Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Vývoj Android aplikace Malý princ

Bc. Michal Smékal

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Michal Smékal

Systémové inženýrství a informatika
Informatika

Název práce

Vývoj Android aplikace Malý princ

Název anglicky

Development of Android application The Little Prince

Cíle práce

Cílem teoretické části práce je představení herního enginu Unity a jeho porování s jinými, běžně komerčně využívanými herními enginy. Dále pak nastudování a představení základních postupů vývoje dvourozměrných počítačových her a her pro Android OS v prostředí tohoto multiplatformního herního enginu s využitím programovacího jazyka C# a závěrem průzkum trhu současných mobilních a počítačových her s tématikou Malého prince za účelem originality praktické části.

Cílem praktické části práce je samotný vývoj unikátní vzdělávací hry Malý princ pro Android OS s využitím základních funkcí enginu Unity, jako například fyziky, animace, kolizí, C# skriptů a jiných. Hra má za cíl sledovat celý děj původní předlohy Malý princ od Antoine de Saint-Exupéryho. Po skončení jednotlivých fází hry budou moci hráči absolvovat testovací kvízy ze základních otázek týkajících se děje, které budou v závěru hry vyhodnoceny.

Metodika

Prvním krokem tvorby této práce je nastudování historie, vlastností, výhod a nevýhod z odborné literatury zabývající se obecným představením Unity Engine. Následujícím krokem je pak nastudování odborné literatury, zabývající se konkrétně vývojem dvourozměrných her a vývojem her pro operační systém android. Následuje nastudování knihy Malý princ z důvodu kompletní znalosti titulu a dodržení dějové linie hry. Posledním krokem teoretické části je průzkum pomocí nejznámějších portálů pro distribuci her, jako například Steam.com nebo Itch.io, za účelem nalezení všech titulů s tématikou malého prince, jejich otestování a návrh modelu hry odlišující se od testovaných titulů.

Tvorba praktické části je pak rozdělena do několika navzájem se prolínajících částí: tvorba základní testovací grafiky v programu Moho 12 a Gimp, tvorba uživatelského rozhraní hry, včetně ovládání, inventáře, systému pro zaznamenávání postupu ve hře, dialogového systému, úkolového systému a metody pro ukládání postupu ve hře, třetí částí bude vytvoření finální grafiky potřebné pro vyobrazení kompletního děje, čtvrtou částí praktické části je vytvoření kompletního děje pomocí nově vytvořených součástí rozhraní hry a finální částí je pak vytvoření souboru ve formátu .apk obsahujícího kompletní hru, a jeho testování.

Doporučený rozsah práce

60-80 stran

Klíčová slova

Mobilní aplikace, Android, Malý princ, Unity Engine, C#

Doporučené zdroje informací

FINNEGAN, Thomas. Learning Unity Android Game Development. 1st ed. Birmingham: Packt Publishing Ltd., 2015. ISBN 978-1-78439-469-1

HALPERN, Jared. Developing 2D Games with Unity: Independent Game Programming with C#. 1. New York: Apress, 2019. ISBN 978-1-4842-3771-7.

MURRAY, Jeff W. C# game programming cookbook for Unity 3D. Boca Raton: CRC Press, [2014]. ISBN 978-1-4665-8140-1.

THORN, Alan. Learn Unity for 2D Game Development. 1. New York: Apress, 2013. ISBN 978-1-4302-6230-5.

Předběžný termín obhajoby

2020/21 LS – PEF

Vedoucí práce

Ing. Dana Vynikarová, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 19. 11. 2020

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 19. 11. 2020

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 20. 03. 2021

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Vývoj Android aplikace Malý princ" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 31.3.2021 _____

Poděkování

Rád bych touto cestou poděkoval Ing. Daně Vynikarové, Ph.D., vedoucí mé diplomové práce za její aktivní přístup, cenné rady, a především velkou trpělivost.

Vývoj Android aplikace Malý princ

Abstrakt

Tato diplomová práce je zaměřena na vývoj vzdělávací aplikace Malý Princ, určené pro mobilní zařízení s operačním systémem Android. Aplikace je vytvořena pomocí multiplatformního herního enginu Unity 3D, který využívá objektově orientovaný programovací jazyk C#. Literární rešerše se zabývá seznámením čtenáře s pravidly tvorby vzdělávacích aplikací a představuje základní pojmy, funkce a uživatelské rozhraní herního enginu Unity 3D.

Praktická část této diplomové práce se skládá z analýzy, návrhu, podrobné implementace a testování aplikace Malý princ. Výsledkem analýzy, plynoucí z průzkumu trhu porovnávaným s požadavky na aplikaci, je potvrzení originality analyzovaného díla. Návrh aplikace se skládá z navržení herní mechaniky a rozvržení uživatelského rozhraní. Na základě návrhu je aplikace implementována pomocí herního enginu Unity3D a programovacího jazyka C#.

Celkovým výsledkem praktické části je předváděcí verze vzdělávací aplikace Malý princ. Aplikace se skládá ze čtyř unikátních herních planet, na kterých hráč potkává čtyři postavy z knižního předlohy: pilota, růži, krále a lampáře. Aplikace podporována operačním systémem Android verze 4.1 a vyšší.

Klíčová slova: mobilní aplikace, vzdělávací hra, Android, Malý princ, herní engine, Unity 3D, C#, OOP, software

Development of Android application The Little Prince

Abstract

This diploma thesis is focused on development of educational application The Little Prince, running on mobile devices with operating system Android. This application is created with multiplatform game engine Unity 3D, which is using object-oriented programing language C#. The literal research familiarizes reader with the rules of making educational games and presenting basic terms, functions, and user interface
of game engine Unity 3D.

The practical part of this diploma thesis consists of analysis, design, build, and testing of application The Little Prince. Result of analysis, comparison between market research and application requirements, claims the application is going to be an original solution. The design of the application consists of game mechanics and user interface design. The building of application, based on design, is made in Unity 3D, using programming language C#.

The result of the practical part is demo version of the application The Little Prince. Application plot consists of four unique game planets, where player can meet four different characters from the book: the pilot, the rose, the king, and the lamplighter. Application is supported by operating system Android, version 4.1 or higher.

Keywords: mobile application, educational game, Android, The Little Prince, game engine, Unity 3D, C#, OOP, software

Obsah

1 Úvod **12**

2 Cíl práce a metodika **13**

2.1 Cíl práce 13

2.2 Metodika 13

3 Teoretická východiska **15**

3.1 Role her ve vzdělávání 15

3.1.1 Didaktické aplikace 15

3.1.2 Kategorizace didaktických aplikací 16

3.1.2.1 Míra interaktivity¨ 16

3.1.2.2 Úroveň vzdělávání 16

3.1.2.3 Poskytování zpětné vazby 16

3.1.2.4 Organizovanost vzdělávání 16

3.1.2.5 Počet uživatelů 17

3.1.2.6 Počet didaktických funkcí 17

3.1.2.7 Zaměření aplikace 17

3.1.3 Potřebné vlastnosti didaktické aplikace 17

3.2 Literární dílo Malý princ 18

3.2.1 Děj Malého prince 18

3.3 Herní engine 19

3.3.1 Základní pojmy 19

3.3.1.1 Asset 19

3.3.1.2 Build 19

3.3.1.3 GUI 19

3.3.1.4 Kolize 20

3.3.1.5 Real-Time Render 20

3.3.1.6 SDK 20

3.3.1.7 Sprite 20

3.3.2 Příklady herních enginů 21

3.3.2.1 Unreal Engine 21

3.3.2.2 Unity3D 22

3.4 Funkce a uživatelské rozhraní Unity3D Engine 23

3.4.1 Uživatelské rozhraní 23

3.4.1.1 Project 24

3.4.1.2 Hierarchy 24

3.4.1.3 Inspector 25

3.4.1.4 Scene 25

3.4.1.5 Game 26

3.4.1.6 Console 26

3.4.1.7 Animator a Animation 27

3.4.1.8 Asset store 27

3.4.2 Vybrané funkce Unity 3D ... 28
3.4.2.1 Transform ... 28
3.4.2.2 Sprite Renderer ... 28
3.4.2.3 Rigid Body ... 29
3.4.2.4 Collider ... 29
3.4.2.5 Canvas ... 29
3.4.2.6 Button ... 30

4 Vlastní práce ... 31
4.1 Analýza ... 31
4.1.1 Požadavky na aplikaci ... 31
4.1.1.1 Požadavky na didaktickou aplikaci ... 31
4.1.1.2 Požadavky z hlediska herní mechaniky ... 32
4.1.2 Dostupné hry s tématikou Malý princ ... 33
4.1.2.1 Steam ... 33
4.1.2.2 Google Play ... 33
4.1.2.3 Itch.io ... 34
4.1.2.4 Zhodnocení existujících titulů ... 35
4.1.3 Závěr analýzy ... 36
4.2 Návrh ... 36
4.2.1 Herní mechanika ... 36
4.2.1.1 Herní postava ... 36
4.2.1.2 Scény a přechody mezi nimi ... 37
4.2.1.3 Předměty a interaktivita ... 38
4.2.1.4 Dialogy ... 39
4.3 Implementace ... 40
4.3.1 Založení projektu ... 40
4.3.2 Postava hráče ... 40
4.3.2.1 Vzhled a animace postavy ... 41
4.3.2.2 Vytvoření herního objektu postavy ... 42
4.3.2.3 Animace postavy ... 42
4.3.2.4 Ovládání postavy ... 43
4.3.2.5 Skok ... 44
4.3.3 Herní prostředí ... 46
4.3.3.1 Planeta ... 46
4.3.3.2 Rotace herního světa ... 47
4.3.4 Uživatelské rozhraní ... 48
4.3.4.1 Rozhraní pro pohyb ... 49
4.3.4.2 Rozhraní inventáře ... 50
4.3.4.3 Rozhraní dialogů ... 52
4.3.4.4 Rozhraní kvízů ... 53
4.3.5 Načítání scén ... 54
4.3.5.1 Třída pro načítání ... 54
4.3.6 Inventář a předměty ... 56
4.3.6.1 Předměty ... 56

4.3.6.2 Dokončení inventáře ... 58
4.3.6.3 Uložení předmětu do inventáře ... 59
4.3.7 Dialogy ... 61
4.3.7.1 Základní dialogová třída ... 62
4.3.7.2 Vytvoření a vyvolání dialogu ... 62
4.3.7.3 Začátek dialogu ... 64
4.3.7.4 Průběh dialogu ... 66
4.3.8 Interakce ... 68
4.3.8.1 Vyvolání metody k použití předmětu ... 68
4.3.8.2 Interakce mezi dvěma předměty ... 69
4.3.8.3 Interakce mezi předmětem a herním objektem ... 71
4.3.9 Animace interakcí ... 73
4.3.9.1 Vytvoření animací interakce ... 73
4.3.9.2 Spuštění animací interakce ... 73
4.3.10 Kvíz ... 74
4.3.10.1 Vytvoření otázek a odpovědí ... 75
4.3.10.2 Třída pro řízení kvízů ... 75
4.3.11 Ukládání a načítání dat ... 78
4.3.11.1 Ukládání dat ... 79
4.3.11.2 Načtení dat ... 81
4.4 Tvorba aplikace a testování ... 82
4.4.1 Testování v prostředí Unity ... 82
4.4.2 Tvorba aplikace ... 83
4.4.3 Testování na mobilních zařízeních ... 85

5 Závěr ... 86

6 Seznam použitých zdrojů ... 87

7 Přílohy ... 90

Seznam obrázků

Obrázek 1 – Výchozí uživatelské rozhraní Unity3D (vlastní tvorba) 23
Obrázek 2 - Okno *Project* (vlastní tvorba) 24
Obrázek 3 - Panel *Hierarchy* (vlastní tvorba) 25
Obrázek 4 - Rozdíl mezi *Scene* (nahoře) a *Game* (dole) (vlastní tvorba) 26
Obrázek 5 - Okna *Animation* (nahoře) a *Animator* (dole) (vlastní tvorba) 27
Obrázek 6 - Ukázka ze hry *The Little Prince VR* (vlastní tvorba) 33
Obrázek 7 - Ukázka ze hry *Little Prince Jump* (vlastní tvorba) 34
Obrázek 8 - Ukázka ze hry *The little prince (vlastní tvorba)* 35
Obrázek 9 - Znázornění možných přístupů k ovládání postavy (vlastní tvorba) 37
Obrázek 10 - Znázornění přechodů mezi scénami (vlastní tvorba) 38
Obrázek 11 - Návrh tříd *Items* a *Inventory* (vlastní tvorba) 39
Obrázek 12 - Animace pohybu hlavní postavy (vlastní tvorba) 41
Obrázek 13 - Časová osa animace pohybu (vlastní tvorba) 42
Obrázek 14 - Přepínání mezi směry pohybu (vlastní tvorba) 43
Obrázek 15 - Herní objekty planety (vlastní tvorba) 47
Obrázek 16 - Rozhraní pro ovládání postavy (vlastní tvorba) 50
Obrázek 17 - Rozhraní inventáře (vlastní tvorba) 50
Obrázek 18 - Herní objekty dialogového rozhraní (vlastní tvorba) 52
Obrázek 19 - Rozhraní pro testovací kvízy (vlastní tvorba) 54
Obrázek 20 - Proměnné předmětu *Konvice* (vlastní tvorba) 58
Obrázek 21 - Proměnné třídy *Inventory Control* (vlastní tvorba) 59
Obrázek 22 - Nastavení události *OnClick* (vlastní tvorba) 68
Obrázek 23 - Časová osa animace pro použití konve (vlastní tvorba) 73
Obrázek 24 - Testování v konzoli (vlastní tvorba) 83
Obrázek 25 - Závěrečné nastavení projektu (vlastní tvorba) 84

Seznam tabulek

Tabulka 1 - Požadavky na herní mechaniku aplikace 32
Tabulka 2 - Srovnání dostupných aplikací s tématikou Malého prince 35
Tabulka 3 - Seznam testovacích zařízení 85

Seznam použitých zkratek

2D – Two-dimensional (dvourozměrné)

3D – Three-dimensional (trojrozměrné)

API – Application programing interface

GUI – Graphical user interface

SDK – Software development kit

VR – Virtual reality

1 Úvod

Vzdělávaní žáků základních a středních škol tradičním způsobem je v současné moderní době problematičtější, neboť pozornost žáků se čím dál častěji upíná k moderním digitálním technologiím. Tvorba vzdělávacích herních aplikací, zaměřených na témata základního a středoškolského učiva tak může pomoci vylepšit, či připomenout, znalosti žáků základních a středních škol k daným tématům.

Vývoj herních aplikací, ať už určených na počítače, či mobilní zařízení, byl vždy nesmírně složitou, dlouhodobou a finančně náročnou záležitostí. Z tohoto důvodu tak není na trhu dostatek vzdělávacích aplikací, neboť jejich finanční vyhlídky nejsou tak lákavé, na rozdíl od běžných herních žánrů. S příchodem volně dostupných herních enginů, jako je například Unity3D nebo Unreal Engine, se však tvorba her stává dostupnější aktivitou i v očích jednotlivce, neboť výrazně snižuje časovou náročnost i potřebné znalosti pro tvorbu multiplatformních aplikací.

Moje motivace při volbě tématu tvorby mobilní vzdělávací aplikace Malý princ pramení především z dlouholetých zkušeností s objektově orientovaným programováním v jazyce C# a tvorbou rastrové grafiky. Abych zúročil obě tyto dovednosti dohromady, věnuji se již druhým rokem studiu tvorby počítačových her a mobilních aplikací v herním enginu Unity 3D. Jelikož výsledná aplikace má, kromě žáků základních škol, pomoci i maturantům při zkoušce z povinné četby, zvolil jsem tématiku Malého prince, neboť ze seznamu povinné maturitní literatury je to knížka mé osobě nejbližší.

Moje diplomová práce se skládá ze dvou částí – teoretické a praktické. V teoretické části je představen herní engine Unity3D spolu s popisem jeho uživatelského rozhraní a základními funkcionalitami. Praktická část se této práce se pak věnuje celkovému vývoji demoverze vzdělávací hry Malý princ, určené jak pro žáky základních škol, tak pro připomenutí povinné maturitní četby žáků středních škol. Výstupem praktické části je kromě uceleného návodu pro tvorbu her v Unity3D také demoverze aplikace Malý princ, určené pro mobilní telefony s operačním systémem Android.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem teoretické části práce je představení herního enginu Unity a jeho porovnání s jinými, běžně komerčně využívanými herními enginy. Dále pak nastudování a představení základních postupů vývoje dvourozměrných počítačových her a her pro Android OS v prostředí tohoto multiplatformního herního enginu s využitím programovacího jazyka C# a závěrem průzkum trhu současných mobilních a počítačových her s tématikou Malého prince za účelem originality praktické části.

Cílem praktické části práce je samotný vývoj unikátní vzdělávací hry Malý princ pro Android OS s využitím základních funkcí enginu Unity, jako například fyziky, animace, kolizí, C# skriptů a jiných. Hra má za cíl sledovat celý děj původní předlohy Malý princ od Antoine de Saint-Exupéryho. Po skončení jednotlivých fází hry budou moci hráči absolvovat testovací kvízy ze základních otázek týkajících se děje, které budou v závěru hry vyhodnoceny.

2.2 Metodika

Prvním krokem tvorby této práce je nastudování historie, vlastností, výhod a nevýhod z odborné literatury zabývající se obecným představením Unity Engine. Následujícím krokem je pak nastudování odborné literatury, zabývající se konkrétně vývojem dvourozměrných her a vývojem her pro operační systém android. Následuje nastudování knihy Malý princ z důvodu kompletní znalosti titulu a dodržení dějové linie hry. Posledním krokem teoretické části je průzkum pomocí nejznámějších portálů pro distribuci her, jako například Steam.com nebo Itch.io, za účelem nalezení všech titulů s tématikou malého prince, jejich otestování a návrh modelu hry odlišující se od testovaných titulů.

Tvorba praktické části je pak rozdělena do několika navzájem se prolínajících částí: tvorba základní testovací grafiky v programu Moho 12, tvorba uživatelského rozhraní hry, včetně ovládání, inventáře, systému pro zaznamenávání postupu ve hře, dialogového systému, úkolového systému a metody pro ukládání postupu ve hře, třetí částí bude vytvoření finální grafiky potřebné pro vyobrazení kompletního děje, čtvrtou částí praktické

části je vytvoření kompletního děje pomocí nově vytvořených součástí rozhraní hry a finální částí je pak vytvoření souboru ve formátu .apk obsahujícího kompletní hru, a jeho testování.

3 Teoretická východiska

3.1 Role her ve vzdělávání

Hra je činnost, vytvářená jedním, či více lidmi, která nutně nemusí mít konkrétní smysl, kromě vytváření radosti a zábavy. Cílem vzdělávacích didaktických her, ať už určených na počítač, či mobilní platformy, je dát takovým hrám smysl i z hlediska nově nabytých poznatků. Nikdy se však nejednalo o plnohodnotné nahrazení učitele. V určité fázi procesu výuky však hry mohou sloužit ke krátkodobému nahrazení činnosti pedagoga. Může se jednat například o testování, procvičování či doplňování učiva. [1]

3.1.1 Didaktické aplikace

Cílem her všech různých žánrů je obvykle dosahování nějakých cílů. Může se jednat o sbírání bodů za splněné úkoly, nejlepší čas v závodních hrách či v případě her pro dospělé zlikvidování všech protivníků. Didaktické aplikace si kladou za cíl dosahování didaktických cílů tak, aby si to dítě či student ne vždy nutně uvědomil. Hru totiž hraje primárně z důvodu pobavení. *„Didaktická hra realizovaná prostřednictvím počítače je činnost jedince (či jedinců), která má podstatu ve virtuálním prostředí simulovaném počítačem a primárně spočívá v rozvoji osobnosti, přičemž dle svého zaměření může poskytovat zábavu, odreagování nebo relaxaci.“* [2] Předchozí citaci lze samozřejmě aplikovat nejen na počítačové platformy, ale v současné době i na mobilní zařízení a jiné platformy. [1]

Aby byla aplikace označována za výukový program, musí být schopna plnit alespoň jednu didaktickou funkci. Mezi tyto funkce patří motivace, expozice učiva, upevnění osvojených dovedností a kontrola vědomostí a dovedností. Existuje řada aplikací, které studenti využívají při studiu velmi často, ale přesto se o didaktické aplikace nejedná. Jde například o aplikace z balíku Microsoft Office, jelikož tyto aplikace neplní žádnou z didaktických funkcí. Didaktické aplikace musí být přehledné, názorné, informačně správné a jednoduché. Takové aplikace se dají využívat jak v prezenční výuce, tak při samostudiu. [2]

3.1.2 Kategorizace didaktických aplikací

Dle Dostála, J. [1] lze na základě průzkumu 148 výukových programů od českých i zahraničních dodavatelů kategorizovat didaktické aplikace na základě následujících kritérii.

3.1.2.1 Míra interaktivity¨

Didaktické aplikace mohou či nemusí být interaktivní. Interaktivita je velmi podstatná vlastnost aplikace, která spočívá alespoň v minimální možnosti ovlivnění průběhu hry hráčem. Může jít například o změnu děje hry, či jen volitelné pořadí plnění povinných úkolů. Interaktivní hra může být pro hráče mnohem poutavější a v případě vzdělávacích her může být navýšena motivace do učení. [1]

3.1.2.2 Úroveň vzdělávání

Výukové aplikace mohou a také jsou tvořeny pro všechny úrovně vzdělávání, od mateřských škol až po školy vysoké. U jednotlivých aplikací musí být vždy při tvorbě zohledňována věková odlišnost uživatelů. Hra musí být uzpůsobena nejen z hlediska látky a obsahu, ale i z hlediska vzhledu tak, aby byla zachována atraktivnost pro danou věkovou skupinu. [2]

3.1.2.3 Poskytování zpětné vazby

Poskytování zpětné vazby je velice důležitou vlastností didaktický aplikací. Žáci by měli z hlediska psychologie vždy dostávat zpětnou vazbu. Pomůže jim to v ujištění, že v daném procesu postupují správně, či pobídne ke korigování postupu. Mohou se tak poučit z vlastních chyb, jelikož prostřednictvím zpětné vazby vědí, kde chybu udělali. Ne všechny vzdělávací aplikace však zpětnou vazbu studentům poskytují. [2] Zpětná vazba v případě hry může být realizována například počítadlem hráčova skóre, získaného ze správných odpovědí, čí herních úspěchů.

3.1.2.4 Organizovanost vzdělávání

Aplikace můžou být určené buďto na přímou školní výuku, či na samostudium. Aplikace na mobilní zařízení jsou obecně vnímané jako vhodné pro samostudium, zatímco

počítačové aplikace, nainstalované ve školních počítačových učebnách jsou spíše vhodné na přímou výuku, či kombinaci obojího. [1]

3.1.2.5 Počet uživatelů

Didaktické aplikace mohou být určené i pro větší počet uživatelů. V tomto případě se pak kromě znalostí či schopností rozvíjí sociální vztahy a týmovou práci. Kooperace může být pro žáky poutavější a více motivující. Z technického hlediska může být taková aplikace realizována dvěma způsoby. Buďto žáci při hře využívají jeden sdílený počítač, kde se mohou například střídat, či mít rozdělené ovládání na každé straně klávesnice, nebo můžou mít aplikaci zapnutou každý na svém zařízení, kde tyto aplikace komunikují přes síť. Z hlediska víceuživatelských vzdělávacích aplikací pro mobilní telefony je druhé řešení výhodnější. [1]

3.1.2.6 Počet didaktických funkcí

Jak bylo zmíněno v kapitole 3.1.1, aplikace musí plnit alespoň jednu didaktickou funkci. Existují tedy aplikace, které plní přesně jednu takovou funkci a také aplikace, které jsou z tohoto hlediska polyfunkční a mohou plnit všechny čtyři funkce naráz. [2]

3.1.2.7 Zaměření aplikace

Poslední zmíněnou možností je rozdělení aplikací na základě vyučovaného předmětu. Aplikace mohou být buďto zaměřené na konkrétní předmět (cizí jazyk, matematika, literatura, …), či mohou být bez předmětového zaměření. Takové aplikace jsou pak určeny například na posilování logického uvažování, sociálních vazeb a podobně.[2]

3.1.3 Potřebné vlastnosti didaktické aplikace

Jak již bylo zmíněno v kapitole 3.1.1, didaktická aplikace musí plnit alespoň jednu didaktickou funkci. Ze všech zmíněných vlastností v kapitole 3.1.2 pak plyne, že je vhodné, ne však bezpodmínečně nutné, aby aplikace poskytovala zpětnou vazbu žákům či studentům a obsahovala alespoň základní interaktivní prvky. Zároveň by také měla být vhodně zvolena cílová platforma, na základě plánovaného využití ve škole, či mimo ni.

3.2 Literární dílo Malý princ

Kniha Malý princ, obsažená v povinné maturitní četbě, je nejznámějším literárním dílem francouzského spisovatele Antoina de Saint-Exupéryho. Je považována za jeden z nejznámějších pohádkových titulů moderní světové literatury. Kniha, vydaná v roce 1943, patří mezi 50 nejprodávanějších knižních titulů světa a byla přeložena do více než 250 jazyků. Dílo, určené jak pro děti, tak dospělé, obsahuje mnoho metafor a přirovnání, které narážejí na rozdílné a mnohdy nepochopitelné chování dospělých z pohledu dětí. [3]

3.2.1 Děj Malého prince

Kniha Malý princ se skládá z 27 kapitol, v nichž vypravěč, postava pilota, seznamuje čtenáře se životem malého prince, kterého potkal po havárii svého letadla na poušti. Po setkání malý princ přibližuje pilotovi život na své malé planetce. Na této planetce žije kromě malého prince pouze jeho neskromná růže, o kterou se pravidelně stará. Malý princ se ale rozhodne svoji růži opustit a cestovat po okolních planetkách.

Na svých cestách potká několik postav, z nichž každá ztělesňuje jednu, pro dospělé typickou a pro děti nepochopitelnou, vlastnost. Jako první narazí malý princ na krále, který za potřebu všem okolo sebe rozkazovat. Přikazuje dokonce běžné činnosti jako je zívnutí, či sezení. Dále potká malý princ domýšlivce, který touží jen po tom být obdivován. Třetí postavou je pijan, který pije a vlastně neví proč. Poté malý princ narazí na byznysmena, člověka, která se zajímá jen o sbírání a ukládání financí, metaforicky znázorněných v podobě hvězd.

Další navštívenou planetkou je lampářova planeta. Lampář je jedinou postavou, ze všech, které zatím potkal, jemu imponující. Lampář znázorňuje věrné, až slepé, plnění povinností, jelikož stále plní příkazy a rozsvěcí a zhasíná lampu, přestože to musí dělat bez přestání každou minutu. Šestou malou planetkou je zeměpiscova planetka. Zeměpisec tráví veškerý čas tvorbou map, ale nikdy neopouští svůj stůl.

Poté již princ dorazí na sedmou planetu, kterou je planeta Země. Zde potká lišku, se kterou se spřátelí a ochočí si ji. Narazí zde na další dvě postavy. První je výhybkář, který přehazuje výhybky lidem, kteří jen spěchají a nemají čas na nic. Druhou je obchodník, který prodává pilulky proti žízni. Princ se však v závěru své cesty touží vrátit zpět na svou

malou planetku, ale jelikož s sebou nemůže vzít i své tělo, dohodne se s hadem, který ho uštkne a princ umírá. [4]

3.3 Herní engine

Herní engine, či také herní framework, je softwarové vývojové prostředí určené k tvorbě her, určených na různé typy platforem, jako například počítače, herní konzole či mobilní zařízení. Skládá se z několika součástí, které za vývojáře řeší ty nejpodstatnější úkony, jako například renderování grafiky, fyziku, propojení objektů se scripty psanými v programovacím jazyku, síťové prvky, ovládání a desítky dalších, pro hru potřebných úkonů. Herní enginy jsou standartně všestranné a nevymezují se jen na konkrétní herní žánry. [5]

3.3.1 Základní pojmy

Pro práci s herním enginem je nutné znát několik základních, pro všechny herní enginy typických, pojmů. Pro valnou část těchto pojmů se nepoužívá a ani neexistuje korektní český překlad.

3.3.1.1 Asset

Pojem *asset* zahrnuje jakoukoli formu externího zdroje, který je nakonec obsažen ve finální hře. Může jít tedy například o obrázky, 3D objekty, audio a video, herní mapy, postavy či efekty. I kód napsaný v programovacím jazyce je považován za asset, jelikož je tvořen mimo herní engine, například v programu Visual Studio, a herní engine pak s tímto kódem pouze pracuje. [6]

3.3.1.2 Build

Termín build označuje proces, při kterém se programovací kód a veškeré herní objekty v enginu převádějí na samostatnou aplikaci. Při tomto procesu se za pomoci SDK programovací kód kompiluje, tedy překládá do spustitelné podoby. Termín build představuje kromě tohoto procesu také synonymum slova verze. [6]

3.3.1.3 GUI

Za grafické uživatelské rozhraní, či v angličtině graphical user interface je v případě herních enginů označován herní objekt, či skupina objektů, které se zobrazují

před veškerými ostatními herními objekty a pomocí kterých se hra ovládána. GUI, je na rozdíl od ostatních herních objektů závislé na velikosti zobrazovacího rozhraní, a ne na velikosti herního světa. Může se skládat z prvků jako jsou tlačítka, texty, panely či obrázky. [7]

3.3.1.4 Kolize

Kolize označuje stav, ve kterém dva různé herní objekty, například objekt *hráč* a objekt *zeď* sdílejí alespoň jeden společný bod. S pomocí této metody je tak možné rozpoznat, zda se dva objekty vzájemně dotýkají a v případě, že se jedná o fyzické neprůchozí objekty, aplikace zabrání jejich vzájemnému průniku. Kolize je také možné využívat jako spouštěč pro nějakou akci. V takovém případě je jeden objekt fyzický a neprůchozí a druhý, sloužící jako spouštěč, je nehmotný. V případě doteku či průniku se pak provede daná akce. [8]

3.3.1.5 Real-Time Render

Rendering označuje proces tvorby obrazu na základě modelu vytvořeného v herním enginu, grafickém či animačním programu. Takový model se běžné skládá z různých objektů, efektů, osvětlení a podobně. Rendering je obvykle velmi výpočetně náročný a dlouhodobý proces. Real-time rendering je metodou, která dokáže obraz vykreslovat v reálném čase, což je důležité právě pro herní průmysl. [9]

3.3.1.6 SDK

SDK (zkratka pro software development kit) je sada vývojových nástroj, která má za úkol automaticky vytvořit z programového kódu či projektu v herním enginu hotovou aplikaci ve formátu kompatibilním s platformou, na kterou je SKD mířeno. Tato sada nástrojů může být buďto integrována přímo do herního enginu nebo instalována zvlášť. [10]

3.3.1.7 Sprite

Jedná se o dvourozměrný bitmapový obrázek, který je součástí vetší scény. U trojrozměrných objektů se sprite používá jako textura jednotlivých polygonů. Dříve byly jako sprite označeny pouze statické dvourozměrné objekty užívané jako například vzdálené

pozadí, v dnešní době je však toto označení používáno i pro menší a animované objekty. [6]

3.3.2 Příklady herních enginů

Herních enginů existují stovky, ne všechny jsou však dostupné pro veřejnost, neboť některé z nich vytvářejí a vlastní herní studia pouze pro svou vlastní potřebu. Tato kapitola uvádí několik příkladů veřejně dostupných a obecně známých enginů.

3.3.2.1 Unreal Engine

Tento herní engine, který se řadí mezi nejčastěji používané herní enginy, byl poprvé představen v roce 1998, kdy byl použit ve zvané Unreal. Jeho první verze byla zaměřena pouze na hry žánru FPS neboli first-person shooter, a nebyla určená pro veřejnost, ale pouze pro potřeby tvůrce tohoto enginu, firmy Epic Games.

V současné době je aktuální verze Unreal Engine 4, která byla vydána v roce 2014. Tato verze enginu je od roku 2015 ke stažení zcela zdarma, a to i v případě komerčního využití. V takovém případě však společnost Epic Games požaduje pětiprocentní podíl z prodejných her v případě, že vývojář utržil v období tří měsíců více než tři tisíce dolarů.

Jádro tohoto enginu je napsané v jazyce C++. Stejný jazyk je pak využíván při tvorbě aplikací samotným uživatelem. Znalost tohoto jazyka však není nutně potřeba, neboť byla společně s verzí 4 představen nástroj Blueprint Visual Scripting, ve které se hry vytvářejí sestavováním předdefinovaných diagramů, zastupujících jednotlivé herní funkcionality.

Vývojové prostředí podporuje operační systémy Windows, MacOS i Linux. Aplikace, vytvořené prostřednictvím Unreal Engine 4 mohou být cíleny, kromě již zmíněných operačních systémů, například na platformy PlayStation 4, Xbox One, Xbox Series X, Nintendo Switch, iOS a Android. Dále tako podporuje platformy pro virtuální realitu Oculus VR, SteamVR, Samsung Gear VR a Google VR. V celkovém součtu podporuje Unreal Engine 20 různých platforem. [11]

V květnu roku 2020 byla oznámena nová verze, Unreal Engine 5, jejíž datum spuštění je odhadováno na konec roku 2021. Bylo také oznámeno, že tato nová verze bude obsahovat funkci zvanou Nanite, která umožňuje zpracovávat reálné fotografie ve vysokém rozlišení a následně je importuje zpracované a připravené pro využití v herním světě. Jinými slovy tato funkce dokáže na základě fotografií vymodelovat herní objekty

jako například předměty či budovy a usnadní tak práci při tvorbě herní grafiky. Tato verze by měla obsahovat plnou podporu pro platformu PlayStation 5. [12]

3.3.2.2 Unity3D

Dalším hojně využívaným herním enginem je Unity3D, od vývojářské společnosti Unity Technologies. První verze Unity3D byla vydána v roce 2005, tehdy se však jednalo o herní engine určený pouze pro platformu Mac OS X. Kompatibilita vývojového prostředí s operačním systémem Windows přišla o rok později, stejně jako podpora tvorby aplikací pro tuto platformu. Na rozdíl od Unreal Engine vznikl Unity3D cíleně jako veřejně dostupný herní engine, nikoli jako engine prvotně určený pro vývoj konkrétní hry.

Od roku 2016 je možné stáhnout Unity spolu s programem Visual Studio Community zcela zdarma, a to za předpokladu, že firma, či jednotlivec, negeneruje roční obrat vyšší než 100 000 dolarů. V takovém případě je už povinností zakoupení licence Plus, Pro nebo Enterprise. Základní placená licence Plus stojí 399$ ročně, licence Pro pak stojí 1 800$ ročně. Poslední verze, Enterprise, stojí 2 000$ měsíčně. První dvě licence jsou určené pro jednotlivé uživatele, verze Enterprise je však určena pro deset uživatelů, či zařízení. [13]

Přestože jádro programu Unity je napsáno v jazyce C++, API (zkratka pro Application Programming Interface) využívá v současné době pouze jazyk .NET C#. V dřívějších verzích API Unity podporovalo také UnityScript, vycházející z JavaScriptu a Boo, což je jazyk založený na syntaxi Pythonu. Unity disponuje rozsáhlou knihovnou oficiálních i komunitou vytvořených doplňků. S pomocí doplňků, jako je například Playermaker, či GameFlow, je možné dosáhnout stejného výsledku jako při použití Blueprint Visual Scripting z Unreal Engine a vytvářet tak aplikace bez znalostí programovacích jazyků. [14]

Vývojové prostředí Unity je možné spustit, stejně jako v případě Unreal Engine, na operačních systémech Windows, Linux a MacOS. Tvorba aplikací je v aktuální verzi 2020.2.2 možná na stejné platformy jako Unreal Engine, je však také rozšířena o možnost tvorby aplikací pro internetové prohlížeče pomocí WebGL, či tvorby aplikací pro chytré televize. Dohromady Unity podporuje 25 různých platforem, tedy o pět více než Unreal Engine. Detailnější rozbor uživatelského rozhraní tohoto enginu je obsažen v kapitole 3.2. [13]

3.4 Funkce a uživatelské rozhraní Unity3D Engine

Unity3D disponuje širokou škálou výchozích funkcí, zajišťujících například fyziku, vzhled objektu, síťové propojení, přehrávání audia a videa, osvětlení, kolize, kamer a desítky dalších operací. Tyto výchozí funkce je možné přiřazovat objektům a ovládat jejich parametry čistě přes uživatelské rozhraní, a není tedy nutné v těchto případech používat programovací jazyky. Cílem této kapitoly je seznámení se základními prvky uživatelského systému a funkcemi enginu Unity3D.

3.4.1 Uživatelské rozhraní

Po založení nového projektu v Unity se otevře samotný program s výchozím nastavením zobrazovaných oken. Tyto okna pokrývají veškeré základní funkce, potřebné k prvotnímu budování nového projektu. Při výchozím nastavení programu se otevřou okna: *Project*, *Scene*, *Inspector*, a *Hierarchy*. Na pozadí se pak otevřou ještě další nepostradatelná okna a to: *Console*, *Animator*, *Animation*, *Game* a *Asset store*. Výchozí zobrazení je složeno ze čtyř panelů: levého, pravého, prostředního a spodního. Nad každým z těchto panelů je zobrazena malá lišta, která umožňuje přepínání mezi aktivními okny v daném panelu. [8]

Obrázek 1 – Výchozí uživatelské rozhraní Unity3D

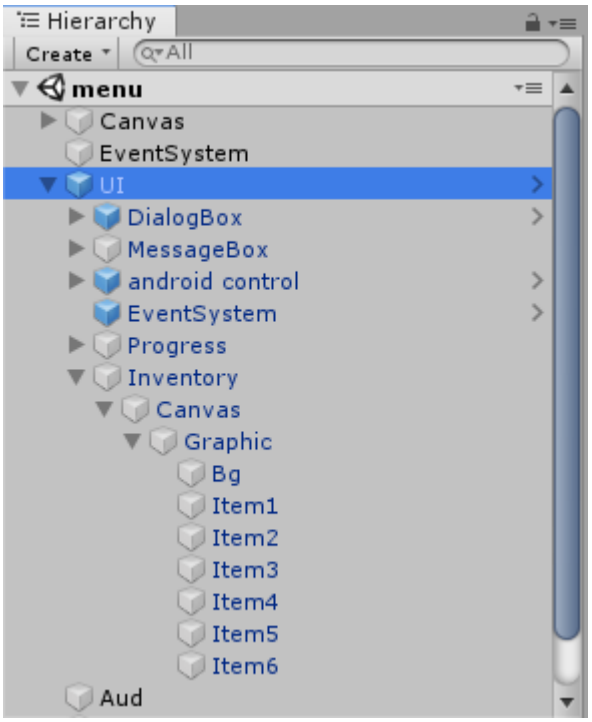
3.4.1.1 Project

Okno *Project* slouží jako základní průzkumník všech souborů obsažených v celém projektu. Veškerý obsah, jako například C# metody, herní objekty, hudba, video, obrázky, 3D objekty jsou zde ukládány a řazeny do složek dle potřeb vývojáře. Mezi těmito soubory je možné vyhledávat podle jména či typu souboru. Nutno podotknou, že velikost a obsah souborů v okně *Project* neurčuje výslednou velikost hry, neboť do hry samotné jsou zařazeny pouze soubory obsažené ve scénách přiřazených do finálního build hry. [15]

Obrázek 2 - Okno *Project*

3.4.1.2 Hierarchy

V tomto okně jsou zobrazeny veškeré herní objekty, které jsou používané v aktuálně načtené scéně. Na rozdíl od okna *Project* zde nejsou herní objekty umístěné ve složkách, ale můžou být umístěné vzájemně do sebe ve vztahu parent-child, ať už kvůli přehlednosti, či kvůli aplikaci určitých parametrů na více objektů naráz. Pokud bude aktuálně zobrazovaná scéna přidána do konečného buildu hry, všechny objekty v okně *Hierarchy* pak budou obsažené i ve finálním buildu hry. [16]



Obrázek 3 - Panel *Hierarchy*

3.4.1.3 Inspector

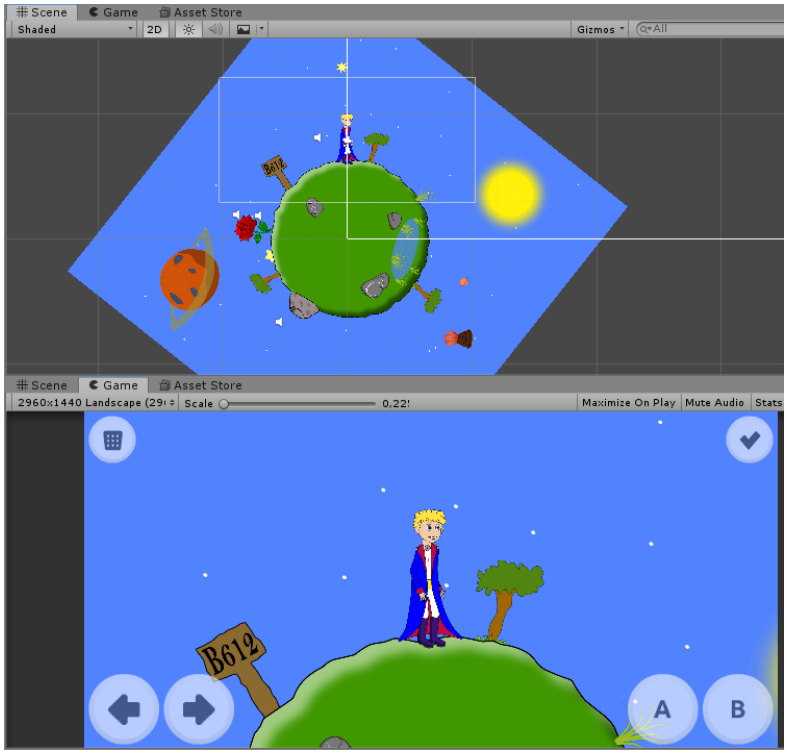
Každá scéna se skládá z několika různých herních objektů (viz Obrázek 3 – Panel *Hierarchy*). Tyto objekty se pak skládají z určitého množství komponentů. Mezi tyto komponenty patří například různé, uživatelem napsané, C# skripty, fyzikální či grafické komponenty a podobně. K zobrazení komponent herního objektu, po jeho rozkliknutí v okně *Hierarchie*, slouží právě panel *Inspector*. V tomto panelu je možné upravování nastavení jednotlivých komponent a nastavování jejich veřejných proměnných. [15]

3.4.1.4 Scene

Posledním z výchoze otevřených oken je okno *Scene*. Každá hra, vytvořená v Unity3D je složena z jedné, čí více scén. Vše, co se ve hře stane, se odehrává na jedné z těchto scén. Okno *Scene* zobrazuje volný pohled na aktuálně otevřenou scénu. Scény se běžně skládají z textur, které můžou být buďto dvou či trojrozměrné, colliderů, kamer, osvětlení a jiných herních objektů. Okno *Scene* slouží k editaci rozměrů a pozice těchto objektů, a tedy k sestavení scény jako celku. [5]

3.4.1.5 Game

Okno *Game* je první z oken, která sice jsou při zapnutí programu Unity výchoze zapnuta, nejsou však ale zobrazena. Toto okno se nachází ve stejném panelu jako *Scene* a pro jeho zobrazení je nutné přepnutí na horní panelové liště. Slouží k náhledu na aktuální scénu. Na rozdíl od *Scene* je však tento pohled vždy zobrazen skrze aktivní kameru spolu se zapnutým uživatelským rozhraním a jde tedy spíše o náhled na scénu z pohledu hráče. [6]



Obrázek 4 - Rozdíl mezi *Scene* (nahoře) a *Game* (dole)

3.4.1.6 Console

V tomto okně se zobrazují veškeré chyby, varování, informace a jiné textové výstupy z programu Unity. Tyto výstupy zůstávají v konzoli zobrazené až do jejich smazání. Smazat tyto hlášky může vývojář kliknutím na tlačítko *Clear* či *Clear on Play*. Pokud je hlášky v konzoli označena jako chyba, lze ji také odstranit opravením uvedené chyby.

Při psaní metod je možné také využívat příkaz *Debug.Log(„text“)*, který vypíše do konzolového okna libovolný text či proměnnou zadanou v závorkách. Toho se hojně využívá při debuggování programu, kdy je v reálném čase například možné vidět, zda program vykonal nějakou konkrétní část kódu, či je tato část z důvodu nějaké chyby

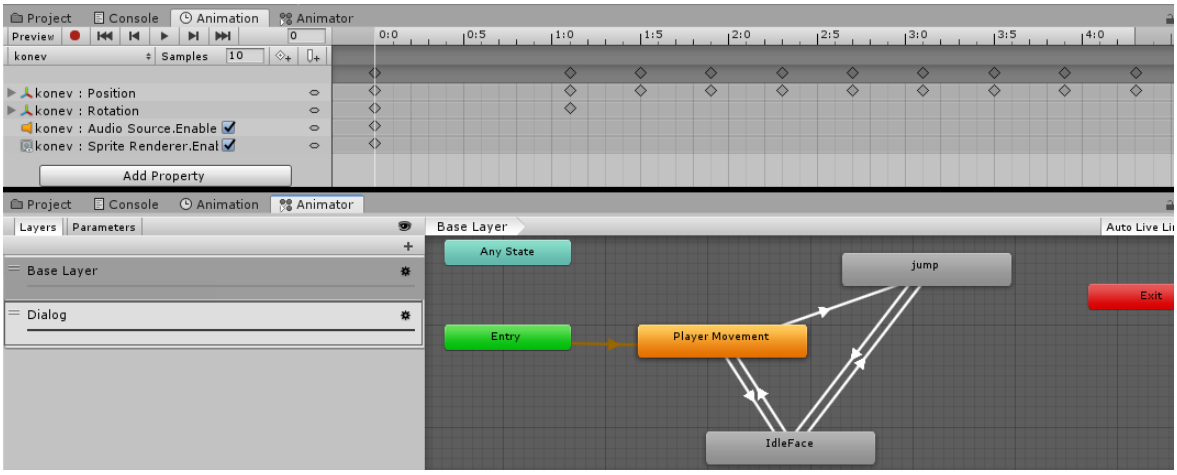
nedosažitelná. Konzolové okno se ve výchozím zobrazení nachází na spodním panelu, spolu s oknem *Project*. [7], [16]

3.4.1.7 Animator a Animation

Animace ve hře může nabývat různého charakteru. Animovat se může například pohyb postavy, počasí, interakce herních předmětů, či také přechody mezi scénami, změna audio stopy a změna podoby UI. Ke tvorbě a ovládání animací slouží okna *Animator* a *Animation.*

Obě tyto okna jsou při výchozím zobrazení umístěna ve spodním panelu, spolu s *Project* a *Console*. Okno *Animation* slouží k vytvoření animace. Po jejím vytvoření a pojmenování se v tomto okně zobrazí časová osa, na které lze provádět všemožné úpravy typu změna pozice, změna podoby, změna velikosti, změna zvukové stopy a jiné. Tyto změny se do časové osy promítají automaticky. Stačí pouze zapnou nahrávání animace a poté každá změna, provedená na herním objektu, pro který je tvořena animace, bude zapsána do časové osy.

V okně *Animator* se pak automaticky vytvoří jakýsi rozcestník mezi vytvořenými animacemi. Mezi animacemi se pak v tomto okně vytváří různě podmíněné přechody, pro přepínání mezi jednotlivými fázemi animace. [15], [16]



Obrázek 5 - Okna *Animation* (nahoře) a *Animator* (dole)

3.4.1.8 Asset store

Posledním zmíněným oknem z výchozího zobrazení uživatelského rozhraní je *Asset store*. Toto okno se nachází na centrálním panelu spolu se *Scene* a *Game*. Po jeho otevření se v tomto panelu otevře internetový obchod nabízející širokou škálu doplňků (assetů)

pro tvorbu aplikací v Unity. Tyto doplňky jsou tvořeny jak firmou Unity Technologies, tak širokou veřejností. Některé z těchto doplňků jsou nabízeny zdarma, převažují však doplňky placené. Jejich ceny se pohybují od jednotek dolarů až po tisíce. Dohromady nabízí obchod téměř sedmdesát tisíc assetů rozřazených do 29 kategorií. [17]

3.4.2 Vybrané funkce Unity 3D

Funkce v programu Unity 3D částečně odrážejí základní pojmy herních enginů z kapitoly *Herni engine*. Zde popisované funkce jsou však již pevnou součástí Unity 3D se všemi vlastnostmi a parametry. Tyto funkce jsou často také nazývány jako komponenty, či třídy.

3.4.2.1 Transform

Transform je základní komponentou všech herních objektů. Vytvoří se vždy spolu s vytvořením herního objektu a není možné ji smazat. Udává souřadnice herního objektu podle hodnot X, Y a Z. Kromě souřadnic obsahuje možnost rotace objektu podle všech tří zmíněných os a také možnost úpravy rozměrů herního objektu, též ve třech zmíněných směrech. Všechny tyto parametry jsou vzájemně děděné mezi nadřazenými a podřazenými objekty. [5]

3.4.2.2 Sprite Renderer

Sprite Renderer je třída, jejíž funkcí je přiřazování bitmapových obrázků na herní prvky. Tato funkce se používá primárně při tvorbě dvourozměrných aplikací. Obsahuje několik důležitých parametrů. Prvním je samotná cesta k obrázku, uloženém v projektu. Druhým je dvojice parametrů, *Flip X* a *Flip Y*, po jejichž zvolení se nastavený obrázek otočí, buďto po ose X nebo po ose Y, případně po obou. Posledními dvěma parametry jsou *Sorting Layer* a *Order in Layer*. V prvním parametru je možné vytvořit různé vrstvy, jimž je přiřazeno pořadí vykreslování. Každý objekt, obsahující třídu *Sprite Renderer*, je pak vykreslován v nějaké z těchto vrstev, což určuje jejích vzájemné překrývání. Druhý parametr, *Order in Layer*, určuje pořadí překrývání, pokud jsou dva či více objektů ve stejné vrstvě. [7]

3.4.2.3 Rigid Body

Třída *RigidBody*, přiřazená k hernímu objektu umožní tomuto objektu komunikaci s fyzikálním enginem programu Unity. Takový objekt se pak stane ovlivnitelný vnějšími silami, jako je například gravitace, či nárazová síla jiných objektů. Tato třída obsahuje několik nastavitelných parametrů. Prvním parametrem je *Body Type*. Tento parametr udává, jaký bude herní objekt typ. První možností je *Dynamic*. Dynamické tělo reaguje na všechny vnější podměty od ostatních objektů, včetně gravitace. Druhou možností je *Kinematic*. Na takový objekt nepůsobí vnější fyzikální podněty, jako například gravitace. Mohou se také pohybovat a mají svou simulovanou hmotnost, podnět k pohybu však musí přijít z kódu. Poslední možností je *Static*. Tato možnost je určená pro objekty, které se pohybovat nebudou vůbec. [15]

3.4.2.4 Collider

Collider je třída uvnitř programu Unity, která má za úkol kontrolovat detekci kolizí. Přiřazuje se všem objektům, které mají své fyzikální vlastnosti či objektům, které složí jako neprůchozí překážky pro hráče. Pokud mají dva objekty, například objekt postavy hráče a objekt zeď, přiřazeny třídu *Collider*, program Unity zabrání jejich vzájemnému průniku. Tato třída obsahuje, kromě jiného nastavení, také metodu *IsTriggerer*. Pokud je tato metoda zapnuta, tedy hodnota *IsTriggerer* je nastavena jako pravda, program nebrání průniku dvou objektů a pouze průnik zaznamená, což je využíváno pro sepnutí nějaké konkrétní akce, pokud se hráč nachází v cílovém místě.

Tato třída je v Unity vytvořena pod několika různými názvy, například *Box Collider, Circle Collider, Capsule Collider*, všechny tyto třídy mají shodné vlastnosti a liší se pouze ve fyzickém tvaru používaného collideru. [18]

3.4.2.5 Canvas

Canvas je označení pro speciální herní objekt, který nese stejnojmennou komponentu. Tento objekt je určený pro vykreslování kompletního uživatelského rozhraní. Nenachází se tak přímo v prostředí herní scény, ale je spíše vnímán jako součást obrazu kamery snímající tuto scénu. V jedné scéně může být umístěno více takových objektů, přičemž jejich pořadí a vzájemné překrývání je určeno pouze pořadím, v jakém jsou umístěny v hierarchii. Herní objekty typu *Canvas* mohou mít konstantní rozměry udávané

v pixelech, či mohou mít nastavenou vlastnost změny rozměrů dle velikosti obrazovky, na které jsou zobrazovány. [16]

3.4.2.6 Button

Button, neboli česky tlačítko, je jeden z několika herních objektů, které jsou součástí *Canvas* a nikoli herního světa. Skládá se primárně ze dvou komponent, *Button* a *Image*. Komponenta *Image* definuje vzhled tohoto tlačítka, primárně pomocí nahraného obrázku. *Button* zajišťuje především spuštění události *OnClick*, při stisknutí tohoto tlačítka. Do události *OnClick* je možné přímo v prostředí Unity přiřazovat neomezené množství akcí. Akce se do tlačítek přiřazují přesunutím herního objektu do události *OnClick*. Po přesunu se zde zobrazí seznam všech tříd, které jsou součástí tohoto konkrétního herního objektu. Po výběru třídy se zobrazí seznam všech metod dané třídy. Vybraná metoda se pak vyvolá vždy, při stisku tohoto tlačítka. [5]

4 Vlastní práce

V rámci praktické části své diplomové práce se věnuji vytvoření demoverze vzdělávací aplikace Malý princ. Praktická část je tvořena ze čtyř kapitol, analýzy, návrhu, implementace a testování. Největší důraz kladu na kapitolu implementace, která se věnuje kompletnímu postupu při tvorbě této aplikace od tvorby herního světa až po vytvoření finálního souboru ve formátu .apk, spustitelného na zařízeních s operačním systémem Android.

4.1 Analýza

Součástí analýzy je definování požadavků na vzdělávací aplikaci jak z hlediska požadavků didaktických aplikací (viz kapitola Didaktické aplikace) tak z hlediska požadavků na herní mechaniku. Dále je zde obsažena analýza současných aplikací na mobilní zařízení a počítače s tématikou Malého prince dostupných na známých distribučních službách.

4.1.1 Požadavky na aplikaci

Požadavky na aplikaci jsou rozděleny do dvou kapitol. První kapitola zahrnuje požadavky, které jsou potřebné z hlediska plnění funkce didaktické aplikace. Druhá kapitola zahrnuje požadavky na herní mechaniku, definované mnou, jakožto vývojářem i zadavatelem aplikace.

4.1.1.1 Požadavky na didaktickou aplikaci

Primárním cílem aplikace Malý princ je z hlediska studenta možnost zopakovat si již osvojené znalosti literárního díla Malý princ. Aplikace tedy nebude sloužit jako zdroj nových vědomostí, a proto nebude plnit funkci expozice učiva. Z hlediska didaktických funkcí musí aplikace plnit funkci motivace, upevnění osvojených dovedností a kontrolu vědomostí a dovedností.

4.1.1.1.1 Motivace

Požadavek na plnění didaktické funkce motivace je již z části plněn samotnou myšlenkou hry, namísto běžného psaného textu. Aplikace musí být jak z hlediska vizuální

podoby, tak z hlediska herní mechaniky atraktivní tak, aby studenty dostatečně zaujala a motivovala k jejímu dokončení.

4.1.1.1.2 Upevnění osvojených dovedností

Aby byl tento požadavek a zároveň primární didaktická funkce této aplikace splněn, je nutné, aby aplikace obsahovala důležité dialogy a interaktivní aktivity pramenící z knihy Malý princ. Hráč si tak bude během hraní této hry osvěžovat znalosti získané četbou knihy.

4.1.1.1.3 Kontrola vědomostí a dovedností

Aby aplikace plnila i požadavek na didaktickou funkci kontroly vědomostí, bude zapotřebí využití určité formy zpětné vazby. Tato zpětná vazba bude fungovat na principu vědomostního kvízu, zobrazovaného na konci každého dohraného kola. Otázky kvízu se budou týkat dialogů a aktivit z předchozího kola.

4.1.1.2 Požadavky z hlediska herní mechaniky

Požadavky na aplikaci a její herní mechaniku jsou uvedeny v následující tabulce.

Tabulka 1 - Požadavky na herní mechaniku aplikace

Název požadavku	Požadavek
Formát aplikace	Aplikace musí být ve formátu .apk, aby byla spustitelná na zařízeních s operačním systémem Android.
Ovládání	Ovládání musí být realizováno prostřednictvím prvků uživatelského rozhraní na dotykové obrazovce.
Uživatelské rozhraní	Uživatelské rozhraní musí obsahovat ovládání pohybu hráče, ovládání interakcí a ovládání inventáře.
Interakce s předměty	Hra musí umožňovat interakci s předměty. Jejich sbírání do inventáře, vzájemnou interakci a interakci s ostatními herními objekty.
Kvíz na ověření znalostí	Pro plnění didaktického požadavku na ověřování znalostí je nutné, aby součástí uživatelského rozhraní aplikace byl kvíz, skládající se z posloupnosti otázek se čtyřmi odpověďmi, z nichž vždy je pouze jedna správně.
Ukládání postupu	Je nezbytně nutné, aby hra umožňovala uložení veškerého dosavadního postupu hráče, včetně všech předmětů v inventáři i po ukončení celé aplikace.
Děj v českém jazyce	Aplikace musí obsahovat důležité dialogy a interakce s postavami psané v českém jazyce.

4.1.2 Dostupné hry s tématikou Malý princ

Pro průzkum již existujících aplikací s tématikou Malého prince jsem zvolil distribuční služby Steam, Google Play a itch.io.

4.1.2.1 Steam

Na službě Steam byly pomocí vyhledávání klíčových slov *Malý princ*, *Little prince* a *La petit prince* nalezeny dvě existující aplikace, *My Little Prince – a jigsaw puzzle tale* a *The Little Prince VR*. První zmiňovaná hra, zařazena do žánru *Nenáročné*, spočívá ve skládání puzzle obrázků, jejichž tématikou je právě Malý princ. Hra v žádném směru nepopisuje děj samotné knihy a nemá českou lokalizaci. [19]

Druhá zmiňovaná hra, *The Little Prince VR*, je určena pro virtuální reality HTC Vive, Oculus Rift a Valve Index. Hra však, přesto že jako datum vydání je uváděno jaro 2018, nebyla ještě vydána a není možné ji zatím stahovat. Hra je zařazena do žánru *Dobrodružné* a dle popisu vývojářů hráč cestuje mezi planetami a jeho cílem je plnit interaktivní úkoly známé z knihy. [20] Oficiální stránky této hry však byly již v roce 2017 dle komunity služby Steam zrušeny a předpokládá se, že vývoj byl přerušen. [21]

Obrázek 6 - Ukázka ze hry *The Little Prince VR*

4.1.2.2 Google Play

Služba Google Play nalezla po vyhledání klíčových výrazů, totožných s kapitolou *Steam*, tři aplikace, určené pro platformy s operačním systémem Android. Jsou to aplikace

Little Prince Jump [22], *The Little Prince* [23] a *The Little Prince – An animated childrens book.* [24]

První a třetí aplikace fungují na totožném principu. Aplikace spočívají pouze v přeskakování předmětů. Postava Malého prince běhá kolem dokola planety, na které se zobrazují překážky, která musí hráč klikáním na dotykovou obrazovku přeskakovat. Tyto aplikace jsou tedy inspirovány dílem Malého prince pouze z vizuálního hlediska, nikoli však hlediska příběhového.

Obrázek 7 - Ukázka ze hry *Little Prince Jump*

Druhá zmíněná aplikace, *The Little Prince – An animated childrens book,* je animovaná elektronická knížka. Neexistují zde tedy žádné herní prvky a jedná se o zkrácenou knižní adaptaci, doplněnou animacemi a zvukovými efekty. Aplikace je lokalizována pouze do anglického jazyka.

4.1.2.3 Itch.io

Na platformě Itch.io, určené především pro nezávislé vývojáře, jsem po vyhledávání již zmíněných klíčových slov nalezl celkem pět her týkajících se tématiky Malého prince.

Prvními třemi hrami jsou *The Little Prince Planet* [25], *Asteroid B-612* [26] a *The Little Prince (Old, bitter and redneck)* [27]. Herní mechanika prvních dvou titulů je naprosto shodná. Hráč obíhá kolem dokola své planety a jeho úkolem je kácet rostoucí

baobaby. Pokud baobab stihne dorůst určité velikosti, hra končí. Třetí titul je doplněn o trojrozměrnou grafiku a satirický vzhled.

Zbylé dva tituly jsou *The little prince* [28] a *The Little Prince Game* [29]. První zmíněný titul je graficky poutavá hra žánru *Survival*. Hráč se v roli Malého prince stará o svou planetu, sází květiny a kácí stromy a zároveň se musí starat i sám o sebe, aby přežil. Hra se celou dobu odehrává na jedné planetě a nesleduje děj díla. Poslední titul je žánru *Plošinovka*, kde hráč překonává jednotlivé úrovně přeskakováním mezi plošinami a vyhýbáním se nepřátelským objektům. Hra opět nesleduje děj díla a je pouze inspirována jeho tématikou.

Obrázek 8 - Ukázka ze hry *The little prince*

4.1.2.4 Zhodnocení existujících titulů

Následující tabulka ukazuje výčet nalezených titulů s herní tématikou, jejich vzájemné srovnání z hlediska typu aplikace, dodržování děje a české lokalizace a srovnání s požadavky na analyzovanou aplikaci *Malý princ*.

Tabulka 2 - Srovnání dostupných aplikací s tématikou Malého prince

Název aplikace	Typ aplikace	Platforma	Česká lokalizace	Následování děje
My Little Prince – a jigsaw puzzle tale [19]	Hra	PC	Ne	Ne
The Little Prince VR [20]	Hra	VR	Ne	Částečně
Little Prince Jump [22]	Hra	Android	Ne	Ne
The Little Prince – An animated childrens book [24]	Animovaná kniha	Android	Ne	Ano
The Little Prince [23]	Hra	Android	Ne	Ano
The Little Prince Planet [25]	Hra	PC	Ne	Ne
Asteroid B-612 [26]	Hra	PC	Ne	Ne

The Little Prince (Old, bitter and redneck) [27]	Hra	PC	Ne	Ne
The little prince [28]	Hra	PC	Ne	Ne
The Little Prince Game [29]	Hra	PC	Ne	Ne
Malý princ	**Hra**	**Android**	**Ano**	**Ano**

4.1.3 Závěr analýzy

Z tabulky 2 v kapitole 4.1.2 je patrné, že za předpokladu, že budou splněny všechny požadavky na aplikaci *Malý princ* z kapitoly 4.1.1, půjde z hlediska originality o ojedinělou aplikaci, neboť žádná z nalezených aplikací se, kromě *The Little Prince – An animated childrens book*, nezabývá hlubším dějem knižní předlohy. Zmíněná aplikace se ději sice věnuje, ale nelze ji zařadit mezi hry, neboť postrádá alespoň základní prvky interakce s hráčem a jde pouze o animovanou knihu, která navíc postrádá, jako všechny ostatní aplikace, lokalizaci do českého jazyka.

4.2 Návrh

Tato kapitola obsahuje návrh samotné aplikace z hlediska jejího celkového fungování, tedy herní mechaniky, a z hlediska uživatelského rozhraní.

4.2.1 Herní mechanika

Návrh herní mechanika je rozdělen do několika částí. Skládá se z herní postavy a jejího ovládání, jednotlivých scén a přechodů mezi nimi, dialogů a interakce s předměty.

4.2.1.1 Herní postava

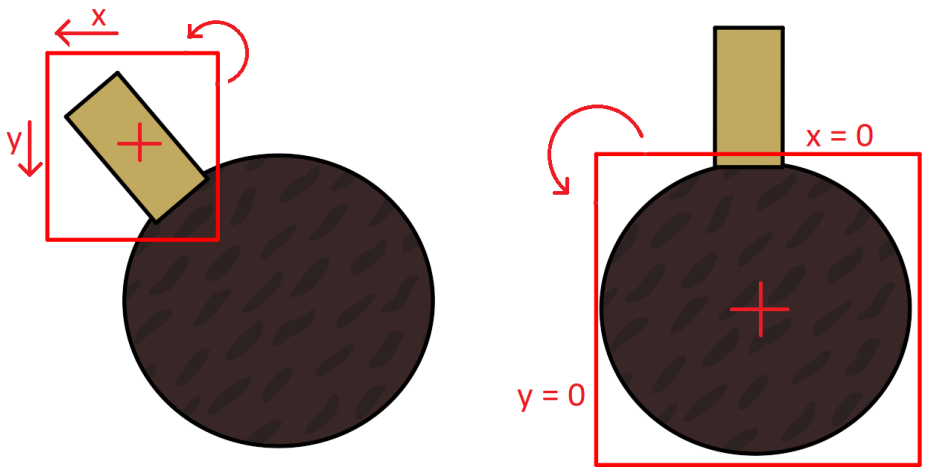
Pro potřeby vytvoření postavy bude nutné vytvořit grafický vzhled postavy a animaci její chůze v programu *Moho*. Tato animace bude vyexportována v sekvenci po sobě jdoucích obrázků ve formátu *.png*.

Herní postava bude vytvořena prostřednictvím jednoho samostatného herního objektu. Tento objekt bude muset mít pro správné fungování přiřazeny čtyři předdefinované funkce z programu Unity. První bude *Sprite Renderer*, který zajistí, aby herní objekt měl vzhled postavy prostřednictvím přiřazeného výchozího obrázku postavy.

Druhou funkcí bude *Animator*, aby byla zajištěna možnost vytvoření animace chůze. Taková animace bude vytvořena postupnými změnami vlastnosti *Sprite* ve *Sprite Renderer* na jednotlivé obrázky vzniklé animací postavy.

Třetí a čtvrtou funkcí bude *Box Collider* a *Rigid Body*. S pomocí těchto funkcí budou zaručeny kolize postavy, a zároveň rozpoznání, zda se herní postava nachází poblíž interaktivních objektů.

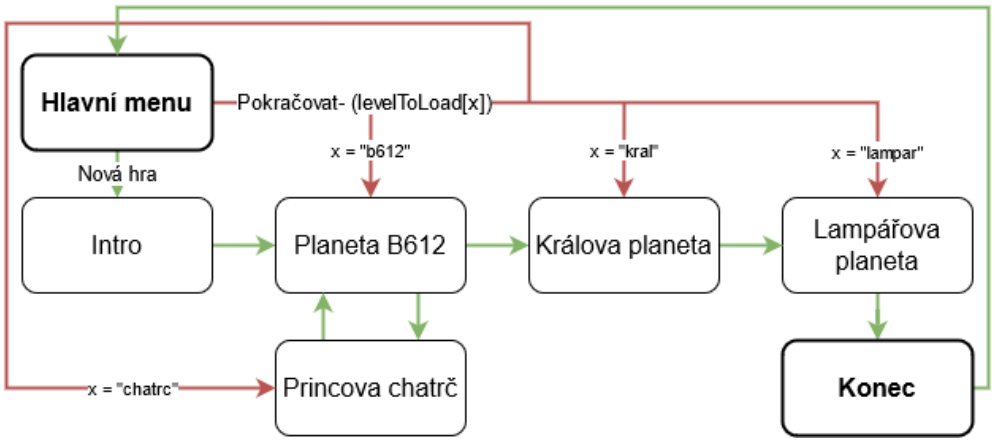
Závěrem bude nutné vytvoření páté funkce, a sice ovládání postavy. Jelikož se bude jednat o dvourozměrnou hru a postava bude obcházet planetu z levé či pravé strany, k ovládání postačí pouze dvě tlačítka, vlevo a vpravo. Postava však bude obcházek kolem celých planet, což vyžaduje přesun po všech třech osách. Takové řešení by ale bylo velmi složité a zbytečně komplikované, jelikož stejného výsledku lze dosáhnou pouze změnou rotace planety a jejího pozadí, kdy postava při pohybu své souřadnice vůbec nebude měnit.



Obrázek 9 - Znázornění možných přístupů k ovládání postavy

4.2.1.2 Scény a přechody mezi nimi

V literárním díle *Malý princ* putuje hlavní postava po několika různých planetách, kde potkává jednotlivé postavy. Jelikož jedním z cílů této aplikace je přesné přiblížení děje, bude nutné vytvořit několik unikátních planet, popsaných v knize, a postav s nimi spjatých. Takovéto planety budou vytvořeny v rámci jednotlivých scén, které se budou postupně načítat ve chvíli, kdy hráč splní všechny požadované úkoly a zmáčkne tlačítko pro cestování na další planetu.



Obrázek 10 - Znázornění přechodů mezi scénami

V rámci této diplomové práce použiji v aplikaci planetku B612, planetku Krále a planetku Lampáře. Pro rozšíření interaktivity přidám ještě vlastní planetku princovy chatrče. Každá z těchto planet bude existovat ve své vlastní scéně. Dalšími třemi scénami bude Hlavní menu, Intro a Konec. V obrázku výše jsou znázorněny jednotlivé scény a přechody mezi nimi. Zelené šipky znázorňuji přirozené postupné přechody v dějové linii. Červené šipky pak ukazují možné přechody v případě, že již existuje uložený soubor s názvem aktuální uložené scény *x*.

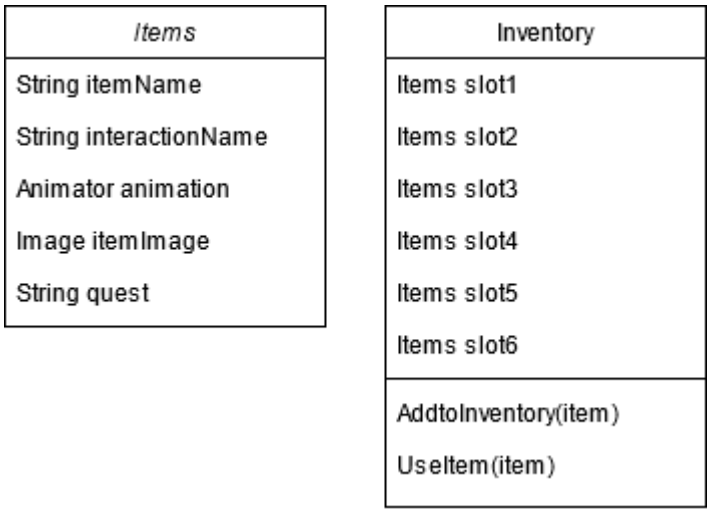
4.2.1.3 Předměty a interaktivita

Aby výsledná aplikace nebyla pouhou animovanou knihou, je nutné přidat několik interaktivních úkolů, které hráč bude muset postupně plnit. Aktuální úkol se bude hráči zobrazovat v uživatelském rozhraní, pro rozkliknutí tlačítka *Úkoly*. Fungující systém předmětů, inventáře a interaktivity bude realizován pomocí třídy *Items* a třídy *Inventory*.

Třída *Items* bude složit pro vytváření jednotlivých konkrétních předmětů, jako například koště, konev a jiných. Bude obsahovat několik klíčových proměnných. Tyto proměnné budou muset stručně, jasně a kompletně definovat konkrétní objekt. Třída *Items* bude tedy obsahovat proměnné pro název předmětu a název předmětu, se kterým bude probíhat interakce. Dále bude obsahovat proměnnou třídy *Image*, která ponese obrázek předmětu a proměnnou třídy *Animator*, která bude mít přiřazenou animaci probíhající interakce.

Třída *Inventory* bude obsahovat šest proměnných typu třídy *Items*. Tyto proměnné, nazvané *slot1* až *slot6*. Dále bude obsahovat metody *AddToInventory* a *UseItem*, využívající vstupní parametr *Item*. Ve výchozím stavu budou mít všechny tyto proměnné

slot nulovou hodnotu. Po kliknutí na předmět, nesoucí třídu *Items,* bude vyvolána metoda *AddtoInventory* s parametrem *Item*, který bude ukazovat právě na zvolený předmět. V inventáři se pak první prázdná hodnota proměnné *slotx* nastaví na hodnotu parametru *Item*. Metoda *UseItem* bude vyvolána kliknutím na předmět v inventáři. Na základě proměnné *interactionName* vyhledá v inventáři předmět, jehož název se bude shodovat s touto proměnnou a provede vzájemnou interakci těchto předmětů (vytvoří předmět nový, nebo spustí animaci).



Obrázek 11 - Návrh tříd *Items* a *Inventory*

4.2.1.4 Dialogy

Pro správné fungování dialogů budou vytvořeny tři třídy. První třída, *Dialog*, bude obsahovat proměnné *DialogName* a *DialogProgres*. První proměnná uvádí název konkrétního dialogu, druhá pak aktuální fázi v tomto dialogu. Dále bude takto třída obsahovat dvě pole proměnných, *names* a *sentences*. V těchto polích budou obsaženy celé dialogy, spolu se jmény postav, které aktuálně daný dialog vyslovují.

Druhá třída, *DialogManager*, bude mít za úkol ovládání celkového průběhu dialogu. Při jeho započetí přiřadí do animace správné obrázky aktuálně diskutujících postav. V průběhu dialogu pak bude přepínat animace hovoru postav na základě toho, která postava právě hovoří. Tuto informaci bude získávat právě z třídy *Dialog* a na jejím základě třída *DialogManager* také aktualizuje zobrazovaný text a zobrazované jméno.

Poslední třídou, používanou pro dialogy, bude třída *DialogTriggerer*. Ta bude mít za úkol dané dialog spouštět. Jednotlivé postavy budou mít přiřazeny právě tuto třídu,

spolu s třídou *dialog*. Pokud hráč stiskne tlačítko pro interakci, třída *dialogTriggerer* ověří, zda je hráč v dosahu postavy a pokud ano, spustí s ní její dialog.

4.3 Implementace

Kompletní tvorbu aplikace Malý princ jsem realizoval v herním enginu Unity 3D, konkrétně ve verzi 2018.3.11. Na psaní kódu v jazyce C# jsem využil přidruženou aplikaci Microsoft Visual Studio Community 2017, konkrétně ve verzi 15.9.13. Na tvorbu grafických prvků a animaci jsem pak využil aplikaci Moho Pro 12 ve verzi 12.2.

4.3.1 Založení projektu

Po zapnutí programu Unity 3D se zobrazí první okno, které vyzívá uživatele k přihlášení. Toto přihlášení není nezbytně nutné a je možno ho přeskočit. Stejné okno pak slouží k prohlížení a načítání stávajících projektů či založení projektu nového. Po kliknutí na tlačítko *New* se zadává jméno projektu, cílová složka pro jeho uložení a vybírá se zde základní šablona pro projekt. Svůj projekt jsem pojmenoval *MalyPrinc* a jako výchozí šablonu jsem zvolil *2D*.

Jelikož je Unity3D multiplatformní engine, na začátku je nutné zvolit, pro jakou platformu bude hra určena. Toto nastavení se zobrazí kliknutím na *File* a zvolením *Build settings*. Zde je možné měnit širokou škálu nastavení celého projektu. Pro účely své práce jsem tedy zvolil platformu Android. Veškeré ostatní nastavení jsem prozatím nechal na jeho výchozích hodnotách.

Unity 3D nemá ve výchozím obsahu třídu, umožňující ovládání prostřednictvím mobilních telefonů. Je však možné tuto třídu, spolu s dalšími doplňky, zdarma stáhnou prostřednictvím *Asset store*. Celý tento balík assetů nese název *Unity Standard Assets* a jeho tvůrcem je přímo studio *Unity Technologies*. Po jeho vyhledání v *Asset store* stačí kliknout na tlačítko *Download* a doplněk se automaticky stáhne do složky projektu.

4.3.2 Postava hráče

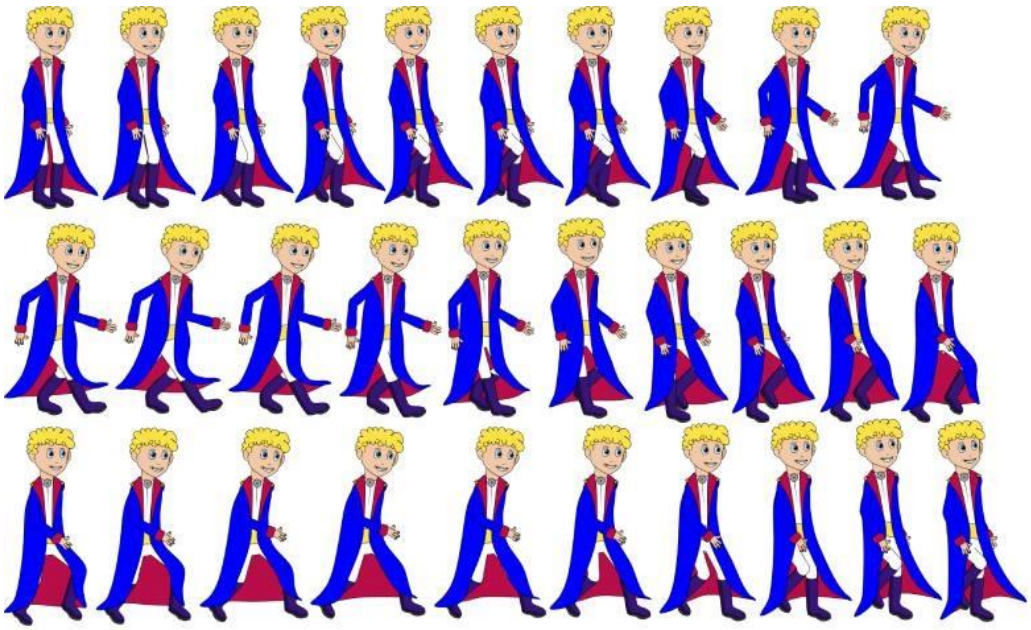
Jako první herní objekt jsem ve své práci vytvořil postavu hráče. Taková postava se skládá pouze z jednoho herního objektu. Tento objekt pak obsahuje několik C# tříd, které řídí jeho celkovou funkci.

4.3.2.1 Vzhled a animace postavy

Aby měla postava vzhled, stačí do projektu nahrát jakýkoli obrázek, který se následně přiřadí hernímu objektu postavy. Z toho důvodu je možné vytvořit postavu v široké škále grafických programů. Z důvodu dřívějších zkušeností s programem Moho Pro 12 jsem si zvolil právě tento program pro vytvoření postavy a její animace.

Postavu jsem v programu vytvořil pomocí běžných kreslících nástrojů Aby mohla být tato postava funkčně animovatelná, je nutné nekreslit ji jako celek, ale po jednotlivých částech, rozdělených do vrstev. Svou postavu jsem vytvořil pomocí 28 vrstev. Tyto vrstvy jsou rozděleny na části jako například levá a pravá dlaň, levé a pravé předloktí, hlava a podobně.

Program Moho disponuje funkcí *bone rigging*. Tato funkce zajišťuje vytvoření kostry pro postavu a následnou manipulaci s touto kostrou. Pro herní postavu jsem tedy vytvořil její kostru a přidělil jednotlivým kostem, které vrstvy (tedy části těla) budou ovládat. Pomocí toho jsem tak získal plně animovatelnou postavu. Na závěr jsem pomocí této funkce vytvořil animaci pohybu postavy, která je dlouhá třicet snímků. Každý z těchto snímků jsem exportoval zvlášť, jako obrázek ve formátu *.png* spolu se snímkem postavy ve výchozí poloze. V okně *Project*, uvnitř programu Unity jsem si vytvořil složku *Art* s podložkou *LittlePrince*. Do této složky jsem pak nahrál všech 31 obrázků malého prince.



Obrázek 12 - Animace pohybu hlavní postavy

4.3.2.2 Vytvoření herního objektu postavy

Nové herní objekty se vytvářejí prostřednictvím okna *Hierarchy*. Po kliknutí pravým tlačítkem myši se zde zobrazí celá škála navrhovaných herních objektů. Jelikož aplikaci tvořím ve dvourozměrném prostředí, postavu hráče jsem vytvořil jako objekt typu *Sprite*, který se nachází v záložce *2D*. Po kliknutí na tento objekt se v pravém okně *Inspector* zobrazí jeho vlastnosti a všechny přiřazené třídy. Herní objekt pro postavu hráče jsem vytvořil jako *Sprite*, a tudíž již automaticky obsahuje třídu *Sprite Renderer*. Tento objekt jsem pojmenoval *hrac*. Pro nastavení obrázku herního objektu tak stačilo nastavit pouze proměnnou *Sprite* v této třídě na obrázek malého prince, nahraný ve složce *Art*.

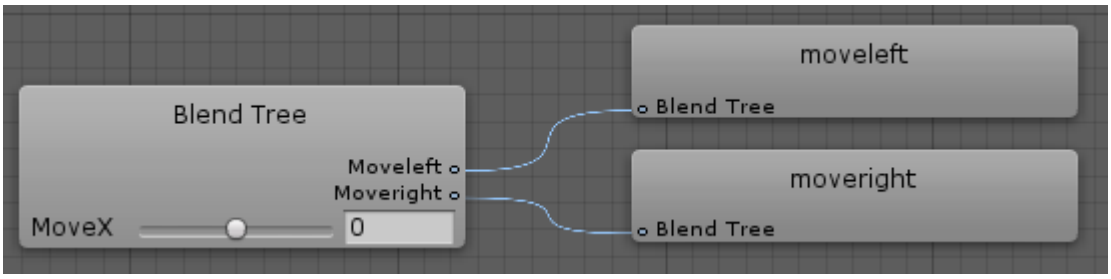
4.3.2.3 Animace postavy

Nová animace se vytváří v okně *Animation*, při čemž musí být rozkliknut herní objekt, pro který se bude animace vytvářet. V tomto případě jsem vytvářel animaci pro herní objekt postavy hráče. Novou animaci jsem pojmenoval *MoveRight*, jelikož se jedná o animaci chůze vpravo. Pro tuto animaci jsem nastavil vlastnost *Samples*, tedy počet snímků za vteřinu, na hodnotu 24. Pro vytvoření kompletní animace pak již stačilo pouze zapnout nahrávání animace a na každém snímku časové osy změnit vlastnost *Sprite* herního objektu na další obrázek v posloupnosti vytvořených obrázků chůze. Animaci pro chůzi vlevo jsem vytvořil obdobným způsobem. Aby však byla postava natočena správným směrem, po zapnutí nahrávání animace *MoveLeft* jsem potvrdil vlastnost *FlipX* ve *Sprite Renderer*, což všechny obrázky otočilo na druhou stranu.

Obrázek 13 - Časová osa animace pohybu

Když jsou všechny animace hotové, je nutné přejít do okna *Animator* a zde nastavit, za jakých podmínek se jednotlivé animace budou spouštět. Vytvořil jsem zde tedy dva parametry, *MoveX* a *PlayerMoving*. Mezi animacemi jsem pak nastavil přechody na základě právě těchto parametrů. Pokud se bude parametr *PlayerMoving* rovnat nule, přepne se animace na výchozí, prázdnou animaci a vzhled postavy se nastaví na výchozí obrázek. Za předpokladu, že bude tento parametr roven jedničce, spustí se rozhodovací strom *PlayerMovement* s parametrem *MoveX*. Pokud tento parametr bude roven jedné,

spustí se animace *MoveRight* a pokud bude roven mínus jedné, spustí se animace *MoveLeft*.



Obrázek 14 - Přepínání mezi směry pohybu

4.3.2.4 Ovládání postavy

Aby bylo možné měnit parametr *MoveX* z přechozí kapitoly a tím tak spouštět animace chůze, je nutné vytvořit první vlastní C# třídu. Vytvořil jsem si tedy v okně *Project* novou složku nazvanou *Scripts*. V Této složce jsem vytvořil pomocí pravého tlačítka na myši nový C# script nazvaný *PlayerController*. Tuto třídu jsem ihned přiřadil k hernímu objektu postavy malého prince. Na začátku jsem deklaroval nové proměnné. Jelikož tato třída bude využívat vstupy z dotykové obrazovky, pro které jsem stáhnul doplněk z *Asset store*, je nutné na začátku C# kódu definovat nový obor příkazů. To jsem provedl pomocí příkazu *using*, konkrétně „*using UnityStandardAssets.CrossPlatformInput;*“.

```
private Animator anim;
public bool playermoving;
private static bool playerExsist;
void Start()
{
anim = GetComponent<Animator>();
}
```

Mezi deklarovanými proměnnými je i proměnná *anim* třídy *Animator*. Této proměnné je nutné po ihned po startu přiradit správný animátor, čehož je docíleno metodou *GetComponent<Animator>()*. Ta vyhledá animátor v herním objektu, na kterém se tato metoda spouští. Jelikož je metoda součástí třídy *PlayerController*, která je přiřazena k hernímu objektu postavy malého prince, přiřadí se animátor právě tohoto herního objektu na proměnnou *anim*. Na základě tohoto přiřazení bude mít celá třída *PlayerController* přístup k parametrům *MoveX* a *PlayerMoving*, které ovládají animaci hráče. Následující kód má za úkol nastavovat tyto parametry dle aktuálního stavu ovládacích tlačítek.

```
Void Update()
{
  playermoving = false
  if (CrossPlatformInputManager.GetAxisRaw("Horizontal") != 0f)
  {
    playermoving = true;
  }
  anim.SetFloat
("MoveX",CrossPlatformInputManager.GetAxisRaw("Horizontal"));
  anim.SetBool("PlayerMoving", playermoving);
}
```

Kód je umístěn v metodě *Update*, která je volána při každém snímku, tedy nepřetržitě. Ve výchozím nastavení je proměnná *playermoving* nastavena na hodnotu *false,* pokud se však vstup na horizontální ose nebude rovnat nule, změní se tato proměnná na *true*, což značí, že třída zaznamenala stisknutí tlačítka *vlevo,* či *vpravo*. Poté se nastaví parametr *MoveX* z animátoru na hodnotu vstupu z horizontální osy a stejně tak se nastaví parametr *PlayerMoving* na hodnotu proměnné *playermoving*. Takto je postava připravena měnit animace v závislosti na jejím ovládání. Toto ovládání, řídící vstupy na horizontální osu, jsem však zatím nevytvořil a bude předmětem kapitoly uživatelského rozhraní.

Zároveň také musím zdůraznit, že na základě návrhu herní mechaniky postava nebude provádět žádný pohyb, a proto ovládání postavy zahrnuje pouze spouštění animací, nikoliv však změnu souřadnic herního objektu postavy. Vytvořením domnělého pohybu se zabývá kapitola *Rotace herního světa,* kde je pohyb postavy simulován rotací celého herního světa.

4.3.2.5 Skok

Skok, který jsem pro postavu vytvořil, plní z hlediska aplikace především estetickou funkci, která je využívána při cestování na jiné planety. Z hlediska praktické práce jako takové však především demonstruje použití fyzikálního enginu. Fyzika pro určitý herní objekt se v prostředí Unity aktivuje přiřazením třídy *RigidBody2D*, případně třídou *RigidBody*. Jelikož aplikace využívá pouze dvourozměrné prostředí, je nutné využití první zmíněné třídy.

Po přiřazení třídy *RigidBody2D* k herním objektu se tento objekt začne okamžitě řídit simulovanými zákony fyziky. V mém případě jsem tedy tuto třídu přiřadil k hernímu objektu hráčovi postavy. Při zapnutí hry je výsledkem nekonečný pád herního objektu směrem dolů, jelikož nemá přidělen žádný reálný fyzický tvar a zároveň se pod ním

nenachází žádný objekt, s podobným fyzickým tvarem, o který by se mohl pád zastavit. Nejedná se samozřejmě o reálný pád, ale o konstantní zmenšování hodnoty Y na souřadnicích daného objektu.

Fyzický tvar objekt získá přidáním třídy *Collider* libovolného tvaru. Pro svou aplikaci jsem zvolil třídu *BoxCollider2D*, která vytvoří collider obdélníkového tvaru o velikosti celé herní postavy. Tyto rozměry je samozřejmě možné editovat dle potřeby. Pomocí těchto kroků získala má postava základní fyzikální vlastnosti a fyzický tvar, detekující kolize s objekty.

Skok postavy se skládá se dvou částí, první části je změna pozice herního objektu postavy. Tyto souřadnice se nejdříve posunou nahoru a po krátké chvíli zase dolu. Druhou částí je vytvoření animace pro tento skok.

Ve chvíli, kdy je herní objekt postavy brán jako fyzikální objekt, změna pozice při skoku je poměrně jednoduchou záležitostí. Lze zde totiž využít metodu *AddForce*, která vymrští herní objekt vzhůru silou, která je parametrem této metody. Herní objekt tedy začne na určitou dobu stoupat vzhůru a poté opět padat, dokud nenarazí na další herní objekt, například zem. Ve své práci jsem použil následující kód, která je součástí třídy *PlayerController*.

```
private bool jumping;
public float jumpingTime;
private float jumpingTimeCounter;
public float jump;
if(CrossPlatformInputManager.GetAxisRaw("Jump")>0.1f&&!jumping)
  {
   jumpingTimeCounter = jumpingTime;
   jumping = true;
   myrigidbody.AddForce(new Vector2(0f, jump),
ForceMode2D.Impulse);
   anim.SetBool("Jump", true);
   }
if (jumpingTimeCounter >= 0)
   {
   jumpingTimeCounter -= Time.deltaTime;
   }
if (jumpingTimeCounter <= 0)
   {
   jumping = false;
   anim.SetBool("Jump", false);
   }
```

Na začátku tohoto kódu jsem deklaroval dvě veřejné a dvě privátní proměnné. Veřejná proměnná *jump* určuje sílu, která bude přidána do metody *AddForce*. Druhá veřejná proměnná, *jumpingTime* určuje časovou dobu trvání jednoho skoku. Privátní

proměnná *jumping* slouží k potvrzení, zda je v danou chvíli skok aktivní a poslední proměnná *jumpingTimeCounter* slouží jako odpočet doby určené v proměnné *jumpingTime*.

Pokud tedy třída *PlayerController* zaznamená na vstupu *Jump* (jako výchozí osy pro vstupy jsou vytvořeny *Horizontal* a *Vertical*, ale pro potřeby tvorby ovládání je možné nadefinovat nové osy vstupů s libovolnými názvy) a zároveň bude proměnná jumping nepravdivá, provede se série příkazů, zahajujících skok postavy. V první řadě se nastaví odpočítávání doby skoku na hodnotu proměnné *jumpingTime*. Proměnná jumping se nastaví po dobu tohoto odpočtu jako pravdivá. Metoda *AddForce* zapůsobí silou na fyzikální objekt herní postavy. Prvním parametrem metody je směr působení síly. Tato síla bude mít směr *x* roven nule a směr *y* bude roven proměnné *jump*. Druhým parametrem této metody je *ForceMode2D.Impulse*, který určuje, že působení síly je pouze jednorázové a ne konstantní. Posledním příkazem, provedeným po stisku vstupu *jump,* je nastavení parametru *Jump* z animátoru anim jako pravdivý, čímž se spustí animace skoku.

Tuto animaci jsem vytvořil stejným postupem, který je pospán v kapitole *Vzhled a animace postavy*. V animátoru herní postavy jsem pouze přidal nový přechod na nově vzniklou animaci *jump*. Tento přechod je podmíněn pravdivou hodnotou na zmíněném parametru *Jump*.

4.3.3 Herní prostředí

Herní svět lze vytvořit nespočetným množstvím způsobů. Jelikož se má aplikace odehrává na rozličných malých planetkách, prostředí každé scény se bude skládat z malé kulaté planety různé velikosti i vzhledu, obsahující různé herní objekty a postavy.

4.3.3.1 Planeta

Celé herní prostředí jsem vytvořil jako potomky jediného herního objektu *Planeta*. Tento objekt obsahuje pouze dvě třídy. První je *CircleCollider,* prostřednictvím něhož postava hráče zůstane stát na vrcholu planety, což zabrání jejímu nekonečnému pádu. Je však důležité, aby byla postava umístěna přesně nad středem planety, jinak fyzikální engine zajistí sklouznutí postavy z planety a její následný nekonečný pád. Druhou třídu, *RotateBackground,* jsem vytvořil sám. Tato třída zajišťuje rotaci planety a jejího pozadí při stisknutí tlačítka pohybu vlevo či vpravo, což simuluje pohyb herní postavy.

Prvním potomkem objektu *Planeta* je *Planeta1*. Tento herní objekt je typu *Sprite Renderer* bez jakýchkoli dalších tříd. Obrázek této planety jsem vytvořil v programu Moho. Z hlediska funkčnosti však postačí jakýkoli kruh ve formátu *.png*, a to z toho důvodu, aby obrázek obsahoval průhledné pozadí. Druhým potomkem je objekt *Background*, též typu *Sprite Renderer*. Jsou zde však dva rozdíly. Vrstva tohoto spritu je zde nastavena na hodnotu *background*, aby bylo pozadí vykresleno za planetou. Druhým rozdílem je vytvořená animace pro pozadí. Animace má na prvním snímku nastavenou rotaci pozadí podle osy Z na hodnotu 0. Na snímku 100 má tuto hodnotu nastavenou na 360. Výsledkem této animace je postupné otáčení pozadí. Spouští se automaticky po načtení scény a po dokončení animace začíná na svém začátku. Dále jsem herní svět doplnil o postavu růže a další grafické prvky, uložené pod nadřazenými herními objekty.

Obrázek 15 - Herní objekty planety

4.3.3.2 Rotace herního světa

Pro simulaci pohybu postavy jsem vytvořil již zmíněnou třídu *RotateBackground*. Tato třída má za úkol při stisknutí tlačítka vlevo či vpravo rotovat celým herním objektem *Planeta* podle osy Z.

```
public float speed;
Transform rectTransform;
Void Start()
{
rectTransform = GetComponent<Transform>();
}
void Update()
{

 if (CrossPlatformInputManager.GetAxisRaw("Horizontal") > 0.1)
 {
  rectTransform.Rotate(new Vector3(0, 0, speed));
 }
 if (CrossPlatformInputManager.GetAxisRaw("Horizontal") < -0.1)
 {
   rectTransform.Rotate(new Vector3(0, 0, -speed));
 }
}
```

Pro začátek jsem deklaroval veřejnou proměnnou *speed*. Tato proměnná určuje, jak rychle se bude celá planeta, včetně jejího pozadí, při pohybu hráče otáčet. Druhá deklarovaná proměnná je třídy *Transform* a nese název *rectTransform*. Tuto proměnnou je nezbytné ihned po aktivaci třídy *RotateBackground* naplnit, tedy určit, který konkrétní transform bude nabývat. Jelikož tato třída má za úkol otáčet herní objekt *Planet*, potřebuje třídu *Transform* právě z tohoto objektu. K jeho získání stačí metoda *GetComponent<Transform>*, jelikož třída *RotateBackground* je umístěna na stejném herním objektu, tedy *Planet*.

Pokud bude na vstupu horizontální osy hodnota vyšší než 0.1, což značí stisknutí tlačítka *vpravo*, při každém snímku se změní rotace planety kolem osy Z o hodnotu proměnné *speed*. Pokud bude hodnota na tomto vstupu nižší než -0.1, změní se rotace planety o zápornou hodnotu proměnné *speed*. Hodnotu této proměnné jsem v uživatelském rozhraní Unity změnil pro každou planetu rozdílně, neboť se planety lišší svou velikostí a chůze herní postavy by tam při příliš pomalém, či příliš rychlém otáčení nevypadala přirozeně. Vytvoření ovládání postavy, které je potřebné k otestování této třídy, popisuji v kapitole *Uživatelské rozhraní*.

4.3.4 Uživatelské rozhraní

Celé uživatelské rozhraní, které zahrnuje ovládání postavy, zobrazování dialogů a zobrazování inventáře jsem vytvořil pod jedním rodičovských herním objektem typu *Canvas*, nazvaným *UI*. Ve třídě *Canvas* u tohoto herního objektu jsem pak změnil

parametr *UI Scale Mode* na hodnotu *Scale with screen size*, což zajistí správné vykreslování uživatelského rozhraní na různě velkých obrazovkách. Ostatní nastavení jsem ponechal na výchozích hodnotách.

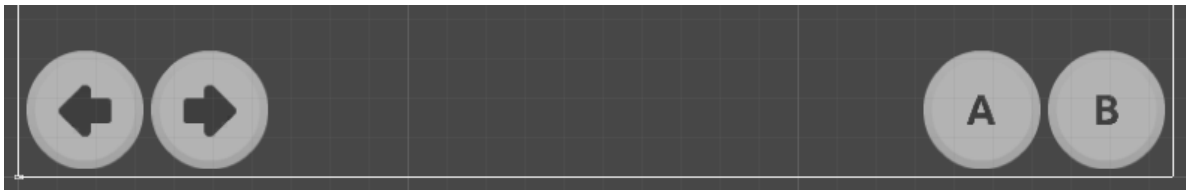
4.3.4.1 Rozhraní pro pohyb

Postava i rotace herního světa jsou již připraveny na ovládání pomocí uživatelského rozhraní. Oba tyto objekty jsou ve svých třídách předpřipraveny na vstupy osy s názvem *Horizontal*. Pro vytvoření takových vstupů jsem přidal pod rodičovský objekt *UI* dva nové herní objekty typu *Button*, pojmenované *vlevo* a *vpravo*. Objekty typu *Button* obsahuje po vytvoření dvě komponenty, *Image*, definující vzhled tlačítka a *Button*, definující vlastnosti samotného tlačítka.

Pro potřeby ovládání na dotykových obrazovkách je nutné přidat ještě třetí komponentu, staženou jako součást *UnityStandarAssets*, nesoucí název *Axis Touch Button*. Účelem této komponenty je snímání ovládacích prvků na dotykové obrazovce pomocí jednotlivých os. Dotykové ovládání nemusí nabývat totiž pouze hodnot 0 a 1, ale pomocí potahování na dotykové obrazovce může nabýt pouze poloviční hodnotu, 0.5.

Po přiřazení komponenty *Axis Touch Button* na obě tlačítka, *vlevo a vpravo*, jsem nastavil o obou tlačítek hodnotu *Axis value*, která je součástí právě zmiňované komponenty, na hodnotu *Horizontal*. Tím je dokončeno spojení mezi postavou, herním světem a ovládáním. Jediný rozdíl, mezi těmito dvěma tlačítky je parametr *Axis Value*, který udává, jaké hodnoty může dané tlačítko dosáhnou. V případě tlačítka *vlevo*, jsem hodnotu parametru nastavil na -1, v případě v*pravo* je pak na 1.

Na závěr jsem u obou tlačítek nastavil vzhled pomocí komponenty *Image* na obrázky šipek, odpovídajících směru ovládání a přesunul tyto tlačítka na vhodnou pozici herní obrazovky, konkrétně do levého spodního rohu. Stejný postup jsem uplatnil při tvorbě třetího a čtvrtého tlačítka, nazvaného *Jump* a *Sprint*. Jediný rozdíl je změna parametru *Axis Name* na hodnotu *Jump* a na hodnotu *Sprint* a jeho výsledné umístění, pro které jsem zvolil pravý spodní roh. Tlačítko pro skok má jako obrázek nastaven na tlačítko s písmenem „*A“* a tlačítko pro sprint má vzhled tlačítka s nápisem „*B“*. V *UI* jsem vytvořil ještě pátý herní objekt, nazvaný *Android Control*, do kterého jsem přesunul všechna čtyři výše zmíněná tlačítka, a to z důvodu přehlednosti a snadného animování ovládání (například skrývání při otevření dialogu či inventáře).

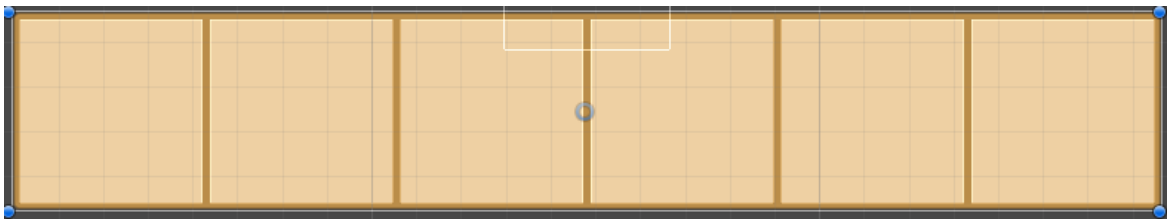


Obrázek 16 - Rozhraní pro ovládání postavy

4.3.4.2 Rozhraní inventáře

Jak jsem již nastínil v kapitole *Návrh*, inventář se bude skládat ze šesti míst pro případné předměty. V této kapitole se zabývám pouze vytvořením vzhledu inventáře a jeho zobrazováním, nikoli řízením jeho činnosti.

Pro potřeby inventáře jsem vytvořil prázdný herní objekt *Inventory*, umístěný v objektu *UI*. Uvnitř tohoto objektu jsem poté vytvořil sedm dalších herních objektů. První z nich, pojmenovaný *background*, je objekt typu *Image* a slouží jako pozadí inventáře. Dalších šest objektů, pojmenovaných *Item1* až *Item6* je typu *Button*. Tyto objekty budou sloužit pro zobrazování předmětů uložených v inventáři. Typ *Button* jsem zvolil pro zjednodušení interakcí předmětů, které budou vyvolávány právě kliknutím na dané tlačítko, jehož komponent *Image* ponese obrázek daného předmětu. Pozadí inventáře má tvar obdélníku, s šířkou téměř celé obrazové plochy a přibližně sedminásobně menší výškou. Objekty *Item1* až *Item6* pak mají tvar čtverce, přičemž jejich výška je téměř totožná s výškou pozadí. Rozmístěné jsou pak pravidelně v celé délce pozadí inventáře.



Obrázek 17 - Rozhraní inventáře

Pro otevření a zavření inventáře jsem vytvořil dvě nové animace pro herní objekt *UI*. První animace spočívá ve výchozím stavu zobrazení, tedy ovládání je viditelné a inventář skrytý. Animace se tak odehrává pouze na prvním snímku, kde je hodnota proměnné *enabled* u herního objektu *Inventory* nastavena na *false* a hodnota totožné proměnné u herního objektu *AndroidControl* má hodnotu *true*. Tuto animaci jsem nazval *InventoryHide*. Druhá vytvořená animace nese název *InventoryShow*. Tato animace má pouze opačné hodnoty *true* a *false* proměnných *enabled* u obou objektů. V animátoru jsem mezi těmito animacemi vytvořil přechody podmíněné parametrem *ShowInventory*. Pokud

je nastaven na *true,* zapne se animace *InventoryShow,* v opačném případě bude aktivní animace *InventoryHide*.

Pro ovládání inventáře jsem vytvořil následující třídu *inventoryControl,* kterou jsem umístil na herní objekt *UI*.

```
private bool inventoryOpen;
public Animator animatorInventory;
public void ShowInventory()
  {
    if (inventoryOpen == false)
      {
      animatorInventory.SetBool("ShowInventory", true);
      inventoryOpen = true;
      }
    else
      {
      animatorInventory.SetBool("ShowInventory", false);
      inventoryOpen = false;
      }
  }
```

Pro tento kód jsem deklaroval dvě proměnné. První z nich, proměnná *inventoryOpen,* která je datového typu bool, slouží jako pomocná proměnná určující, zda je inventář otevřený, či zavřený. Druhá proměnná je třídy *Animator* a slouží k nastavení konkrétního animátoru, který bude následně řízen. Jelikož je tato proměnná deklarovaná jako veřejná, je možné tento animátor přiřadit prostřednictvím uživatelského rozhraní unity. Do proměnné *animatorInventory*, umístěném v komponentě *InventoryControl* na herním objektu *UI* jsem tažením myši přesunul právě zmíněný herní objekt. Třída tak rozpozná, že řídí animátor herního objektu *UI*.

Pokud se pomocná proměnná *inventoryOpen* rovná hodnotě *false*, tedy že inventář zatím není otevřen, nastaví metoda parametr *ShowInventory* u animátoru na hodnotu *true* a zároveň nastaví *true* i u pomocné proměnné. Následná animace skryje herní ovládání a zobrazí inventář. Pokud je metoda vyvolána znova, pomocná proměnná je rovna *true* a tím pádem se hodnota parametru *ShowInventory* změní na *false* a stejně tak se změní hodnota pomocné proměnné. Tím pádem se skryje inventář a znovu zobrazí herní ovládání.

Tuto metodu je však nutné nějakým způsoben vyvolat. Z toho důvodu jsem uvnitř herního objektu *Android Control* vytvořil již páté tlačítko, umístěné do levého horního rohu. U tohoto tlačítka jsem využil, na rozdíl od těch předchozích, výchozí komponentu *Button*. Do události *OnClick()* tohoto tlačítka jsem přesunul herní objekt *UI*. Ze seznamu nabízených tříd jsem vybral třídu *InventoryControl* a ze seznamu jejích metod metodu *ShowInventory*. Tím jsem propojil tlačítko inventáře s funkcí jeho zobrazení a skrytí.

4.3.4.3 Rozhraní dialogů

V této kapitole se budu, obdobně jako v kapitole předchozí, věnovat pouze grafickému hledisku dialogových oken, a nikoliv jeho samotné funkčnosti. Na začátku jsem vytvořil uvnitř herního objektu *UI* prázdný herní objekt, nazvaný *DialogBox*. Uvnitř nově vzniklého objektu jsem vytvořil celkem sedm dalších objektů. Prvním objektem je *Panel*, jehož velikost jsem nastavil na automatické roztažení přes celou herní obrazovku. Panelu jsem nastavil bílou barvu a název *Background*. Jak již název napovídá, panel slouží jako pozadí při průběhu dialogů.

Druhý a třetí objekt je typu *Image*. Velikost každého z těchto objektů je přibližně 80 % výšky a 20 % šířky obrazovky, přičemž první z těchto objektů jsem pojmenoval *Malý princ* a přesunul ho do levé části obrazovky. Druhý jsem pak pojmenoval *Postava* a přesunul ho do pravé části obrazovky. Tyto objekty složí pro promítání postav, mezi kterými je veden dialog. První z nich je již konkretizován názvem malého prince, neboť dialog vždy bude probíhat mezi princem a druhou postavou. Z toho důvodu jsem také již nastavil obrázek tohoto objektu na výchozí postavu malého prince. U objektu druhé postavy jsem z důvodu testování vzhledu nastavil obrázek postavy krále, kterou jsem vytvořil v programu Moho.

Čtvrtý herní objekt, který jsem pojmenoval *DiagBackground,* je typu *Image*, zabírá přibližně 20 % spodní části obrazu a je určen jako pozadí pro zobrazovaný text. Další dva herní objekty, pojmenované *Name* a *Text* jsou typu *Text*. První je určen na zobrazování jména aktuálně hovořící postavy, druhý pak zobrazuje text aktuálního dialogu. Posledním objektem je pak *Button*, pojmenovaný *Continue*. Toto tlačítko bude v kapitole *Dialogy* využito k přepínání dalších fází dialogu.

Obrázek 18 - Herní objekty dialogového rozhraní

Zobrazování dialogového okna jsem vyřešil totožným způsobem, jako u zobrazování inventáře, tedy dvě animace, kdy jedna, výchozí animace zobrazuje ovládací rozhraní a dialog je skrytý. Druhá animace pak naopak ovládací rozhraní skryje a zobrazí dialog. Volání této metody pro zobrazení či skrytí dialogu však, na rozdíl od inventáře, není provedeno pomocí tlačítka ale odlišným způsobem, kterému se věnuji v kapitole Dialogy.

4.3.4.4 Rozhraní kvízů

Uživatelské rozhraní pro znalostní kvízy bude jedinou výjimkou v kapitole uživatelského rozhraní, neboť jako jediné nebude součástí objektu *UI* a nebude se tak přesouvat mezi scénami. Naopak každá scéna bude mít svůj vlastní herní objekt *Quiz* s unikátními otázkami, vztaženými k ději v aktuální scéně.

Jelikož se jedná o součást uživatelského rozhraní, herní objekt jsem vytvořil samozřejmě jako typ *canvas* s roztažitelností přes celou herní obrazovku. V tomto objektu jsem vytvořil objekt typu *panel*, který slouží jako neprůhledné pozadí celého kvízu, a nastavil jsem jeho barvu na modrou. Další objekty jsem pak vytvořil jako podřazené právě tomuto panelu.

Prvním z objektů je objekt typu *Image*, nazvaný *Question*. Samotný tento objekt slouží pouze jako pozadí pro zadané otázky, a proto jsem pod ním vytvořil ještě další objekt typu *text*, který bude otázky zobrazovat. Dále jsem pod objektem *panel* vytvořil čtyři tlačítka, pojmenovaná postupně *asnwer0* až *answer3*, která budou sloužit jako možnosti pro odpověď. Jednotlivé možnosti odpovědí budou vepsány v parametrech *text* těchto tlačítek.

Obrázek 19 - Rozhraní pro testovací kvízy

4.3.5 Načítání scén

Pro potřeby své aplikace jsem vytvořil čtyři unikátní scény. Každá z nich představuje jednu planetu, kterou může hráč navštívit Všechny tyto planety jsem vytvořil stejným způsobem, jaký jsem popisoval v kapitole týkající se herního prostředí. Rozdíly mezi těmito scénami jsou tak pouze ve vzhledu jednotlivých herních objektů a v jejich celkovém množství. Jediným parametrem, rozdílným na těchto planetách je *RotateBackground*, neboť má každá planeta rozdílné rozměry a tím pádem i rychlost její rotace. V této kapitole se věnuji vytvoření způsobu přechodu mezi těmito scénami.

4.3.5.1 Třída pro načítání

Aby bylo možné načítání nových scén realizovat, vytvořil jsem třídu *LoadLevel* jejíž funkcí je nejen načtení jiných scén, ale také ověření, zda se postava hráče nachází na konkrétním místem, které je určeno pro cestu do další scény. První část kódu, obsahující deklaraci proměnných a ověření, zda hráč stojí u objektu pro cestování má následující podobu.

```
public string levelToload;
public Image loadingImage;
private bool inRange;

private void OnTriggerEnter2D(Collider2D other)
  {
   if (other.gameObject.name == "hrac")
     {
     inRange = true;
     }
  }
private void OnTriggerExit2D(Collider2D other)
  {
   if (other.gameObject.name == "hrac")
     {
     inRange = false;
     }
  }
```

Před deklarací proměnných jsem přidal na začátek této třídy tři nové *using*. První z nich, *UnityStandardAssets.CrossPlatformInput* a to z důvodu, že po načtení nové scény bude nutné, vynulovat hodnoty na vstupních osách *Horizontal* a *Jump*. Druhý *using*, *UnityEngine.SceneManagement*, přidá do třídy nový obor příkazů, které pracují právě se scénami, jako například metoda *LoadScene*. Posledním přidaným *using* je *UnityEngine.UI*, který přidává příkazy pracující s canvas objekty. Pomocí canvas vytvořím obrázek, zobrazující se při načítání nové scény.

První proměnnou, kterou jsem v této třídě deklaroval je třída *levelToLoad*. Hodnota této proměnné ponese název scény, která se má načíst. Proměnná třídy *Image*, nazvaná *loadingImage* bude sloužit k odkázání na obrázek, který se bude zobrazovat vždy při načítání scény. Poslední proměnná je typu bool a nese název *inRange*. Ta bude sloužit k informování, zda se hráč nachází na místě určeném pro cestování.

Zbylá část zmíněného kódu složí právě k rozpoznání, kde se hráč nachází v blízkosti herního objektu, jehož komponentou je právě tato třída *LoadLevel*. Tento objekt má kromě zmíněné třídy ještě komponentu *BoxCollider* s nastavenou hodnotou *Is Triggerer* na hodnotu *true*, což z tohoto neviditelného objektu vytváří spouštěč. Objekt jsem pojmenoval *Loading*. Metoda *OnTriggererEnter2D* se vyvolá ve chvíli, kdy kterýkoli jiný herní objekt, nesoucí collider, protne objekt *Loading*. Pokud se tento objekt jmenuje *hrac*, nastaví se hodnota proměnné *inRande* na *true*. Metoda *OnTriggererExit2D* se pak vyvolá ve chvíli, kdy se tyto dva herní objekty přestanou protínat. V tu chvíli se hodnota proměnné *inRange* znovu přenastaví na *false*. Další část kódu pak zajišťuje samotné načtení nové scény.

```
void Update()
  {
  if (inRange == true)
    {
    CrossPlatformInputManager.SetAxisZero("Horizontal");
    CrossPlatformInputManager.SetAxisZero("Jump");
    inRange = false;
    loadingImage.enabled = true;
    SceneManager.LoadSceneAsync(levelToload);
}}
```

Pokud se proměnná *inRange* rovná hodnotě *true*, vykoná se série příkazů zajišťujících načtení nové scény. Nejdříve se vynulují hodnoty vstupů na osách *Horizontal* a *Jump* prostřednictvím metody *SetAxisZero*. Hodnota proměnné *inRange* se nastaví zpět na hodnotu *false*. Poté se přes celou herní obrazovku zobrazí obrázek *loadingImage*, značící, že právě probíhá načítání a metoda *LoadSceneAsync* načte scénu, uvedenou v proměnné *levelToload*. Jelikož je tato proměnná deklarována jako veřejná, lze její hodnotu nastavit přímo v rozhraní *Unity* u každého herního objektu s třídou *LoadLevel* zvlášť.

V praxi to tedy funguje tak, že pokud se hráčova postava dotkne neviditelného objektu s třídou *LoadLevel*, načte se další scéna. Ve své práci jsem tyto načítací objekty umístil na každé scéně výše, než postava při chůzi dosáhne, a proto musí hráč použít klávesu pro skok, aby na objekt dosáhl. Jelikož jsou tyto objekty neviditelné, umístil jsem ke každému z nich herní objekt typu *Sprite Renderer*, nesoucí obrázek planety, na kterou hráč bude cestovat.

4.3.6 Inventář a předměty

Vzhled inventáře a jeho zobrazování jsem vytvořil již v kapitole *Rozhraní inventáře*. Unity Engine však neobsahuje žádný typ herních objektů, které by mohly samy o sobě reprezentovat předměty. Z toho důvodu se v této kapitole nebudu věnovat pouze vytvoření třídy pro řízení činnosti inventáře, ale zároveň vytvoření nového typu herního objektu.

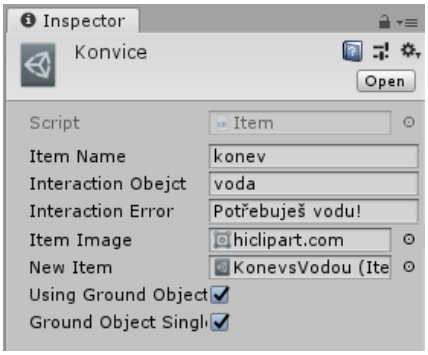
4.3.6.1 Předměty

Nový typ herních objektů lze definovat pomocí jazyka C# jakožto třídu. Vytvořil jsem tedy novou třídu s názvem *Item*. Tato třída neobsahuje žádné metody, ale pouze deklaraci několika proměnný, které budou definovat nové předměty.

```
[CreateAssetMenu(fileName="New Item",menuName = "Assets/Item")]
public class Item : ScriptableObject
{
    public string itemName;
    public string interactionObejct;
    public string interactionError;
    public Sprite itemImage;
    public Item newItem;
    public bool usingGroundObject;
    public bool groundObjectSingle;
    private bool itemCount;
}
```

První řádek kódu přidá možnost pro vytváření nových instancí tohoto herního objektu pomocí pravého kliknutí myši uvnitř průzkumníku projektu. Proměnná *fileName* zde definuje, jak se bude každý takový nově vytvořený předmět výchoze jmenovat. Druhá proměnná pak definuje cestu a název tohoto nově vytvářeného herního objektu.

Dále jsem vytvořil osm proměnných, které každý herní předmět definují. Proměnná *itemName* značí jméno předmětu, *interactionObject* nese název jiného předmětu, který je potřebný k interakci s daným předmětem, *interactionError* je proměnná pro zprávu, která se hráči zobrazí v případě, že se interakce nepodaří, například z důvodu, že hráč nemá druhý potřebný předmět. Proměnná typu *Sprite* obsahuje obrázek předmětu, tedy jeho ikonu a zároveň jeho reálný vzhled. Proměnná typu *Item* (což je objekt, který právě popisuji), nesoucí název *newItem* v sobě bude mít uložen jiný potencionální předmět, který vznikne interakcí dvou předmětů. Pokud například hráč bude muset vyrobit zástěnu pro růži a ta se vyrábí spojením kladiva a dřeva, předmět dřevo, i předmět kladivo budou mít v této proměnné odkaz na předmět zástěny. Proměnná typu bool, pojmenovaná *usingGroundObject* uvádí, zda předmět uložený v inventáři potřebuje k interakci nějaký vnější objekt, umístěný ve scéně. Pokud ne, znamená to, že interakce vzniká mezi dvěma předměty, uloženými v inventáři. *ItemCound* je proměnná, kterou využívám pro kontrolní počet předmětů v inventáři. Poslední proměnná *groundObjectSingle* označuje, zda použití předmětu, jehož interakce je nastavena na nějaký vnější objekt (*usingGroundObject*se rovná *true*), vytvoří předmět nový, či spustí nějakou akci, například animaci.



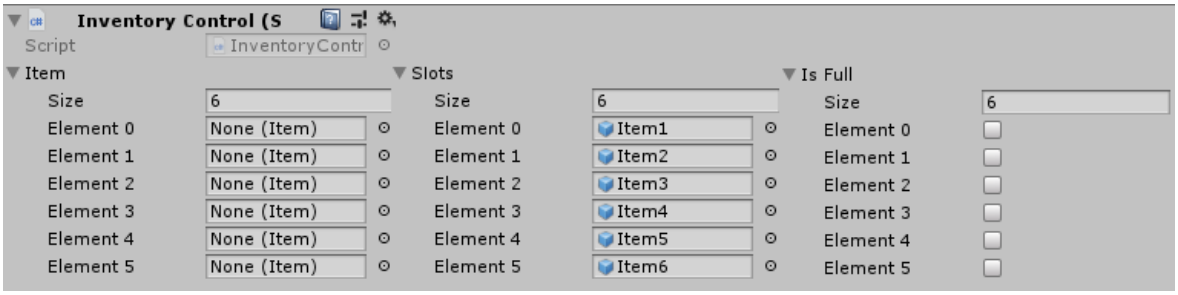
Obrázek 20 - Proměnné předmětu *Konvice*

S pomocí tohoto nového typu herních objektů jsem každý nový předmět vytvořil pravým kliknutím myši v okně *Project*, zde jsem vybral záložku *Assets* a objekt *Item*. Poté jsem každý předmět přejmenoval a v okně *Inspector* nastavil hodnoty všech jeho předmětu tak, aby odpovídaly jeho funkci.

4.3.6.2 Dokončení inventáře

Inventář jsem vytvořil již v kapitole *Rozhraní inventáře,* ale v současné chvíli nedokáže obsahovat žádný předmět, neboť se skládá pouze z prázdných tlačítek. Aby bylo do inventáře možné ukládat předměty, vytvořil jsem třídu *InventoryControl*, která kromě metod souvisejících s ukládáním a interakcí bude obsahovat pole proměnných, které budou značit uložené předměty. Tuto třídu jsem přidal jako komponentu herního objektu *InventoryControl*, který je součástí objektu *UI*.

Na začátku této třídy jsem tedy deklaroval několik proměnných a polí, do kterých bude vpisován každý předmět, uložený v inventáři. První pole veřejných proměnných typu *Item*, které jsem pojmenoval *item*, slouží k ukládání celých herních objektů typu *Item* v kompletní podobě, ve které jsem je vytvářel v minulé kapitole. Velikost tohoto pole jsem nastavil na 6 elementů, jelikož do inventáře bude možno uložit maximálně šest předmětů. Druhé pole proměnných je obecného typu *GameObject*. Pojmenoval jsem ho *slots* a nastavil jeho velikost na šest elementů. Pole je deklarováno jako veřejné, a proto je k němu možné přistupovat přímo v programu Unity. Do elementu jedna až šest jsem tedy prostřednictvím Unity přesunul herní objekty *Item1* až *Item6*, které jsou součástí objektu *Inventory* v herním objektu *UI*. Posledním použité pole je typu proměnných bool. Nazval jsem ho *isFull* a určuje, zda je místo v inventáři obsazené, či nikoli. Výsledkem těchto tří polí je možné udržovat předměty v inventáři.



Obrázek 21 - Proměnné třídy *Inventory Control*

4.3.6.3 Uložení předmětu do inventáře

Pro uložení předmětu je zapotřebí dvou tříd. Jednu, která má za úkol řízení inventáře, jsem již vytvořil v předchozí kapitole a pojmenoval *InventoryControl*. V této třídě vytvořím metodu *AddToInventory* se vstupním parametrem *Item,* která při zavolání uloží do inventáře předmět uložený v parametru. Druhá třída bude komponentou každého předmětu, nacházejícího se v prostředí herního světa. Jejím úkolem bude rozpoznání, zda se hráč nachází v blízkosti konkrétního předmětu, a v případě, že zmáčkne tlačítko pro interaktivitu, vyvolá tato třída metodu *AddToInventory* a jako parametr pošle předmět, se kterým aktuálně hráč provádí interakci. Kód metody *AddToInvetory* jsem vytvořil následovně:

```
public void AddToInvetory(Item itm)
 {
  for (int i = 0; i < slots.Length; i++)
  {
    if (isFull[i] == false)
    {
    item[i] = itm;
    isFull[i] = true;
    slots[i].GetComponent<Image>().sprite = itm.itemImage;
    itemCount = itemCount + 1;
    break;
    }
  }
 }
```

Jak jsem již zmínil výše, metoda má vstupní parametr typu *Item* a jeho název je *itm*. Metoda začíná cyklem, jehož počet iterací je roven počtu míst v inventáři. Cyklus tedy obsahuje šest iterací. Při každé iteraci hledá element pole *IsFull*, který je roven *false*. Poté, co je takové místo na pozici *i* nalezeno, nastaví se proměnná v poli *item* na pozici *i* na hodnotu vstupujícího parametru *itm*. V poli *IsFull* se hodnota na pozici *i* změní na *true* a v poli *slots* se hernímu objektu na pozici *i* nastaví *sprite* komponenty *Image* na obrázek vstupního parametri *itm*. Na závěr se navýší kontrolní počet předmětů v inventáři o jeden.

Po provedení těchto příkazů je průběh cyklu přerušen a již nedojde k dalším iteracím. Jinými slovy, metoda nalezne první prázdné místo v inventáři, zaplní ho předmětem, který metodu vyvolal a toto místo označí za obsazené. Na závěr nastaví obrázek místa na obrázek vstupujícího předmětu a přeruší cyklus.

Aby mohla být metoda volána, vytvořil jsem třídu *ObjectInteraction*, která bude součástí každého herního objektu, představujícího předměty. Tato třída bude ovládána tlačítkem *Sprint,* vytvořeným v kapitole *Uživatelské rozhraní*. Třída nejdříve ověřuje, zda se hráč nachází v blízkosti daného herního objektu. Tento objekt má jako svou komponentu kromě této třídy ještě collider v libovolném tvaru a velikosti.

```
private InventoryControl inventory;
public Item item;
private bool interaction;
private void OnTriggerEnter2D(Collider2D other)
  {
    if (other.gameObject.name == "hrac")
    {
    interaction = true;
    }
  }
private void OnTriggerExit2D(Collider2D other)
  {
    if (other.gameObject.name == "hrac")
    {
    interaction = false;
  } }
```

Pro rozpoznání, zda se hráč nachází v blízkosti objektu slouží pomocná proměnná *interaction*. Pokud se protne collider herního objektu spolu s colliderem hráče, nastaví se tato proměnná na *true*. Pokud se toto protnutí zruší, přenastaví se zpět na *false*. Dále jsem zde deklaroval dvě další proměnné. První, typu *InventoryControl*, odkazuje na herní objekt, jehož komponentou je právě *InventoryControl*. Tím objektem je samozřejmě inventář. Druhá proměnná, *item,* odkazuje na konkrétní předmět, který bude právě psanou třídou řízen. Jelikož je proměnná nastavena jako veřejná, lze tento předmět přesunou do proměnné uvnitř uživatelského rozhraní Unity.

```
void Update()
  {
  if (CrossPlatformInputManager.GetAxisRaw("Sprint") > 0.1f &&
interaction == true)
    {
    if (inventory.itemCount == 6)
      {
      FindObjectOfType<MessageBox>().ShowTimeMessage("Plný
inventář!");
      }
    else
      {
      inventory.AddToInvetory(item);
      gameObject.SetActive(false);
      interaction = false;
      }
    }
  }
```

K provedení výše zmíněného kódu jsem musel opět přidat nový using, konkrétně *UnityStandardAssets.CrossPlatformInput*, kvůli snímání vstupu s osou *Jumping*. Pokud bude vstup na této ose vyšší než 0.1 a zároveň bude pomocná proměnná *interaction* rovna *true,* algoritmus pokročí dále. Pokud se bude kontrolní součet předmětů rovnat šesti, vypíše se varovná hláška upozorňující na plný inventář. V ostatních případech bude zavolána metoda *AddToInventory* se vstupním parametrem *item*. Zároveň se nastaví vlastnost *enabled* u aktuálního herního objektu, tedy předmětu nacházejícím se v herním světě, na hodnotu *false*, čímž dojde z vypnutí objektu, včetně jeho vzhledu i collideru. Objekt tak již nebude dále ve hře dostupný, jelikož se předmět stal součástí inventáře.

4.3.7 Dialogy

Rozhraní dialogů je v základu tvořeno ze dvou animovaných postav, mezi kterými probíhá dialog a textovým polem, zobrazujícím aktuální část dialogu. Řízení dialogů tedy bude obsahovat několik funkcí. První z nich dialog vyvolá, zobrazí UI dialogu a nastaví obrázky postav dle toho, které postavy se právě chystají hovořit. Druhá funkce bude převádět texty a jména z proměnných, ve kterých jsou napsány a uloženy, do textového pole. Poslední z hlavních funkcí bude přepínání mezi animacemi postav dle aktuálního jména postavy v textovém poli. Tedy postava, která bude aktuálně dle textového pole hovořit, bude hovořit i prostřednictvím přehrávání animace řeči. Všechny tyto funkce budou implementovány prostřednictvím tří tříd, *Dialog, DialogManager* a *DialogTriggerer.*

4.3.7.1 Základní dialogová třída

Tato třída sama o sobě nemá žádné metody. Jedná se o třídu, která definuje všechny proměnné, potřebné pro konkrétní dialogy. Třída nebude používána jako komponenta herních objektů, ale jako pole proměnných v jiných třídách. Aby bylo možné využívat třídu pouze jako základnu proměnných a nikoli jako celou komponentu, musí být taková třída serializovatelná. Toho lze docílit příkazem *System.Serializable*, umístěným před deklarací samotné třídy.

```
[System.Serializable]
public class Dialog {
    public string dialogName;
    public string[] names;
    [TextArea(1, 10)]
    public string[] sentences;
}
```

První proměnnou typu string jsem nazval *dialogName*. Tato proměnná určuje pouze název dialogu jako celku a do jeho průběhu nezasahuje. Druhá proměnná, deklarovaná jako pole stringů, je nazvána *names*. Jak již název napovídá, toto pole bude obsahovat jednotlivá jména postav, které v konkrétním dialogu hovoří. Ve většině případů se tak v poli budou střídat vždy pouze dvě jména. Před poslední proměnnou jsem využil příkaz *TextArea* s parametry 1 a 10. Tento příkaz zaručuje, že následná proměnná bude mít v prostředí Unity zvětšené pole pro zapisování. První parametr udává minimální a výchozí počet řádků. Druhý parametr určuje maximální počet řádků textového pole hodnoty proměnné. Poslední proměnnou, která bude mít právě kvůli *TextArea* zvětšena textová pole, je pole proměnných typu string *sentences*. Pole bude obsahovat jednotlivé věty rozhovoru. Stejná pozice v polích *names* a *sentences* musí vždy obsahovat správné páry. Například pokud je na druhé pozici v poli *names* uvedeno jméno „Malý princ“, musí být na druhé pozici v poli *sentences* uvedena věta, která opravdu patří k dialogu prince.

4.3.7.2 Vytvoření a vyvolání dialogu

Dialog hráč otevře stisknutím tlačítka pro interakci. Dialog však bude otevřen pouze za předpokladu, že se hráč nachází v blízkosti postavy, která je k dialogu určena. Pro vyvolání dialogu jsem tedy vytvořil třídu nazvanou *DialogTriggerer*. Tuto třídu jsem přiřadil jako komponentu je všem postavám, se kterými hráč provádí dialogy. Konkrétně se tedy jedná o postavy: pilot, růže, lampář a král.

```
public Dialog dialog;
private bool inRange;
private void OnTriggerEnter2D(Collider2D other)
  {
  if (other.gameObject.name == "hrac")
    {
    inRange = true;
    }
  }
private void OnTriggerExit2D(Collider2D other)
  {
  if (other.gameObject.name == "hrac")
    {
    inRange = false;
    }
  }
```

Na začátku třídy jsem deklaroval proměnnou typu *Dialog*. Jelikož je *Dialog* serializovatelná třída, promítnou se všechny její proměnné, včetně možnosti jejich editace, do třídy *DialogTriggerer*. Druhou proměnnou je pomocná proměnná typu bool s názvem *inRange*, která určuje, zda je hráč v dosahu dialogové postavy. Třída na začátku pomocí metod *OnTriggererEnter* a *OnTrigererExit* ověři, zda se hráč nachází v blízkosti herních postav. Aby tyto dvě metody fungovaly správně, herní postava musí, kromě komponenty *DialogTrigger,* obsahovat ještě komponentu *BoxCollider* se zapnutou funkcí *IsTriggerer*. Pokud je tedy hráč v dosahu, proměnná *inRange* je rovna hodnotě *true*. V případě že není v dosahu, proměnná je ve výchozí hodnotě *false*.

```
void update
{
if (CrossPlatformInputManager.GetAxisRaw("Sprint") > 0.1f &&
inRange == true)
 {
FindObjectOfType<DialogManager>().StartDialog(dialog);
}}
```

Hráč dialog zahájí stiskem interakčního tlačítka a pokud je v blízkosti herního objektu dialogové postavy, dialog je opravdu vyvolán. To je ověřeno pomocí podmínky, kdy je vstup na ose *Sprint* vyšší než hodnota 0.1 a zároveň je proměnná *inRange* rovna *true*. Jelikož je součástí podmínky ovládání pomocí *CrossPlatformInputManager*, je nutné na začátek třídy přidat nový using, konkrétně *UnityStandardAssets.CrossPlatformInput*. Pokud je obojí splněno, je vyhledána pomocí *FindObjectOfType<DialogManager>* třída, respektive herní objekt obsahující třídu, *DialogManager* a vyvolána její metoda *StartDialog* se vstupním parametrem *dialog*, což je vlastně kompletní soustava jmen a vět nastavených u herního objektu v komponentě *DialogTriggerer*. Tyto soustavy dialogů jsem vytvořil každé herní postavě na základě knižní předlohy malého prince. Dialogy jsem

pochopitelně zkracovat do co nejmenší možné míry tak, aby byla zachována hlavní myšlenka rozhovoru.

4.3.7.3 Začátek dialogu

Pro řízení chodu dialogů, což zahrnuje změny textových polí a změny animací, jsem vytvořil novou třídu *DialogManager*. Tuto třídu jsem nastavil jako komponentu herního objektu *DialogBox*, který je součástí uživatelského rozhraní. Tento objekt jsem vytvořil v kapitole *Rozhraní dialogů*. Tato třída bude pracovat s prvky uživatelského rozhraní, a proto jsem na jejím začátku přidal nový using *UnityEngine.SceneManagement*.

```
using UnityEngine.SceneManagement;

public class DialogManager : MonoBehaviour
{
    public Text nameDiag;
    public Text textDiag;
    public Animator animatorDiag;
    private Queue<string> diagSentences;
    private Queue<string> diagNames;
    private string[] dialogPhaseName;
    private string[] dialogPhaseSentence;
```

Na začátku třídy *DialogManager* jsem kromě již zmíněného usingu deklaroval několik nových proměnných. První dvě veřejné proměnné typu *Text* slouží jako odkazy na textová pole, která se budou v proběhu dialogu aktualizovat. Proměnná *nameDiag* odkazuje na herní objekt textového typu s názvem *Name*, který je součástí rozhraní dialogů. Proměnná *textDiag* pak podobně odkazuje na objekt *Name*. Odkazy jsem vytvořil v rozhraní Unity přetažením objektů do proměnných. Třetí veřejná proměnná odkazuje na komponentu animátor, u která poté bude třída řídit animace. Animace pro dialogy jsem vytvářel na herním objektu *DialogBox*, takže jsem tento objekt prostřednictvím Unity přesunul do proměnné *animatorDiag*. Další dvě deklarované proměnné jsou typu *Queue*, tedy fronty, s datovým typem string. Tyto proměnné se naplní vstupním texty a jmény ze vstupního parametru *dialog* metody *StartDialog* a budou postupně přeskakovat mezi jednotlivými fázemi dialogu. Poslední dvě pole proměnných slouží k podobnému účelu jako předešlé fronty. Tato pole převezmou celý obsah vstupního textu a vstupních jmen a budou tento obsah postupně zapracovávat do front. Po deklaraci proměnných jsem vytvořil metodu *StartDialog* se vstupním parametrem *dialog*.

```
public void StartDialog(Dialog dialog)
  {
  animatorDiag.SetBool("IsOpen", true);
  dialogPhaseName = dialog.names;
  dialogPhaseSentence = dialog.senteces;
  diagNames.Clear();
  diagSentences.Clear();

  foreach (string name in dialogPhaseName)
   {
    diagNames.Enqueue(name);
   }
  foreach (string sentence in dialogPhaseSentence)
   {
    diagSentences.Enqueue(sentence);
   }
```

Kód výše představuje první část metody *StartDialog*. Do metody vstupuje kompletní instance třídy *Dialog*, tedy název dialogu, určité množství jmen postav a totožné množství dialogových vět. Metoda nastavením parametru *IsOpen* u animátoru *animatorDiag* na hodnotu *true* spustí animaci dialogového rozhraní, kterou jsem vytvářel v kapitole *Rozhraní dialogů*, čímž dojde ke skrytí ovládacího rozhraní a zobrazení rozhraní dialogového. Poté se nastaví pole *dialogPhaseName* a *dialogPhaseSentence* na hodnoty vstupující do proměnné. Fronty, které jsem deklaroval na začátku se po startu kompletně vyčistí od předchozích dialogů pomocí metody *Clear*. Pomocí dvou cyklů *foreach* se z výše zmíněných polí postupně přesunou všechna jména a všechny věty do připravených očištěných front pod proměnné *name* a *sentence*.

```
  DisplayNextName();
  DisplayNextSentence();

  switch (dialog.dialogName)
  {
   case "rose":
     animatorDiag.SetBool("roseDiag", true);
    break;
   case "pilot":
     animatorDiag.SetBool("roseDiag", true);
    break;
   case "lamp":
     animatorDiag.SetBool("lampDiag", true);
    break;
   case "king" :
     animatorDiag.SetBool("kingDiag", true);
    break;
  }
}
```

Zbylá část metody *StartDialog* vyvolá metody *DisplayNextName* a *DisplayNextSentence*, jejichž úkolem je změna textu v rozhraní. Jelikož ale metoda *StartDialog* volá nový dialog, nejedná se v tomto případě o změnu textu, ale spíše jeho prvotní nastavení. Na závěr metody jsem přidal switch, jehož cílem je nastavení spuštění výchozí animace dialogu. Tedy nastavení obrázků dialogových postav na konkrétní účastníky rozhovoru. Pro tuto funkci jsem vytvořil čtyři animace, které mají na prvním snímku nastaven sprite pravé postavy na konkrétní postavu dané animace. Přepínání mezi nimi je pak vedeno vždy z výchozí prázdné animace na základě parametrů zmíněných v kódu výše.

4.3.7.4 Průběh dialogu

Samotný průběh celého dialogu po jeho zobrazení budou vést již zmíněné metody *DisplayNextName* a *DisplayNextSentence*, spolu s třídou *EndDialogue*.

```
public void DisplayNextSentence()
 {
  if (diagSentences.Count == 0)
    {
     EndDialogue();
     return;
    }
   string sentence = diagSentences.Dequeue();
   textDiag.text = sentence;
 }
```

První poměrně jednoduchou metodou je *DisplayNextSentence*. Metoda ověřuje, zda se pořád nacházejí nějaké prvky ve frontě dialogových vět a pokud ne, tedy pokud se počet prvků ve frontě rovná nule, vyvolá metodu *EndDialogue* a přeruší výkon celé metody pomocí *return*. Pokud však ve frontě ještě zbývají nějaké věty, nastaví proměnnou *sentence* na hodnotu první věty v pořadí fronty a poté změní vlastnost *text* u herního objektu *textDiag* na hodnotu právě zmíněné proměnné *sentence*.

Metoda *DisplayNextName* pracuje v základu na úplně totožném principu. Jediným rozdílem je, že metoda zapíná také animace mluvy dle aktuálního jména, uvedeného v proměnné *name*. Její součástí je tedy, kromě kódu téměř totožného s metodou *DisplayNextSentece* následující switch.

```
switch (name)
{
  case "Malý princ":
      animatorDiag.SetBool("roseTalking", false);
      animatorDiag.SetBool("lampTalking", false);
      animatorDiag.SetBool("kingTalking", false);
      animatorDiag.SetBool("pilotTalking", false);
      animatorDiag.SetBool("lamparTalking", false);
      animatorDiag.SetBool("princeTalking", true);
    break;
  case "Růže":
      StopPrince();
      animatorDiag.SetBool("roseTalking", true);
    break;
  case "Lampář":
      StopPrince();
      animatorDiag.SetBool("lamparTalking", true);
    break;
  case "Král":
      StopPrince();
      animatorDiag.SetBool("kingTalking", true);
    break;
  case "Pilot":
      StopPrince();
      animatorDiag.SetBool("pilotTalking", true);
    break;
}
```

Tento switch přepíná mezi různými animacemi na základě parametrů *jménopostavyTalking*. Pro tyto potřeby jsem vytvořil animaci všech hovořících postav následujícím způsobem.

Postavu, nakreslenou v programu Moho jsem znovu vyexportoval jako obrázek ve formátu .png. U této postavy jsem však předtím vymazal ústa. V programu Moho jsem nakreslil samostatně osm různých tvarů úst, simulujících mluvu a všechny jsem též vyexportoval ve formátu .png a nahrál do projektu v Unity. U herního objektu dialogu jsem vytvořil nový herní objekt typu sprite, který jsem v rozhraní dialogů umístil do pozice úst postavy a nastavil hodnotu spritu na výchozí obrázek zavřených úst. Každou animaci postavy jsem pak vytvořil pouze změnami obrázku u tohoto spritu přibližně při každém třetím snímku. Animace pak při přehrání mění tvary úst postavy a simuluje tak její mluvu.

Výše zmíněný switch se řídí proměnnou *name*, tedy jménem postavy, která aktuálně hovoří. Pokud je tou postavou malý princ, nastaví se parametry spouštění animací u všech postav, kromě jeho samotného, na *false*, jelikož kód v aktuálně podobě nedokáže rozpoznat se kterou postavou princ hovoří. V případě, že hovoří kterákoli jiná postava, nastaví se parametr princovy animace na *false* prostřednictvím jednoduché metody *StopPrince* a parametr pro spuštění animace dané postavy se přenastaví na false.

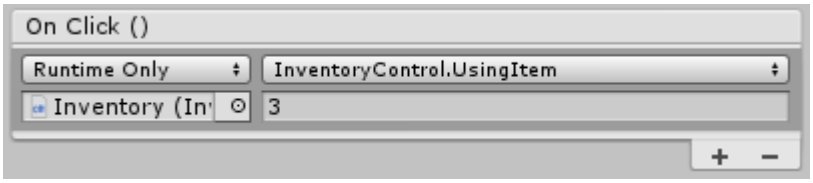
Pokud již metody *DisplayNextName* a *DisplayNextSentence* nemají žádná další jména a věty k zobrazení, vyvolá se metoda *EndDialogue*. Ta nastaví parametry všech animací opět na *false* a použitím *animatorDiag.SetBool("IsOpen", true)* skryje uživatelské rozhraní dialogů a zobrazí opět rozhraní pro ovládání hráče.

4.3.8 Interakce

Metoda, která řídí interakci předmětů, je rozdělena do dvou částí v závislosti na hodnotě proměnné *usingGroundObject*. V jednou případě probíhá interakce dvou předmětů v inventáři, v druhém pak interakce jednoho předmětu z inventáře a jednoho herního objektu. Kód v těchto případech je z tohoto důvodu odlišný, a proto se mu v této kapitole budu věnovat odděleně ve dvou podkapitolách. V třetí podkapitole se pak budu věnovat vyvolání výše zmíněné metody.

4.3.8.1 Vyvolání metody k použití předmětu

Jelikož jsem herní objekt inventáře vytvořil celkem ze šesti tlačítek, je zde možné využití události *OnClick* u každého z nich. Tímto způsobem jsem tedy postupoval při vyvolání potřebné interakce předmětů. Na začátek jsem vytvořil ve třídě *InventoryControl* novou metodu, nazvanou *UsingItem*, jejímž vstupním parametrem je proměnná typu int nazvaná *slot*. Tento parametr značí, který ze šesti potencionálních předmětů v inventáři má být využit k interakci. Každému z tlačítek *Item1* až *Item6*, které jsou součástí herního objektu *Inventory*, jsem nastavil událost *OnClick*. Konkrétně jsem do každého z tlačítek přesunul herní objekt *Inventory*, čímž program Unity automaticky nabídnul třídy, které jsou součástí tohoto objektu. Třídu i metodu jsem vždy zvolil totožnou, *InventoryControl* s metodou *UsingItem*. Metoda má však nastaven parametr *slot*, a tak jsem pro každé tlačítko nastavil tento parametry odlišně, od jedné do šesti. Pokud tak uživatel klikne na předmět na druhé pozici v inventáři, zavolá se proměnná *UsingItem* s parametrem 2.



Obrázek 22 - Nastavení události *OnClick*

4.3.8.2 Interakce mezi dvěma předměty

Po vyvolání metody *UsingItem*, určí podmínka, zda je hodnota proměnné *usingGroundObject* rovna *true* či *false*. Pokud je rovna *false*, znamená to, že interakce bude probíhat mezi dvěma předměty z inventáře. Taková interakce bude mít podobu vzniku nového předmětu, uvedeného v proměnné *newItem* obou předmětů a následného smazání původních dvou předmětů. Na začátek metody jsem definoval jednu novou proměnnou. Proměnná *inventoryControlCount* slouží jako kontrolní počet

```
public void UsingItem(int slot)
  {
  if (isFull[slot] != false)
    {
    if (item[slot].usingGroundObject == false)
      {
      for (int i = 0; i < slots.Length + 1; i++)
       {
       if (item[i] != null)
        {
        if (item[slot].itemName == item[i].interactionObejct)
          {
          CreateNewItem(i, slot);
          break;
          }
        if (inventoryControlCount == slots.Length + 1)
          {
FindObjectOfType<MessageBox>().ShowTimeMessage(item[slot].inter
actionError);
          }
        }
  } } }
```

Metoda je složena z kaskády podmínek. První z nich slouží k ověření, zda se ve slotu, na který hráč klikne, nějaký předmět vůbec nachází. Pokud ne, nestane se nic a metoda skončí. Pokud ano, algoritmus pokračuje dále k již výše zmíněné podmínce, která v tomto případě určuje, že interakce bude mezi dvěma předměty. Dalším krokem je spuštění cyklu s počtem iterací rovným počtu míst v inventáři navýšeným o jedna, tedy se sedmi iteracemi. Cyklus prozkoumá všechna místa v inventáři. Pokud je místo prázdné, přejde k další iteraci a dalšímu místu. Pokud prázdné není, další podmínka porovná, zda se název předmětu, na který hráč kliknul, shoduje s hodnotou proměnné *interactionObject* u předmětu na místě, ve kterém se právě cyklus nachází. Pokud ne, cyklus přejde k další iteraci. Pokud se však název předmětu a proměnná *interactionObject* shodují, znamená to, že se v inventáři nachází oba předměty potřebné k vzájemné interakci a zavolá se metoda *CreateNewItem* se vstupními parametry *i* a *slot*, tedy s čísly pozic obou předmětů, mezi kterými k interakci dochází. Pokud však nebude v žádné ze šesti iterací nalezena shoda,

spustí se sedmá iterace, při které se zobrazí varovná hláška s textem proměnné *interactionError*, uložené v předmětu.

Jak jsem již zmínil, metoda *CreateNewItem* má dva parametry, oba jsou typu *int* a nazval jsem je *Item1* a *Item2*. Čísla těchto parametrů odkazují na pozice dvou předmětů v inventáři, které jsou součástí interakce.

```
private Item itemHold;
public void CreateNewItem(int item1, int item2)
  {
  itemHold = item[item1].newItem;
  DestroyItem(item1);
  DestroyItem(item2);
  AddToInvetory(itemHold);
}
```

Pro potřeby této metody jsem vytvořil jednu pomocnou proměnnou typu *Item*. Pro vytvoření nového předmětu je nutné uvolnit místo v inventáři. Toho je dosaženo smazáním stávajících dvou předmětů vyvoláním metody *DestroyItem* s parametrem odkazujícím na místo v inventáři, ze kterého se předmět maže. Předměty musí být smazány dříve, než je vytvořen předmět nový. Z toho důvodu jsem vytvořil tuto pomocnou proměnnou, která v sobě zachová odkaz na nový předmět, jelikož původní odkaz je smazán spolu s původními předměty. Po nastavení této proměnné *itemHold* na hodnotu *newItem* z předmětu číslo 1 se oba původní předměty smažou. Poté je zavolána již zmíněná metoda *AddToInventory* s předmětem uloženým v proměnné *itemHold*. Tím se v inventáři vytvoří nový předmět.

```
public void DestroyItem(int slot)
  {
  item[slot] = null;
  slots[slot].GetComponent<Image>().sprite = null;
  isFull[slot] = false;
  itemCount = itemCount - 1;
  }
```

Výše zmíněná metoda, *DestroyItem,* přijímá číslo pozice předmětu, který má být odstraněn. V poli *item,* na přijaté pozici, nastaví hodnotu na *null,* čímž se odstraní hodnota původního předmětu. Stejný postup se aplikuje na *sprite* komponenty *Image* v poli *slots*. V poli *isFull* na dané pozici se pole označí jako neobsazené hodnotou *false* a v kontrolním součtu předmětů v inventáři se odečte hodnota 1. Tímto je předmět kompletně smazán z inventáře.

4.3.8.3 Interakce mezi předmětem a herním objektem

V případě, že je při zavolání metody *UsingItem* proměnná *usingGroundObject* u předmětu rovna *true,* očekává se interakce mezi jedním předmětem v inventáři a jedním objektem ve scéně. Tato interakce může mít podoby. První je vytvoření nového předmětu, například použití předmětu konvice u objektu jezírka vytvoří nový předmět, konvici s vodou. Druhou podobou interakce je spuštění animace, kdy se například při použití předmětu konvice s vodou u objektu růže spustí animace zalévání.

Pro tento případ jsem vytvořil novou třídu *GroundObjectInteraction*. Tato třída je využívána jako komponenta herních objektů využívaných pro interakci. Její činnost spočívá v detekci, zda se hráč nachází v blízkosti dotčeného herního objektu. Tato funkce je vytvořena pomocí metod *OnTriggererEnter2D* a *OnTriggererExit2D*, jejichž činnost jsem již popsal v kapitole *Načítání scén* a *Inventář a předměty*. Pokud je hráč v blízkosti tohoto herního objektu, nastaví se proměnná *groundItemName* ze třídy *InventoryControl* na hodnotu názvu předmětu. Pokud hráč opustí oblast předmětu, nastaví se tato proměnná na hodnotu *null*.

Prostřednictvím metody v přechozím odstavci si je tak třída *InventoryControl*, která řídí činnost inventáře, vždy vědoma interaktivních objektů v okolí hráče. Dokáže tak provádět interakci mezi předmětem z inventáře a herním objektem ve scéně.

```
if (item[slot].usingGroundObject == true)
  {
  if (item[slot].interactionObejct == groundItemName)
    {
    if (item[slot].groundObjectSingle == true)
      {
      CreateNewItem(slot, 7);
      }
    else
      {
      PlayAnimation(slot);
      dialogManager.ShowInventory();
      DestroyItem(slot);
      }
    }
  else
    {
FindObjectOfType<MessageBox>().ShowTimeMessage(item[slot].inter
actionError);
    }
  }
```

Jak jsem již zmínil na začátku této kapitoly, metoda *UsingItem* rozpozná, že se jedná o interakci mezi předmětem v inventáři a herním objektem na základě hodnoty *true*

v proměnné *usingGroundObjetct*. Poté metoda porovná hodnotu proměnné *interactionObject* u předmětu, na nějž hráč kliknul, s názvem objektu, získaným ze třídy *GroundObjectInteraction*. V případě, že se názvy neshodují, vyvolá se zpráva s textem proměnné *interactionError*. Pakliže se ale názvy předmětů shodují, další podmínka rozliší, zda se jedná o interakci, která vytvoří nový předmět v inventáři, nebo interakci, která spustí animaci. Toto rozhodnutí závisí na hodnotě proměnné *groundObjectSingle*. Pokud je její hodnota *true*, vyvolá se metoda *CreateNewItem* se dvěma parametry. Prvním je pozice předmětu v inventáři. Ten bude následně smazán a místo něj vytvořen nový předmět. Druhým parametrem je pomocná hodnota 7.

```
public void CreateNewItem(int item1, int item2)
{
  itemHold = item[item1].newItem;
  DestroyItem(item1);
  if (item2 != 7)
  {
  DestroyItem(item2);
  }
  AddToInvetory(itemHold);
}
```

Metodu *CreateNewItem* jsem původně vytvořil pro dva vstupující předměty z inventáře. Pro interakci jednoho předmětu a jednoho herního objektu jsem ji tak musel nepatrně změnit. Jak jsem již zmínil, místo herního objektu vstupuje do metody na parametru *item2* hodnota 7. V takovém případě se nevykoná žádný příkaz, týkající se potencionálního druhého předmětu, jelikož žádný ani neexistuje. Pokud má však parametr *item2* reálnou hodnotu odkazující na pozici v inventáři, tedy maximálně hodnotu 6, druhý předmět se smaže stejně, jako předmět první.

Pokud se však v přechozí metody rovná proměnná *groundObjectSingle* hodnotě *false*, nevytvoří se nový předmět, ale vyvolá se nová metoda *PlayAnimation* se vstupním parametrem pozice předmětu. Této metodě se věnuji v kapitole *Animace interakce*. Zároveň se vyvolá metoda *ShowInventory*, která zobrazuje a skrývá inventář. Jelikož je jednoznačné, že je v danou chvíli inventář otevřen, neboť jeho prostřednictvím hráč vyvolává interakci, výsledkem je zavření inventáře tak tak bylo možné plně sledovat animaci předmětu. Na závěr se předmět prostřednictvím *DestroyItem* vymaže, jelikož byl již použit.

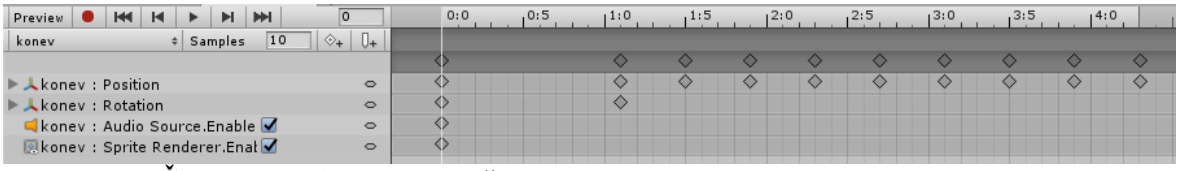
4.3.9 Animace interakcí

Použití předmětu, určeného pro interakci s herním objektem, znamená vymazání tohoto předmětu z inventáře. Hráč v tu chvíli ale nevidí žádný výsledek své akce, a kromě zmíněného smazání nic nenapovídá, že byl úkol splněn. Proto jsem pro každý herní předmět vytvořil animace zobrazující použité tohoto předmětu.

4.3.9.1 Vytvoření animací interakce

Animace je nutné vytvářet vždy ve scéně, ve které je plánováno jejich přehrání. Z toho důvodu jsem pro každý předmět, používaný na scéně vytvořil jeho animaci. Pro účely těchto animací jsem vytvořil nové herní objekty pro jednotlivé předměty. Každý z těchto objektů je typu *SpriteRenderer*, jelikož jeho účelem je pouze zobrazení obrázku. Pozici předmětů jsem vždy zvolil v závislosti herního objektu, se kterým provádějí interakci.

Například pro animaci zalévání růže jsem vytvořil herní objekt *konvice*. Tento objekt jsem přesunul nad růži na princově planetě a vypnul možnosti *enabled* u sprite rendereru, čímž se stal objekt neviditelný. Dále jsem vytvořil animaci *konev*, která na prvním snímku zapne možnost *enabled*, což má za následek zobrazení objektu. Pomocí změn pozice objektu na jednotlivých snímcích jsem vytvořil animaci zalévání. Na konci animace zapne animátor druhou animaci *konevHide*, která opět změní možnost *enabled* na *false* a animace zmizí. Tato animace je zároveň nastavena jako výchozí, takže při spuštění scény se přehrává právě ona. Přechod z výchozí animace *konevHide* na animaci *konev* je zajištěn spouštěčem s názvem *play*. Obdobným způsobem jsem vytvořil animace pro všechny herní předměty.



Obrázek 23 - Časová osa animace pro použití konve

4.3.9.2 Spuštění animací interakce

Jak jsem již zmínil v kapitole *Interakce mezi předmětem a herním objektem*, spouštění animací má na starosti nově vytvořená *PlayAnimation*, kterou jsem umístil do třídy *InventoryControl*. Problémem však je, že tato třída je součásti inventáře, který je

součástí objektu *UI*. Tento objekt se přesouvá mezi scénami a nenachází se tak pevně na konkrétní scéně, a proto není možné standartně asociovat animace z jednotlivých scén s touto třídou. Pro řešení tohoto problému jsem vytvořil novou třídu, nazvanou *AnimationHolder*, kterou jsem umístil na každou scénu do herního objektu, pod kterým jsou shromažďovány animace vytvořené v předchozí kapitole. Tato třída je složena pouze z dvou polí proměnných, první pole proměnných typu string, nazvané *animationName* nese názvy animací. Druhé pole, typu *Animator*, nazvané *animations* nese jednotlivé animace předmětů. Do tohoto pole jsem v prostředí *Unity* přesunul všechny objekty, který jsem vytvořil v předchozí kapitole. Do pole jmen jsem pak ve stejném pořadí, ve kterém jsou v druhém poli uloženy animace, napsal názvy jednotlivých animací. Tyto názvy musí být pro správné fungovaní totožné s názvy předmětů z inventáře.

```
private AnimationHolder animHold;
public void PlayAnimation(int slot)
 {
 animHold = FindObjectOfType<AnimationHolder>();
 for (int i = 0; i < animHold.animationName.Length; i++)
   {
   if (animHold.animationName[i] == item[slot].quest)
     {
     animHold.animations[i].SetTrigger("play");
     break;
     }
   }
 }
```

Před začátkem této metody jsem definoval novou proměnnou, nazvanou *animHold*. Do této proměnné typu *AnimationHolder* je po vyvolání metody *PlayAnimation* uložen veškerý obsah třídy *AnimationHolder* z aktuální scény. Metoda tak pomocí této třídy zná všechny animace předmětů, nacházející se na scéně. Následně se spustí cyklus, jehož počet iterací je roven počtu animací uložených ve zmíněné třídě. V případě, že se při běhu cyklu nalezne shoda, mezi názvem animace, uloženým v poli třídy *AnimationHolder*, a mezi názvem předmětu, který vstupuje do této metody, spustí se animace uložená na pozici aktuální iterace pomocí triggeru *play*.

4.3.10 Kvíz

Pro správné fungování kvízu vytvořím několik tříd, jejichž funkcí bude držení otázek a správných odpovědí, propojení s uživatelským rozhraním kvízů a aktualizace textu v tomto rozhraní na základě aktuální otázky a sčítání správných a špatných odpovědí. Funkce pro zobrazení kvízu bude navázána na načítání další scény, tedy na třídu

LoadLevel. Tuto třídu upravím tak, aby se při jejím zavolání nenačetla nová scéna, ale kvíz, pokud existuje, a teprve po dokončení tohoto kvízu se načte nová scéna.

4.3.10.1 Vytvoření otázek a odpovědí

Pro vytváření nových otázek a odpovědí, jak správných, tak špatných jsem vytvořil novou třídu *Quiz*. Tato třída neobsahuje žádné metody a její funkce je pouze určení jednotlivých proměnných, používaných pro otázky a odpovědi.

```
[System.Serializable]
public class Quiz
{
    public string question;
    public string[] answers;
    public int CorrectAnswer;
}
```

Přidání řádku *Systém.Serializable* umožní této třídě serializaci, čímž bude možné ukládat stav jednotlivých hodnot při využití v jiné třidě. To je důležité, jelikož další třída, ovládající činnost kvízu bude využívat celý list této třídy, kdy každý prvek z pole bude mít své proměnné z třídy *Quiz*. Proměnná typu string, nazvaná *question,* je určena pro jednotlivé otázky. Pole proměnných typu string, nazvané *answers* je určeno pro odpovědi k dané otázce. Odpovědi budou vždy čtyři, tedy pole bude mít vždy tuto velikost. Poslední proměnná je typu int, tedy číselné hodnoty, jelikož odkazuje na číslo správné odpovědi. Jelikož však pole vždy začíná hodnotou 0, je zde nutné počítat odpovědi od nuly do tří, nikoli od jedné do čtyř.

4.3.10.2 Třída pro řízení kvízů

Pro řízení činnosti kvízů, které se nacházejí na každé scéně, jsem vytvořil novou třídu *QuizManager*, Tuto třídu jsem přidal jako komponentu pro všechny objekty s názvem *Quiz*, jejichž vytváření jsem popsal v kapitole *Rozhraní kvízů*. Pro začátek jsem deklaroval jednotlivé používané proměnné. Před tím jsem však do třídy přidal nový using, konkrétně *UnityEngine.UI*, jelikož třída bude pracovat s textem otázek a odpovědí.

```
    public List<Quiz> quiz;
    public GameObject[] options;
    public int currentQuestion;
    public Text questionTxt;
    public LoadLevel loading;
```

První proměnná je typu *Quiz*, konkrétně jde o *List* těchto proměnných. Pomocí listu bude možné do této třídy zapisovat jednotlivé otázky, odpovědi a označení správných odpovědí. Druhá proměnná je pole herních objektů nazvaná *options*. Jelikož je veřejného

typu, nastavil jsem v prostředí Unity velikost tohoto pole na 4 elementy přesunul do nich postupně herní objekty *answer0* až *answer3*, tedy tlačítka, která jsou součástí uživatelského rozhraní kvízu. Třetí proměnná slouží jako číselné značení aktuálně zobrazované otázky. Další proměnná, *questionTxt*, odkazuje na herní objekt typu *Text*, nazvaný *question*, který jsem vytvářel jako součást uživatelského rozhraní kvízu. Tento objekt jsem zároveň do této veřejné proměnné přesunul v prostředí Unity. Poslední deklarovaná proměnná je typu třídy *LoadLevel*. Účelem této veřejné proměnné je odkaz na třídu *LoadLevel* v aktuální scéně tak, aby mohla třída *QuizManager* po dokončení kvízu vyvolat metodu pro načtení nové scény.

```
private void Start()
 {
   GenerateQuestions();
 }
public void Correct()
 {
   quiz.RemoveAt(currentQuestion);
   GenerateQuestions();
 }
```

Jako první metodu ve třídě *QuizManager* jsem vytvořil velice jednoduchou metodu, nazvanou *Correct*. Tato metoda má za úkol, po obdržení odpovědi, odstranění otázky z listu a vyvolat metodu *GenerateQuestions*, která vybere další, dosud nezodpovězenou otázku z listu. Vyvolání metody *GenerateQuestions* jsem zároveň přidal do metody *Start*, aby byla ihned po zobrazení kvízu jedna otázka vygenerována. Jako druhou metodu jsem vytvořil právě *GenerateQuestions*.

```
void GenerateQuestions()
  {
  if (quiz.Count > 0)
    {
    currentQuestion = Random.Range(0, quiz.Count);
    questionTxt.text = quiz[currentQuestion].question;
    SetAnswers();
    }
  else
    {
    loading.Load();
    this.gameObject.SetActive(false);
    }
 }
```

Pokud bude počet prvků v listu *quiz* větší než nula, vygeneruje se náhodně nová otázka ze všech aktuálně obsažených otázek v listu. Textu herního objektu *questionTxt* se nastaví na hodnotu otázky z listu s indexem vygenerovaným v předchozím kroku a vyvolá se další metoda, *SetAnswers*. Jelikož se v přechozí zmíněné metodě vždy z listu otázka po

zodpovězení odstraní, počet prvků v listu vždy skončí na hodnotě 0. V tu chvíli se spustí metoda *Load* ze třídy *LoadLevel* a objekt kvízu se nastaví jako neaktivní.

Poslední metodou třídy *QuizManager* je již zmíněná metoda *SetAnswers*. Úkolem této třídy je nastavení správného textu (odpovědí) na jednotlivá tlačítka kvízu.

```
void SetAnswers()
  {
  for (int i = 0; i < options.Length; i++)
    {
     options[i].GetComponent<AnswerScript>().isCorrect = false;
     options[i].transform.GetChild(0).GetComponent<Text>().text
= quiz[currentQuestion].answers[i];
     if (quiz[currentQuestion].CorrectAnswer == i)
      {
      options[i].GetComponent<AnswerScript>().isCorrect = true;
      }
    }
  }
```

Nejdříve se spustí cyklus s počtem iterací rovným počtu odpovědí, respektive počtu tlačítek, tedy čtyřem. Každé tlačítko má jako komponentu třídu *AnswerScript*, které se ještě budu věnovat a zároveň má každé tlačítko u události *OnClick* nastaveno vyvolání metody *Answer* z této třídy. V první řádce cyklu tedy metoda nastaví u každého tlačítka hodnotu *isCorrect* jako *false*. Tím pádem označí všechny tlačítka jako špatné odpovědi. Poté postupně v každé iteraci cyklu nastaví text jednotlivých tlačítek na jednotlivé odpovědi z listu *quiz*. Jelikož text tlačítek je samostatný herní objekt, který je tlačítku podřazen, k jeho dosažení jsem použil metodu *GetChild*, která sestoupí v herním objektu o jednu vrstvu níže a teprve poté začne pomocí *GetComponent<Text>* vyhledávat textovou komponentu. Pokud se proměnná *CorrectAnswer* z listu otázek rovná hodnotně aktuální iterace, nastaví se v současné iteraci hodnota *isCorrect* na *čímž* se označí tlačítko, nesoucí správnou odpověď.

Pro vyhodnocování správných a špatných odpovědí jsem vytvořil dvě další třídy, jedna, nesoucí název *Progress*, je určena pro zaznamenávání postupu hráče ve hře. Je tedy přiřazena jako komponenta k hernímu objektu hráče a přesouvá se s ním napříč scénami. Tato třída nese pouze dvě jednoduché proměnné typu int, *score* a *allScore*. *Score* zaznamenává správné odpovědi a *allScore* zaznamenává všechny odpovědi, aby tak aplikace mohla na konci vyhodnotit hráčovu úspěšnosti. Druhou třídou je již zmiňovaný *AnswerScript*. Jeho úkolem je provádění akcí v případě správné a v případě špatné odpovědi.

```
    public QuizManager quizManager;
    public Progress progres;
    private void Start()
    {
        progres = FindObjectOfType<Progress>();
    }
    public void Answer()
    {
        if (isCorrect)
        {
            progres.allskore = progres.allskore + 1;
            progres.skore = progres.skore + 1;
            Debug.Log("Correct");
            quizManager.Correct();
        }
        else
        {
            progres.allskore = progres.allskore + 1;
            Debug.Log("Wrong");
            quizManager.Correct();
     }  }
```

Pro tuto třídu jsem definoval pouze dvě proměnné, odkazující na jiné herní objekty. První je *QuizManager*, ta odkazuje na herní objekt kvízu. Jelikož se tlačítka s komponentou *AnswerScript* nacházejí, spolu s objektem kvízu, vždy na stejné scéně, přiřazení jsem provedl prostřednictvím rozhraní v Unity. Druhá proměnná *progres* však odkazuje na objekt hráče s jeho třídou *Progress*, který není pevně součástí scény, ale přesouvá se mezi nimi. Z toho důvodu je po startu třídy tento objekt hráče vyhledán a přiřazen pomocí *FindObjectOfType<Progress>*. Pokud je tlačítko, na které hráč klikne, označeno předchozí třídou jako správné, přičte se hodnota jedna k proměnné *score* a zároveň k proměnné *allScore*. Pro kontrolu a účely testování zároveň vypíše do konzole zprávu „Correct“. Na závěr se vyvolá metoda *Correct* ze třídy *QuizManager*. Ta smaže aktuální otázku a vygeneruje novou. V případě, že je hodnota *isCorrect* u tlačítka nastavena jako *false*, čímž je tlačítko označeno jako špatná odpověď, přičte se hodnota 1 pouze k proměnné *allScore* a do konzole se vypíše zpráva „Wrong“. Na závěr se opět vyvolá již zmíněná metoda *Correct*. V tuto chvíli je již dokáže správně vyhodnocovat správné a špatné odpovědi a navyšovat hráčovo skóre.

4.3.11 Ukládání a načítání dat

V Unity3D existuje vícero způsobů, jak zachovat data jako například aktuální skóre, obsah inventáře či pozice i po ukončení aplikace. Ve své práci jsem zvolil nejjednodušší způsob uložení dat, a to třídu ***PlayerPrefs***. Tato třída dokáže trvale ukládat datové typy

string, bool a float s jejich unikátním klíčem (názvem). V případě operačního systému Android se tato data ukládají do složky: /data/data/pkg-name/shared_prefs/pkg-name.xml. [30]

4.3.11.1 Ukládání dat

Pro ukládání dat jsem vytvořil třídu *SaveManager.cs*. Tato třída musí být přítomna ve všech scénách, proto jsem ji přiřadil k hernímu objektu *UI,* který má přiřazenou metodu *DontDestroyOnLoad()* k zachování mezi jednotlivými scénami. Prvně jsem deklaroval proměnné využité pro tuto třídu, poté jsem vytvořil metodu *SaveData(),* ve které se budou postupně ukládat všechna potřebná data. Jelikož mezi ukládanými daty je i název aktuálně zapnuté scény, je potřeba do třídy přidat nový obor příkazů pomocí nového *using*, konkrétně *UnityEngine.SceneManagement*. tomu bude mít třída přístup k názvu scény.

```
public Progress progres;
public InventoryControl inventory;
public ItemCatalog itemCatalog;

public void SaveData()
{
  PlayerPrefs.SetInt("played", 1);
  PlayerPrefs.SetString("currentScene",
SceneManager.GetActiveScene().name);
  PlayerPrefs.SetInt("score", progres.skore);
  PlayerPrefs.SetInt("allscore", progres.allskore);
}
```

Definováním proměnných je zde myšleno definování tříd *Progress, InventoryControl* a *ItemCatalog*. Z prvních dvou zmíněných tříd budou ukládána data. Z třídy *Progress* jsou zde konkrétně ukládány hodnoty *skore* a *allskore*, tedy hráčem dosažené skoré a maximální možné skóre. Dále je zde uložen klíč played s hodnotou jedna. Tento klíč slouží při zapnutí aplikace k rozpoznání, zda existují nějaká uložená data. Jak jsem zmínil v předchozím odstavci, je nutné také uložit název aktuální scény. Tento název je získán metodou *GetActiveScene().name* a poté uložen pod jménem *currentScene*. Ze třídy *InventoryControl* je pak ukládán aktuální obsah celého inventáře, čehož jsem dosáhl pomocí následujícího cyklu, umístěného také uvnitř metody *SaveData()*.

```
for(int i = 0; i < inventory.slots.Length; i++)
{
 if (inventory.isFull[i] == true)
 {
 PlayerPrefs.SetString("slot" + i, inventory.item[i].itemName);
 }
 else
 {
 PlayerPrefs.SetString("slot" + i, null);
 }
}
```

Tento cyklus má shodný počet iterací s délkou pole *slots* v inventáři, tedy cyklus proběhne tolikrát, kolik je míst v inventáři. Pokud je na i-té pozici nastavena hodnota proměnné *isFull* jako pravda, uloží se pomocí *PlayerPrefs* proměnná typu string s názvem *slot + i* a hodnotou rovnající se názvu předmětu. Pokud je na i-té pozici nastavena hodnota proměnné *isFull* jako nepravda, uloží se stejná proměnná jako v předchozím případě, pouze s hodnotou *null*.

Velmi podobný algoritmus je pak použit také na uložení pole hodnot *doneOrNot* ve třídě *Progress*, které uvádí, jaké herní úkoly byly již splněny. Rozdíl je pouze v tom, že třída *PlayerPrefs* neumožňuje ukládat hodnoty typu *bool,* a proto je zde přidána podmínka, která říká, že pokud je na i-té pozici nastavena hodnota jako pravda, tedy že i-tý úkol byl již splněn, uloží se do třídy *PlayerPrefs* proměnná typu *Int* s hodnotou 1. Pokud je hodnota uložena jako nepravda, uloží se *Int* s hodnotou 0.

```
for (int i = 0; i < progres.doneOrNot.Length; i++)
  {
  if (progres.doneOrNot[i] == true)
    {
    PlayerPrefs.SetInt("progres" + i, 1);
    }
  else
    {
    PlayerPrefs.SetInt("progres" + i, 0);
    }
  }
```

Naprosto stejným algoritmem jsou pak uloženy také hodnoty z pole *picked* z třídy *ItemCatalog*, které říká, jaké předměty byly hráčem již vzaty do inventáře a tím pádem zabraňuje v jejich opětovném načtení ve scéně.

Takto je hotova kompletní metoda pro ukládání dat, kterou je pouze nutné vyvolat ve správný čas. Metodu je možné volat například po načtení nové scény či uložení herního předmětu do inventáře. Pro svou práci jsem však zvolil způsob automatického ukládání při vypnutí hry. Problém tohoto způsobu však vidím v odlišném chování na různých platformách. Pro použití na operačním systému Windows by stačilo metodu *SaveData*

vložit do metody *OnApplicationQuit*. V případě operačního systému Android je tato metoda však nefunkční. Zde totiž funguje metoda *OnApplicationPause* s parametrem *pause*. Výsledný kód, funkční na OS Android, ale nikoli na Windows, vypadá takto.

```
private void OnApplicationPause(bool pause)
{
if (pause){
SaveData();
}}
```

4.3.11.2 Načtení dat

Načítání dat probíhá ve stejné třídě jako jejich ukládání, tedy ve třídě *SaveManager.cs*. Pro vyvolání sady příkazů, které načtou data do aplikace, jsem vytvořil metodu *LoadData*.

```
public void LoadData()
{
if (PlayerPrefs.GetInt("played") == 1)
{
SceneManager.LoadScene(PlayerPrefs.GetString("currentScene"));
progres.skore = PlayerPrefs.GetInt("score");
progres.allskore = PlayerPrefs.GetInt("allscore");
```

Při zavolání metoda nejdříve podmínkou *if* ověří, zda existují uložená data vyhledáním *PlayerPrefs* s klíčem *played*. Pokud klíč s hodnotou 1 existuje, začne metoda nastavovat proměnné v aplikaci dle uloženého obsahu. Nedříve vyvolá načtení nové scény s názvem, který je shodný s názvem uloženým v *PlayerPrefs*. Poté nastaví proměnné hráčova aktuálního skóre a maximálního dosažitelného skóre ve třídě *Progress*. Následuje obnovení předmětů v inventáři, které je realizováno pomocí dvou cyklů *for*.

```
for (int i = 0; i < inventory.slots.Length; i++)
{
for (int j = 0; j < itemCatalog.items.Length; j++)
{
if (PlayerPrefs.GetString("slot" + i) ==
itemCatalog.items[j].itemName)
{
inventory.AddToInvetory(itemCatalog.items[j]);
}}}
```

První cyklus má počet iterací *i* roven velikosti inventáře. Uvnitř něj se při každé iteraci spouští druhý cyklus. Ten má počet iterací *j* roven počtu veškerých herních předmětů, uvedených v poli *items*, které je součástí třídy *ItemCatalog*.

Pokud se hodnota klíče s názvem *slot + i* rovná názvu předmětu na pozici *j* v *ItemCatalog.items*, přidá se tento předmět do inventáře. Jinými slovy cyklus vezme nejprve první název předmětu, uložený v *PlayerPrefs* a porovná ho s názvy všech

předmětů uložených v katalogu. Pokud najde shodu v názvu, shodný předmět přidá do inventáře.

Na závěr metoda *LoadData* provádí nastavení pole proměnných *doneOrNot* ze třídy *Progress* a pole *picked* ze třídy *ItemCatalog*.

```
for (int i = 0; i < progres.doneOrNot.Length; i++)
{
if (PlayerPrefs.GetInt("progres" + i) == 1)
{
progres.doneOrNot[i] = true;
}
else
{
progres.doneOrNot[i] = false;
}
}
```

Metodu *LoadData* je nutné poté v některé fázi spouštění hry vyvolat, aby se mohla požadovaná data načíst. Ve své práci jsem použil následující způsob: při zapnutí hry se porovná, zda existuje *PlayerPrefs* s klíčem *played* a uloženou hodnotou 1. Pokud ano, znamená to, že existují uložená data a v herním menu se zobrazí dvě tlačítka, „Pokračovat“ a „Nová hra“. V případě kliknutí na tlačítko „Pokračovat“ se vyvolá metoda *LoadLevel*, čímž se načte uložená scéna spolu s uloženými proměnnými. V případě kliknutí na „Nová hra“ se vyvolá metoda *ResetData*, jejímž jediným obsahem je příkaz *PlayerPrefs.DeleteAll()*. Tento příkaz vymaže veškeré uložené hodnoty a hra se tak spustí od úplného začátku.

4.4 Tvorba aplikace a testování

Na konci kapitoly *Implementace* má hra již všechny základní náležitosti z hlediska kódu i z hlediska nastavení v enginu Unity, a tak lze přejít k vytvoření samotné aplikace ve formátu .apk a jejímu otestování.

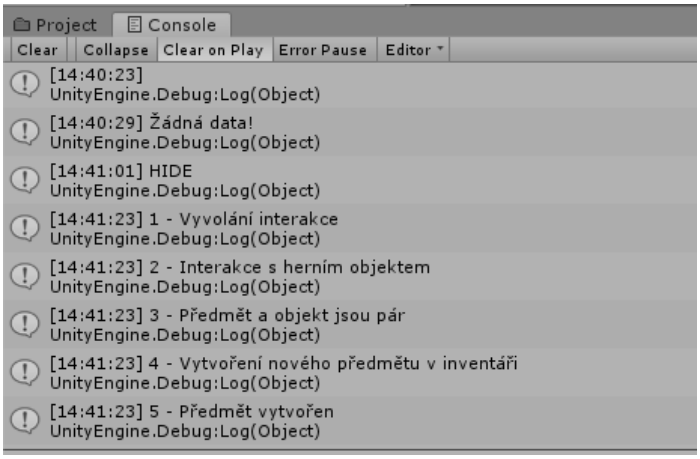
4.4.1 Testování v prostředí Unity

Herní engine Unity 3D a zároveň program Visual Studio poskytují možnost pro automatizované testovaní neboli unit testing. Takové testování ale vyžaduje další, ne příliš jednoduché psaní testovacích sekvencí a je vždy dobré promyslet, zda se jsou unit testy pro daný projekt vhodné, neboť jejich tvorba časově převyšuje dobu komplexního

uživatelského textování, které jsem pro svou práci zvolil já. Testování jsem prováděl prostřednictvím herního režimu přímo v programu Unity.

Pro zjišťování, zda vytvořené větvené algoritmy dosáhnou na všechny části kódu jsem používal příkaz *Debug.Log(„zpráva“),* který vypíše informaci zprávu či proměnnou zadanou v závorkách do konzole jako informaci. Tento příkaz jsem postupně umisťoval do různých částí tříd a metod s rozdílnými zprávami pro snadnou identifikaci.

Například při testování nefungující metody *UsingItem* z třídy *InventoryControl* aplikace nepoužila při kliknutí žádný předmět, ale zároveň do konzole nevypsala žádnou kritickou chybu. Jelikož se *UsingItem* skládá z kaskády podmínek a cyklů, umístil jsem pod každou podmínku a cyklus příkaz *Debug.Log,* do jehož zprávy jsem umístil čísla od jedničky výše. Jelikož posloupnost končila číslem 5, ale v konzoli při testování se zobrazilo vždy maximálně číslo 4, umožnilo mi to identifikaci chyby ve čtvrté podmínce, která porovnávala dvě proměnné, z nichž jedna byla špatně zvolena.

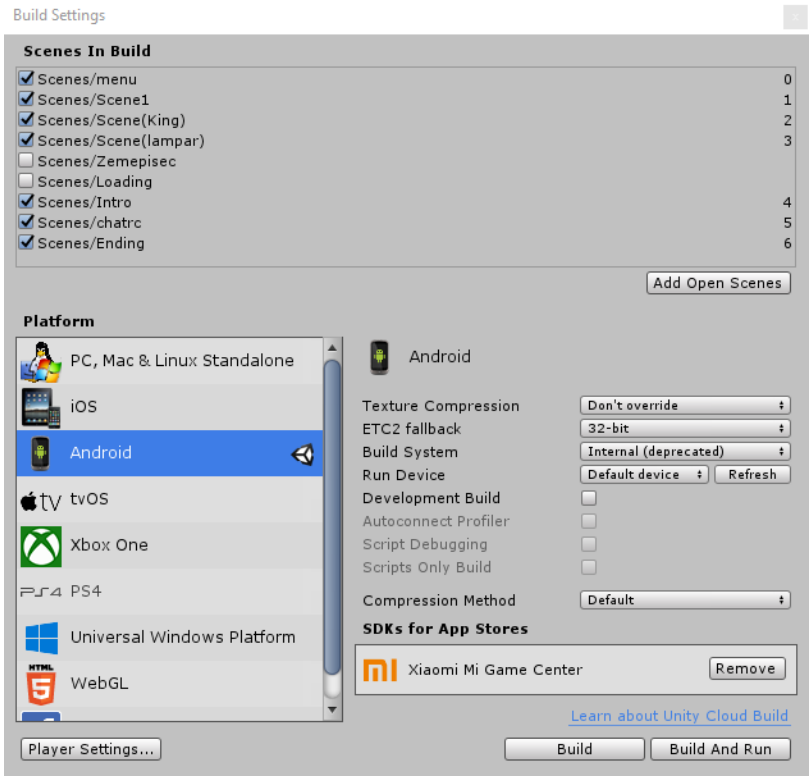


Obrázek 24 - Testování v konzoli

4.4.2 Tvorba aplikace

Aplikace je v současném stavu již připravená pro automatické sestavení výsledného apk souboru, neboť již na začátku praktické části, při vytváření nového projektu jsem nastavil jako cílovou platformu Android. Pro sestavení aplikace je nutné otevřít okno *Build Settings* v záložce *File*. V tomto okně existuje záložka *Scenes In Build*. Do této záložky je nutné přetáhnout všechny scény, které budou obsažené ve finální aplikaci, neboť Unity výslednou aplikaci sestaví právě z těchto scén. Na základě toho budou v aplikaci obsaženy pouze soubor, C# skripty a jiné assety, které jsou přiřazeny k nějakému z herních objektů v těchto scénách. Pokud je například v projektu uloženo několik hudebních skladeb, ale

na žádné scéně nejsou spuštěny ani přiřazeny, Unity je do finálního buildu nezahrne, čímž se nebude uměle zvětšovat velikost výsledné aplikace. Do okna *Scenes In Build* jsem tak přetáhl scény s princem, králem, lampářem, pilotem, princovou chatrčí a dvě nově vytvořené scény, *menu* a *Ending*, které obsahují pouze tlačítka pro start hry a závěrečné zhodnocení prostřednictvím textového pole. Ostatní nastavení okna *Build Settings* jsem ponechal na výchozích hodnotách a jako software development kit jsem taktéž ponechal výchozí *Xiaomi Mi Game Center*



Obrázek 25 - Závěrečné nastavení projektu

V tuto chvíli je ještě možné zvolení jiné cílově platformy. Jediným problémem bude ovládání hlavní postavy, neboť ovládání bylo vytvářeno s ohledem na dotykovou obrazovku, a tak pro využití například pro počítače by bylo nutné přetvoření ovládání na konkrétní klávesy. Po stisknutí tlačítka *Build* se zobrazí průzkumník souborů pro výběr cílového uložiště a názvu souboru. Po jeho zvolení Unity začne automaticky prostřednictvím SKD vytvářet výsledný soubor formátu .apk. Výsledné soubory jsou v programu Unity ve výchozím nastaveny na maximální možnou podporu starších verzí Android. Tato podpora začíná u verze Android 4.1, každopádně je nutno podotknout, že u takto starých zařízení bude sice možné aplikace spustit, ale mohou nastat vážné výkonnostní problémy.

4.4.3 Testování na mobilních zařízeních

Výsledný soubor „malyprinc.apk“ jsem nahrál do několika mobilních zařízeních s operačním systémem Android OS. Tato zařízení měla rozdílné verze operačního systému Android a zároveň rozdílné rozlišení obrazovky. Konkrétně se jednalo o následující zařízení.

Tabulka 3 - Seznam testovacích zařízení

Název zařízení	Operační systém
Xiaomi Mi A2 Lite	Android 10
HUAWEI MediaPad T5	Android 8
Xiaomi Redmi Note 8 Pro	Android 10
Lenovo A3500	Android 4.4
Huawei P Smart	9.1

Aplikace vykazovala na všech zařízeních totožné vlastnosti z hlediska stability a funkčnosti s výjimkou zařízení Lenovo A3500, které sice svou verzí Androidu kompatibilní bylo, ale z důvodu vysokého rozlišení aplikace mělo zařízení kritické výkonnostní chyby, a proto bylo z testování dále vyřazeno. Uživatelské rozhraní se na zbylých zařízeních vždy správně přizpůsobilo potřebnému rozlišení, s drobnými odchylkami v jeho rozložení. Každé ze zařízení při instalaci upozornilo na potencionální nebezpečnost aplikace, jelikož mnou nebyla registrována na Google Play a jedná se tak o neověřenou aplikaci.

Žádné ze zařízení nemělo problém s výkonem při hře a hra běžela plynule. Veškeré herní funkce byly otestovány na všech zařízeních s totožnými pozitivními výsledky. Ukládání dat po vypnutí aplikace také proběhlo na všech zařízeních úspěšně a hra se vždy úspěšně načetla do původní situace. Pro účely testování byla na každém zařízení aplikace dokončena rozdílnými uživateli, a to minimálně dvakrát. Žádný z uživatelů nenahlásil problém ovlivňující úspěšné dokončení a z toho důvodu považuji testování za úspěšně dokončené. Jedinou výjimkou v testování bylo zařízení Lenovo A3500, jehož rok výroby je 2013 a z důvodu kritických výkonnostních problémů nebylo možné aplikaci plně provozovat. Vzhledem ke staří zařízení však považuji výsledek za očekávaný.

5 Závěr

V rámci této diplomové práce jsem vytvořil interaktivní vzdělávací aplikaci Malý princ, určenou pro zařízení s operačním systémem Android OS. Celou aplikaci jsem tvořil základními funkcionalitami herního enginu Unity 3D s využitím programovacího jazyka C# a grafické aplikace Moho.

Z analýzy aplikací dostupných na portálech Google Play, itch.io a Steam vyplynulo, že existuje celá řada aplikací s tématikou Malý princ. Všechny tyto aplikace byly však dílem Malý princ pouze tematicky inspirovány a žádná z nich neobsahovala dějovou linii literárního díla. Zároveň také žádná ze zkoumaných aplikací nebyla zaměřena na vzdělávání, a proto jsem usoudil, že plánovaný koncept bude originální.

Herní mechaniku aplikace jsem navrhoval tak, aby splňovala tři ze čtyř základních didaktických funkcí, konkrétně motivaci, upevnění osvojených dovedností a kontrolu dovednosti prostřednictvím zpětné vazby, a tím tak splňovala primární předpoklad vzdělávací aplikace.

Při tvorbě samotné aplikace jsem představil základní funkce programu Unity 3D potřebné pro tvorbu dvourozměrných aplikací. Tyto funkce jsem zahrnul do tvorby běžných herních prvků. Konkrétně jsem pro aplikaci vytvořil hráčovu postavu Malého prince, pohybující se prostřednictvím uživatelského rozhraní ve vytvořeném herním světě. Dalšími herními funkcemi, obsaženými v aplikaci jsou předměty a inventář, dialogový systém, animace, prolínání mezi scénami, interakce a závěrečný vzdělávací kvíz.

Výsledná vzdělávací aplikace pokrývá první část literárního díla Malý princ, kde se hráč setká s postavou pilota, růže, krále a lampáře. Na základě dialogů s postavami je pak zobrazen na konci scén kvíz, ověřující znalosti právě zažitých událostí. Na závěr jsem aplikaci uložil ve formátu .apk a nechal otestovat různými uživateli na pěti odlišných zařízení, lišících se verzí operačního systému Android a rozlišením obrazovky. Uživatelské testování nepoukázalo na žádný problém nefunkčnosti vytvořených komponent.

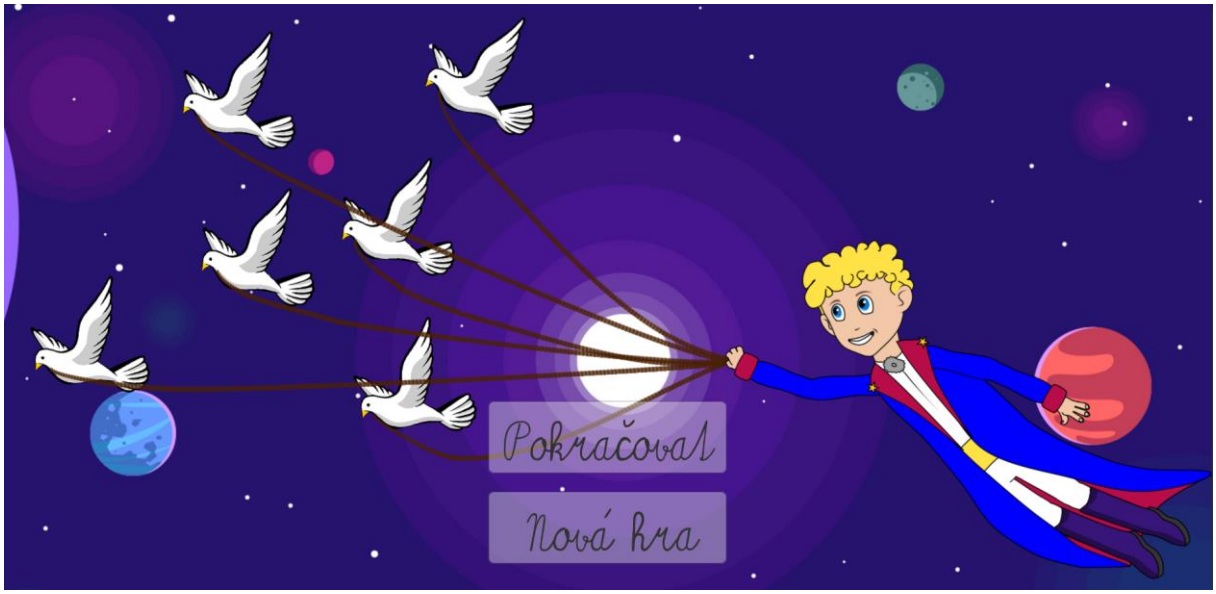
6 Seznam použitých zdrojů

1. DOSTÁL, Jiří. INSTRUCTIONAL SOFTWARE AND COMPUTER GAMES – TOOLS OF MODERN EDUCATION. *Journal of Technology and Information Education*. 2009, **2009**(1), 23–27. ISSN 803-537X.
2. DOSTÁL, Jiří. *Výukové programy*. Olomouc: Univerzita Palackého v Olomouci, 2011. ISBN 978-80-244-2782-9.
3. LOHNES, Kate. The Little Prince: Analysis and reception. *Britannica* [online]. Chicago: Encyclopædia Britannica, ©2021 [cit. 2021-03-03]. Dostupné z: https://www.britannica.com/topic/The-Little-Prince
4. SAINT-EXUPÉRY, Antoine de. *Malý princ*. Praha: Dobrovský, 2014. Omega (Dobrovský). ISBN 978-80-7390-220-9.
5. HALPERN, Jared. Developing 2D Games with Unity: Independent Game Programming with C#. Berkeley, CA: Apress, © 2019. ISBN 978-1-4842-3771-7.
6. THORN, Alan. Learn Unity for 2D Game Development. 1. New York: Apress, 2013. ISBN 978-1-4302-6230-5.
7. TAKOORDYAL, Kishan. Beginning unity Android game development: from beginner to pro. [United States]: Apress, [2020]. ISBN 978-1-4842-6001-2.
8. How does a Physics Engine work? *Harold Serrano* [online]. Untold Engine, © 2013-2020 [cit. 2021-03-20]. Dostupné z: https://www.haroldserrano.com/blog/how-a-physics-engine-works-an-overview
9. Real-time rendering in 3D. *Unity* [online]. San Francisco: Unity Technologies, © 2021 [cit. 2021-03-20]. Dostupné z: https://unity3d.com/real-time-rendering-3d
10. What is the Difference Between an API and an SDK? *NORDIC APIS* [online]. Nordic APIs AB, © 2013-2021 [cit. 2021-03-20]. Dostupné z: https://nordicapis.com/what-is-the-difference-between-an-api-and-an-sdk/
11. Multi-platform development. UNREAL ENGINE [online]. Cary, North Carolina: Epic Games, ©2004-2021 [cit. 2021-02-22]. Dostupné z: https://www.unrealengine.com/en-US/features/multi-platform-development
12. Sharing and Releasing Projects: Information on developing for platforms other than PC. UNREAL ENGINE [online]. Cary, North Carolina: Epic Games, ©2004-2021 [cit. 2021-02-22]. Dostupné z: https://docs.unrealengine.com/en-US/SharingAndReleasing/index.html

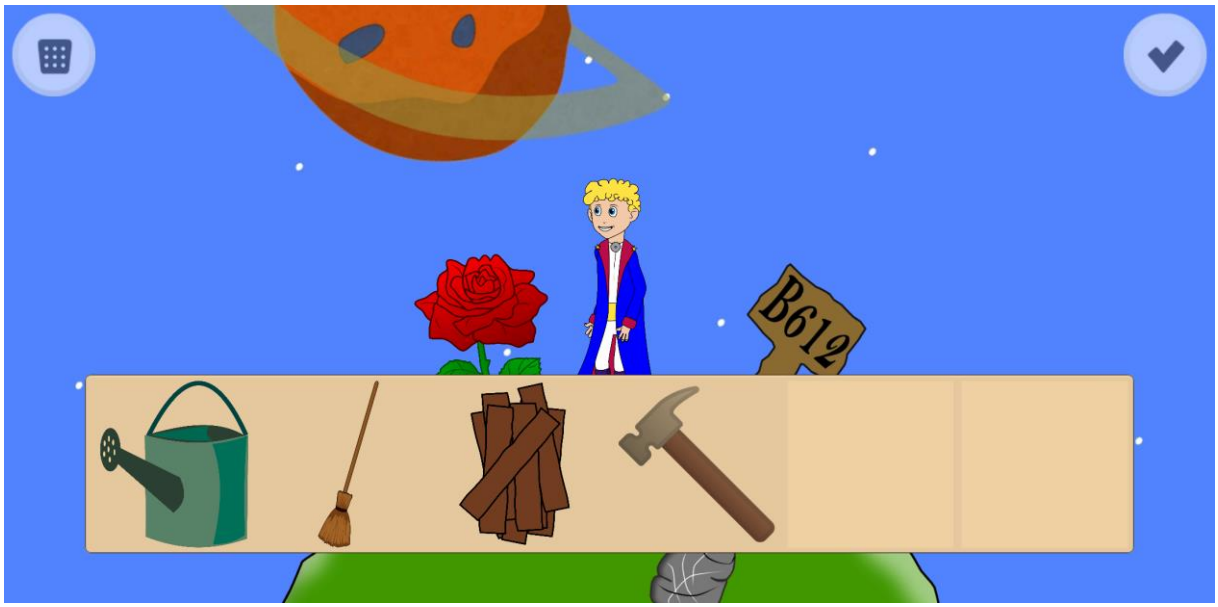
13. Plans and pricing. Unity Store [online]. San Francisco: Unity Technologies, © 2021 [cit. 2021-02-22]. Dostupné z: https://store.unity.com/#plans-business
14. UnityScript versus JavaScript. Unifycommunity [online]. San Francisco: Unity Technologies, © 2018, 21 November 2018 [cit. 2021-02-22]. Dostupné z: http://wiki.unity3d.com/index.php/UnityScript_versus_JavaScript
15. FINNEGAN, Thomas. Learning Unity Android Game Development. Birmingham: Packt Publishing, 2015. ISBN 978-1-78439-469-1.
16. MURRAY, Jeff W. C# game programming cookbook for Unity 3D. Boca Raton: CRC Press, [2014]. ISBN 978-1-4665-8140-1.
17. Unity Asset Store. Unity Asset Store [online]. San Francisco: Unity Technologies, © 2021 [cit. 2021-02-22]. Dostupné z: https://assetstore.unity.com/
18. Colliders. *Unity Documentation: Unity Manual* [online]. San Francisco: Unity Technologies, © 2020 [cit. 2021-03-20]. Dostupné z: https://docs.unity3d.com/Manual/CollidersOverview.html
19. My Little Prince - a jigsaw puzzle tale. *STEAM* [online]. Bellevue: Valve Corporation, © 2021 [cit. 2021-03-02]. Dostupné z: https://store.steampowered.com/app/1399660/My_Little_Prince__a_jigsaw_puzzle_tale/
20. The Little Prince VR. *STEAM* [online]. Bellevue: Valve Corporation, © 2021 [cit. 2021-03-02]. Dostupné z: https://store.steampowered.com/app/707310/The_Little_Prince_VR/
21. The Little Prince VR: Is this game still on development? *STEAM* [online]. Bellevue: Valve Corporation, © 2021, 27. pro. 2017 [cit. 2021-03-02]. Dostupné z: https://steamcommunity.com/app/707310/discussions/0/1620599015878240144/
22. Little Prince Jump. *Google Play* [online]. Mountain View: Google, ©2021 [cit. 2021-03-02]. Dostupné z: https://play.google.com/store/apps/details?id=com.CincoSEIS.LittlePrinceJumps
23. The Little Prince. *Google Play* [online]. Mountain View: Google, ©2021 [cit. 2021-03-02]. Dostupné z: https://play.google.com/store/apps/details?id=com.divertap.games.principito

24. The Little Prince – An animated children's book. *Google Play* [online]. Mountain View: Google, ©2021 [cit. 2021-03-02]. Dostupné z: https://play.google.com/store/apps/details?id=com.larixpress.littleprince
25. The Little Prince Planet. *Itch.io* [online]. itch, ©2021 [cit. 2021-03-02]. Dostupné z: https://maddi9.itch.io/thelittleprinceplanet
26. Asteroid B-612. *Itch.io* [online]. itch, ©2021 [cit. 2021-03-02]. Dostupné z: https://scientistsarepeopletoo.itch.io/asteroid-b-612
27. The Little Prince (Old, bitter and redneck). *Itch.io* [online]. itch, ©2021 [cit. 2021-03-02]. Dostupné z: https://cerebraltiger.itch.io/the-little-prince
28. The little prince. *Itch.io* [online]. itch, ©2021 [cit. 2021-03-02]. Dostupné z: https://terman-emil.itch.io/the-little-prince
29. The Little Prince Game. *Itch.io* [online]. itch, ©2021 [cit. 2021-03-02]. Dostupné z: https://jeremythepug.itch.io/the-little-prince-game
30. PlayerPrefs: description. Unity DOCUMENTATION [online]. San Francisco: Unity Technologies, © 2020, 2021-02-09 [cit. 2021-02-22]. Dostupné z: https://docs.unity3d.com/ScriptReference/PlayerPrefs.html

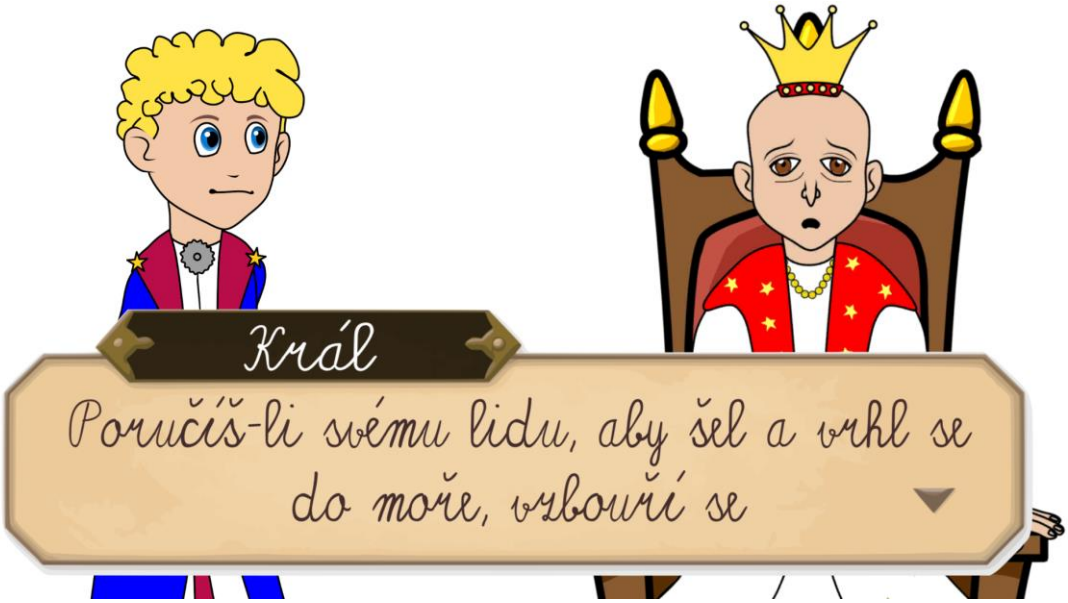
7 Přílohy



Příloha 1 - Obrázek ze hry Malý princ (Hlavní menu)



Příloha 2 - Obrázek ze hry Malý princ (Princova planeta a inventář)



Příloha 3 - Obrázek ze hry Malý princ (Rozhovor s králem)

Příloha 4 - Obrázek ze hry Malý princ (Lampářova planeta)

Příloha 5 - Obrázek ze hry Malý princ (Setkání s pilotem)

Příloha 6 - Obrázek ze hry Malý princ (Interiér princovy chatrče)

Příloha 7 - Obrázek ze hry Malý princ (Testovací kvíz)