ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

PROVOZNĚ EKONOMICKÁ FAKULTA

KATEDRA INFORMAČNÍHO INŽENÝRSTVÍ

Bakalářská práce

Vývoj 3D aplikace s využitím Unity

Jan Roček

zadání

Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Autor práce:	Jan Roček
Studijní program:	Systémové inženýrství a informatika
Obor:	Informatika
Vedoucí práce:	Ing. Jiří Brožek, Ph.D.
Garantující pracoviště:	Katedra informačního inženýrství
Název práce:	**Vývoj 3D aplikace s využitím Unity**
Název anglicky:	**Using Unity for 3D application development**
Cíle práce:	Cílem práce je popsat možnosti tvorby 3D aplikací s využitím engine Unity a demonstrovat zjištěné poznatky formou ukázkové 3D aplikace.
Metodika:	Metodika práce je založena na studiu a analýze odborných informačních zdrojů. Na základě syntézy zjištěných poznatků budou popsány možnosti využití grafického engine Unity pro tvorbu 3D aplikací. Dále bude navržena a implementována ukázková 3D aplikace využívající engine Unity a demonstrující jeho možnosti. Budou zhodnoceny výhody využití Unity nejen s ohledem na usnadnění vývoje aplikace ale i možnosti jejího nasazení na různých platformách.
Doporučený rozsah práce:	35-40 stran
Klíčová slova:	Unity, 3D, polygon, aplikace, C#, Visual Studio, engine

Doporučené zdroje informací:

1. MAREŠ, A. 1001 tipů a triků pro C# 2010. Brno: Computer Press, 2011. ISBN: 978-80-251-3250-0
2. MILLER, T. Programujeme 3D hry v jazyce C#. Brno: Computer Press, 2007. ISBN: 80-251-1126-1
3. ROUDENSKÝ, P. -- KHORSHID MOKTHAR M. Programujeme hry v jazyce C#. Brno: Computer Press, 2011. ISBN: 978-80-251-3355-2
4. SHARP, J. Microsoft Visual C 2010 krok za krokem. Brno: Computer Press, 2010. ISBN: 978-80-251-3147-3

Předběžný termín obhajoby: 2015/16 LS - PEF

Elektronicky schváleno: 20. 2. 2016
Ing. Martin Pelikán, Ph.D.
Vedoucí katedry

Elektronicky schváleno: 20. 2. 2016
Ing. Martin Pelikán, Ph.D.
Děkan

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Vývoj 3D aplikace s využitím Unity" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 8. 3. 2016 _____

Poděkování

Mé poděkování patří panu Ing. Jiřímu Brožkovi, Ph.D. za vedení mé bakalářské práce, vstřícnost při konzultacích a mnoho cenných rad, které mi poskytl.

Vývoj 3D aplikace s využitím Unity

Using Unity for 3D application development

Shrnutí

Tato bakalářská práce se zabývá vývojem 3D aplikace pomocí multiplatformního enginu Unity 3D v jazyce C# pro operační systém Windows. Teoretická část popisuje historický náhled do vývoje 3D aplikací a zvažuje možnosti moderních vývojových prostředků. Jsou zde probrány klady a zápory dostupných 3D enginů. Praktická část se zabývá samotným vývojem 3D aplikace, kde je podrobně popsán průběh vývoje se závěrečným zhodnocením. Výsledkem je funkční 3D hra pro dva hráče z izometrického pohledu.

Klíčová slova

Unity, 3D, polygon, aplikace, C#, Visual Studio, engine.

Summary

This bachelor thesis deals with the development of 3D applications using the Unity 3D multiplatform engine in C# scripting language for the Windows operating system. The theoretical part describes the historical insight into the development of 3D applications and considers the possibilities of modern development tools. There are discussed the pros and cons of available 3D engines. The practical part deals with the development of 3D applications which described in detail the course of development with a final evaluation. The result is a functional 3D game for two players from the isometric view.

Key Words

Unity, 3D, polygon, application, C#, Visual Studio, engine.

Obsah

1. Úvod

Za posledních 50 let vývoj prostorových aplikací zaznamenal řadu důležitých změn, o které se zasadil v největším měřítku zejména herní průmysl. Na samém počátku sloužilo k dojmu prostorového světa postupné vykreslování jednoduchých 2D obrázků. Postupem času se vyvíjely jak grafické prvky, tak způsob programování her. Vznikaly různé způsoby vykreslování 3D světa, ale také knihovny předpřipravených funkcí pro 3D hry, ze kterých se postupně definoval pojem „Engine“.

Vývoj technologie a razantní růst výkonu počítačů umožnil, že se dnes grafické prvky pohybují na pomezí reality, vznikají komplexní vývojářské nástroje a vývoj je stále dostupnější a snadnější.

2. Cíl práce a metodika

2.1 Cíl práce

Cílem práce je popsat tvorbu 3D aplikací, její historii a moderní vývojářské prostředky se zaměřením na multiplatformní engine Unity 3D. Tyto informace budou demonstrovány v praktické části, kde bude vytvořena 3D aplikace pomocí vývojářského prostředí Unity 3D a jazyka C#.

2.2 Metodika

Teoretická část představuje náhled do historie vývoje 3D aplikací a příklady moderních vývojových prostředků. Vzhledem k zastarávání tištěných zdrojů je metodika práce založena na studiu a analýze především internetových zdrojů. Na základě syntézy zjištěných poznatků jsou popsány možnosti využití grafického engine Unity pro tvorbu 3D aplikací. V praktické části je implementována ukázková 3D aplikace prostřednictvím Unity, kde jako hlavní zdroj slouží programátorská dokumentace na oficiálních stránkách. Jsou zde zváženy výhody použití Unity a možnosti nasazení na jiných platformách.

3. Teoretická část

3.1 Pojmy

V oblasti vývoje 3D aplikací se nachází několik důležitých termínů a zkratek, které jsou převzaty především z anglického jazyka a bez tohoto kontextu mohou mít samostatně jiný nebo pozměněný význam. V následující části proto budou vysvětleny základní pojmy související především s vývojem 3D aplikací.

3.1.1 Polygon

Polygon ve 3D grafice je základní stavební součástí každého 3D objektu. Ve hrách se polygonový model využívá zejména kvůli nízkým nárokům na vykreslení objektů.[1]

3.1.2 Engine

Engine je v počítačové terminologii považován za jádro programu. V 3D grafice se jedná o komplexní software, který zajišťuje vykreslování grafických objektů, obsahuje nástroje pro simulaci fyzikálních zákonů a poskytuje vývojáři prostředí pro naprogramování aplikace.[2]

3.1.3 IDE

Zkratka IDE pochází z anglického sousloví „Integrated Development Environment“. Jedná se o software určený k programování a vývoji aplikací.[3]

3.1.4 Asset

V prostředí Unity 3D se Assetem rozumí jakýkoliv objekt, který je použitý v aplikaci. Může se jednat o 3D model, animaci, zvukovou stopu, ale i skript.[4]

3.1.5 Bitmapa

Bitmapa nebo bitmapová grafika je způsob vykreslování a uložení grafických informací, kde se obrázky skládají z jednotlivých pixelů zarovnaných do mřížky. Počet pixelů každého grafického objektu je pevně dán a tudíž se objekt, po určité úrovni zvětšení, rozostří.[5]

3.1.6 Vektorová grafika

Oproti Bitmapové grafice jsou zde veškeré grafické informace uloženy v křivkách. Každá křivka má svůj tvar, barvu a délku. Vektorová grafika umožňuje teoreticky nekonečnou možnost zvětšování a zmenšování grafických objektů, protože se tím pouze mění parametry křivek.[5]

3.2 Historie 3D aplikací

Vývoj 3D aplikací se postupně vyvinul dvěma směry. První směr spočívá ve vývoji aplikace od samého počátku, z kterého se postupem času vyvíjel směr druhý tj. použití herního enginu a předpřipravených knihoven, kde 3D aplikace vznikají prakticky skládáním hotových komponent do jednoho funkčního celku.

3.2.1 70. léta

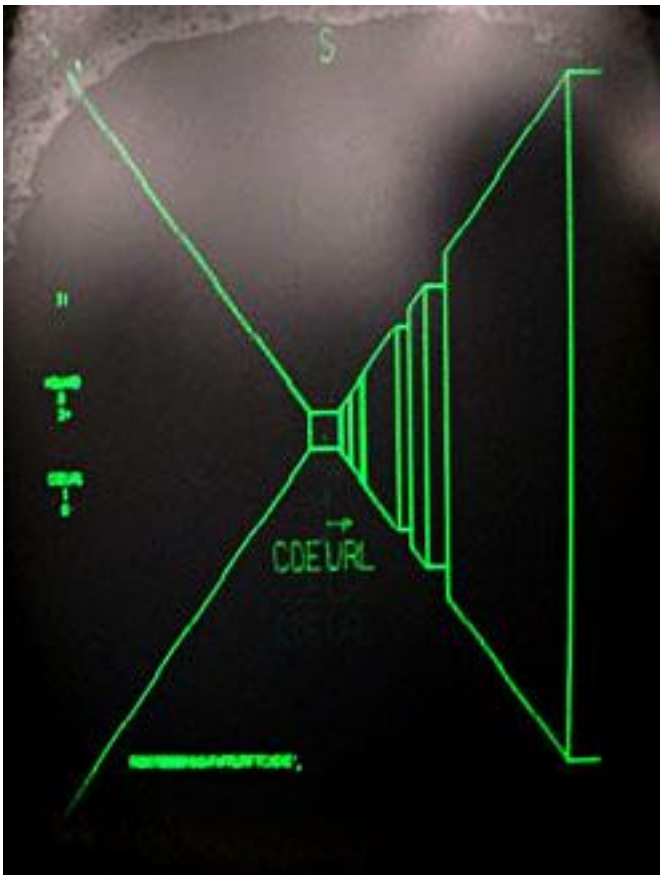
V 70. letech vzniklo mnoho 3D her a každá pojmula pojem prostorového vnímání odlišně. V tomto období najdeme jak první pokusy o drátový grafický model, tak pokročilejší drátový model založený na vektorech pro vektorové monitory. Zároveň, ale vznikaly hry s pseudo 3D efektem, které prostorový dojem utvářely pouze postupným zvětšováním jednoduchých 2D obrázků. Vývoj v této době byl velmi obtížný, protože neexistovali žádné knihovny a počítače v té době neměly dostatečný výkon a funkce pro vytváření aplikace v třetím rozměru. Vzhledem k ohromnému množství 3D aplikací zde vyjmenuji jen ty nejzásadnější.

Prvenství ve 3D připadá hře Maze War, která neoficiálně vyšla pro úzký okruh lidí v roce 1973 pro platformu Imlac PDS-1. Hru napsal Steve Colley v NASA Ames Research Center (ARC) v Kalifornii.[6]

Steve Colley se původně snažil o zobrazení jednoduchých 3D modelů na Imlac PDS-1. Největší složitost byla v práci s operacemi sinus a cosinus na pomalém stroji, kde neexistovali funkce násobení a dělení. První z experimentů byla jednoduchá rotující krychle, která se skládala pouze z linek. Následoval nápad s vytvořením bludiště, které bylo založené na stejném principu.[6]

Hra se skládala z pole 16x16, kde se hráč mohl pohybovat po krocích vpřed, vzad nebo se otočit o 90° vlevo či vpravo. Na počátku šlo pouze o procházení bludiště k cíli z pohledu první osoby. Jeden z kolegů Steva Colleyho přišel s nápadem přidání více lidí do bludiště. *(„Maze was popular at first but quickly became boring. Then someone had the idea to put people in the maze.“)* [6]

V roce 1974 následoval úspěšný pokus s propojením více počítačů Imlac pomocí sériových portů, které přenášelo souřadnice hráčů. Nakonec bylo přidáno střílení, a tak vznikla první střílečka z vlastního pohledu. [6]

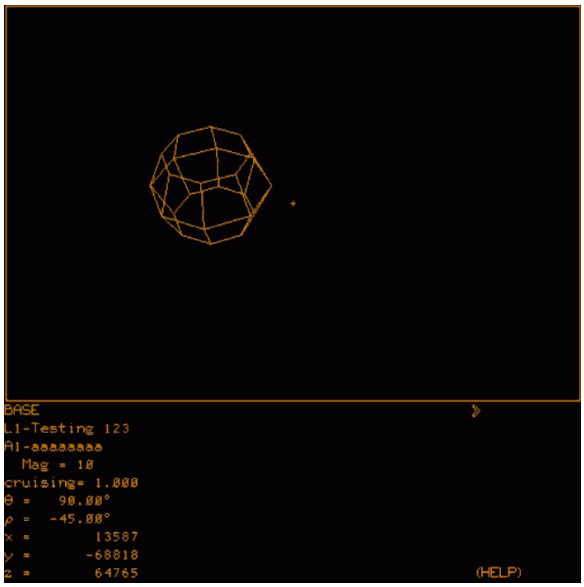


Obrázek 1- Maze War na Imlac PDS-1

Zdroj: http://www.old-computers.com/museum/software/mazewar_ss2.jpg

Téměř souběžně s hrou Maze War vznikl titul Spasim. Spasim je zkratka z anglického sousloví „Space simulator“. Naprogramoval ho Jim Bowery pro PLATO Network, což byl systém pro počítačovou výuku, který obsahoval stovky grafických terminálů s rozlišením 512x512 pixelů.[7]

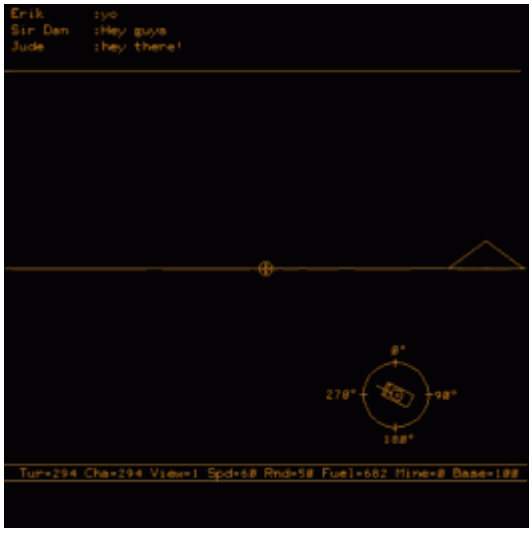
Spasim byla první oficiální multiplayerová 3D hra pro až 32 hráčů. Jednalo se o vesmírný simulátor, který obsahoval 4 planetární soustavy, na kterých se mohlo v jeden okamžik pohybovat až 8 hráčů. Hra umožňovala volný pohyb prostorem za využití grafického drátového modelu, kde pozice hráčů byla obnovována zhruba každou 1 sekundu.[7]



Obrázek 2 - Spasim pro PLATO Network

Zdroj: https://upload.wikimedia.org/wikipedia/ru/5/54/Spasim_gameplay.png

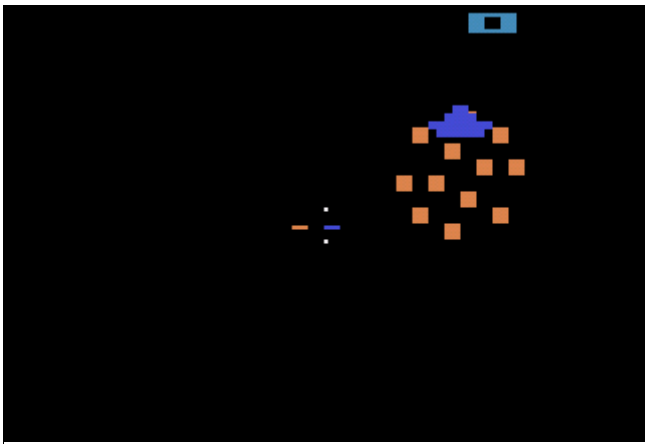
Pro PLATO Network bylo v rámci programu počítačové výuky vyvinuto mnoho her, ale jen málo z nich přinášelo něco nového. Nejinovativnější byl v té době tankový simulátor Panther, který naprogramovali John Edo Haefeli a Nelson Bridwell v roce 1975. Revoluční změna, kterou hra přinesla, bylo použití škálovatelných vektorových křivek. [8]



Obrázek 3 - Panther pro PLATO Network

Zdroj: http://s6.photobucket.com/user/greyshark/media/panther-1975.png.html

V roce 1977 vyšla hra Stars Ship. Star ship vyvinula firma Atari pro platformu Atari 2600. Tato hra je typickým příkladem využití pseudo 3D efektů. Jedná se o jednoduchou hru, kde je úkolem nasbírat co nejvíce bodů za sestřelené nepřátelské lodě a zároveň se vyhýbat střelám. Prostorový dojem zde vytváří jednoduché 2D obrázky, které se pohybují v určitém směru a zároveň se zvětšují v závislosti s přibližováním hráče.[9]



Obrázek 4 - Star ship pro Atari 2600

Zdroj: http://www.atarimania.com/2600/screens/star_ship_2.gif

Výše uvedené tituly udaly základní koncept, který se v následujících letech vylepšoval a přicházel v mnoha obměnách od různých autorů.

3.2.2 80. léta

80. léta kromě vylepšování 3D prvků z předchozích let přinesla jen jediný přelomový technologický prvek a tím je použití polygonů.

V roce 1983 Atari vyvinulo hru I Robot, vůbec první hru, kde byl použit polygonový model. Design navrhl Dave Theurer a hra byla určená pro herní automat Atari Unique. O výpočetní výkon se staral 8 bitový procesor Motorola 6809 s frekvencí 1,5 MHz a 4 procesory AMD 2901, které se staraly především o matematické výpočty. K zobrazení se využíval rastrový CRT monitor s rozlišením 256x232 pixelů s možností zobrazení až 96 barev.[10]

Obrázek 5 - I,Robot pro Atari Unique

Zdroj: https://i.ytimg.com/vi/RRN1PYXc-hg/maxresdefault.jpg

3.2.3 90. léta

V 90. letech vzniklo mnoho enginů pro tvorbu 3D prostředí. První vyšel v roce 1990. Jedná se o engine id Tech 1 od společnosti id Software. Engine id Tech 1 známý též jako Doom engine vytvořil John Carmac ve spolupráci s Johnem Romerem, Davem Taylorem a Paulem Radkem. Id Tech 1 byl navržený především pro hru Doom a Doom 2, ale byl použit pro vznik řady dalších her, jedná se například o hry Heretic či Hexen. Nejedná se o plnohodnotný 3D engine, protože veškeré objekty ve hře jsou reprezentovány jako 2D textury, ale ve výsledné hře se pohybujeme plynule ve 3D prostředí. Protože jsou veškeré objekty dvourozměrné, tak nebylo možné ve hře vytvářet patra, ale to naopak umožnilo plynulejší chod na méně výkonných počítačích.[11][12]

Obrázek 6 – Doom

Zdroj: http://image.dosgamesarchive.com/screenshots/doom2.gif

V roce 1992 vydalo studio NovaLogic proprietární engine Voxel. Název je odvozen ze slov volumetric a pixel. Jedná se o engine, který renderuje terén na základě vykreslování pixelů s různou výškou. Za pomocí tohoto enginu byly vytvořeny hry jako Comanche, Blade Runner, Command & Conquer: Red Alert 2 a Master of Orion 3.[11]

Obrázek 7 - Comanche

Zdroj: http://www.mobygames.com/images/shots/l/242955-comanche-maximum-overkill-dos-screenshot-voxel-space-rivers.png

Jako poslední engine z tohoto období zmíním Quake engine (id Tech 2) od společnosti id Software z roku 1996, který byl prvním enginem renderujícím skutečné polygonální 3D objekty. Jako první podporoval 3D hardwarovou akceleraci a OpenGL knihovny.[13]

Obrázek 8 - Quake

Zdroj: http://cdn4.digitaltrends.com/wp-content/uploads/2011/06/quake.jpg

3.3 Moderní vývojové prostředky

3.3.1 Unity 3D

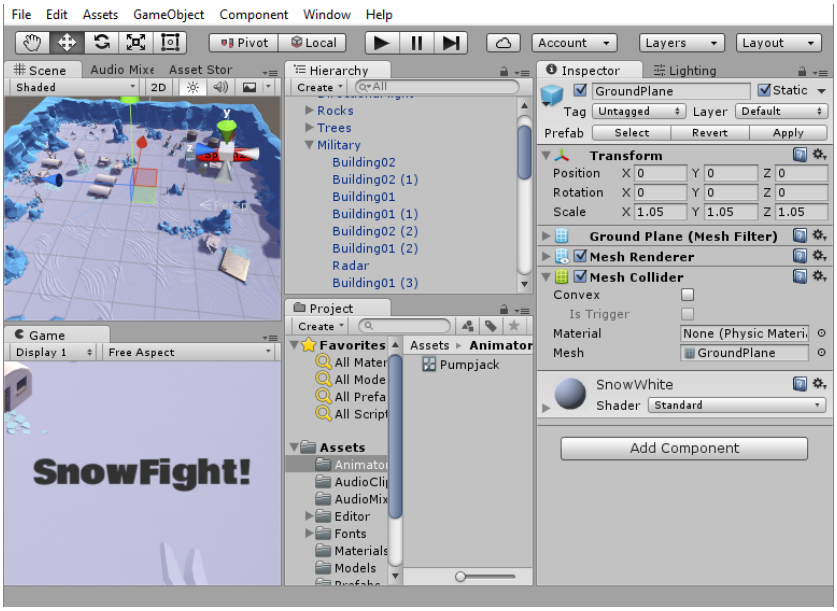
Unity 3D je multiplatformní engine vyvíjený společností Unity Technologies, primárně určený pro tvorbu interaktivního mediálního obsahu, zejména se jedná o vývoj 3D her. Unity vyšel v roce 2005 jako dílo spolupráce Davida Helgasona a Nicholase Francise. Jedná se o kompletní sadu nástrojů a knihoven včetně vlastního vývojového prostředí. Aktuálně poskytuje podporu pro vývoj až na 23 různých platformách. Nabízí možnost vývoje pro operační systémy Windows, MacOS a Linux, mobilní operační systémy, systémy virtuální reality, chytré televize, herní konzole a v poslední řadě i webové aplikace.[14]

Obrázek 9 - logo Unity

Zdroj: http://forum.unity3d.com/attachments/logo-titled-png.16698/

Unity využívá jako hlavní programovací jazyky C# a JavaScript. Obsahuje technologie pro výpočet fyzikálních vlastností objektů v reálném čase Nvidia PhysX3 a Box2D. Pro grafické operace podporuje knihovny DirectX a OpenGL. V Unity je možné zároveň vytvářet jednoduché 3D objekty a animované částicové efekty. Dále nabízí širokou podporu formátů pro importování vlastních objektů či stažení již vytvořených objektů z oficiálního eshopu Asset Store i jiných zdrojů.[15]

Unity 3D je k dispozici ve verzi Unity 3D Personal Edition zcela zdarma, ovšem tato verze je ořezaná o mnoho funkcí. Dále nabízí Professional Edition za měsíční poplatek 75$ nebo zakoupení licence za 1500$. Unity povoluje prodej aplikací i pro licenci Personal Edition, ale při překročení ročního výdělku 100 000$ je pro další legální užívání nutno zakoupit Profesional licenci.[16]



Obrázek 10 - Unity 3D IDE

Zdroj: Vlastní

Vývoj v Unity 3D má oproti konkurenci zásadní výhodu v podobě podpory komunity. Přímo na stránkách Unity je mnoho návodů a rad od zkušených programátorů z řad uživatelů, ale i od vývojářského týmu Unity. Dokumentace Unity 3D je obsáhlá a snadno se v ní orientuje. AssetStore umožňuje sdílení objektů, ale i možnost výdělku prodejem vlastních assetů. Vývojové prostředí je přehledné a prakticky většina operací se dá jednoduše provést přetažením komponent na daný objekt. Zároveň je možné vše hned otestovat spuštěním hry přímo v editoru. Další podstatnou výhodou je množství platforem, na které je možné prostřednictvím Unity vyvíjet hry.

Mezi nevýhody řadím absenci mnoha komponent, které jsou dostupné pouze v placené verzi, především nástroje analyzující výpočetní náročnost projektu a optimalizační funkce. V Unity také není možné dosáhnout takové grafické úrovně s jakou pracují profesionální nástroje velkých vývojářských společností.

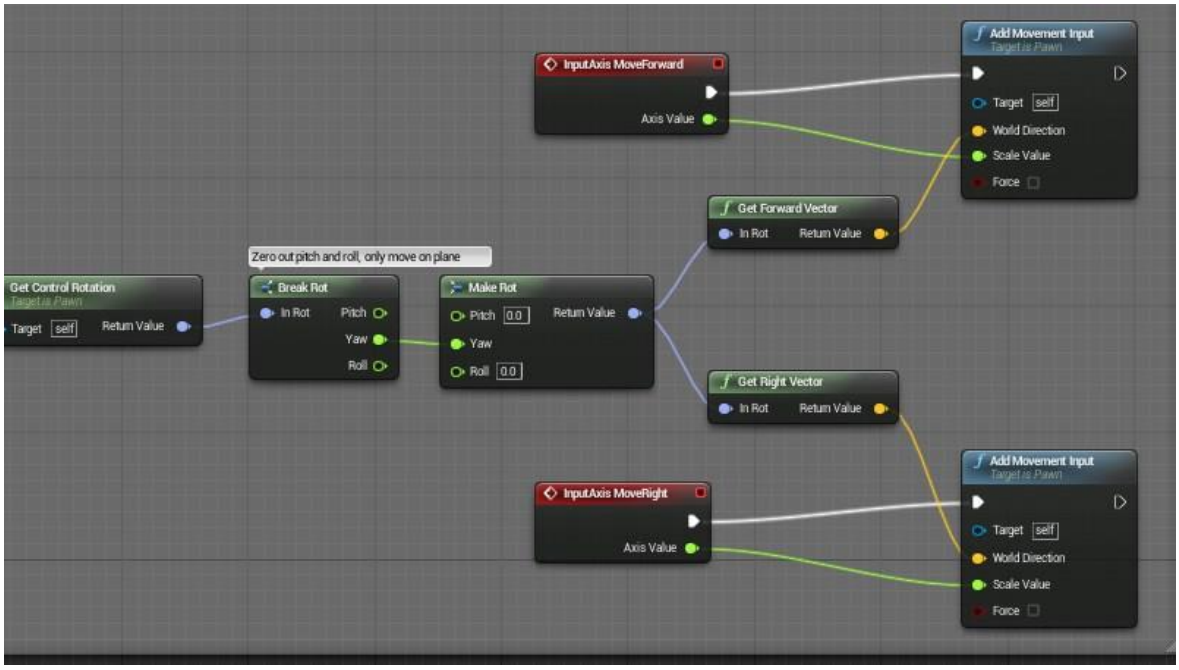
3.3.2 Unreal Engine 4

Unreal Engine 4 je komplexní sada vývojářských nástrojů pro vytváření her na úrovni komerčních titulů. Unreal Engine vyvíjí firma Epic Games již od roku 1998 a s jeho pomocí vytvořila mnoho úspěšných herních sérií např. Unreal Tournament a Gears of War. Unreal Engine 4 podporuje vývoj především pro operační systémy Windows a Mac OS X, mobilní operační systémy Android a iOS, herní konzole a v poslední verzi byla přidána i podpora pro vývoj aplikací na systémy virtuální reality.[17]

Obrázek 11 - Unreal Engine logo

Zdroj: http://images.thisisxbox.com/2015/03/1425334231-unreal-engine-logo.png

Unreal Engine 4 využívá jako hlavní programovací jazyk C++, ale v nové verzi byla představena funkce Blueprint Visual Scripting, která umožňuje programovat herní mechanismy bez znalosti programovacího jazyka. Tato funkce funguje na principu spojování diagramů, z nichž každý diagram zastupuje nějakou naprogramovanou funkci. Tato nová funkce zároveň přichází i s vlastním debuggerem, který umožňuje sledovat průběh funkcí a hodnot jejich proměnných přímo při spuštěné hře. Spojení těchto nástrojů poskytuje vývojáři výbornou vizualizaci vyskytnutých problémů a pomáhá v jejich nápravě.[18]



Obrázek 12 - Blueprint Visual Scripting

Zdroj: http://i1.wp.com/blog.digitaltutors.com/wp-content/uploads/2014/12/Visual-Scripting-image.jpg

Unreal Engine poskytuje profesionální nástroje pro tvorbu animací 3D objektů, nástroje pro tvorbu kaskádových efektů, pomocí kterých lze vytvářet věrohodné animace ohně, prachu a podobných částic. Díky pokročilým animačním nástrojům Matinee Cinematics lze vytvářet i filmy a dynamické scény. Dále Unreal Engine podporuje pokročilé renderovací funkce knihoven DirectX 11 a DirectX 12. To umožňuje rozsáhlé možnosti osvětlení scény, odrazy objektů v reflexních materiálech a spoustu dalších funkcí.[18]

Do roku 2009 byl Unreal Engine pouze proprietární software s uzavřeným kódem. V roce 2009 uvolnila společnost Epic Games nástroj Unreal Development Kit pro nekomerční účely zdarma.[17]

Licenční politika od roku 2012 umožňuje využívání zdarma plné verze Unreal Engine 4 a to i pro komerční účely. Ovšem pokud bude zisk za čtvrt roku přesahovat 3000$, tak je nutné odvádět 5% ze zisku jako poplatek.[17]

Hlavní výhodou Unreal Engine 4 je množství poskytovaných profesionálních nástrojů, které využívají studia profesionálních vývojářů a to zcela zdarma. Zejména se jedná o pokročilé nástroje pro tvorbu animací a nástroje pro ladění a optimalizaci. Unreal Engine 4 také umožňuje vytvářet hry na té nejvyšší grafické úrovni.

Za nevýhodu oproti Unity 3D považuji eshop Marketplace, který není tolik rozsáhlý jako Unity AssetStore a položky jsou poměrně drahé. Další nevýhodou je výpočetní náročnost u malých a jednoduchých projektů. Unreal Engine generuje mnoho zbytečného a náročného kódu a tak může být problematické spustit projekt na starších počítačích.

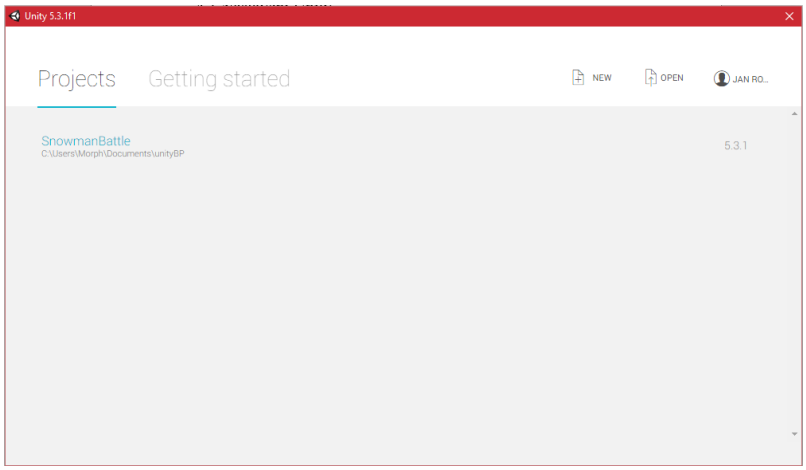
4. Praktická část

V této části práce bude popsán průběh vývoje aplikace za pomocí vývojového prostředí Unity 3D. Jedná se o vývoj jednoduché hry pro dva hráče na jednom počítači. Hráči jsou reprezentováni modelem sněhuláka a pohybují se ve čtvercové aréně. Cílem hry je porazit protivníka házením sněhových koulí. Hru jsem pojmenoval Snowmen Battle.

4.1 Nastavení Unity

4.1.1 Spuštění Unity

Při prvním zapnutí Unity 3D je nutné přihlásit se k účtu, který lze zdarma na internetových stránkách unity3d.com zaregistrovat. Po přihlášení se vybírá licence, pod kterou Unity budeme užívat. Na výběr je Personal Edition nebo Profesional Edition. U Profesional Edition je nutné zadat sériové číslo, tudíž jediná možnost výběru je Personal Edition. Celý nezkompilovaný projekt spolu s funkční aplikací se nachází na přiloženém CD.



Obrázek 13 - Výběr projektů

Zdroj: Vlastní

Po přihlášení a výběru licence se zobrazí základní nabídka s možnostmi výběru existujícího projektu nebo vytvoření nového. Při vytváření nového projektu lze nastavit, zda se jedná o 3D nebo 2D aplikaci a je zde možné naimportovat potřebné assety. Samozřejmostí je možnost volby umístění projektu a jeho jméno.

Pro projekt jsem zvolil opensource balíčky modelů, které jsou zdarma dostupné v Asset Store.

4.1.2 Rozhraní Unity 3D

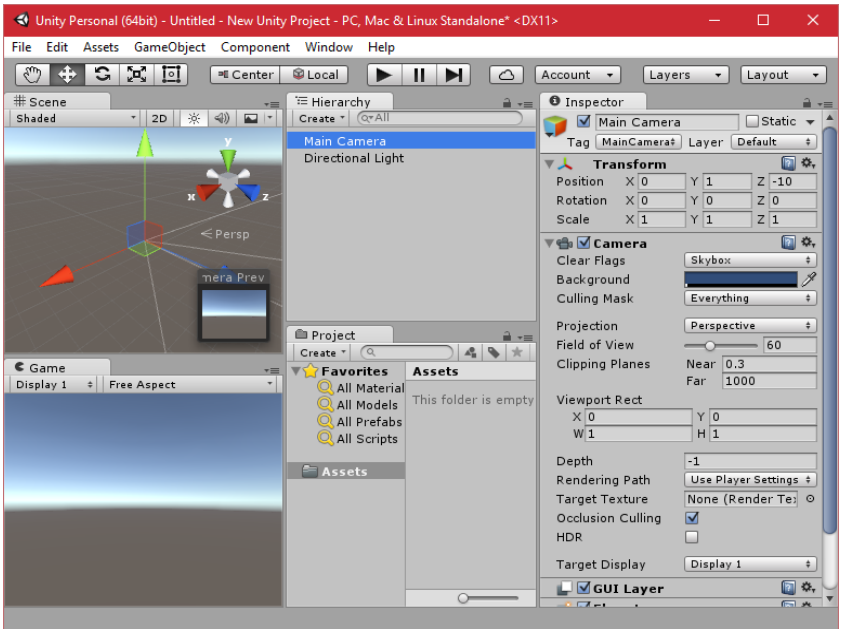
Základní rozhraní Unity (Obrázek 14) obsahuje 5 důležitých oken. První je okno Hierarchy. Hierarchy obsahuje seznam všech objektů nacházejících se na scéně a je možné zde objekty i vytvářet pomocí záložky Create nebo kliknutím pravým tlačítkem myši na již existující objekt. Zde je možné vytvořit i objekt dědící jeho vlastnosti.

Pro vizuální interpretaci objektů uložených v Hierarchy slouží okno Scene, kde objektům můžeme upravovat velikost, polohu a vlastnosti některých komponent. V tomto okně se vývojář může volně pohybovat a upravovat vizuální detaily.

Velmi podobné oknu Scene je okno Game, které zobrazuje náhled do hry a v tomto okně je možné hru spustit a hrát či krokovat její průběh a odladit případné chyby.

Pro detailní nastavení vlastností objektů a jejich komponent slouží okno Inspector. V tomto okně je možné modifikovat veškeré základní vlastnosti objektů jako je poloha a velikost, ale i přidávání dalších komponent, které upravují funkcionalitu objektu.

Okno Project slouží k zobrazení všech importovaných assetů a také objektů, které se do scény nahrávají pomocí skriptů a nejsou trvalou součástí scény.



Obrázek 14 - Rozhraní Unity

Zdroj: Vlastní

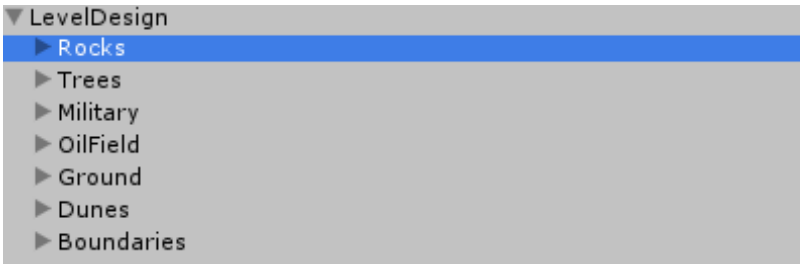
4.2 Tvorba prostředí

4.2.1 Teoretický návrh arény

Aréna, ve které se pohybují hráči, je čtvercového tvaru ohraničená mantinely. V aréně se nacházejí objekty, které slouží jako překážky hráčům i projektilům. Ve dvou protilehlých rozích se nachází místa, kde se objevují hráči na začátku každého kola. Aréna je osvětlena pod mírným úhlem, aby vynikly stíny objektů.

4.2.2 Realizace arény

Jako první věc pro udržení přehlednosti jsem vytvořil strukturu prázdných objektů, které slouží ke kategorizace 3D modelů na scéně, ale i pro snazší orientaci a usnadnění práce při výběru objektů. Názvy těchto objektů jsou odvozeny z názvů 3D modelů, které obsahují.



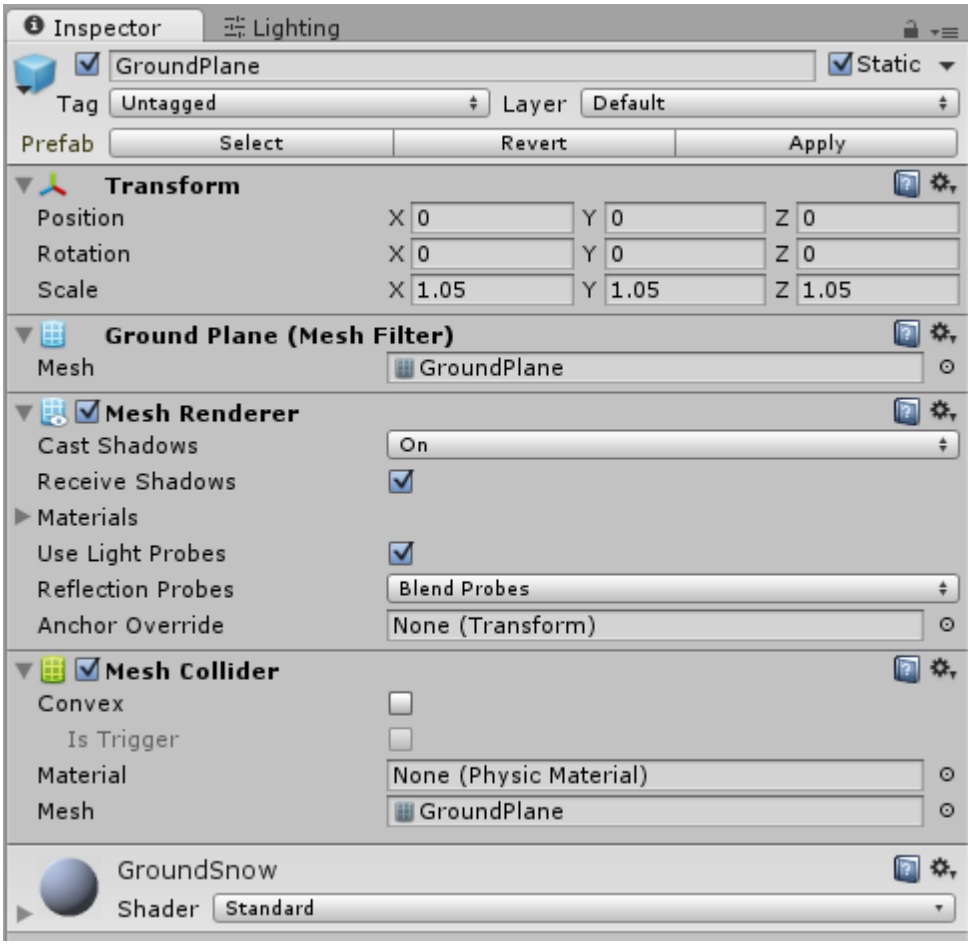
Obrázek 15 - Struktura 3D modelů na scéně

Zdroj: Vlastní

Jako první objekt je dobré vytvořit plochu, po které se hráči mohou pohybovat. Tato plocha se dá vytvořit jednoduše pomocí záložky Create v okně Hierarchy. Konkrétně se tento objekt nachází pod záložkou 3D object > Plane. Tento objekt jsem přejmenoval na GroundPlane a přetáhl na položku Ground v již vytvořené hierarchii objektů. Takto vytvořený objekt obsahuje následující komponenty (Obrázek 16):

- Transform – Zajišťuje nastavení polohy a velikosti objektu.
- GroundPlane (Mesh Filter) – Reprezentuje objekt, který je následně vykreslován pomocí komponenty Mesh Renderer.
- Mesh Renderer – Tato komponenta zajišťuje vykreslení objektu a jeho povrch pokryje texturou neboli materiálem.
- Mesh Collider – Collider je komponenta, která zajišťuje, že se objekt stává fyzickou překážkou a znemožňuje tak „procházení zdí“.

- Default-Material – Tato komponenta reprezentuje materiál, který je vykreslen na povrchu objektu.



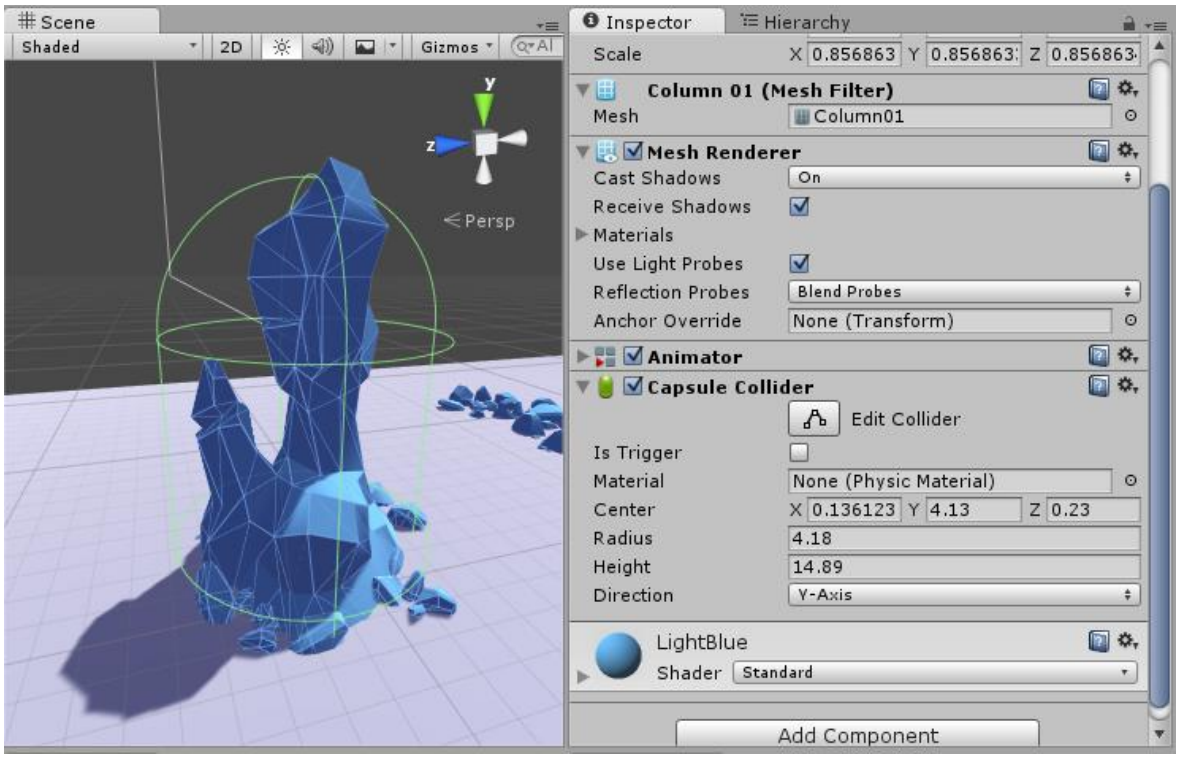
Obrázek 16 - Nastavení objektu GroundPlane

Zdroj: Vlastní

Všechny hodnoty pozice a rotace u komponenty Transform jsou nastaveny na hodnotu 0 a velikost je nastavena na hodnotu 1,05.

Protože Default-Material neodpovídá vzhledu, kterého jsem chtěl docílit, bylo nutné vytvořit nový materiál. V okně project jsem vytvořil novou složku s názvem Materials. V této složce jsem vytvořil nový materiál a pojmenoval jej GroundSnow. Pro přiřazení nového materiálu k objektu stačí chytit materiál pravým tlačítkem myši a přetáhnout jej na daný objekt. Vizuální nastavení vzhledu materiálu jsem provedl až v momentě, kdy byla aréna sestavena, abych docílil co nejlepšího možného vizuálního dojmu.

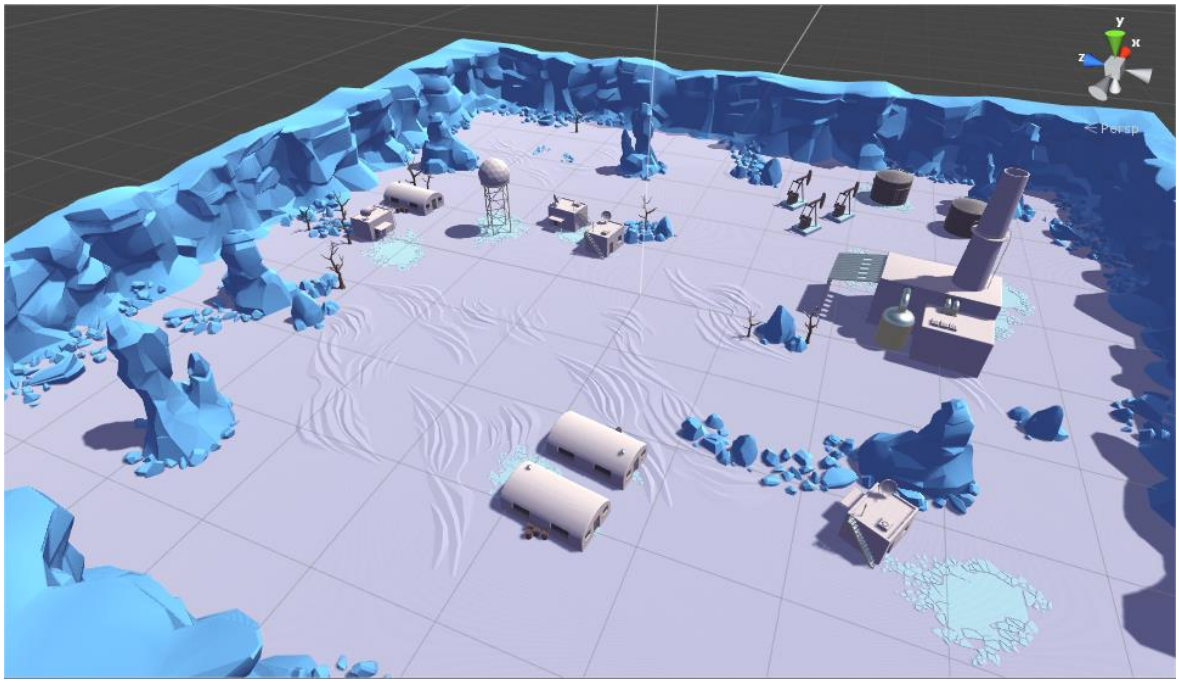
Ostatní 3D modely se importují přetažením z příslušné složky v okně Project do příslušné kategorie v okně Hierarchy. Po vložení 3D modelu do scény lze objekt libovolně přesouvat. Tyto 3D modely nemají vlastní přidělené materiály, které je nutné vytvořit a neobsahují komponenty typu collider. Komponentu collider přidáme tlačítkem „Add component“ v okně Inspector po výběru daného modelu ve scéně. Pro většinu objektů v této hře jsou použity tzv. primitivní collidery, které zahrnují jednoduché tvary jako kvádr, kouli nebo tvar kapsle. Pro účel hry není třeba generovat složité collidery, které vychází z tvaru 3D modelu. Primitivními collidery se také snižuje výpočetní náročnost hry. K jednomu 3D modelu je možné přidělit více colliderů a jejich kombinací pokrýt veškerou plochu daného modelu.



Obrázek 17 - Nastavení collideru

Zdroj: Vlastní

Hotové objekty lze duplikovat a tím zajistit větší členitost prostředí. Duplikované objekty obsahují všechny komponenty jako rodičovský objekt, tudíž je stačí pouze přemístit na vhodné místo.



Obrázek 18 - Výsledná aréna

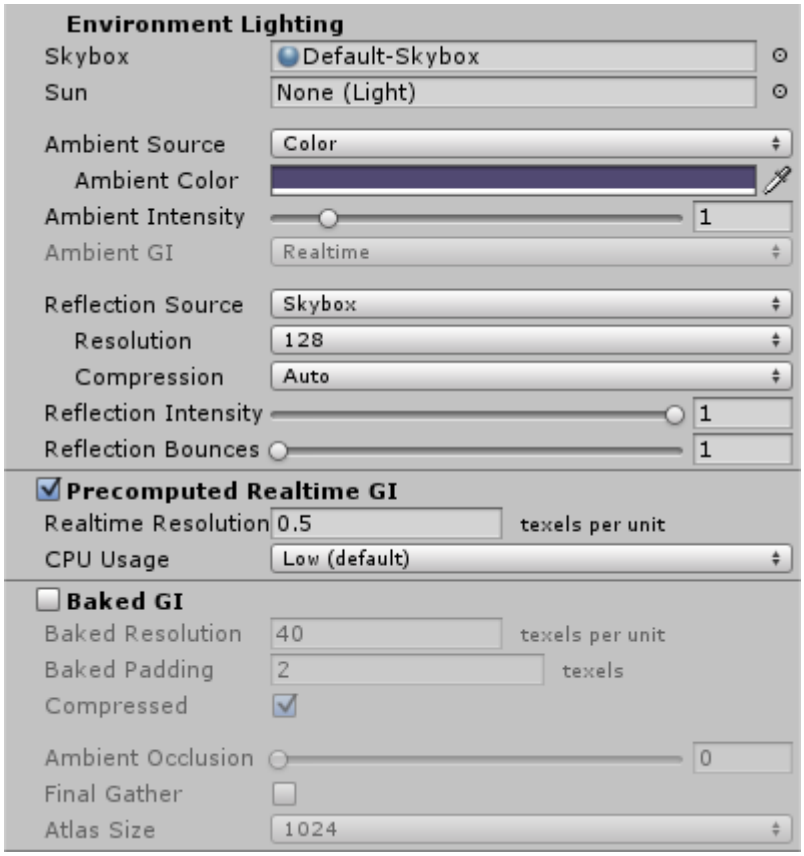
Zdroj: Vlastní

4.2.3 Osvětlení scény

Při vytvoření nového projektu se vytvoří základní světelný zdroj s názvem Directional light v okně Hierarchy. Toto světlo jsem ponechal v základním nastavení a pouze jsem upravil souřadnice a sklon světla.

Kromě světelných zdrojů viditelných v hierarchii objektů se na scéně nachází světlo generované prostředím tzv. Environmentální osvětlení. Pod záložkou Windows se nachází položka Lighting, která obsahuje nastavení environmentálního osvětlení (Obrázek 19).

Protože ve hře je využíván pohled ze shora a není zde tedy využíván žádný skybox, nastavil jsem položku Ambient Source na hodnotu color a zvolil barvu generovaného světla. Další změnu jsem provedl v položce Baked GI, kterou jsem úplně vypnul. Baked GI slouží k šetření výpočetní kapacity na zařízeních, které nemají dostatečné prostředky pro generování světla v reálném čase. V praxi to znamená, že se najdou všechny statické světelné zdroje na scéně a Baked GI je předvypočítá a uloží do světelné mapy a tak je v průběhu programu nemusí propočítávat znovu. Světlo generované v reálném čase naopak vypadá mnohem lépe, proto jsem se rozhodl tuto funkci nevyužívat.



Obrázek 19 - Nastavení environmentálního osvětlení

Zdroj: Vlastní

4.3 Tvorba hráče

Objekt reprezentující hráče (Obrázek 20) je načítán do hry pomocí skriptu a tak je nutné tento model uchovávat jako tzv. prefab. To znamená, že nebude při spuštění hry uchováván v hierarchii objektů, ale bude uložený v okně s assety. Ovšem pro účel editace je nutné jej přetáhnout do hierarchie objektů.

Hráči jsem na počátku přidělil komponety collider a rigidbody. Komponenta rigidbody zajistí, že objekt, kterému je přidělen, je ovlivňován fyzikálními zákony enginu. To znamená, že může spadnout z výšky nebo může být odsunut působením jiného objektu. Tento objekt měl vlastní texturovaný materiál, který jsem se rozhodl zachovat.

Důležitým nastavením je u hráče změnit položku Layer na hodnotu Players. To slouží k rozlišení od ostatních objektů na scéně.

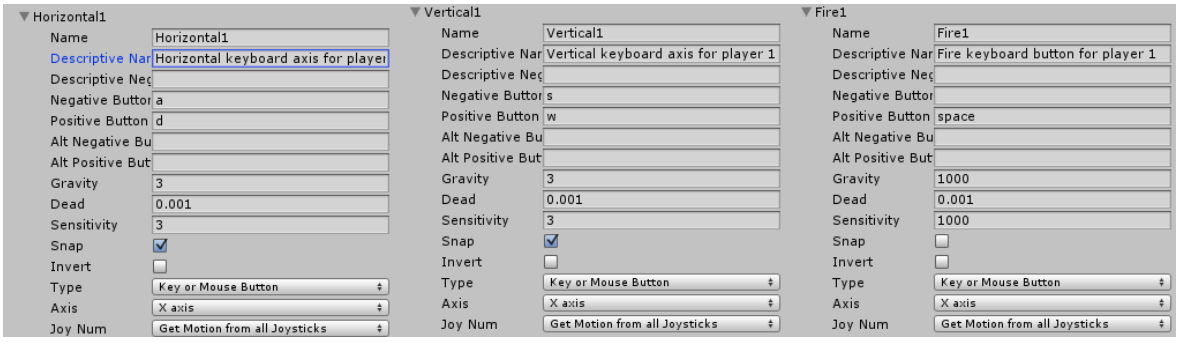
Obrázek 20 - Model hráče

Zdroj: Vlastní

4.3.1 Pohyb hráče

Pro zajištění pohybu hráče je třeba programu přidělit vstupní klávesy. K přidělení kláves slouží Input manager, který se nachází pod záložkou Edit > Project Settings >Input.

Hráč je ovládán čtyřmi směrovými klávesami pro pohyb a jedním tlačítkem pro střelbu. Pro jednoho hráče jsem tedy vyhradil 3 položky v Input manageru a pojmenoval je Horizontal1, Vertical1, Fire1. Klávesy pro pohyb prvního hráče jsou WASD a jako tlačítko pro střelbu je použitý mezerník. Pro druhého hráče jsem postupoval obdobně, jen jsem koncovou číslici zaměnil za „2“ a přidělil mu kurzorové klávesy pro pohyb a enter pro střelbu.



Obrázek 21 - Input manager

Zdroj: Vlastní

Dalším krokem bylo vytvoření skriptu jako komponenty hráčského objektu. Propojení Input manageru a skriptu je provedeno v následují ukázce kódu.

```
private void Start()
{
    priv_MovementAxisName = "Vertical" + pub_PlayerNumber;
    priv_TurnAxisName = "Horizontal" + pub_PlayerNumber;
}

private void Update()
{
    priv_MovementInputValue = Input.GetAxis(priv_MovementAxisName);
    priv_TurnInputValue = Input.GetAxis(priv_TurnAxisName);
}
```

Metoda `Start()` je volána pouze jednou a používá se pro prvotní nastavení proměnných ve skriptu. V tomto případě se do privátních proměnných `priv_MovementAxisName` a `priv_TurnAxisName` načte řetězec, kterým je pojmenována daná kategorie kláves v Input manageru.

V metodě `Update()` jsou do `priv_MovementInputValue` a `priv_TurnInputValue` periodicky načítány hodnoty ze vstupních kláves. Načítání hodnot probíhá zavoláním třídy `Input` a její metody `GetAxis` , která bere jako parametr název dané kategorie vstupních kláves.

Samotné metody pro pohyb a otáčení hráče jsou volány v metodě `FixedUpdate()`, která je načítána v pravidelných intervalech nezávisle na snímkové frekvenci, zatímco `Update()` je volána při každém načtení snímku. Metoda `FixedUpdate()` je doporučována pro použití při práci s komponentami Rigidbody, zejména kvůli přesnému propočítávání souřadnic.

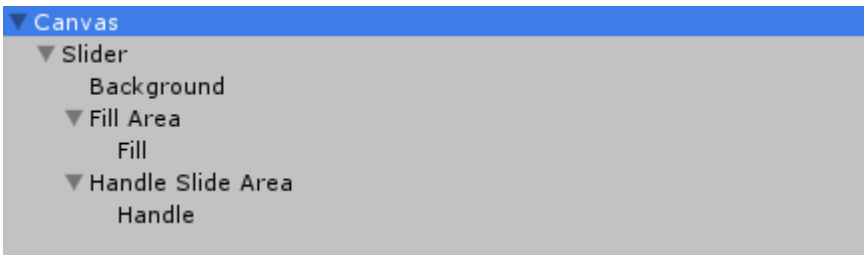
V následující ukázce kódu jsou zobrazeny funkce pro pohyb a otáčení hráče. Třída `Vector3` slouží práci se souřadnicemi ve 3D. Do proměnné `movement` je propočítávána vzdálenost na základě vstupní hodnoty (`priv_MovementInputValue`) a nastavené konstanty pro rychlost pohybu (`pub_Speed`). Položka `transform.forward` není nic jiného než zkratka pro určení osy, po které se hráč pohybuje vpřed nebo vzad. V soustavě os (X,Y,Z) se tato zkratka dá pro zjednodušení reprezentovat jako soustava čísel (0,0,1). Hodnota `Time.deltaTime` vyjadřuje časový rozdíl mezi načítáním jednotlivých snímků. Její využití zajistí plynulý pohyb objektu ve hře. Pro posunutí hráče o vypočtenou vzdálenost slouží metoda `MovePosition()`, která potřebuje jako parametr jen cílové souřadnice, tudíž stačí sečíst stávající polohu a vypočtenou vzdálenost.

```
    private void Move()
    {
        Vector3 movement = transform.forward * priv_MovementInputValue *
pub_Speed * Time.deltaTime;
        priv_Rigidbody.MovePosition(priv_Rigidbody.position + movement);
    }

    private void Turn()
    {
        float turn = priv_TurnInputValue * pub_TurnSpeed * Time.deltaTime;

        Quaternion turnRotation = Quaternion.Euler(0f, turn,0f);

        priv_Rigidbody.MoveRotation(priv_Rigidbody.rotation * turnRotation);
    }
```

Metoda `Turn()` pracuje velmi podobně. V první fázi se vypočítá hodnota rotace založená na konstantě rychlosti otáčení (`pub_TurnSpeed`) a vstupní hodnoty z klávesnice (`priv_TurnInputValue`). Zde se nepočítá s pohybem po osách, ale s rotací. Veškeré operace s rotacemi reprezentuje třída `Quaternion`, obsahující metodu `Euler()`, která vypočtenou číselnou hodnotu převede na rotaci. Celý kód s komentářem se nachází v příloze.

4.3.2 Životy hráče

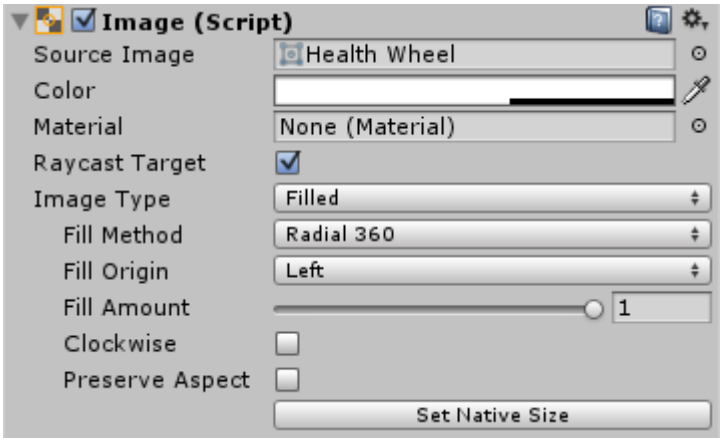
Životy hráče jsou reprezentovány barevným kruhem, který se nachází pod objektem hráče. Základní stavební kámen podobného prvku je Slider, ten se při vytváření nového objektu nachází v záložce UI. Prvky UI si automaticky vytváří mateřský objekt nazvaný Canvas neboli plátno. Canvas jsem přetáhl na hráče a tím jsem docílil toho, že se bude pohybovat spolu s hráčem. Dále jsem jej upravil na příhodnou velikost a pozici. Slider (Obrázek 22) po vytvoření obsahuje objekty Background, Fill Area a Handle Area. Pro účely zobrazení životů není Handle Area potřeba a tak jsem jej smazal. Důležité nastavení u Slideru je odškrtnout položku Interactable, která umožňuje se sliderem ve hře manipulovat.



Obrázek 22 - Canvas

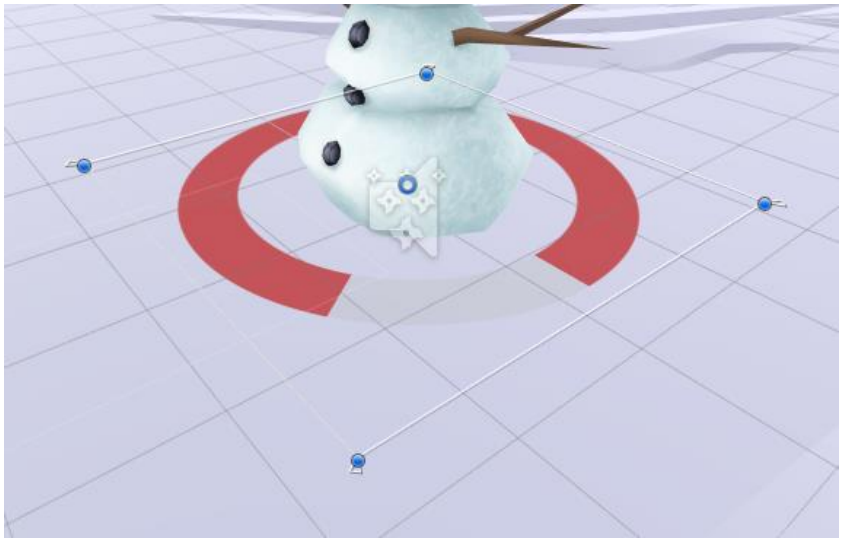
Zdroj: Vlastní

Objektu Fill a Background jsem přidělil obrázek kruhu a nastavil Fill Method na hodnotu radial 360.



Obrázek 23 - Fill

Zdroj: Vlastní



Obrázek 24 - Health bar

Zdroj: Vlastní

V další fázi jsem vytvořil skript, který udržuje rotaci ukazatele životů na výchozích hodnotách, aby se životy neotáčely společně s hráčem, což by pod určitým úhlem vedlo k nepřehlednosti. Tento skript jsem umístil jako komponentu k objektu HealtSlider.

Následující kód je velmi jednoduchý. Na počátku načte do proměnné výchozí hodnoty pozice rotace objektu HealthSlider a v metodě `Update()` jsou hodnoty pozic neustále z této proměnné obnovovány. Celý kód s komentářem se nachází v příloze.

```
private void Start()
{
    priv_RelativeRotation = transform.parent.localRotation;
}


private void Update()
{
    if (pub_UseRelativeRotation)
        transform.rotation = priv_RelativeRotation;
}
```

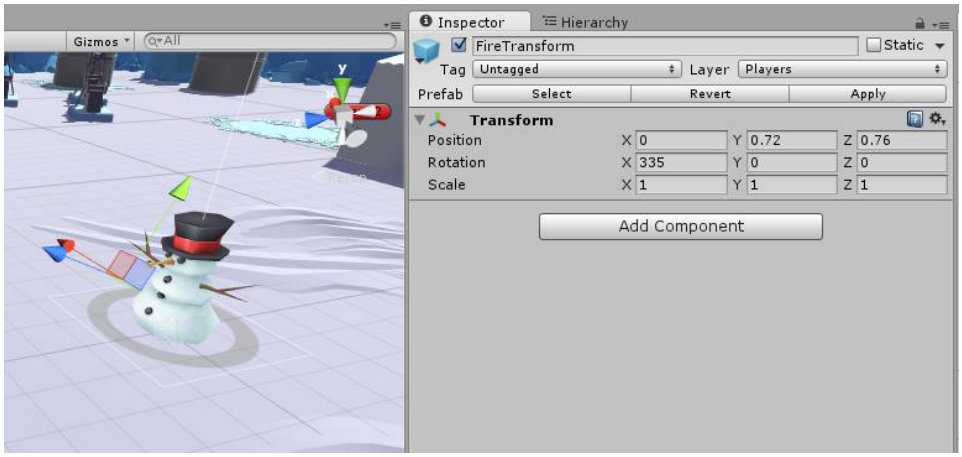
Hráči jsem přidělil další skript, který obsahuje několik stěžejních funkcí. Jedná se o úvodní nastavení životů v metodě `OnEnable()`, veřejnou funkci `TakeDamage()` pro odečtení určitého počtu životů a funkci `SetHealthUI()`, která stav životů nastaví do grafického ukazatele. Celý kód s komentářem se nachází v příloze.

```
private void OnEnable()
{
    priv_CurrentHealth = pub_StartingHealth;

    SetHealthUI();
}

public void TakeDamage(float amount)
{
    priv_CurrentHealth -= amount;

    SetHealthUI();

}

private void SetHealthUI()
{
    pub_Slider.value = priv_CurrentHealth;
    pub_FillImage.color = pub_HealtColor;
}
```

4.3.3 Střelba

V této hře jsem chtěl, aby délka letu sněhové koule byla závislá na době držení tlačítka určeného pro střelbu. Čím déle bude hráč držet tlačítko, tím sněhovou kouli dohodí dále. Základem pro střelbu je vytvoření projektilu, který při kolizi způsobí hráči poškození. Tomuto projektilu je nutné udat nějaký směr a rychlost.

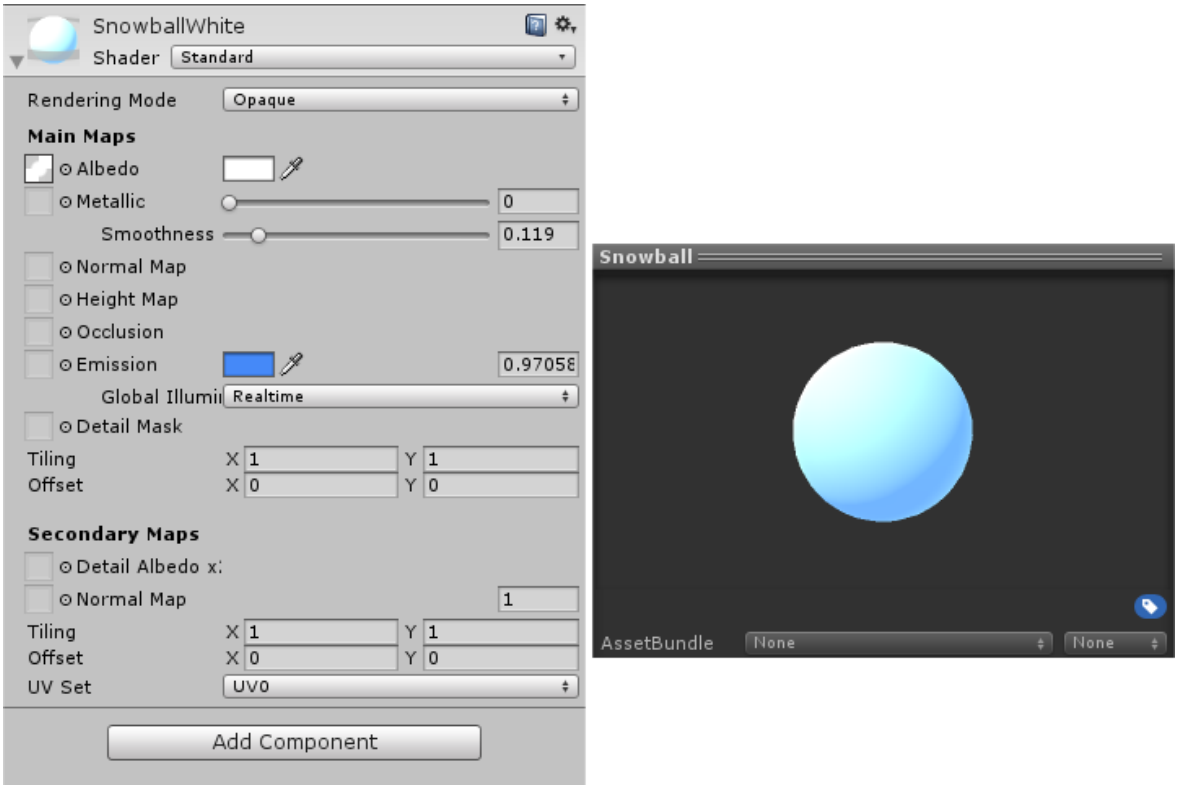
Pro usměrnění střely jsem pod objektem hráče vytvořil prázdný objekt potomka, který jsem přesunul před hráče a nastavil mu určitý sklon. Tento objekt jsem pojmenoval FireTransform (Obrázek 25) podle funkce, kterou bude mít a podle jediné komponenty, kterou obsahuje.



Obrázek 25 - FireTransform

Zdroj: Vlastní

Jako další je nutno vytvořit projektil, který bude hráčem vystřelen. V okně Hierarchy jsem vytvořil 3D objekt - kouli. Podobně jako při vytváření země má tento objekt v základu vlastní collider. U tohoto collideru je nutné zaškrtnout pole s názvem Is Trigger. Toto pole zajistí, že collider může při nárazu spustit nějakou událost. V tomto případě bude událostí způsobení poškození hráči. Střela musí být zároveň ovlivnitelná působením fyzikálních zákonů, proto jsem přidal komponentu Rigidbody. Poslední základní věcí je vytvoření materiálu, který by odpovídal vzhledu sněhové koule (Obrázek 26).



Obrázek 26 - Materiál sněhové koule

Zdroj: Vlastní

Pro zajištění funkcionality projektilu jsem vytvořil skript, který obsahuje funkci na propočítání poškození v závislosti na vzdálenosti dopadu sněhové koule od hráče. Následující ukázka kódu obsahuje funkci spuštěnou komponentou collider díky nastavení Is Trigger. Nejprve se do pole `Collider[] colliders` pomocí metody `OverlapSphere()` načtou veškeré collidery vrstvy Players v určeném dosahu okolo pozice projektilu. Následně je toto pole procházeno cyklem. Abych docílil efektu odstrčení hráče při zásahu, tak jsem z právě procházeného collideru nahrál komponentu Rigidbody cílového objektu do proměnné

`Rigidbody targetRigidbody`. Přesně pro tuto funkci má třída `Rigidbody` metodu `AddExplosionForce`, která vypočítá sílu odstrčení objektu rigidbody od pozice projektilu v určeném okruhu a objekt odsune.

V dalším kroku cyklu je třeba odečíst určité množství životů v závislosti na přesnosti zásahu. Opět inicializuji příslušnou komponentu cílového objektu, tentokrát se jedná o vytvořený skript, který pracuje s životy hráče. Tuto komponentu jsem inicializoval do proměnné `SnowmanHealth targetHealth`. Ve skriptu SnowmanHealth se nachází veřejná metoda `TakeDamage()`, jejíž funkce je odečtení životů hráče. Této metodě jako parametr předáváme hodnotu poškození.

```
private void OnTriggerEnter(Collider other)
{
    Collider[] colliders = Physics.OverlapSphere(transform.position,
    pub_ExplosionRadius, pub_SnowmanMask);

    for (int i = 0; i < colliders.Length; i++)
    {
        //Inicializace komponenty RigidBody z položky pole colliders
        Rigidbody targetRigidbody = colliders[i].GetComponent<Rigidbody>();

        if (!targetRigidbody)
            continue;

       //metoda AddExplosionForce odtlačí cílový objekt rigidBody směrem od
       pozice projektilu v určeném rádiu
       targetRigidbody.AddExplosionForce(pub_ExplosionForce,
       transform.position, pub_ExplosionRadius);

        //Inicializace skriptu SnowmanHealth z cílového objektu
       SnowmanHealth targetHealth =
       targetRigidbody.GetComponent<SnowmanHealth>();

        if (!targetHealth)
            continue;

        //Zavolání metody TakeDamage() umístěné ve skriptu SnowmanHealth
        targetHealth.TakeDamage(CalculateDamage(targetRigidbody.position));
    }


    //zničení objektu sněhové koule
    Destroy(gameObject);

}
```

Výpočet hodnoty uděleného poškození probíhá v metodě `CalculateDamage()`, tato metoda počítává hodnotu uděleného poškození ze vzdálenosti hráče od projektilu. Rozdílem pozice projektilu a pozice hráče se vypočítá vektor mezi projektilem a cílem. Ke zjištění délky vektoru slouží funkce `magnitude` třídy `Vector3`. Při výpočtu poškození je nutné, aby se hodnota poškození pohybovala mezi nulou a určenou hodnotou maximálního poškození, která je uložená v proměnné `pub_MaxDamage`. Odečtením délky vektoru od maximálního dosahu exploze a následným vydělením tohoto rozdílu maximálním dosahem jsem získal procentuální vyjádření přiblížení hráče a ohniska exploze. Toto číslo se pohybuje v rozmezí 0 až 1 a je tedy možné získat hodnotu uděleného poškození prostým vynásobením s maximálním možným poškozením. Funkce `Mathf.Max(0f, damage)` zajistí, že se vždy poškození bude pohybovat v cíleném rozsahu. Celý kód s komentářem se nachází v příloze.

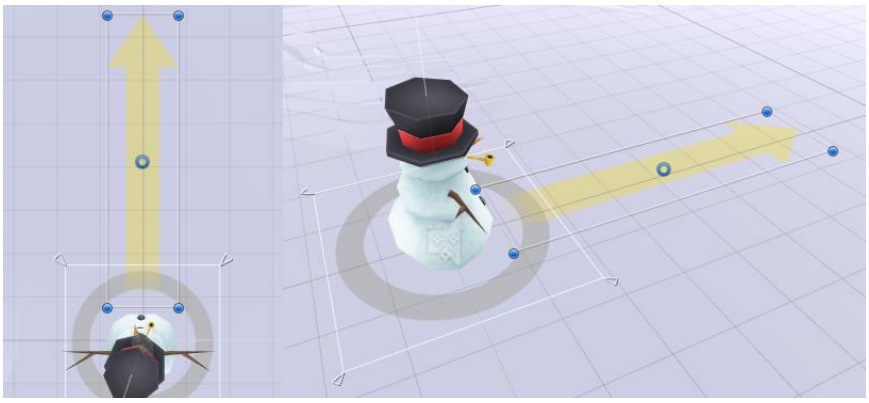
```
private float CalculateDamage(Vector3 targetPosition)
    {
        //Vypočtení vektoru mezi projektilem a cílem
        Vector3 distanceFromTarget = targetPosition - transform.position;

        //Vypočtení relativní hodnoty vzdálenosti
       float distance = (pub_ExplosionRadius - distanceFromTarget.magnitude) /
      pub_ExplosionRadius;

        //výpočet poškození
        float damage = distance * pub_MaxDamage;

        //Ujištění že minimální poškození bude vždy 0
        damage = Mathf.Max(0f, damage);

        return damage;
    }
```

Sněhovou kouli je třeba uchovávat jako asset, protože bude volána skrze kód. Celý objekt jsem proto přetáhnul z okna Hierarchy do složky prefabs v okně Project a smazal ho ze scény.

V poslední fázi jsem vytvořil skript, který načte vytvořenou sněhovou kouli a vystřelí ji v závislosti na délce držení tlačítka. A pro orientaci směru a síly výstřelu jsem přidal ukazatel ve tvaru šipky, který se s délkou držení tlačítka pro výstřel zvětšuje.

Podobně jako u ukazatele životů jsem vytvořil objekt slider, tentokrát již pod existujícím objektem Canvas, který je umístěný pod objektem hráče. Jediná provedená nastavení byla

nastavení pozice a velikosti, změna Fill Image na obrázek šipky, vypnutí položky Interactable, smazání nežádoucích objektů (background a handle area) a v poslední řadě přejmenování na AimSlider.



Obrázek 27 - AimSlider

Zdroj: Vlastní

Posledním krokem bylo vytvoření skriptu u objektu hráče. Následující ukázka kódu zobrazuje klíčové funkce. Funkce zajišťující střelbu reprezentuje řetěz čtyř podmínek pod metodou Update. V první podmínce program zjišťuje, zda je aktuální úroveň nabití vyšší než nastavené maximum a zároveň hráč ještě nevystřelil. Pokud ano, nastaví se maximum do proměnné s aktuální silou nabití a zavolá se metoda provádějící výstřel (`Fire();`). Další podmínka detekuje stlačení tlačítka pro střelbu a vyresetuje aktuální sílu nabití na určené minimum. Pokud je tlačítko pro střelbu drženo, tak se v následující podmínce zvětší úrovně nabití střely a provede se grafická interpretace na ukazateli. Poslední podmínka detekuje puštění tlačítka a volá metodu `Fire()`.

```
    private void Update()
    {
        pub_AimSlider.value = pub_MinLaunchForce;   //nastavení grafické
interpretace minimálního nabití

        //řetěz podmínek zajišťující nabíjení a výstřel
        if (priv_CurrentLaunchForce >= pub_MaxLaunchForce && !priv_Fired)
        {
            //výstřel pokud aktuální hodnota nabití přesáhne maximum
            priv_CurrentLaunchForce = pub_MaxLaunchForce;
            Fire();
        }
        else if (Input.GetButtonDown(priv_FireButton))
        {
```

```
            //stlačení tlačítka výstřelu
            priv_Fired = false;
            priv_CurrentLaunchForce = pub_MinLaunchForce;
        }
        else if (Input.GetButton(priv_FireButton) && !priv_Fired)
        {
            //držení tlačítka výstřelu a nabíjení
            priv_CurrentLaunchForce += priv_ChargeSpeed * Time.deltaTime;

            pub_AimSlider.value = priv_CurrentLaunchForce;
        }
        else if (Input.GetButtonUp(priv_FireButton) && !priv_Fired)
        {
            //uvolnění tlačítka výstřelu a výstřel
            Fire();
        }
    }
```

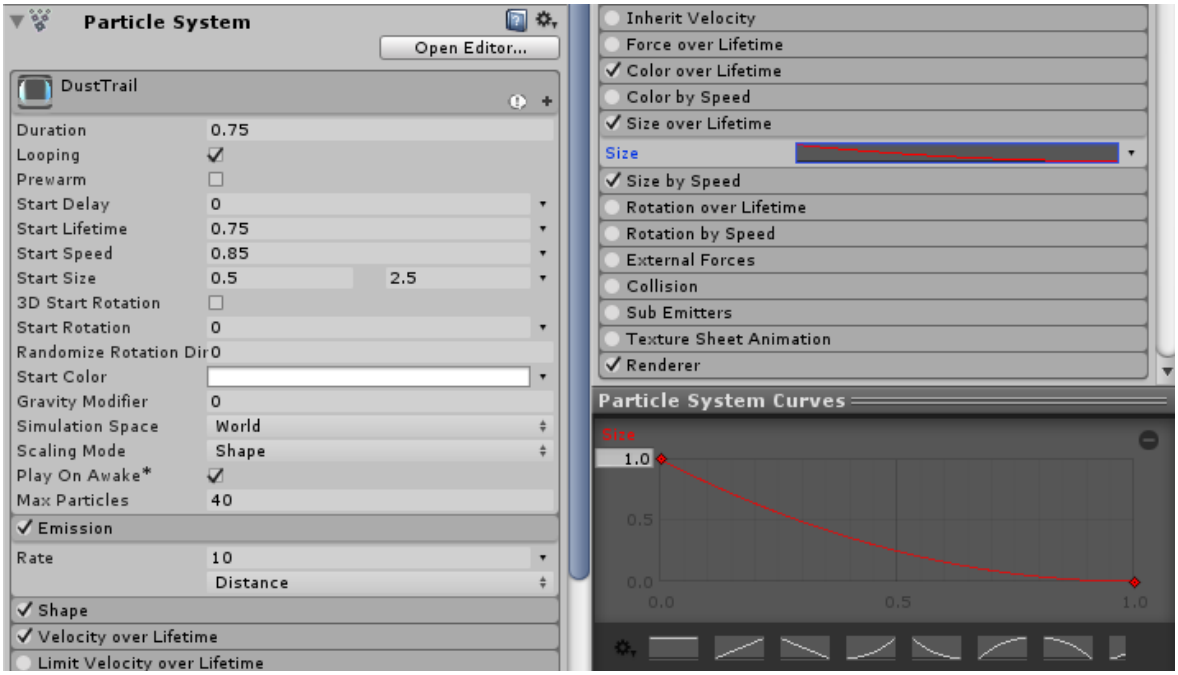
Metoda `Fire()` vytváří instanci rigidbody objektu projektilu pomocí funkce `Instantiate()`, která zároveň objektu přidělí pozici a rotaci objektu FireTransform. Následně tomuto objektu přidělí rychlost pohybu vypočtenou z aktuální úrovně nabití a vektoru, který udá směr pohybu. Jako poslední je zde vyresetování úrovně nabití. Celý kód s komentářem se nachází v příloze.

```
    private void Fire()
    {
        priv_Fired = true;

        //načtení sněhové koule do instance třídy Rigidbody
        Rigidbody snowballInstance = Instantiate(pub_Snowball,
       pub_FireTransform.position, pub_FireTransform.rotation) as Rigidbody;

        //Výstřel pomocí pozičních hodnot objektu FireTransform
        snowballInstance.velocity = priv_CurrentLaunchForce *
       pub_FireTransform.forward;

        //vyresetování hodnoty nabití
        priv_CurrentLaunchForce = pub_MinLaunchForce;
    }
```

4.3.4 Částicové efekty a audio

V Unity je možné vytvářet jednoduché animace, které se využívají například pro efekty prachu, exploze, ohně nebo deště. V této aplikaci využívám dva typy částicových efektů. Jeden generuje částice v závislosti na rychlosti pohybu a druhý je používán pro exploze. S explozemi souvisí i přehrání zvukových efektů.

Částicové efekty se vytváří stejně jako ostatní objekty pomocí tlačítka Create v okně Hierarchy a položky Particle Systems. Jako první jsem vytvořil efekty, které používám jako sněhovou stopu za hráčem a jako efekt odpadávajícího sněhu za sněhovou koulí. Pro tyto částicové efekty jsem využil stejný materiál jako pro sněhovou kouli. K docílení generování částic na základě rychlosti pohybu jsem nastavil položku Emission na distance a zároveň jsem změnil položku Size over Lifetime na klesající křivku. Touto křivkou jsem docílil toho, že se generované částice postupně zmenšují (Obrázek 28). Takto nastavený Particle System jsem přetáhl na objekty sněhové koule a hráče.



Obrázek 28 - Particle System

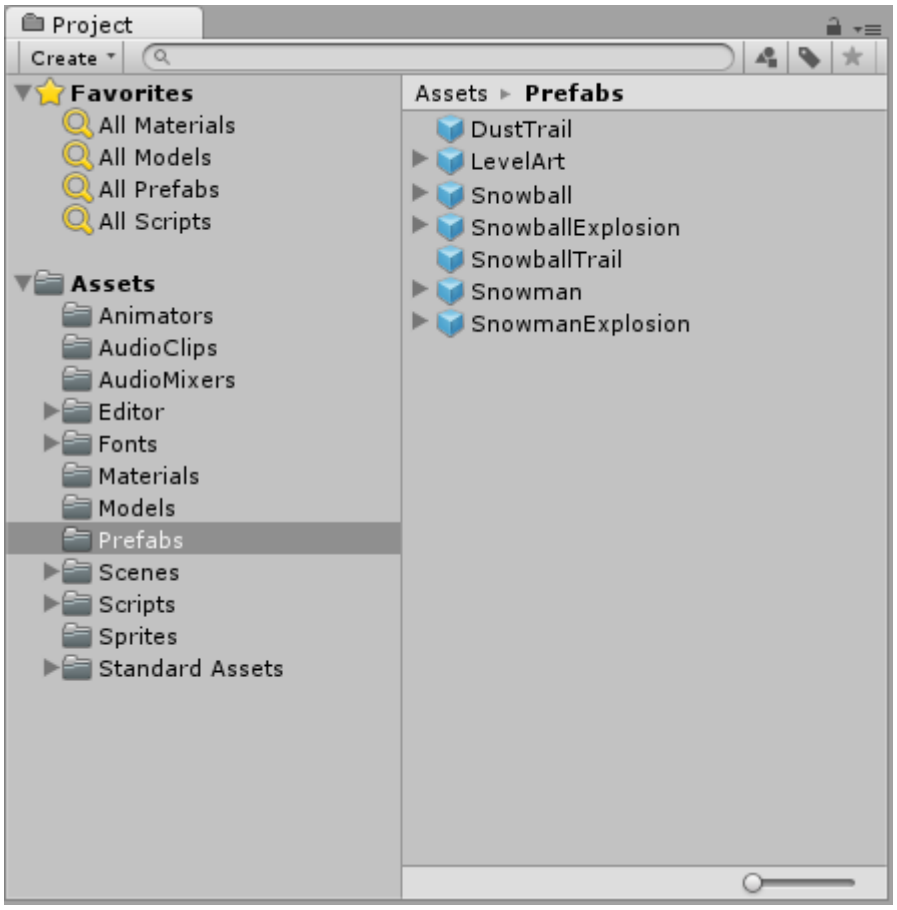
Zdroj: Vlastní

Podobným způsobem jsem vytvořil i efekt rozpadu sněhové koule a hráče. Jediný zásadní rozdíl spočívá v nastavení Emission na hodnotu Time a přidělení komponenty Audio Source, do které jsem z assetů přidělil zvukový efekt. Zároveň je nutné umístit potřebný kód pro přehrání na klíčová místa ve skriptech. Následující ukázka kódu představuje přehrání částicových efektů a audiostopy. Nejdříve se částicový efekt umístí na pozici objektu. V dalším kroku se aktivuje a přehraje.

```
priv_ExplosionParticles.transform.position = transform.position;
priv_ExplosionParticles.gameObject.SetActive(true);
priv_ExplosionParticles.Play();
priv_ExplosionAudio.Play();
```

4.3.5 Dokončení hráče

Hráči jsou do hry nahráváni pomocí kódu. Je tedy nutné je uložit, podobně jako sněhovou kouli, v podobě assetu a následně jej vymazat ze scény. Do složky Prefabs jsem si současně uložil i objekt LevelDesign a veškeré hotové částicové efekty.



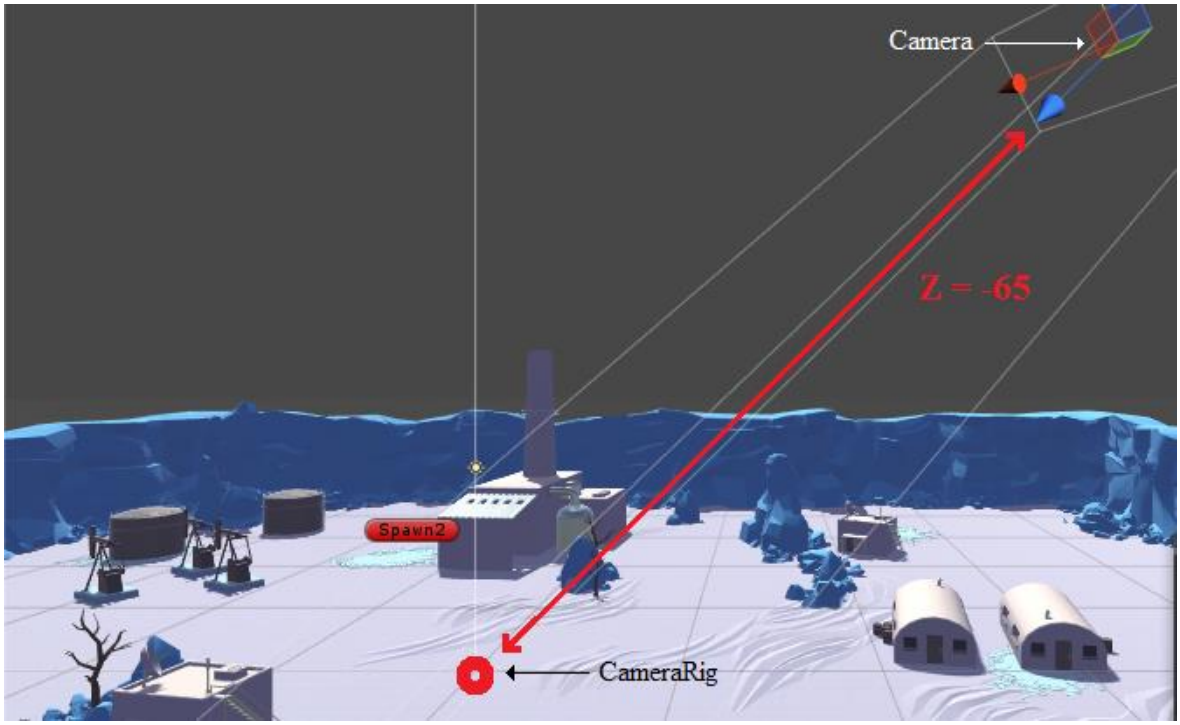
Obrázek 29 - Prefabs

Zdroj: Vlastní

4.4 Nastavení kamery a pohyb kamery

Nastavení kamery jsem zvolil takové, aby byla zaměřená na hráče. Chtěl jsem, aby se kamera přibližovala, pokud jsou hráči blízko sebe a zároveň se oddalovala se zvětšující se vzdáleností mezi hráči. Klíčové je, aby se nikdy hráč neobjevil mimo dohled kamery.

V první fázi jsem Kameře vytvořil mateřský prázdný objekt CameraRig, který je umístěný na souřadnicích (0,0,0). CameraRig má nastavený sklon os X = 40 a Y = 60. Samotný objekt s komponentou Camera jsem pouze odsadil nastavením souřadnice Z na hodnotu -65. Tímto jsem docílil konstantní vzdálenosti mezi herní plochou a kamerou, která ji snímá (Obrázek 30).



Obrázek 30 - Nastavení Camera a CameraRig

Zdroj: Vlastní

Funkcionalitu kamery jsem upravil skriptem, který ovládá pozici kamery a přiblížení. V následující ukázce kódu se nachází metoda zajišťující pohyb kamery. V metodě `Move()` se volá metoda, která počítá středové souřadnice mezi hráči na scéně a uloží je do proměnné `priv_TargetPosition`. Přesunutí kamery zajistí metoda `Vector3.SmoothDamp()`, která bere jako hlavní parametr aktuální pozici kamery (`transform.position`), cílovou pozici

(priv_TargetPosition), proměnnou (ref priv_MoveVelocity), kterou si funkce sama při každém volání edituje a číselnou hodnotu (pub_DampTime), která vyjadřuje zpomalení přesunu pro hladší pohyb.

```
private void Move()
    {
        //volání funkce, která najde středové souřadnice mezi hráči
        CameraPosition();

        //plynulé přesunutí kamery na cílovou pozici
        transform.position = Vector3.SmoothDamp(transform.position,
priv_TargetPosition, ref priv_MoveVelocity, pub_DampTime);
    }
```

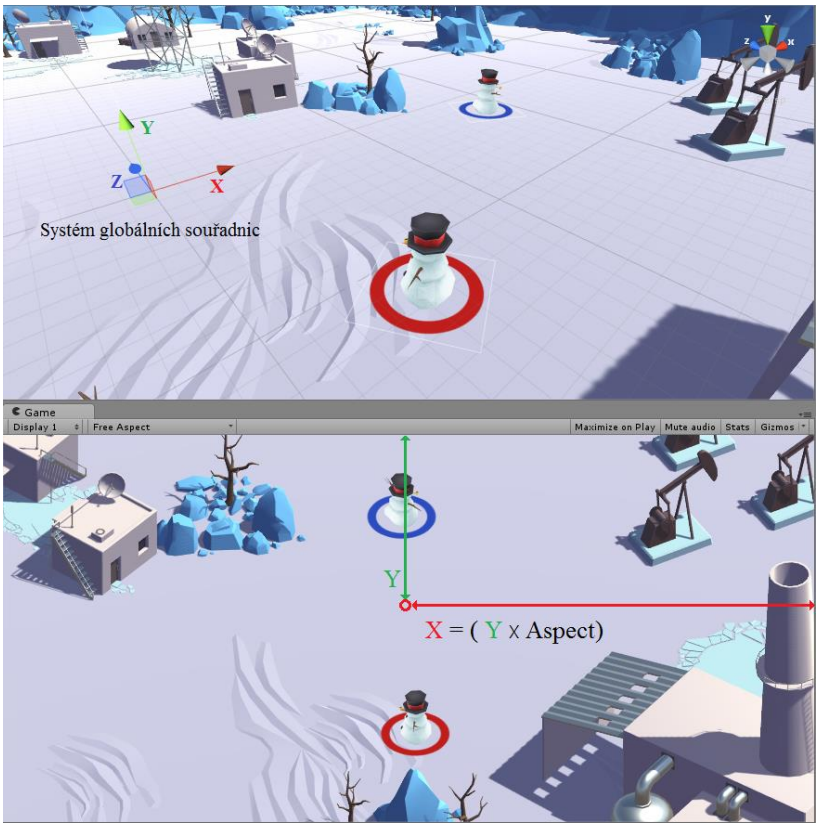
Metoda hledající středovou pozici mezi hráči se nazývá CameraPosition. Tato metoda prochází pole, kde jsou uloženy souřadnice všech hráčů na scéně. Do proměnné avPos jsou souřadnice hráčů sečteny a nakonec jsou vyděleny počtem hráčů. Tímto výpočtem jsem získal průměr souřadnic mezi hráči na ploše. Tyto nové souřadnice jsem uložil do proměnné priv_TargetPosition, která vyjadřuje cílové souřadnice kamery.

```
private void CameraPosition()
    {
        Vector3 avPos = new Vector3();
        int numTargets = 0;

        //procházení pole ve kterém jsou uloženy souřadnice hráčů
        for (int i = 0; i < pub_Targets.Length; i++)
        {
            //Pokud je objekt aktivní provedou se výpočty
            if (pub_Targets[i].gameObject.activeSelf) {

            //Sčítání souřadnic všech cílových objektů
            avPos += pub_Targets[i].position;

            //počet cílů
            numTargets++;
            }
        }
        //výpočet průměrné pozice
        if (numTargets > 0)
            avPos /= numTargets;

        //udržení hodnot osy Y na základní hodnotě
        avPos.y = transform.position.y;

        //naplnění proměnné cílovými souřadnicemi
        priv_TargetPosition = avPos;
    }
```

Druhou důležitou funkcí je funkce zajišťující adekvátní přiblížení kamery. Následující kód funguje téměř totožně jako metoda Move(). Rozdíl je jen v třídě, ze které se volá metoda `SmoothDamp`. Místo souřadnic se zde využívá číslo, vyjadřující velikost přiblížení kamery a pro určení přiblížení je použita metoda `CameraSize()`.

```
    private void Zoom()
     {
         //volání metody, která počítá doporučené přiblížení kamery a uložení hodnoty
do proměnné
         float size = CameraSize();
         //přiblížení kamery
         priv_Camera.orthographicSize =
Mathf.SmoothDamp(priv_Camera.orthographicSize, size, ref priv_ZoomSpeed,
pub_DampTime);
     }
```

Metoda `CameraSize()` pracuje se souřadnicemi v lokálním prostoru kamery. Lokální souřadnice kamery nepracují s osami v prostoru, ale se středovými osami náhledu snímané obrazovky (Obrázek 31).



Obrázek 31 - Rozdíl mezi globálními souřadnicemi a lokálními souřadnicemi kamery

Zdroj: Vlastní

V následujícím kódu je naprogramována funkce zmenšování a zvětšování kamery v závislosti na vzdálenosti hráčů od sebe. Nejprve jsem převedl do lokálních souřadnic kamery hodnotu cílové pozice pomocí metody `InverseTransformPoint()` a uložil je do proměnné `LocalTargetPos`. V dalším kroku se nachází cyklus for, který podobně jako v metodě `CameraPosition()` prochází pole se souřadnicemi hráčů na scéně. Pokud narazí na aktivní objekt hráče, tak opět převede jeho souřadnice do lokálního systému souřadnic. Rozdílem souřadnic cílové pozice kamery (střed kamery) a souřadnic hráče jsem získal vzdálenost hráče od středu kamery. V poslední fázi cyklu se do proměnné size ukládá vždy co největší hodnota vzdálenosti mezi středem kamery a pozicí hráče. Aby hráči nebyli na okraji obrazovky, přičítám do proměnné size hodnotu, která zvětší prostor mezi hráčem a okrajem obrazovky. Jako poslední je zde ujištění, že se kamera nikdy nepřiblíží až příliš k zemi, pokud budou hráči u sebe. Celý kód s komentářem se nachází v příloze.

```
    private float CameraSize()
    {
        //Převedení globálních souřadnic cílové pozice do lokálních hodnot kamery
(výhled)
        Vector3 LocalTargetPos =
transform.InverseTransformPoint(priv_TargetPosition);
        float size = 0f;
        //procházení pole ve kterém jsou uloženy souřadnice hráčů
        for (int i = 0; i < pub_Targets.Length; i++)
        {
            //Pokud je objekt aktivní, provedou se výpočty
            if (pub_Targets[i].gameObject.activeSelf) {
            //Převedení pozic hráčů z globálních hodnot do lokálních hodnot kamery
            Vector3 LocalPLayerPos =
transform.InverseTransformPoint(pub_Targets[i].position);
            //Vypočtení vzdálenosti hráče od středu kamery
            Vector3 CenterPosToTarget = LocalPLayerPos - LocalTargetPos;
            //Porovnání hodnot size a vzdálenosti hráče od středu kamery na
vertikální ose a nahrání větší hodnoty do proměnné size
            size = Mathf.Max (size, Mathf.Abs (CenterPosToTarget.y));
            //Porovnání hodnot size a vzdálenosti hráče od středu kamery na
horizontální ose a nahrání větší hodnoty do proměnné size
            size = Mathf.Max (size, Mathf.Abs (CenterPosToTarget.x) /
priv_Camera.aspect);
            }
        }

        //zvětšení odsazení okrajů kamery od hráče
        size += pub_ScreenEdgeBuffer;

        //Ujištění že velikost kamery nebude nikdy větší než nastavené minimum
        size = Mathf.Max(size, pub_MinSize);

        return size;
    }
```

4.5 Herní mechanismy

Tato část se zabývá poslední fází vývoje hry. Jsou zde naprogramovány herní mechanismy starající se o načítání hráčů na herní plochu, počítání odehraných kol, počítadlo výher a pravidla hry.

Aby program věděl kam načíst hráče, tak jsem vytvořil dva nové prázdné objekty, pojmenoval je Spawn1 a Spawn2. Následně jsem je přesunul do protilehlých rohů arény. Na místě těchto objektů se budou na počátku každého kola objevovat hráči.

Dalším důležitým prvkem je komunikace s uživateli. K tomuto účelu jsem vytvořil nový objekt Canvas a v něm vytvořil další objekt, který reprezentuje textové pole. V textovém poli lze nastavit používaný font, velikost písma a barvu písma.

Obrázek 32 - Message Canvas

Zdroj: Vlastní

Text, který se objeví v tomto textovém poli, uživatel uvidí na obrazovce. Toto textové pole slouží k zobrazování výsledků herních kol a text do něj bude vkládat skript nazvaný GameManager.

GameManager je skript, který musí být vždy přítomný na scéně. Proto jsem vytvořil v hierarchii prázdný objekt, který jsem pojmenoval GameManager a u něj jsem vytvořil jako komponentu skript, který jsem nazval stejně. Tento skript prakticky kontroluje celý průběh

hry. Tento skript velmi úzce spolupracuje se skriptem SnowmanManager, který má za úkol nastavení hráčů.

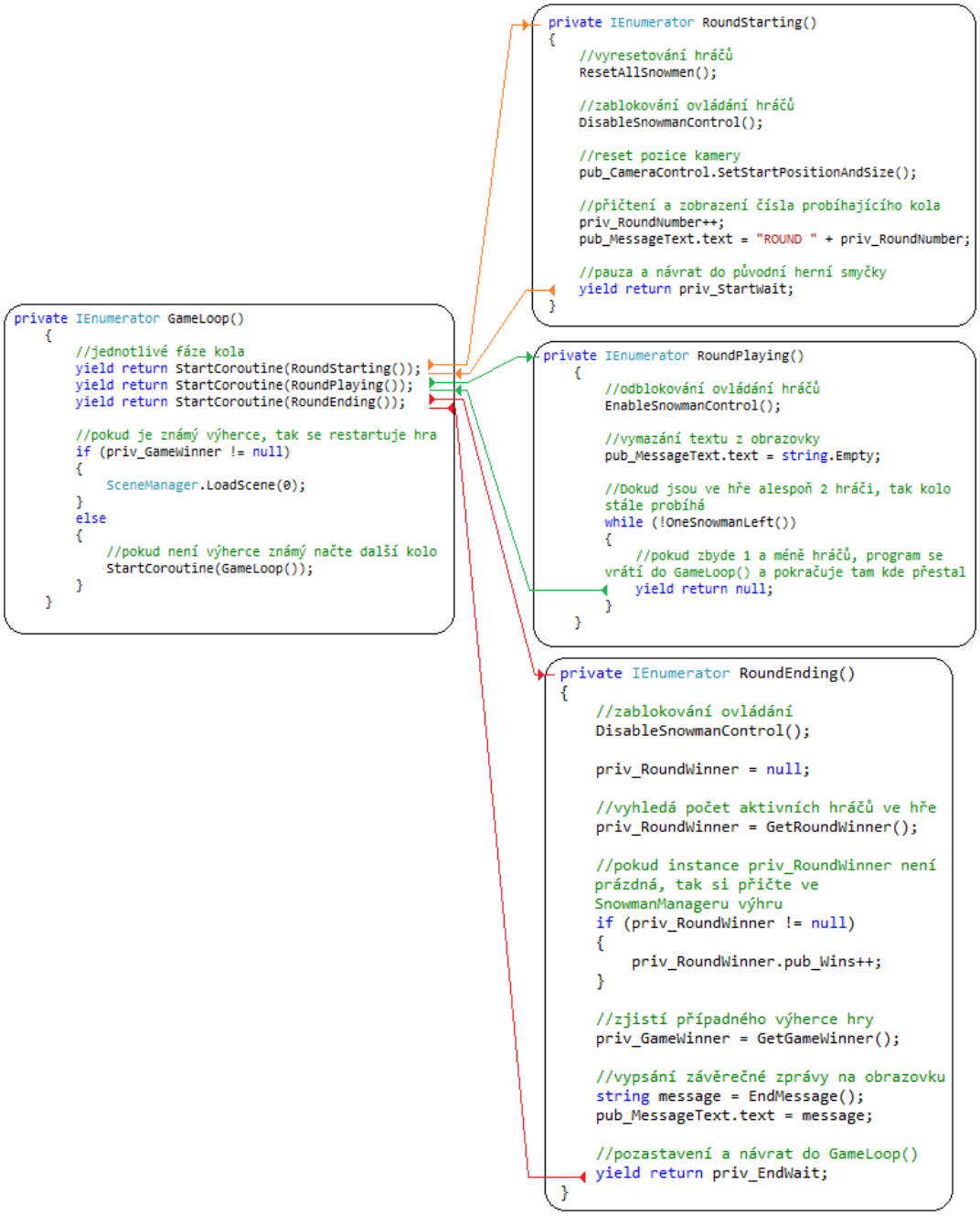
Každý hráč ve hře má svojí instanci třídy SnowmanManager, která přidělí každému hráči číslo a barvu. Instance SnowmanManager zároveň uchovává počet vyhraných kol. V poslední řadě tato třída obsahuje metody pro zablokování ovládání hráčů a funkci pro reset hráčů. Tyto funkce jsou využívány ve skriptu GameManager. Kód skriptu SnowmanManager se nachází v příloze.

GameManager při startu hry načte instance všech hráčů do hry, které jsou třídy SnowmanManager. Nastavení skriptu, který ovládá kameru souřadnice hráčů, které používá ve svých výpočtech.

Nejdůležitější část kódu GameManageru spustí funkce StartCoroutine(GameLoop), která spustí smyčku obsahující fáze kola. Coroutine pracuje s metodami třídy IEnumerator, která se od klasické metody liší používáním klíčového slova yield. Yield se užívá pro pozastavení kódu na určitý časový úsek. Následující ukázka kódu obsahuje metodu GameLoop.

```
private IEnumerator GameLoop()
{
    //jednotlivé fáze kola
    yield return StartCoroutine(RoundStarting());
    yield return StartCoroutine(RoundPlaying());
    yield return StartCoroutine(RoundEnding());

    //pokud je známý výherce, tak se restartuje hra
    if (priv_GameWinner != null)
    {
        SceneManager.LoadScene(0);
    }
    else
    {
        //pokud není výherce známý načte další kolo
        StartCoroutine(GameLoop());
    }
}
```

Tato metoda je třídy IEnumerator, která ve svém kódu využívá klíčového slova yield. Kód probíhá v této metodě normálně, ale pokud narazí na klíčové slovo yield, průběh kódu se pozastaví a čeká, dokud není splněná určitá podmínka nebo neuběhne určité zpoždění. Na následujícím obrázku (Obrázek 33) se nachází metody z kódu GameManager, které využívají klíčového slova yield. Na obrázku je znázorněna spolupráce mezi těmito metodami.



Obrázek 33 - Coroutine

Zdroj: Vlastní

5. Zhodnocení vývoje aplikace

Vývoj aplikace pomocí Unity 3D je poměrně rychlý. Práci značně usnadňuje manipulace s objekty pomocí jednoduchého přetažení, čímž se jednoduše a efektivně vytváří herní prostředí, objekty dědičnosti a přidělování vlastností herním objektům. Engine Unity obsahuje širokou škálu komponent, které umožňují vizuální nastavení bez znalosti vnitřních funkcí dané komponenty. Veškeré provedené nastavení je navíc možné si okamžitě prohlédnout v grafickém zobrazení scény přímo v editoru. Není nutné se zdržovat kompilací hry, ale je možné hru v editoru spustit a ověřit její funkčnost. Všechny tyto vlastnosti dohromady umožňují vývojáři investovat velké množství ušetřeného času do programování funkcí samotných principů dané hry.

Výsledkem praktické části je dokonale funkční hra pro dva hráče, jejíž vývoj v Unity zabral jen zlomek času v porovnání s vývojem her bez použití herního enginu. Rozdíl mezi těmito směry vývoje aplikací je i ve vývoji aplikace na různé platformy. V Unity stačí pozměnit nastavení vstupů a při dokončení aplikace je možné hru jednoduše zkompilovat například pro herní konzoli i přes to, že původně byla určena pro stolní počítač. Při vývoji aplikace „od nuly“ však toto není možné a aplikace se musí programovat prakticky celá znovu, aby mohla být spuštěna na jiné platformě. Jedinou nevýhodou může být horší optimalizace aplikace a její velikost, ale vzhledem ke stále rostoucímu výkonu počítačů a ohromné časové úspoře, kterou vývoj v Unity nabízí, jsou tyto nedostatky téměř nepodstatné.

6. Závěr

V první kapitole byla probrána historie 3D aplikací od samého počátku, kdy neexistovaly žádné nástroje pro usnadnění jejich vývoje, až po vznik komplexních herních enginů, na kterých je založeno mnoho her a aplikací. V druhé kapitole jsou rozebrány možnosti dostupných moderních vývojových prostředků Unity 3D a Unreal Engine 4.

Jádrem této práce byla praktická část, kde je popsán průběh vývoje aplikace pomocí Unity 3D. V úvodní části je probráno rozhraní, se kterým vývojář pracuje a vlastnosti Unity. Dále následuje popis vývoje samotné aplikace od modelování herního prostředí až po programování herních mechanismů. V poslední části je zhodnocen průběh vývoje v Unity a jeho výhody, které přináší.

Hlavním cílem práce bylo vytvořit aplikaci pomocí enginu Unity 3D a demonstrovat jeho vlastnosti v popisu průběhu vývoje. Tento cíl byl splněn.

7. Seznam použitých zdrojů

7.1 Literatura a internetové zdroje

1. Digitaltutors. Key 3D modeling terminology to master [online] [cit. 21. 10. 2015]. http://blog.digitaltutors.com/basic-3d-modeling-terminology/
2. Gamecareerguide. What is game engine [online] [cit. 21. 10. 2015]. http://www.gamecareerguide.com/features/529/what_is_a_game_.php
3. Techtarget. Integrated development environment [online] [cit. 21. 10. 2015]. http://searchsoftwarequality.techtarget.com/definition/integrated-development-environment
4. Unity 3D. Manual – Asset Workflow [online] [cit. 21. 10. 2015]. http://docs.unity3d.com/Manual/AssetWorkflow.html
5. Prepressure. Bitmap versus vector [online] [cit. 21. 10. 2015]. http://www.prepressure.com/library/file-formats/bitmap-versus-vector
6. Digibarn. Stories from the maze wars 30 years [online] [cit. 21. 10. 2015]. http://www.digibarn.com/history/04-VCF7-MazeWar/stories/colley.html
7. Oocities. Spasim (1974) The First First-Person-Shooter 3D Multiplayer Networked Game [online] [cit. 21. 10. 2015]. http://www.oocities.org/jim_bowery/spasim.html
8. Wikipedia. Panther (1975 video game) [online] [cit. 21. 10. 2015]. https://en.wikipedia.org/wiki/Panther_%281975_video_game%29
9. Strategywiki. Star ship [online] [cit. 21. 10. 2015]. http://strategywiki.org/wiki/Star_Ship
10. Atarihq. I-Robot [online] [cit. 21. 10. 2015]. http://www.atarihq.com/danb/irobot.shtml
11. Maximumpc. Doom to dunia a visual history of 3D game engines [online] [cit. 18. 12. 2015]. http://www.maximumpc.com/doom-to-dunia-a-visual-history-of-3d-game-engines/
12. Doomwiki. Doom rendering engine engines [online] [cit. 18. 12. 2015]. http://doomwiki.org/wiki/Doom_rendering_engine
13. Bluesnews. Ramblings in Realtime [online] [cit. 18. 12. 2015]. http://www.bluesnews.com/abrash/
14. Unity 3D. Build once deploy anywhere [online] [cit. 7. 1. 2016].

http://unity3d.com/unity/multiplatform
15. Unity 3D. Unity 5 engine overview [online] [cit. 7. 1. 2016]. http://unity3d.com/unity/engine-features
16. Unity 3D. FAQ Licencing & Activation [online] [cit. 7. 1. 2016]. https://unity3d.com/unity/faq
17. Unreal engine. Frequently asked questions [online] [cit. 7. 1. 2016]. https://www.unrealengine.com/faq
18. Unreal engine. Unreal engine features [online] [cit. 7. 1. 2016]. https://www.unrealengine.com/unreal-engine-4

8. Přílohy

8.1 Seznam obrázků

8.2 Seznam příloh

Příloha č.1 – Zdrojový kód skriptu SnowmanMovement.cs

```
using UnityEngine;
/// <summary>
/// Třída SnowmanMovement zajišťuje pohyb hráče
/// </summary>
public class SnowmanMovement : MonoBehaviour
{
    /* veřejné proměnné */
    public int pub_PlayerNumber = 1;            //číslo hráče pro získání vstupních
kláves z Input Manageru
    public float pub_Speed = 12f;               //Rychlost pohybu hráče
    public float pub_TurnSpeed = 180f;          //Rychlost otáčení hráče

    /* privátní proměnné */
    private string priv_MovementAxisName;       //kategorie v Input manageru pro
pohyb vpřed a vzad
    private string priv_TurnAxisName;           //Kategorie v Input manageru pro
otáčení vlevo a vpravo
    private Rigidbody priv_Rigidbody;           //Proměnné pro inicializaci
komponenty Rigidbody
    private float priv_MovementInputValue;      //Hodnota předávaná ze vstupních
kláves pro pohyb
    private float priv_TurnInputValue;          //Hodnota předávaná ze vstupních
kláves pro otáčení

    /* metoda Awake() je volána jako první a používá se k inicializaci potřebných
komponent */
    private void Awake()
    {
        priv_Rigidbody = GetComponent<Rigidbody>(); //Inicializace komponenty
rigidbody
    }

    /* metoda OnEnable() je zavolána jednou v momentě, kdy se objekt stane aktivním
*/
    private void OnEnable ()
    {
        priv_Rigidbody.isKinematic = false;         //vypnutím isKinematic zapneme
možnost ovládání hráče
        priv_MovementInputValue = 0f;               //nastavení úvodních vstupních
hodnot pro pohyb
        priv_TurnInputValue = 0f;                   //nastavení úvodních vstupních
hodnot pro otáčení
    }

    /* metoda OnDisable() je zavolána v případě, že objekt se stane neaktivním např.
při úmrtí*/
    private void OnDisable ()
    {
        priv_Rigidbody.isKinematic = true;          //vypnutí možnosti ovládání
hráče
    }

    /* metoda Start() je zavolána jednou až po funkci Awake(), používá se k úvodnímu
nastavení */
    private void Start()
    {
```

```
        priv_MovementAxisName = "Vertical" + pub_PlayerNumber;  //Uložení jmena
kategorie pro pohyb v Input manageru
        priv_TurnAxisName = "Horizontal" + pub_PlayerNumber;    //Uložení jmena
kategorie pro otáčení v Input manageru
    }

    /* metoda Update() je volána při načtení každého snímku, čas načtení může být
trochu rozdílný*/
    private void Update()
    {
        priv_MovementInputValue = Input.GetAxis(priv_MovementAxisName); //získání
hodnoty ze vstupních kláves pro pohyb
        priv_TurnInputValue = Input.GetAxis(priv_TurnAxisName);         //získání
hodnoty ze vstupních kláves pro otáčení
    }
    /* FixedUpdate() je načítán periodicky vždy se stejnou časovou odchylkou,
používá se např. pro manipulaci s komponenty RigidBody */
    private void FixedUpdate()
    {
        Move();     //volání funkce pro pohyb
        Turn();     //volání funkce pro otáčení
    }


    /* Move() slouží pro pohyb vpřed a vzad */
    private void Move()
    {
        Vector3 movement = transform.forward * priv_MovementInputValue * pub_Speed *
Time.deltaTime; //vypočtení vzdálenosti pohybu
        priv_Rigidbody.MovePosition(priv_Rigidbody.position + movement); //pohyb
objektu na součet aktuální vzdálenosti a vypočtené vzdálenosti
    }
    /* Turn() slouží k otáčení hráče*/
    private void Turn()
    {
        float turn = priv_TurnInputValue * pub_TurnSpeed * Time.deltaTime;
//vypočtení vzdálenosti rotace

        Quaternion turnRotation = Quaternion.Euler(0f, turn,0f);
//převedení vypočtené vzdálenosti na úhel rotace

        priv_Rigidbody.MoveRotation(priv_Rigidbody.rotation *
turnRotation);//provedení změn v rotaci
    }
}
```

Příloha č.2 – Zdrojový kód skriptu SnowmanHealth.cs

```
using UnityEngine;
using UnityEngine.UI;

public class SnowmanHealth : MonoBehaviour
{
    public float pub_StartingHealth = 100f;       //Počáteční životy
    public Slider pub_Slider;                     //Reference na ukazatel životů
    public Image pub_FillImage;                   //Reference na výplň ukazatele
životů
    public Color pub_HealtColor = Color.red;      //Barva životů
```

```
    public GameObject pub_ExplosionPrefab;        //Reference na kompletní animaci
exploze se zvukem


    private AudioSource priv_ExplosionAudio;          //Zvuk exploze
    private ParticleSystem priv_ExplosionParticles;   //Particle efects Animace
exploze
    private float priv_CurrentHealth;                 //Stávající počet životů
    private bool priv_Dead;                           //Uložení stavu živý (false),
nebo mrtvý (true)

    /* metoda Awake() je volána jako první a používá se k inicializaci potřebných
komponent */
    private void Awake()
    {
        priv_ExplosionParticles =
Instantiate(pub_ExplosionPrefab).GetComponent<ParticleSystem>();  //Inicializace
particle efects exploze
        priv_ExplosionAudio = priv_ExplosionParticles.GetComponent<AudioSource>();
//Inicializace zvuku

        priv_ExplosionParticles.gameObject.SetActive(false);    //Deaktivace
instance Particle Efektů v Hierarchy
    }

    /* metoda OnEnable() je zavolána jednou v momentě, kdy se objekt stane aktivním
*/
    private void OnEnable()
    {
        priv_CurrentHealth = pub_StartingHealth;    //Nastavení počtu životů na
výchozí hodnotu
        priv_Dead = false;

        SetHealthUI();
    }

    /* Veřejná Metoda TakeDamage odečte určitý počet životů na základě vstupního
parametru */
    public void TakeDamage(float amount)
    {
        priv_CurrentHealth -= amount;

        SetHealthUI();

        if (priv_CurrentHealth <= 0f && !priv_Dead)     //Podmínka ověřuje stav
životů, pokud zemře přehraje explozi
        {
            OnDeath();
        }
    }

    /* Nastaví hodnotu životů do grafického ukazatele */
    private void SetHealthUI()
    {
        pub_Slider.value = priv_CurrentHealth;
        pub_FillImage.color = pub_HealtColor;
    }

    /* metoda OnDeath Slouží k přehrání exploze pokud hráč zemře */
    private void OnDeath()
```

```
    {
        priv_Dead = true;
        priv_ExplosionParticles.transform.position = transform.position;
        priv_ExplosionParticles.gameObject.SetActive(true);
        priv_ExplosionParticles.Play();
        priv_ExplosionAudio.Play();
        gameObject.SetActive(false);
    }
}
```

Příloha č.3 – Zdrojový kód skriptu UIDirectionalControl.cs

```
using UnityEngine;

public class UIDirectionControl : MonoBehaviour
{
    public bool pub_UseRelativeRotation = true;  //Zapnutí relativního pozicování
ukazatele


    private Quaternion priv_RelativeRotation;    //Proměnná do které se při načtení
načtou defaultní souřadnice rotace


    private void Start()
    {
        priv_RelativeRotation = transform.parent.localRotation; //Načtení základních
souřadnic
    }


    private void Update()
    {
        if (pub_UseRelativeRotation)
            transform.rotation = priv_RelativeRotation;    //Udržování souřadnic na
defaultních hodnotách
    }
}
```

Příloha č.4 – Zdrojový kód skriptu SnowmanShooting.cs

```
using UnityEngine;
using UnityEngine.UI;

public class SnowmanShooting : MonoBehaviour
{
    public int pub_PlayerNumber = 1;            //číslo hráče pro určení ovládání v
Input manageru
    public Rigidbody pub_Snowball;              //reference na sněhovou kouli
    public Transform pub_FireTransform;         //reference na objekt FireTransform
u hráče
    public Slider pub_AimSlider;                //reference na ukazatel střely
    public AudioSource pub_ShootingAudio;       //reference na komponentu
AudioSource
    public AudioClip pub_ChargingClip;          //reference na zvuk nabíjení
```

```
    public AudioClip pub_FireClip;              //reference na zvuk výstřelu
    public float pub_MinLaunchForce = 15f;      //minimální síla nabití
    public float pub_MaxLaunchForce = 30f;      //maximální síla nabití
    public float pub_MaxChargeTime = 0.75f;     //maximální čas nabíjení


    private string priv_FireButton;             //Spojení ovládání s Input managerem
    private float priv_CurrentLaunchForce;      //aktuální síla nabití
    private float priv_ChargeSpeed;             //rychlost nabíjení
    private bool priv_Fired;                    //hráč vystřelil/nevystřelil

    /* metoda OnEnable() je zavolána jednou v momentě, kdy se objekt stane aktivním
*/
    private void OnEnable()
    {
        priv_CurrentLaunchForce = pub_MinLaunchForce;   //nastavení minimální
hodnoty nabití
        pub_AimSlider.value = pub_MinLaunchForce;       //nastavení grafické
interpretace minimálního nabití
    }

    /* metoda Start() je zavolána jednou na začátku, používá se k úvodnímu nastavení
*/
    private void Start()
    {
        priv_FireButton = "Fire" + pub_PlayerNumber;    //Spojení ovládání
konkrétního hráče s Input managerem

        priv_ChargeSpeed = (pub_MaxLaunchForce - pub_MinLaunchForce) /
pub_MaxChargeTime;   //výpočet rychlosti nabíjení
    }

    /* metoda Update() je volána při načtení každého snímku, čas načtení může být
trochu rozdílný*/
    private void Update()
    {
        pub_AimSlider.value = pub_MinLaunchForce;   //nastavení grafické
interpretace minimálního nabití

        //řetěz podmínek zajišťující nabíjení a výstřel
        if (priv_CurrentLaunchForce >= pub_MaxLaunchForce && !priv_Fired)
        {
            //výstřel pokud aktuální hodnota nabití přesáhne maximum
            priv_CurrentLaunchForce = pub_MaxLaunchForce;
            Fire();
        }
        else if (Input.GetButtonDown(priv_FireButton))
        {
            //stlačení tlačítka výstřelu
            priv_Fired = false;
            priv_CurrentLaunchForce = pub_MinLaunchForce;

            //přehrání zvuku nabíjení
            pub_ShootingAudio.clip = pub_ChargingClip;
            pub_ShootingAudio.Play();
        }
        else if (Input.GetButton(priv_FireButton) && !priv_Fired)
        {
            //držení tlačítka výstřelu a nabíjení
            priv_CurrentLaunchForce += priv_ChargeSpeed * Time.deltaTime;
```

```
            pub_AimSlider.value = priv_CurrentLaunchForce;
        }
        else if (Input.GetButtonUp(priv_FireButton) && !priv_Fired)
        {
            //uvolnění tlačítka výstřelu a výstřel
            Fire();
        }
    }

    /* Metoda Fire() načte objekt sněhové koule z assetů a vystřelí jej */
    private void Fire()
    {
        priv_Fired = true;

        //načtení sněhové koule do instance třídy Rigidbody
        Rigidbody snowballInstance = Instantiate(pub_Snowball,
pub_FireTransform.position, pub_FireTransform.rotation) as Rigidbody;

        //Výstřel pomocí pozičních hodnot objektu FireTransform
        snowballInstance.velocity = priv_CurrentLaunchForce *
pub_FireTransform.forward;

        //přehrání zvuku výstřelu
        pub_ShootingAudio.clip = pub_FireClip;
        pub_ShootingAudio.Play();

        //vyresetování hodnoty nabití
        priv_CurrentLaunchForce = pub_MinLaunchForce;
    }
}
```

Příloha č.5 – Zdrojový kód skriptu SnowballExplosion.cs

```
using UnityEngine;

public class SnowballExplosion : MonoBehaviour
{
    public LayerMask pub_SnowmanMask;               //Nastavení vrstvy ve které
budou objekty ovlivněny - Layer: Players
    public ParticleSystem pub_ExplosionParticles;   //Efekt rozpadu sněhové koule
při kolizi
    public AudioSource pub_ExplosionAudio;          //Zvuk sněhové koule při kolizi
    public float pub_MaxDamage = 100f;              //Máximální možné poškození
    public float pub_ExplosionForce = 1000f;        //Síla exploze projektilu
    public float pub_MaxLifeTime = 2f;              //Maximální délka životnosti
projektilu
    public float pub_ExplosionRadius = 5f;          //Velikost radiu kde může být
hráč poškozen

    /* metoda Start() je zavolána až po funkci Awake(), používá se k úvodnímu
nastavení */
    private void Start()
    {
        Destroy(gameObject, pub_MaxLifeTime);       //Tato funkce zničí objekt po
uplnutí stanovené doby
    }
```

```
    /* metoda OnTriggerEnter je spuštěná vždy když dojte ke kolizi objektu
(nastavení: Is Trigger) */
    private void OnTriggerEnter(Collider other)
    {
        //Metoda OverlapSphere načte do pole colliders veškeré collidery vrstvy
PLayers, které se nacházejí v určeném rádiu okolo pozice projektilu
        Collider[] colliders = Physics.OverlapSphere(transform.position,
pub_ExplosionRadius, pub_SnowmanMask);

        //tento cyklus prochází pole colliders
        for (int i = 0; i < colliders.Length; i++)
        {
            //Inicializace komponenty RigidBody z položky pole colliders
            Rigidbody targetRigidbody = colliders[i].GetComponent<Rigidbody>();

            //Pokud objekt neobsahuje komponentu rigidbody, cyklus for přeskočí k
dalšímu cyklu
            if (!targetRigidbody)
                continue;

            //metoda AddExplosionForce odtlačí cílový objekt rigidBody směrem od
pozice projektilu v určeném radiu
            targetRigidbody.AddExplosionForce(pub_ExplosionForce,
transform.position, pub_ExplosionRadius);

            //Inicializace skriptu SnowmanHealth z cílového objektu
            SnowmanHealth targetHealth =
targetRigidbody.GetComponent<SnowmanHealth>();

            //Pokud objekt neobsahuje komponentu SnowmanHealth, cyklus for přeskočí
k dalšímu cyklu
            if (!targetHealth)
                continue;

            //Zavolání metody TakeDamage() umístěné ve skriptu SnowmanHealth
            targetHealth.TakeDamage(CalculateDamage(targetRigidbody.position));
        }

        //přehrání partikulárních efektů exploze a zvuku a jejich ukončení
        pub_ExplosionParticles.transform.parent = null;
        pub_ExplosionParticles.Play();
        pub_ExplosionAudio.Play();
        Destroy(pub_ExplosionParticles.gameObject, pub_ExplosionParticles.duration);

        //zničení objektu sněhové koule
        Destroy(gameObject);

    }

    /* Metoda vypočítávající poškození ze vzdálenosti projektilu a cíle */
    private float CalculateDamage(Vector3 targetPosition)
    {
        //Vypočtení vektoru mezi projektilem a cílem
        Vector3 distanceFromTarget = targetPosition - transform.position;

        //Vypočtení relativní hodnoty vzdálenosti
        float distance = (pub_ExplosionRadius - distanceFromTarget.magnitude) /
pub_ExplosionRadius;

        //výpočet poškození
```

```
        float damage = distance * pub_MaxDamage;

        //Ujištění že minimální poškození bude vždy 0
        damage = Mathf.Max(0f, damage);

        return damage;
    }
}
```

Příloha č.6 – Zdrojový kód skriptu CameraControl.cs

```
using UnityEngine;

public class CameraControl : MonoBehaviour
{
    public float pub_DampTime = 0.2f;                   //malé časové spoždění pro
pohyb kamery
    public float pub_ScreenEdgeBuffer = 4f;             //prostor mezi hráčem a
horní/dolní hranou obrazovky
    public float pub_MinSize = 8f;                      //minimální velikost kamery
(maximální možné přiblížení)
    [HideInInspector] public Transform[] pub_Targets;   //pole pozic všech hráčů na
scéně


    private Camera priv_Camera;                         //reference na komponentu
camera
    private float priv_ZoomSpeed;                       //rychlost přibližování
kamery
    private Vector3 priv_MoveVelocity;                  //reference na proměnou
třídy Vector3 se kterou pracuje metoda SmoothDamp
    private Vector3 priv_TargetPosition;               //Pozice na kterou se kamera
snaží dostat

    /* metoda Awake() je volána jako první a používá se k inicializaci potřebných
komponent */
    private void Awake()
    {
        //inicializace komponenty Camera
        priv_Camera = GetComponentInChildren<Camera>();
    }

    /* FixedUpdate() je načítán periodicky vždy se stejnou časovou odchylkou,
používá se např. pro manipulaci s komponenty RigidBody */
    private void FixedUpdate()
    {
        Move();
        Zoom();
    }

    /* Metoda Move() přemístí kameru na určenou pozici */
    private void Move()
    {
        //volání funkce, která najde středové souřadnice mezi hráči
        CameraPosition();

        //plynulé přesunutí kamery na cílovou pozici
```

```
        transform.position = Vector3.SmoothDamp(transform.position,
priv_TargetPosition, ref priv_MoveVelocity, pub_DampTime);
    }

    /* Metoda, která vyhledá středovou pozici mezi hráči a souřadnice uloží do
priv_DesiredPosition */
    private void CameraPosition()
    {
        Vector3 avPos = new Vector3();
        int numTargets = 0;

        //procházení pole ve kterém jsou uloženy souřadnice hráčů
        for (int i = 0; i < pub_Targets.Length; i++)
        {
            //Pokud je objekt aktivní provedou se výpočty
            if (pub_Targets[i].gameObject.activeSelf) {

            //Sčítání souřadnic všech cílových objektů
            avPos += pub_Targets[i].position;

            //počet cílů
            numTargets++;
            }
        }

        //výpočet průměrné pozice
        if (numTargets > 0)
            avPos /= numTargets;

        //udržení hodnot osy Y na základní hodnotě
        avPos.y = transform.position.y;

        //naplnění proměnné cílovými souřadnicemi
        priv_TargetPosition = avPos;
    }

    /* Zoom() zajišťuje přiblížení kamery */
    private void Zoom()
    {
        //volání metody, která počítá doporučené přiblížení kamery a uložení hodnoty
do proměnné
        float size = CameraSize();

        //přiblížení kamery
        priv_Camera.orthographicSize =
Mathf.SmoothDamp(priv_Camera.orthographicSize, size, ref priv_ZoomSpeed,
pub_DampTime);
    }

    /* metoda která počítá přiblížení kamery*/
    private float CameraSize()
    {
        //Převedení globálních souřadnic cílové pozice do lokálních hodnot kamery
(výhled)
        Vector3 LocalTargetPos =
transform.InverseTransformPoint(priv_TargetPosition);

        float size = 0f;

        //procházení pole ve kterém jsou uloženy souřadnice hráčů
```

```
        for (int i = 0; i < pub_Targets.Length; i++)
        {
            //Pokud je objekt aktivní, provedou se výpočty
            if (pub_Targets[i].gameObject.activeSelf) {

            //Převedení pozic hráčů z globálních hodnot do lokálních hodnot kamery
            Vector3 LocalPLayerPos =
transform.InverseTransformPoint(pub_Targets[i].position);

            //Vypočtení vzdálenosti hráče od středu kamery
            Vector3 CenterPosToTarget = LocalPLayerPos - LocalTargetPos;

            //Porovnání hodnot size a vzdálenosti hráče od středu kamery na
vertikální ose a nahrání větší hodnoty do proměnné size
            size = Mathf.Max (size, Mathf.Abs (CenterPosToTarget.y));

            //Porovnání hodnot size a vzdálenosti hráče od středu kamery na
horizontální ose a nahrání větší hodnoty do proměnné size
            size = Mathf.Max (size, Mathf.Abs (CenterPosToTarget.x) /
priv_Camera.aspect);
            }
        }

        //zvětšení odsazení okrajů kamery od hráče
        size += pub_ScreenEdgeBuffer;

        //Ujištění že velikost kamery nebude nikdy větší než nastavené minimum
        size = Mathf.Max(size, pub_MinSize);

        return size;
    }

    /* Veřejná metoda pro počáteční nastavení kamery */
    public void SetStartPositionAndSize()
    {
        CameraPosition();

        transform.position = priv_TargetPosition;

        priv_Camera.orthographicSize = CameraSize();
    }
}
```

Příloha č.7 – Zdrojový kód skriptu SnowmanManager.cs

```
using System;
using UnityEngine;

[Serializable]
public class SnowmanManager
{
    public Color pub_PlayerColor;                          //Barva hráče
    public Transform pub_SpawnPoint;                       //souřadnice Spawnpointu
    [HideInInspector] public int pub_PlayerNumber;         //číslo hráče
    [HideInInspector] public string pub_ColoredPlayerText; //obarvený text
PLAYER+číslo
```

```
    [HideInInspector] public GameObject pub_Instance;       //reference na objekt
hráče
    [HideInInspector] public int pub_Wins;                  //počet vyhraných kol


    private SnowmanMovement priv_Movement;          //reference na skript
SnowmanMovement
    private SnowmanShooting priv_Shooting;          //reference na skript
SnowmanShooting
    private SnowmanHealth priv_Health;              //reference na skript
SnowmanHealth
    private GameObject priv_CanvasGameObject;       //reference na objekt obsahující
životy a ukazatel nabití u hráče


    public void Setup()
    {
        //Inicializace potřebných komponent
        priv_Movement = pub_Instance.GetComponent<SnowmanMovement>();
        priv_Shooting = pub_Instance.GetComponent<SnowmanShooting>();
        priv_Health= pub_Instance.GetComponent<SnowmanHealth>();
        priv_CanvasGameObject =
pub_Instance.GetComponentInChildren<Canvas>().gameObject;

        //Přidělení čísla hráče komponentám
        priv_Movement.pub_PlayerNumber = pub_PlayerNumber;
        priv_Shooting.pub_PlayerNumber = pub_PlayerNumber;

        //Obarvení životů hráče jeho barvou
        priv_Health.pub_HealtColor = pub_PlayerColor;

        //Obarvení textu Player+číslo
        pub_ColoredPlayerText = "<color=#" +
ColorUtility.ToHtmlStringRGB(pub_PlayerColor) + ">PLAYER " + pub_PlayerNumber +
"</color>";

    }

    //Funkce pro zablokování ovládání
    public void DisableControl()
    {
        priv_Movement.enabled = false;
        priv_Shooting.enabled = false;

        priv_CanvasGameObject.SetActive(false);
    }

    //Funkce pro odblokování ovládání
    public void EnableControl()
    {
        priv_Movement.enabled = true;
        priv_Shooting.enabled = true;

        priv_CanvasGameObject.SetActive(true);
    }

    //Reset hráče na pozici jeho spawnu
    public void Reset()
    {
        pub_Instance.transform.position = pub_SpawnPoint.position;
```

```
        pub_Instance.transform.rotation = pub_SpawnPoint.rotation;

        pub_Instance.SetActive(false);
        pub_Instance.SetActive(true);
    }
}
```

Příloha č.8 – Zdrojový kód skriptu GameManager.cs

```
using UnityEngine;
using System.Collections;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class GameManager : MonoBehaviour
{
    public int pub_NumRoundsToWin = 5;      //počet vítězných kol potřebných k
celkové výhře
    public float pub_StartDelay = 3f;       //Délka pauzy před spuštěním kola
    public float pub_EndDelay = 3f;         //Délka pauzy na konci kola
    public CameraControl pub_CameraControl; //Reference na skript ovládající kameru
    public Text pub_MessageText;            //Reference na Testové pole v
MessageCanvas
    public GameObject pub_SnowmanPrefab;    //Reference na objekt hráče uložený ve
složce s assety
    public SnowmanManager[] pub_Snowmen;    //Reference na skript SnowmanManager,
který provádí nastavení hráčů


    private int priv_RoundNumber;               //číslo aktuálně probíhajícího kola
    private WaitForSeconds priv_StartWait;      //Pauza před spuštěním kola
    private WaitForSeconds priv_EndWait;        //Pauza na konci kola
    private SnowmanManager priv_RoundWinner;    //hráč který vyhrál kolo
    private SnowmanManager priv_GameWinner;     //hráč který vyhrál hru


    private void Start()
    {
        //inicializace objektů časové odezvy
        priv_StartWait = new WaitForSeconds(pub_StartDelay);
        priv_EndWait = new WaitForSeconds(pub_EndDelay);

        //načtení hráčů do hry
        SpawnAllSnowmen();

        //načtení hráčů do skriptu ovládajícího kameru
        SetCameraTargets();

        //začátek herní smyčky
        StartCoroutine(GameLoop());
    }

    /* metoda která do hry načte všechny hráče */
    private void SpawnAllSnowmen()
    {
        //procházení pole hráčů
        for (int i = 0; i < pub_Snowmen.Length; i++)
```

```
        {
            //Instancializace objektu hráče ze složky assetů a nastaví hráči pozici,
kde se objeví
            pub_Snowmen[i].pub_Instance = Instantiate(pub_SnowmanPrefab,
pub_Snowmen[i].pub_SpawnPoint.position, pub_Snowmen[i].pub_SpawnPoint.rotation) as
GameObject;
            //Přidělení čísla hráči
            pub_Snowmen[i].pub_PlayerNumber = i + 1;
            //Nastavení atributů hráče ve SnowmanManageru
            pub_Snowmen[i].Setup();
        }
    }

    /* Načte objekty hráčů do skriptu, který ovládá kameru */
    private void SetCameraTargets()
    {
        //vytvoření pole pozic, kde budou později uloženy pozice hráčů
        Transform[] targets = new Transform[pub_Snowmen.Length];

        //procházení pole pozic hráčů
        for (int i = 0; i < targets.Length; i++)
        {
            //uložení pozic hráčů do pole
            targets[i] = pub_Snowmen[i].pub_Instance.transform;
        }
        //předání skriptu CameraControl pole souřadnic
        pub_CameraControl.pub_Targets = targets;
    }

    /* herní smyčka */
    private IEnumerator GameLoop()
    {
        //jednotlivé fáze kola
        yield return StartCoroutine(RoundStarting());
        yield return StartCoroutine(RoundPlaying());
        yield return StartCoroutine(RoundEnding());

        //pokud je známý výherce, tak se restartuje hra
        if (priv_GameWinner != null)
        {
            SceneManager.LoadScene(0);
        }
        else
        {
            //pokud není výherce známý načte další kolo
            StartCoroutine(GameLoop());
        }
    }

    /* počáteční fáze kola */
    private IEnumerator RoundStarting()
    {
        //vyresetování hráčů
        ResetAllSnowmen();

        //zablokování ovládání hráčů
        DisableSnowmanControl();

        //reset pozice kamery
        pub_CameraControl.SetStartPositionAndSize();
```

```
        //přičtení a zobrazení čísla probíhajícího kola
        priv_RoundNumber++;
        pub_MessageText.text = "ROUND " + priv_RoundNumber;

        //pauza a návrat do původní herní smyčky
        yield return priv_StartWait;
    }

    /* průběh kola */
    private IEnumerator RoundPlaying()
    {
        //odblokování ovládání hráčů
        EnableSnowmanControl();

        //vymazání textu z obrazovky
        pub_MessageText.text = string.Empty;

        //Dokud jsou ve hře alespoň 2 hráči, tak kolo stále probíhá
        while (!OneSnowmanLeft())
        {
            //pokud zbyde 1 a méně hráčů, program se vrátí do GameLoop() a pokračuje
tam kde přestal
            yield return null;
        }
    }

    /*konečná fáze kola*/
    private IEnumerator RoundEnding()
    {
        //zablokování ovládání
        DisableSnowmanControl();

        priv_RoundWinner = null;

        //vyhledá počet aktivních hráčů ve hře
        priv_RoundWinner = GetRoundWinner();

        //pokud instance priv_RoundWinner není prázdná, tak si přičte ve
SnowmanManageru výhru
        if (priv_RoundWinner != null)
        {
            priv_RoundWinner.pub_Wins++;
        }

        //zjistí případného výherce hry
        priv_GameWinner = GetGameWinner();

        //vypsání závěrečné zprávy na obrazovku
        string message = EndMessage();
        pub_MessageText.text = message;

        //pozastavení a návrat do GameLoop()
        yield return priv_EndWait;
    }

    /* Zjištění počtu aktivních hráčů ve hře */
    private bool OneSnowmanLeft()
    {
        int numSnowmansLeft = 0;
```

```
            for (int i = 0; i < pub_Snowmen.Length; i++)
            {
                if (pub_Snowmen[i].pub_Instance.activeSelf)
                    numSnowmansLeft++;
            }

            //pokud se ve hře nachází 1 a méně vrátí true, jinak false
            return numSnowmansLeft <= 1;
        }

        /* zjistí který hráč vyhrál kolo */
        private SnowmanManager GetRoundWinner()
        {
            //pokud zjistí aktivního hráče ve hře, vrátí ho jako objekt Snowmanager
            for (int i = 0; i < pub_Snowmen.Length; i++)
            {
                if (pub_Snowmen[i].pub_Instance.activeSelf)
                    return pub_Snowmen[i];
            }

            return null;
        }

        /* zjistí výherce hry */
        private SnowmanManager GetGameWinner()
        {
            //propočítává počet výher jednotlivých hráčů
            for (int i = 0; i < pub_Snowmen.Length; i++)
            {
                //porovnává počet výher hráče a nastavené množství
                if (pub_Snowmen[i].pub_Wins == pub_NumRoundsToWin)
                    return pub_Snowmen[i];
            }

            return null;
        }

        /* Zpráva při konci kola */
        private string EndMessage()
        {
            string message = "DRAW!";

            //vypíše výherce kola
            if (priv_RoundWinner != null)
                message = priv_RoundWinner.pub_ColoredPlayerText + " WINS THE ROUND!";

            message += "\n\n\n\n";

            //vypisuje počet výher jednotlivých hráčů
            for (int i = 0; i < pub_Snowmen.Length; i++)
            {
                message += pub_Snowmen[i].pub_ColoredPlayerText + ": " +
pub_Snowmen[i].pub_Wins + " WINS\n";
            }

            //vypíše který hráč vyhrál hru
            if (priv_GameWinner != null)
                message = priv_GameWinner.pub_ColoredPlayerText + " WINS THE GAME!";
```

```
        return message;
    }

    //vyresetování hráčů
    private void ResetAllSnowmen()
    {
        for (int i = 0; i < pub_Snowmen.Length; i++)
        {
            pub_Snowmen[i].Reset();
        }
    }

    //povolení ovládání hráčů
    private void EnableSnowmanControl()
    {
        for (int i = 0; i < pub_Snowmen.Length; i++)
        {
            pub_Snowmen[i].EnableControl();
        }
    }

    //zablokování ovládání hráčů
    private void DisableSnowmanControl()
    {
        for (int i = 0; i < pub_Snowmen.Length; i++)
        {
            pub_Snowmen[i].DisableControl();
        }
    }
}
```