Jihočeská univerzita v Českých Budějovicích
Přírodovědecká fakulta

Vývoj hry typu arcade na bázi herního enginu

Bakalářská práce

Jiří Hauser

Vedoucí práce: Ing. Jan Fesl

České Budějovice 2018

Hauser, J., 2018: Vývoj hry typu arcade na bázi herního enginu. [Development of an arcade game based on a game engine. Bc. Thesis, in Czech.] – 44 p., Faculty of Science, University of South Bohemia, České Budějovice, Czech Republic.

Anotace

Účelem práce je teoreticky podrobně popsat vývoj hry pomocí herního enginu a následně tuto implementovat. V rešeršní části bude provedeno srovnání dostupných herních enginů a provedena selekce vhodného řešení pro implementaci. Implementace bude provedena v programovacím jazyce C#, návrh bude proveden objektově. Model bude znázorněn pomocí UML diagramu. Práce by měla všem jejím čtenářům poskytnout dostatečnou teoretickou průpravu pro vytvoření libovolné hry o stejném technologickém základu.

Abstract

The goal of this thesis is to describe in detail the development of a videogame using selected gaming engine and then implement this game. Theoretical part of this thesis deals with comparison and selection of a suitable gaming engine. Practical part describes the processes involved in game desing, development and implementation. Implementation is done in C# programming language utilizing object-oriented design. The application model is described using UML diagrams. This thesis should provide anyone with sufficient theoretical knowledge for creating any games on a similar technological basis.

Prohlášení

Prohlašuji, že svoji bakalářskou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

V Českých Budějovicích dne 18. 4. 2018

Jiří Hauser

Poděkování

Za odborné rady, podporu a trpělivost děkuji svému vedoucímu práce, Ing. Janu Feslovi. Za korekturu a podporu děkuji Kateřině Broncové a Janě Hauserové. Za grafický návrh hlavní postavy děkuji Štěpánu Chmelovi.

Obsah

1 Úvod **1**

2 Herní engine **3**

2.1 Komparace herních enginů 3

2.1.1 Unreal Engine 3

2.1.2 GameMaker Studio 4

2.1.3 Unity3D 4

2.2 Výběr herního enginu 4

3 Vývoj v Unity3D **5**

3.1 Scéna 6

3.2 GameObject 6

3.3 Script 7

3.4 Komponenta 9

3.4.1 Rigidbody2D 10

3.4.2 Collider2D 10

3.4.3 Sprite Renderer 11

3.5 Animace 12

3.5.1 Animation Clip 12

3.5.2 Animation Controller 13

3.5.3 Komponenta Animator 14

3.6 Kamera 15

3.7 Uživatelské rozhraní 16

3.7.1 Text 17

3.7.2 Button 18

3.7.3 Image 18

3.8 Scriptování v C# 19

3.8.1 Přístup ke komponentám a GameObjectům 19

3.8.2 Tvorba a ničení GameObjectu 21

3.8.3 Komunikace mezi scripty 21

3.8.4 Coroutines 22
3.9 Build 24

4 Praktická část **27**

5 Závěr **31**

6 Odkazy a literatura **33**

Seznam obrázků **38**

Seznam příloh **39**

Příloha A – Diagram hry UAI Journey **41**

Příloha B – Obsah přiloženého CD **43**

1 Úvod

V dnešní době je hraní her stále běžnější volnočasovou aktivitou, a proto se vývojáři snaží předkládat uživateli stále originálnější hry a cílí na různé platformy. Vývoj her je ovšem komplexní disciplínou, která v sobě spojuje prvky grafiky, hudby, programování a designu. Proto vznikají herní enginy, které pomáhají s vývojem, a vývojář se tak může soustředit na prvky, které odlišují jeho hru od jiných.

Tato práce si klade za cíl teoreticky popsat vývoj 2D hry typu arcade na bázi herního enginu a v praktické části tohoto využít a 2D arcade hru vytvořit. Na začátku práce jsou srovnány tři herní enginy a vybrán ten, který je pro účely práce nejvhodnější. Dále je popsán vývoj ve vybraném herním enginu se zaměřením na klíčové prvky důležité pro vývoj ve 2D. Podrobně je popsán editor herního enginu, herní objekty, animace, kamera, uživatelské rozhraní a scriptování.

Tato práce může sloužit jako učební materiál pro vývojáře, kteří začínají nebo chtějí začít vyvíjet 2D hry s pomocí herního enginu. Česká literatura se této problematice příliš nevěnuje, a proto jsou vývojáři odkázáni na cizojazyčné prameny, nebo oficiální manuál herního enginu, který je ovšem velmi rozsáhlý.

V práci nejsou překládány některé termíny do češtiny z důvodu zachování názvosloví užívané herním enginem.

2 Herní engine

Při vývoji hry od úplného základu se řeší implementace mechanizmů, jako jsou např. reprezentace grafických objektů ve scénách, vykreslování grafiky, zvuky, kolize objektů a jejich fyzika. Tyto aspekty jsou pro většinu her společné, ať už se jedná o RPG, strategii či arkádu. Není tedy třeba toto implementovat pro každou hru zvlášť a je výhodné použít již vytvořený software. Takovému softwaru říkáme herní engine.

Herních enginů je mnoho. Některé jsou k dispozici zdarma, jiné jsou za poplatek. Existují i specializované jen na vývoj ve 2D nebo 3D. Tato práce má ovšem za úkol vytvořit arkádovou hru ve 2D, a proto bude vybrán herní engine, který toto podporuje.

2.1 Komparace herních enginů

V práci budou srovnány tři vybrané herní enginy - *Unreal Engine*, *GameMaker Studio* a *Unity3D*. Komparace bude zaměřena na specifické aspekty, a to podporu 2D, programovací jazyk, podporu platforem, popularitu, cenu a nabízené funkce navíc. Na základě srovnání bude vybrán herní engine, který bude pro účely této práce nejvhodnější.

2.1.1 Unreal Engine

V roce 1998 vytvořila firma *Epic Games* svůj herní engine s názvem *Unreal Engine*. Podporuje vývoj ve 3D i 2D. Programovacím jazykem je *C++*. Vývojové prostředí lze spustit na *Windows* a *MacOS*. Podporuje celou řadu platforem od běžných desktopových jako je *Windows*, *MacOS* nebo *Linux*, přes mobilní *Android* a *iOS*, až nově k virtuálním realitám. Herní engine nabízí vizuální scriptovací nástroj *Blueprint*, který umožňuje prototypovat, vyvíjet a vydávat hry bez nutnosti psaní kódu. *Unreal Engine* je zdarma, ovšem po vydání hry si žádá 5% ze zisku.[1]

2.1.2 GameMaker Studio

GameMaker Studio vytvořila v roce 1999 firma *YOYO Games*. Tento engine podporuje pouze vývoj ve 2D. Programovacím jazykem je *GML* (Game Maker Language), což je vlastní programovací jazyk založený na programovacím jazyce *C*. Vývojové prostředí je pro *Windows* a *MacOS*. Podporuje také většinu z hlavních platforem, kromě virtuální reality. Nabízí vlastní obrázkový editor a *room editor* na vytváření herních úrovní. Cena se pohybuje v rozmezí 99–1500 USD v závislosti na požadované platformě a délce licencování.[2]

2.1.3 Unity3D

Firma *Unity Technologies* vytvořila herní engine *Unity3D* v roce 2005. Navzdory svému názvu podporuje kromě 3D plně i vývoj ve 2D. Programovacím jazykem je *C#* nebo *JavaScript*. Vývojové prostředí je, stejně jako u předchozích dvou zmíněných enginů, pro *Windows* a *MacOS*. Podporuje více než 25 platforem. Stává se tak herním enginem s největší podporou různých platforem od běžných desktopových a mobilních, přes konzole až k chytrým televizím a virtuálním realitám. Nabízí All-In-One editor, který si může uživatel rozšířit a přizpůsobit. Vlastní obchod rozšíření, kódů, assetů, hudby a dalších. *Unity3D* je zdarma pro jedince nebo firmy s hrubým ročním příjmem do 100 000 USD.[3]

2.2 Výběr herního enginu

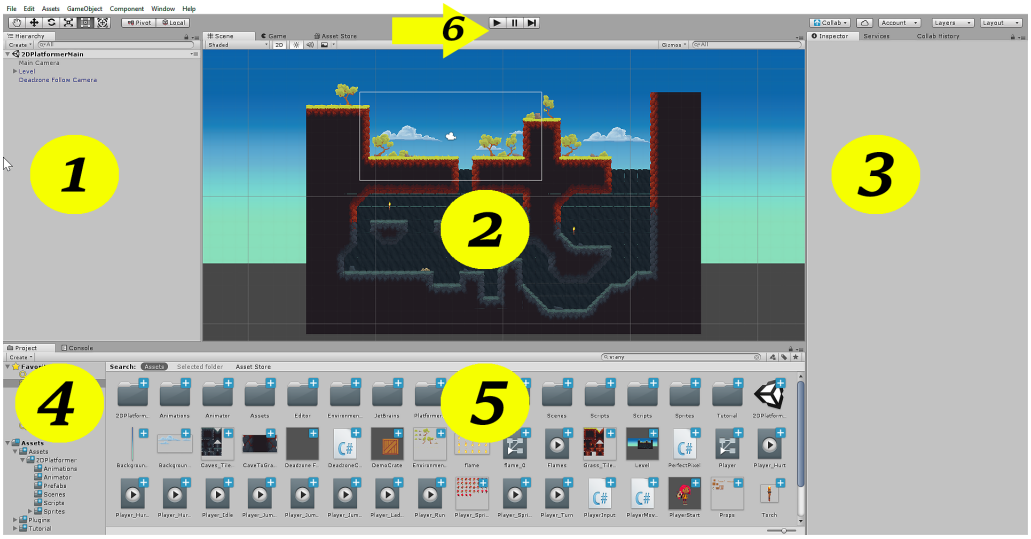
Z uvedených enginů bude použit *Unity3D*, protože splňuje všechny požadavky, které byly na začátku kapitoly stanoveny. Oproti ostatním uvažovaným nabízí vyšší programovací jazyk a největší podporu platforem. V roce 2016 bylo 34 % z prvních 1000 mobilních her zdarma vytvořeno v *Unity3D* [4]. Jedná se tak o nejpoužívanější a nejpopulárnější herní engine pro vývoj na mobilních platformách.

Obrázek 2.1: Logo Unity3D [II]

3 Vývoj v Unity3D

V této kapitole bude přiblíženo vývojové prostředí herního enginu *Unity3D* (dále jen Unity) a jeho nejdůležitější části. Tato práce popisuje Unity ve verzi 2017.3.

Vývoj v Unity lze rozdělit do dvou kategorií. Tvorba scény a specifikace chování hry. Tvorba scény se provádí v editoru, který se objeví po vytvoření nového projektu. Zde se mohou specifikovat veškeré aspekty týkající se vzhledu hry, jako jsou např. herní objekty, animace, světla a jiné.



Obrázek 3.1: Editor

Na obrázku 3.1 je zobrazen editor a jeho části. Nalevo (pod číslem 1) se nachází hierarchie herních objektů (*GameObject*). Uprostřed (pod číslem 2) je vidět herní scéna. Všechny herní objekty jsou zde reprezentovány v grafické podobě. Při selekci herního objektu se objeví jeho jednotlivé komponenty v *inspektoru*, který je umístěn napravo (pod číslem 3). Souborový systém našeho projektu je vidět ve stromové struktuře v dolním levém rohu (pod číslem 4). Tento systém se souhrnně označuje jako *Assets* a jedná se o veškeré soubory, které jsou součástí projektu. Detail každé složky souborového systému se zobrazí po její selekci, uprostřed v dolní části editoru (pod číslem 5). Uprostřed, v horní části editoru (pod číslem 6), se nachází ovládací

tlačítka. Zleva prvním tlačítkem *play* se editor přepne do herního režimu a zobrazí se hra, která poté může být hrána a testována. Opětovným kliknutím tohoto tlačítka se editor přepne zpět. Ostatní ovládací tlačítka slouží k pozastavení hry a k jejímu krokování po snímcích.

Chování hry se implementuje v programovacím jazyce. Tato práce bude používat programovací jazyk *C#*.

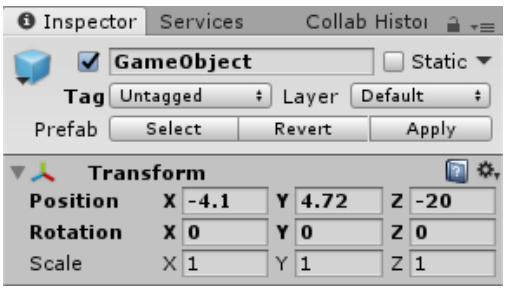
3.1 Scéna

Prostředí, ve kterém jsou vidět všechny herní objekty, se nazývá scéna. Vzhled hry ve scéně odpovídá vzhledu hry při hraní. Herní objekty mohou ve scéně měnit svou polohu, velikost nebo rotaci. Scénu lze tlačítkem v horním panelu přepnout na 2D nebo 3D zobrazení. Pro účely této práce je třeba přepnout scénu na 2D zobrazení. Scény se mohou tvořit a může se mezi nimi přecházet. Při změně scény se všechny herní objekty zničí a načtou se herní objekty ze scény, do které přecházíme. Toto se ovšem dá programátorsky ošetřit a definovat herní objekty, které přetrvávají i přes změnu scény. Každá část hry se může považovat za jednu scénu. Například pro hlavní menu a jednotlivé úrovně hry se vytvoří vlastní scény.

3.2 GameObject

Herní objekt (dále *GameObject*) je základní stavební kámen vývoje her v Unity. Každý objekt ve scéně je *GameObject*. Dle definice se jedná o základní třídu pro všechny entity ve scéně.[7] *GameObjectu* můžeme nastavit jméno, tag a vrstvu. Jménem je *GameObject* reprezentován v *hierarchii*. Tag slouží jako reference na *GameObject*. Více *GameObjectů* může mít stejný tag, tím se dá přistupovat k více *GameObjectům* jako ke skupině. Zařazením do vrstvy se *GameObject* izoluje od interakce s objekty v jiné vrstvě. *GameObject* se může aktivovat nebo deaktivovat zaškrtnutím pole nalevo od jeho názvu v *inspektoru*.

Ke *GameObjectu* se dají připojit komponenty. Komponenty definují a specializují *GameObject*. Každý prázdný *GameObject* má základní komponentu *Transform*, která definuje jeho pozici, rotaci a velikost. Kromě předpřipravených komponent jako např. *sprite renderer* a *box collider 2D* – popsáno níže v sekci 3.4 – můžeme tvořit i vlastní komponenty - *scripty*.



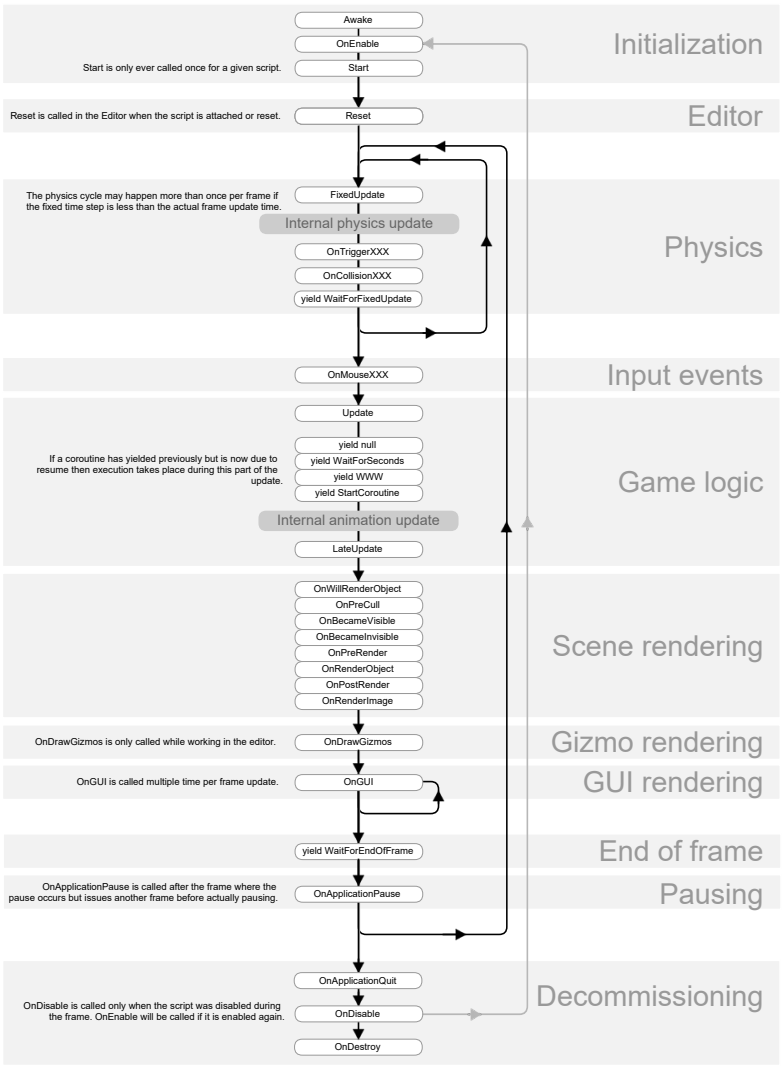
Obrázek 3.2: GameObject a komponenta Transform

GameObjecty se tvoří tlačítkem *create* v horní liště hierarchie nebo přetažením do scény ze souborového sytému. *GameObjecty* se do sebe mohou vnořovat. Vnořený *GameObject* má relativní pozici vůči svému rodičovskému *GameObjectu*. To znamená, že pokud se bude rodičovský *GameObject* pohybovat, jeho vnořené *GameObjecty* se budou pohybovat s ním.

Pokud se *GameObject* zkopíruje z *hierarchie* do souborového systému, stane se z něj tvz. *Prefab* - tedy předpřipravený *GameObject*. *Prefab* se takto může znovu použít i v jiných scénách a není nutné znovu tvořit *GameObject* včetně všech jeho komponent. Je tedy výhodné z často používaných *GameObjectů* udělat *Prefab*. Když se upraví *Prefab*, upraví se tím i všechny *GameObjecty*, které byly z *Prefabu* vytvořené.

3.3 Script

Script je kód napsaný v programovacím jazyce *C#* nebo *Javascript*, který se chová jako komponenta. Aby se dal *script* připojit ke *GameObjectu*, musí dědit z třídy *MonoBehaviour*. Tato třída nabízí virtuální funkce, které řídí životní cyklus aplikace. Na obrázku 3.3 je zobrazen diagram znázorňující životní cyklus aplikace a pořadí, ve kterém se funkce volají.

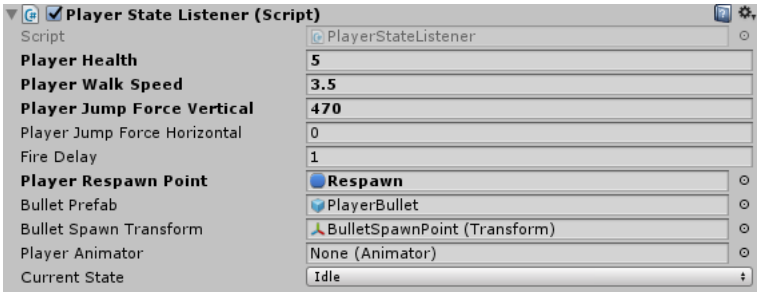


Obrázek 3.3: Životní cyklus Unity [I]

Ihned po startu scény se zavolá funkce `Awake()`. Před zobrazením prvního snímku (*frame*) se zavolá funkce `Start()`. Obě tyto funkce se zavolají právě jednou pro každý *GameObject*, ke kterému je připojen *script* s těmito funkcemi. Je výhodné inicializovat proměnné a atributy právě v těchto funkcích, protože inicializace proběhne před začátkem aktualizační smyčky a nezatěžuje tedy výpočetní výkon při hraní hry.

Během každého snímku se volají aktualizační funkce. Funkce `Update()` se provede během snímku jen jednou a slouží k aktualizaci chovaní *GameObjectu*. Funkce `FixedUpdate()` slouží ke změnám fyziky, pokud *GameObject* fyziku využívá. Funkce `LateUpdate()` se zavolá jako poslední aktualizační funkce. Specifikuje se v ní chování, které se má stát po `Update()` nebo po aktualizaci grafiky. Pokud je *GameObject* např. voják a vysílá ze špičky zbraně paprsek, je vhodné aktualizovat pozici vojáka ve funkci `Update()` a pozici paprsku ve funkci `LateUpdate()`, aby se paprsek vždy vysílal ze špičky zbraně. Ve funkci `OnGUI()` se mohou specifikovat prvky uživatelského rozhraní.

Při aktivaci objektu se zavolá funkce `OnEnable()` a při deaktivaci `OnDisable()`. Funkce `OnDisable()` se zavolá pouze, pokud byl předtím objekt aktivován. Před zničením *scriptu* se zavolá `OnDestroy()`. Tato funkce se tedy zavolá i tehdy, pokud zničíme *GameObject*, ke kterému je *script* připojený.



Obrázek 3.4: Script jako komponenta

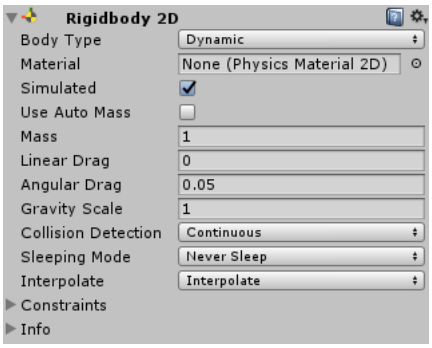
Script jako komponenta by měl mít, stejně jako ostantní komponenty, interaktivní pole, které se zobrazují v editoru. Tak jako je vidět na obrázku 3.4. Tato pole slouží k počáteční inicializaci atributů. Pokud se atribut ve *scriptu* označí jako `public` nebo označí tagem `[SerializeField]`, zpřístupní se v editoru a tím vzniknou interaktivní pole atributu. Přetažením *GameObjectu* nebo komponenty do pole atributu s odpovídajícím datovým typem, mohou takto vznikat i reference na tyto objekty.

3.4 Komponenta

Jak již bylo v minulých sekcích této práce zmíněno, komponenty definují a specializují *GameObject*. Unity nabízí sadu již předpřipravených komponent. Dále budou popsány některé komponenty, které jsou pro tvorbu 2D arkádové hry nezbytné.

3.4.1 Rigidbody2D

Komponenta *Rigidbody2D* zahrne *GameObject* pod kontrolu fyzikálního enginu Unity. Na *GameObject* začne působit gravitace a ostatní síly, které se mohou programátorsky definovat. Na obrázku 3.5 jsou zobrazeny atributy, které specifikují chování fyziky na *GameObject*.



Obrázek 3.5: Rigidbody2D

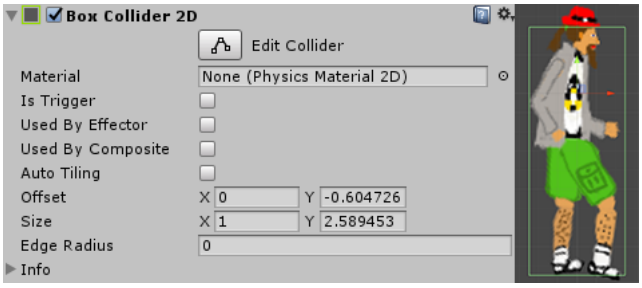
Důležitým atributem je *Body Type*, který se dá nastavit na *Dynamic*, *Kinematic* a *Static*. *Dynamic* je pod plnou kontrolou fyzikálního enginu Unity a působí na něj gravitace i ostatní síly. Na *Body Type Kinematic* působí fyzikální engine pod kontrolou uživatele a ve výchozím stavu na něj nepůsobí gravitace ani ostatní síly. *Static Body Type* je navrhnut pro nepohyblivé objekty. Dalšími atributy se nastaví dodatečné chování fyziky objektu, jako např. hmotnost, působení gravitace nebo omezení pozice a rotace. Komponenta *Rigidbody2D* je nezbytná k monitorování kolizí s ostatními objekty.[8]

Ve výchozím stavu se fyzikální engine Unity chová realisticky v porovnání se skutečným světem. Ovšem v arkádové hře uživatelé očekávají upravené chování fyziky. Hlavní hrdina skáče dál i výš a padá pomaleji než průměrný člověk. Je tedy třeba upravit atributy komponenty *Rigidbody2D*, aby se docílilo požadovaného výsledku.

3.4.2 Collider2D

Komponenty typu *Collider* definují tvar objektu pro účely fyzické kolize. Kolize dvou objektů nastává při střetu jejich komponent typu *Collider2D*. Rozlišuje se více *Colliderů* podle tvaru objektu - *Box Collider 2D*, *Circle Collider 2D*, *Polygon Collider 2D* a další. [9] Na obrázku 3.6 je zobrazena komponenta *Box Collider 2D* a její reprezentace ve scéně. *Box Collider 2D* obalí zeleným rámečkem *GameObject*,

který je tímto zahrnut do kolizního systému. Když nyní do *GameObjectu* narazí jiný *GameObject* s komponentou typu *Collider2D*, zasáhne fyzikální systém Unity a vypočítají se síly, které na oba objekty zapůsobí.

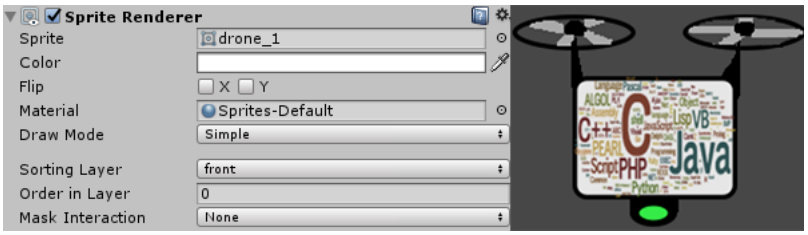


Obrázek 3.6: Box Collider 2D

Často má *GameObject* více komponent typu *Collider2D*, kde každá má jiný účel. K tomuto slouží atribut *Is Trigger*, který určuje, zda *Collider2D* slouží jako spouštěč nějaké akce. Takový *Collider2D* poté fyzicky nekoliduje s ostatními *Collidery2D*, ale pouze zaznamenává kolize v reálném čase. Programátorsky na tyto události reaguje funkce `OnTriggerEnter(Collider)[10]` ze třídy *MonoBehaviour*, kde se specifikuje požadovaná akce. Například kulka vypálená ze zbraně, po střetu s nepřítelem, sníží jeho zdraví a sama se při nárazu zničí.

3.4.3 Sprite Renderer

Z názvu komponenty *Sprite Renderer* vyplývá, že vykresluje *sprite*. *Sprite* je 2D grafický objekt[11]. Zjednodušeně řečeno je *sprite* 2D obrázek. Na obrázku 3.7 je zobrazena komponenta *Sprite Renderer* a *sprite*, který vykresluje. V této komponentě se může specifikovat *sprite*, který se má vykreslit a jeho barevný odstín. *Sprite* můžeme převrátit podle osy x nebo y atributem *Flip*. Ve scéně se často obrázky překrývají, jejich pořadí definujeme zařazením obrázků do vrstev a jejich pořadím ve vrstvě. K tomu slouží atributy *Sorting Layer* a *Order in Layer*. Vrstvy se mohou vytvářet a obecně platí pravidlo, že vrstvy s vyšším pořadovým číslem, překrývají vrstvy s nižším pořadovým číslem. *Order in Layer* specifikuje pořadí obrázků ve specifikované vrstvě a platí stejné pravidlo pro překrývání jako u vrstev.



Obrázek 3.7: Sprite Renderer a Sprite

3.5 Animace

Pokud by se běžící člověk vyfotil několikrát za vteřinu, vznikly by snímky zaznamenávající jeho pohyb v čase. Kdyby se tyto snímky poté vzali a v přesném pořadí zaměňovaly, vytvořila by se iluze pohybu člověka. Tomuto říkáme animace. Stejným způsobem se tvoří animace v Unity. Snímkům s pohybem říkáme *Sprite Sheet*, což je soubor, obvykle s příponou `.png`, který obsahuje více snímků, stejně jako je zobrazeno na obrázku 3.8. Unity nabízí vlastní *Sprite Editor*, který umí automaticky oříznout jednotlivé snímky podle jejich alfa kanálu. Po oříznutí vzniknou z jednotlivých snímků obrázky (*sprite*), z kterých můžeme udělat animaci.

Obrázek 3.8: Sprite Sheet

Animace obecně oživují hry všech žánrů. Hráč je zvyklý, že hrdina může chodit, krčit se nebo například plavat a není to jen neměnný obrázek, který se pohybuje po herní scéně. Pro tyto potřeby Unity nabízí propracovaný animační systém, který se skládá ze třech základních pilířů - *Animation Clip*, *Animation Controller* a komponenta *Animator*. Animačnímu systému Unity se také přezdívá *Mecanim[12].*

3.5.1 Animation Clip

Animation Clip neboli animační klip je základním stavebním prvkem Animace. Každý jednotlivý klip by měl obsahovat právě jednu animovanou akci. Když se například animuje pohyb hráče, měl by jeho běh mít vlastní klip a stejně tak i např. skok, krčení a padání.

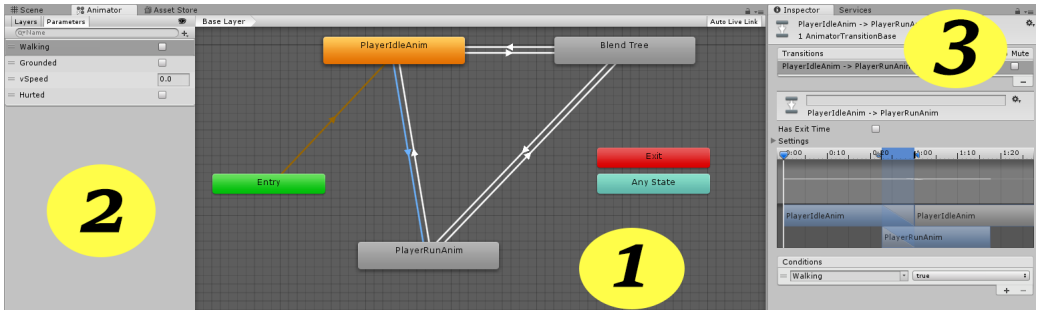
Animovat v Unity lze téměř vše. Animace se ukládají do souborů s příponou `.anim` a mohou se importovat z externích zdrojů nebo vytvářet v editoru. Tato práce vytváří animace pouze v editoru. Animace se v editoru Unity vytvářejí v prostředí nazvaném *Dope Sheet,* jak lze vidět na obrázku 3.9. Toto prostředí je velmi mocné a dovoluje řídit celou animaci tak, že uživatel má kontrolu nad každým snímkem animace. Animovat lze, kromě samotných obrázků (*sprite*), téměř všechny změny týkající se grafiky.[21]

Obrázek 3.9: Dope Sheet

Základní obrázek se vloží do scény, a tím se z něj stane *GameObject*. Animace se začne nahrávat po kliknutí na červené kolečko v horní liště prostředí *Dope Sheet*. Poté se zaznamenají všechny změny týkající se obrázku v inspektoru a ve scéně. Mohou se tedy animovat i změny pozice, rotace, měřítka, barev a dokonce i aktivování či deaktivování obrázku. Tímto lze kontrolovat, který obrázek ze *Sprite Sheet* je použit pro jednotlivý snímek animace. Na časové ose se změny promítají jako kosočtverce a vertikální čára slouží jako ukazatel časové pozice, kam se změny zapisují. Skončení animace se zaznamená opětovným kliknutím na červené kolečko. Animaci lze spustit tlačítkem *play* (ikona šipky) v horní liště prostředí *Dope Sheet*. Po spuštění je animace přehrávána a objekt je animován ve scéně.

3.5.2 Animation Controller

Animation Controller (dále jen *controller*) je stavový automat, kde stavy reprezentují animace (*Animation Clip*) a přechody mezi nimi jsou řízeny podmínkami a parametry. Animace nemůže být přehrána bez *controlleru*, zatímco *controller* může obsahovat více animací. *Controller* se uloží do souboru s příponou `.controller`. Na obrázku 3.10 je zobrazen *controller* a části, ze kterých se skládá.



Obrázek 3.10: Animation Controller

Uprostřed, pod číslem 1, jsou zobrazeny jednotlivé animace jako stavy stavového automatu a přechody mezi nimi jako orientované šipky. Oranžový stav reprezentuje výchozí bod po startu hry. Každý stav nemusí být nutně animace, ale může být

prázdný stav, reprezentující určitý stav hry. Unity předkládá již vytvořené stavy *Entry*, *Exit* a *Any State*, které definují speciální chování *controlleru* po startu hry nebo určují, kdy má skončit. Vlevo, pod číslem 2, jsou vidět parametry. Parametry jsou proměnné typu `int`, `float`, `bool` nebo `trigger`, které se vytvoří po stisknutí ikony plus v horní části, označené číslem 2. K těmto proměnným je umožněn programátorský přístup a je tedy možné měnit jejich hodnoty pomocí *scriptů*. Pro přístup k těmto proměnným a nastavení jejich hodnoty se využívají funkce

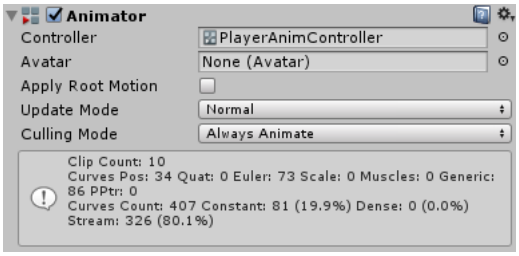
- `Animator.SetBool(string name, bool value)`
- `Animator.SetFloat(string name, bool value)`
- `Animator.SetInteger(string name, bool value)`
- `Animator.SetTrigger(string name)`[13]

Po kliknutí na přechod mezi stavy (orientovaná šipka) se zobrazí v *inspektoru* jeho detail (pod číslem 3), kde se pomocí parametrů definují podmínky pro uskutečnění přechodu mezi stavy.[21]

Příkladem může být monstrum a hráčův meč. V *controlleru* animací monstra se vytvoří parametr typu `trigger` s názvem *smrt* a definuje se podmínka, využívající tento parametr pro přechod do animace smrti monstra. Zároveň se nastaví přechod ze stavu animace smrt do stavu *Exit* a nespecifikuje se podmínka pro tento přechod. Tím se řekne *controlleru*, že skončí, po přehrání animace smrt. Ve *scriptu* připojeném na *GameObject* monstra se kontroluje kolize meče a monstra a pokud k takové kolizi dojde, funkcí `Animator.SetTrigger(„smrt")`nastavíme trigger *smrt* a dojde k přechodu do animace smrt.

3.5.3 Komponenta Animator

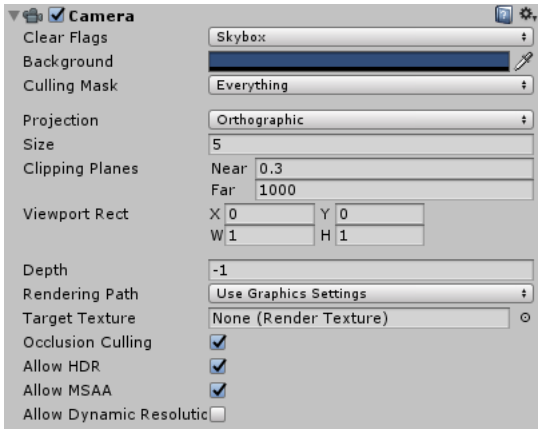
Aby se uplatnily vytvořené animace připravené v *controlleru*, je třeba je připnout na *GameObject*. K tomu slouží komponenta *Animator*, která je zobrazena na obrázku 3.11. Jediný atribut, který je důležitý pro tuto komponentu při vytváření hry ve 2D, je *Controller*. Tento atribut specifikuje *Animation Controller*, který bude použit pro animaci *GameObjectu*. Ostatní atributy se využijí jen pro 3D humanoidní modely a nejsou proto pro tuto práci relevantní.[21]



Obrázek 3.11: Animator

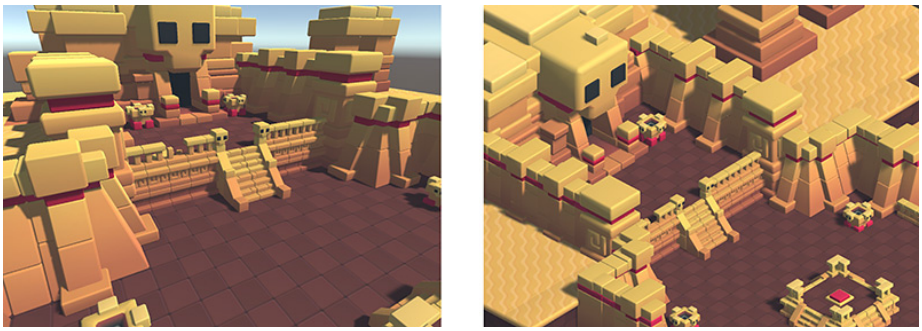
3.6 Kamera

Kamera je prostředek, který zachytává a zobrazuje herní svět hráči.[15] V Unity je kamera každý *GameObject* s komponentou *Camera*. Tato komponenta je vidět na obrázku 3.12. Kamer může být více a každá může sloužit k jinému účelu a zachytávat jinou část scény.



Obrázek 3.12: Camera

Při vývoji hry ve 2D je u komponenty *Camera* nejdůležitější atribut *Projection*. Tento atribut definuje, jak bude kamera renderovat objekty. Může se nastavit na *Orthographic* nebo *Perspective*. Na obrázku 3.13 jsou zobrazeny obě nastavení kamery a jejich rozdílné vykreslení té samé scény. *Orthographic* kamera nevykresluje hloubku.[23] Nezáleží tedy jak daleko se od kamery pozorovaný objekt nachází, protože bude vykreslen vždy stejně velký. Na druhou stranu *Perspective* kamera vykresluje scénu a objekty včetně jejich hloubky. Scéna pak vypadá více realisticky. Při vývoji 2D hry je důležité nastavit kameru jako *Orthographic*.



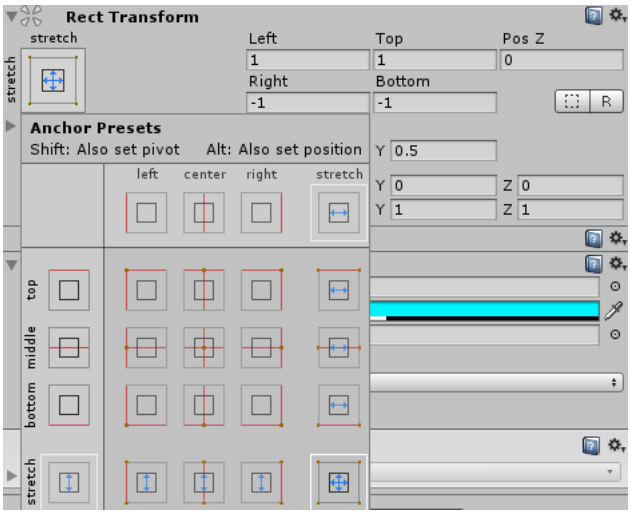
Obrázek 3.13: Perspective (vlevo) and Orthographic (vpravo) camera [III]

Atribut *Size* definuje velikost záběru kamery.[15] Čím je velikost větší, tím větší část scény hráč uvidí. Atribut *BackGround* určuje barvu, která bude zobrazena na místech, kde nebude žádný vykreslitelný objekt. Atributem *Target Texture* se nastaví výstup, který kamera použije pro renderování. Tohoto se využije například ve chvíli, kdy se na obrazovce zobrazuje i minimapa. V takovémto případě je minimapa výstup, který je nastaven v atributu *Target Texture* separátní kamery, která na scénu nahlíží z výšky.

3.7 Uživatelské rozhraní

Uživatelské rozhraní neboli *User Interface* (UI) představuje způsob interakce a zobrazování informací uživateli skrze grafické prostředky. Unity nabízí intuitivní systém UI od verze 4.6.[21]Pro přidání prvku UI do scény se musí nejprve vytvořit *Canvas*.

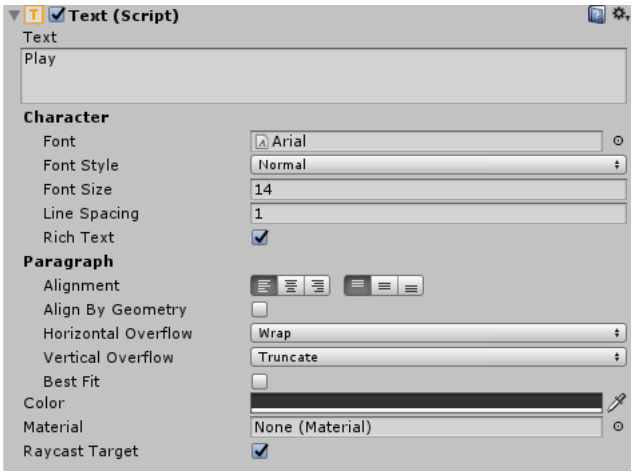
Canvas je *GameObject*, který ve scéně definuje oblast, kam můžeme vkládat UI prvky.[22] Pokud do hierarchie rovnou vložíme některý UI prvek, *Canvas* se automaticky vytvoří jako jeho rodičovský objekt. Ve výchozím stavu zabírá *Canvas* celou obrazovku. UI prvků je mnoho, ale pro účely této práce nám postačí *Text*, *Button* a *Image*. Všechny UI prvky mají komponentu *Rect Tranform*, která specifikuje pozici a ukotvení prvku vzhledem k jeho rodičovskému objektu. Komponenta *Rect Transform* je zobrazena na obrázku 3.14.



Obrázek 3.14: Rect Transform

3.7.1 Text

UI prvek *Text* je *GameObject* s komponentou *Text*, která je zobrazena na obrázku 3.15. Pomocí atributů této komponenty můžeme specifikovat text, tedy řetězec znaků, které se zobrazí na obrazovce. Dále také font, velikost písma, barvu nebo např. zarovnání.

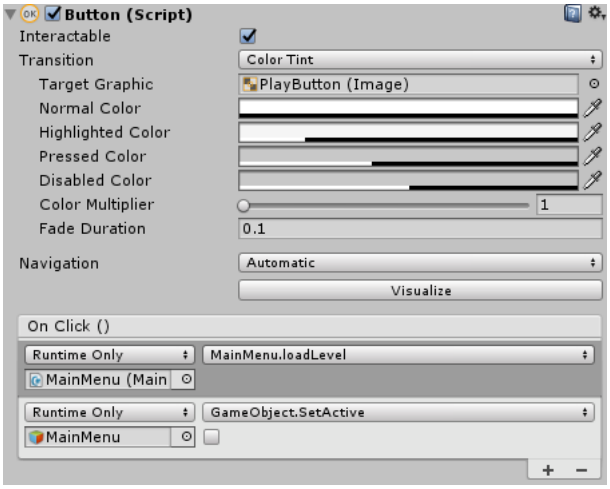


Obrázek 3.15: UI Text

Takto vytvořený *GameObject* může být použit i jako popisek jiného UI prvku. Například tlačítko Nová Hra se skládá z UI prvků *Button* a *Text*, kde *Text* tvoří řetězec znaků „Nová Hra”.

3.7.2 Button

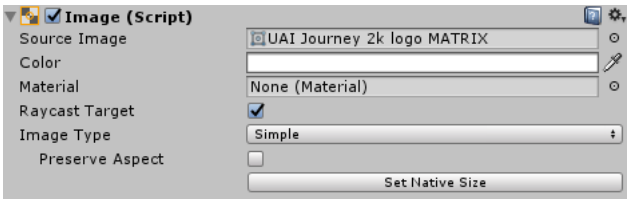
UI prvek *Button* (tlačítko) je *GameObject* s komponentou *Button*, která je zobrazena na obrázku 3.16. Tato komponenta nabízí řadu atributů, které specifikují barevné změny po přejetí kurzoru přes tlačítko nebo při jeho zmáčknutí. Základní funkcionalitou tlačítka je vyvolat akci po stisknutí. Unity poskytuje možnost tyto akce definovat v editoru. V poli `On Click ()` se mohou specifikovat *GameObjecty* a jejich scripty. Z rozbalovací nabídky se poté vyberou jejich `public` funkce, které se zavolají po stisknutí tlačítka.



Obrázek 3.16: UI Button

3.7.3 Image

UI prvek *Image* (obrázek) je jednoduchý *Gameobject* s komponentou *Image*. Atributem *Source Image* nastavíme obrázek (*sprite*), který chceme, aby se v UI zobrazil. Atribut *Color* nastavuje jeho barevný odstín.



Obrázek 3.17: UI Image

3.8 Scriptování v C#

Scriptováním se definuje chování hry. *Scripty* reagují na uživatelské vstupy, definují a kontrolují akce během hraní, monitorují objekty a mnoho dalšího. Unity má vlastní techniky a praktiky, které používá pro programátorský přístup k *GameObjectům* a komponentám vytvořeným v editoru.

Předvytvořené komponenty Unity, jako instance objektů, nabízí řadu funkcí, které se obecně používají při scriptování a nejsou dostupné z editoru. Například komponenta *Transform* nabízí funkce pro pohyb a rotace *GameObjectu*. Seznam těchto funkcí je velmi rozsáhlý a často se mění, proto v této kapitole budou popsány jen základní struktury potřebné pro pochopení programování v Unity. Aktualizovaný seznam komponent a funkcí včetně ukázek kódu je dostupný v dokumentaci Unity.[6]

3.8.1 Přístup ke komponentám a GameObjectům

V editoru se mohou měnit atributy komponent přímo v *inspectoru*. Ovšem ve *scriptu* se musí nejprve zajistit přístup k dané komponentě. Nejčastějším případem je zpřístupnění komponenty připnuté na stejný *GameObject*, na kterém se nachází i *script*. Důležitým faktem je, že komponenta je instance třídy.[16] Ve *scriptu* se tedy vytvoří reference na tuto instanci pomocí funkce *GetComponent*.

```
private void Start()
{
   RigidBody2D rb = GetComponent<RigidBody2d>();
}
```

Výpis 3.1: Přístup ke komponentě

Po získání reference se může přistupovat k atributům komponenty stejně jako v *inspectoru* editoru. Např. `rb.mass = 10f`. Navíc se mohou volat `public` funkce, které komponenta nabízí. Např. `rb.AddForce(Vector2.left)`.

Další možností přístupu ke komponentě je nalinkováním přes proměnou v *inspectoru*. Pokud ve *scriptu* označíme atribut tagem `[SerializeField]` nebo jako `public`, objeví se atribut i v *inspectoru* editoru, stejně jako je popsáno v sekci 3.3. Přetažením komponenty do pole atributu vznikne reference na instanci komponenty. Takto mohou vznikat i reference na komponenty v jiných *GameObjectech* nebo na samotné *GameObjecty*. Ve *scriptu* se s proměnnou pracuje, jako by již referenci obsahovala, ovšem předpokládá se, že nalinkování v editoru proběhne před startem hry.

```
public RigidBody2D rb;
private void Update()
{
   rb.AddForce(Vector2.left * speed)
}
```

Výpis 3.2: Komponenta s referencí přes editor

Kromě předávání referencí v editoru nebo v rámci jednoho *GameObjectu*, je potřeba lokalizovat *GameObjecty* a komponenty a tvořit reference i během hraní hry. K tomuto účelu nabízí Unity dvě funkce. `Transform.Find()` a `GameObject.Find()`. Pokud se během hraní (*runtime*) referencuje *GameObject*, který je vnořený rodičovskému *GameObjectu*, na kterém je umístěn *script*, použije se funkce `Transform.Find(string name)`. Návratový typ této funkce je *transform* a pro získání reference na *GameObject* se využije `Transform.gameObject`.[17]

```
private GameObject hrac;
private GameObject zbran;

private void Start()
{
//referencujeme GameObject ve scene
   hrac = GameObject.Find("HlavniCharakter")
//hledame GameObject se jmenem Zbran, ktery je vnoreny GameObjectu hrac
   zbran = hrac.transform.Find("Zbran").gameObject;
}
```

Výpis 3.3: Reference na vnořený *GameObject*

Funkce `GameObject.Find()` slouží k nalezení reference na *GameObject* kdekoliv v hierarchii, pokud se může identifikovat podle jména.

Referencovat se mohou i kolekce *GameObjectů* podle jejich *tagu*. K tomu slouží funkce `GameObject.FindWithTag()`, která vrací první *GameObject* se specifikovaným *tagem* a `GameObject.FindGameObjectsWithTag()`, která vrátí všechny *GameObjecty* se specifikovaným *tagem* jako kolekci.[16]

```
private GameObject hrac;
private GameObject[] nepratele;

private void Start()
{
   hrac = GameObject.FindWithTag("Hrac");
    enemies = GameObject.FindGameObjectsWithTag("Nepritel");
}
```

Výpis 3.4: Reference pomocí tagu

3.8.2 Tvorba a ničení GameObjectu

Pro vytvoření *GameObjectu* se používá funkce `Instantiate(Object original)`, která vytvoří kopii objektu specifikovaného v parametru funkce. Vytvářený objekt nemusí být přítomný ve scéně. Častějším případem je instanciování *GameObjectu* uloženého jako *prefab*.[18]

Pro zničení *GameObjectu* se používá funkce `Destroy(Object obj, float t = 0.0f)`. Tato funkce přijímá objekt, který bude zničen jako první parametr a volitelně se může nastavit i prodleva v sekundách, po které dojde ke zničení. Touto funkcí lze tedy zničit kromě *GameObjectu* i komponentu. Jak již bylo v minulých kapitolách řečeno, při změně scény dojde k zničení všech *GameObjectů* a vytvoření nových z načítané scény. Často je ovšem žádoucí, aby *GameObject* přetrval přes všechny scény, pokud například sleduje hráčovo celkové skóre. Toho lze dosáhnout použitím funkce `DontDestroyOnLoad(Object target)`.

```
public GameObject kulkaPrefab;

private void vypalKulku()
{
   Instantiate(kulkaPrefab);
}

private void znicObjekt(GameObject obj)
{
   Destroy(obj);
}

private void prezijNacitani(GameObject obj)
{
   DontDestroyOnLoad(obj);
}
```

Výpis 3.5: Tvorba a ničení *GameObjectu*

3.8.3 Komunikace mezi scripty

Script je naprogramovanou komponentou, a proto pro něj platí i stejné možnosti referencování, jak bylo popsáno v 3.8.1. Scripty tedy mohou uchovávat reference na instance jiných *scriptů* a s jejich pomocí spolu komunikovat. Při programovaní hry je často výhodné, když existuje pouze jedna instance objektu, se kterou se komunikuje. Toto platí hlavně pro herní *scripty*, které běží po celou dobu hraní i přes změnu úrovně nebo pozastavení hry. Pro takovéto scripty se uplatňuje tvz. *Singleton*.

Singleton je běžně používaný návrhový vzor, který povoluje vytvořit pouze jednu instanci konkrétní třídy.[20] *Singleton* využívá statické proměnné uchovávající referenci na objekt, ve kterém se deklaruje. Brání vytvoření nové instance nebo objektu, a proto existuje v jednu chvíli pouze jedna instance, se kterou mohou ostatní scripty komunikovat. Níže je ukázka *Singletonu*, která je použita v praktické části této práce pro *GameManager script*.

```
public class GameManager : MonoBehaviour {

   public static GameManager Instance;

void Start ()
{
      if (Instance != null)
      {
         Destroy(gameObject);
      }
      else
      {
         DontDestroyOnLoad(gameObject);
         Instance = this;
      }
}}
```

Výpis 3.6: *Singleton*

3.8.4 Coroutines

Při tvorbě hry je obvyklé vykonávat náročné úlohy na pozadí, aby nezpomalovaly chod hry a nesnižovaly *frame rate*. Příkladem může být funkce, která zaznamenává stav objektů ve scéně. Pokud by taková funkce běžela uvnitř `LateUpdate()`, vždy při jejím vykonávání by se hra zastavila a pokračovala by až po dokončení této funkce. Řešením je vykonávat tuto úlohu na pozadí.

Kromě klasického programátorského řešení využívající vlákna, poskytuje Unity vlastní způsob, jak se vypořádat s tímto problémem. Nabízí funkce označované jako *Coroutines*. *Coroutines* jsou funkce využívající generický interface `IEnumerator`, jako návratový typ. To umožňuje Unity sledovat stav funkce po několik iterací. *Coroutine* využívá operátoru `yield`, který pozastaví funkci na aktuálním řádku instrukce a vrátí výpočetní prostředky zpět programu, který poté pokračuje ve vykonávání ostatních instrukcí.[21]

Operátorem `yield` specifikujeme na jakou dobu nebo za jakých podmínek se má pokračovat ve vykonávání *coroutine*. V praxi se nejčastěji používá pozastavení funkce

na jeden snímek (*frame*), nebo na určitý časový interval. Následující kód zobrazuje funkci `fade()`, která každý snímek sníží alfa kanál obrázku o 10 %, až obrázek zcela zmizí. Tato funkce pozastavuje vykonávání funkce operátorem `yield return null`, který pozastaví funkci na jeden snímek. Postupné vyblednutí obrázku bude tedy trvat 11 snímků.

```
IEnumarator fade(){
   for (float f = 1f; f >= 0; f -= 0.1f) {
        Color c = SpriteRenderer.color;
        c.a = f;
        SpriteRenderer.color = c;
        yield return null;
    }
}
```

Výpis 3.7: *Coroutine* s pozastavením po snímku

Hra má ovšem na rozdílných zařízeních různý *frame rate* a hráč by nemusel, na zařízení s vysokou snímkovou frekvencí, vyblednutí obrázku zaznamenat. Proto jde *coroutine* pozastavit na určitý časový interval operátorem `yield return new WaitForSeconds(float Seconds)`. Následující kód zobrazuje opět funkci `fade()`, ale s využitím tohoto operátoru. Vyblednutí obrázku se takto bude na různých zařízeních chovat stejně a proběhne v intervalu jedné sekundy.

```
IEnumarator fade(){
   for (float f = 1f; f >= 0; f -= 0.1f) {
        Color c = SpriteRenderer.color;
        c.a = f;
        SpriteRenderer.color = c;
        yield return new WaitForSeconds(.1f);
    }
}
```

Výpis 3.8: *Coroutine* s pozastavením po 0.1s

Funkce typu *coroutine* se volají příkazem `StartCoroutine(IEnumerator coroutine)`, takto se *coroutine* spustí a skončí až po svém proběhnutí do konce, nebo po zavolání funkce `StopAllCoroutines()`, která ovšem zastaví všechny právě prováděné *coroutines*. Pokud se ovšem pro zavolání *coroutine* použije příkaz `StartCoroutine(„název coroutine", param1, param2, ...)`, lze přerušit její provádění příkazem `StopCoroutine(„název coroutine")`.[21]

```
// coroutine
IEnumerator myCoroutine(float num) {
   // telo funkce
}
// spustime coroutine funkci myCoroutine
void Start(){
   StartCoroutine("myCoroutine", .04f);
}
// pri prvnim pruchodu funkce LateUpdate() zastavime
// coroutine myCoroutine
void LateUpdate(){
   StopCoroutine("myCoroutine");
}
```

Výpis 3.9: Spuštění a zastavení *Coroutine*

Při programování *coroutines* by se mělo postupovat obezřetně. Na rozdíl od vláken nenabízí totiž paralelismus. *Coroutine* běží ve výchozím stavu na stejném vlákně jako funkce Unity cyklu. Při špatném naprogramování mohou přerušit tento cyklus, a tím i zastavit hru. *Coroutines* jsou na druhou stranu užitečným pomocníkem v situacích, kdy se řídí průběh vykonávaných instrukcí pomocí času, nebo se vytváří efekty, jako je například výše uvedená funkce `fade()`.

3.9 Build

Sestavení aplikace, neboli její *build*, se nastavuje v Unity v okně *Build Settings*. Toto okno, které lze vidět na obrázku 3.18, se zobrazí po stisknutí **Ctrl + Shift + B,** nebo přes volbu v nástrojové liště **File > Build Settings**. V horní části (pod číslem 1) je seznam všech scén v projektu a zaškrtnutím políčka před jejich názvem se scéna zahrne do sestavení aplikace. Na každém řádku seznamu je zobrazeno jméno a index scény. Pomocí těchto indexů, nebo názvů lze ke scénám programátorsky přistupovat.

V dolní části (pod číslem 2) je seznam dostupných platforem a napravo od něj (pod číslem 3) se nachází detail platformy s dodatečnými nastaveními. Po zvolení platformy a stisknutí tlačítka *Switch Platform* se změní cílová platforma a assety se re-importují s formátem vhodným pro cílovou platformu. Zvolená platforma je v seznamu označena ikonou Unity.[19]

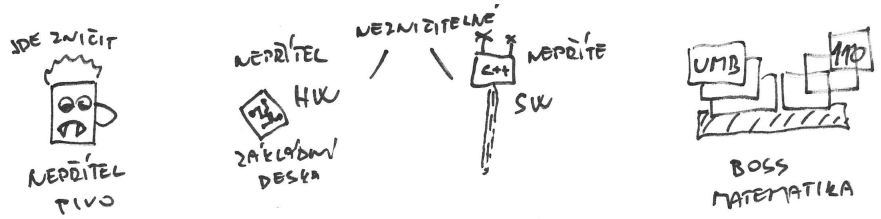
Sestavení aplikace se provede stisknutím tlačítka *Build*, které se nachází ve spodní části okna *Build Settings*.

Obrázek 3.18: Build Settings

4 Praktická část

Úkolem praktické části této práce je vytvořit 2D arcade hru a přitom aplikovat znalosti získané v teoretické části. Na začátku vývoje je třeba určit základní požadavky, které má hra splňovat. V počátečních fázích jsou nejlepšími pomocníky papír s tužkou, které pomohou roztřídit strukturu a ujasnit funkcionalitu výsledné hry.

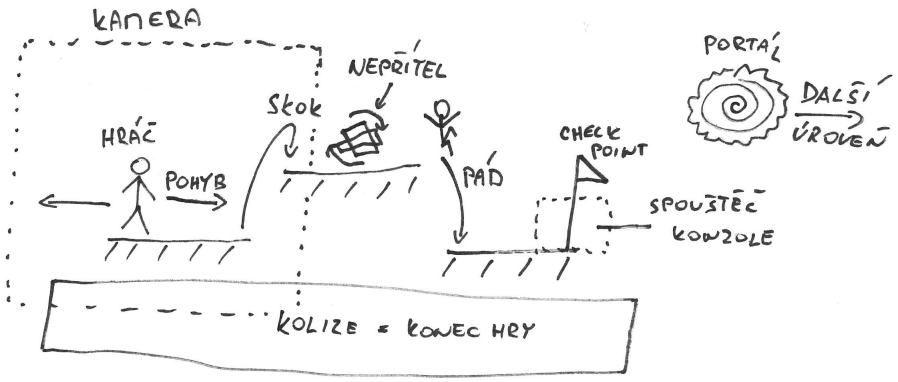
Hra má název „*UAI Journey*" a je o studentovi ústavu aplikované informatiky (UAI) na Přírodovědecké fakultě Jihočeské univerzity, který se ocitá na začátku svého studia. Jeho úkolem je projít všechny úrovně a porazit závěrečného bosse. V cestě mu stojí nepřátelé - létající hardware, software a chodící pivo. Bossem je kniha matematiky, která symbolizuje největší překážku prvního semestru studia na UAI. Úrovně jsou složeny z plošin, po kterých se hráč pohybuje a skáče z jedné na druhou. Hráč má na začátku hry definovaný počet životů a skóre, které se zvyšuje po poražení nepřítele a snižuje po ztrátě všech životů, nebo po pádu z plošiny. Nepřátelé chodí po plošinách a po interakci s hráčem mu uberou život a mohou ho např. i odhodit z plošiny dolů. Hráč může střílet na nepřátele a tím je ničit a zvyšovat tak své skóre. Hrou provází virtuální konzole, která se objevuje ve specifických okamžicích hry a radí hráči, jak pokračovat dál. Postupem vpřed úrovní prochází hráč záchytné body, které definují pozice, na kterých se hráč objeví po ztrátě všech životů, nebo po pádu z plošiny.



Obrázek 4.1: Náčrtek nepřátel

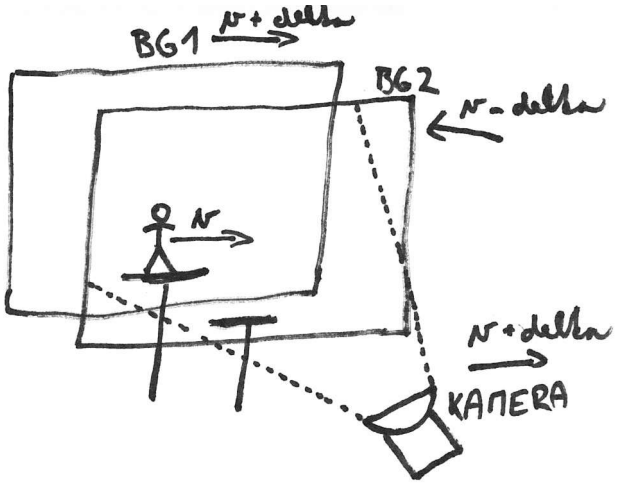
Hra je vyvíjena pro mobilní prostředí a proto má definované dotykové ovládání přes UI prvky. Hra obsahuje scény pro jednotlivé úrovně hry, hlavní menu a závěrečnou obrazovku. Hráč umí běhat vpravo i vlevo, skákat, padat, střílet a umírat. Pro tyto akce jsou vytvořené animace. Stejně tak jsou vytvořené animace i pro

chování nepřátel a závěrečného bosse. Na konci každé úrovně je portál. V případě kolize hráče s portálem se zobrazí informace o skóre, počtu smrtí a času stráveného hraním dané úrovně. Poté se načte nová úroveň a tyto statistiky jsou vynulovány a hráči je vracen plný počet životů. Skóre mu ovšem přetrvává přes všechny úrovně až do konce hry. Po poražení závěrečného bosse se zobrazí informace o dohrání hry. Hra se může pozastavovat. Po každém načtení úrovně, pozastavení nebo vypnutí hry se hra uloží. Uložení probíhá formou serializace statistik hráče, aktuální polohy a úrovně do datového souboru. Po spuštění hry je možné pokračovat ve hraní v bodě, kde hráč skončil, pokud byla hra předtím uložena. Kolize s nepřáteli je signalizována vibrováním zařízení. Celou hrou doprovází hráče polyfonní melodie, které se mění s každou úrovní. Po pozastavení hry je možné zobrazit nastavení, kde si uživatel může přizpůsobit hlasitost hudby a popřípadě vypnout vibrace.



Obrázek 4.2: Rozvržení úrovně

Kamera sleduje pohyb hráče úrovní. Pozadí úrovní je složeno z několika vrstev, které se pohybují rozdílnými rychlostmi oproti pohybu kamery, a tím vytvářejí *parallax* efekt (náčrtek na obrázku 4.3), který dodává na dynamice hry. Pozadí je uspořádáno do fronty vedle sebe a tvoří dlaždice. Při pohybu kamery se dlaždice, které zmizí ze záběru, zařadí na začátek fronty a jsou připraveny znovu na zobrazení. Pozadí se tak tvz. roluje vpřed a vzad. Tím se hra vyhne nutnosti vytvářet velká pozadí přes celé scény a ušetří tím místo i nároky na grafické prostředky.



Obrázek 4.3: Parallax efekt

Diagram hlavních *GameObjectů* a jejich naprogramovaných komponent je zobrazen v Příloze A. *GameObject Player* (hráč) je naprogramován jako stavový automat. Stavy jsou např. *idle, left, right, jump, landing, falling, takingDMG, immortal, kill, resurrect a firingWeapon*. Jsou definovány možné přechody mezi těmito stavy a delegát, kterým ostatní objekty mohou změnit stav hráče. Centrálním *GameObjectem* je *GameManager*, který přetrvává přes všechny úrovně a zaznamenává dotyk hráče, sleduje celkové skóre, řídí změnu úrovní či ukládá a načítá hru. Dále také vyvolává menu při pozastavení hry, nebo po přechodu do nastavení. *GameObject Enemy* se pohybuje po úrovni po předem specifikované trase a má definované akce pro kolize s hráčem, nebo kulku vypálenou hráčem. Závěrečný boss (*MathBoss*) je *GameObject*, u kterého je naprogramovaná základní umělá inteligence a náhodné chování. Boss tak skáče z místa na místo a rozevírá své stránky (*GameObject MathBossPage*), které ubírají životy. V úrovních jsou rozmístěny spouštěče (*DialogueTrigger*) akcí, které v případě kolizí s hráčem pozastaví hru a vyvolají virtuální konzoli (*GameObject Dialog*), ve které se objeví zpráva pro hráče. Hra je naprogramována modulárně a hlavní *GameObjecty* jsou připraveny ve formě *Prefab*. Tím je usnadněno další rozšíření hry a tvorba nových úrovní.

Cílovou platformou pro hru je mobilní operační systém Android v minimální verzi 4.1 *Jelly Bean*. Velikost UI prvků a grafických objektů ve hře je nastavena tak, aby odpovídala měřítku displeje výsledného zařízení, přičemž ve výchozím stavu toto měřítko odpovídá rozlišení 1920x1080 (*Full HD*). Hra je testována na 5 mobilních

zařízeních a 1 tabletu. Zařízení mají rozdílné úhlopříčky, verze Android i grafické a výpočetní prostředky.

Po otestování, byla hra elektronicky podepsána a nahrána na obchod aplikací *Google Play* v uzavřené beta verzi a zpřístupněna dvěma testerům. V současné době (březen, 2018) je hra zdarma v produkční verzi zpřístupněna pro region Česká republika a na stránce obchodu *Google Play*[24] je možné hru stáhnout nebo se podívat na recenze a hodnocení uživatelů.

5 Závěr

Práce si kladla za cíl popsat vývoj 2D hry typu arcade na bázi herního enginu a v praktické části toto implementovat. V práci je popsán vývoj hry v herním enginu Unity3D se zaměřením na důležité prvky, které jsou pro vývoj ve 2D nezbytné. Čtenář je seznámen s herním enginem Unity3D, jeho stavebními elementy a způsobem, jakým herní engine přistupuje k vytváření hry. Dále je popsán způsob tvoření animací, práce s grafikou a uživatelským rozhraním. Přiblíženo je také scriptování, včetně způsobů komunikace objektů v rámci herního enginu a obecně používaný návrhový vzor.

V praktické části je toto implementováno. Vznikla 2D mobilní hra typu arcade „*UAI Journey*" pro platformu Android, která je dostupná zdarma z oficiálního obchodu *Google Play*.[24] Zdrojové kódy je možno zobrazit na platformě *GitHub*.[25]

6 Odkazy a literatura

[1] What is Unreal Engine. *Unreal Engine* [online]. North Carolina: Epic Games, c2004-2018 [cit. 2018-02-21]. Dostupné z: `https://www.unrealengine.com/en-US/what-is-unreal-engine-4`

[2] GameMaker Studio 2. *YoYoGames* [online]. North Carolina: YoYo Games, c2013-2018 [cit. 2018-02-21]. Dostupné z: `https://www.yoyogames.com/gamemaker/features`

[3] Unity 2017: The world-leading creation engine for gaming. *Unity3d* [online]. San Francisco: Unity Technologies, c2018 [cit. 2018-02-21]. Dostupné z: `https://unity3d.com/unity`

[4] Unity - Fast Facts. *Unity3d* [online]. San Francisco: Unity Technologies, c2018 [cit. 2018-02-21]. Dostupné z: `https://unity3d.com/public-relations`

[5] . *Unity3d* [online]. San Francisco: Unity Technologies, c2018 [cit. 2018-02-21]. Dostupné z: `https://unity3d.com/public-relations`

[6] *ScriptReference* [online]. San Francisco: Unity Technologies, c2018 [cit. 2018-03-30]. Dostupné z: `https://docs.unity3d.com/ScriptReference/`

[7] *GameObject* [online]. San Francisco: Unity Technologies, c2018 [cit. 2018-02-15]. Dostupné z: `https://docs.unity3d.com/ScriptReference/GameObject.html`

[8] *RigidBody2D* [online]. San Francisco: Unity Technologies, c2018 [cit. 2018-02-16]. Dostupné z:`https://docs.unity3d.com/Manual/class-Rigidbody2D.html`

[9] *Collider2D* [online]. San Francisco: Unity Technologies, c2018 [cit. 2018-02-16]. Dostupné z:`https://docs.unity3d.com/Manual/Collider2D.html`

[10] *OnTriggerEnter2D* [online]. San Francisco: Unity Technologies, c2018 [cit. 2018-02-15]. Dostupné z:`https://docs.unity3d.com/ScriptReference/Collider2D.OnTriggerEnter2D.html`

[11] *Sprites* [online]. San Francisco: Unity Technologies, c2018 [cit. 2018-02-01]. Dostupné z:`https://docs.unity3d.com/Manual/Sprites.html`

[12] *Animation Clips* [online]. San Francisco: Unity Technologies, c2018 [cit. 2018-01-10]. Dostupné z:`https://docs.unity3d.com/Manual/AnimationClips.html`

[13] *Animator* [online]. San Francisco: Unity Technologies, c2018 [cit. 2018-02-15]. Dostupné z:`https://docs.unity3d.com/ScriptReference/Animator.html`

[14] *Cameras Overview* [online]. San Francisco: Unity Technologies, c2018 [cit. 2018-02-25]. Dostupné z:`https://docs.unity3d.com/Manual/CamerasOverview.html`

[15] *Camera class* [online]. San Francisco: Unity Technologies, c2018 [cit. 2018-03-20]. Dostupné z:`https://docs.unity3d.com/Manual/class-Camera.html`

[16] *Controlling GameObjects Components* [online]. San Francisco: Unity Technologies, c2018 [cit. 2018-03-21]. Dostupné z:`https://docs.unity3d.com/Manual/ControllingGameObjectsComponents.html`

[17] *Transform* [online]. San Francisco: Unity Technologies, c2018 [cit. 2018-01-11]. Dostupné z:`https://docs.unity3d.com/ScriptReference/Transform.html`

[18] *Creation and Destruction of GameObjects* [online]. San Francisco: Unity Technologies, c2018 [cit. 2018-03-22]. Dostupné z:`https://docs.unity3d.com/Manual/CreateDestroyObjects.html`

[19] *Build Settings* [online]. San Francisco: Unity Technologies, c2018 [cit. 2018-03-27]. Dostupné z:`https://docs.unity3d.com/Manual/BuildSettings.html`

[20] MURRAY, Jeff W. *C# Game Programming Cookbook for Unity 3D*. Taylor & Francis, 2014. ISBN 978-1-4665-8142-5

[21] GOBOLD, Ashley a JACSON, Simon. *Mastering Unity 2D Game Development*. Second Edition. Packt Publishing, 2016. ISBN 978-1-78646-345-6

[22] SAPIO,Francesco a SAHER, Abdelrahman. Unity 5.x 2D Game Development Blueprints. Pact Publishing, 2016. ISBN 978-1-78439-310-6

[23] CALABRESE, Dave. *Unity 2D Game Development*. Pact Publishing, 2014. ISBN 978-1-84969-256-4

[24] *UAI Journey* on Google Play [online]. Google, c2018 [cit. 2018-03-30]. Dostupné z:`https://play.google.com/store/apps/details?id=com.JiriHauser.test`

[25] *UAI Journey on GitHub* [online]. GitHub, Inc., c2018 [cit. 2018-02-15]. Dostupné z:`https://github.com/Kamony/UAI-Journey`

Seznam obrázků

2.1 Logo Unity3D [II] 4

3.1 Editor 5
3.2 GameObject a komponenta Transform 7
3.3 Životní cyklus Unity [I] 8
3.4 Script jako komponenta 9
3.5 Rigidbody2D 10
3.6 Box Collider 2D 11
3.7 Sprite Renderer a Sprite 11
3.8 Sprite Sheet 12
3.9 Dope Sheet 13
3.10 Animation Controller 13
3.11 Animator 15
3.12 Camera 15
3.13 Perspective (vlevo) and Orthographic (vpravo) camera [III] 16
3.14 Rect Transform 17
3.15 UI Text 17
3.16 UI Button 18
3.17 UI Image 18
3.18 Build Settings 25

4.1 Náčrtek nepřátel 27
4.2 Rozvržení úrovně 28
4.3 Parallax efekt 29

[I] Životní cyklus Unity. Autor: Unity Technologies. *Unity3d* [online]. [cit. 2018-03-08]. Dostupné z: `https://docs.unity3d.com/uploads/Main/monobehaviour_flowchart.svg`

[II] Unity 3D. Autor: Unity Technologies. *Unity3d* [online]. [cit. 2018-03-25]. Dostupné z: `https://unity3d.com/files/images/ogimg.jpg`

[III] Camera - Perspective and orthographic, Autor: Unity Technologies. *Unity3d* [online]. [cit. 2018-03-16]. Dostupné z: `https://docs.unity3d.com/uploads/Main/CameraPerspectiveAndOrtho.jpg`

Seznam příloh

- Příloha A – Diagram hry „UAI Journey”
- Příloha B – Obsah přiloženého CD

Příloha A – Diagram hry „UAI Journey”

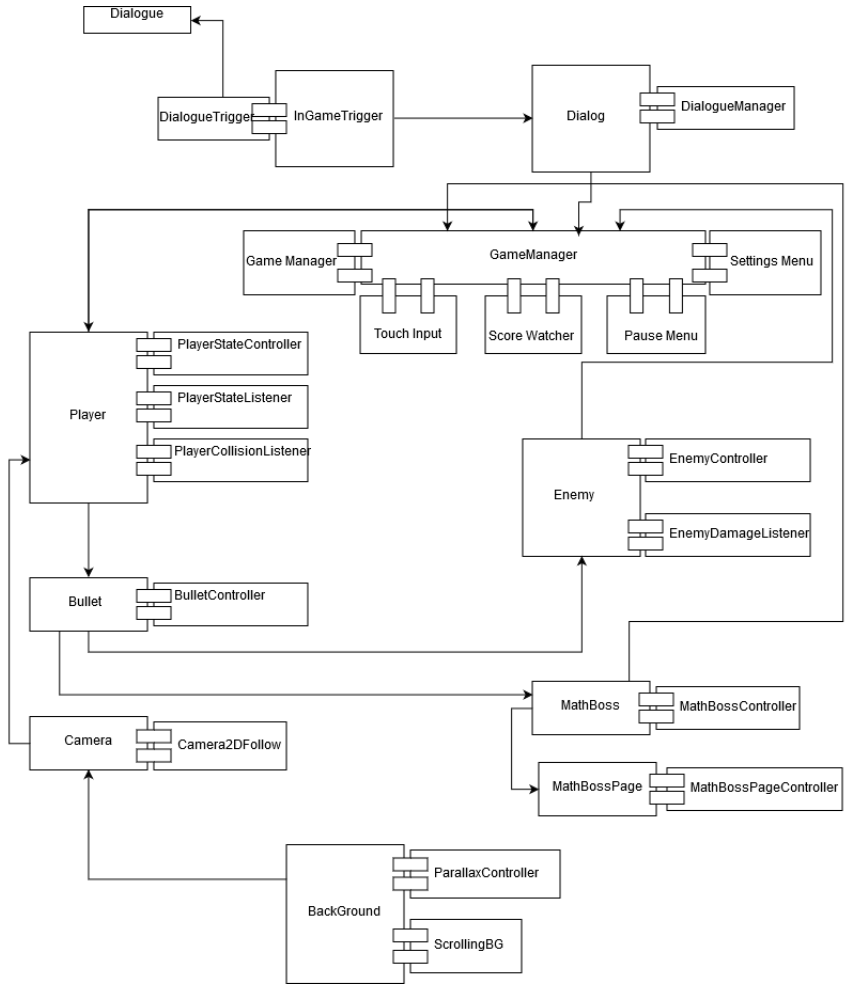


Diagram zobrazuje strukturu hlavních objektů ve hře a jejich vztahy. V diagramu jsou vyobrazeny *GameObjecty* a k nim připojené scripty jako komponenty. Například *GameObject* „*Dialog*” a jeho komponenty *DialogueManager*. Šipky znázorňují komunikaci mezi *GameObjecty*, která probíhá nejčastěji pomocí delegátů, nebo *singletonů*.

Příloha B - Obsah přiloženého CD

UAI_Journey.unitypackage - balík assetů spustitelný v editoru unity

UAIJourney_release.apk - spustitelný soubor na platformě Android. Aktuální verze hry nahraná na Google Play (1.4.2018)

UAIDiagram.png - Diagram hry „UAI Journey"

JiriHauser_bakPrace.pdf - Bakalářská práce ve formátu pdf