Mendelova univerzita v Brně
Provozně ekonomická fakulta

Tvorba herního enginu pro vývoj her pomocí knihovny Qt

Bakalářská práce

Vedoucí práce:
Ing. Jaromír Landa, Ph.D.

Martin Hnátek

Brno 2016

Děkuji panu Ing. Jaromíru Landovi, Ph.D. že mi umožnil tuto práci vypracovat a za jeho rady a připomínky.

Čestné prohlášení

Prohlašuji, že jsem tuto práci: **Tvorba herního enginu pro vývoj her pomocí knihovny Qt**
vypracoval samostatně a veškeré použité prameny a informace jsou uvedeny v seznamu použité literatury. Souhlasím, aby moje práce byla zveřejněna v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách ve znění pozdějších předpisů, a v souladu s platnou *Směrnicí o zveřejňování vysokoškolských závěrečných prací.*
Jsem si vědom, že se na moji práci vztahuje zákon č. 121/2000 Sb., autorský zákon, a že Mendelova univerzita v Brně má právo na uzavření licenční smlouvy a užití této práce jako školního díla podle § 60 odst. 1 Autorského zákona.
Dále se zavazuji, že před sepsáním licenční smlouvy o využití díla jinou osobou (subjektem) si vyžádám písemné stanovisko univerzity o tom, že předmětná licenční smlouva není v rozporu s oprávněnými zájmy univerzity, a zavazuji se uhradit případný příspěvek na úhradu nákladů spojených se vznikem díla, a to až do jejich skutečné výše.

Brno 2016 ....

Abstract

Hnátek, Martin. Creation of game engine for game development with Qt. Bachelor Thesis. 2016

Bachelor thesis focuses on creating game engine in C++ with qt as framework and creating example appliacation based on engine. First part discuss alternative game engines and game engines in general and second part describes design and implementation of game engine and example application.

Key words:
Game engine, Qt, C++, QML, LiquidFun, box2D

Abstrakt

Hnátek, Martin. Tvorba herního enginu pro vývoj her pomocí knihovny Qt. 2016

Bakalářská práce se zabývá tvorbou herního enginu napsaného v C++ s pomocí qt frameworku a následné tvorby ukázkové aplikace založené na vytvořeném enginu. První část práce popisuje konkureční řešení a existující enginy druhá část práce diskutuje návrh a implementaci samotného enginu a ukázkové aplikace.

Klíčová slova:
Herní engine, Qt, C++, QML, LiquidFun, box2D

Obsah

1 Úvod a cíl práce 9
1.1 Úvod . . . 9
1.2 Cíl práce . . . 9

2 Definice pojmů 10
2.1 Herní engine . . . 10
2.2 Dělení . . . 10
2.2.1 2D - enginy . . . 10
2.2.2 3D - enginy . . . 10
2.2.3 2.5D . . . 11
2.2.4 Enginy se zaměřením na určitý žánr . . . 11
2.3 Konkurenční enginy založené na qml . . . 11
2.3.1 V-Play . . . 11
2.3.2 Bacon2D . . . 11
2.3.3 VoltAir . . . 11

3 Technologie 12
3.1 Fyzikální engine . . . 12
3.1.1 Liquidfun . . . 12
3.2 Qt . . . 12
3.2.1 MOC . . . 12
3.2.2 QML . . . 13
3.2.3 QT Creator . . . 13

4 Návrh řešení 15
4.1 Požadavky . . . 15
4.2 Analýza alternativních řešení . . . 15
4.2.1 Fyzika . . . 16
4.3 Navrhovaná architektura . . . 16
4.4 Typy nabízené herním enginem . . . 17
4.4.1 Typ Hra - Game . . . 17
4.4.2 Typ Scéna - Scene . . . 18
4.4.3 Typ Vrstva - Layer . . . 18
4.4.4 Typ Entita - Entity . . . 18
4.5 Typ PhysicsEntity . . . 19
4.5.1 Obdelník - RectangleFixture . . . 20
4.5.2 Kruh - Circle . . . 20
4.5.3 Polygon - Polygon . . . 20
4.5.4 Typ ParticleDefinition . . . 20
4.6 Vlastnosti částic . . . 20
4.6.1 Solidní - Solid . . . 20
4.6.2 Rigidní - Rigid . . . 20

4.6.3 Elastické - Elastic . . . 21
4.6.4 Barevné míchání . . . 21
4.6.5 Prášek - Powder . . . 21
4.6.6 Tažné . . . 21
4.6.7 Viskozit-ní . . . 21
4.6.8 Statický tlak . . . 21
4.6.9 Zeď . . . 21
4.6.10 Bariera . . . 22
4.6.11 Zombie . . . 22
4.6.12 Tvorba částic . . . 22
4.7 Podpora pro klouby . . . 23
4.7.1 Vzdálenostní kloub - Distance Joint . . . 23
4.7.2 Prizmatický kloub - Prismatic Joint . . . 23
4.7.3 Kladkový kloub - Pulley Joint . . . 23
4.7.4 Ozubený kloub - GearJoint . . . 23
4.7.5 Kolo - WheelJoint . . . 24
4.7.6 Svar - WeldJoint . . . 24
4.7.7 Lano - RopeJoint . . . 24
4.7.8 Třecí kloub - FrictionJoint . . . 24

5 Implementace 25
5.1 Možnosti . . . 25
5.1.1 Tvorba vlastního qml typu . . . 25
5.2 Kompilace . . . 26
5.2.1 Soubor qmldir . . . 27
5.2.2 Soubor qmltypes . . . 27
5.2.3 Game . . . 27
5.2.4 Scéna - Scene . . . 27
5.2.5 Vrstva - Layer . . . 28
5.2.6 Entita - Entity . . . 29
5.2.7 Manažer entit - entity manager . . . 29
5.3 Integrace s fyzikální knihovnou . . . 29
5.3.1 Fyzikální těleso . . . 30
5.3.2 Částice . . . 31
5.4 Klouby . . . 31

6 Tvorba ukázkové aplikace 33
6.1 Návrh hry . . . 33
6.2 Návrh GUI . . . 34
6.3 Implementace . . . 34
6.3.1 Gui . . . 35
6.3.2 ImageButton . . . 35
6.3.3 Vytváření objektů . . . 35

6.3.4 Implementace objektů 36
6.3.5 Částice 36
6.3.6 Klouby 37

7 Diskuze **38**
7.1 Možná rozšíření 38
7.1.1 Částice 38
7.1.2 Kompletní integrace s fyzikálním enginem 38

8 Závěr **39**

9 Reference **40**

Přílohy **43**

A CD se zdrojovými kódy a ukázkovou aplikací **44**

Seznam obrázků

Obrázek 1: Hlavní obrazovka Qt creatoru **14**

Obrázek 2: Ukázka elastických částic. **21**

Obrázek 3: Ukázka míchání barev. **22**

Obrázek 4: Potomci třídy Fixture **30**

Obrázek 5: Ukázka fyzikálního sandboxu algoodo **33**

1 Úvod a cíl práce

1.1 Úvod

Ačkoliv od tvorby první počítačové hry uběhlo skoro 70 let, počítačové hry jsou stále velmi populární. Není divu, že je velká poptávka (ESA, 2015, s. 12) pro tento druh aplikací. Společně s poptávkou roste i požadavek na nástroje, které umožní rychlý vývoj těchto aplikací, a to nejlépe pro co nejvíce platforem zároveň. Jedním z nejpodstatnějších softwarových nástrojů pro tvorbu her je právě herní engine, který se v nějaké podobě vyskytuje v každé hře, která byla kdy vyvinuta.

1.2 Cíl práce

Cílem této práce je navržení a implementace obecného herního enginu pod frameworkem QT v C++. Návrh herního enginu bude vycházet z analýzy již existujících herních enginů. Důraz bude kladen na jednoduchost použití. Celý engine bude realizován jako rozšíření deklarativního jazyka qml, který je součástí Qt frameworku a který je navržený tak, aby umožnil rychlou tvorbu aplikací bez větší ztráty výkonu.

Částí bakalářské práce bude tvorba ukázkové aplikace (hry), která bude založená na tomto enginu a která bude využívat co nejvíce možností nabízených herním enginem.

Výsledek práce bude zveřejněn jako opensource pod licencí LGPL.

2 Definice pojmů

2.1 Herní engine

Herní engine je možné definovat (WARD Jeff, 2008) jako framework, který slouží k jednodušší tvorbě interaktivních aplikací. Hlavním účelem je usnadnění tvorby her, ale najde své uplatnění i v tvorbě jednoduchých fyzikálních simulací a jiných multimediálních aplikací, které se v žádném případě za hry považovat nedají. Herní engine, jako takový se většinou snaží jeho uživatele oprostit od některých běžných operací, které jsou potřeba k vytvoření hry a značně tím usnadnit jejích uživateli vývoj hry samotné. Tyto operace mohou být docela nízkoúrovňově jako je například vytvoření okna pro hru nebo zpracování uživatelského vstupu po vysokoúrovňové koncepty, jako je správa herních objektů. Některé herní enginy spolu s sebou přináší i různé podpůrné nástroje pro tvorbu her, asi nejpopulárnějším herním enginem tohoto typu je Unity3D, který umožňuje snadnou tvorbu her.

2.2 Dělení

Herní enginy je možné rozdělit do několika kategorií, a to například na základě dimenze, ve které se má aplikace udávat nebo na základě zaměření jednotlivých enginů. Některé enginy se totiž místo obecného řešení zabývají řešením konkrétních problémů. Příkladem může být engine Wintermute (NEDOMA Jan, 2016), který se zabývá pouze tvorbou adventur. Tato práce se bude zabývat jen elementárním dělením a to na 2D a 3D enginy.

2.2.1 2D - enginy

Je druh enginů, který se zabývá čistě tvorbou dvojrozměrných her. Mohlo by se zdát, že je v dnešní době tento druh herních enginů nepotřebný a zcela nahraditelný trojrozměrnými enginy. Není to však pravda, protože s nástupem indie vyvojářů se tyto enginy staly velmi populární. Jedním z mnoha příkladů, mimo již uváděný Wintermute, je GameMaker (YoYo Games, 2015), který je velmi oblíbený především pro svojí jednoduchost.

2.2.2 3D - enginy

Jedná se o enginy, které se zaměřují na tvorbu trojrozměrných her. Jejích velkou výhodou je to, že se dají použít i na 2D hry a, to tak, že se kamera pohybuje pouze po 2 osách a tu třetí nemusí využívat. Tímto způsobem lze dosáhnout zajímavého a velmi efektního mixu 2D a 3D. Příkladem velmi populárního herního enginu je Unity 3D (Unity3D, 2015)

2.2.3 2.5D

Je specifický druh enginů, který je sice 2D, ale pomocí různých speciálních technik snaží navodit dojem trojrozměrného prostoru. Mezi nejznámější enginy tohoto druhu je id Tech pod kterým byla vyvinuta hra Doom. V tomto případě bylo požadovaného trojrozměrného efektu dosaženo pomocí techniky zvané raycasting. Mezi další běžně používane techniky patří bump mapping, normal mapping a jiné.

2.2.4 Enginy se zaměřením na určitý žánr

Některé herní enginy jsou navrženy specificky pro tvorbu her určitého žánru. To se většinou odráží v možnostech, které engine nabízí a hlavně pak v poskytovaném rozhraní (API). Jedním z mnoha takových enginů je například engine Wintermute (NEDOMA Jan, 2016), který se zaměřuje především na tvorbu adventůr.

2.3 Konkurenční enginy založené na qml

V současné existuje několik herních enginů, které používají qml. V této sekce se nachází jejích přehled.

2.3.1 V-Play

Je robustní řešení, které mimo jiné obsahuje modul pro monetizaci aplikací. Jeho hlavní výhodou je přítomnost vlastního herního editoru, který lze spustit i za běhu hry. Pro fyzikální simulaci používá Box2D. (V-Play, 2016)

2.3.2 Bacon2D

Bacon2D je opensource řešení, které je zaměřeno hlavně na vyvoj a tvorbu her do Ubuntu phone (KEVIN VanDine, 2016). Stejně jako v-play používá k simulaci fyziky knihovnu box2d jmenovitě její bindingy pro qml.

2.3.3 VoltAir

VoltAir je zajímavý opensource engine od společnosti Google, který slouží právě pro demnostraci fyzikálního enginu liqiudfun. Jeho hlavní nevýhodou je to, že byl napsán pro podporu stejnojmené plošinovky. Tomuto odpovídá i jeho návrh, který není dostatečně obecný nato, aby v něm bylo možné napsat jakoukoliv 2D hru. (Google, 2016f)

3 Technologie

Kapitola se bude z volbou a popisem vhodných technologii pro tvorbu herního enginu.

3.1 Fyzikální engine

Jíž název napovídá, že fyzikální engine se stará především o fyzikální simulaci. Fyzikální engine je důležitou součástí mnoha her, protože nejenom že nabízí unikátní herní mechaniky ve formě různých fyzikálních puzzlů a jiných herních mechanik, ale i tím, že umožňujě zvýšený realismus výsledného herního světa. Příkladem hry, která je založená čistě na jediné mechanice vycházející z fyzikálního enginu je populární mobilní hra AngryBirds. Hlavní herní mechanikou v AngryBirds.

3.1.1 Liquidfun

Knihovna liquidfun je rozšíření populárního fyzikálního enginu box2D, který přidává možnost tvorby částic. Díky tomuto lze rozšířit simulaci například o vodu, což umožňuje zavést některé zajímavé herní mechaniky. Kromě častic nabízí knihovna liquidfun také všechny možnosti fyzikálního enginu Box 2D, který umožňuje simulaci pevných objektů a klouby, které umožní spojit více objektů dohromady. Velkou výhodou fyzikálního enginu liquidfun je kromě jeho relativní nenáročnosti i množství podporovaných platforem. Kromě Windows, OS X a Linuxu totiž také podporuje Android a IOS, je použitelný jak pro hry na desktopech, tak pro mobilní platformy.

3.2 Qt

Qt je populární softwarový framework, který nabízí kolekci knihoven a nástrojů pro vývoj softwaru. Ačkoliv jeho hlavní použití je pro tvorbu grafických aplikací, obsahuje i podporu pro síťovou komunikaci, vícevláknové programování a další. Výhodou Qt je velké množství platforem pro které tento framework umožňuje a také to, že kromě jazyka c++ podporuje mnoho dalších populárních programovacích jazyků (Qt Company, 2016a). V současné době je možné Qt používat s následujícími jazyky C++, C#, Python, Java, Ring, Ruby, Basic, Ada 2005, Perl, D, Lua, Haskell, OCaml, PHP, Chicken Scheme Jak již bylo zmíněno Qt lze používat pro velké množství platforem zde je jejích výčet Linux, OS X, Windows XP a výše, QNX, VxWorks, Windows Embedded 7, Android, iOS, Windows Phone, Windows RT.

3.2.1 MOC

Je nutné podtknout, že ačkoliv je Qt napsáno v c++ nejedná se o úplně validní c++ kód, protože před jeho kompilací je použit takzvaný MOC. Zjednodušeně řečeno je MOC program, který přebírá hlavičkový soubor c++ a generuje podpůrný c++ kód na základě maker, které jsou specifické pro Qt. Toto umožňuje mimo jiné právě

přívětivou syntaxi pro systém signálů a slotů a také zlepšenou podporu pro runtime type information (RTTI), které jsou v Qt hojně využívané (Qt Company, 2016d).

3.2.2 QML

Jedná se o deklarativní programovací jazyk určený pro popis uživatelských rozhraní. Jazyk QML je příznivý i pro začínající programátory a umožňuje některé komplikovanější a méně přívětivé stránky vývoje složitějších multimediálních aplikací. Toho je dosaženo pomocí intuitivní stromové struktůry, kterou qml používá (Qt Company, 2016f).

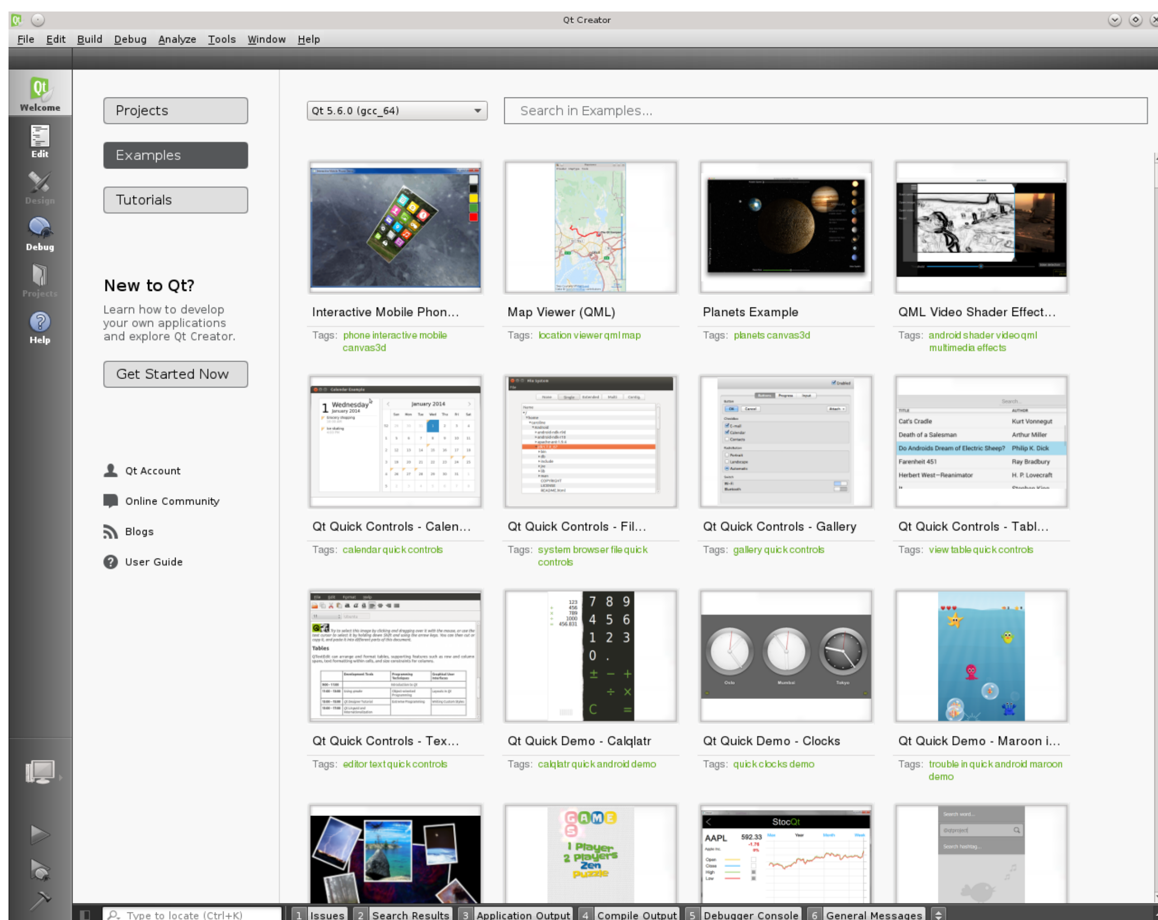
Listing 1: Minimální ukázka qml kódu. Zdroj: (Qt Company, 2016h)

```
import QtQuick 1.0

Rectangle {
    width: 200
    height: 200
    color: "blue"

    Image {
        source: "pics/logo.png"
        anchors.centerIn: parent
    }
}
```

Pro účely skriptování je pak možné použít programovacího jazyka javascript, který je plně integrován s QML. QML bylo navrženo tak, aby mohlo být snadno rozšířitelné pomocí c++.

3.2.3 QT Creator

Jak jíž bylo řečeno Qt obsahuje velké množství podpůrných aplikací. Asi nejzajímavější je Qt Creator, což je plně funkční IDE, které je určene především pro vývoj programů pro Qt ale lze jej použít i pro tvorbu jiných aplikací. Ilustrace qtCreatoru lze nálezt na obrázku 1.



Obrázek 1: Hlavní obrazovka Qt creatoru

4 Návrh řešení

4.1 Požadavky

Pokud chceme naplnit požadavky, které jsou popsány v úvodu (především pak rychlý vývoj) je nutné zvolit vhodnou abstrakci (dále jen api) mezi enginem a uživatelem. V této části se budeme zabývat popisem vhodného api a z tohoto návrhu pak vyvodíme návrh aplikace.

4.2 Analýza alternativních řešení

Ze tří herních enginů popsaných v rešeršní části práce nabízí obecné řešení pouze dva. Opensource je jen jeden z nich. S ohledem na tato fakta se v této části budeme zabývat analýzou tohoto enginu - Bacon2D. S ohledem na cíl jednoduchosti, který je definován v úvodu by bylo dobré se podívat, zda api Bacon2D tento požadavek splňuje. Níže je proto uvedená minimální aplikace, kterou lze nalézt na stránkách Bacon2D (KEVIN VanDine, 2016).

```
Game {
        id: game
        width: 400
        height: 250
        currentScene: scene

        Settings {
                id: settings
                property int highScore: 0
                property bool noSound: false
        }
        Scene {
                id: scene
                physics: true
                width: parent.width
                height: parent.height
                Entity {
                        width: parent.width
                        height: parent.height
                        updateInterval: 50
                        behavior: ScriptBehavior {
                                script: {
                                        var newPos = entity.x + 5
                                        entity.x = newPos > parent.width ? 0 : newPos
                                        console.log("update: x -> ", entity.x)
                                }
                        }
                        Rectangle {
                                width: 50; height: 50
                                color: "red"
                        }
                }
        }
}
```

Z výše uvedeného kódu se dá už od pohledu poznat co taková aplikace bude dělat, a to dokonce bez toho, aby byl uživatel seznámen s API daného enginu. Architekturu Bacon2D nelze hodnotit jinak než kladně. Jediné, co by se dalo změnit je systém chování (behaviours), který je zbytečně složitý.

4.2.1 Fyzika

Bacon2D poskytuje i abstrakci pro fyzikální simulaci. Dosahuje toho abstrakcí nad existujícím řešením v podobě pluginu Box2D QML, který poskytuje snadné rozhraní pro fyzikální engine box2D.

4.3 Navrhovaná architektura

Jak jíž bylo uvedeno, je důležité, aby dané api bylo jednoduché na použití, ale je také třeba, aby nebylo příliš omezující. Na základě těchto podmínek a po prostudování řešení poskytovaných alternativními enginy jsem došel k následujícímu návrhu. Základní architektura aplikace je inspirovaná tou, kterou nabízí engine Bacon2D. Nicméně byly provedeny určité změny a rozšíření toho, co nabízí Bacon2D. Tyto změny a rozšíření jsou posány níže.

```
Game
{
	Scene
	{
		id: Level1
		Layer
		{
			id: Popredi
			Entity{
				id: MysliciObjekt
				function tick(delta)
				{
					// logika objektu
				}
			}
			PhysicsEntity{
				bodyType: PhysicsEntity.DYNAMIC
				fixture: RectangleFixture{
					width: 50
					height: 20
				}
				Rectangle{
					width: 50
					height: 20
				}
			}
		}
	}
}
```

4.4 Typy nabízené herním enginem

V této kapitole lze nalézt výčet a popis datových typů nabízených herním enginem. Ke každému typu je uveden jak jeho význam, tak možnosti, které poskytuje uživateli. U některých složitějších konstrukcí je uveden i krátký příklad.

4.4.1 Typ Hra - Game

Celou aplikaci obaluje typ *Game*, který jí bude poskytovat informace "zvenčí". Tohoto lze dosáhnout přes několik metod. První metodou je getMousePos, která vrací poslední zaznamenanou pozici myši. Další jsou metody isLeftMouseButton- Pressed, isRightMouseButtonPressed, isMiddleMouseButtonPressed, díky kterým se může uživatel dozvědět, zda hráč drží některé z tlačítek myši. Poslední metoda, která umožňuje uživateli získat stav myši je getMouseDelta, která získá natočení kolečka myši. Datový typ *Game* samozřejmě také poskytuje možnost dotazovat se na stav klávesnice, což lze provéstpřes metodu isPressed, která jako argument přebírá identifikační číslo dané klávesy a vrací, zda byla klávesa stisknuta či nikoliv. Příslušný kód klávesy se nachází v globálním prostoru Qt, který je přístupný i z qml. Game obaluje instance datového typu Scéna. Může mít libovolné množství scén, aktivní je

však v jednu chvíli pouze jedna. Aktuální scénu lze nastavit přes numerický atribut currentScene. Neaktivní scény jsou skryty a nejsou obnovovány

4.4.2 Typ Scéna - Scene

Scéna podobně jako v Bacon2D slouží jako obal pro vrstvy. Obecně reprezentuje "stav"hry. Může se jednat například o první level nebo hlavní menu.

4.4.3 Typ Vrstva - Layer

Vrstva se dá popsat jako samostatná herní plocha, která má oddělenou fyzikální simulaci od ostatních vrstev. S ohledem na to, že každá vrstva obsahuje vlastní fyzikální svět, bylo logickým krokem nastavení tohoto fyzikálního světa poskytovat přes danou vrstvu. Toto zahrnuje nastavení gravitace s pomocí atributů gravityX a gravityY, které nastavují velikost gravitace v dané ose. Další možností, kterou vrstva nabízí pro manipulaci fyzikálního světa, je tvorba částic a manipulace s částicemi. Tvorba částic je podrobněji rozepsána v sekci částice. Pro manipulaci s částicemi slouží atribut *maximumParticleCount*, který umožňuje nastavit horní limit pro počet částic ve fyzikálním světě. Další nastavení je možné provést z typu definice částic, jehož popis lze též nalézt v sekci částice. Další možností, kterou typ vrstva nabízí je takzvaný raycasting. Raycasting umožňuje vyslat paprsek a vrátit nejbližší fyzikální těleso, které protíná tento paprsek. K tomuto účelu je určená metoda raycast, která přebírá počáteční a koncový bod, které definují paprsek. Tato metoda poté vrátí instanci typu *RaycastResult*, který obsahuje výsledek vyhledávání. Typ *RaycastResult* poskytuje dva atributy. Atribut entity obsahuje fyzikální entitu, která byla zasažena. Pokud je none, tak to označuje, že žádná entita nebyla nalezena. Atribut hitPoint, pak obsahuje informace o pozici průsečíku paprsku s tělesem.

```
var vysledek = vrstva.raycast(Qt.point(0.0, 0.0), Qt.point(800, 600))
console.log(vysledek.entity, vysledek.hitPoint)
```

Raycasting tak například umožňuje jednoduchou implementaci střelných zbraní do akčních her nebo může sloužit jako senzor pro jednoduchou AI. Další možností, kterou vrstva nabízí, je také možnost zastavit nebo spustit obnovování entit pomocí metod pause a unPause. Poslední možností, kterou vrstva nabízí je vyhledávání entit v oblasti specifikované obdelníkem. K tomuto lze použít metodu getEntitiesInRect, která přebírá specifikaci daného obdélníku a vrací pole nalezených entit.

```
var nalezene = vrstva.getEntitiesInRect(Qt.rect(0, 0, 10, 10))
for(i = 0; i < nalezene.length; i++)
{
        console.log(nalezene[i].getID());
}
```

4.4.4 Typ Entita - Entity

Každá vrstva může obsahovat libovolný počet tzv. Entit. Entita představuje "myslící"objekt ve hře. Může se jednat například o hráče, nepřítele nebo třeba mrak,

který se pohybuje konstantní rychlostí. U každé entity lze specifikovat metodu tick, která se zavolá při každém snímku (ticku) enginu. Vstupním parametrem této funkce je tzv. delta, což je počet milisekund, který uběhl od posledního snímku, podělený tisícem. Tento parametr lze použít například pro eliminaci závislosti na snímkové frekvenci. Příkladem může být případ, kdy chceme entitou posunout každý snímek o určitou předem danou vzdálenost například o 10 pixelů $x+ = 10$. Tento kód není ideální, protože se bude chovat různě v závislosti na tom, kolik snímku za vteřinu může počítač, na kterém hra běží, vykreslit. Abychom to napravili můžeme použít právě parametr delta, který nám zařídí, že bude změna dané hodnoty probíhat v závislosti na uplynulém čase místo na uplynulých snímcích $x+ = 10$ getDelta(). Další vyu žití tohoto parametru je například k jednoduchému časování, kdy suma předchozích parametrů delta udává uplynulý čas. Toto je vhodné například pro střelbu, kdy není žádoucí, aby se projektil vytvořil v každém snímku, ale například jednou za dvě sekundy. Každá entita má unikátní numerický identifikátor, který lze získat pomocí metody getID. Entita také může kdekoliv zjistit poslední deltu se kterou byla zavolána pomocí metody getDelta. Toto se hodí především pro zjišťování delty mimo metodu tick.

4.5 Typ PhysicsEntity

Druhý typ entity je fyzikální entita. U tohoto typu entity dochází před zavoláním metody tick k synchronizaci její pozice a rotace s fyzikálním objektem, ke kterému je tato entita přiřazená. Tato entita registruje kolize s ostatními fyzikálními entitami a předává je uživateli pomocí signálů sensorContactBegin a sensorContactEnd. Oba signály, pak přebírají entitu (přes parametr entity), která s danou fyzikální entitou kolidovala. Níže je uvedený příklad entity, která při započetí kolize změní barvu čtverce na červenou a při ukončení kolize smaže entitu, která do ní narazila.

```
PhysicsEntity{
        fixture: RectangleFixture{
                width: 50
                height: 50
        }
        Rectangle{
                id: ctverec
                width: 50
                height: 50
                color: "blue"
        }
        onSensorContactStart:{
                ctverec.color = "red"
        }
        onSensorContactEnd: {
                ctverec.color = "blue"
                entity.destroy()
        }
}
```

Výše je také vidět povinný atribut fixture, který definuje tvar fyzikálního tělesa. Do tohoto atributu lze vložit následující tvary:

4.5.1 Obdelník - RectangleFixture

Přiřadí fyzikálnímu tělesu tvar obdelníku. Velikost obdelníku se nastavuje pomocí atributů *width* pro šířku a *height* pro výšku.

4.5.2 Kruh - Circle

Kruh definuje kruhový tvar pro fyzikální těleso. Velikost kruhu lze nastavit pomocí atributu *radius*.

4.5.3 Polygon - Polygon

Polygon umožňuje definovat vlastní geometrii. Toto probíhá pomocí atributu *points*, který definuje body ze kterých se skládá polygon.

4.5.4 Typ ParticleDefinition

Použitá fyzikální knihovna mimo jiné umožňuje práci s částicovými systémy. Pro tento účel se používá typ ParticleDefinition, který umožňuje nadefinovat vlastnosti jednotlivých částic a poté je vytvářet. Toto se odráží v návrhu datového typu, kde v samotném typu nadefinujeme požadované údaje a poté zavoláme jednu z možných metod pro vytvoření částic.

4.6 Vlastnosti částic

Tato sekce bude pojednávat o různých vlastnostech částic, které nabízí částicový model fyzikálního enginu. Druh je určen nastavením flagu v příslušné definici. Je nutné podotknout, že v knihovně liquidfun umožňuje vytvořit jak jednotlivou částici, tak i skupinu částic. Hlavním rozdílem mezi skupinou částic a jednotlivou částicí je to, že skupině částic je možné přiřadit některé vlastnosti navíc. Jedná se především o vlastnost rigid, kterou nelze přiřadit jednotlivým částicím z důvodu algoritmu používaným fyzikálním enginem. (Google, 2016d)

4.6.1 Solidní - Solid

Pevné částice neumožňují vniknutí cizích objektů do jejích vnitřku. Pokud by nastala situace, že se nějaký objekt dostal dovnitř částice, částice jej vytlačí na svůj povrch.

4.6.2 Rigidní - Rigid

Skupina částic, která nikdy nemění svůj tvar.

4.6.3 Elastické - Elastic

Skupina částic, která umožňuje plastickou deformaci.

Obrázek 2: Ukázka elastických částic.

4.6.4 Barevné míchání

Pokud je tento flag povolen dochází při styku jednotlivých částic ke změně jejích barev. Míchání vzniká na základě následujícího vzorce $vyslednaBarva = (barvaA - barvaB) * koeficientMichani$

4.6.5 Prášek - Powder

Vlastnost, která zařídí, že se budou částice chovat jako prach.

4.6.6 Tažné

Tažné částice emulují efekt povrchového napětí. V dokumentaci se lze dočíst, že tuto vlastnost lze použít například pro simulaci kapky vody.

4.6.7 Viskozit-ní

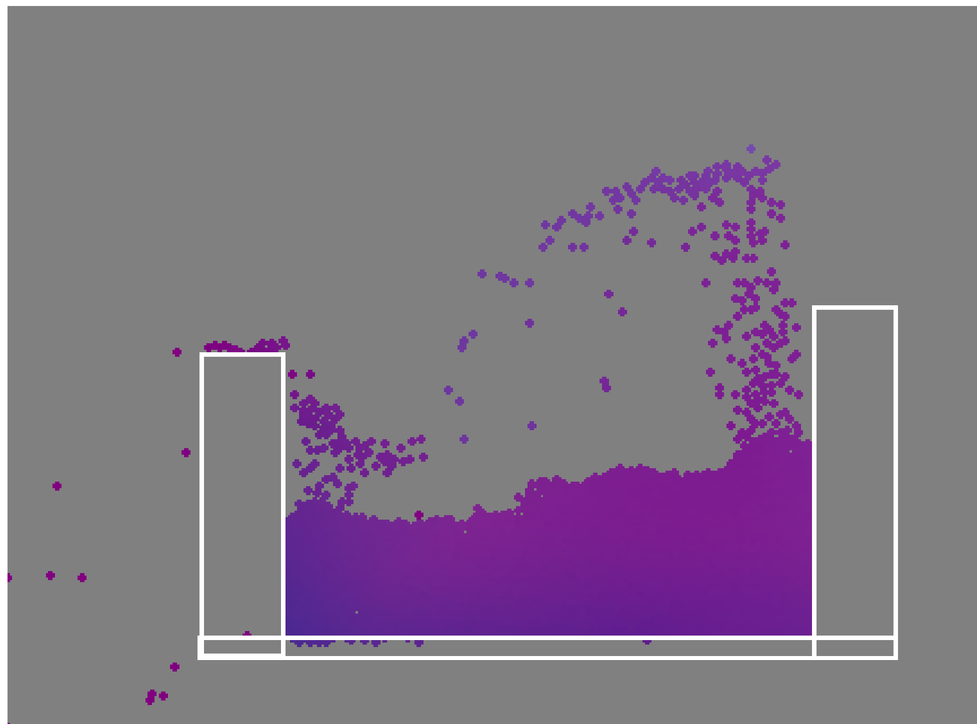
Viskozit-ní částice mají tendenci držet se u sebe. Chovají se podobně jako například olej.

4.6.8 Statický tlak

Při povolení této vlastnosti se částice pod tlakem začnou zmenšovat.

4.6.9 Zeď

Částice se chovají jako nehybná zeď.



Obrázek 3: Ukázka míchání barev.

4.6.10 Bariera

Částice, které majít tuto vlastnost nastavenou vzdorují částicím, které se do ních snaží vniknout.

4.6.11 Zombie

Zombie částice jsou zničeny ihned po jejích vytvoření.

4.6.12 Tvorba částic

Použití částic v herním enginu je docela jednoduché. Nejdříve je nutné vytvořit definici částic (typ ParticleDefinition). Typ částic lze specifikovat pomocí atributu *flags*, do kterého je možné vložit pole typů ParticleFlag, které specifikují typ částic. Částicím lze také specifikovat barvu pomocí parametru color. Další možností je specifikace takzvané síly, která specifikuje soudržnost částic. Udává se pomocí čísla v rozmezí 0.0 (nejslabší) a 1.0 (nejsilnější). (Google, 2016c) Toto nastavení je možné provést pomocí atributu *strength*.

```
ParticleDefinition{
        id:definice
        color: "red"
        strength: 0.7
        flags: [
```

```
                ParticleFlag{
                        flag: ParticleFlag.ELASTIC
                }
        ]
}
```

Tento typ je pak možné vložit do metody *createParticleBox*, která vytvoří skupinu částic o tvaru a velikosti specifikovaného obdélníku.

```
vrstva.createParticleBox(Qt.rect(0, 0, 50, 50), definice);
```

4.7 Podpora pro klouby

Klouby umožňují spolu spojit 2 fyzikální tělesa. Takto spojená tělesa navzájem ovlivňují svojí pozici nebo rotaci v závislosti na tom, jaký kloub je spojuje. Spojení proběhne jedině v případě, že jsou specifikovány obě tělesa. U každého kloubu je nutné specifikovat tělesa, které bude spojovat. Toto lze provést přes atributy bodyA a bodyB. Engine v současné době poskytuje následující klouby:

4.7.1 Vzdálenostní kloub - Distance Joint

Vzdálenostní kloub patří k nejjednodušším kloubům, které liquidfun nabízí. Tento kloub se jednoduše snaží udržovat konstantní vzdálenost mezi spojovanými tělesy. Kloub se také může chovat jako pružina a to v závislosti na nastavení dvou parametrů frequencyHz (frekvence) a dampingRation (koeficient tlumení). (Google, 2016c) Níže se nachází ukázka použití vzdálenostního kloubu.

```
DistanceJoint{
        bodyA: telesoA
        bodyB: telesoB
        frequencyHz: 50.0
        dampingRation: 1.0
}
```

4.7.2 Prizmatický kloub - Prismatic Joint

Je kloub, který umožňuje pohyb jenom po specifikované ose.

4.7.3 Kladkový kloub - Pulley Joint

Kloub, který simuluje kladku. Mimo dvou fyzikálních těles je nutné specifikovat ”strop”pro každé ze spojovaných těles.

4.7.4 Ozubený kloub - GearJoint

Je kloub, který zajišťuje, že se budou obě tělesa chovat, jako by byla ozubená kola.

4.7.5 Kolo - WheelJoint

Kloub simuluje kolo takovým způsobem, že těleso B rotuje okolo tělesa A.

4.7.6 Svar - WeldJoint

Svar zamezuje jakýkoliv relativní pohyb mezi tělesem A a tělesem B.

4.7.7 Lano - RopeJoint

Kloub, který omezuje maximální vzdálenost mezi bodem A a bodem B.

4.7.8 Třecí kloub - FrictionJoint

Kloub, který simuluje tření mezi tělesem A a tělesem B.

5 Implementace

Tato kapitola se bude zabývat samotnou implementací pluginu. Vycházející z návrhu popsaného v předchozí kapitole.

5.1 Možnosti

Nejdříve musíme zjistit jaké možnosti qt potažmo qml nabízí. Z dokumentace k qt (Qt Company, 2016f) vyplývá, že komunikace mezi qml a C++ může probíhat jedním z následujících způsobů:

- voláním qml funkce
- tvorbou qml typu z C++
- propojením pomocí signálu
- změnou atributů některého z existujících qml objektů z C++

Pro účely této práce postačí první tři.

5.1.1 Tvorba vlastního qml typu

Provádí se tvorbou nové třídy, která dědí od nějakého potomka třídy QObject, popřípadě od QObjectu samotného. Také je potřeba v definici třídy zavolat makro Q_OBJECT, které je nezbytné, pro zařazení třídy do MOC. Dále je ve třídě možné deklarovat metody pomocí makra Q_INVOKABLE, které zařídí, že danou metodu bude možné volat z qml. Další možností je použití atributů, které se deklarují s pomocí makra Q_PROPERTY a u kterých je potřeba napsat obslužné funkce (gettery a settery). Níže uvádím jednoduchý příklad, který definuje třídu MujTyp, která dědí od QQuickItem (Item v qml) a má jeden atribut nastavitelnyAtribut a jednu metodu mojeMetoda. Níže se nachází ukázková implementace.

```
class MujTyp : public QQuickItem
{
        Q_OBJECT
        Q_PROPERTY(bool nastavitelnyAtribut
        READ nastavitelnyAtribut
        WRITE nastavitelnyAtribut
        NOTIFY nastavitelnyATributChanged)
        public:
        explicit MujTyp(QQuickItem* parent = 0);
        Q_INVOKABLE void mojeMetoda(qreal parameter);
        bool nastavitelnyAtribut() const;
        void setNastavitelnyAtribut(bool hodnota);
        signals:
        void nastavitelnyAtributChanged();
};
```

Po vytvoření tohoto typu je třeba ho zaregistrovat. Dá se to provést pomocí metody qmlRegisterType v instanci hlavní třídy pluginu. První tři parametry metody představují namespace pluginu a jeho verzi. Čtvrtý pak určuje, jak se bude typ v jazyce qml jmenovat.

```
qmlRegisterType<MujTyp>("NazevPluginu", 1, 0, "MujTyp");
```

Výsledný typ by se pak dal použít v qml následujícím způsobem.

```
import NazevPluginu 1.0
MujTyp {
        id: MujTypId
        nastavitelnyAtribut: false
        onNastavitelnyAtributChanged:{
                console.log('Zmena␣atributu')
        }
}
// Volani metody
MujTypId.mojeMetoda(50.0)
```

Je nutné podotoknout, že ačkoliv je tato metoda pro registraci typů používaná nejčastěji není jediná. Příkladem další metody, která je využitá v herním enginů je metoda registerUncreatableType, který vytvoří neinstanciovatelný typ. Toto je vhodné například pro bázové třídy, které nemají samy o sobě využití.

5.2 Kompilace

Výsledek kompilace je dynamická knihovna, která bude QML aplikací načtena. Předtím než jí však bude možno použít je nutné nejdřív splnit několik podmínek. První podmínkou je nutnost přiložení souboru qmldir, který je popsaný níže. Dále je nutné vytvořit složku s názvem pluginu a tuto složku umístit do složky s pluginy. (Qt Company, 2016e)

5.2.1 Soubor qmldir

Soubor qmldir se používá pro definici qml modulu. Mimo jiné se v něm může nacházet identifikátor modulu, který specifikuje název plugin, název hlavní třídy pluginů a jiné věci bez kterých by se qt neobešlo.

5.2.2 Soubor qmltypes

Po kompilaci pluginu je vhodné vygenerovat soubor qmltypes. Soubor qmltypes popisuje typy, které daný plugin poskytuje. Ačkoliv tento krok není nutný pro používání pluginu, poskytuje QtCreatoru informace potřebné pro automatické doplňování u typů. Soubor qmltypes lze vygenerovat nástrojem *qmlplugindump]*, který jej vygeneruje na základě cesty k pluginu a názvu daného pluginu. Níže je minimální ukázka použití nástroje *qmlplugindump*.

```
qmlplugindump Umbra 1.0 Umbra/
```

Výsledný soubor je potřeba přiložit do složky s pluginem. Pokud QtCreator nenajde soubor qmltypes, pokusí se jej automaticky vygenerovat, tato procedura je však dosti náchylná k chybám a většinou neuspěje. (Qt Company, 2016e)

5.2.3 Game

Typ Game je implementován jako potomek třídy QQuickItem. Jak jíž bylo zmíněno obklopuje všechny ostatní elementy. Jeho hlavním úkolem je každý snímek zavolat na aktivní scéně metodu update a předat jí počet milisekund od minulého snímku. U QML aplikací je požadováno, aby běžely plynulých 60 snímku za vteřinu, je tedy vhodné, aby se samotný engine byl obnovován také 60 snímku za vteřinu, což je přibližně každých 16 milisekund. Její dalším úkolem je odchytávání uživatelského vstupu, který pak poskytuje přes metody *getMousePos*, *isLeftMouseButtonPressed*, *isRightMouseButtonPressed*, *isMiddleMouseButtonPressed*, *getMouseDelta*. Tohoto je dosaženo přetížením metody eventFilter, která je volaná pokaždé, když je v udanostmi smyčce registrován jakýkoliv uživatelský vstup. Aby to fungovalo je ovšem nutné třídu registrovat jako událostní. Toto se provádí přes metodu *installEventFilter* třídy QQuickWindow. Při použití této metody lze dosáhnout lepší odezvy uživatelského vstupu.

5.2.4 Scéna - Scene

Scéna obaluje všechny vrstvy a její jediný úkol je obnování všech vrstev, které se v ní nachází. Tohoto je dosaženo pomocí metody QQuickItem findChildren (Qt Company, 2016g), která umožňuje v daném objektu nalézt všechny potomky určitého typu.

```
void Scene::think(int delta)
{
        QList<Layer*> layers = findChildren<Layer*>();
        for(Layer* layer : layers)
        {
                layer->think(delta);
        }
}
```

5.2.5 Vrstva - Layer

Vrstva uchovává manažer entit, který se stará o obnovování entity a také se stará o vykreslování částic.

Vykreslování probíhá přetížením metody *updatePaintNode*, která umožňuje měnit a vytvářet geometrii prvku, který jej implementuje. Metoda funguje pomocí úpravy takzvaného scénové grafu (Scene graph). Metoda je volána pokaždé, když má dojít k překreslení grafu toto se většinou stane po zavolání metody *update* ale je možné, že si jí vykreslovací jádro qt zavolá sámo. Samotná úprava grafu scén probíhá s pomocí argumentu *oldNode*, který představuje předchozí stav překreslovaného objektu. Když se metoda zavolá poprvé má tento argument hodnotu **nullptr** proto je potřeba ho nejprve inicializovat. S ohledem na to, že typ **QSGNode** poskytuje pouze nejzákladnější metody je nutné použít některého z jeho potomků. Nejvhodnějším kandidátem je třída **QSGGeometryNode**, která umožňuje generovat geometrie dle specifických požadavků uživatele. V případě částic je potřeba rychle vykreslit velké množství barevných bodů. Tohoto lze dosáhnout vytvořením instance třídy **QSGGeometryNode** s hodnotou *QSGGeometry::defaultAttributes_ColoredPoint2D* a následnou alokací potřebného místa přes metodu allocate. Posledním krokem je získání pole bodů přes metodu *vertexDataAsPoint2D* a nastavit všem bodům vhodnou pozici a barvu a výsledný uzel vrátit. V dalších voláních není třeba metodu volat ale lze jí pouze přetypovat například pomocí funkce *static_cast* na **QSGGeometryNode** a pokračovat jako v prvním případě. (Qt Company, 2016i)

Další možnost, kterou vrstva nabízí, je jíž diskutovaný raycasting. Probíhá to s využitím metody *rayCast* u fyzikálního světa, která přejímá tak zvaný callback, což je třída, která se chová jako filtr pro zasažené objekty paprskem. (Google, 2016e) Třída *Layer* v sobě obsahuje implementaci callbacku, který vždy vrátí nejbližší objekt. Stači ho vytvořit a vložit jeho adresu do parametru metody *rayCast*. Další dva parametry reprezentují počátek a konec paprsku.

Každou vrstvu je možné pozastavit, což je provedeno pomocí metod pause a unPause. Jediné, co tyto metody dělají je to, že nastavují atribut m_paused na True/False v závislosti na zavolané metodě. Tento atribut je poté kontrolován v mětodě think pokud je nastaven na false. Tak proběhne obnovení všech entit a následné překreslení částic jinak ne. Níže je uvedená ukázka obnovování entit ve vrstvě.

```
void Layer::think(int delta)
{
        if(!m_paused)
        {
                m_entityManager.update(delta);
                update();
        }
}
```

5.2.6 Entita - Entity

Volání metody tick je dosaženo využitím metody *invokeMethod* třídy **QMetaObject**.

5.2.7 Manažer entit - entity manager

Tato třída, jak jíž název napovídá se stará o správu entit. Stará se nejenom o jejích obnovování ale také poskytuje pomocné metody pro jejích vyhledávání a mazaní. Dělá to tak, že si uchovává kolekci entit, které se zaregistrovali u příslušné vrstvy. Jelikož každá vrstva simuluje vlastní svět má také vlastní manažer entit. Manažer entit se také stará o správu fyzikálního světa především pak o jeho obnovování v metodě *update*. Pro obnovování používá techniku fixních časových kroků (FIELDLER Glenn, 2016). Tato technika umožňuje fyzikální simulaci se chovat předvídatelně i v případě přerušení programu na delší dobu z důvodu vyššího využití systémových zdrojů.

```
m_accumulator += delta / 1000.0;

while(m_accumulator >= m_step)
{
        m_world.getWorld()->Step(m_step, 8, 3, 3);
        m_accumulator -= m_step;
}
```

5.3 Integrace s fyzikální knihovnou

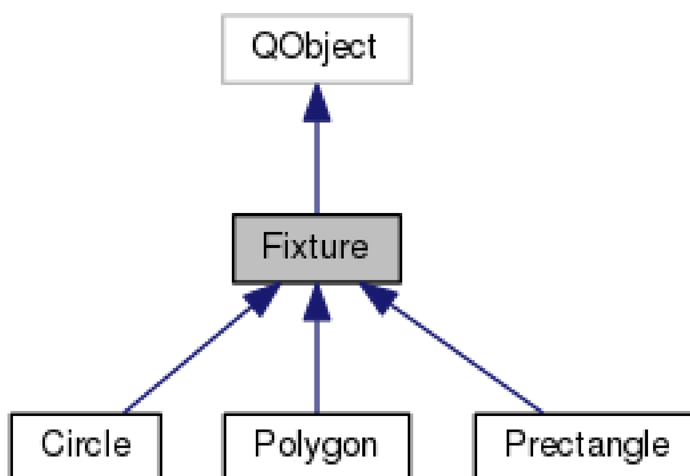
Ačkoliv fyzikální knihovna liquidfun přináší spoustu výhod je nutné si uvědomit, že jeho použití také přináší určité problémy, které je potřeba vyřešit. Prvním problémem, který přináší je fakt, že pozice, které jdou ”z”a ”do”fyzikálního enginu nejsou v pixelech ale v metrech. Protože je pro většinu aplikací poměr 1 px = 1 m nepraktický je nutné provézt převod z metrů na pixely a z pixelů na metry. O toto se automaticky stará fyzikální entita. Převodní poměr je sdílen pomocí třídy WorldWrapper, která obaluje třídu b2World (třída reprezentující svět v fyzikálním enginu liquidfun), který jedná se o implementaci návrhového vzodu přepravka (PECINOVSKÝ, Rudolf, 2007, s. 83).

5.3.1 Fyzikální těleso

Jak jíž bylo zmíněno fyzikální těleso je reprezentováno třídou PhysicsEntity, která se chová podobně jako normální entita s tím rozdílem, že před zavoláním funkce tick (pokud existuje) nastaví svojí pozici a rotaci tak, aby odpovídala fyzikálnímu tělesu, které spravuje. Další rozdíl je v tom, že poskytuje několik metod pro manipulaci s fyzikálním tělesem.

Každé fyzikální těleso potřebuje fixturu, která definuje jeho tvar a vlastnosti, jako je tření, odrazivost, hustota. Hmotnost není třeba nastavit, protože si ho těleso odvodí na základě fixtury. Ačkoliv fyzikální engine podporuje více tvarů pro jedno fyzikální tělo.(Google, 2016a) Engine popisovaném v této práci jej z důvodu jednoduchosti nepodporuje.

Pro správnou inicializaci fyzikálního tělesa já také nutné specifikovat takzvanou armaturu. Armatura popisuje tvar a fyzikální vlastnosti tělesa. V enginu je toto reprezentováno kterýmkoliv z potomků třídy Fixture.



Obrázek 4: Potomci třídy Fixture

```
PhysicsEntity{
        bodyType: PhysicsEntity.DYNAMIC
        fixture: RectangleFixture{
                width: 10
                height: 20
        }
        Rectangle{
                width: 10
                height: 20
        }
}
```

Při přiřazení armatůry dochází k její automatické registraci s fyzikálním tělesem. Při změně armatury dochází ke smazání původní a zaregistrování nové.

5.3.2 Částice

Jak jíž bylo řečeno fyzikální engine umožňuje několik druhů částic. Pro nastavení částic se používá buď instance třídy ParticleDefinition nebo ParticleGroup definition v závislosti na tom, jestli definujeme jednotlivou částici nebo skupinu částic. K definici typu se pak používají prvky enumerace ParticleType nebo ParticleGroupType. S ohledem na to, že tyto prvky jsou převeditelné na celé čislo, lze je jednoduše kombinovat s pomocí bitových operací.

Přidávání určitého flagu lze realizovat pomocí logického operátoru OR.

Listing 2: Nastavení flagu pomocí binárního operátoru OR

```
particleDef.pf |= b2_solidGroup;
```

Kontrolu, zda se daný flag nachází je možné provést s pomocí logického operátoru AND.

```
particleDef.pd != particleDef.pd & flag > 0;
```

Samotný typ ParticleDefinition pouze uchovává definici skupiny částic a poskytuje jednoduché rozhraní pro komunikaci s danou definicí.

Z důvodu jistých omezení není možné mít atribut pole enumeraci. Asi nejjednodušší je danou enumeraci obalit do třídy a poté jí obalit instancí třídy QmlListProperty se kterou se dá manipulovat jako s polem. Protože je nutné při změně tohoto atributu změnit i hodnoty v definici částic, je potřeba použít konstruktor u kterého jsou operace nad uchovávanou kolekcí specifikovány uživatelem. Při přidání nového prvku do pole je tento prvek automaticky přidán i do definice částic, kterou si současná instance uchovává.

Posledním krokem pro tvorbu částic je jejích registrace do světa. To se provádí přes metodu createParticleGroup, která přebírá definici skupiny částic a vrací skupinu částic.

```
void Layer::createParticleBox(const QRect &box,
                              ParticleDefinition* definition)
{
        auto def = definition->getParticleGroupDef();
        auto pixelsPerMeter = m_entityManager.getWorld()->getPixelsPerMeter();
        def.position.Set(box.x() / pixelsPerMeter,
                         -box.y() / pixelsPerMeter);
        b2PolygonShape* boxShape = new b2PolygonShape();
        m_particleSystem->SetDestructionByAge(def.lifetime > 0.0);
        boxShape->SetAsBox(box.width() / (pixelsPerMeter * 2.0),
                           box.height() / (pixelsPerMeter * 2.0));
        def.shape = boxShape;
        m_particleSystem->CreateParticleGroup(def);
}
```

5.4 Klouby

Každý kloub dědí od abstraktní třídy **Joint** od které přebírá atributy *bodyA* a *bodyB*, které umožňují specifikovat fyzikální tělesa určená k spojení s pomocí kloubu.

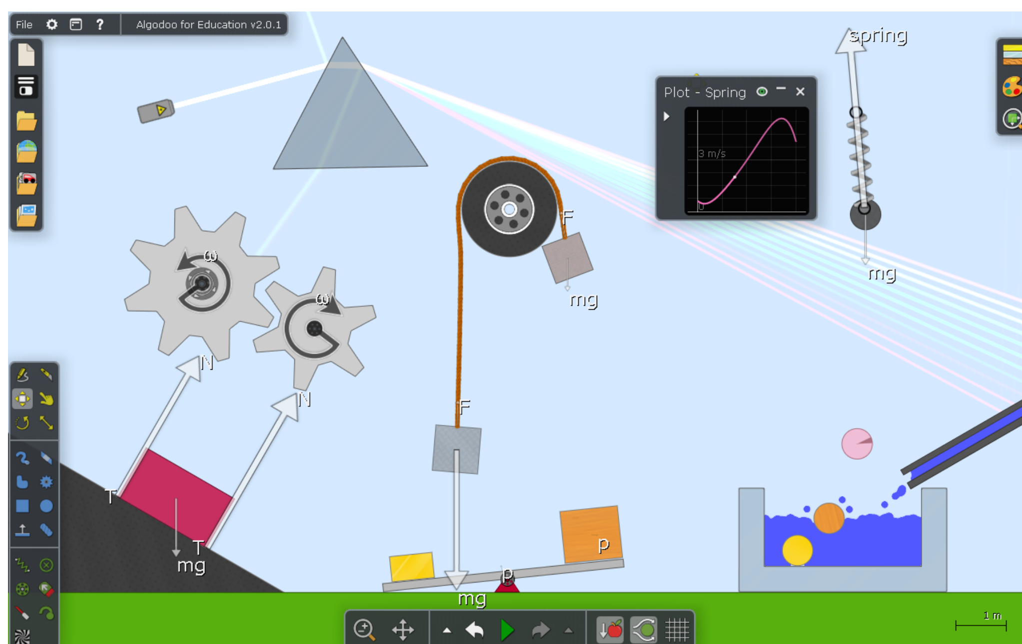
Ačkoliv existuje mnoho druhů různých kloubů všechny se inicializují obdobným způsobem. Základem je použití metody initialize u kterékoliv definice kloubu a následným zavoláním metody createJoint, která vytvoří samotný kloub. Ačkoliv je proces vytvoření nového kloubu podobný, není stejný z toho vyplývá, že implementace každého kloubu musí být jiná. To je odraženo v návrhu, kde v případě definice nového kloubu je potřeba pouze zdědit třídu Joint a přetížit metodu *linkBodies*, která se zavolá v případě potřeby inicializace nového kloubu.

Níže je uveden příklad inicializace otočného kloubu.

```
m_definition.Initialize(m_bodyA->getBody(),
                        m_bodyB->getBody(),
                        centerPoint);
m_joint = static_cast<b2Joint*>(world->getWorld()->CreateJoint(&m_definition));
```

6 Tvorba ukázkové aplikace

Pro demonstraci schopností herního enginu padla volba na 2D sandboxovou hru. Je to především proto, že tento žánr nejvíce demonstruje a vyzvedává schopnosti fyzikální knihovny liquidfun. Sandboxové hry jsou unikátní žánr her, kde je před uživateli (hráči) předvedena sada nástrojů a je jen na něm a na jeho kreativitě, jak s nimi naloží. Ačkoliv hra samotná nemá žádný cíl, má potenciál hráče zabavit na dlouhé hodiny jenom na základě jeho kreativity. Je to zároveň dobré médium, které umožní ukázat všechny důležité aspekty herního enginu. Mezi nejpopulárnější 2D fyzikální sandboxy patří algoodo, dříve též známé jako phun. Algoodo lze vidět na obrázku 7 (Algoryx, 2016), kterým se tato aplikace bude inspirovat. Aplikace dostala název Machinator.



Obrázek 5: Ukázka fyzikálního sandboxu algoodo

6.1 Návrh hry

Jak jíž bylo výše uvedeno hra by měla demonstrovat většinu, ne-li všechny vlastnosti herního enginu. To se odráží i v návrhu, proto bude hra nabízet hráči následující nástroje:

- Nástroje pro tvorbu objektů:
  - Obdelník
  - Kruh

  - Trojúhelník
  - Skupina částic ve tvaru obdelníku
- Klouby
  - Vzdálenostní kloub (distance joint)

Výběr nástrojů demonstruje všechny důležité aspekty herního enginu a umožní hráči prozkoumat jeho možnosti a to především po stránce fyziky.

Hráč bude moci jednotlivé nástroje vybrat a položit je na herní plátno. U každého objektu by pak hráč měl mít možnost změny pozice a velikosti. V jediném případě ve kterém to nepůjde jsou skupiny částic, což vychází z limitací enginu.

6.2 Návrh GUI

GUI (Graphical user interface) je jedna z nejpodstatnějších částí celé aplikace. Je to totiž první věc, které si uživatel všimne. Navíc u sandboxových aplikací většinou téměř veškerá komunikace probíhá přes gui je nezbytné, aby bylo gui pohodlné pro uživatele a zároveň aby mu moc nepřekáželo v jeho tvorbě.

Ideální gui by mělo splňovat tyto požadavky:

- Minimálnost (musí zabírat co nejmenší množství obrazovky)
- Jednoduchost
- Pohodlnost
- Přehlednost

Z důvodu naplnění jednoduchosti a minimálnosti bylo rozhodnuto použití piktogramů s vhodným kontrastním barevným schématem. Piktogramy byly vytvořeny v programu Inkscape, nástroji pro tvorbu vektorové grafiky. Samotné menu je jednoduchá lišta obsahující všechno potřebné ovládání.

Asi největší výhodou tohoto přístupu je minimalizace prostoru potřebného k zobrazení menu a tím pádem zvětšení herní plochy.

Dalším krokem je návrh kontrolního schématu. Jelikož většina ovládání vyžaduje precizní pohyby, jako je například přesun objektu z jednoho místa na druhé,jsem se rozhodl, že celé ovládací schéma postavím na pohybu myši. Každý objekt bude možné vytvořit s pomocí kliknutí levého tlačítka myši. Samotná tvorba, pak probíhá výběrem oblasti, kterou bude tvořený objekt zabírat.

6.3 Implementace

Tato sekce bude diskutovat samotnou implementaci jednotlivých prvků Machinátoru. S ohledem na demonstrační účel aplikace bude při popisu kladen důraz především na ty prvky, které využívají samotný herní engine.

6.3.1 Gui

Samotné menu je implementováno s pomocí prvku Rectangle, který obaluje různé kontrolní prvky. Jmenovitě je to níže uvedený element ImageButton, Slider z Qt.Controls, a ListView, který slouží k výběru jednotlivých nástrojů.

6.3.2 ImageButton

Je typ, který byl vytvořen specificky pro použiti v Gui. Chová se jako spínač, který má dva rozdílné stavy - zapnuto a vypnuto. Ke každému stavu je přiřazen obrázek, který se zobrazí v případě, že se spínač do něj přepne. Přepnutí probíhá kliknutím na spínač přes levé tlačítko myši. Obrázky lze nastavit přes atributy urlOn a urlOff. ImageButton také umožňuje reagovat na signál clicked, který se vyšle v případě, že došlo ke změně stavu. Níže je uvedená ukázkové použití.

```
ImageButton{
        id: pauseButton
        urlOn: "Game/Tools/pause_on.png"
        urlOff: "Game/Tools/pause_off.png"
        onClicked: {
                playButton.selected = false
                myLayer.pause()
        }
}
```

Samotná implementace je pak provedena jako prvek Item, který obaluje instanci typu MouseArea a dva obrázky, z níchž jeden je vždy skrytý, pokud MouseArea detekuje kliknutí v závislosti na tom v jakém je stavu schová jeden obrázek a zobrazí druhý. Zároveň také přehodí atribut selected, který označuje stav.

6.3.3 Vytváření objektů

Vytváření objektů je proces, který vyžaduje několik kroků. Prvně je potřeba zjistit, že chce uživatel něco vytvořit. Z návrhu ovládání vyplývá, že k tomuto dochází v případě, že hráč klikne levým tlačítkem myši kamkoliv na herní plátno. O reagování na tuto událost se stará entita *DummySpawner*, která se každý snímek pomocí metody *tick* ptá na to, zda hráč nestiskl levé tlačítko myši. Pokud ano, tak se podívá, zda uplynulo od posledního vytvoření alespoň 0.5 sekund. V případě splnění této podmínky zavolá metodu spawn, která si z gui vytáhne aktuální vybraný nástroj a vytvoří ho. Toto většinou probíhá vytvořením přiřazené entity, která má za účel zjistit velikost daného objektu a poté tento objekt vytvořit. Výběr velikosti probíhá s pomocí entity **SelectionRectangle**, která sleduje hráčovu myš do té doby než nepustí levé tlačítko myši. Poté vyvolá událost release. Tento prvek je pak použit ve výše zmíněných entitách, které pouze reagují na signál released tak, že vytvoří objekt dané velikosti. Níže se nachází ukázka takového typu a to impmlementace typu, který tvoří obdelník.

```
SelectionRectangle {
        width: 50
        id: selector
        property var component : Qt.createComponent("Crate.qml")

        onReleased: {
                if(component.status === Component.Ready)
                {
                        console.log(gui.staticBody)
                        var cm = component.incubateObject(myLayer, {
                                "x" : rect.x,
                                "y" : rect.y,
                                "width" : rect.width,
                                "height" : rect.height,
                                "bodyType" : gui.staticBody ?
                                        PhysicEntity.STATIC : PhysicEntity.DYNAMIC
                        })
                        if(cm.status != Component.Ready)
                        {
                                cm.onStatusChanged = function()
                                {
                                        selector.visible = false
                                }
                        }
                        else
                        {
                                selector.visible = false
                        }
                }
        }
}
```

S ohledem na to, že hru jako takovou lze kdykoliv zastavit pomocí tlačítka pauza je nutné, aby entita DummySpawner byla v jiné vrstvě než všechny ostatní. Při zastavení hry totiž nedochází k zastavení celé hry ale dojde k zastavení v jednotlivých vrstvách.

6.3.4 Implementace objektů

Objekty jsou poté realizovány jako fyzikální entita vhodného tvaru, která v sobě obsahuje instanci typu MouseArea, který jí umožňuje reagovat na různé podněty myši. Prvním podnětem je drag, který umožňuje plynulé přetažení objektu z jednoho místa na druhé, dalším podnětem je otočení kolečka myši, které umožňuje rotaci daného objektu. Při najetí myší nad objekt se objekt zvýrazní, aby dal uživateli najevo, že je označený a může s ním manipulovat.

6.3.5 Částice

Částice jsou implementovány podobně, jako všechny ostatní objekty s tím rozdílem, že po jejích vytvoření jíž nejsou schopny reagovat na podněty hráče.

6.3.6 Klouby

Klouby jsou výjimečné tím, že po jejích vytvoření čekají na to, až hráč klikne pravým tlačítkem myši na dva objekty a tím jím umožní se propojit. Samotné propojení pak probíhá vytvořením typu odpovídajícího kloubu a přiřazení těles na které uživatel klikl.

7 Diskuze

V práci byl vypracován moderní herní engine založený na nejnovější verzi Qt a to Qt 5.6. Jeho výhodou, oproti konkurenčním enginum svého druhu je to, že má integrovanou podporu pro fyzikální engine liquidfun, což jeho uživateli umožňuje vytvářet částicové efekty, kterých by v alternativních enginech těžko dosáhl.

Ačkoliv engine neposkytuje zcela všechny možnosti nabízené fyzikálním enginem liquidfun jeho návrh umožňuje snadné přidání zbývajících možností. A možnosti, ktere poskytuje jsou dostatečné i pro tvorbu komplexnějších her.

Schopnosti samotného herního enginu demonstruje aplikace Machinátor. Ve které si hráč může vyzkoušet všechny podstatné možnosti nabízené herním enginem. Především pak možnosti nabízené fyzikálním enginem.

Výsledné řešení i s ukázkovou aplikací bylo zveřejněno na `https://github.com/Hnatekmar/Umbra`.

7.1 Možná rozšíření

I přesto, že engine v současné době nabízí dostatečné možnosti pro tvorbu her, bylo by možné ho ještě rozšířit. Tato sekce se bude zabývat možnými rozšířeními stávajícího řešení.

7.1.1 Částice

Ačkoliv je implementace částic v současné době více než dostačující, bylo by jí možné z několika ohledů vylepšit. Prvním takovým krokem by mohlo být vylepšení vykreslovacího kódu. Ačkoliv je použití metody *updatePaintNode* pro vykreslování částic pohodlné, je zřejmé, že použitím čistého opengl by se celý proces mohl zrychlit. Dalo by se to udělat například tak, že by se pozice částic nahrály přes VBO do paměti grafické karty. Poté by se s pomocí vertex shaderu upravila pozice na obrazovce a výsledek by se obarvil s pomocí fragment shaderu. Tato metoda by pravděpodobně umožnila vykreslit větší počet částic na obrazovku.

Další možností pro rozšíření je návrh api pro lepší manipulaci s částicemi. V současné době nad nimi není kontrola moc velká.

7.1.2 Kompletní integrace s fyzikálním enginem

I přesto, že jsou všechny důležité části fyzikálního enginu integrované již v současné implementaci, některé věci, jako je například podpora pro lana (b2Rope) v enginu zatím chybí.

8 Závěr

Tato práce se zabývala tvorbou herního enginu realizovaného jako plugin do deklarativního jazyka QML a následnou tvorbou ukázkové aplikace, která demonstrovala funkčnost daného řešení. V rámci práce byl popsán návrh enginu, který vznikl na základě analýzy konkurenčních enginů, implementace daného řešení s pomocí Qt. Funkčnost celého řešení je pak ukázana implementací ukázkové aplikace Machinátor.

Hlavním přínosem práce je především tvorba moderního herního enginu postaveném na Qt, který na rozdíl od konkurenčních řešení přináší podporu částicových efekty s pomocí fyzikálního enginu liquidfun. Celý engine je pak díky jeho architektuře snadno rozšířitelný i o možnosti, které v současné době neposkytuje.

Řešení bylo následně zveřejněno jako opensource. Celý engine je dostupný komukoliv, kdo má zájem o tvorbu her.

Bakalářská práce splňuje všechny body vytyčené v zadání i v sekci Cíl práce.

9 Reference

ALGORYX. *Algoodo.* Algoodo [online]. [cit. 2016-04-25]. Dostupné z: http://www.algodoo.com/.

ESA. *ESSENTIAL FACTS ABOUT THE COMPUTER AND VIDEO GAME INDUSTRY.* ESA [online]. s. l.: ESA, 2015, s. a. [cit. 2015-12-20]. Dostupné z: http://www.theesa.com/wp-content/uploads/2015/04/ESA-Essential-Facts-2015.pdf.

FIELDLER GLENN. *Gaffer on Games | Fix your Timestep!.* Gaffer on Games [online]. s. l.: THE NETWORK PROTOCOL COMPANY, INC, s.a. [cit. 2016-05-15]. Dostupné z: http://gafferongames.com/game-physics/fix-your-timestep/.

GOOGLE *Liquidfun Programmer's Guide: Bodies.* Liquidfun Programmer's Guide: Bodies[online]. s. l.: s. n., 2016a, s. a. [cit. 2016-05-20]. Dostupné z: http://google.github.io/liquidfun/Programmers-Guide/html/md__chapter06__bodies.html.

GOOGLE *Liquidfun Programmer's Guide: Fixtures.* Liquidfun Programmer's Guide: Fixtures[online]. s. l.: s. n., 2016b, s. a. [cit. 2016-05-20]. Dostupné z: https://google.github.io/liquidfun/Programmers-Guide/html/md__chapter07__fixtures.html.

GOOGLE *Liquidfun Programmer's Guide: Joints.* Liquidfun Programmer's Guide: Joints[online]. s. l.: s. n., 2016c, s. a. [cit. 2016-05-20]. Dostupné z: https://google.github.io/liquidfun/Programmers-Guide/html/md__chapter08__joints.html.

GOOGLE *Liquidfun Programmer's Guide: Particle Module.* Liquidfun Programmer's Guide: Particle Module[online]. s. l.: s. n., 2016d, s. a. [cit. 2016-05-20]. Dostupné z: https://google.github.io/liquidfun/Programmers-Guide/html/md__chapter11__particles.html.

GOOGLE *Liquidfun Programmer's Guide: World Class.* Liquidfun Programmer's Guide: World Class[online]. s. l.: s. n., 2016e, s. a. [cit. 2016-05-20]. Dostupné z: https://google.github.io/liquidfun/Programmers-Guide/html/md__chapter10__world.html.

GOOGLE. *Voltair.* The VoltAir Project[online]. s. l.: s. n., 2016f, s. a. [cit. 2016-03-18]. Dostupné z: http://google.github.io/VoltAir/doc/main/html/index.html.

KEVIN VANDINE. *Bacon2D.* Bacon2D [online]. s. l.: s. n., 2016, s. a. [cit. 2016-02-18]. Dostupné z: http://bacon2d.com/.

NEDOMA Jan *Wintermute Engine* Wintermute Engine [online]. s. l.: s. n., 2016, s. a. [cit. 2016-02-18]. Dostupné z: http://dead-code.org/.

PECINOVSKÝ, Rudolf. *Návrhové vzory: [33 vzorových postupů pro objektové programování]*. Brno: Computer Press, 2007. ISBN 978-80-251-1582-4..

JASANI Tejas *The top 10 engines that can help you make your game.* VentureBeat [online]. s. l.: s. n., 2014, 20. 08. 2014 [cit. 2015-12-18]. Dostupné z: http://venturebeat.com/2014/08/20/the-top-10-engines-that-can-help-you-make-your-game/.

Qt Company. *Qt wiki* [online]. s. l.: s. n., 2016a, [cit. 2016-03-10]. Dostupné z: https://wiki.qt.io/About_Qt.

Qt Company. *Qt Quick Best Practices.* Qt Quick Best Practices[online]. s. l.: s. n., 2016b, [cit. 2016-05-15]. Dostupné z: https://wiki.qt.io/Qt_Quick_Best_Practices.

Qt Company. *Qt wiki* Qt wiki[online]. s. l.: s. n., 2016c, [cit. 2016-03-10]. Dostupné z: http://doc.qt.io/qt-5/qtquick-scenegraph-customgeometry-example.html.

Qt Company. *Using the Meta-Object Compiler (moc).* Using the Meta-Object Compiler (moc)[online]. s. l.: s. n., 2016d, [cit. 2016-03-15]. Dostupné z: http://doc.qt.io/qt-5/moc.html.

Qt Company. *How to Create Qt Plugin.* How to Create Qt Plugin[online]. [cit. 2016-05-10]. 2016e Dostupné z: http://doc.qt.io/qt-5/plugins-howto.html.

Qt Company. *QML Applications.* QML Applications[online]. s. l.: s. n., 2016f, [cit. 2016-03-15]. Dostupné z: http://doc.qt.io/qt-5/qmlapplications.html.

Qt Company. *QQuickItem Class.* QQuickItem Class[online]. s. l.: s. n., 2016g, [cit. 2016-03-15]. Dostupné z: http://doc.qt.io/qt-5/qquickitem.html.

Qt Company. *Qt 5.6.* Qt 5.6[online]. s. l.: s. n., 2016h, [cit. 2016-04-26]. Dostupné z: http://doc.qt.io/qt-5/.

Qt Company. *http://doc.qt.io/qt-5/qtquick-scenegraph-customgeometry-example.html* [online]. s. l.: s. n., 2016i, [cit. 2016-03-15]. Dostupné z: http://doc.qt.io/qt-5/qtquick-scenegraph-customgeometry-example.html.

V-Play. *V-Play.* V-Play[online]. s. l.: s. n., 2016, s. a. [cit. 2016-02-19]. Dostupné z: http://v-play.net/.

UNITY3D *Unity3D.* Unity3D[online]. s. l.: s. n., 2015, s. a. [cit. 2015-12-18]. Dostupné z: http://unity3d.com/.

WARD JEFF. *What is a Game Engine?.*GameCareerGuide [online]. s. l.: s. n., 2008, s. a. [cit. 2015-12-20]. Dostupné z: http://www.gamecareerguide.com/features/529/what_is_a_game_.php.

YOYO GAMES. *YoYo Games.* YoYo Games [online]. s. l.: s. n., 2015, s. a. [cit. 2015-12-18]. Dostupné z: http://www.yoyogames.com/.

Přílohy

A CD se zdrojovými kódy a ukázkovou aplikací

Součástí práce je CD se zdrojovými kódy a zkompilovanou ukázkovou aplikací Machinátor pro platformu Windows.