieho módu.

### 3.1.7 Minimalizmus

To je ďalší kľúčový faktor, ktorý hral úlohu v navrhovaní LWJGL. To čo nie je potrebné, aby bolo v knižnici, tak to tam jednoducho nie je. Pôvodným zámerom pri vyvíjaní tejto knižnice bolo vytvoriť knižnicu, ktorá bude poskytovať požadované minimum potrebné pre prístup k hardwaru, ku ktorému Java prístup nemala. A v takomto duchu sa aj naďalej vedie vývoj tejto knižnice. Aj preto bol napríklad odstránený GLU (OpenGl Utility Library) [17], pretože bol väčšinou nepotrebný pre herných vývojárov.

# 4 Autodesk 3D Studio Max

Táto kapitola sa venuje v krátkosti histórii programu Autodesk 3D Studio Max a popisuje základný prehľad jeho architektúry. Kapitola bola čerpaná z [1].

## 4.1 História

Autodesk 3D Studio Max (3DS Max) debutoval na trhu v roku 1990 ako vizualizačný nástroj pre profesionálov v architektúre, stavebníctve a inžinierstve hľadajúcich spôsob ako previesť ich 2D CAD/CAM výkresy do 3D prostredia, ktoré by ich nápady dokázal zobraziť zo všetkých uhlov. Každá animácia bola závislá od svetiel, prípadne kamery, ktorými sa dalo pohybovať. Napriek tomu však tento produkt dokázal vykresliť obrázky, ktorých kvalita presahovala možnosti všetkých dovtedy existujúcich programov. Neskôr sa v tomto programe začali zohľadňovať aspekty pre tvorbu 3D grafiky pre interaktívne počítačové hry a multimédia. A tak 3DS Max, predstavený v roku 1996, zaznamenal výrazný posun od iba vizualizačného nástroja s možnosťou obmedzenej animácie až k prostrediu s podporou plnej animácie.

3DS Max sa skladá z veľkého počtu rozšírení organizovaných okolo hlavného jadra programu. A to vďaka integrovanému vývojárskemu prostrediu, ktoré umožňuje skúseným programátorom v jazyku C tieto rozšírenia doplňovať a taktiež vďaka vlastnému skriptovaciemu jazyku Maxscript, v ktorom je možné vytvárať rôzne rozšírenia aj pre menej skúsených užívateľov. Zásluhou týchto možností sa 3DS Max uplatnilo vo filme, počítačových hrách, simuláciách, vizualizáciách a 3D grafike na internete.

## 4.2 Transformácie

3DS Max je v podstate polygonálny modelovací nástroj. Práca s polygónmi je veľmi efektívna, pretože pre lokalizáciu bodu v priestore sú potrebné iba tri súradnice (X, Y, Z). Pre vytvorenie čiary je potrebné vytvoriť dva body a funkciu, ktorá tieto dva body spojí. Pre vytvorenie plochy je potrebné lokalizovať tri body, spojiť ich v trojuholník a vymaľovať jednu stranu. Po takomto spojení vzniká plocha v priestore zvaná polygón.

V programe 3DS Max sa všetky transformácie objektov (pohybovanie, rotovanie, škálovanie) odohrávajú v prostredí zvanom Svet vesmíru. Vymodelovanie objektu prebieha pomocou manipulácie jeho geometrických komponentov (sub-objektov), napríklad vrchol, hrana, plocha a polygón, použitím ich vlastného súradnicového systému nazvaného Lokálne súradnice. Transformácie týchto komponentov na sub-objektovej úrovni sú zvyčajne brané ako sub-objektové modelovanie.

# 5 Android

Táto kapitola sa snaží priblížiť Android ako taký, ponúka krátky pohľad do jeho histórie a konkrétnejšie popisuje prehľad jeho architektúry a internetový obchod s aplikáciami Google Play. Kapitola bola čerpaná z [18, 19, 20, 21].

## 5.1 Priblíženie Android platformy

Android je mobilný operačný systém založený na jadre Linuxu. Ide o plnohodnotný operačný systém s užívateľským rozhraním, špecifikáciou ovládačov a bohatým ekosystémom externých aplikácií. Jeho najväčšou výhodou, ale zároveň nevýhodou je to, že je nezávislý od hardvéru, veľkosti obrazovky či rozlíšenia a dokáže abstrahovať od čipovej súpravy a konkrétneho výrobcu. Je dostupný vo forme bezplatného open source riešenia, takže si ho môže ktokoľvek stiahnuť, upraviť podľa potreby a nainštalovať na svoje zariadenie. Android je celkom otvorený systém, od modulov linuxového jadra, knižníc, aplikačného programového rozhrania až po základné aplikácie. To dáva výrobcom možnosť prispôsobiť systém nasadenému hardwaru a taktiež lepšie integrovať svoje aplikácie. Je šírený pod MIT licenciou [22], čo umožňuje tretím stranám využívať tento systém pre najrôznejšie účely. Jedine pre komerčné využitie je potrebná licencia.

Android ponúka plnohodnotné nástroje pre vývoj aplikácií prostredníctvom Software Development Kit (SDK). SDK je dostupné pre najbežnejšie operačné systémy, a to Linux, Windows a Mac. K dispozícii je v troch rôznych verziách:

- Základnej – knižnice, debugging, emulácia, testovanie.
- Odporúčanej – debugging na zariadení, kódy, dokumentácia.
- Plnej – Google API, Social API, ostatné platformy SDK.

Emulátor dokonca umožňuje testovanie aplikácií bez nutnosti mať zariadenie fyzicky k dispozícii, čo je veľmi vhodné pre začínajúcich programátorov, ktorý sa snažia ušetriť.

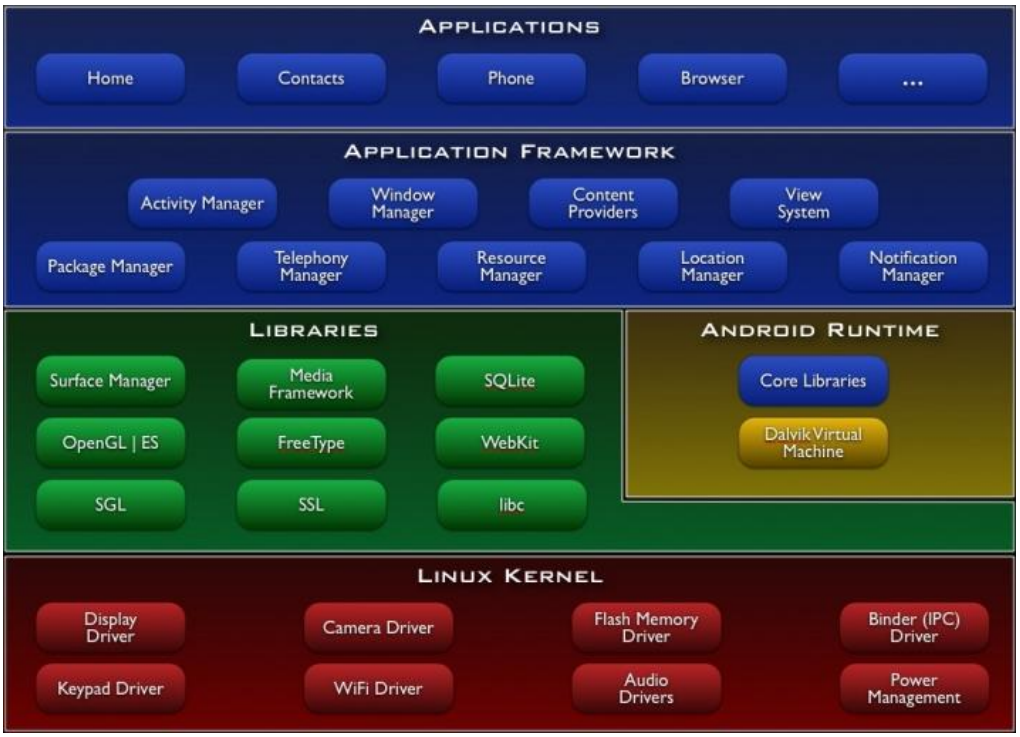
Systém Android vyvíja organizácia **Open Handset Alliance**, ktorej súčasťou sú desiatky firiem vrátane tých najznámejších v mobilnej branži, ako napríklad Google, Samsung, HTC, Motorola, Intel, Nvidia, Qualcomm a ďalšie. Ide o jeden z mála operačných systémov, ktoré podporujú viacero platforiem, a aj preto ho je možné vidieť v zariadeniach najrôznejších značiek. To však prináša na jednej strane značnú nevýhodu, pretože chýba optimalizácia systému na konkrétnu platformu, avšak na strane druhej je Android multiplatformový s možnosťou prispôsobenia a vytvorenia vlastnej nadstavby Androidu, tak ako to robí väčšina známych výrobcov, napríklad Samsung TouchWiz, HTC Sense, Sony Timescape, Motorola Motoblur a iné. Čo sa týka aktualizácií systému, tie nie sú vždy ihneď dostupné pre všetky zariadenia a užívateľ si musí počkať na konkrétnu aktualizáciu od svojho výrobcu.

## 5.2 História

Spoločnosť Android Inc. vznikla v októbri roku 2003 a založili ju štyria páni v kalifornskom Palo Alto. V tom čase nikto nedával Androidu veľkú šancu na úspech. Zlom nastal o dva roky neskôr, v auguste 2005, keď Google odkúpil Android Inc. za 50 miliónov dolárov (v súčasnosti je jeho hodnota tisícnásobne vyššia) a odštartoval tak svoj vstup do pola mobilných telefónov. V novembri 2007 bolo založené konzorcium **Open Handset Alliance**, ktoré stojí za vývojom Androidu dodnes. V tom čase vyšiel zároveň aj vývojársky kit SDK. O rok neskôr v septembri 2008 prišiel v USA na trh HTC Dream (G1), ktorý bol prvým smartfónom s Androidom 1.0, a v tom čase mal na trhu inteligentných telefónov podiel zanedbateľných 0,5 %. Vo februári 2009 vyšiel Android 1.1 ako aktualizácia pre G1 a od tohto okamihu každá ďalšia aktualizácia systému dostala kódové označenie po zákusku a vždy so sebou prinášala aj novú funkcionalitu pre Android platformu. V apríli 2009 bol vydaný **Android 1.5 Cupcake**, ktorý bol prvým masovým Androidom a priniesol so sebou podporu pre natívne knižnice v Android aplikáciách. V septembri toho istého roka prišiel **Android 1.6 Donut** a priniesol podporu pre rôzne rozlíšenia obrazovky. V zapätí nato vyšiel v októbri **Android 2.0 Eclair**, ktorý zahrňoval podporu pre multi-dotykový display. Na konci roka 2009 mal Android na trhu podiel skoro 4 % s mierne stúpajúcou tendenciou. Prelom prišiel v máji 2010, keď bola predstavená verzia **Android 2.2 Froyo**, ktorá priniesla just-in-time (JIT) kompiláciu do Dalvik Virtual Machine (DVM), na ktorom bežia všetky Java aplikácie v Android platforme, čím umožnila oveľa rýchlejšie spúšťanie aplikácií. Táto verzia zastupuje najviac predaných smartfónov s Androidom na svete (vyše 50 % v porovnaní s ostatnými verziami). V decembri 2010 prišla jedna z najznámejších verzií **Android 2.3 Gingerbread**, ktorá pridala nový konkurentný garbage collector pre Dalvik VM. Android sa začal približovať k pätinovému podielu na trhu, keď dosahoval hodnotu 17,7 %. Vo februári 2011 bola uvedená aktualizácia na verziu 2.3.3 a v máji 2011 bol vydaný prvý Android určený pre tablety vo verzii **Android 3.0 Honeycomb**, ktorý priniesol od základu nové užívateľské rozhranie. Celosvetový podiel Androidu stúpol na 22,20 % a zároveň bol ohlásený **Android 4.0 ICS (Ice Cream Sandwich)**, ktorý v októbri roku 2011 priniesol upravené a unifikované užívateľské rozhranie pre telefóny aj tablety. V júli roku 2012 prišla zatiaľ posledná verzia **Android 4.1 Jelly Bean**, ktorá priniesla množstvo vylepšení dizajnu systému, možnosť viacerých užívateľov v systéme spolu s novými vstavanými nastaveniami pre vývojárov. Túto verziu dnes používa drvivá väčšina novo predstavených dvojjadrových a štvorjadrových tabletov. Na prelome rokov 2012 a 2013 každý druhý smartfón obsahuje operačný systém Android a v súčasnosti má celosvetový podiel 46 % na trhu inteligentných telefónov a dokonca 57 % podiel medzi tými, ktorí si prvýkrát kupujú smartfón [20, 21].

## 5.3 Architektúra

Operačný systém Android má päťvrstvovú architektúru. Oficiálnym vývojovým prostredím je Eclipse (nevyhnutný ADT plugin). Návrh jeho architektúry je nasledujúci [23]:




**Obrázok 5.1: Architektúra Android platformy (Zdroj [23])**

### 5.3.1 Linux Kernel

Android funguje na Linux jadre verzii 2.6, ktoré pôsobí ako abstraktná vrstva medzi hardvérovou a softvérovou časťou a garantuje portabilitu. Zaobstaráva služby jadra systému, ako je bezpečnosť, správa pamäte, riadenie procesov, sieťový zásobník a ovládače.

### 5.3.2 Libraries

Obsahuje knižnice písané v C/C++, ktoré sú vývojárom dostupné cez Android Application Framework a sú používané rôznymi komponentmi Android platformy.

- **System C library** – BSD knižnica odvodená zo štandardnej systémovej C knižnice (libc), ktorá je optimalizovaná pre vstavané zariadenia založené na Linuxe.
- **Media Libraries** – fungujú na základe PacketVideo's OpenCORE. Tieto Knižnice podporujú prehrávanie a nahrávanie mnohých populárnych audio a video formátov a zobrazovanie statických obrázkových súborov, napríklad MPEG4, H.264, MP3, AAC, AMR, JPG, PNG a ďalšie.

- **Surface Manager** - riadi prístup k display subsystému a plynulo preskladáva 2D a 3D grafické vrstvy z rôznych aplikácií.
- **LibWebCore** – moderný webový engine pre prehliadač, na ktorého základe fungujú internetové prehliadače v Androide. Podporuje aj vložené náhľady webových stránok
- **SGL** - základný 2D grafický engine**.**
- **3D libraries** – Knižnice pre vykresľovanie 3D grafiky. Implementácia je založená na OpenGL ES 1.0 API. Používajú buď hardvérovú 3D akceleráciu (ak je dostupná), alebo vstavané vysoko optimalizované 3D softwarové rastrovanie.
- **FreeType** – vykresľovanie bitmapového a vektorového písma.
- **SQLite** – odľahčená knižnica pre prístup k relačným databázam, ktorá je dostupná všetkým aplikáciám.

### 5.3.3 Android Runtime

Android obsahuje sadu základných knižníc, ktoré poskytujú väčšinu dostupných funkcionalít implementovaných v základných knižniciach programovacieho jazyka Java. Knižnice sa svojím obsahom blížia platforme Java Standard Edition. Hlavný rozdiel je v neprítomnosti knižníc pre užívateľské rozhranie AWT a Swing, ktoré boli nahradené knižnicami užívateľského rozhrania pre Android a pridaním knižnice Apache pre prácu so sieťou.

Každá Android aplikácia spúšťa svoj vlastný proces s vlastnou inštanciou Dalvik Virtual Machine (DVM). Je implementovaný takým spôsobom, aby mohlo na jednom prístroji efektívne bežať viacero procesov súčasne. Využíva základné vlastnosti linuxového jadra, ako je správa pamäte, práca s vláknami a podobne. DVM je registrovo orientovaná architektúra, kde preklad Android aplikácie prebieha skompilovaním zdrojového Java kódu do Java byte kódu pomocou rovnakého kompilátora, ako sa používa v prípade prekladu Java aplikácií. Potom sa prekompiluje Java byte kód pomocou vstavaného Dalvik kompilátora a výsledný Dalvik byte kód je spustený na DVM. Ten spúšťa súbory v spustiteľnom Dalvik formáte `.dex`, ktorý je optimalizovaný tak, aby mal čo najmenšie pamäťové nároky.

Vznik špeciálneho virtuálneho stroja DVM bol iniciovaný z dvoch dôvodov. Prvým dôvodom bolo obídenie licenčných poplatkov za Java Virtual Machine, ktorý na rozdiel od jazyka Java a jeho knižníc nie je voľne šíriteľný. Druhým dôvodom bola optimalizácia virtuálneho stroja pre mobilné zariadenia, a to predovšetkým v oblasti pomeru úspory energie a výkonu [22, 23].

### 5.3.4 Application Framework

Tým, že Android poskytuje otvorenú vývojovú platformu, ponúka vývojárom možnosť vytvárať veľmi robustné a inovatívne aplikácie. Pre vývojárov je táto vrstva najdôležitejšia, pretože vďaka nej majú možnosť využiť výhody hardvérového vybavenia zariadenia, majú prístup k informáciám

o lokalite zariadenia, môžu spúšťať služby na pozadí, nastavovať upozornenia a alarmy, pridávať oznámenia a mnoho ďalších funkcionalít.

Vývojári majú plný prístup k API, ktoré zabezpečuje prístup k službám využívaným v základných aplikáciách (grafika, obsah, médiá a iné). Architektúra je navrhnutá tak, aby zjednodušovala opakované použitie jednotlivých komponentov. Každá aplikácia môže zverejňovať svoje vlastnosti a nastavenia, ktoré môžu následne využívať ostatné aplikácie (v závislosti na bezpečnostných obmedzeniach API). Základom všetkých aplikácií je súbor služieb a systémov, vrátane:

- Bohatej a rozšíriteľnej sady pohľadov (**Views**), ktoré môžu byť použité pri vytváraní aplikácie, vrátane zoznamov, mriežok, textových polí, tlačidiel, a dokonca aj v rovine webového prehliadača.
- Poskytovateľov obsahu (**Content Providers**), ktorí umožňujú aplikáciám prístup k dátam z iných aplikácií (napríklad ku kontaktom) alebo zdieľať vlastné dáta.
- Manažéra zdrojov (**Resource Manager**), ktorý poskytuje prístup k nekódovým zdrojom, ako sú lokalizované reťazce, grafika a vzorové súbory.
- Manažér oznámení (**Notification Manager**), ktorý povoľuje všetkým aplikáciám zobrazovanie vlastných upozornení v stavovom riadku.
- Manažér aktivít (**Activity Manager**), ktorý riadi životný cyklus aplikácií a poskytuje spoločné navigačné rozhranie pre orientáciu v zásobníku.

### 5.3.5 Applications

Android je dodávaný so základnou sadou predinštalovaných aplikácií vrátane e-mailového klienta, kalendára, máp, prehliadača, kontaktov a podobne. Rozšírenie ponúkajú aplikácie dostupné cez Google Play ako aplikácie tretích strán. Aplikácie majú štyri základné komponenty:

- **Activity** – reprezentuje obrazovku a grafické používateľské rozhranie.
- **Service** – realizuje vykonávanie akcií na pozadí, ide o samostatné vlákno/proces.
- **Content provider** – poskytuje ako komponent prístup k dátam, súborom a databázam.
- **Broadcast receiver** – komponent reagujúci na oznámenia systému.

## 5.4 Android Market/ Google Play

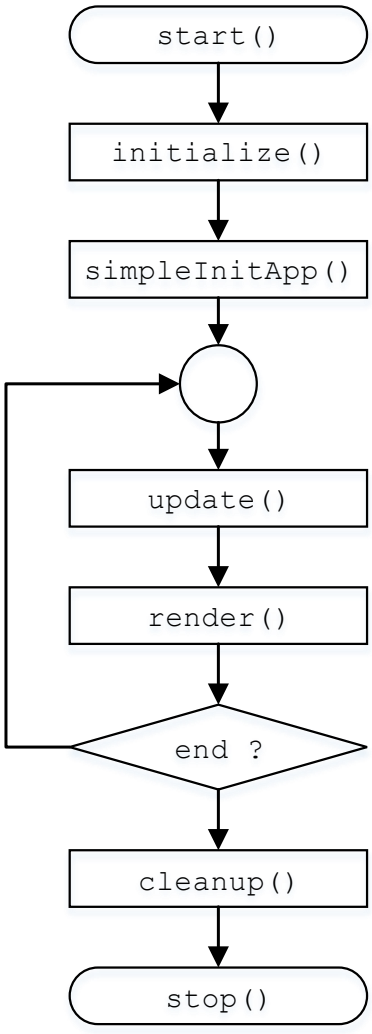
Android má vlastné miesto na sťahovanie aplikácií. Toto miesto sa v začiatkoch vývoja Androidu nazývalo Android Market, no neskôr od tohto názvu upustilo a premenovalo sa na Google Play, pričom po aktualizácii v mobilnom zariadení je uvádzaný názov Play Store. Na tomto mieste sú dostupné státisíce aplikácií, ktoré je možné sťahovať priamo zo svojho zariadenia prostredníctvom aplikácie Play Store alebo cez webový prehliadač. Google play je miesto, ktoré umožňuje vývojárom publikovať svoje aplikácie pre ostatných užívateľov Android zariadení.

Aplikácie sa dajú rozdeliť do dvoch kategórii. Prvou kategóriou sú voľne dostupné neplatené aplikácie. Druhou kategóriou sú práve platené aplikácie, ktoré tvoria zdroj príjmu pre Android, Google a pre vývojárov týchto aplikácií. Sťahované aplikácie mali kedysi limit 50 MB, no po jeho poslednom navýšení na 4 GB sa otvára možnosť vyvíjať naozaj prepracované aplikácie po všetkých stránkach. Google Play je dostupný v 190 krajinách a počet stiahnutých aplikácií minulý rok presiahol magickú hranicu 10 miliárd. Medzi najväčších sťahovačov aplikácií patria používatelia z Južnej Kórey, Hongkongu, Taiwanu, USA, Singapuru a Švédska. Najväčší podiel na sťahovaných aplikáciách majú hry, potom zábavné aplikácie, nástroje a aplikácie určené na komunikáciu [20].

# 6 Návrh aplikácie

Táto kapitola sa venuje návrhu hernej aplikácie, jej štruktúre a následne ponúka zhrnutie poznatkov, prečo je výhodné pre vývojárov vytvárať aplikácie pomocou jMonkey Engine pre platformu Android a prečo je práve toto prepojenie témou tejto diplomovej práce.

Základnou triedou pre vytvorenie aplikácie v jME 3 je `com.jme3.app.SimpleApplication`. Avšak pri vytváraní zložitejších aplikácií (hlavne hier) je možné, ba až žiaduce, túto triedu rozšíriť o požadovanú funkcionalitu ďalšími metódami pomocou dedenia. `SimpleApplication` je tým pádom schopná ponúknuť prístup k štandardným prvkom potrebných pre vývoj hier ako sú graf scény (`rootNode`), Asset manager, užívateľské rozhranie (`guiNode`), vstupy, zvukový systém, simulácia fyziky a podobne. Táto trieda ponúka niekoľko hlavných metód, ktoré sú potom prístupné pre prepisovanie kvôli väčšej voľnosti pri implementácii vlastnej aplikácie. Medzi ne patria:




**Obrázok 6.1: Herná slučka (Zdroj: vlastný)**

- `initialize()`- toto je prvá metóda, ktorá sa zavolá po metóde `start()`, teda po spustení aplikácie. Slúži k inicializácii jME systému a k vytvoreniu zobrazovacieho okna, v ktorom sa bude všetko odohrávať.
- `simpleInitApp()`- metóda volaná po ukončení `initialize()`. Spúšťa inicializáciu 3D grafu scény. Prebieha tu načítanie modelov, textúr, animácií a všetkých elementov používaných v aplikácii. Tie sú následne pripojené ku koreňovým uzlom.
- `update()`- zaisťuje aktualizáciu elementov v 3D scéne v čase. Táto metóda predstavuje hlavnú aktualizačnú slučku a je vykonávaná neustále až do konca aplikácie.
- `render()`- zaisťuje vykresľovanie elementov do 3D scény v čase. Táto metóda je volaná po každej iterácii aktualizačnej slučky.
- `reinit()`- metóda je volaná ak zobrazovací systém potrebuje znovu inicializovať, napríklad pri zmene rozlíšenia obrazovky.
- `cleanup()`- táto metóda je volaná po celkovom ukončení aktualizačnej slučky. A dôjde tu k vyčisteniu systému.
- `stop()`- je volaná tesne pred úplným ukončením hlavnej aplikácie a spôsobí jej korektné ukončenie.

Tieto metódy implementujú hlavnú funkcionalitu aplikácie. Vďaka nim je možné sprístupniť vysokorýchlostnú hernú slučku, kde každá jej iterácia môže byť spracovaná tak rýchlo ako to môže CPU alebo GPU zvládnuť, prípadne po nastavení vertikálnej synchronizácie je možné rýchlosť iterácií obmedziť na nastavenú hodnotu. `SimpleApplication` umožňuje vytvárať aplikačné stavy, medzi ktorými sa môže aplikácia prepínať a každý z týchto stavov môže obsahovať svoju vlastnú slučku pre aktualizáciu `update()`, pracovať v OpenGL vlákne a obsahovať podporu pre tieňovanie objektov v scéne. Taktiež umožňuje do hry implementovať riadiace triedy, vďaka ktorým je možné bližšie špecifikovať jednotlivé chovanie objektov v grafe scény ako aj ich fyzikálne vlastnosti pre integráciu fyziky do aplikácie [24].

## 6.1 Zhrnutie

Android zaznamenal za posledné štyri roky taký obrovský vývoj a rozmach, že dosiahol polovičný trhový podiel v oblasti inteligentných telefónov a dominanciu na poli tabletov. A vďaka jeho multiplatformovým vlastnostiam a rôznym verziám je isté, že počet predaných zariadení, a tým pádom aj počet užívateľov, bude rásť. Preto sa v súčasnej dobe veľké množstvo programátorov zameriava práve na vývoj aplikácii pre platformu Android, pretože neustále sa rozrastajúca komunita používateľov tohto systému, či už prostredníctvom mobilných telefónov alebo tabletov, je pre nich dobrou zárukou pre uplatnenie svojich aplikácií na trhu. V dnešnej konkurencii je veľmi dôležité nielen správne načasovanie príchodu svojej aplikácie na trh, ale aj to ako bude daná aplikácia vyzerať

po vizuálnej a programovej stránke a veľmi podstatným faktorom je aj to, koľko času zaberie jej vývoj od návrhu až po nasadenie.

Tieto a všetky vyššie spomenuté fakty v tejto práci o jME a Android platforme ma viedli k tomu, aby sa predmetom tejto diplomovej práce stala práve možnosť využitia 3D herného enginu jMonkey Engine pri vývoji hernej aplikácie pre platformu Android. V tomto prepojení je ukrytý veľký potenciál do budúcnosti, pretože ponúka možnosť v pomerne krátkom čase vyvinúť kvalitnú a po vizuálnej ako aj programátorskej stránke dokonale konkurencieschopnú aplikáciu, ktorá môže po jej umiestnení na Google Play zaujať a osloviť veľké množstvo koncových užívateľov aj vďaka minimálnym nákladom na reklamu, čo môže v neposlednom rade priniesť vývojárom podarených a inovatívnych hier a aplikácií nemalý zisk.

Preto je potrebné aplikáciu podať v atraktívnej forme pre užívateľa. A práve z tohto dôvodu budú do hry implementované všetky dôležité súčasti, ktoré by mala herná aplikácia v dnešnej dobe obsahovať. Počnúc úvodnou animáciou, jednoduchým, príjemným a intuitívnym užívateľským rozhraním, obrazovkou informujúcou o priebehu načítavania prostriedkov hry, externými 3D modelmi, zvukovými efektmi, aplikačnými stavmi, riadiacimi triedami, cez vizuálne efekty, jednoduchú fyziku až po prívetivé a nekomplikované ovládanie hry samotnej, čo je jedným z najdôležitejších faktorov, ktoré rozhodujú o tom či bude aplikácia úspešná alebo nie.

Koncept aplikácie bude pozostávať z niekoľko hlavných objektov ako statických, tak aj dynamických. Hlavným prvkom hry bude dynamický objekt – užívateľom ovládateľný futuristický motocykel, ktorý sa bude pohybovať po statickom objekte dráhy. Tá bude vymedzená hranicami, v rámci ktorých bude motocyklu dovolené pohybovať sa. Na dráhe budú umiestnené klasické prekážky, ktorým sa musí motocykel vyhýbať a bonusové prekážky, ktoré mu budú pomáhať. Počas pohybu po dráhe bude možné z motocyklu strieľať, čím bude možné odstraňovať tieto prekážky z cesty. Pre všetky tieto objekty bude integrovaná fyzika a možnosti vzájomnej interakcie. Aplikácia bude poskytovať pre užívateľa niekoľko úrovní s rôznou náročnosťou, kde táto náročnosť sa bude postupne zvyšovať s každou ďalšou dosiahnutou úrovňou. Náročnosť bude zvyšovaná prostredníctvom väčšieho počtu prekážok na dráhe, menšieho počtu bonusov, dlhšou dobou nabitia zbraňového systému a zvyšujúcou sa hodnotou skóre, ktoré bude potrebné nahrať pre dosiahnutie ďalšieho levelu.

Všetky tieto vyššie spomenuté súčasti pre platformu Android je možné implementovať vďaka prepracovanej hernej knižnici Java Monkey Engine spoločne s externými knižnicami, ktoré sú súčasťou jej balíčka, a to NiftyGUI pre užívateľské rozhranie a jBullet pre integráciu fyziky. Aj napriek tomu, že sa jedná o open source knižnice, ponúkajú širokú paletu možností pri vytváraní komplexných hier v jazyku Java, vďaka čomu sa tieto hry dajú vytvoriť pomerne rýchlo, jednoducho a hlavne efektívne.

# 7 Implementácia aplikácie

Táto kapitola sa venuje implementácii hernej aplikácie pre platformu Android. Popisuje štruktúru aplikácie, postup práce, vlastnosti jednotlivých elementov a ich dôležité prvky, integráciu fyziky, zvukového systému, užívateľského rozhrania, multi-dotykového ovládania, efektov a ďalších potrebných prvkov, ktoré sú potrebné pre vytvorenie vizuálne prívetivej a ľahko ovládateľnej aplikácie. Námety, na čo všetko je potrebné myslieť pri tvorbe hry a ako pri tom postupovať som čerpal z [25].

## 7.1 Štruktúra aplikácie

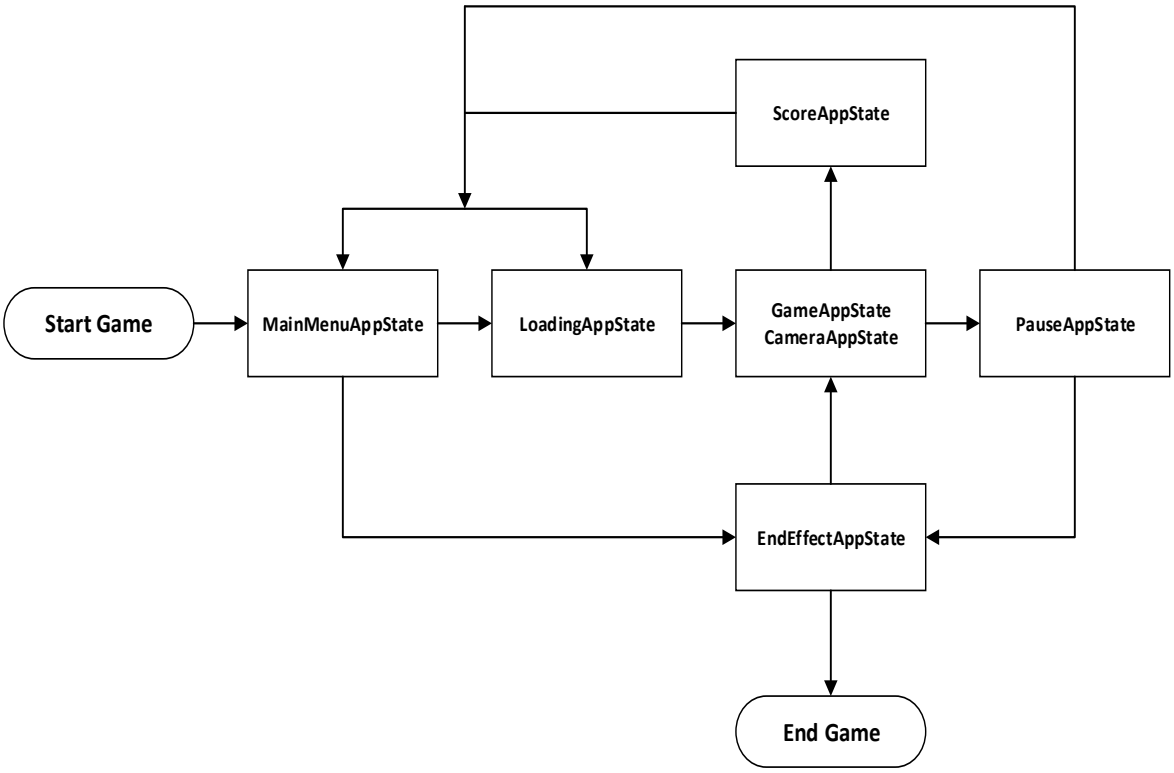
Celá hra pozostáva z viacerých balíkov a tried, tak aby odzrkadľovala logické členenie a funkcionalitu hry. Balíky sú rozdelené do troch hlavných skupín:

- `AppStates` – tento balík zoskupuje všetky aplikačné stavy, v ktorých sa môže hra nachádzať. Medzi týmito stavmi sa môže aplikácia prepínať čo umožňuje jej logické členenie na menšie časti. Každý aplikačný stav má priamy prístup k hlavným prvkom aplikácie, koreňovým uzlom, vstupom, zdrojom, fyzike a podobne. Hra je rozdelená na tieto aplikačné stavy:
  - `MainMenuAppState` – predstavuje stav pre hlavné užívateľské menu aplikácie.
  - `LoadingAppState` – tento stav slúži pre načítanie zdrojov aplikácie a zobrazovanie jeho priebehu na obrazovke.
  - `GameAppState` – hlavný stav, v ktorom dochádza k riadeniu hry samotnej.
  - `CameraAppState` – stav, ktorý kontroluje ovládanie kamery v aplikácii.
  - `PauseAppState` – predstavuje stav pre vyvolanie pauzového menu počas hry.
  - `ScoreAppState` – predstavuje stav, v ktorom dochádza k výpočtu nahratého počtu bodov po ukončení jednej úrovne hry alebo po jej celkovom ukončení.
  - `EndEffectAppState` – stav, ktorý poskytuje potrebný čas pre záverečné animácie po ukončení aplikácie, a tým zaručuje jej správne ukončenie.
- `Controllers` – tento balík zoskupuje všetky riadiace triedy, ktoré umožňujú lepšie špecifikovať chovanie vytvorených objektov v hre, ich fyzikálne vlastnosti a prispôsobiť ich presne podľa požiadaviek aplikácie.
  - `PlayerControl` – ovláda chovanie futuristického motocyklu v scéne.
  - `MissileControl` – riadi chovanie vystrelenej strely z motocyklu.
  - `FloorControl` – špecifikuje chovanie dráhy.

- `BorderControl` – ovláda chovanie hraníc umiestnených popri dráhe.
- `ObstacleControl` – riadi chovanie prekážok na dráhe.
- `HUDTextControl` – špecifikuje chovanie vytvoreného textu na obrazovke.

Týmto jednotlivým riadiacim triedam sa bude táto práca podrobnejšie venovať v nasledujúcich kapitolách.

- `Managers` – tento balík zoskupuje všetky triedy, ktoré riadia chovanie aplikácie samotnej. Patria tu triedy:
  - `GameStateManager` – riadi prepínanie medzi logickými stavmi aplikácie podľa toho, v ktorom aplikačnom stave sa aplikácia nachádza. Môže nadobúdať hodnoty: `MAIN_SCREEN`, `LOADING_LEVEL`, `RUNNING`, `PAUSED`, `FINISHED`, `SCORE`, `ENDEFFECT` a `NONE`.
  - `MyAppStateManager` – obsahuje všetky používané aplikačné stavy a poskytuje k nim prístup pre ich aktiváciu či deaktiváciu.
  - `SoundManager` – riadi všetky prvky zvukového systému v aplikácii.
  - `UIManager` – má na starosti korektné prepínanie medzi aplikačnými stavmi a aktiváciu k nim relevantných užívateľských menu.




**Obrázok 7.1: Vzťahy aplikačných stavov (Zdroj: vlastný)**

Hlavnou triedou v aplikácii je `RacingGame3D`, ktorá dedí z triedy `SimpleApplication`. Obsahuje metódu `main`, v ktorej dochádza pred samotným spustením aplikácie k nastaveniu

vykresľovacieho okna, verzie vykresľovacieho enginu LWJGL, vertikálnej synchronizácie a k vypnutiu zobrazovania úvodnej (splash) obrazovky s nastaveniami, pretože tá na platforme Android nemá podporu. Preto je potrebné všetko nastaviť ručne ešte pred samotným spustením aplikácie.

Po jej spustení metódou `start()` je následne volaná metóda `initialize()`, v ktorej sa zistia zadané nastavenia, na základe ktorých sa vytvorí zobrazovacia plocha aplikácie. Inicializujú sa hlavné prvky hry, a to vykresľovacie systémy (ViewPorts), do ktorých sú pripojené hlavné koreňové uzly `rootNode` a `guiNode`, kamera, vstupy a pripojí sa počiatočný aplikačný stav.

Ďalej sa volá metóda `simpleInitApp()`, v ktorej dochádza k vytvoreniu riadiacich tried z balíka `Managers`. Vytvára sa tu a následne pripája do manažéra stavov špecifický aplikačný stav `BulletAppState()`, ktorý má na starosti integráciu fyziky do aplikácie. Prebiehajú v ňom všetky skryté výpočty pre fyzikálny svet a všetky pripojené objekty do tohto stavu sa podriaďujú týmto fyzikálnym zákonom a sú nimi ovplyvňované.

Dochádza tu tiež k vytvoreniu `View Frustum` pre kameru, vďaka ktorému sa bližšie špecifikujú jej vykresľovacie vlastnosti. V tomto kroku je už aplikácia pripravená pre pripojenie prvého vlastného aplikačného stavu, ktorým je `MainMenuAppState`, pomocou metódy `attach()`. Každý aplikačný stav dedí triedu `AbstractAppState`, ktorá implementuje základné metódy pre správu jednotlivých aplikačných stavov [26]:

- `initialize()` – táto metóda je vždy volaná ako prvá po pripojení aplikačného stavu do manažéra stavov.
- `stateAttached()` – táto metóda je volaná po metóde `initialize()` a umožňuje prevádzať špecifické úpravy pre daný aplikačný stav.
- `stateDetached()` – táto metóda je volaná po odpojení stavu z manažéra stavov metódou `detach()`. Umožňuje napríklad odstránenie špecifických úprav pre daný stav.
- `setEnabled()` – dovoľuje dočasné aktivovanie alebo deaktivovanie stavu a prevádzanie akcií im prislúchajúcim.
- `update()` – aktualizuje prevádzané transformácie špecifických objektov v čase.
- `cleanup()` – je volaná po odpojení stavu z manažéra stavov a prevádza uvoľňovanie prostriedkov vytvorených v tomto stave.

Ďalšie aplikačné stavy sú do manažéra stavov pripájané v logickom poradí a o ich správnu aktiváciu, prípadne deaktiváciu sa stará trieda `UIManager`. V tejto triede dochádza aj k vytvoreniu a napojeniu užívateľského rozhrania NiftyGUI pre jednotlivé aplikačné stavy.

Po ukončení metódy `simpleInitApp()` sa zobrazí na obrazovke hlavné menu hry. Po stlačení tlačidla `start` dôjde k deaktivácii aplikačného stavu `MainMenuAppState` a k pripojeniu stavu pre načítanie hry `LoadingAppState` a hlavného herného stavu

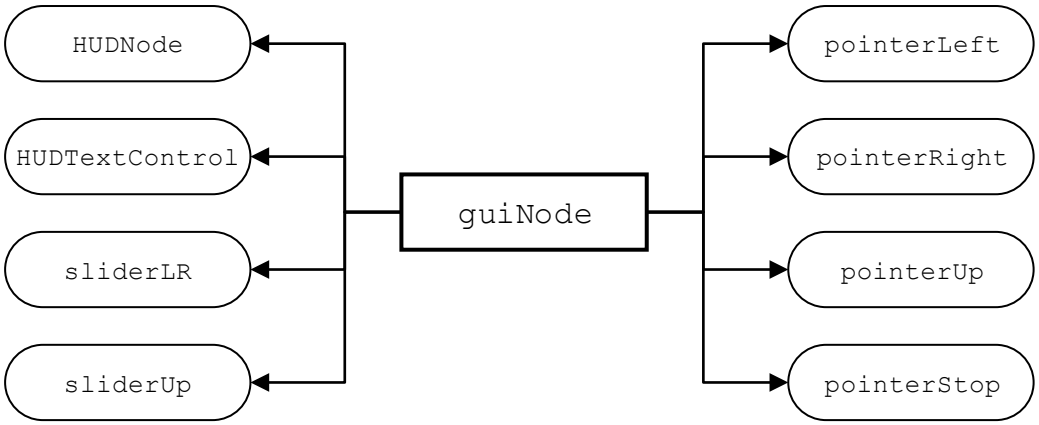
GameAppState. Pomocou manažéra GameStateManager sa nastaví aktívny stav pre načítanie hry LOADING_LEVEL, čím sa umožní renderovanie a aktualizácia iba potrebným prvkom hry. Na obrazovke je zobrazovaný priebeh načítania, zatiaľ čo v pozadí dochádza k samotnému načítavaniu (viď Obrázok 7.9: Načítavanie herných prostriedkov). Riadenie zobrazovania priebehu inicializácie objektov, modelov, textúr, zvukov a všetkých ostatných elementov vykresľovaných v scéne prebieha v stave GameAppState. K načítaniu dochádza v jeho metóde update(), kde s každým krokom iterácie tejto aktualizačnej slučky dochádza k postupnému načítaniu určitých herných prostriedkov. Po ukončení každého kroku načítania je odoslaná informácia do aplikačného stavu LoadingAppState, aby aktualizoval zobrazovaný priebeh na obrazovke.

Takéto postupné načítanie v krokoch je potrebné z dôvodu, aby bolo možné zobraziť priebeh načítania. Pretože metóda render(), ktorá slúži pre vykresľovanie elementov na obrazovku je volaná vždy až po každej iterácii aktualizačnej slučky. Takže ak by k načítaniu prostriedkov dochádzalo v jednom kroku, nebolo by možné zachytiť priebeh načítania hry, pretože by sa priebeh načítania neaktualizoval priebežne, ale naraz až po dokončení celého načítavania prostriedkov. Po ukončení načítania sa stav LoadingAppState automaticky deaktivuje a odpojí a aktivovaný ostane iba stav GameAppState. Pomocou manažéra GameStateManager sa nastaví aktívny stav pre hru RUNNING, čím sa umožní renderovanie a aktualizácia načítaných elementov v scéne, zvukového systému, užívateľského rozhrania, reakcií na vstupy a podobne.

Metóda cleanup() je volaná bezprostredne po ukončení hlavnej hernej slučky a má na starosti dôsledne uvoľnenie alokovaných prostriedkov z pamäte. Metóda stop() je volaná tesne pred ukončením aplikácie a slúži na korektné ukončenie aplikácie. Kvôli jej správnemu fungovaniu na platforme Android je potrebné v tejto metóde natvrdo zavolať System.exit(), pretože Android svoje bežiace aplikácie neukončuje, ale ich len uvedie do neaktívneho stavu, čo v prípade hernej aplikácie jME nie je žiaduce.

## 7.2 Koreňové uzly

jME 3 so sebou prináša nové názvy koreňových uzlov. V hre sú použité dva hlavné koreňové uzly, a to rootNode a guiNode. Všetky ostatné uzly, ktoré chceme, aby boli v hre viditeľné, je potrebné pripájať práve pod tieto dva koreňové uzly. Je dobré dodržovať určité konvencie, a to, že pod guiNode sa pripájajú iba elementy, ktoré chceme zobrazovať na obrazovke ako HUD (Head-Up-Display) pomocou metódy getGuiNode().attachChild(). Ostatné grafické elementy, ktoré reprezentujú už konkrétne objekty v hre sa pripájajú pod uzol rootNode metódou getRootNode().attachChild().

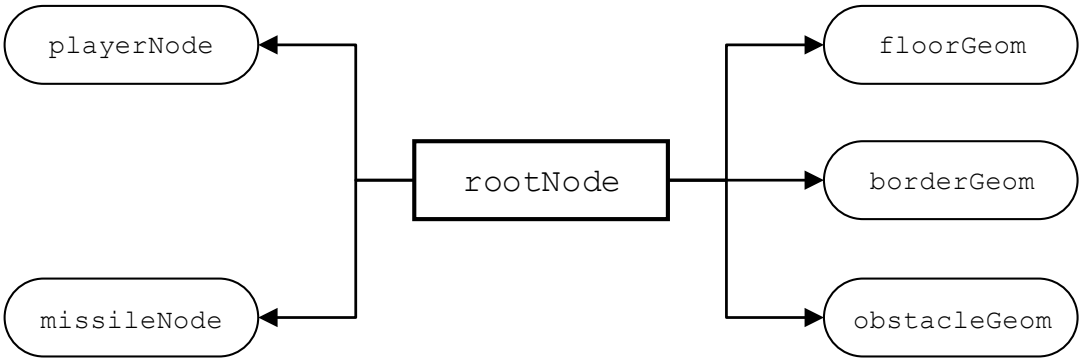



**Obrázok 7.2: Koreňový uzol `guiNode` (Zdroj: vlastný)**

Ako je vidieť na obrázku, ku koreňovému uzlu `guiNode` je pripojených niekoľko objektov. Tieto objekty sú buď priamo tvorené z elementov vytvorených pomocou jednoduchých 2D textov z triedy `com.jme3.font.BitmapText` a obrázkov z triedy `com.jme3.ui.Picture` alebo z uzlov, ktoré pod sebou zoskupujú ďalšie objekty pre ich lepšiu manipuláciu. Všetky GUI elementy sa inicializujú v metóde `initHUD()` v triede `GameAppState`.

Pod uzol `HUDNode` sú pripájané elementy, ktoré slúžia ako vizuálna reprezentácia informácií na obrazovke HUD. Informujú užívateľa o základných informáciách, o ktorých by mal mať prehľad pri hraní hry. Ako napríklad aktuálne dosiahnuté skóre, počet životov a o niektorých udalostiach, ku ktorým môže dôjsť v priebehu hry, napríklad ku kolízii, zobrazení informácie o dočasnom zlepšení nejakej hernej vlastnosti (Power UP) a podobne. Je tu pripojený ešte jeden funkčný element `pauseButton`, ktorý reprezentuje tlačidlo na vyvolávanie pauzového menu hry a jeden zoskupujúci uzol `shootButton`, ktorý pod sebou združuje všetky potrebné prvky pre vytvorenie tlačidla pre strieľanie.

Osobitnými prvkami pripojenými pod `guiNode` sú zobrazované texty vytvárané pomocou triedy `HUDTextControl`, ktorých pripájanie, respektíve odpájanie sa riadi automaticky a 2D obrázky reprezentujúce funkčné elementy, ktoré umožňujú vizualizovať ovládanie futuristického motocyklu. Tieto funkčné elementy sú už jednotlivo pripájané, respektíve odpájané, pretože ich zobrazovanie je nutné riadiť samostatne. Prvkom `sliderUp`, `sliderLR`, `pointerStop`, `pointerLeft`, `pointerRight`, `pointerUp` sa bude táto práca podrobnejšie venovať v kapitole 7.5 Multi-dotykové ovládanie a objektom triedy `HUDTextControl` v 7.6 HUDTextControl.




**Obrázok 7.3: Koreňový uzol `rootNode` (Zdroj: vlastný)**

Na uzol `rootNode` sú pripájane všetky ostatné uzly, ktoré už reprezentujú 3D objekty na scéne. Pre lepšiu čitateľnosť a hlavne ovládateľnosť zobrazovania elementov v scéne je dobré mať tieto objekty rozdelené do logických skupín. Ako je vidieť z obrázka uzol `playerNode` je hlavným uzlom pre objekt futuristického motocyklu a pre ďalšie uzly a objekty, ktoré s ním súvisia. Geometria `floorGeom` reprezentuje jednotlivý diel objektu dráhy, po ktorej sa motocykel pohybuje. Geometria `borderGeom` slúži pre vytvorenie hraníc dráhy. Geometria `obstacleGeom` predstavuje jednotlivú prekážku umiestnenú na dráhe. Uzol `missileNode` reprezentuje strelu vystrelenú z motocyklu a všetky objekty, ktoré s ňou súvisia.

Uzlom `playerNode` a `missileNode` a geometriám `floorGeom`, `borderGeom` a `obstacleGeom` sa bude táto práca bližšie venovať v kapitole 7.8 Fyzikálne objekty v hre.

### 7.2.1 Stavy vykresľovania

Pre uzol `rootNode` sú v hre vytvorené dva doplňujúce stavy vykresľovania. Prvým je `ZBufferState`, ktorý slúži k zobrazovaniu pixelov od nastavenej pozície kamery až do vzdialenosti, po ktorú je kamera schopná vidieť. Kamere sa nastavuje View Frustum počas inicializácie hry v metóde `simpleInitApp()`, a to nasledujúcim spôsobom.

```
float aspect = cam.getWidth() / cam.getHeight();
cam.setFrustumPerspective(45f, aspect, 1f, 150f);
```

Kde premenná `aspect` predstavuje pomer šírky a výšky zobrazovacej plochy kamery v závislosti od veľkosti vykresľovacieho okna. Kamere sa následne nastaví uhol, pomer `aspect` a vzdialenosti, od a po ktorú je kamera schopná vykresľovať objekty v scéne. Elementy v scéne za touto hranicou nebudú vykresľované, pokiaľ neprídu bližšie k objektu kamery. Tento stav prispieva aj k estetickejšiemu vykresľovaniu objektov. Objekty sa na scéne neobjavujú nárazovo a okamžite, ale vyzerajú akoby sa ich jednotlivé časti postupne objavovali na horizonte.

Druhým spôsobom vykresľovania je `CullState`, ktorý spôsobí, že odvrátené časti objektov nebudú zobrazované. To sa docieľuje nastavením vlastností materiálu, ktorý sa priraďuje jednotlivým objektom v scéne.

`getAdditionalRenderState().setFaceCullMode(FaceCullMode.Back)`, kde toto nastavenie je dobré používať aj v prípade, ak je potrebné objekt v scéne dočasne zneviditeľniť bez toho, aby ho bolo nutné odpájať od svojho rodičovského uzlu. Nastavením vlastnosti materiálu `FaceCullMode.FrontAndBack` sa zamedzí vykresľovaniu odvrátených a aj privrátených častí objektu.

Vďaka týmto spôsobom renderovania je zaručená väčšia plynulosť hry a systém tak nie je zbytočne zaťažovaný vykresľovaním nepotrebných častí scény.

### 7.2.2 Osvetlenie

K tomu, aby boli použité materiály, a teda objekty, ktoré majú nastavené tieto materiály viditeľné na scéne, je potrebné do scény pridať osvetlenie. K vytvoreniu svetla dochádza v metóde `initLights()`, kde je na celú scénu aplikované priame svetlo z triedy `DirectionalLight`, ktoré svieti na všetky objekty z rovnakého uhlu a v určitom smere. Tento smer sa nastavuje metódou `setDirection()`. Druhé svetlo v scéne sa vytvára z triedy `AmbientLight` a udáva globálny farebný odtieň materiálov. Na všetky objekty v scéne svieti rovnako a z každej strany. Nastavením `setColor()` sa nastaví jeho požadovaný farebný tón.

Vytvorené svetlá sú pripojené ku koreňovému uzlu `rootNode` pomocou metódy `getRootNode().addLight()`, čím sa zaobstará, že každý element v scéne, ktorý bude pripojený pod koreňový uzol `rootNode`, bude automaticky ovplyvnený týmito svetlami.

## 7.3 Zvukový systém

Zvukový systém v prostredí jME 3 postupom času prešiel viacerými dôležitejšími zmenami a svojou architektúrou poskytuje ešte jednoduchšiu možnosť implementácie ako v predošlých verziách jME. Prehrávanie zvukov je zaobstarávané jednoduchými metódami z triedy `com.jme3.audio.AudioNode`. Pre spustenie, zastavenie, prípadne pozastavenie zvuku slúžia metódy `play()`, `stop()` a `pause()`. Nová verzia jME 3 prináša so sebou novinku v podobe metódy `playInstance()`, ktorá umožňuje prehrávanie zvuku ako inštanciu zvukového uzlu, čo je pri prehrávaní špeciálnych zvukových efektov priam nevyhnutné, pretože dovoľuje prehrávať ten istý zvuk viackrát v scéne zároveň a bez ovplyvňovania sa navzájom či ukončenia predošlých inštancií toho istého zvuku. Preto sú všetky zvuky v hre reprezentujúce špeciálne efekty prehrávané práve touto metódou. Vďaka tejto novej metóde sa odstránili chyby pri prehrávaní zvukov z prechádzajúcich verzií jME, kde bolo vždy potrebné pred spustením zvuku manuálne overovať, či už daný zvuk nebol raz spustený. A ak bol, tak bolo nevyhnutné zvuk najprv zastaviť a následne opäť spustiť. Taktiež bolo potrebné aktualizovať zvukový systém po každej iterácii aktualizačnej slučky `update()`.

V novej verzii prebieha inicializácia a aktualizácia zvukového systému v rodičovskej triede `com.jme3.audio.AudioData`, ktorá obstaráva celé jeho fungovanie automaticky už od spustenia aplikácie. Zvukový systém podporuje načítanie dvoch formátov zvukového súboru, a to Ogg Vorbis audio (`.ogg`) alebo PCM Wave (`.wav`). Oba formáty majú svoje výhody a nevýhody. Kvôli platforme Android sú v aplikácii použité súbory typu `.wav`, aj keď sú z hľadiska miesta na disku menej úsporné. Formát `.ogg` spôsoboval nežiaduce vedľajšie účinky ako zhoršenú plynulosť hry pri prehrávaní zvukov alebo ich neprehratie. To je spôsobené tým, že na platforme Android je tento formát prevádzaný do pamäte modulačnou metódou pulznej kódovej modulácie PCM (Pulse Code Modulation) [27]. Pri formáte `.wav` sa tieto chyby nevyskytovali, keďže sa jedná o bezkompresný formát PCM.

Použité zvuky v aplikácii boli prevzaté zo stránky `http://www.freesound.org`. Konverzie, úpravy a vytváranie niektorých zvukových súborov boli prevádzané pomocou externého programu Fruity Loops Studio 9, ktorý pri práci so zvukom ponúka veľkú škálu nastavení a efektov.

Riadenie zvukového systému, inicializácia zvukových súborov a ich manažovanie v rámci aplikácie má na starosti trieda `SoundManager`. Tá po vytvorení objektu tejto triedy v metóde `simpleInitApp()` v hlavnej triede `RacingGame3D` inicializuje načítanie zvukových súborov do aplikácie pomocou metódy `initSound()`. Pre vytvorenie zvukového uzlu je volaná metóda `AudioNode(getAssetManager(), "path/to/sound_file", false)`, kde prvý parameter predstavuje manažéra prostriedkov aplikácie, druhý fyzickú cestu k požadovanému zvukovému súboru a tretí parameter nastavuje, či sa má daný zvukový súbor priamo streamovať z disku alebo nie. Zvuky sú načítavané v závislosti od svojej dĺžky. Krátke zvukové súbory sú celé načítavané do pamäte a väčšinou sa jedná o zvuky predstavujúce špeciálne efekty. Dlhšie súbory sú streamované priamo z disku a reprezentujú hudbu, ktorá sa prehráva počas behu aplikácie.

Vytvoreným zvukom je následne potrebné nastaviť určité parametre. Pomocou `setLooping(true)` sa nastavuje možnosť opakujúceho sa prehrávania zvuku po jeho dohratí bez nutnosti ho opäť spúšťať. Väčšine zvukov predstavujúcich špeciálne efekty je automatické opakovanie zakázané. Výnimku tvorí iba zvuk motora futuristického motocyklu. Hlasitosť je nastavená metódou `setVolume(1f)`, čo je prednastavená hodnota. Pre vytvorenie efektu priestorového zvuku slúži `setPositional(true)`. K tomuto parametru sa viaže nastavenie pozície zvukového uzlu v priestore scény pomocou metódy `setLocalTranslation(X,Y,Z)`, kde `X`,`Y`,`Z` predstavujú súradnice v jednotlivých osiach. U dynamických zvukoch, kde je vyžadované, aby sa pohybovali spoločne so svojim rodičovským uzlom sa namiesto pripojenia uzlu ku koreňovému uzlu `rootNode` pripájajú priamo k objektu, s ktorým sa majú po scéne pohybovať. Na pripojenie zvukových súborov slúži obdobná metóda ako pri ostatných uzloch `attachChild()`. Posledná vec, ktorú je potrebné urobiť pre dosiahnutie efektu priestorového zvuku je, že zvukový systém je potrebné pohybovať v rámci scény spoločne s nejakým objektom

k čomu slúži metóda `getListener().setLocation()`. Tým sa nasimulujú uši užívateľa a najvhodnejšou objektom v tomto prípade je pozícia motocyklu v scéne.

V hre je konkrétne implementovaných niekoľko zvukových súborov. Jeden zvuk, ktorý slúži ako hudba do pozadia hry a viacero špeciálnych zvukových efektov, ktoré sú spustené pri nejakej udalosti. Napríklad pri načítaní úvodného menu hry, interakcii tlačidiel menu, kolízii motocyklu s prekážkami, prípadne po kolízii s bonusmi, vystrelení a dopade strely, záverečnej explózii motocyklu, ak došlo k zníženiu počtu životov na nulu, prípadne počas pohybu motocyklu. Zvuk pre akceleráciu má nastavenú možnosť opakovať sa. To je z dôvodu docielenia zvuku motora, ktorý pracuje v určitých otáčkach a v závislosti od rýchlosti motocykla sa tieto otáčky zvyšujú, či znižujú. Na simuláciu tohto efektu slúži metóda `setPitch(1f)`, ktorá dokáže počas behu aplikácie nastavovať výšky prehrávaného zvuku. Žiaľ táto metóda nie je dostupná na platforme Android a nastavovanie výšok nemá na prehrávanie zvuku žiadny efekt. Preto bolo potrebné pre zvuk motora vytvoriť viac samostatných zvukových súborov s rozdielnymi výškami a postupným prehrávaním a zastavovaním odpovedajúcich zvukov simulovať efekt zrýchľujúceho sa alebo spomaľujúceho sa zvuku motora.

Platforma Android v súčasnej dobe neumožňuje plnú zvukovú podporu pre svoje aplikácie kvôli reštrikciám API, čo negatívne vplýva na možnosti plnohodnotnej práce so zvukom.

## 7.4 Grafické užívateľské rozhranie

Grafické užívateľské rozhranie (GUI – Graphics User Interface) je jedným z najdôležitejších faktorov pri vytváraní hry. Je potrebné klásť doraz na jeho jednoduchosť, komplexnosť a v neposlednom rade aj na jeho estetickú výpovednú hodnotu.

Celý systém grafického užívateľského rozhrania a jeho logiku v rámci aplikácie má na starosti trieda `UIManager`. Tá obsahuje metódy, ktoré slúžia pre riadenie jeho chovania a pre korektné prepínanie medzi jeho jednotlivými časťami. Pripájaním, odpájaním, aktiváciou alebo deaktiváciou jednotlivých aplikačných stavov vyvoláva špecifické metódy potrebné pre správne fungovanie užívateľského rozhrania. To by sa dalo z hľadiska implementácie rozdeliť na dve hlavné časti:

- HUD – implementované pomocou natívnych elementov knižnice jME.
- NiftyGUI užívateľské menu – implementované pomocou externej knižnice NiftyGUI.

### 7.4.1 Vlastné typy písma

Java Monkey Engine SDK umožňuje programátorom využiť možnosť vytvorenia vlastného typu písma použitého v aplikácii. Slúži k tomu plugin nazvaný *Font Creator* [28], ktorý zo systémových typov písma dokáže vytvoriť vlastný formát pre použitie v aplikácii.

Vlastné typy systémového písma boli prevzaté zo stránky `http://www.dafont.com` a následne upravované prostredníctvom externého grafického nástroja Adobe Photoshop CS5, aby sa docielila požadovaná farba a rovnaká šírka jednotlivých znakov. *Font Creator* vygeneruje dva súbory. Prvý súbor s koncovkou `.png` predstavuje grafickú reprezentáciu písma.

**Obrázok 7.4: Grafická reprezentácia typu písma (Zdroj: vlastný)**

Druhý súbor s koncovkou `.fnt` predstavuje špecifický textový formát pre jME, ktorý v sebe nesie údaje o veľkostiach, medzerách, odsadeniach, hraniciach a ďalších vlastnostiach jednotlivých znakov reprezentujúcich písmo. Tento súbor je potrebné nahrať do manažéra prostriedkov aplikácie pomocou metódy `getAssetManager().loadFont("PATH/TO/FONTS/customFont.fnt")` a metódou `myApp.setHUDFont(customFont)` nastaviť typ písma, ktoré sa bude používať pre bitmapové texty v rámci aplikácie. To by malo zaručovať ich kompatibilitu, vysokú prispôsobiteľnosť a ich rýchlejšie generovanie.

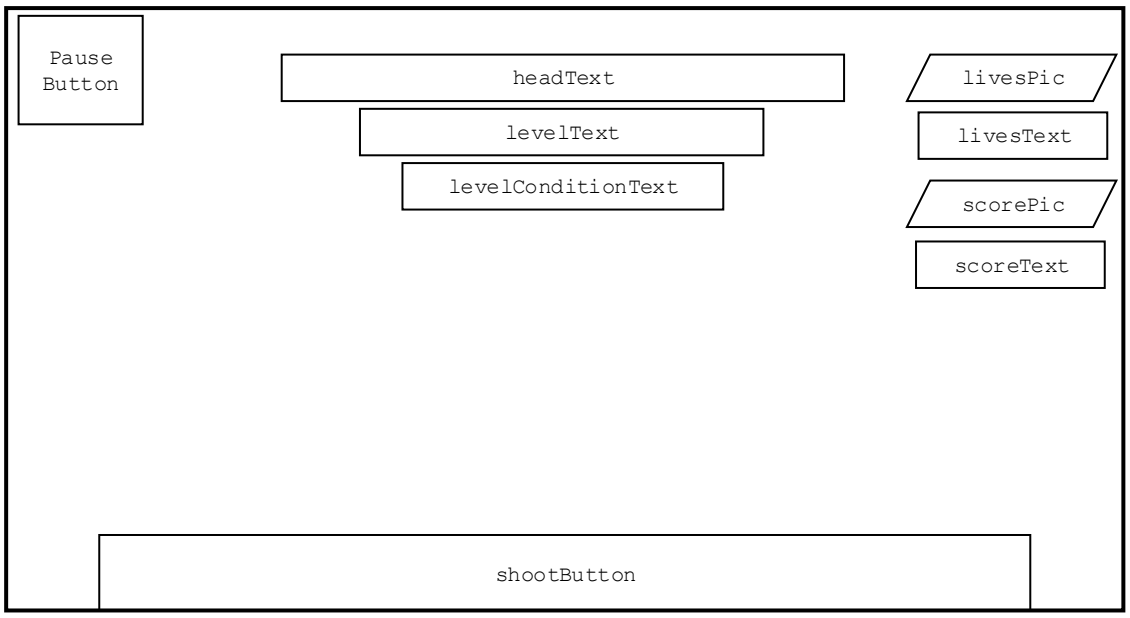
Takto vytvorené vlastné typy písma sú použité v aplikácii v oboch častiach grafického užívateľského rozhrania ako v HUD, tak aj v užívateľských menu vytvorených pomocou NiftyGUI.

### 7.4.2 HUD

Prvú časť predstavujú dvojrozmerné elementy, ktoré sú zobrazované na obrazovke. Slúžia predovšetkým ako vizuálna reprezentácia informácii na HUD, ktoré poskytujú užívateľovi základné informácie, o ktorých by mal mať prehľad počas hrania hry. HUD je tvorený z jednoduchých 2D textov z triedy `com.jme3.font.BitmapText` a z 2D obrázkov z triedy `com.jme3.ui.Picture`, ktoré sa vytvárajú v metóde `initHUD()` v triede `GameAppState`.

Tieto elementy sú pripájané pod uzol `HUDNode`, čo je hlavný uzol pre zobrazovanie HUD na obrazovke. Tento uzol sa pripája k alebo odpája z koreňového uzlu `guiNode` v závislosti na tom, či je hra aktívna alebo nie. Pripojenie, a s tým zároveň zviditeľnenie svojich potomkov prebieha pomocou metódy `getGuiNode().attachChild()`. Tým, že sú tieto elementy zoskupené

pod týmto uzlom, dochádza k ich automatickému zobrazeniu alebo odstráneniu z obrazovky v závislosti na stave rodičovského uzlu. To umožňuje jednoduchšie ovládanie zobrazovania týchto elementov, keďže nie je potrebné ich jednotlivé pripájanie, respektíve odpájanie.




**Obrázok 7.5: Rozmiestnenie HUD elementov (Zdroj: vlastný)**

Ako je vidieť na obrázku, rozmiestnenie jednotlivých HUD elementov je také, aby čo najmenej prekážalo pri hraní a pritom prinášalo potrebné informácie. To je docielené aj vďaka transparentnosti jednotlivých elementov.

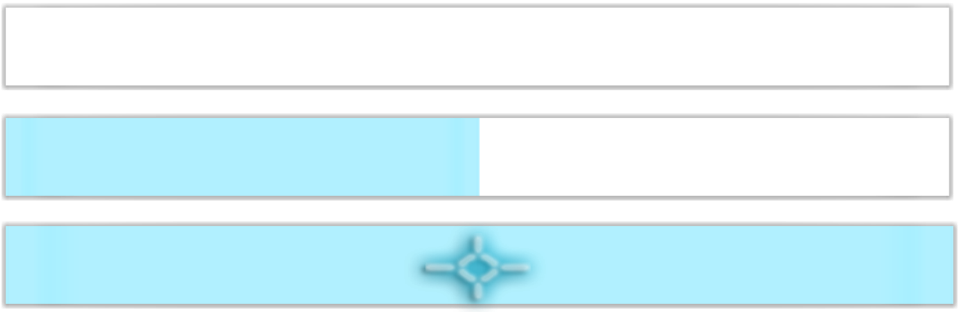
V pravom hornom rohu sa nachádzajú elementy `livesPic` a `scorePic`, ktoré reprezentujú 2D obrázky na obrazovke a slúžia ako popis pre elementy `livesText` a `scoreText`. Tie zobrazujú na obrazovke konkrétne informácie o aktuálnom počte zostávajúcich životov a o aktuálne dosiahnutom skóre. Tieto elementy sú 2D texty, ktoré sa s každou iteráciou hernej slučky `update()` menia tak, aby odzrkadľovali vždy aktuálne informácie.

Hlavným textom umiestneným na obrazovke je 2D text `headText`. Ten sa dočasne zobrazuje v strede obrazovky počas nejakej udalosti v hre. Napríklad po kolízii motocyklu s bonusom, počas celej doby trvania módu nezraniteľnosti, keď sa vypíše text „*invincible!!!*" a podobne. Texty `levelText` a `levelConditionText` sa zobrazujú pod hlavným textom a poskytujú informáciu o aktuálnom hernom leveli, v ktorom sa hráč nachádza a o podmienkach, ktoré je potrebné splniť pre jeho úspešné ukončenie.

V ľavom hornom rohu je umiestnený obrázok `pauseButton`, ktorý po dotyku vyvolá zastavenie aplikačného stavu `GameAppState` a zapne stav `PauseMenuAppState` pre zobrazenie pauzového menu hry.

V spodnej časti obrazovky sa nachádza uzol `shootButton`, ktorý v sebe zoskupuje viacero obrázkov a slúži ako tlačidlo pre strieľanie z futuristického motocyklu. Skladá sa z viacerých častí,

a to z pravej `shootButtonR`, ľavej `shootButtonL`, hornej a dolnej hranice `shootButtonM` a z vnútra `shootButtonInside`. Tieto časti je potom možné zväčšovať, prípadne zmenšovať bez straty kvality a pri zachovaní rovnakej proporcionality v závislosti od rozlíšenia obrazovky. Zmena veľkosti vnútra tlačidla slúži pre docielenie efektu zobrazovania priebehu času, kedy bude možné znovu vystreliť, pretože opätovné vystrelenie je možné až po uplynutí určitého času, ktorý sa predlžuje s každou ďalšou dosiahnutou úrovňou hry. Na znamenie, že je motocykel opäť pripravený vystreliť, sa v strede tlačidla zobrazí značka `shootButtonMark`. Kvôli lepšiemu vizuálnemu efektu je tlačidlo pre strieľanie implementované práve týmto spôsobom.



**Obrázok 7.6: Stavy tlačidla `shootButton` (Zdroj: vlastný)**

Špecifickými prvkami HUD sú elementy pre vizualizáciu ovládania pohybu motocyklu. Skladajú sa z bežcov `sliderLR` a `sliderUp` a ukazovateľov v rámci týchto bežcov `pointerLeft`, `pointerRight`, `pointerStop` a `pointerUp`. Funkčnosti týchto prvkov sa bude podrobnejšie venovať kapitola 7.5 Multi-dotykové ovládanie.

**Obrázok 7.7: Vizualizácia ovládania pohybu motocyklu (Zdroj: vlastný)**

#### 7.4.2.1 Rozmiestnenie HUD elementov

Všetky elementy v rámci HUD majú nastavenú relatívnu pozíciu. Sú rozmiestňované tak, aby spĺňali konvencie pre zachovanie rovnakého rozloženia na rôznych typoch obrazoviek a s rôznou hodnotou ich rozlíšenia. K tomu slúži metóda `setLocalTranslation(X,Y,D)`, ktorá funguje trochu inak ako metóda s tým istým názvom, ktorá je používaná pri umiestňovaní uzlov, prípadne geometrií

v scéne. Parametre `X` a `Y` predstavujú obdobne rozmiestnenie v `x`-ovej a `y`-ovej ose na obrazovke. Ľavý dolný roh predstavuje súradnice `[0,0]` a pravý horný roh zase `[W,H]`, kde `W` predstavuje šírku a `H` výšku obrazovky. Rozdielom je parameter `D`, ktorý udáva hĺbku textu zobrazovaného na obrazovke. To znamená, že bitmapové texty s rozdielnou hĺbkou sa budú prekrývať v špecifikovanom poradí na základe tohto parametra.

### 7.4.3 NiftyGUI užívateľské menu

Všetky grafické užívateľské menu sú implementované pomocou externej knižnice NiftyGUI [29, 30]. Je to Java knižnica, ktorá podporuje vytváranie interaktívneho užívateľského rozhrania pre hry a podobné aplikácie. Využíva LWJGL pre OpenGL vykresľovanie. Vytvorené GUI môže byť buď celé uložené v XML súboroch alebo celé naprogramované priamo pomocou jazyka Java. Java kód je použitý aj pri reakciách na generované udalosti vzniknuté interakciou užívateľa s GUI. Vďaka tomu poskytuje možnosť jednoducho modifikovať jednotlivé grafické elementy počas behu aplikácie. Tie sú vytvárané pomocou vstavaných metód pre vytváranie textu, obrázkov, panelov a sú poziciované v rámci svojej rodičovskej vrstvy. Priebeh vytvárania grafických elementov v podstate spočíva v ich umiestňovaní v rámci vytvorených vrstiev. NiftyGUI poskytuje okrem typických komponent ako tlačidlá, rámce, zoznamy, textové oblasti a iné, aj možnosť vytvárať rôzne efekty, animácie, nadväzovanie rôznych reakcií na udalosti a podobne. Všetky tieto možnosti sú v súčasnosti veľmi potrebné pre vybudovanie komplexného GUI rozhrania na vysokej úrovni a pri zachovaní vysokej efektivity pri jeho implementácii. Medzi jeho ďalšie výhody patria:

- Poskytuje vysoký výkon a minimálne pamäťové nároky.
- Podporuje hlavné OpenGL väzby pre Javu (JOGL, LWJGL).
- Jednoduchá integrácia do existujúcich vykresľovacích systémov ako sú jME 3, jME 2, LWJGL, JOGL, Slick2D a dokonca aj do Java2D.
- Vysoká voľnosť pri vytváraní GUI rozhraní.

Pripojenie tejto knižnice do vykresľovacieho systému jME prebieha v triede `UIManager`. Tá v konštruktore po svojom vytvorení v metóde `simpleInitApp()` v hlavnej triede `RacingGame3D` inicializuje NiftyGUI volaním metódy `initializeNifty()`. V nej dochádza k vytvoreniu väzby medzi jME a NiftyGUI vytvorením vykresľovacieho okna pomocou metódy

```
NiftyJmeDisplay niftyDisplay = new NiftyJmeDisplay
                        (
                              getAssetManager(),
                              getInputManager(),
                              getAudioRenderer(),
                              getGuiViewPort()
                        );
```

, ktorá prepojí NiftyGUI display s triedami jME pre manažment herných prostriedkov, manažment vstupov, zvukovým správcom a vykresľovacím systémom jME.

Do vykresľovacieho systému sa vloží takto vytvorený display pomocou metódy `getGuiViewPort().addProcessor(niftyDisplay)`. Po inicializácii obrazovky je ešte potrebné následne vytvoriť objekt triedy `de.lessvoid.nifty.Nifty nifty = niftyDisplay.getNifty()`, do ktorého priradíme inicializovaný `Nifty` systém. Je potrebné, aby tento objekt bol dostupný zo všetkých tried používaných v aplikácii, pretože prostredníctvom neho je umožnený prístup k vykresľovaciemu `Nifty` systému, ktorý dovoľuje pohodlne pracovať s prostriedkami, ktoré tento systém ponúka pri vytváraní a obsluhe grafického užívateľského rozhrania.

Pre vytvorenie užívateľského menu je potrebné po vytvorení `Nifty` objektu vytvoriť obrazovky `screens`, ktoré budú reprezentovať jednotlivé časti GUI. Pomocou metódy `ScreenBuilder("start")` sa vytvorí `Nifty` obrazovka s jednoznačným identifikátorom `start`. Pomocou tohto identifikátora je potom možné pristupovať k jednotlivo vytvoreným obrazovkám, prípadne sa medzi nimi prepínať, či vykonávať určité úlohy im vlastné. V každej obrazovke je potrebné špecifikovať ovládač `controller(MainMenuAppState)`, ktorý prislúcha jednotlivým aplikačným stavom v závislosti od toho, v ktorom bola `Nifty` obrazovka vytvorená. Ďalej je potrebné vytvoriť vrstvu `layer(new LayerBuilder("background"))`, v ktorej sa potom vytvárajú ďalšie grafické elementy prislúchajúce pod túto vrstvu. Počet vrstiev nie je špecifikovaný a je im dovolené sa prekrývať, čím je možné dosahovať rôzne efekty.

Vytvorené obrazovky potom môžu predstavovať jednotlivé užívateľské menu, HUD alebo napríklad aj úvodnú a záverečnú animáciu hry. Fantázii sa medze nekladú a dá sa povedať, že funkčnosť a vzhľad vytvorených `Nifty` obrazoviek sú limitované schopnosťami programátora. Všetky `Nifty` obrazovky v aplikácií sú natívne naprogramované v Jave bez použitia XML súborov.

NiftyGUI má vlastného správcu vstupov prepojeného s manažérom vstupov jME, ktorý dokáže reagovať na vstupy a vykonávať špecifické reakcie im prislúchajúce. To sa docieli pomocou implementovania metódy `interactXY()` pre daný grafický element, kde za `XY` sa môžu dosadiť rôzne hodnoty v závislosti od toho, akú reakciu požadujeme. Napríklad pre reakciu na kliknutie myší je to `interactOnClick(param())`. Metóda môže obsahovať aj parameter, ktorý špecifikuje metódu, ktorá bude volaná ako výsledok interakcie.

Vynikajúcou pomôckou pre odlaďovanie skrytých chýb vzniknutých pri vytváraní užívateľských menu pomocou NiftyGUI, prípadne pre lepšiu a presnejšiu manipuláciu s ich elementami je metóda `setDebugOptionPanelColors(true)`, ktorá prekrýva každý vytvorený grafický element inou farbou a umožňuje tak jednoduchšie rozlíšenie hraníc medzi jednotlivými prvkami vo vykresľovanom okne.

Na rozdiel od prvej časti grafického užívateľského rozhrania HUD, je celá druhá časť vytvorená práve pomocou NiftyGUI a dá sa rozdeliť na viacero logických podčastí:

- Úvodná animácia.
- Hlavné menu hry.
- Načítavanie herných prostriedkov.
- Pauzové menu hry.
- Zvukové menu.
- Menu pre nahrané skóre.
- Záverečná animácia.

Po spustení aplikácie sa pred prvým príchodom do hlavného menu hry zobrazí na obrazovke úvodná animácia. Tá pozostáva z dvoch rovnako veľkých častí, ktoré pomocou efektu pohybu, zmeny veľkosti, prekrývania a pozvoľného miznutia vytvárajú úvodné logo hry.

Po doznení úvodnej animácie sa zobrazí hlavné menu hry. To pozostáva zo 4 tlačidiel vytvorených pomocou metódy `TextBuilder()`, ktorým je pridelený vlastný typ písma, šírka, výška, zarovnanie v rámci rodičovského elementu a efekty pri zobrazení a ukončení zobrazovania `Nifty` obrazovky. Ako je vidieť na obrázku, tlačidlo `Start` spúšťa hru. Tlačidlo `HowTo` prepína obrazovku z hlavného menu na obrazovku, ktorá podáva informácie o ovládaní hry. `Credits` zase prepína na obrazovku, ktorá zobrazuje informácie o autorovi. Z týchto dvoch obrazoviek sa po stlačení tlačidla `Back` dostane užívateľ opäť do hlavného menu. Posledné tlačidlo `Exit` vyvolá obrazovku pre ukončenie, kde po stlačení tlačidla `YES` sa zavolajú metódy potrebné pre vyčistenie systému a ukončí sa celá aplikácie, prípadne po stlačení tlačidla `NO` sa vráti obrazovka pre hlavné menu.


**Obrázok 7.8: Hlavné menu hry (Zdroj: vlastný)**

Po stlačení tlačidla `start` dôjde k prepnutiu `Nifty` obrazovky na obrazovku pre načítanie herných prostriedkov.


**Obrázok 7.9: Načítavanie herných prostriedkov (Zdroj: vlastný)**

Pauzové menu hry sa zobrazí po stlačení tlačidla v pravom hornom rohu počas hrania hry. Tlačidlá v ňom sú vytvorené obdobne ako v hlavnom menu hry. Pri jeho zobrazení dochádza k pozastaveniu iterácie hernej slučky `update()` v aplikačnom stave `GameAppState` a povolí sa iterácia slučky `update()` v aplikačnom stave `PauseMenuAppState`, ktorá umožní iba aktualizácie vstupov. Tlačidlom `Resume` sa užívateľ dostane naspať do hry a obrátene sa povolia a zastavia iterácie slučiek `update()`. Po stlačení tlačidla `Restart` sa hra reštartuje a znova sa inicializujú herné prostriedky. Pri stlačení `Main menu` sa hra ukončí a prepne sa obrazovka do hlavného menu hry. Tlačidlo `Exit`, obdobne ako v hlavnom menu, aj tu spôsobí ukončenie celej aplikácie.


**Obrázok 7.10:Pauzové menu hry (Zdroj: vlastný)**

Zvukové menu je prístupné počas oboch obrazoviek pre hlavné a aj pauzové menu hry. Pozostáva z dvoch tlačidiel vytvorených pomocou metódy `ImageBuilder()`, ktorým je pridelený vlastný obrázok, šírka, výška, zarovnanie v rámci rodičovského elementu a efekty pri zobrazení a ukončení

zobrazovania obrazovky. Tlačidlá `music` a `sfx` povoľujú, prípadne zakazujú prehrávanie hudby alebo špeciálnych zvukových efektov v aplikácii.


**Obrázok 7.11:Zvukové menu hry (Zdroj: vlastný)**

Menu pre nahrané skóre sa zobrazí buď po úspešnom dokončení aktuálneho levelu alebo po jeho neúspešnom ukončení. Vytvorené menu zobrazuje informácie ohľadom počtu bodov z nazbieraných bonusov, zostrelených prekážok, prejdenej vzdialenosti, zasiahnutých prekážok motocyklom a celkového nahraného skóre.


**Obrázok 7.12: Menu pre nahrané skóre (Zdroj: vlastný)**

Posledná logická časť predstavuje `Nifty` obrazovku, ktorá je volaná vždy, keď je potrebné, aby sa úspešne dokončili vizuálne efekty volané pri prepínaní obrazoviek, prípadne pri celkovom ukončení aplikácie. Táto obrazovka sa nachádza v aplikačnom stave `EndEffectAppState` a zapína sa spoločne s týmto stavom. Po jeho pripojení do manažéra aplikačných stavov a aktivácii pomocou metódy `enableEndEffectAppState()` volanej v triede `UIManager`, dôjde k povoleniu jeho iteračnej slučky `update()`. V nej beží časovač, vďaka ktorému sa oddiaľuje prepnutie do druhého požadovaného aplikačného stavu, prípadne ukončenie aplikácie, a tým umožňuje korektné dokončenie vizuálnych efektov na prvej obrazovke, prípadne záverečnej animácie pred ukončením aplikácie.

## 7.5 Multi-dotykové ovládanie

Z dôvodu lepšej ovládateľnosti futuristického motocyklu na platforme Android je v hre implementované multi-dotykové ovládanie, čo znamená, že aplikácia dokáže reagovať na viacero súčasných vstupov, reprezentujúcich dotyky na obrazovke. Pre dosiahnutie tohto efektu je v hre potrebné implementovať triedu `com.jme3.input.controls.TouchListener`, ktorú dedí hlavný herný aplikačný stav `GameAppState`. Z tejto triedy je potom potrebné implementovať metódu `onTouch()`, ktorá je automaticky volaná vždy, keď dôjde k vstupu, čo znamená, že dôjde k dotyku na obrazovke. To, na ktoré udalosti bude táto metóda reagovať, sa určí pomocou namapovania udalostí do manažéra vstupov aplikácie pomocou metódy `getInputManager().addMapping(TOUCHALL, new TouchTrigger (TouchInput.ALL ))`, kde `TOUCHALL` predstavuje vlastný identifikátor v podobe reťazca, ku ktorému je potom naviazaná vstupná udalosť. `TouchTrigger()` obsahuje parameter, ktorý predstavuje, na ktoré všetky udalosti bude reagovať. `TouchInput.ALL` predstavuje všetky možné typy dotykov na obrazovke, ktoré môžu nastať. Rovnakým spôsobom je vytvorené mapovanie pre natívne tlačidlo platformy Android `Menu` na spodnej lište obrazovky za použitia `TouchInput.KEYCODE_MENU`, ktorá sa naviaže na vlastný identifikátor `TOUCHKEYMENU`.

K takto vytvoreným mapovaniam je potrebné pridať poslucháča udalostí `getInputManager().addListener(this, new String[] {TOUCHALL, TOUCHKEYMENU})`. Vďaka čomu bude zaručené automatické volanie metódy `onTouch()` vždy, keď dôjde k dotyku obrazovky, respektíve k dotyku tlačidla `Menu`. Prvý parameter predstavuje aktuálnu triedu, ktorá musí pre správne fungovanie dediť z triedy `TouchListener`. Druhý parameter reprezentuje pole reťazcov, ktoré obsahuje identifikátory, ku ktorým sú naviazané udalosti. Keďže má aplikácia reagovať na všetky typy dotykov, postačuje špecifikovať iba tieto dva identifikátory.

Po vytvorení mapovania a jeho nadviazania na poslucháča udalostí je aplikácia pripravená reagovať na vstupy z dotykov na obrazovke. Metóda `onTouch(String binding, TouchEvent evt, float tpf)` obsahuje tri parametre. Po vyvolaní udalosti vstupu, prvý parameter obsahuje názov identifikátora. V tomto prípade premenná `binding` bude obsahovať reťazec identifikátora `TOUCHALL` alebo `TOUCHKEYMENU`, v závislosti od toho či dôjde k dotyku na obrazovke alebo tlačidla `Menu` na spodnej lište obrazovky.

Druhým parametrom je objekt triedy `TouchEvent`, ktorý v sebe nesie všetky potrebné informácie ohľadom dotyku, respektíve dotykov na obrazovke. Napríklad súradnice dotyku na obrazovke sa zistia pomocou `evt.getX()`, `evt.getY()` pre `x`-ovú, respektíve `y`-ovú súradnicu. Metóda `evt.getPointerID()` vracia zase identifikátor určujúci poradie dotyku. Každý ďalší dotyk bude mať tento identifikátor zvýšený o jednotku, čím sa dá rozlišovať medzi

jednotlivými dotykmi na obrazovke, a tým vytvoriť podporu pre multi-dotykové ovládanie. Metóda `evt.getType()` vracia identifikátor pre prebiehajúci typ dotyku. Ten závisí od počtu prstov aktuálne sa dotýkajúcich obrazovky, tlaku, ťahu a rýchlosti dotyku.

Posledným tretím parametrom je `tpf` (`Time per Frame`), ktorý predstavuje čas potrebný na vykreslenie jedného snímku hry. Vďaka nemu je možné synchronizovať udalosti tak, aby prebiehali rovnako rýchlo na rôzne výkonných zariadeniach.

Celé multi-dotykové ovládanie motocyklu je implementované v prepínači (switch) v metóde `onTouch()`, ktorý vyvoláva určité reakcie v závislosti od toho, k akému typu dotyku došlo. K dispozícii je celá rada typov dotykov. Kompletný zoznam je k dispozícii cez programovú dokumentáciu Javadoc knižnice jME 3 [31]. V hre sú implementované tieto tri typy dotykov:

- `MOVE`
- `DOWN`
- `UP`

Typ dotyku `MOVE` nastáva vtedy, keď sa užívateľ pohybuje prstom po dotykovej obrazovke. V závislosti od toho, či sa pohybujeme prstom po ľavej alebo pravej časti obrazovky, vyvoláva zobrazenie určitých ovládacích prvkov pre riadenie pohybu motocyklu, a to bežcov a ukazovateľov na obrazovke HUD. Tie sa zobrazia po ich pripojení ku koreňovému uzlu `guiNode`.

Bežec `sliderLR` sa zobrazuje po dotyku v rôznej časti ľavej polovice obrazovky. Slúži pre zabáčanie motocyklu vpravo, respektíve vľavo v závislosti od pozície prstu na tomto bežci. Pre lepšiu vizuálnu interakciu sú na tomto bežci zobrazované doplnkové ukazovatele, ktoré informujú o smere bočenia. `PointerLeft` sa zobrazí pri bočení vľavo, `PointerRight` pri bočení vpravo a `PointerStop` sa zobrazuje vtedy, keď je prst v strede bežca a motocykel nezabáča ani do jednej zo strán. Taktiež sa ukazovateľ do tejto pozície vráti automaticky vždy, keď je prst zdvihnutý z obrazovky, aby došlo k vyrovnaniu naklonenia motocyklu na dráhe. V závislosti od vzdialenosti aktuálneho dotyku od toho počiatočného, ktorý určuje stred bežca, sa určuje veľkosť hybnej sily pre motocykel. Tá je obmedzená hranicami bežca a je vynásobená časom potrebným na vykreslenie dvoch za sebou idúcich snímkov. Je to kvôli synchronizácii rovnakej rýchlosti pohybu motocyklu na rôznych zariadeniach s rôznym výkonom. Tento údaj sa ďalej spracúva pri vypočítavaní pohybu futuristického motocyklu na scéne pomocou metódy `handlePlayerMovements()`.

Zobrazovanie bežca `sliderUp` funguje obdobne ako bežca `sliderLR` a slúži pre akceleráciu, prípadne brzdenie motocyklu. Zobrazuje sa po dotyku v rôznej časti pravej polovice obrazovky. Doplnkový ukazovateľ `PointerUp`, má v tomto prípade okrem vizuálnej funkcie aj funkčnú. Po zdvihnutí prstu z obrazovky sa uloží posledná pozícia ukazovateľa v rámci bežca. Táto pozícia potom slúži pri nasledujúcom dotyku, keď je bežec nacentrovaný na túto pozíciu v takom stave v akom bol tesne pred odpojením z HUD pri predchádzajúcom dotyku. To je z dôvodu, aby

nedochádzalo k nechcenému brzdeniu, prípadne akcelerácii pri nasledujúcom dotyku, keďže je možné tento bežec vyvolať v rôznej časti pravej obrazovky.

V závislosti od pozície aktuálneho dotyku od pozície, ktorá predstavuje hranicu medzi akceleráciou a brzdením, sa vypočítava hybná sila pre akceleráciu alebo brzdenie motocyklu. Hranica sa nachádza v jednej štvrtine bežca, aby sa umožnilo viac stupňov rýchlosti akcelerácie. Obdobne sa tieto údaje ďalej spracúvajú v metóde `handlePlayerMovements()`.

Typ dotyku `DOWN` je vyvolávaný pri dopade prstu na obrazovku a v aplikácií sa používa pre tlačidlo na vyvolanie pauzového menu hry v ľavom hornom rohu obrazovky, prípadne pri dotyku tlačidla `Menu` na spodnej lište obrazovky úplne vpravo. Tento typ dotyku však slúži predovšetkým pre tlačidlo umožňujúce strieľanie z motocyklu umiestneného v spodnej časti obrazovky.

Typ dotyku `UP` je vyvolaný pri uvoľnení prstu z obrazovky. Slúži na identifikáciu, kedy sa majú z obrazovky odstrániť jednotlivé bežce a ukazovatele v závislosti od toho, v ktorej časti obrazovky je prst zodvihnutý a pre uloženie pozície ukazovateľa `pointerUp` pre správne opätovné zobrazenie bežca `sliderUp` pri ďalšom dotyku. Identifikuje sa tu aj možnosť, či nedošlo k zodvihnutiu prstu v oblasti tlačidla pre strieľanie a ak k nemu došlo, tak sa nevykonávajú žiadne nasledujúce akcie pre bežce a ukazovatele. Ak by tomu tak nebolo, tak by pri každom vystrelení došlo k odstránení ovládacích prvkov motocyklu, a to samozrejme nie je žiaduce.

Takto implementovaný `TouchListener` dovoľuje potom reagovať aplikácii na viac vstupov súčasne, a tým maximalizovať ovládateľnosť motocyklu na dotykovej obrazovke.

## 7.6 HUDTextControl

Táto riadiaca trieda bola vytvorená z dôvodu lepšej informovanosti užívateľa o nárazových zmenách skóre a potreby automatizácie vytvárania, pohybu a odstraňovania týchto textov na obrazovke HUD. Tieto texty sa zobrazujú pri určitých udalostiach, ktoré môžu nastať počas hrania hry ako napríklad kolízie motocyklu s prekážkami, zobratie bonusu, kolízie vystrelených striel s prekážkami a podobne. Keď nastane udalosť, ktorá si vyžaduje nárazovú zmenu skóre, dôjde k zobrazeniu špecifického textu, ktorý sa môže líšiť v závislosti od typu udalosti. Tento text sa po vytvorení pre dosiahnutie lepšieho estetického efektu pohybuje od stredu obrazovky smerom nahor s časom. Po dosiahnutí maximálnej výšky, čo je hodnota rozlíšenia obrazovky pre výšku, sa text odstráni. Takýchto udalostí však môže nastať viacero súčasne.

**Obrázok 7.13: Paralelné textové polia na HUD (Zdroj: vlastný)**

Spôsob zobrazovania informatívnych textov popísaných vyššie v kapitole 7.4.2 HUD nie je dostačujúci pre docielenie efektu paralelne sa pohybujúcich textových polí po obrazovke. Pretože pre vytvorenie tohto efektu takýmto spôsobom, by bolo potrebné mať predpripravených niekoľko uzlov, ktoré by sa pripájali ku koreňovému uzlu `guiNode` a nastavovala by sa im pozícia v rámci obrazovky s každou iteráciou slučky `update()`. Keďže vopred nevieme koľko takýchto udalostí môže nastať, musel by byť počet uzlov dostatočné veľký, aby pokryl maximálny počet možných udalostí. Taktiež je potrebné brať v úvahu, že každý uzol by mohol v závislosti od typu udalosti niesť iný text, mať inú farbu a podobne. Preto by bolo potrebné spravovať zbytočne veľký počet uzlov, pracne ich pripájať, meniť ich pozíciu v rámci obrazovky a overovať podmienky odstránenia z HUD pre každý jeden uzol. Z hľadiska výkonnosti aplikácie na platforme Android by to malo značný dopad na zhoršenie jej plynulosti.

Trieda `HUDTextControl` poskytuje automatické spravovanie týchto uzlov a čo je najdôležitejšie, vytvorí iba toľko uzlov, koľko je potrebných, aby pokryli aktuálne prebiehajúce udalosti na obrazovke. Po každej udalosti, ktorá si vyžaduje zobrazenie tohto textu, dochádza k vytvoreniu objektu triedy `HUDTextControl`. V konštruktore tejto triedy potom dochádza k vytvoreniu textu s požadovanými parametrami a k jeho pripojeniu ku koreňovému uzlu pomocou metódy `createHUDText(hudFont, positive, score, textType)`. Prvý parameter `hudFont` predstavuje typ písma a jeho vlastnosti, ktoré sa priradia textu. Na základe hodnoty druhého parametra `positive` sa textu priradí farba, kde červená predstavuje negatívnu udalosť, napríklad stratenie jedného života a zelená pozitívnu udalosť, napríklad zobratie bonusu. Tretí parameter `score` priradzuje textu jeho hodnotu, čiže to, čo sa na obrazovke skutočne zobrazí, väčšinou je to práve hodnota výšky skóre, ktorá sa buď pripočíta alebo odpočíta z celkového skóre. Posledný parameter `textType` určuje o aký typ udalosti sa jedná. Či sa zobrazuje text po zasiahnutí prekážky, bonusu pre zvýšenie počtu životov alebo bonusu pre aktiváciu módu nezničiteľnosti.

Hlavnou metódou tejto triedy je `controlUpdate()`, ktorú je možné implementovať vďaka dedeniu z triedy `com.jme3.scene.control.AbstractControl`. Touto metódou je možné docieliť požadované chovanie vytvoreného textu v rámci HUD v čase. Tento text s každou iteráciou slučky `controlUpdate()` mení svoju pozíciu v `y`-ovej osi smerom nahor. Po dosiahnutí maximálnej výšky povolenej pozície textu na obrazovke, prípadne po uplynutí času povoleného pre zobrazovanie, dôjde k jeho automatickému odstráneniu z HUD a zároveň z koreňového uzlu `guiNode`. Tak isto sa automaticky odstráni aj samotný objekt triedy `HUDTextControl`, a to pomocou metódy `removeControl(this)`, čím sa značne obmedzí využívanie pamäťových prostriedkov.

Vďaka takémuto spôsobu implementácie tohto efektu je možné paralelne zobrazovať pohybujúce sa texty na HUD bez badateľných zmien výkonnosti aplikácie bežiacej na platforme Android.

## 7.7 Integrácia fyziky

Java Monkey Engine 3 má v sebe vstavanú podporu pre integráciu fyziky pomocou externej knižnice **jBullet** [32], ktorá vychádza z knižnice **Bullet** [33]. Táto integrácia sa dá docieliť prostredníctvom balíka `com.jme3.bullet`. Vďaka tejto knižnici jME 3 poskytuje kompletné, mierne upravené, ale plne zabalené **Bullet API**, ktoré používa natívne jME 3 matematické objekty, a to vektory, kvaternióny a iné ako vstupné, respektíve výstupné dáta. Sprístupňuje pôvodné `Bullet` objekty ako sú `RigidBodies`, `Constraints` (v jME 3 sa nazývajú `Joints`) a `CollisionShapes`. Hlavným objektom pre integráciu fyziky je fyzický svet `PhysicsSpace`, do ktorého je potrebné pripojiť všetky objekty, u ktorých vyžadujeme, aby mali aktivovanú fyziku. Je dokonca možné vytvoriť niekoľko fyzických svetov pre spustenie simultánnych nezávislých fyzických simulácií [34].

K vytvoreniu objektu fyzického sveta `PhysicsSpace` dochádza v metóde `simpleInitApp()`. Tu sa vytvára objekt triedy `com.jme3.bullet.BulletAppState`, ktorý predstavuje špecifický aplikačný stav `BulletAppState`. Ten je potom potrebné pre aktiváciu pripojiť k manažérovi aplikačných stavov. Vytvorený fyzický svet `PhysicsSpace` je prístupný pre všetky vytvorené aplikačné stavy v aplikácii prostredníctvom volania metódy `getBulletAppState().getPhysicsSpace()`.

Pre `PhysicsSpace` je potom dostupných niekoľko nastavení. Prvým základným je možnosť nastavenia pôsobenia gravitácie pomocou `setGravity(new Vector3f(0, -9.81f, 0))`, kde prednastavenou hodnotou v ose `y` je hodnota gravitačného zrýchlenia Zeme. Toto nastavenie bude automaticky nastavené každému objektu, ktorý bude do fyzického sveta pripojený. V aplikácií je ponechané toto prednastavené nastavenie. Druhým je možnosť nastavenia spôsobu súbehu aktualizačnej `update()` slučky. Knižnica jBullet zatiaľ neumožňuje GPU akceleráciu, avšak jME 3

svojou implementáciou umožňuje beh tohto aplikačného stavu v paralelnom vlákne pomocou nastavenia `setThreadingType(BulletAppState.ThreadingType.PARALLEL)`. Tretím základným nastavením je špecifikovanie presnosti vypočítavania fyzických kalkulácií pomocou metódy `setAccuracy(1f/60f)`. Čím väčšia je presnosť, čiže hodnota počtu výpočtov za sekundu, tým pomalšia bude aplikácia.

Aktualizačná slučka fyziky implementovanej pomocou knižnice jBullet beží na frekvencii 60 snímok za sekundu (FPS). Táto frekvencia nie je fixovaná v závislosti na aktuálnom počte FPS aplikácie. Ak je aktuálna hodnota FPS aplikácie vyššia než frekvencia aktualizácie fyziky, zobrazia sa interpolované pozície pre fyzické objekty scény. Ak je FPS aplikácie nižšie, do slučky pre aktualizáciu fyzického sveta sa vstúpi viackrát behom jedného snímku, aby sa tak dorátali vynechané kalkulácie. Aktualizácia a synchronizovanie fyzického stavu v rámci `BulletAppState` sa dejú nasledovne [34]:

1. Vypočítavanie kolízií (`BulletAppState.update()`).
2. Aktualizácia aplikácie (volanie hlavnej `update()` slučky a následne vedľajších `update()` slučiek v riadiacich triedach a aplikačných stavoch).
3. Synchronizácia a aplikácia fyziky v scéne (`updateLogicalState()`).
4. Vstup do slučky pre aktualizáciu fyziky (pred alebo paralelne podľa nastavenia s vykresľovaním `Application.render()`).

Takto vytvorený fyzický svet je potrebné naplniť objektmi, ktoré majú byť fyzikou ovplyvňované. V aplikácii sa nachádza niekoľko objektov, ktoré podliehajú fyzikálnym zákonom, a to futuristický motocykel, podlaha, ktorá predstavuje dráhu, po ktorej sa motocykel pohybuje, hranice podlahy, ktorými je vymedzený pohyb motocykla, prekážky a bonusy na dráhe a strela, predstavujúca výbušný projektil vystrelený z motocykla. Pre každý z týchto objektov je pre aktiváciu fyziky potrebné:

- Vytvoriť kolízny tvar (`CollisionShape`). Ten predstavuje zjednodušený tvar pôvodného tvaru objektu, na základe ktorého sa budú vypočítavať fyzikálne udalosti. Vďaka tomuto zjednodušenému tvaru sa dosiahne značného zrýchlenia simulácie fyziky oproti tomu, keby sa tieto výpočty prevádzali na pôvodnom tvare objektu, pretože tento tvar by mohol byť až príliš zložitý v závislosti od prepracovania modelu.
- Vytvoriť inštanciu riadiacej triedy `PhysicsControl` z kolízneho tvaru a hodnoty predstavujúcej hmotnosť objektu. Tieto triedy priamo dedia z tried `BulletPhysics` pre integráciu fyziky a sú odporúčanou cestou pre simuláciu fyziky v jME 3.
- Pripojiť objekt riadiacej triedy rodičovskému objektu. Tým sa docieli, že tento rodičovský objekt, jeho pohyb, rotácia a ďalšie vlastnosti sú ovládané prostredníctvom objektu riadiacej triedy. Tým sa zaručí neustály priebeh výpočtu simulácie fyziky na rodičovskom objekte.

- Pripojiť rodičovský objekt ku koreňovému uzlu `rootNode`, čím sa povolí vykresľovanie objektu v scéne.
- Pripojiť riadiaci objekt do `PhysicsSpace`, čím sa aktivujú fyzické vlastnosti tohto objektu.
- Iba pre objekty, ktoré majú mať implementované rozhranie na naslúchanie kolízií je nutné vytvoriť `PhysicsCollisionListener`. Pre objekty, u ktorých chceme dôslednejšie špecifikovať ich chovanie v rámci aktualizačnej slučky pre fyziku, je potrebné implementovať rozhranie `PhysicsTickListener`.

Objekty sa do fyzického sveta `PhysicsSpace` pripájajú pomocou metódy `add()`. Keďže fyzikálne objekty ostávajú vo fyzickom svete aj po odpojení ich rodičovského objektu zo scény, je nevyhnutné tieto fyzikálne objekty odpájať ručne. K tomu slúži metóda `remove()`.

# 7.8 Fyzikálne objekty v hre

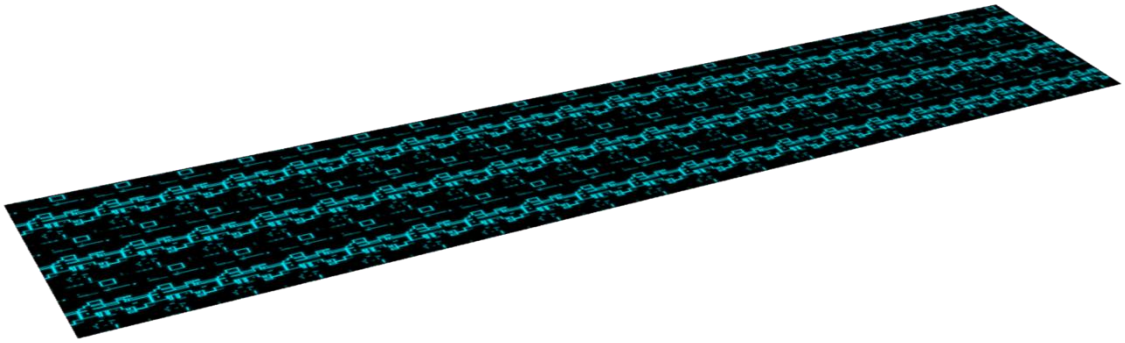
Fyzikálnym objektom v hre je potrebné pre ich zviditeľnenie v scéne priradiť nejaký materiál. Ten sa nastavuje pomocou metódy `setMaterial()`. Každý materiál obsahuje určité nastavenia, ktoré špecifikujú jeho vizuálne vlastnosti, a tým aj vizuálne vlastnosti objektu, ktorému bude materiál priradený. Ak má materiál niesť v sebe nejaký vzor, musí mu byť nastavená textúra pomocou metódy `setTexture()`.

Textúry pre všetky objekty v aplikácií sú načítavané prostredníctvom manažéra prostriedkov Asset Manager. Textúry sú tvorené z obrázkov vytvorených v externom programe Adobe Photoshop CS5 vo formáte `.png`. Podklady pre textúry v hre boli prevzaté zo stránky `http://www.filterforge.com`. Tie však prešli výraznými úpravami pre docielenie požadovaného vzhľadu objektov.

Po dodržaní týchto úvodných krokov je potom možné prejsť k integrácii fyziky pre tieto jednotlivé objekty podľa návodu uvedeného v kapitole 7.7 Integrácia fyziky.

## 7.8.1 Geometria floorGeom a borderGeom

Jedným z hlavných objektov v scéne je objekt dráhy. Ten je v aplikácií reprezentovaný geometriami `floorGeom` a `borderGeom`. Celá dráha v hre predstavuje statický objekt a je tvorená z menších dielov, ktoré na seba nadväzujú. Tieto jednotlivé časti sú tvorené natívnymi jME 3 objektmi. Je to hlavne z dôvodu, že objekt dráhy nevyžaduje zložitejšiu štruktúru a na jeho vytvorenie postačujú interné prostriedky, ktoré jME 3 poskytuje.



**Obrázok 7.14: Geometria `floorGeom` (Zdroj: vlastný)**

V hre sa stretneme so štyrmi farebnými odtieňmi dráhy. Podľa toho, v ktorom leveli sa hráč nachádza, má jednotlivý diel dráhy priradený iný materiál. Toto spektrum odtieňov dráh je ľahko rozšíriteľné, pretože k tomu postačuje iba vytvorenie a následné priradenie nového typu materiálu pre objekt dráhy.

U geometriách `borderGeom` tvoriacich hranice dráhy je požadované, aby v scéne plnili iba funkčnú a nie vizuálnu úlohu. Preto im je priradený materiál, ktorý ich v scéne zneviditeľňuje. To je docielené vyššie spomenutým nastavením, kde sa materiálu nastaví mód pre vykresľovanie `FaceCullMode.FrontAndBack`, tak aby nevykresľoval privrátené ani odvrátené trojuholníky.

#### 7.8.1.1 Integrácia fyziky

Po vytvorení objektov geometrie `floorGeom`, respektíve `borderGeom` s požadovanými rozmermi a nastavením im prislúchajúcim materiálom v metóde `initFloor()`, respektíve `initBorder()`, je možné pokračovať s integráciou fyziky pre tieto objekty.

Keďže dráha je reprezentovaná jednoduchým statickým objektom, nie je nutné jej vytvárať kolízny tvar ručne. Ten sa vytvorí automaticky pri vytvorení inštancie riadiacej triedy `FloorControl`, respektíve `BorderControl` a prispôsobí sa presne tvaru tohto objektu.

To sa docieli vďaka dedeniu triedy `RigidBodyControl`, ktorá zabezpečuje vytvorenie kolízneho tvaru a fyzického objektu samotného. Ďalej je nutné, aby trieda ešte dedila z triedy `PhysicsControl`, ktorá sprístupňuje ovládanie chovania fyzického objektu vo fyzickom svete. Parameter triedy `FloorControl(0)` predstavuje hodnotu hmotnosti, ktorá sa objektu priradí. Hodnota 0 požaduje od triedy `RigidBodyControl` vytvorenie statického objektu, ktorý bude mať absolútnu hmotnosť a nebude možné, aby jeho pozíciu ovplyvňovali ostatné fyzické objekty v scéne. Taktiež ostatným objektom nie je umožnené prechádzať cez objekty s nulovou hmotnosťou, čím sa docieli požadované vymedzenie plochy dráhy. Táto riadiaca trieda sa pripojí ku geometrii `floorGeom` pomocou metódy `addControl()`. Následne je geometria pripojená ku koreňovému uzlu `rootNode` a objekt riadiacej triedy do fyzického sveta `PhysicsSpace`. Tým sa docieli integrácia a aktivácia fyzických vlastností pre geometriu `floorGeom`. Obdobným spôsobom

dochádza k integrácii a aktivácii fyziky aj u objektov geometrie `borderGeom`, ktoré reprezentujú hranice dráhy.

Keďže objekty sa z fyzického sveta `PhysicsSpace` neodpájajú automaticky po odpojení ich rodičovského objektu zo scény, je nevyhnutné tieto fyzikálne objekty odpájať ručne. K zautomatizovaniu tohto procesu slúžia práve riadiace triedy `FloorControl` a `BorderControl`, ktoré vďaka dedičnosti sprístupňujú vlastnú aktualizačnú slučku `update()`. Tá slúži ku zisťovaniu podmienok odstránenia ako rodičovského objektu zo scény tak aj objektu riadiacej triedy z fyzického sveta. Tieto objekty sa automaticky odstránia v okamihu, keď sa motocykel nachádza na nasledujúcom vygenerovanom diely dráhy a prekročí jeho polovičnú hranicu dĺžky. Ak by sa tieto objekty neodpájali, dochádzalo by napríklad po reštarte alebo po zmene levelu hry ku neúmernému zaplňovaniu fyzického sveta, prípadne ku kolíziám s týmito už nepoužívanými objektmi, čo by malo výrazný dopad na plynulosť aplikácie na platforme Android.

#### 7.8.1.2 Generovanie dráhy

Ako je spomenuté vyššie celková dráha je poskladaná z viacerých jednotlivých dielov. Tie musia pre docielenie uceleného vizuálneho obrazu dráhy na seba dôkladne nadväzovať. K tomuto účelu slúži metóda `handleFloorGeneration()`, ktorá vytvorí ďalšie časti dráhy vždy, keď motocykel prekročí polovicu súčtu dĺžok naposledy pridaných dielov.

Pre napájanie jednotlivých dielov dráh za sebou je ešte nutné vedieť hodnotu aktuálneho súčtu dĺžok už vygenerovaných častí. Na základe tohto súčtu sa vypočíta presná pozícia pre ďalší diel dráhy. Po získaní tejto pozície je potom možné na koniec posledného pripojeného dielu dráhy pripojiť ďalšiu časť bez žiadnych vizuálnych nezrovnalostí. Ak by táto pozícia nebola vypočítaná presne, mohlo by dochádzať k nežiaducim javom ako je zasekávanie motocykla, prekážok, prepadávanie objektov cez dráhu a podobne.

Súčasne s tvorbou dielu dráhy dochádza obdobným spôsobom aj k tvorbe jeho hraníc, kde jediný rozdiel spočíva v umiestnení týchto hraníc s inou hodnotou `x`-ovej súradnice. Hranice sú generované z oboch strán dráhy, a tým vymedzujú možnosť pohybu motocyklu po dráhe v `x`-ovej súradnici.

Tieto jednotlivé časti dráhy spolu so svojimi hranicami nie sú vytvárané naraz počas jedného kroku aktualizačnej slučky `update()`, ale sú generované v niekoľkých krokoch po sebe tak, aby sa zamedzilo nežiaducemu spomaľovaniu aplikácie pri generovaní ďalších častí dráhy.

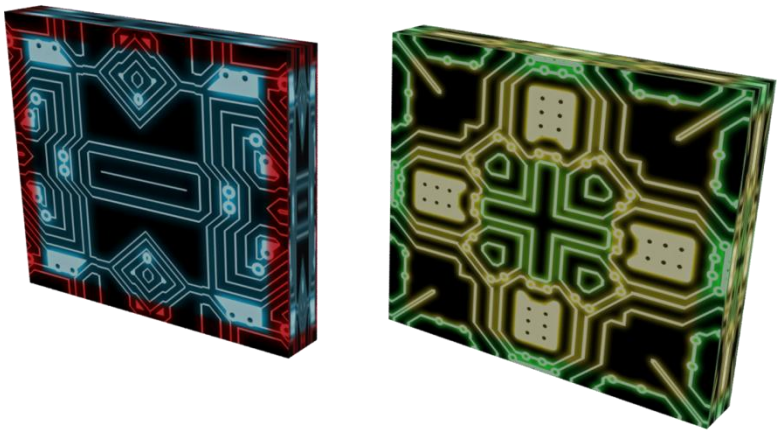
### 7.8.2 Geometria obstacleGeom

Geometria `obstacleGeom` reprezentuje objekt prekážky v scéne. V rámci hry na ňom dochádza k najlepšiemu rozpoznaniu chovania objektu v rámci fyzického sveta, pretože tento objekt najviac

interaguje so všetkými ostatnými objektmi v scéne, odráža sa od nich, spôsobuje kolízie a podobne. Geometria prekážky je taktiež tvorená pomocou natívnych jME 3 objektov.

#### 7.8.2.1 Typy prekážok

V hre sa stretneme s dvoma typmi prekážok. Prvým je prekážka, do ktorej keď narazí motocykel, tak dôjde k zníženiu počtu životov a k odčítaniu určitej hodnoty zo skóre. Druhým typom je prekážka vo forme bonusu, ktorá v sebe ukrýva vylepšenie nejakej vlastnosti objektu motocykla a po kolízií s týmto typom prekážky dôjde k pripísaniu určitej hodnoty ku skóre. Ako je vidieť na obrázku prekážky aj bonusy majú vlastný materiál, na základe ktorého sa rozpoznávajú v scéne.



**Obrázok 7.15: Prekážky v hre (Zdroj: vlastný)**

To, o ktorý typ prekážky sa jedná, sa určuje pomocou nasledujúceho výpočtu v metóde `addBrick()`. Pomocou metódy `FastMath.nextRandomInt(X, Y)` sa náhodne určí celé číslo z intervalu <X,Y>. Rozmedzie tohto intervalu sa určuje na základe levelu, v ktorom sa hráč nachádza. Na začiatku hry je pravdepodobnosť, že bude vygenerovaný bonus, zvolená na hodnotu 2 ku 15. S pribúdajúcim levelom sa táto pravdepodobnosť zmenšuje. Ak generátor určí číslo, ktoré prislúcha bonusovej prekážke, dôjde v podstate k nahradeniu jednej klasickej prekážky týmto bonusom a zabratiu jej pozície v scéne.

V hre sú implementované dva typy bonusov. Prvým je bonus, ktorý zvyšuje počet životov o jeden, avšak nikdy neprevýši maximálny počet životov, ktorý je vopred stanovený. Druhým je bonus, ktorý aktivuje mód nezraniteľnosti na určitú dobu. V tomto móde potom môže hráč narážať do prekážok, za čo mu budú pripisované kladné body do skóre.

Typ bonusu, ktorý v sebe prekážka ukrýva je generovaný náhodne a k jeho určeniu dochádza až pri vyhodnocovaní kolízie motocykla s bonusovou prekážkou. To je kvôli tomu, aby užívateľ vopred nevedel, o ktorý typ bonusu sa jedná. Spektrum možných bonusov je ľahko rozšíriteľné, čím vzniká možnosť pomerne jednoducho doimplementovať ich ďalšie typy do hry.

#### 7.8.2.2 Integrácia fyziky

Po vytvorení geometrie `obstacleGeom` s danými rozmermi a nastavením požadovaného materiálu v metóde `initObstacle()` je následne potrebné pre prekážku vytvoriť kolízny tvar, pretože sa bude jednať o dynamický objekt v scéne. Kolízny tvar je vytvorený z triedy `BoxCollisionShape` v tvare kvádra s rovnakými rozmermi ako geometria `obstacleGeom`. Ďalej je potrebné vytvoriť inštanciu riadiacej triedy pre prekážky `ObstacleControl`, ktorá tak isto dedí z tried `RigidBodyControl` a `PhysicsControl`. Rozdiel oproti vytváraniu riadiacej triedy pre `floorGeom` spočíva v tom, že objekt riadiacej triedy je pre prekážku vytváraný s viacerými parametrami `ObstacleControl(boxCollisionShape, type, 15f)`. Kde prvý parameter predstavuje vopred vytvorený kolízny tvar. Druhý parameter určuje, či sa jedná o klasickú alebo o bonusovú prekážku. A tretí parameter predstavuje hodnotu hmotnosti, ktorá bude danému objektu priradená. Čím má objekt väčšiu hmotnosť, tým je ťažšie s ním hýbať v rámci scény, a tým menej reaguje na kolízie s inými objektmi. Funguje to na obdobnom princípe ako je tomu v reálnom svete. Tento parameter určuje, že sa jedná o dynamický objekt, a preto je mu ešte potrebné nastaviť kĺzavý faktor pomocou metódy `setFriction()`. Čím väčší je tento faktor, tým väčší odpor kladie tento objekt pri kolízií s povrchom iného objektu, napríklad dráhy.

Následne už len postačuje pripojiť objekt riadiacej triedy ku geometrii, tú pripojiť ku koreňovému uzlu `rootNode` a objekt riadiacej triedy do fyzického sveta `PhysicsSpace`. Tým sa docieli integrácia a aktivácia fyzických vlastností pre geometriu `obstacleGeom`.

Obdobne aj tu má riadiaca trieda za úlohu vo vlastnej aktualizačnej slučke `update()`, mimo iné zisťovať podmienky automatického odstránenia objektu z fyzického sveta. Navyše sa tu riadi zmena polohy prekážky s každou iteráciou slučky. Táto zmena môže byť vyvolaná po kolízii s inými objektmi v scéne. Metóda `getMotionState().applyTransform(spatial)` zaručí správnu aktualizáciu transformácie objektu prekážky, a tým umožňuje užívateľovi pozorovať chovanie prekážky v rámci fyzikálneho systému.

Geometria `obstacleGeom` a objekt jej riadiacej triedy sa automaticky odstránia zo scény v okamihu, keď sa nachádzajú za motocyklom a mimo zorného poľa kamery, prípadne keď je prekážke nastavená premenná `removeObstacle`, ktorá slúži pre identifikáciu ručného odstránenia.

Najväčší prínos automatického odstraňovania fyzikálnych objektov zo scény, čo sa týka výkonnosti a plynulosti aplikácie je práve pri odstraňovaní objektov prekážok, pretože tieto objekty sú v scéne najpočetnejšie.

#### 7.8.2.3 Generovanie prekážok

Pri návrhu prekážok pre hru bol kladený dôraz na zlepšenie hrateľnosti hry, a preto sú objekty prekážok do tunela generované v náhodnom počte z určitého intervalu. Rozmedzie tohto intervalu

taktiež závisí od aktuálne dosiahnutého levelu a s pribúdajúcim levelom sa počet prekážok na dráhe zvyšuje, čím sa zaručuje zvyšujúca sa náročnosť hry.

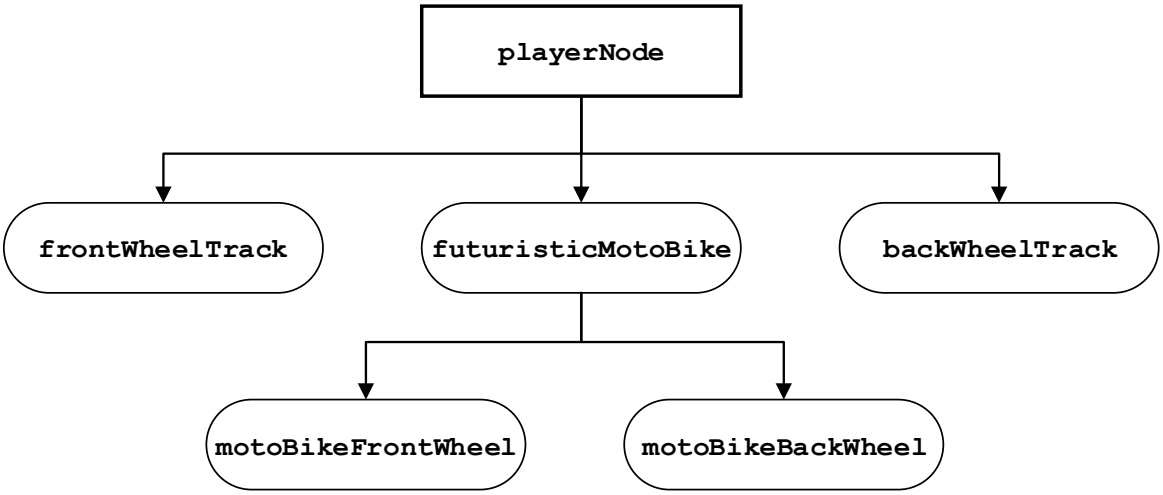
Ku generovaniu prekážok dochádza v metóde `handleWallGeneration()`, ktorá sa volá toľkokrát, koľko sa má vygenerovať prekážok do naposledy pridaného dielu dráhy. Každý diel dráhy vďaka náhodnému generovaniu obsahuje iný počet prekážok. Umiestnenie prekážky na dráhe je taktiež generované náhodne. Výpočet finálnych súradníc prekážky prebieha v metóde `computeWallPosition()` nasledujúcim spôsobom.

Najprv sa náhodne vyberie `x`-ová súradnica polohy prekážky z intervalu medzi dvoma `x`-ovými súradnicami reprezentujúcimi ľavú a pravú hranicu dráhy. Potom sa na základe náhodného počtu prekážok určených pre daný diel dráhy vypočítava `z`-ová súradnica. Prvá prekážka sa náhodne generuje z intervalu predstavujúceho začiatok aktuálneho dielu dráhy a koncového dielu dráhy deleným počtom prekážok pre daný diel dráhy. Druhej prekážke sa potom začiatočný interval posúva o hodnotu vypočítanej `z`-ovej súradnice predchádzajúcej prekážky plus hodnota hrúbky prekážky, aby nedošlo ku kolízii, či prekrývaniu prekážok navzájom. Ku koncovému intervalu sa potom pripočítava hodnota podielu koncovej súradnice dráhy a počtu prekážok. A takto ďalej sa posúvajú hranice intervalu pre generovanie ďalších prekážok v poradí. Tento algoritmus sa opakuje pri každom novo vygenerovanom diely dráhy.

Ku generovaniu prekážok nedochádza počas jedného kroku aktualizačnej slučky `update()` v aplikačnom stave `GameAppState`. Každá ďalšia prekážka je generovaná s odstupom 10 krokov. To je z dôvodu predchádzania neúmerného počtu kalkulácií kolízií a aktualizácií transformácií pri vytvorení a umiestnení nového fyzikálneho objektu do scény. Pretože každý nový objekt je bezprostredne po svojom pridaní v aktívnom móde, kým „nedosadne" na svoju pozíciu. Po dosadnutí sa objekt uvedie do stavu nečinnosti a aktivuje sa znovu, až keď dôjde k zmene jeho polohy. Vďaka tomu sa podarilo na platforme Android predísť nežiaducemu spomaleniu aplikácie počas generovania prekážok na dráhu.

### 7.8.3 Uzol playerNode

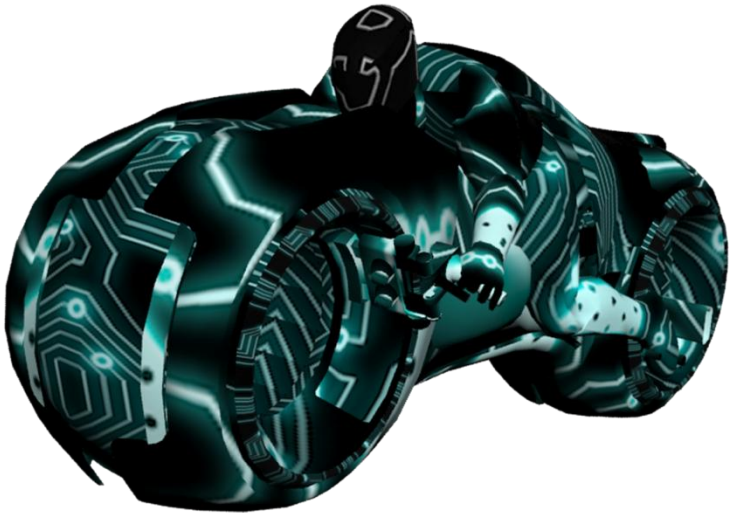
Uzol `playerNode` predstavuje rodičovský uzol, ktorý pod sebou zoskupuje všetky uzly, geometrie a objekty, z ktorých sa skladá výsledný model futuristického motocyklu.




**Obrázok 7.16: Hierarchia grafu uzlu `playerNode` (Zdroj: vlastný)**

Ako je vidieť na obrázku, na uzol `playerNode` sa napája uzol `futuristicMotoBike`, do ktorého sa priamo pripája model tela motocyklu. Pod tento uzol sa potom pripájajú uzly `motoBikeFrontWheel` a `motoBikeBackWheel`, do ktorých sa priamo pripájajú modely pre predné a zadné koleso motocykla. K uzlu `playerNode` sú ešte na pozícii predného a zadného kolesa pripojené dva objekty z triedy `ParticleEmitter` `frontWheelTrack` a `backWheelTrack`, ktoré slúžia pre zanechávanie stôp motocykla pri pohybe po dráhe.

Motocykel v hre predstavuje dynamický objekt a ako jediný objekt v hre reaguje na užívateľské vstupy z dotykov po obrazovke zariadenia. Model futuristického motocyklu bol prevzatý zo stránky `http://thefree3dmodels.com`. Tento model bolo potom potrebné rozdeliť na menšie časti, upraviť a vymodelovať ich do požadovanej vizuálnej podoby. Na tieto časti bolo potom nanesených viacero rôznych materiálov, aby sa docielilo zviditeľnenie a futuristický vzhľad týchto jednotlivých dielov motocyklu. Všetky úpravy boli prevedené prostredníctvom externého programu 3D Studio Max. Výsledný model bol potom spolu s materiálovou knižnicou vyexportovaný do formátu `.obj`. Tento formát bolo potom možné prekonvertovať spolu s priradeným materiálom do natívneho binárneho formátu `.j3o` pre jME 3.



**Obrázok 7.17: Model futuristického motocyklu (Zdroj: vlastný)**

#### 7.8.3.1 Integrácia fyziky

Po načítaní externých 3D modelov pre jednotlivé časti objektu futuristického motocyklu a ich priradeniu k prislúchajúcim uzlom v metóde `initPlayer()`, je potrebné vytvoriť kolízny tvar. Ten je nutné vytvárať iným spôsobom ako tomu bolo u geometriách `floorGeom` alebo `obstacleGeom`.

Pre zachovanie rýchlosti simulácie fyziky na objekte motocyklu je potrebné vytvoriť kolízny tvar z viacerých jednoduchších častí. Najprv sa vytvorí objekt triedy `CompoundCollisionShape`, ktorý bude tieto jednotlivé časti v sebe zoskupovať. Potom sa vytvoria dva kolízne tvary z triedy `SphereCollisionShape` v tvare gule, ktoré budú predstavovať predné a zadné koleso motocyklu a jeden kolízny tvar z triedy `BoxCollisionShape` v tvare kvádru, ktorý bude reprezentovať telo motocyklu. Tieto tvary je potrebné pripojiť na správnej pozícii do objektu triedy `CompoundCollisionShape` metódou `addChildShape()`. Na takto poskladanom kolíznom tvare sa následne bude vypočítavať simulácia fyziky ako by sa jednalo o jeden objekt.

jME 3 poskytuje viacero možností vytvárania kolíznych tvarov. Napríklad aj jeho automatizované vytvorenie presne podľa pôvodného tvaru modelu pomocou metódy z triedy `CollisionShapeFactory.createDynamicMeshShape()`. Avšak takýto kolízny tvar výrazne spomaľoval plynulosť aplikácie, a preto bol nahradený ručne poskladaným kolíznym tvarom, ktorý postačuje pre simuláciu fyziky a výpočet kolízií motocyklu v dostatočnej miere.

Následne je možné vytvoriť inštanciu riadiacej triedy pre objekt motocyklu `PlayerControl(compoundShape, 250f)`, ktorá dedí z tried `RigidBodyControl` a `PhysicsCollisionListener`. Kde prvý parameter predstavuje poskladaný kolízny tvar a druhý parameter určuje hmotnosť objektu, ktorá bude motocyklu priradená. Objekt motocyklu je na dráhe

najťažším pohybujúcim sa objektom. Na základe tejto hodnoty sa bude potom vypočítavať sila nárazu do prekážok. Tým, že je hmotnosť motocyklu podstatne väčšia ako hmotnosť prekážky, dôjde po náraze motocyklu do prekážky k jej odrazeniu. K tomu, aby sa zabránilo nežiaducim rotáciám motocyklu po viacnásobných nárazoch do prekážok, prípadne hraníc dráhy, je potrebné vypnúť faktor rotácie, a to metódou `setAngularFactor(0f)`. Detekcii kolízií pomocou triedy `PhysicsCollisionListener` sa bude táto práca venovať v kapitole 7.9 Kolízie.

Keďže sa jedná o pohybujúci sa objekt v scéne, je potrebné docieliť, aby sa tento objekt pohyboval po dráhe plynulo. K tomu slúži metóda `createWheels()` v riadiacej triede `PlayerControl`, v ktorej dôjde k vytvoreniu 4 neviditeľných kolies, ktoré budú pripojené pod kolízny tvar objektu motocykla a vytvoria štvorkolesový podvozok. Na ňom už bude umiestnený konkrétny model motocykla pomocou metódy `setLocalTranslation()`. Štvorkolesový z toho dôvodu, aby bola zaručená lepšia stabilita motocyklu na dráhe a aj pri jeho nakláňaní a kolíziách. Následne je kolesám ešte potrebné nastaviť kĺzavý faktor `setFrictionSlip(0.001f)` na úplné minimum, aby tieto kolesá produkovali čo najmenší odpor pri svojom pohybe po dráhe. Vďaka týmto kolesám, ktoré podliehajú fyzikálnym zákonom a môžu sa plynulo otáčať v smere pohybu motocykla sa docieli aj plynulosť jeho pohybu po dráhe.

Takto vytvorený objekt riadiacej triedy `PlayerControl` je potom pripojený k uzlu `playerNode`. Následne už len postačuje pripojiť tento uzol ku koreňovému uzlu `rootNode` a objekt riadiacej triedy do fyzického sveta `PhysicsSpace`. Celý objekt motocyklu je potom premiestnený na začiatok dráhy pomocou metódy pre umiestňovanie fyzikálnych objektov v scéne `setPhysicsLocation()`. Tým sa dokončí integrácia a aktivácia fyzických vlastností pre uzol `playerNode`, a teda celého objektu motocykla.

Obdobne aj tu má riadiaca trieda za úlohu vo vlastnej aktualizačnej slučke `update()`, aktualizovať zmenu polohy motocykla v scéne s každou iteráciou slučky pomocou metódy `getMotionState().applyTransform(spatial)`.

Keďže v každom leveli hry sa vytvorí iba jeden objekt riadiacej triedy `PlayerControl`, nie je potrebné obstarávať jeho automatické odstránenie, ale postačuje ho odstraňovať manuálne vždy buď po reštarte aplikácie alebo po zmene herného levelu. K odstráneniu dochádza v metóde `removefromPhysics()` vždy pred inicializáciou a načítaním nových objektov relevantných pre daný level.

#### 7.8.3.2 Transformácie futuristického motocyklu

Na objekt motocyklu v scéne sú aplikované nasledujúce transformácie:

- Pohyb motocyklu.
- Rotácia jednotlivých častí motocyklu.

K pohybu motocyklu dochádza na základe vypočítaných veľkostí hybných síl. Hodnoty týchto síl sa získavajú spôsobom, ktorý je popísaný v kapitole 7.5 Multi-dotykové ovládanie. Tieto údaje sa ďalej spracovávajú v metóde `handlePlayerMovements()`, a to nasledujúcim spôsobom.

Najprv sa získa vektor aktuálnej lineárnej rýchlosti metódou `getLinearVelocity()`. Pre získanie rýchlosti v ose `Z`, čiže pre pohyb motocyklu dopredu, je tento vektor potrebné vynásobiť pomocou metódy `multLocal(0,0,1)`, ktorá lokálne vynuluje vektor v ose `X` a `Y`. K `Z`-ovej hodnote vektoru sa potom lokálne pripočíta veľkosť hybnej sily pomocou metódy `addLocal(0,0,updowndistance)`, kde `updowndistance` uchováva hodnotu hybnej sily vypočítanej z pohybu ukazovateľa na bežci `sliderUp`. Takýmto spôsobom pripočítavania hybnej sily sa docieli efekt postupnej akcelerácie alebo postupného brzdenia motocyklu.

Obdobným spôsobom sa vypočíta vektor pre určenie rýchlosti v ose `X`, čiže pre zabáčanie motocyklu do strán. Akurát sa po získaní lineárnej rýchlosti vynásobí vektor tak, aby sa vynulovali osy `Y` a `Z` a k `X`-ovej hodnote vektora sa pripočíta hodnota hybnej sily vypočítanej z pohybu ukazovateľov po bežci `sliderLR`. Týmto sa docieli postupný pohyb motocyklu do strán.

Tieto výsledné vektory pre akceleráciu a zabáčanie sú obmedzené určitou hranicou maximálnej rýchlosti pre akceleráciu, respektíve zabáčanie. Ak dôjde k prekročeniu tejto hranice, od vektorov sa namiesto pripočítania odpočíta hodnota odpovedajúca naposledy pridanej hybnej sily v príslušných osiach pomocou metódy `subtractLocal()`, čím sa docieli neprekročenie a udržovanie maximálnej rýchlosti v oboch smeroch pohybu motocyklu.

Pre docielenie pohybu fyzikálneho objektu v rámci `PhysicsSpace` slúži niekoľko metód. Od aplikovania jednoduchého dočasného impulzu až po konzistentné priradenie rýchlosti. Pohyb motocyklu v hre sa nastavuje práve pomocou konzistentného priradenia hybnosti, k čomu slúži metóda `setLinearVelocity(moveDirection)`, kde parameter `moveDirection` predstavuje vektor, ktorý v sebe obsahuje veľkosti síl v jednotlivých osiach, a tým sa motocyklu priradí správna rýchlosť v požadovanom smere.

Toto nastavenie rýchlosti je aplikované na objekt riadiacej triedy `PlayerControl`. Tým, že je tento objekt pripojený pod uzol `playerNode` sa docieli, že všetci potomkovia tohto uzlu budú rovnako ovplyvnený týmto pohybom objektu riadiacej triedy. Výsledkom bude konzistentný pohyb všetkých častí motocyklu. Tento výpočet pre určenie veľkosti vektora pre pohyb motocyklu prebieha pri každej iterácii aktualizačnej slučky `update()`, čím sa zaručuje rýchla odozva pohybu motocyklu na vstupy z dotykov po obrazovke.

Druhou transformáciou objektu motocyklu je rotácia jeho jednotlivých častí pri určitých udalostiach.

Prvý typ rotácie nastáva pri udalosti bočenia motocyklu do strán. Pre lepšiu vizuálnu interakciu je aplikovaná rotácia na uzol `futuristicMotoBike`, ktorá spôsobuje nakláňanie jednotlivých častí motocyklu. Výpočet rotácie prebieha v metóde `handlePlayerLeaning()`.

Toto naklonenie sa úmerne vypočítavá vzhľadom k rýchlosti bočenia na základe rotačného uhlu, a to nasledujúcim spôsobom. Hodnota tohto rotačného uhlu sa akumuluje priebežne počas celého trvania bočenia motocyklu, ale nikdy nepresiahne maximálnu hodnotu, pretože by mohlo dôjsť k nežiaducemu preklopeniu motocyklu. V závislosti od toho či motocykel zabáča doľava alebo doprava je hodnota tohto rotačného uhlu buď kladná alebo záporná. Z tohto uhlu je potom potrebné vypočítať rotačný kvaternion pomocou metódy `rotQuat.fromAngles(0,0,angleZ)`, kde `angleZ` predstavuje hodnotu rotačného uhlu v osi `Z` a `rotQuat` už konkrétny rotačný kvaternion, z ktorého sa dopočíta výsledná rotácia, ktorá sa následne aplikuje na uzol `futuristicMotoBike` pomocou metódy `setLocalRotation(rotQuat)`. Po zodvihnutí prstu z obrazovky a sa motocykel automaticky navráti do pôvodnej polohy. Toto navrátenie sa vypočítava obdobným spôsobom z rotačného kvaterniónu, kde jediná zmena je v tom, že sa hodnota rotačného uhlu postupne znižuje na nulu.

Vďaka aplikácií rotácie na uzol `futuristicMotoBike` dochádza k automatickej rotácii aj jeho potomkov, čo sú obe kolesá a stopy, ktoré zanechávajú. Tento výpočet sa prevádza s každou iteráciou aktualizačnej slučky `update()`. Táto rotácia nemôže byť aplikovaná na hlavný rodičovský uzol `playerNode`, pretože by sa nakláňal aj podvozok, čo by mohlo spôsobiť nežiaduce zaseknutie motocyklu medzi jednotlivými dielmi dráhy, prípadne medzi jej hranicami.

Druhý typ rotácie, ktorá sa vypočítava, je rotácia oboch kolies pri pohybe motocyklu dopredu v osi `X` a vychyľovanie týchto kolies pri jeho bočení v ose `Y`. K výpočtu rotačných uhlov a rotačného kvaternionu dochádza obdobne ako pri výpočte nakláňania motocyklu s tým rozdielom, že rotačný kvaternión sa vypočítava z dvoch rotačných uhlov `fromAngles(angleX, -angleY, 0)`, kde `angleX` je rotačný uhol, ktorý určuje rotáciu kolies v smere pohybu motocyklu dopredu a vypočíta sa ako hodnota lineárnej rýchlosti v osi `X` krát čas potrebný na vykreslenie dvoch za sebou idúcich snímkov. To je z dôvodu, aby sa kolesá točili rovnako rýchlo na rôzne výkonných zariadeniach. Uhol `angleY` obsahuje hodnotu vypočítanú pri nakláňaní celého motocyklu, ale pri výpočte sa použije jeho opačná hodnota, aby sa zaručilo vychýlenie kolies do správnej strany.

#### 7.8.3.3 Kamera

Veľmi dôležitým faktorom, ktorý ovplyvňuje dojem z hry je zobrazovanie scény. Nájdenie vhodného pohľadu na scénu úzko súvisí s nastavením vlastností kamery. Knižnica jME 3 poskytuje niekoľko typov kamier, no najlepším riešením v tomto prípade je kamera vytvorená z triedy `ChaseCamera`, ktorá umožňuje automatické sledovanie zadaného objektu, ku ktorému je pripojená.

Pre objekt kamery je v aplikácii vytvorený vlastný aplikačný stav `CameraAppState`. V tomto stave dochádza k vytvoreniu objektu kamery ako takej. V konštruktore tejto triedy sú následne tejto kamere nastavené parametre, ktoré špecifikujú jej chovanie počas prenasledovania motocyklu. `ChaseCamera` sleduje objekt z tretieho pohľadu, čo poskytuje dobrý prehľad počas

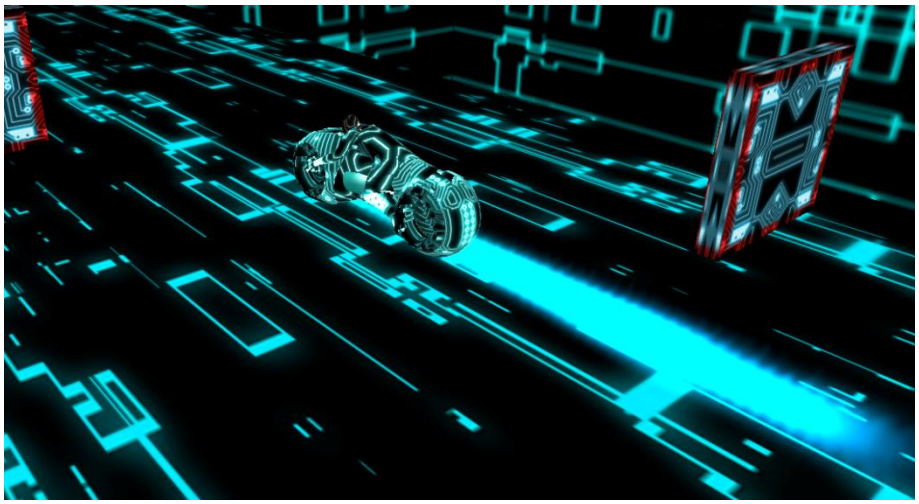
hrania. Základným nastavením je nastavenie hodnoty citlivosti kamery na pohyb prenasledovaného objektu `setChasingSensitivity()`, kde čím menšia je táto hodnota, tým pomalšie bude kamera reagovať na pohyb. S týmto nastavením súvisí nastavenie `setSmoothMotion()`, ktoré aktivuje plynulejšiu akceleráciu kamery, prípadne jej brzdenie. Ďalším jedným zo základných nastavení je nastavenie minimálnej a maximálnej povolenej vzdialenosti kamery od sledovaného objektu `setMinDistance()` a `setMaxDistance()`. Táto vzdialenosť závisí od rýchlosti, ktorou sa objekt pohybuje. Čím je tento objekt rýchlejší, tým sa vzdialenosť medzi kamerou a sledovaným objektom zväčšuje a naopak. To ako bude kamera rotovať okolo objektu, napríklad pri bočení do strán je možné nastaviť pomocou `setRotationSensitivity()`. Kamere je možné nastaviť ešte množstvo doplňujúcich nastavení ako je vertikálny, horizontálny uhol sledovania a mnoho ďalších.

Pre aktiváciu tejto kamery je potrebné najprv aktivovať tento aplikačný stav `CameraAppState` spoločne so stavom `GameAppState`, v ktorom má táto kamera fungovať. Aktivovanú kameru je potom ešte potrebné pripojiť k uzlu, ktorý bude objektom jej sledovania, čo bude uzol `playerNode`, ktorý predstavuje motocykel, a to pomocou metódy `playerNode.addControl(camAppState.getCamera())`.

Všetkými týmito nastaveniami parametrov pre kameru je možné jednoducho zlepšiť dojem užívateľa z pohľadu na objekty v scéne, a tým zaručene prispieť k atraktívnosti hry.

#### 7.8.3.4 Efekt stôp motocykla

Pre dosiahnutie efektu zanechávania stôp od kolies motocykla na dráhe sú k uzlu `playerNode` pripojené objekty triedy `ParticleEmitter` `frontWheelTrack` a `backWheelTrack`. Tie sú pripojené pod tento uzol z dôvodu, aby po strate života pri náraze na prekážku, kedy objekt motocykla po určitý čas na dráhe bliká, boli jeho stopy stále viditeľné.



**Obrázok 7.18: Efekt stôp kolies motocyklu (Zdroj: vlastný)**

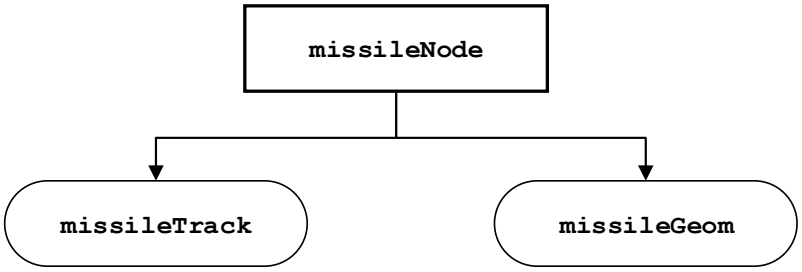
K vytvoreniu tohto efektu je potrebné vytvoriť objekty triedy `ParticleEmitter`, ktorá slúži na vytvorenie systému pre tvorbu čiastočiek, pomocou ktorých sa budú simulovať stopy od kolies na dráhe. `ParticleEmitter` vykresľuje sériu plochých ortogonálnych obrázkov, ktoré na sebe v podstate nesú materiál s určitou textúrou s alfa kanálom. Pomocou tejto textúry sa potom vytvára ich vzor a umožňuje meniť ich farbu presne podľa požiadaviek užívateľa. Pomocou tohto systému sa dajú docieliť vizuálne dokonalé efekty, avšak vyžaduje to určitú pracnosť pri nastavovaní správnych parametrov.

Po vytvorení týchto objektov je potrebné nastavenie chovania týchto čiastočiek v scéne. Základnými nastaveniami sú počiatočné a koncové farebné spektrum spoločne s alfa kanálom, ktorý nastavuje ich transparentnosť, ich počiatočná a koncová veľkosť, minimálna a maximálna životnosť, objekt, v ktorom sa budú náhodne generovať a smer ich vypúšťania z tohto objektu, pôsobenie gravitácie na tieto čiastočky, množstvo generovaných čiastočiek za sekundu a množstvo ďalších nastavení, ktoré sa dajú naštudovať z [35].

Pre aktiváciu a povolenie vykresľovania je nakoniec potrebné pripojiť objekty `frontWheelTrack` a `backWheelTrack` ku svojmu rodičovskému uzlu a nastaviť ich pozíciu pod objekty kolies motocykla, aby čo najvierohodnejšie simulovali zanechávanie ich stôp na dráhe.

### 7.8.4 Uzol missileNode

Pre väčší zážitok z hry je v aplikácii implementovaná možnosť streľby z futuristického motocykla. K tomu slúži uzol `shootButton` reprezentujúci tlačidlo v spodnej časti obrazovky. Po jeho stlačení, respektíve dotyku, dôjde k vytvoreniu objektu strely pomocou metódy `createMissile()`.




**Obrázok 7.19: Hierarchia grafu uzlu `missileNode` (Zdroj: vlastný)**

Ako je vidieť na obrázku, uzol `missileNode` predstavuje rodičovský uzol, ktorý pod sebou zoskupuje geometriu `missileGeom`, ktorá reprezentuje objekt vystrelenej strely a objekt triedy `ParticleEmmiter missileTrack`, ktorý slúži pre dosiahnutie efektu zanechávania stopy počas letu strely. Tento efekt sa docieli rovnakým spôsobom ako vytvorenie efektu zanechávania stôp od kolies motocyklu. Viď obrázok 7.19 Efekt stôp kolies motocyklu.

#### 7.8.4.1 Integrácia fyziky

Po vytvorení geometrie `missileGeom` s danými rozmermi a nastavením požadovaného materiálu je potrebné získať hranice strely v tvare kocky pomocou metódy `getWorldBound()` a tie rozšíriť násobením vektora na tvar kvádru, kde predĺžený tvar bude v ose `Z`. To z dôvodu, aby bolo možné včas detekovať jej kolízie s inými objektmi, pretože z týchto hraníc sa neskôr vytvorí kolízny tvar.

Ten je pre strelu vytvorený z triedy `BoxCollisionShape` v tvare kvádra s rovnakými rozmermi ako hranice jej geometrie. Následne je potrebné z tohto kolízneho tvaru vytvoriť inštanciu riadiacej triedy `MissileControl(missileCollisionShape, 20f)`, ktorá dedí z tried `RigidBodyControl`, `PhysicsCollisionListener` a `PhysicsTickListener`. Druhý parameter, ktorý predstavuje hmotnosť strely je nastavený tak, aby po náraze do prekážky dokázala na ňu preniesť dostatočnú kinetickú silu, ktorá spôsobí jej pohyb v scéne. Detekcii kolízií pomocou tried `PhysicsCollisionListener` a `PhysicsTickListener` sa bude táto práca venovať v kapitole 7.9 Kolízie.

Po vytvorení fyzikálneho objektu strely je potrebné nastaviť jej správne umiestnenie, keďže strela bude vystrelená z objektu motocykla. K tomu taktiež poslúžia vyššie spomínané rozšírené hranice vypočítané z geometrie strely. Pretože okrem umiestnenia pozície strely na pozíciu motocykla v scéne je ešte potrebné, aby strela bola posunutá o rozmery jej hraníc, a to z dôvodu, aby nedochádzalo ku kolízii medzi strelou a motocyklom hneď po jej vystrelení. Pozícia motocyklu sa získa z polohy objektu riadiacej triedy pomocou `getPhysicsLocation()`. Tak isto je ešte potrebné nastaviť smer strely tak, aby odpovedala rotácii motocykla. Tá sa získa pomocou metódy `getPhysicsRotation()`. Rýchlosť strely je nastavená na 75-násobok vektora smeru, v ktorom bude strela vystrelená a k nej sa ešte pripočíta aktuálna rýchlosť motocykla. Po získaní týchto údajov a ich prepočítaní do spoločného vektora je potom možné strelu vystreliť v požadovanom smere a s požadovanou rýchlosťou pomocou metódy pre pohyb fyzikálnych objektov `setLinearVelocity()`.

Následne už len postačuje pripojiť objekt riadiacej triedy k uzlu `missileNode`, ten pripojiť ku koreňovému uzlu `rootNode` a objekt riadiacej triedy do fyzického sveta `PhysicsSpace`. Tým sa docieli integrácia a aktivácia fyzických vlastností pre objekt strely.

Obdobne aj tu má riadiaca trieda za úlohu vo vlastnej aktualizačnej slučke `update()` zisťovať podmienky automatického odstránenia. K odstráneniu objektu strely zo scény a objektu jej riadiacej triedy z fyzického sveta môže dôjsť dvojakým spôsobom. Buď po kolízii strely s prekážkou, prípadne s dráhou alebo po uplynutí určitého času od vystrelenia. Odstránenie na základe času je implementované z dôvodu, aby objekt riadiacej triedy nechcene neostal v `PhysicsSpace` a zbytočne ho nezaplňoval. To by mohlo mať nežiaduce účinky po reštarte aplikácie či zmene levelu, keď by mohlo dôjsť k nežiaducim kolíziám s týmito zabudnutými objektmi. To by mohlo mať taktiež v konečnom dôsledku dopad na plynulosť aplikácie.

## 7.9 Kolízie

Jedným z dôležitých faktorov, na ktoré treba myslieť pri vytváraní trojrozmernej hry je interakcia medzi jej objektmi v scéne. A keďže táto aplikácia bola vytváraná s plnou podporou fyziky pre tieto objekty, je priam nevyhnutné, aby v aplikácii dochádzalo k výpočtu kolízií a špecifikácii chovania objektov počas nich. V hre dochádza k výpočtu kolízie v dvoch riadiacich triedach, a to `PlayerControl` a `MissileControl`, kde obe tieto triedy musia pre aktiváciu naslúchania ku kolíziám dediť rozhranie `PhysicsCollisionListener`. Z tohto rozhrania je potom potrebné implementovať abstraktnú triedu `collision(PhysicsCollisionEvent event)`, kde dochádza k riadeniu a špecifikovaniu chovania pri kolízií. Objekt `PhysicsCollisionEvent` predstavuje objekt, ktorý reprezentuje všetky udalosti spojené s kolíziou. Vďaka nemu je potom možné zisťovať informácie o tom, ktoré objekty kolidujú, ako silno, aký kladú odpor a mnoho ďalších užitočných informácií [36].

### 7.9.1 Kolízie futuristického motocyklu

Po implementovaní rozhrania pre naslúchanie kolízií je potrebné špecifikovať chovanie pre kolidujúce objekty. Meno prvého objektu `A` vstupujúceho do kolízie, pokiaľ ho má špecifikované, sa v podobe reťazca získava pomocou metódy `event.getNodeA().getName()`, prípadne pomocou metódy `event.getObjectA()` pre objekty, ktoré špecifikované meno nemajú. Podobným spôsobom sa tiež získava meno druhého objektu `B` volaním `event.getNodeB().getName()`, prípadne `event.getObjectB()`. Na základe porovnania týchto reťazcov s názvami objektov vystupujúcich v hre je možné pristúpiť ku špecifikovaniu ich chovania. Toto porovnanie je však nutné pre dva kolidujúce objekty `A` a `B` prevádzať obojstranne, keďže vopred nie je možné určiť ich poradie vstupu do kolíznej udalosti `PhysicsCollisionEvent`. Preto sa je potrebné vysporiadať s oboma možnými variantmi ich vstupu.

#### 7.9.1.1 Typy kolízie motocyklu

V riadiacej triede `PlayerControl` dochádza k zisťovaniu dvoch typov kolízie, a to medzi motocyklom a prekážkou a medzi dvoma prekážkami navzájom.

Pri prvom type kolízie medzi objektom motocykla a objektom prekážky sa volá metóda `handleCollisionWithBrick(PhysicsCollisionObject object)`, kde premenná `object` predstavuje objekt prekážky. Následne sa na základe typu prekážky prevádzajú určité akcie.

Pri náraze na klasickú prekážku je odrátaný jeden život motocykla a aktivuje sa pre neho mód nezraniteľnosti, ktorý počas trvania tohto módu ochraňuje motocykel pred ďalšou kolíziou. Taktiež

dôjde k odčítaniu určitej hodnoty z aktuálne nahraného skóre, k vytvoreniu odpovedajúceho informatívneho textu z triedy `HUDTextControl` a k prehratiu odpovedajúceho zvukového efektu.

Ak sa jedná o typ bonusovej prekážky pre zvýšenie počtu životov, motocyklu sa pripočíta jeden život, aktuálne nahrané skóre sa zvýši o určitú hodnotu, vytvorí sa odpovedajúci informatívny text, prehrá sa odpovedajúci zvukový efekt a nastaví sa indikátor pre túto prekážku `setRemoveObstacle(true)`, ktorý spôsobí, že jej vlastná riadiaca trieda vyvolá svoje automatické odstránenie, a tým aj odstránenie objektu bonusovej prekážky zo scény. K týmto všetkým udalostiam dôjde aj pri kolízií motocyklu s bonusovou prekážkou pre aktiváciu módu nezraniteľnosti, avšak namiesto pripočítania životu motocykla sa na určitú dobu aktivuje mód nezraniteľnosti. Tu by sa dal vymedziť ešte jeden typ kolízie, keď dôjde ku kolízií motocykla počas trvania nezraniteľnosti s klasickou prekážkou, kedy nedochádza k zníženiu počtu životov a namiesto odpočítania sa pripočítava určitá hodnota k aktuálne nahranému skóre.

Každému typu prekážky sa po kolízii s motocyklom nastaví indikátor `setCollisionDetected(true)`, ktorý zamedzuje opätovnému vypočítavaniu kolízií s týmito už raz kolidujúcimi prekážkami.

Pri druhom type kolízie medzi dvomi prekážkami navzájom dochádza taktiež k nastaveniu tohto indikátora pre obe kolidujúce prekážky. To z dôvodu zamedzenia prevádzania nechcených výpočtov kolízií, prípadne, aby napríklad nedochádzalo k odpočítavaniu života motocyklu pri náraze na ležiacu prekážku na zemi. Ak je jednou z týchto dvoch prekážok bonusová prekážka, dôjde k jej automatickému odstráneniu zo scény.

### 7.9.2 Kolízie strely

V triede `MissileControl` prebieha okrem priameho zisťovania kolízií s objektom strely, ktoré funguje na rovnakom princípe ako je spomenuté vyššie, implementovaná schopnosť zisťovať kolízie s ďalšími objektmi, ktoré sa nachádzajú v dosahu jeho explózie. Preto je súčasne s objektom strely vytvorený ďalší objekt z triedy `PhysicsGhostObject`. Ten dokáže automaticky sledovať svoj rodičovský objekt a keďže je neviditeľný a nemá žiadnu hmotnosť, nezasahuje ani do diania na scéne. Jeho hlavnou úlohou je pasívna detekcia kolízií. K tomu slúži jeho kolízny tvar s rozmermi gule o veľkosti, ktorá bude predstavovať výbušný rádius strely a jej tlakovú vlnu.

Pre správne fungovanie tohto spôsobu vypočítavania kolízie je potrebné, aby trieda `MissileControl` dedila ešte jedno rozhranie `PhysicsTickListener`, ktorá sprístupňuje metódy `prePhysicsTick()` a `physicsTick()`, vďaka ktorým je možné bližšie špecifikovať chovanie pred a v rámci aktualizačnej slučky pre fyziku ešte pred vykresľovaním snímku.

Pre aktiváciu volania týchto metód je potrebné v okamihu, keď nastane kolízia strely, pridať do fyzického sveta `PhysicsSpace` poslucháča tikov aktualizačnej slučky pre fyziku `addTickListener(this)` v metóde `collision()`. Následne sa do neho pripojí objekt

triedy `PhysicsGhostObject` a dôjde k volaniu metódy `prePhysicsTick()`, ktorej úlohou je odstránenie poslucháča kolízií `removeCollisionListener(this)` pre objekt strely, kvôli tomu, aby sa zamedzilo nechcenému opätovnému nahlasovaniu kolízie.

Týmto je všetko pripravené pre volanie metódy `physicsTick()`. Tu dochádza k hlavnému výpočtu kolízie objektu `PhysicsGhostObject` s ostatnými objektmi v jeho dosahu, ktorých zoznam sa zistí pomocou metódy `getOverlappingObjects()`. Na každý objekt v tomto zozname je potom v závislosti od jeho vzdialenosti od strely, aplikovaný výpočet smeru a sily, na základe ktorého bude tento objekt posunutý. K posunu dochádza pomocou metódy `applyImpulse(vector, Vector3f.ZERO)`, kde prvý parameter `vector` reprezentuje hodnotu vypočítanej sily a smeru, ktorým sa tento impulz aplikuje a druhý parameter určuje počiatočné miesto tohto impulzu, ktoré je nastavené na nulový vektor. Po dokončení výpočtov a aplikácie impulzov pre tieto objekty sa na konci metódy `physicsTick()` odstráni z `PhysicsSpace` posluchač tikov `removeTickListener(this)` a objekt triedy `PhysicsGhostObject`.

Po dokončení volania týchto metód dôjde v metóde `collision()` k prehratiu zvukového efektu pre explóziu strely, nastavenie jeho pozície v scéne a k odstráneniu objektu riadiacej triedy `MissileControl` spolu s objektom strely z fyzického sveta.

#### 7.9.2.1 Typy kolízie strely

V riadiacej triede `MissileControl` dochádza k zisťovaniu niekoľkých typov kolízie, a to priama kolízia strely s objektom dráhy, klasickej prekážky, bonusovej prekážky alebo nepriama kolízia s týmito objektmi prostredníctvom explózie strely a jej tlakovej vlny.

Pri priamej kolízii sa volá obdobná metóda `handleCollisionWithBrick()` ako pri kolíziách motocyklu, ktorá taktiež na základe typov kolidujúcich objektov prevádza určité akcie.

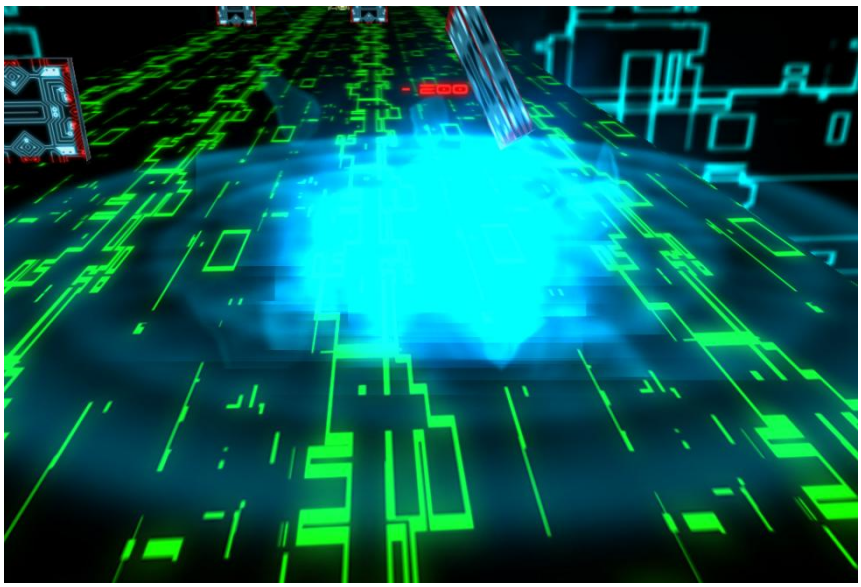
Pri priamom strete strely s klasickou prekážkou sa pripočíta určitá hodnota k aktuálne nahranému skóre, zobrazí sa informatívny text a prekážke sa deaktivuje možnosť výpočtu ďalších kolízií, aby napríklad nedošlo k odrátaniu života motocyklu po náraze už na zasiahnutú prekážku. Pri priamej kolízii strely s bonusovou prekážkou je postup rovnaký ako keby do prekážky narazil motocykel, akurát je za takto zasiahnutý bonus na diaľku pripočítaná menšia hodnota k aktuálne nahranému skóre. Pri kolízii strely a dráhy dôjde k výbuchu strely a neprevádza sa žiadna akcia.

Pri nepriamej kolízii nedochádza k zvyšovaniu aktuálneho skóre. Na základe výpočtu sily v metóde `physicsTick()`, ktorá bude aplikovaná na prekážky v dosahu explózie a tlakovej vlny sa určuje či je prekážkam ponechaná možnosť ďalšieho výpočtu kolízie alebo nie. Ak je táto sila natoľko silná, že danú prekážku viditeľne posunie, tak sa prevádzajú akcie podľa typu prekážky, kde klasickej prekážke sa deaktivuje možnosť ďalšieho výpočtu kolízií a bonusová prekážka sa úplné

odstráni zo scény pomocou nastavenia indikátora pre automatické odstránenie, čím sa hráč pripraví o možnosť tento bonus získať.

### 7.9.3 Efekt explózie

Tento efekt je prevádzaný obdobne ako efekt zanechávania stop od kolies motocyklu. Kvôli vierohodnejšiemu a vizuálne prepracovanejšiemu efektu explózie je tento výbuch skladaný z viacerých objektov triedy `ParticleEmitter`. V hre môžu nastať dva typy explózie. K prvému typu dôjde vždy, keď nastane kolízia objektu strely s inými objektmi v scéne, kde spoločne s týmto výbuchom dôjde k prehratiu zvukového efektu pre explóziu strely. Druhý typ nastáva v momente, keď sa motocyklu po kolízii s prekážkou zníži celkový počet životov na nulu. Keďže spoločne s touto explóziou prichádza k úplnému koncu hry, je táto explózia omnoho väčšia a je doprevádzaná na záver výsmešným zvukovým efektom.



**Obrázok 7.20: Explózia motocyklu (Zdroj: vlastný)**

# 8 Testovanie a limitácie jME aplikácie

Pri vytváraní hernej aplikácie pre platformu Android pomocou open source herného enginu JME 3 som sa stretol s viacerými limitáciami, ktoré spôsobovali či už menšie alebo väčšie problémy pri prepájaní týchto dvoch technológií. Tie boli badateľné hlavne počas priebehu vývoja tejto aplikácie, kedy sa funkcionalita jednotlivých naprogramovaných častí testovala najprv na notebooku s operačným systémom Windows 7 a až následne na zariadení s Androidom. Žiaľ veľa z týchto limitácií nie je spomenutých v dokumentácii jME 3, a preto bolo vyriešenie a odstraňovanie niektorých problémov pomerne časovo náročné.

Najprv je však potrebné vymedziť na akých zariadeniach s operačným systémom Android je možné spúšťať aplikácie vytvorené pomocou jME 3. Tieto zariadenia by mali mať operačný systém Android 2.2 a vyšší a je priam nevyhnutné, aby ich grafické karty podporovali aspoň OpenGL ES 2.0 [36]. Táto kapitola poskytuje zhrnutie limitácií technológie jME 3 a popisuje priebeh testovania vytvorenej aplikácie pre Android.

## 8.1 Testovanie

Testovanie aplikácie prebiehalo primárne na dvoch zariadeniach s operačným systémom Android, a to na tablete Google Nexus 7 a chytrom telefóne Sony Xperia S.

K tomuto účelu výborne poslúžila vstavaná podpora v jME 3 pre testovanie aplikácie pomocou natívneho aplikačného stavu `StatsAppState`. Ten poskytuje po svojom pripojení k manažérovi stavov zobrazenie štatistickej tabuľky (`StatsView`), ktorá obsahuje údaje pre dôkladné otestovanie. Tá je umiestnená v ľavom dolnom rohu obrazovky a zobrazovanie jej jednotlivých častí sa dá ovládať pomocou metód `setDisplayFps(true)` a `setDisplayStatView(true)`, pre aktiváciu, respektíve deaktiváciu zobrazovania FPS, prípadne ďalších štatistík na obrazovke. Táto tabuľka v priebehu hry vyzerá nasledovne:

```
FrameBuffers (M) = 0
FrameBuffers (F) = 0
FrameBuffers (S) = 0
Textures (M) = 37
Textures (F) = 26
Textures (S) = 22
Shaders (M) = 10
Shaders (F) = 9
Shaders (S) = 7
Objects = 43
Uniforms = 110
Triangles = 23948
Vertices = 19507
Frames per Second: 60
```

**Obrázok 8.1: Štatistická tabuľka `StatsView` (Zdroj: vlastný)**

Kde jednotlivé položky predstavujú:

- `FrameBuffers` – zobrazuje celkový počet použitých vykresľovacích vrstiev. Tie sa používajú pri post-processingu doplnkových efektov v aplikácií. Ak tento post-processing nie je používaný, tieto hodnoty by mali mať hodnotu nula.
- `Textures` – zobrazuje celkový počet rozdielnych textúr použitých v scéne.
- `Shaders` – udáva celkový počet shaderov použitých pre simuláciu efektov.
- `Objects` – informuje o celkovom počte objektov v OpenGL pipeline. Sú to objekty pripojené pod koreňové uzly `rootNode` a `guiNode`.
- `Uniforms` – udáva celkový počet premenných `Uniforms`. To sú preddefinované premenné použité ako parametre pri výpočtoch v shaderoch. Obsahujú dáta ako matice, vektory, čas, farbu a iné.
- `Triangles` – predstavuje celkový počet trojuholníkov objektov zobrazovaných v scéne.
- `Vertices` – informuje o celkovom počte vrcholov objektov zobrazovaných v scéne.

Okrem týchto hlavných položiek táto tabuľka ešte rozlišuje ich typ, ktorý môže byť:

- `Memory(M)` – udáva počet položiek aktuálne v OpenGL pamäti.
- `Frame(F)` – zobrazuje počet položiek (viditeľných) v aktuálnom snímku.
- `Switches(S)` – informuje o počte položiek, ktoré boli prepnuté z a do pamäte počas posledného snímku.

Aj na základe týchto štatistík prebiehalo testovanie aplikácie. Žiaľ `StatsView` v súčasnej dobe nepodporuje žiadne štatistiky ohľadom fyzického sveta a objektov v ňom, kde práve tieto údaje by boli pri ladení plynulosti aplikácie po integrácii fyziky veľmi prospešné [37].

Obrázok 8.1: Štatistická tabuľka `StatsView` predstavuje hodnoty namerané počas hry v priebehu prvej úrovni vo finálnej verzii aplikácie, čo znamená po skončení procesu jej ladenia a optimalizácie.

jME 3 v súčasnej dobe nepodporuje využívanie FrameBufferov a s ním spojené vytváranie efektov pomocou post-processingových filtrov ako napríklad efekty vodnej hladiny, rozmazávanie, vyžarovanie objektov, rozptyl svetla a iných [38]. Preto všetky hodnoty v tabuľke pri položke `FrameBuffers` obsahujú nulu.

Čo sa týka hodnôt pre položku `Textures`, číslo 37 u položky `M` predstavuje skoro konečné číslo počtu textúr v pamäti. To dosahuje maximálnu hodnotu 40 v 4 úrovni hry, kedy sa prestriedajú všetky možné farebné odtiene dráhy. Hodnoty F a S kolísajú približne na rovnakej úrovni počas celého priebehu hry. Ideálne rozloženie týchto hodnôt je v prípade, keď nie je hodnota S oveľa vyššia ako F, pretože nedochádza k zbytočne väčšiemu prepínaniu textúr z a do pamäte ako je ich aktuálne použitie v scéne.

Ohľadom údajov u položky `Shaders` a `Uniforms`, sa dá povedať že je to podobné ako u položky `Textures`, kde sa tieto hodnoty počas priebehu hry radikálne nemenia, pretože je ich využitie pre vytváranie efektov maximálne optimalizované.

Veľmi dôležitou položkou, ktorá udáva aktuálny počet vykresľovaných geometrií v poslednom snímku je `Objects`. Vďaka jej hodnotám sa dali pohodlne lokalizovať problémy s používaním, či vytváraním príliš veľkého počtu objektov. Ideálne hodnoty sú stanovené niekde na hranici 100 až 200 objektov pre desktopové počítače [37]. Dá sa teda povedať, že hodnota 43 predstavuje prijateľnú hodnotu pre platformu Android. Tento údaj sa ešte mierne zvyšuje s každou ďalšou dosiahnutou úrovňou hry, keďže sa zvyšuje aj počet generovaných prekážok.

Položky `Triangles` a `Vertices` slúžili pre informáciu ohľadom zložitosti použitých objektov v scéne. Aj na ich základe bol model motocyklu mierne upravený, aby neobsahoval príliš veľký počet trojuholníkov. Maximálna hranica je stanovená približne okolo 100 000 vrcholov pre scénu, kde nad touto hranicou by mala grafická karta veľké problémy z hľadiska výkonu. Pretože nie každý užívateľ vlastní veľmi výkonnú grafickú kartu je ideálnym rozmedzím pre počet vrcholov stanovený medzi 10 000 až 50 000 [37]. V hre tieto hodnoty kolísajú okolo 25 000 pre trojuholníky a 20 000 pre vrcholy v závislosti od aktuálneho levelu, čo sú dá sa povedať ideálne hodnoty.

## 8.2 Zistené problémy pri testovaní

Zistené problémy pri testovaní aplikácie by sa dali rozdeliť do menších logických podčastí:

- Logovanie a načítavanie aplikácie.
- Prehrávanie zvukových súborov.
- Použitie efektov v aplikácii.
- Integrácia fyziky do aplikácie.
- Limitácie GUI a multi-dotykového ovládania.
- Podpísanie aplikácie.

### 8.2.1 Logovanie a načítavanie aplikácie

Prvým vážnejším problémom bolo zistenie, že vypisovanie logov počas behu aplikácie na platforme Android malo výrazný dopad na jej plynulosť. Tieto logy vytvára jME 3 pri úvodnej inicializácii systému, kedy dochádza k vytvoreniu vykresľovacieho okna pomocou LWJGL, vytvoreniu objektov manažéra vstupu, zvuku, herných prostriedkov, kamery a ďalších a hlavne pri získavaní informácií ohľadom schopností grafickej karty. Taktiež dochádza k logovaniu pri vytváraní uzlov, geometrií, pri ich pripájaní a odpájaní z rodičovských uzlov a podobne. Najvážnejší dopad z pohľadu výkonnosti som však zaznamenal pri vytváraní užívateľského rozhrania pomocou knižnice NiftyGUI. Tá logovala vytvorenie každého elementu a prevedenie každej akcie, čo pri rozsiahlosti vytvoreného

GUI značne spomaľovalo celú aplikáciu. Riešenie spočívalo v nájdení kľúčových slov pre vypnutie NiftyGUI logovania v jej dokumentácii. To sa následne dalo obmedziť iba na nevyhnutné hlásenia systémových chýb a podobne pomocou metód:

```
Logger.getLogger("de.lessvoid.nifty").setLevel(Level.SEVERE)
Logger.getLogger("NiftyInputEventHandlingLog").setLevel(Level.SEVERE)
```

Vypnutie natívneho jME 3 logovania bolo potrebné urobiť v každej vytvorenej triede `Logger.getLogger(ClassName.class.getName()).setLevel(Level.SEVERE)`, kde `ClassName` predstavuje meno triedy, pre ktorú sa má logovanie vypnúť. Po vypnutí logovania bolo možné spustiť aplikáciu na platforme Android s hodnotou skoro 60 FPS, čo predstavuje maximálnu hodnotu, ktorú je možné dosiahnuť pri integrovanej fyzike.

Veľmi dlho riešeným problémom bol čas potrebný pre načítanie herných prostriedkov. V počiatku vývoja hry trvalo načítanie prostriedkov a prechod do herného stavu `GameAppState` cez 30 sekúnd aj po odstránení logovania aplikácie čo bola neakceptovateľná doba. Identifikovať problém pri načítavaní sa podarilo vďaka stavu `LoadingAppState` a zobrazovaniu priebehu načítavania, na základe ktorého boli identifikované dlhotrvajúce metódy pri inicializácii objektov. Tento problém spôsobovala veľkosť textúr. Najdlhšie trvajúcou metódou bola `initBackground()`, ktorá v hre vytvárala pozadie z textúry s rozmermi 1024x1024, kvôli lepšej kvalite pozadia. Nakoniec však bolo potrebné urobiť kompromis a dosiahnuť relatívne krátky čas načítavania hry pri prijateľnej kvalite textúr. Ideálna hodnota rozlíšenia bola 512x512 pre textúru pozadia, 256x256 pre textúru dráhy, prekážok a motocyklu a 64x64 pre ostatné objekty. Po nastavení veľkosti textúr sa prvé načítavanie hry zrýchlilo na približne 12 sekúnd. Načítavanie pri zmene levelu, prípadne po reštarte sa podarilo znížiť na hranicu okolo 2 sekúnd vďaka inicializácii iba potrebných objektov.

### 8.2.2 Prehrávanie zvukových súborov

Taktiež jedným z väčších problémov z hľadiska nájdenia riešenia, práve kvôli absencii jeho popisu v dokumentácii bolo zaistenie prehrávania zvukových súborov. Pre docielenie prehrávania zvuku na platforme Android bolo potrebné ručné nahranie zvukových súborov do priečinku `Mobile\Assets\Sounds`, pretože prehrávač MediaPlayer, ktorý slúži pre prehrávanie zvukov na Androide nebol schopný používať zvukové súbory z priečinku `Assets` v koreňovom adresári hry. Avšak používanie iných prostriedkov z tohto adresára ako materiály, textúry, modely a podobne nespôsobovalo ostatným Android manažérom žiadne problémy.

Ako je spomenuté v kapitole 7.3 Zvukový systém, Android v súčasnej dobe neumožňuje plnú zvukovú podporu pre svoje aplikácie kvôli reštrikciám API. A práve tento fakt má negatívny vplyv na plnohodnotnú prácu so zvukom. Ďalšími problémami zistenými pri testovaní zvukového výstupu aplikácie bola kvalita prehrávaných zvukov, ktorá nebola úplné ideálna. Tu sa podarilo o niečo

zlepšiť použitím bezkompresného formátu PCM Wave (`.wav`). Tým sa odstránili aj rôzne chyby pri prehrávaní súborov, napríklad kolísanie FPS pri prehrávaní zvukových efektov. Taktiež bola zistená nefunkčnosť metódy `setPitch()`, ktorá by jednoducho vyriešila prehrávanie zvuku motora motocyklu a jeho výšky v závislosti od jeho rýchlosti. Preto bolo potrebné pre docielenie simulácie rýchlosti motora použiť viac zvukových súborov s rôznou výškou a vymyslieť algoritmus pre správne prepínanie medzi týmito zvukmi. Taktiež nefunguje použitie metódy `setVolume()`, pre nastavenie hlasitosti prehrávania jednotlivých zvukových efektov a hudby. Preto nastavenie správnej hlasitosti pri prehrávaní na Androide si vyžadovalo manuálne a prácne nastavovanie pomocou externého programu.

### 8.2.3 Použitie efektov v aplikácii

Pri implementovaní efektov do aplikácie bol prvým hlavným problémom spôsobovanie systémových chýb a padanie celej aplikácie pri použití tieňov. Tieňovanie muselo byť z aplikácie odstránené, pretože v súčasnej dobe verzia jME 3 nepodporuje možnosť vytvárať aplikácie s podporou tieňovania a ďalších efektov spomenutých vyššie.

Ďalšie jednoduchšie efekty ako oheň, explózia vytvárané pomocou triedy `ParticleEmitter`, je nutné používať rozumne, pretože príliš veľa efektov prehrávaných naraz v scéne má taktiež výrazný dopad na rýchlosť aplikácie. Preto bolo potrebné nájsť rozumný kompromis medzi vizuálnou stránkou hry a jej plynulosťou. Pôvodne sa v aplikácii vytváral objekt pre efekt výbuchu vždy pri vytvorení objektu strely, kde po jeho kolízií došlo k vypusteniu všetkých čiastočiek a po ich zániku sa tento objekt spolu s objektom strely odstránil zo scény. Avšak na základe údajov z tabuľky `StatsView`, kde som sledoval hodnoty pre počet vytvorených objektov, trojuholníkov a textúr v scéne, som objavil chybu v nedôkladnom odstraňovaní objektov z triedy `ParticleEmitter`. Tieto čísla rástli po každom vystrelení a vybuchnutí strely. To viedlo po vystrelení určitého počtu striel počas hrania hry k postupnému znižovaniu FPS aplikácie až na neúnosnú mieru z pohľadu hrateľnosti. Riešenie bolo v presunutí inicializácie efektov výbuchu do aplikačného stavu `GameAppState`. Takto sa vytvoril iba jeden zhluk efektov, ktorý sa používa pri každej simulácii výbuchu strely. Najprv je však potrebné pri každej vystrelenej strele zistiť pozíciu jej kolízie v scéne, a tú použiť pre správne umiestnenie efektov. Na tejto pozícii následne dôjde k vypusteniu čiastočiek pre simulovanie výbuchu a po jeho dokončení ostávajú tieto objekty naďalej aktívne v scéne. Pri následnej kolízii dôjde k premiestneniu pozície efektov a opätovnému vypusteniu ich čiastočiek do scény. Tento spôsob presúvania efektu výbuchu na správne miesto ušetril herné prostriedky, pretože sa nevytvárajú stále nové objekty pre každý výbuch ako aj zamedzil znižovaniu FPS aplikácie. Fungovanie tohto prístupu sa potvrdilo na základe štatistickej tabuľky, kde hodnoty pre počet vytvorených objektov a trojuholníkov v scéne nevykazovali zvyšujúcu sa tendenciu.

Veľmi dôležitým nastavením, ktoré umožňuje zviditeľnenie efektov v rámci scény na platforme Android, je nastavenie používania správneho typu pre vytvorenie čiastočiek. Android nepodporuje nastavenie `ParticleMesh.Type.Point`, ktoré poskytuje kvalitnejšie vykresľovanie efektov. Je tu potrebné použiť menej kvalitný typ `Triangle`.

### 8.2.4 Limitácie GUI a multi-dotykového ovládania

Zobrazovanie textov na HUD obrazovke Androidu nie je taktiež úplne ideálne. Tu dochádzalo k veľmi častým chybám, hlavne kvôli posúvaniu textu, keď sa menila jeho dĺžka. Napríklad pri zmene trojciferného čísla na dvojciferné alebo pri zmene šírky aktuálne zobrazovaných znakov. Preto bolo potrebné najprv upraviť šírku znakov na fixnú šírku pre všetky číslice, aby sa čo najviac zamedzilo posunom, ošetriť pripájanie textov na HUD tak, aby nedochádzalo k ich opätovnému pripájaniu pokiaľ už boli raz pripojené a obmedziť možnosti týchto prechodov medzi rôznou dĺžkou textu na minimum.

Z dôvodu zachovania maximálnej plynulosti hry a podpory multi-dotykového ovládania nebola HUD obrazovka vytváraná pomocou NiftyGUI, ale za pomoci natívnych jME 2D textov a obrázkov. Pôvodne bol HUD vytvorený taktiež prostredníctvom NiftyGUI, avšak jeho použitie spôsobilo zníženie a kolísanie hodnoty FPS okolo čísla 30. Táto hodnota ešte stále nemala výrazný vplyv na hrateľnosť avšak NiftyGUI neumožňuje v súčasnej dobe rozumne podporovať multi-dotykové ovládanie a tieto dva hlavné dôvody ma viedli k tomu, aby bol HUD implementovaný natívnymi jME objektmi. Týmto prístupom bolo možné dosiahnuť, že FPS počas hrania dosahuje hodnoty blížiacim sa maximu čo je 60.

Nemilým prekvapením bol aj fakt, že NiftyGUI neponúka nejaký unifikovaný spôsob prepínania medzi `Nifty` obrazovkami a obrazovkami bez `Nifty` rozhrania tak, aby došlo ku správnemu dokončeniu animovaných efektov. Čo malo za následok zobrazovanie rôznych artefaktov na obrazovke a v niektorých prípadoch až pád celej aplikácie. Preto je v aplikácií implementovaný aplikačný stav `EndEffectAppState`, ktorý spôsobuje korektné dokončenie týchto efektov pri prepínaní obrazoviek alebo pri ukončení celej aplikácie.

Čo sa týka problémov pri implementácií multi-dotykového ovládania, bol pôvodný plán pre namapovanie streľby na rýchle dvojité ťuknutie `DOUBLETAP`. Hráč by tak nemusel vždy triafať presne vymedzený priestor pre tlačidlo streľby, ale mohol by týmto typom ťuknutia hocikde na obrazovke vystreliť. Žiaľ `TouchListener` umožňuje rozoznávanie tohto typu dotyku iba v prípade prvého dotyku obrazovky. Pri každom ďalšom prste na obrazovke už nedokáže rozlišovať typ dotyku `DOUBLETAP` a generuje iba typy `DOWN`, `UP` a `MOVE`, prípadne `SCROLL`. Preto by hráč nemohol strieľať súčasne s ovládaním motocyklu, a preto je pre zachovanie plnej podpory multi-dotykového ovládania namapované strieľanie na tlačidlo v spodnej časti obrazovky.

### 8.2.5 Integrácia fyziky do aplikácie

Z pohľadu limitácii knižnice jBullet pre integráciu fyziky do aplikácie som sa stretol s niekoľkými problémami, ktoré si vyžadovali viac či menej sofistikované riešenia.

Ako je spomenuté v predchádzajúcich kapitolách, veľkým problémom pre dosiahnutie plynulého a ustáleného chodu aplikácie bola integrácia fyziky. Keďže objekty pripájané do fyzického sveta `PhysicsSpace` sa neodstraňovali automaticky po odstránení svojho rodičovského uzlu zo scény, dochádzalo k jeho neúmernému preplňovaniu pri vytváraní ďalších fyzikálnych objektov, k nežiaducim kolíziám nových fyzických objektov so starými a hlavne k znižovaniu plynulosti hry. Kde po každej zmene levelu, prípadne po reštarte hry sa FPS znižovalo o približne 5 až 10 snímkov za sekundu. Tento problém sa podarilo úplne odstrániť pomocou implementovania riadiacich tried, kde každý vytvorený objekt v hre je nimi riadený. Vďaka nim sa zaručuje automatické odstraňovanie týchto objektov z fyzického sveta a zároveň ich rodičovských objektov zo scény pri zachovaní plynulosti hry a jej dobrej hrateľnosti.

Pre objekt motocyklu som chcel vytvoriť kolízny tvar za pomoci vstavaných tried na to určených. Tie dokážu vytvoriť kolízny tvar presne podľa hraníc modelu. Takýto tvar by bol najlepší z hľadiska detekcie kolízií. Na Android platforme však tieto automatické kolízne tvary spôsobovali veľmi nízke FPS. Tu si to vyžiadalo riešenie s manuálnym vytvorením kolízneho tvaru pre motocykel v tvare dvoch gúľ pre predné a zadné koleso a kváder pre jeho telo. Po presnom umiestnení týchto tvarov sa podarilo kolízie detekovať rovnako dobre ako keby bol tento tvar vytváraný automaticky.

Taktiež jedným z problémov bolo nastavenie hmotnosti motocyklu v rámci fyzického sveta. Príliš ľahký objekt motocyklu bol náchylný na preklápanie, či úplné vytočenie pri vyšších rýchlostiach a pri kolízii s prekážkou neodovzdal dostačujúcu kinetickú silu pre jej odrazenie. Vyššie čísla zase spôsobovali závažnú chybu, kedy objekt motocykla prepadával cez statické objekty dráhy. Preto bolo potrebné nájsť dostatočný kompromis medzi ideálnou váhou, ktorá je udržateľná na dráhe a zároveň poskytuje dostatočnú kinetickú silu pre odrážanie prekážok. Riešením bolo nastavenie ideálnej hmotnosti motocyklu spoločne s jeho osadením na štvorkolesový podvozok, ktorý slúži pre jeho lepšiu stabilitu a vypnutie možnosti jeho rotácie metódou `setAngularFactor(0f)`.

jME 3 ponúka pri integrácii fyziky možnosť nastavenia kolíznych skupín. Ich použitím sa môže značne zvýšiť rýchlosť simulácie fyziky, pretože dochádza k výpočtu kolízii iba medzi chcenými kolíznymi skupinami. Tým by sa neumožňovala detekcia kolízií medzi dráhou a motocyklom, prípadne prekážkami a medzi strelou a motocyklom, pretože tie nie sú potrebné. Tieto kolízne skupiny však nie sú na Androide podporované. Preto sa napríklad v hre musí objekt strely manuálne posunúť mimo hraníc motocyklu, aby nedochádzalo k výbuchu strely hneď po jej vytvorení. Znížiť počet výpočtov sa podarilo pomocou detekcie mena kolidujúceho objektu, kde ak

jedným z dvoch kolidujúcich objektov je meno dráhy, tak sa výpočet ďalej neprevádza. Samozrejme tento spôsob nie je tak ideálny ako keby fungovali kolízne skupiny.

Pôvodne som v hre plánoval vytvoriť objekt prekážky z menších častí, ktoré by sa po náraze rozleteli. Toto riešenie však nebolo možné hlavne kvôli nefunkčnosti kolíznych skupín, pretože by dochádzalo k neustálej detekcii kolízie medzi jednotlivými časťami prekážky, a to by vzhľadom na počet prekážok na dráhe výrazne spomaľovalo celú aplikáciu. Preto je objekt prekážky tvorený jedným objektom, na ktorom je stále možné dostatočne dobre pozorovať simuláciu fyziky.

### 8.2.6 Podpísanie aplikácie

Posledným riešeným problémom vzhľadom k výkonnosti aplikácie bolo podpisovanie aplikácie samotnej. Vďaka tomu bolo možné na záver vytvoriť výsledný inštalačný súbor `.apk` v `release` móde. Ten zaručil o niečo lepšiu výkonnosť aplikácie ako v pôvodnom `debug` móde. Porovnanie plynulosti aplikácie na niektorých zariadeniach s platformou Android je uvedené v Tabuľke 8.1: Porovnanie výkonností Android zariadení a notebooku. Celý priebeh jej podpisovania bol vykonávaný podľa návodu na domovskej stránke vývojárov platformy Android [39].

## 8.3 Záverečné porovnanie zariadení

Na záver po vyriešení všetkých problémov a limitácií pri prepájaní technológií jME 3 a Android som mohol pristúpiť k otestovaniu vytvorenej aplikácie na konkrétnych zariadeniach, ktoré spĺňali požiadavky pre jej spustenie, a ktoré sa mi podarilo zaobstarať do testu. Vďaka faktu, že vytvorená aplikácia je po malých úpravách spustiteľná na viacerých platformách som taktiež kvôli lepšej výpovednej hodnote do testu zahrnul notebook HP Pavilion dv5 1060EC, na ktorom bola táto hra vyvíjaná, kde testovanie prebiehalo na operačnom systéme Windows 7 a 4 zariadenia s operačným systémom Android 4.1 Jelly Bean, a to tablety Google Nexus 7 a Samsung Galaxy Note 10.1 a chytré telefóny Sony Xperia S a Samsung Galaxy SII. Jednotlivá špecifikácia týchto zariadení sa dá dohľadať po zadaní kľúčových slov do internetového prehliadača.
Testy boli rozdelené do niekoľkých kritérií:

- Úvodné spustenie hry, kde sa meral čas od spustenia aplikácie až po zobrazenie úvodného loga.
- Rýchlosť prvého načítania herných prostriedkov.
- Rýchlosť opätovného načítania herných prostriedkov, napríklad po zmene levelu alebo po reštarte aplikácie.
- Počet FPS v 1 a 4 úrovni hry počas `debug` módu.
- Počet FPS v 1 a 4 úrovni hry počas `release` módu.

| | Úvodné Spustenie hry [sek.] | Rýchlosť prvého načítania hry [sek.] | Rýchlosť opätovného načítania hry [sek.] | Level 1 Debug [FPS] | Level 4 Debug [FPS] | Level 1 Release [FPS] | Level 4 Release [FPS] |
|---|---|---|---|---|---|---|---|
| HP Pavilion dv5 | 3.5 | 2.5 | 1.5 | 60 | 60 | 60 | 60 |
| Galaxy Note 10.1 | 4 | 11.5 | 1.5 | 59 | 59 | 59 | 59 |
| Nexus 7 | 4 | 12 | 1.5 | 59 | 59 | 59 | 59 |
| Xperia S | 5.5 | 14 | 2 | 39 | 35 | 48 | 44 |
| Galaxy SII | 5.5 | 14.5 | 2 | 37 | 34 | 46 | 43 |

**Tabuľka 8.1: Porovnanie výkonnosti Android zariadení a notebooku (Zdroj: vlastný)**

Ako je vidieť z tabuľky, kde sú uvedené zaokrúhlené a spriemerované hodnoty z 10 meraní, testy aplikácie na týchto zariadeniach dopadli celkom priaznivo. Hodnoty pri prvom načítaní hry na zariadeniach s Androidom sú výrazne vyššie ako na notebooku. To je spôsobené načítaním textúr do pamäte. Z tohto času práve skoro polovica prislúcha načítaniu pozadia s textúrami o rozmeroch 512x512. Na slabších zariadeniach sa dajú docieliť lepšie výsledky FPS po znížení presnosti výpočtov simulácie fyziky alebo obmedzení efektov v aplikácii. Napríklad výbuch po dopade strely simulovať iba jedným objektom triedy `ParticleEmitter`, vypnúť zanechávanie stôp od predného kolesa motocyklu, prípadne strely a podobne.

Na základe prevedených testov je teda možné tvrdiť, že po dôkladnej optimalizácii aplikácie a zvolení jej správnej štruktúry je možné docieliť ideálne hodnoty FPS na rôznych typoch zariadení. Za referenčnú hodnotu pre ideálne FPS môžeme považovať jeho hodnotu nad číslom 30 [37]. Samozrejme úroveň plynulosti aplikácie a dosiahnutie prijateľných výsledkov FPS záleží nielen na type a veľkosti aplikácie, ale aj na skúsenostiach a schopnostiach programátora vzhľadom k navrhnutiu jej správnej štruktúry, prevedenia jej implementácie ako aj na jeho časových možnostiach.

# 9 Záver

Operačný systém Android je síce na svete iba od roku 2003, ale práve za posledné tri-štyri roky zaznamenal taký rozmach a vývoj ako nijaký iný mobilný operačný systém. Každý deň je aktivovaných viac ako milión zariadení s týmto systémom. A viac ako jeden a pol miliardy aplikácií a hier je stiahnutých z virtuálneho obchodu Google Play každý mesiac [40]. Tieto fakty sú veľkým predpokladom, že počet programátorov, ktorý sa zameriava na vývoj aplikácií pre platformu Android bude aj naďalej rásť. Tým však bude stúpať aj konkurencia, a o tom či bude aplikácia úspešná alebo nie budú práve rozhodovať nielen faktory ako sú správne načasovanie príchodu aplikácie na trh, jej dokonalosť po vizuálnej stránke a po stránke hrateľnosti, ale aj doba od jej návrhu cez implementáciu až po jej konečné nasadenie, a hlavne s tým spojené náklady vynaložené na jej vývoj.

Pri vytváraní tejto práce som sa stretol s pozitívami aj negatívami ohľadom dostupnosti študijných materiálov. Negatívom, ktorý stojí za spomenutie bolo určite to, že oboznámenie s technológiou Java Monkey Engine 3 nebolo úplne triviálne, pretože dostupnosť študijných materiálov, či odborných náučných kníh venovaných tejto hernej knižnici, je naozaj nízka. Jediným dostupným materiálom bola oficiálna stránka jME a jej fórum. Preto študovanie a zoznámenie sa s touto knižnicou bolo celkom časovo náročné. Oproti tomu však značným pozitívom bol fakt, že Android platforma sa nepotyká s rovnakým problémom. K dispozícii je naozaj veľké množstvo dostupných materiálov, či už prostredníctvom textov a návodov na internete alebo odborných kníh.

Po prekročení týchto počiatočných problémov sa mi podarilo úspešne zvládnuť techniky potrebné pre vývoj hernej aplikácie pre Android platformu. Od tohto momentu sa práca na tejto diplomovej práci stala pre mňa veľmi zaujímavou a prínosnou, pretože mi umožnila prehĺbiť si znalosti ako z tvorby počítačových hier v Jave, programovaní samotnom, tvorbe 3D modelov v programe 3D Studio Max, tak aj v tvorbe zvukových efektov v programe FL Studio.

Cieľom tejto diplomovej práce bolo zoznámenie sa s herným enginom jMonkey Engine 3 a platformou Android a naštudovanie možností prepojenia týchto dvoch technológií pri tvorbe trojrozmernej hry čo sa mi úspešne podarilo. K demonštrácii tohto prepojenia slúži herná aplikácia, ktorú som sa snažil podať v atraktívnej forme pre užívateľa. Práve z tohto dôvodu sú v nej implementované všetky dôležité súčasti, ktoré by mala herná aplikácia v dnešnej dobe obsahovať. Počnúc úvodnou animáciou, jednoduchým, príjemným a intuitívnym užívateľským rozhraním, obrazovkou informujúcou o priebehu načítavania prostriedkov hry, externými 3D modelmi, zvukovými efektmi, cez vizuálne efekty, plne funkčnú integráciu fyziky až po implementáciu viacerých úrovní hry a multi-dotykového ovládania. Všetky tieto súčasti je možné implementovať vďaka prepracovanej hernej knižnici jME 3 spoločne s externými knižnicami, ktoré sú súčasťou jej balíčka, a to NiftyGUI pre užívateľské rozhranie a jBullet pre integráciu fyziky.

Po oboznámení a vyhnutí sa niekoľkým súčasným obmedzeniam, ktoré sú podrobnejšie spomenuté v kapitole č. 8 Testovanie a limitácie jME aplikácie, je v tomto prepojení ukrytý veľký potenciál do budúcnosti, pretože poskytuje možnosti pomerne jednoducho, efektívne a za krátky čas vyvinúť kvalitnú a po vizuálnej ako aj programátorskej stránke dokonale konkurencieschopnú aplikáciu pri relatívne nízkych nákladoch, keďže sa jedná o open source riešenie. Na prepájaní technológií jME a Androidu sa však stále aktívne pracuje, a preto je tu veľký predpoklad, že veľa z týchto obmedzení sa odstráni s príchodom novej verzie jME.

Prínos tejto práce je v poskytnutom podrobnom návode pre vývoj herných aplikácií pre Android platformu pomocou open source enginu jME 3, v zhrnutí všetkých poznatkov, dôležitých faktov a techník potrebných pre zvládnutie tohto prepojenia ako aj v zhrnutí limitácií a obmedzení, ktorým sa je potrebné v súčasnej dobe vyhnúť. Toto všetko je doplnené o praktickú ukážku, ktorá vďaka tomu, že bola vyvíjaná modulárne pomocou aplikačných stavov a riadiacich tried, môže poslúžiť ako základná kostra pre ďalších záujemcov, od ktorej sa môžu odraziť, prípadne ju rozšíriť.

V budúcnosti by sa do hry mohla implementovať umelá inteligencia pre pretekanie sa so súpermi, prípadne z tejto hry vytvoriť sieťovú hru pre viacerých hráčov, doplniť hru o ďalšie objekty rôznych tratí, prekážok, bonusov, motocyklov a zvukových či špeciálnych efektov.

# Literatúra


[1] ŠEVČÍK, M. *Závodní hra ve 3D.* Bakalárska práca, Vysoké učení technické v Brně, Fakulta informačních technologií, 2010.

[2] Introduction jME [online]. [cit. 2012-12-23]. Dostupné z WWW: <http://jmonkeyengine.org/introduction>

[3] jME Tutorials and Documentations [online]. [cit. 2012-12-23]. Dostupné z WWW: <http://www.jmonkeyengine.org/wiki/doku.php/jme3>

[4] Open Source Initiative [online]. [cit. 2012-12-23]. Dostupné z WWW: <http://opensource.org/licenses/bsd-license.php>

[5] jMonkey Engine [online]. [cit. 2012-12-23]. Dostupné z WWW: <http://en.wikipedia.org/wiki/JMonkey_Engine>

[6] LWJGL [online]. [cit. 2012-12-23]. Dostupné z WWW: <http://www.lwjgl.org/about.php>

[7] The Scene Graph [online]. [cit. 2012-12-23]. Dostupné z WWW: <http://www.jmonkeyengine.org/wiki/doku.php/jme3:the_scene_graph>

[8] Light and Shadow [online]. [cit. 2012-12-23]. Dostupné z WWW: <http://www.jmonkeyengine.org/wiki/doku.php/jme3:advanced:light_and_shadow>

[9] Audio in jME3 [online]. [cit. 2012-12-23]. Dostupné z WWW: <http://www.jmonkeyengine.org/wiki/doku.php/jme3:advanced:audio>

[10] Model File Formats [online]. [cit. 2012-12-23]. Dostupné z WWW: <http://www.jmonkeyengine.org/wiki/doku.php/jme3:beginner:hello_asset>

[11] Importing and Viewing a Model [online]. [cit. 2012-12-23]. Dostupné z WWW: <http://www.jmonkeyengine.org/wiki/doku.php/sdk:model_loader_and_viewer>

[12] ASE File Format [online]. [cit. 2012-12-23]. Dostupné z WWW: <http://wiki.beyondunreal.com/Legacy:ASE_File_Format>

[13] Asset Manager [online]. [cit. 2012-12-23]. Dostupné z WWW: <http://www.jmonkeyengine.org/wiki/doku.php/jme3:advanced:asset_manager>

[14] jMonkey Engine SDK Documentation [online]. [cit. 2012-12-23]. Dostupné z WWW: <http://www.jmonkeyengine.org/wiki/doku.php/sdk>

[15] Java Virtual Machine [online]. [cit. 2012-12-23]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Java_Virtual_Machine>

[16] J2ME [online]. [cit. 2012-12-23]. Dostupné z WWW: <http://sk.wikipedia.org/wiki/J2ME>

[17] OpenGL Utility Library [online]. [cit. 2012-12-23]. Dostupné z WWW: <http://en.wikipedia.org/wiki/OpenGL_Utility_Library>

[18] SILVA, V. *Advanced Android 4 Games.* New York: Apress, 2012. 302 s. ISBN 978-1-4302-4060-0.

[19] ZECHNER, M. *Beginning Android Games.* New York: Apress, 2011. 679 s. ISBN 978-1-4302-3043-4.

[20] ORGONÁŠ, J. Android: História, súčasnosť a budúcnosť. *PC Revue*. 2012, č. 4, s. 34-35. ISSN 1335-0226.

[21] Android (operační systém) [online]. [cit. 2012-12-22]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Android_(operační_systém)>

[22] Licence MIT [online]. [cit. 2012-12-22]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Licence_MIT>

[23] Android Architecture [online]. [cit. 2012-12-22]. Dostupné z WWW: <http://developer.android.com/about/versions/index.html>

[24] SimpleApplication [online]. [cit. 2012-12-24]. Dostupné z WWW: <http://jmonkeyengine.org/wiki/doku.php/jme3:intermediate:simpleapplication>

[25] DAVISON, A. *Programování dokonalých her v Javě*. Brno: Computer Press, 2006. 904 s. ISBN 80-7226-944-5.

[26] Application States [online]. [cit. 2013-04-06]. Dostupné z WWW: <http://jmonkeyengine.org/wiki/doku.php/jme3:advanced:application_states>

[27] Pulse-code Modulation [online]. [cit. 2013-04-08]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Pulse-code_modulation>

[28] Font Creator [online]. [cit. 2013-04-10]. Dostupné z WWW: <http://jmonkeyengine.org/wiki/doku.php/jme3:external:fonts>

[29] Nifty - GUI [online]. [cit. 2013-04-11]. Dostupné z WWW: <http://nifty-gui.lessvoid.com>

[30] NiftyGui The Manual [online]. [cit. 2013-04-11]. Dostupné z WWW: <http://switch.dl.sourceforge.net/project/nifty-gui/nifty-gui/nifty-gui-the-manual-v1.0.pdf>

[31] JavaDoc jME 3 [online]. [cit. 2013-04-12]. Dostupné z WWW: <http://jmonkeyengine.org/javadoc>

[32] jBullet [online]. [cit. 2013-04-14]. Dostupné z WWW: <http://jbullet.advel.cz>

[33] Bullet Physics Library [online]. [cit. 2013-04-14]. Dostupné z WWW: <http://bulletphysics.org/wordpress>

[34] Advanced Physics [online]. [cit. 2013-04-14]. Dostupné z WWW: <http://jmonkeyengine.org/wiki/doku.php/jme3:advanced:physics>

[35] Particle Emmiter [online]. [cit. 2013-04-20]. Dostupné z WWW: <http://jmonkeyengine.org/wiki/doku.php/jme3:advanced:particle_emitters>

[36] Physics Listeners [online]. [cit. 2013-04-21]. Dostupné z WWW: <http://jmonkeyengine.org/wiki/doku.php/jme3:advanced:physics_listeners>

[37] Optimizing Your Game Using Statistics [online]. [cit. 2013-04-28]. Dostupné z WWW: <http://jmonkeyengine.org/wiki/doku.php/jme3:advanced:statsview>

[38] Android Support in jME [online]. [cit. 2013-04-27]. Dostupné z WWW: <http://jmonkeyengine.org/wiki/doku.php/jme3:android>

[39] Signing Your Applications [online]. [cit. 2013-04-28]. Dostupné z WWW: <http://developer.android.com/tools/publishing/app-signing.html>

[40] Android, popular mobile platform [online]. [cit. 2012-04-28]. Dostupné z WWW: <http://developer.android.com/about/index.html>

# Zoznam príloh

Príloha 1. DVD so zdrojovými textami, programovou dokumentáciou a inštalačným `.apk` súborom.