

Absolvování individuální odborné praxe

Individual Professional Practice in the Company

Miroslav Kačeriak

Bakalářská práce

Vedoucí práce: Ing. Pavel Dohnálek, Ph.D.

Ostrava, 2021

Abstrakt

Cieľom tejto bakalárskej práce je popísať absolvovanie odbornej praxe v českom nezávislom hernom štúdiu Perun Creative. Prvá časť si kladie za cieľ charakterizovať štúdio ako také a uviesť základnú klasifikáciu vyvíjaného produktu, teda počítačovej hry Hobo: Tough Life. Následne by som chcel špecifikovať moje pracovné zaradenie v rámci štruktúry spoločnosti a uviesť najdôležitejšie technológie, s ktorými som mal možnosť pracovať. Hlavnú časť tejto práce tvorí popis realizácie konkrétnych projektov. Tie sa týkali najmä vývoja pomocných nástrojov, optimalizácie a prevodu hry na iné platformy, ale aj implementácie konkrétnych herných mechaník. Záver je potom venovaný celkovému zhodnoteniu absolvovanej odbornej praxe so zameraním na uplatnené a novonadobudnuté znalosti a zručnosti.

Klíčové slová

Unity engine; C#; herný vývoj, automatizácia, optimalizácia, vývojárske nástroje

Abstract

Main purpose of this bachelor thesis is to describe how my individual professional practise in czech indie game studio Perun Creative went. First part aims to characterize the studio as such and give a basic classification of the developed product, i.e. the computer game Hobo: Tough Life. Afterwards, I would like to specify my position within the company structure and mention the most important technologies that I had the opportunity to work with. The main part of this thesis is dealing with the implementation of specific projects. These were mainly focused on the creation of development tools, optimization and porting of the game to other platforms, but also the implementation of some game mechanics. The conclusion is then aimed to the overall evaluation of the completed individual professional practice with focus on applied and newly acquired knowledge and skills.

Keywords

Unity engine; C#; game development, automatisisation, optimization, development tools

Podakovanie

Rád by som na tomto mieste poďakoval mojím dvom kolegom, a zároveň zakladateľom herného štúdia Perun Creative, Bc. Jánovi Polachovi a Bc. Jirkovi Vašicovi za prejavenú dôveru, trpezlivosť a ochotu pomôcť počas absolvovania odbornej praxe. Podakovanie patrí aj mojej priateľke za podnetné rady a postrehy poskytnuté pri písaní tejto práce.

Obsah

Zoznam použitých symbolov a skratiek	6
Zoznam obrázkov	8
Zoznam výpisov zdrojového kódu	9
1 Úvod	10
2 Popis firmy a pracovného zaradenia	11
2.1 Popis firmy a vyvíjaného produktu	11
2.2 Pracovné zaradenie	12
3 Použité technológie	13
3.1 Programovací jazyk C# a architektúra .NET	13
3.2 Unity Engine	14
3.3 Verzovací systém Git	14
3.4 GitLab CI/CD	15
3.5 Epic Online Services	16
4 Zadané projekty a ich riešenia	17
4.1 Vývoj pomocných nástrojov pre QA oddelenie a programátorov	17
4.1.1 Evidovanie chybových reportov (Mantis Report)	17
4.1.2 Vytváranie „Patch notes“ pre QA oddelenie (GitLogger)	28
4.1.3 Automatizované zostavenie hry na serveri (Build Server)	31
4.2 Optimalizácia a prevod hry na ďalšie platformy	38
4.2.1 Optimalizácia a refaktorovanie kódu	38
4.2.2 Prevod hry na OS Linux	42
4.2.3 Implementácia Epic Online Services	42
4.3 Ostatné projekty	42
4.3.1 Systém líhania hráčskej postavy	42

Zoznam použitých skratiek a symbolov

AI	– Artificial intelligence
API	– Application Programming Interface
CD	– Continuous Delivery
CD	– Continuous Deployment
CI	– Continuous Integration
CLR	– Common Language Runtime
DLL	– Dynamic Link Library
EOS	– Epic Online Services
GC	– Garbage Collector
HTTPS	– Hypertext Transfer Protocol Secure
IL	– Intermediate Language
IP	– Internet Protocol
JIT	– Just-In-Time
JSON	– JavaScript Object Notation
LFS	– Large File Storage
MSS	– Maximum Segment Size
MTU	– Maximum transmission unit
NPC	– Non-playable Character
OS	– Operating System
PC	– Personal Computer
QA	– Quality Assurance
REST	– Representational State Transfer
RPG	– Role Playing Game
SSH	– Secure Shell
SSL	– Secure Sockets Layer
TCP	– Transmission Control Protocol
URL	– Uniform Resource Locator
WPF	– Windows Presentation Foundation

XML – eXtensible Markup Language

Zoznam obrázkov

2.1	Logo spoločnosti Perun Creative	11
3.1	Platformy podporované Unity engine-om [5]	14
3.2	Princíp fungovania Gitab CI/CD [7]	15
4.1	Ručné vloženie reportu do služby Mantis Bug Tracker	18
4.2	Dátový balíček [11]	24
4.3	Užívateľské rozhranie aplikácie Mantis Report	26
4.4	Okno úpravy obrázka aplikácie Mantis Report	27
4.5	Vytvorená karta v aplikácii Microsoft Teams	31
4.6	Custom editor pre scriptable object BuildSettings.asset	35
4.7	Varovanie v custom editore	36
4.8	Namerané hodnoty z nástroja Unity Profiler	40

Zoznam výpisov zdrojového kódu

4.1	Odoslanie nového reportu na server	19
4.2	Odchytenie stlačenia klávesovej skratky v spustenej hre	20
4.3	Ukážka použitia direktívy UNITY_EDITOR	21
4.4	Vytváranie lokálneho a vzdialeného reportu	22
4.5	Metóda ZipAndSendToLinkManager triedy ReportingManager	23
4.6	Postupné čítanie dát zo sieťového prúdu	24
4.7	Odchytenie stlačenia klávesovej skratky v editore Unity	25
4.8	Získanie veľkosti okna Unity editoru	25
4.9	Implementácia kreslenia v prostredí Windows Forms	27
4.10	Metóda na spustenie skriptu v programe Powershell	29
4.11	Dynamické vytváranie vlastného príkazu git log	29
4.12	Regulárny výraz na rozdelenie tela <i>commitu</i>	30
4.13	Vytvorenie a odoslanie správy do služby Microsoft Teams	30
4.14	Obsah súboru .gitlab-ci.yml	33
4.15	Skript zabezpečujúci zostavenie hry na serveri	34
4.16	Načítanie dát zo scriptable objectu	35
4.17	Kostra triedy určenej pre vykreslenie vlastného okna v Editore Unity	36
4.18	Extrahovanie zvolených scén pomocou masky	37
4.19	Extrahovanie zvolených scén pomocou masky	38

Kapitola 1

Úvod

V rámci mojej odbornej praxe som dostal možnosť nahliadnuť za oponu herného vývoja v českom nezávislom štúdiu Perun Creative. Nakoľko sa o hry a herný priemysel dlhodobo zaujímam, táto firma a jej tvorba mi bola vopred známa. Aj po prednáške jej dvoch spoluzakladateľov a zároveň programátorov, ktorá sa uskutočnila v priestoroch Vysokej školy báňskej, som bol stále prekvapený vysokou technologickou úrovňou ich prvého projektu. Firmu som chcel kontaktovať so žiadosťou o prácu nezávisle od odbornej praxe, no keď som sa dozvedel, že už niekoho práve na odbornú prax hľadajú, rozhodol som sa to využiť. Pracovnému pohovoru predchádzalo zaslanie programátorského portfólia zloženého zo školských ale aj vlastných prác. Samotný pohovor potom prebiehal online s oboma programátormi Bc. Jánom Polachom a Bc. Jirkou Vašicou, ktorí sa neskôr stali aj mojimi kolegami.

Do firmy som nastúpil na konci životného cyklu projektu, takže som sa nemohol podieľať na vývoji základných herných mechaník. Naopak to znamenalo nutnosť dôkladne sa s celým projektom zoznámiť a pochopiť, ako jednotlivé časti fungujú. Počas praxe som sa podieľal na širokej škále väčších aj menších projektov z oblastí ako sú automatizácia a zefektívnenie vývojárskych či testerských postupov, nasadenie projektu, sieťová infraštruktúra a došlo aj na nejaké tie herné mechaniky. Pri riešení rôznych problémov mi okrem kolegov boli nápomocné aj rôzne teoretické znalosti nadobudnuté počas vysokoškolského štúdia.

Kapitola 2

Popis firmy a pracovného zaradenia

2.1 Popis firmy a vyvíjaného produktu

Perun Creative s.r.o [1] je české nezávislé herné štúdio, ktoré od roku 2015 vyvíja počítačovú hru Hobo: Tough Life [2].

Hra samotná by sa dala charakterizovať ako RPG z mestského prostredia, kde sa hráč ocitne v role bezdomovca. Ústrednou hernou mechanikou je snaha prežiť v nehostinnom prostredí ulice. Okrem prežitia na hráča čaká aj pútavý príbeh a možnosť hrať kooperatívne až s troma ďalšími hráčmi. Hobo: Tough Life v súčasnosti vychádza na platformách Microsoft Windows a Linux. V budúcnosti sa počíta aj s vydaním na konzolách novej a starej generácie.

Štúdio Perun Creative má aktuálne dve pobočky. Prvá sa nachádza v Ostrave a jej osadenstvo tvoria výhradne programátori. Pobočka v Olomouci naopak slúži pre menej technicky zameranú časť firmy a síce pre grafika, herného dizajnéra a komunitného manažéra. Štúdio tvorí menej ako desať vývojárov a teda sa svojou veľkosťou radí k menším. Využíva ale aj služby externých pracovníkov prípadne spoločností špecializovaných na testovanie, zvukovú stránku hry a v neposlednom rade aj na preklad textov do rôznych svetových jazykov.



Obr. 2.1: Logo spoločnosti Perun Creative

2.2 Pracovné zaradenie

Môj prínos štúdiu Perun Creative spočíval hlavne v automatizácii a zefektívnení jednotlivých interných postupov resp. vývoji pomocných nástrojov pre QA oddelenie či ostatných programátorov. Do tejto kategórie by som zaradil projekty ako automatizované zostavenie hry na serveri, zefektívnenie spôsobu nahlasovania jednotlivých problémov prípadne prepracovanie importovania objektov zo starej verzie projektu do novej. K zefektívneniu práce rozhodne prispel aj návrh a implementácia systému klávesových skratiek v hernom engine Unity (bližšie popísanom v sekcii 3.2) či program na vytvorenie tzv. „Patch notes“. Ten bol primárne určený pre testerov, ale do budúcnosti sa plánuje jeho využitie aj pri informovaní hráčov o novinkách v rámci hry.

Okrem vyššie uvedených projektov som pracoval na novom systéme líhania postavy k spánku, optimalizácii a uvedeníu hry na OS Linux, prevode online časti hry z platformy Steam na platformu EOS a ďalších menších projektoch.

Kapitola 3

Použité technológie

V nasledujúcej kapitole by som rád v krátkosti zhrnul najdôležitejšie technológie, s ktorými som sa v rámci odbornej praxe stretol. Niektoré som aktívne využíval počas celého obdobia praxe a ich osvojovanie teda prebiehalo organicky. Iné boli špecifické pre konkrétny projekt a danú technológiu som si musel naštudovať počas realizácie projektu. V niektorých prípadoch bolo nutné sa zoznámiť s viacerými technológiami, aby som bol schopný posúdiť ich výhody a nevýhody pri nasadení na konkrétny projekt a vybrať tú správnu. Informácie som poväčšine čerpal z dokumentácií k daným technológiám aj keď nie vždy bola ich úroveň dostatočná.

3.1 Programovací jazyk C# a architektúra .NET

C# [3] je objektovo orientovaný, typovo bezpečný programovací jazyk umožňujúci vytvárať aplikácie v .NET ekosystéme. Syntax jazyka C# vychádza z programovacích jazykov C a C++. Narozdiel od nich ale ponúka vyššiu úroveň abstrakcie. Tá sa prejavuje napríklad na úrovni správy pamäte, čo z jazyka C# robí voľbu číslo jedna pre začínajúcich programátorov, ktorí by si chceli skúsiť herný vývoj na vlastnej koži.

Architektúra .NET [3] potom ponúka okrem virtuálneho stroja CLR, ktorý vykonáva JIT kompiláciu IL kódu do strojových inštrukcií, aj sadu veľmi užitočných knižníc. Tieto knižnice sú organizované do menných priestorov a poskytujú širokú škálu metód vhodných napríklad na prácu so súborami či sieťovou infraštruktúrou. Veľmi užitočné sú aj nástroje na syntaktickú analýzu XML prípadne platforma Windows Forms, ktorá sa v rámci mojej odbornej praxe vo firme Perun Creative ukázala byť vhodná hlavne na rýchlu tvorbu vývojárskych nástrojov. Kvôli limitáciám tejto technológie sa ale v budúcnosti plánuje prechod na systém WPF.

3.2 Unity Engine

Unity Engine [4] je platforma pre tvorbu 2D a 3D interaktívneho obsahu renderovaného v reálnom čase. Najväčšie uplatnenie nachádza pri tvorbe malých až stredne veľkých hier, ale je čoraz častejšie využívaný aj v iných odvetviach ako napríklad automobilový či filmový priemysel. Je dostupný pre operačné systémy Windows, Linux a Mac OS. Distribuuje sa zdarma pre študentov alebo jednotlivcov do určitého finančného obratu a za ročný poplatok pre firmy, ktorého výška závisí od rôznych faktorov. Samotný engine je napísaný v jazyku C++ a umožňuje vývoj obsahu v jazyku C# (bližšie popísanom v sekcii 3.1) pre širokú škálu platforiem, čo znázorňuje obrázok 3.1.



Obr. 3.1: Platformy podporované Unity engine-om [5]

3.3 Verzovací systém Git

Git [6] je open source distribuovaný systém na správu verzií vyvinutý Linusom Thorvaldsom a komunitou okolo OS Linux v roku 2005. Umožňuje vrátiť celý projekt alebo vybrané časti do predchádzajúceho stavu, porovnávať zmeny v súboroch, prípadne efektívnu kolaboráciu viacerých vývojárov na spoločnom projekte. Každý člen tímu má k dispozícii kompletný repozitár vrátane histórie jednotlivých súborov, čo pridáva ďalšiu vrstvu ochrany proti výpadkom a poruchám hardvéru. Po každom spustení príkazu *commit* sa uloží aktuálny stav projektu, ktorý je možné spätne dohľadať v prípade nejakého problému.

S gitom je možné pracovať priamo z príkazového riadku ale pre väčšie projekty ako je aj hra Hobo: Tough Life je takýto prístup značne nepraktický a neefektívny. Existuje ale množstvo programov tretích strán, ktoré umožňujú pracovať s gitom z užívateľského rozhrania. Jedným takým je aj nástroj Sourcetree, s ktorým som v rámci mojej odbornej praxe pracoval.

Nakolko git samotný nie je dobre prispôbený na správu verzií veľkých súborov, bolo nutné využívať aj jeho rozšírenie Git LFS.

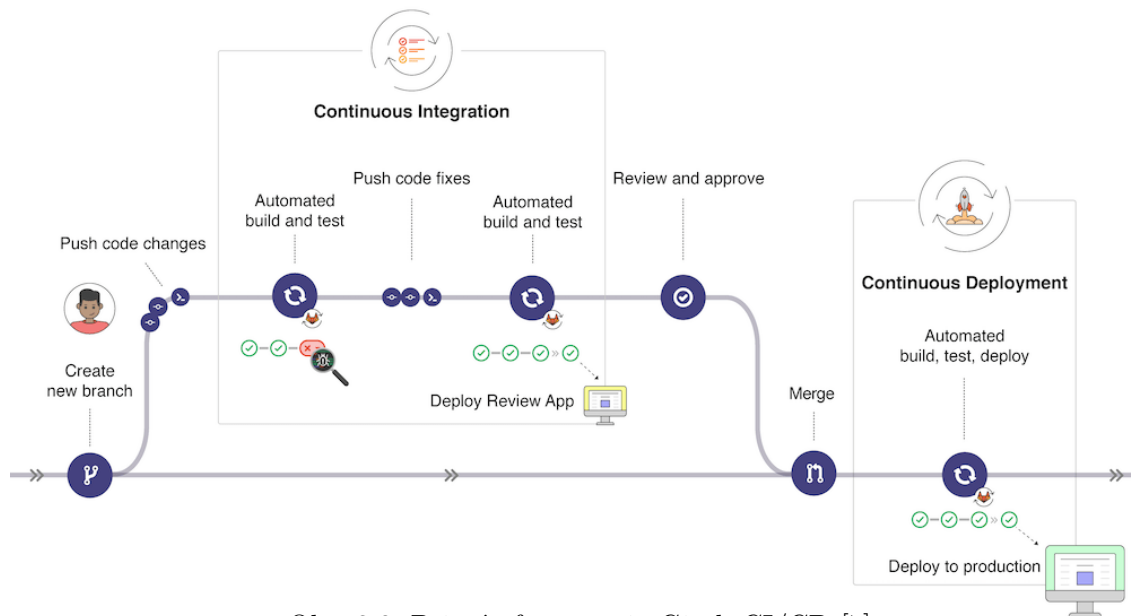
3.4 GitLab CI/CD

Gitlab CI/CD [7] je súčasť nástroja Gitlab a slúži na vývoj softvéru prostredníctvom kontinuálnych metód ako:

- **Continuous Integration (CI)** - pre každé pridanie zmien do repozitára je možné automatizovane spustiť sadu skriptov, ktorých účelom je projekt zostaviť či otestovať.
- **Continuous Delivery (CD)** - pridáva ďalšiu vrstvu nad rámec CI, umožňuje zostavený a otestovaný kód nasadiť, vyžaduje to ale manuálnu akciu.
- **Continuous Deployment (CD)** - funguje podobne ako Continuous Delivery ale nasadenie prebieha automaticky.

Spoločnou filozofiou týchto metód je teda po každej iteratívnej zmene v projekte kód zostaviť, otestovať a prípadne aj nasadiť. Hlavnou výhodou takého prístupu je redukovanie množstva chýb, ktoré by sa inak dostávali do ďalších iterácií a mohli by spôsobiť problémy, keby neboli odchytené v zárodku. Zároveň si kladú za cieľ znížiť potrebu manuálneho zásahu do jednotlivých automatizovaných procesov na minimum.

V rámci mojej odbornej praxe som sa stretol hlavne s metódou CI. Možnosti automatizovaného testovania hry touto metódou sú veľmi obmedzené, o to viac je ale užitočné automatizované zostavenie na serveri. Zostavenie tohto typu projektu na pracovnej stanici je typicky náročné ako časovo tak aj na výpočtový výkon a do značnej miery teda spomaľuje vývoj ako taký. Bolo teda logickým krokom zamerať sa v rámci optimalizácie interných procesov aj na tento aspekt.



Obr. 3.2: Princíp fungovania Gitab CI/CD [7]

3.5 Epic Online Services

Epic Online Services alebo EOS [8] je skupina nástrojov od spoločnosti Epic Games, ktoré sa snažia zjednodušiť a zjednotiť niektoré aspekty herného vývoja naprieč platformami. Typicky sa zameriavajú na kooperatívnu či kompetetívnu zložku hry pre viacerých hráčov, zahŕňajú však aj služby ako nákup herných predmetov za reálne peniaze, zber hráčskych štatistik, prípadne získavanie herných úspechov. Za normálnych okolností je nutné tieto služby implementovať zvlášť takmer pre každú platformu, na ktorú bude hra vychádzať. EOS sú zdarma dostupné pre vývojárov a podporujú širokú škálu platforiem, čo znamená väčšiu slobodu z pohľadu hráča ale aj z pohľadu vývojára. Hráč môže ďalej ťažiť aj z online synchronizácie postupu hrou naprieč viacerými platformami, prípadne z prístupu k úspechom či priateľom z jedného centrálného miesta.

Nakoľko sa do budúcnosti počíta s vydaním hry *Hobo: Tough Life* na rôzne platformy, ukázalo sa použitie EOS ako veľmi výhodné na náhradu doterajšieho systému pre hru viacerých hráčov. Hlavnou výhodou tohto riešenia by bola vyššie spomínaná multiplatformovosť. V dobe písania tohto textu sa však počíta s nasadením EOS spolu s niektorými službami dostupnými len na konkrétnej platforme.

Kapitola 4

Zadané projekty a ich riešenia

V tejto kapitole by som rád podrobne prebral konkrétne projekty na ktorých som v rámci mojej odbornej praxe pracoval. Poradie v akom budú prezentované nutne neodzrkadľuje poradie ich vypracovania, ale budú usporiadané do logických celkov podľa typu projektu.

Čo sa časovej náročnosti týka, nie vždy je možné ju určiť presne nakoľko niektoré z projektov boli vyvíjané inkrementálne podľa spätnej väzby QA oddelenia alebo nových požiadaviek od ostatných programátorov.

4.1 Vývoj pomocných nástrojov pre QA oddelenie a programátorov

4.1.1 Evidovanie chybových reportov (Mantis Report)

Časová náročnosť:

Nasadenie: 8 dní, pridanie ďalšej funkcionality podľa požiadaviek: spolu 8 dní.

Úvod do problému:

Ako som spomenul v sekcii 3.4 možnosti automatizovaného testovania hry tohto rozsahu sú na rozdiel od bežného softvéru značne obmedzené. Veľké štúdiá si vytvárajú vlastné nástroje založené na AI, nikdy sa to však úplne neobíde bez tvrdej práce ľudí z QA oddelenia ľudovo nazývaných aj ako tester. Úlohou testera je teda hrať celú hru alebo jej určené časti, nájsť a následne ohlásiť nájdené chyby.

Takéto ohlásenie chyby je značne neefektívny proces nakoľko na pozadí zahŕňa hneď niekoľko ďalších krokov ako minimalizácia hry, otvorenie stránky so službou Mantis Bug Tracker, nahranie snímky obrazovky a nakoniec aj samotné vyplnenie detailov reportu. Niektoré z týchto detailov sa navyše medzi jednotlivými reportmi nemenia ako napríklad informácie o platforme či operačnom systéme. Ručné vyplnenie reportu znázorňuje obrázok 4.1. Táto neefektívnosť samozrejme priamo úmerne rastie s počtom nájdených chýb, čo môžu byť aj vyššie jednotky denne. Mojou úlohou bolo teda optimalizovať tento postup.

Navrhované riešenia:

Nakoľko bola toto moja prvá úloha v rámci odbornej praxe, mal som len minimálne skúsenosti s webovými technológiami ako je REST API a celkovo klient-server architektúrou, navrhol som riešenie postavené na nejakom nástroji na automatizovanie webového prehliadača. Takýmto nástrojom je napríklad Selenium WebDriver, s ktorým som sa stretol pri práci na vlastných projektoch. Ten je veľmi obľúbený napríklad pri testovaní webových aplikácií. Dokázal by zapnúť webový prehliadač, či už s užívateľským rozhraním alebo bez, otvoriť požadovanú stránku a vyplniť detaily reportu za užívateľa. V spojení s nejakým vlastným nástrojom na ukladanie snímky obrazovky, prípadne nástrojom, ktorý by testerovi umožnil vypísať zhrnutie či popis reportu priamo v hre by sa naozaj jednalo o relatívne dobré riešenie. Po užívateľovi by to ale vyžadovalo nutnosť inštalácie nejakého konkrétneho prehliadača v požadovanej verzii, aby bola zaistená správna kompatibilita a veľmi pravdepodobne by sa v budúcnosti objavili aj ďalšie problémy s nasadením či používaním.

Po preskúmaní ďalších možností a následnej porade s kolegami som sa teda rozhodol dať prednosť riešeniu postavenému na už spomínanom REST API a ak by sa to ukázalo ako nerealizovateľné spätne sa vrátiť k môjmu prvému nápadu.

✎ Vložte detaily reportu							
* Kategória	Gameplay ▾						
Reprodukovateľnosť	vždy ▾						
Dôležitosť	veľká ▾						
Priorita	urgentná ▾						
Vybrať profil	ALEBO vyplň ▲ <table><tr><td>Platforma</td><td>Windows</td></tr><tr><td>OS</td><td>Windows 10</td></tr><tr><td>Verzia</td><td>(10.0.0) 64bit</td></tr></table>	Platforma	Windows	OS	Windows 10	Verzia	(10.0.0) 64bit
Platforma	Windows						
OS	Windows 10						
Verzia	(10.0.0) 64bit						
Verzia produktu	1.00.008 ▾						
Priradený	mkaceriak ▾						
Cieľová verzia	1.00.008 ▾						
*Zhrnutie	Krátke zhrnutie						
*Popis	Podrobnejší popis						
Kroky k vyvolaniu	1. ... 2. ...						

Obr. 4.1: Ručné vloženie reportu do služby Mantis Bug Tracker

Realizácia:

Pri preskúvaní možností založených na REST API som narazil na open source projekt MantisSharp [9]. Po zoznámení sa s týmto projektom a niekoľkými pokusmi ho „ohnúť“ pre účely môjho projektu som sa rozhodol, že bude jednoduchšie napísať si aplikáciu sám. Využil som k tomu dve triedy z projektu MantisSharp, a sice *RestClient* a *MantisClient*, ktoré som následne ešte ďalej modifikoval. Trieda *RestClient* vykonáva najnižšiu úroveň komunikácie so serverom a sice odosiela GET a POST žiadosti. Trieda *MantisClient* má potom dve úlohy. Na jednej strane prijíma dáta z mojej aplikácie, zabalí ich do formy vhodnej pre transport a následne ich predá triede *RestClient* na odoslanie metódou POST. Na strane druhej transformuje dáta získané zo servera pomocou metódy GET do C# tried vhodných na následné použitie. Tento postup pridáva určitý level abstrakcie do celej aplikácie.

Problematiku odosielania dát na server som sa rozhodol demonštrovať na pridaní nového reportu vo výpise 4.1. Aplikácia vytvorí na základe vstupných dát inštanciu triedy *Issue*, ktorú predá metóde *SendIssue*. Tá ju následne zabalí ako JSON objekt a ďalej predá v metóde *ExecutePost* na finálne odoslanie. Získavanie dát funguje obdobne len opačným smerom.

Služba Mantis Bug Tracker po úspešnom pridaní nového reportu tento report vráti v odpovedi, čo som ďalej využil na informovanie užívateľa o úspešnom zaevidovaní, ktoré som doplnil o serverom pridelený identifikátor.

```
public int SendIssue(Issue issue) {
    string uri = this.GetUri(issuesUri);
    int id = -1;

    restClient.ExecutePost(uri, () => JsonConvert.SerializeObject(issue),
        reader => {
            string answerFromServer = reader.ReadToEnd();
            JObject obj = JObject.Parse(answerFromServer);
            id = (int)obj["issue"]["id"];
        });

    return id;
}
```

Výpis 4.1: Odoslanie nového reportu na server

Server spočiatku na všetky žiadosti reagoval chybou „401 Unauthorized“. Po dôkladnom preskúmaní problému sa ukázalo, že chyba je na strane serveru a bolo nutné zasiahnuť do jeho kódu. Nakoľko som k tomuto kódu nemal prístup, požiadal som kolegu, či by mi s tým mohol pomôcť. Ukázalo sa,

že server filtroval všetky požiadavky, ktoré sa pokúšali o autorizáciu pomocou tzv. „Bearer tokenu“. Spoločne sa nám tento problém však podarilo vyriešiť.

Systém nahlasovania reportov mal byť pôvodne implementovaný ako súčasť hry. To sa ale ukázalo ako problematické z hľadiska zachytávania užívateľského vstupu v Unity engine. V praxi by to znamenalo, že ak by užívateľ popisoval chybu napríklad do nejakého textového poľa, hra samotná by naďalej vykonávala akcie na základe stlačených kláves. Rozhodli sme sa teda začleniť tento systém do už existujúceho vývojárskeho nástroja s názvom HoboThor. Tento nástroj je veľmi komplexný a jeho popis by bol nad rámec tejto kapitoly.

Logika aplikácie bola teda rozdelená do dvoch častí. Časť, s ktorou interaguje užívateľ bola realizovaná ako Windows Forms aplikácia začlenená do nástroja HoboThor a časť, ktorá túto aplikáciu spustí bola implementovaná priamo do hry a vyvolá sa stlačením klávesovej skratky Shift + F11, ak je hra spustená s podporou vývojárskych nástrojov.

Kontrola, či boli stlačené konkrétne klávesy musí prebiehať každú snímku, aby sa zabezpečilo, že odchytenie prebehne úspešne. Kód zabezpečujúci túto funkcionality bol teda vložený do metódy *OnUpdate* triedy *ReportingManager*, čo znázorňuje výpis 4.2.

Projekt Hobo: Tough Life do veľkej miery stojí na hierarchickej štruktúre tried, tzv. „Manageroch“, ktorí, ako názov napovedá obstarávajú určité časti aplikácie. Metóda *OnUpdate* triedy *ReportingManager* je teda každú snímku volaná inou triedou, ktorá je vyššie v hierarchickej štruktúre projektu. Na najvyššom stupni hierarchie potom stojí trieda *MainManager*. Tá obsahuje metódu *Update*, ktorá je priamo volaná natívnym C++ kódom enginu Unity a táto metóda potom obsahuje volania metód *OnUpdate* jednotlivých „Managerov“, ktorí jej náležia.

```
public void OnUpdate() {
    if (Keyboard.current.f11Key.wasPressedThisFrame) {
        if (Keyboard.current.leftShiftKey.isPressed) {
            screenSaved = false;
            StartCoroutine(StartCreateReport());
            return;
        }
    }

    if (screenSaved) {
        ZipAndSendToLinkManager();
        screenSaved = false;
    }
}
```

Výpis 4.2: Odchytenie stlačenia klávesovej skratky v spustenej hre

Kód vo výpise 4.2 teda každú snímku testuje, či bola stlačená klávesa F11 a to práve v tom danom snímku. Toto zabezpečí, že sa blok kódu tejto podmienky vykoná len raz, bez ohľadu na to, ako dlho užívateľ danú klávesu držal stlačenú. Ak je popri tom stlačená aj klávesa Shift, spustí sa tzv. „Coroutine“. V tomto prípade je ňou mnou definovaná metóda *StartCreateReport*. Výhoda korutiny spočíva v možnosti pozastaviť vykonávanie svojho kódu. Pozastavenie môže byť na programátorom definovanú dobu alebo do doby než nastane určitá udalosť. V tomto prípade bolo tou udalosťou kompletne vykreslenie aktuálneho snímku a to z dôvodu zachytenia tohto snímku ako obrázku.

Metóda *StartCreateReport* má za úlohu zozbierať rôzne dáta o aplikácií alebo systéme, na ktorom je spustená. Ide o dáta ako verzia aplikácie, pozícia a rotácia kamery v momente vyvolania akcie či posledný záznam v logovacom systéme. Tieto dáta sú následne spolu so spomínanou snímku obrazovky a naposledy uloženým postupom hrou uložené na disk. Zložka, do ktorej sú súbory uložené závisí od toho, či je hra spustená z editoru Unity alebo samostatne. Takéto vetvenie kódu sa realizuje pomocou direktívy `UNITY_EDITOR`, ktorá je spolu s ďalšími platformovo závislými direktívami definovaná samotným editorom. Jednoduchý príklad použitia takýchto direktív demonštruje výpis 4.3. Nakoľko táto zložka obsahuje celú históriu reportov, jednotlivé reporty majú poradové čísla a každý ďalší dostane pridelené poradové číslo o jedna väčšie ako ten predchádzajúci.

```
#if UNITY_EDITOR
    Debug.Log("Editor");
#else
    Debug.Log("Standalone");
#endif
```

Výpis 4.3: Ukážka použitia direktívy `UNITY_EDITOR`

Následne sa logika aplikácie delí na dve vetvy. Prvá obstaráva vytvorenie lokálneho reportu, druhá vzdialeného. Možnosť vzdialeného nahlasovania chýb, teda odoslanie reportu z PC na ktorom hra nie je spustená bola pridaná až neskôr v súvislosti s prevodom hry na iné platformy. Oba spôsoby odoslania reportu potom demonštruje výpis 4.4.

Odoslanie reportu na server z lokálneho PC pokračuje vytvorením tzv. „Mantis argumentu“. Ide o textový súbor, ktorého obsah je cesta k naposledy vytvorenému reportu. Ten je uložený do zložky, kde sa nachádza spúšťač súbor nástroja HoboThor. Následne je tento nástroj spustený a pokiaľ je pri tomto procese prítomný už spomínaný „Mantis argument“, nespustí sa hlavné okno aplikácie, ale len okno určené na odoslanie reportu. Po prečítaní obsahu je súbor samozrejme zmazaný, aby neovplyvnil ďalšie spustenie nástroja HoboThor.

Pokiaľ ide o odoslanie reportu zo vzdialeného PC ukázala sa ako problematická doba ukladania snímky obrazovky na disk. Kvôli tomuto problému musí aplikácia najskôr počkať na uloženie súboru a až následne vykonávať ďalšie inštrukcie. Toho som docielil jednoduchou slučkou, ktorá sa sama

ukončí v prípade nájdania požadovaného súboru alebo po pretečení určitého času. To slúži ako ochrana pred zacyklením.

```
// Local report
if (HBTLLink_Manager.sender_ServerIP == HBTLLink_Manager.Instance.LocalIPAddress().
    ToString()) {
    string exeFile = GameConfiguration.pathConfig.exeFile;
    if (File.Exists(exeFile)) {
        var argFile = GameConfiguration.pathConfig.mantisArguments;

        if (File.Exists(argFile))
            File.Delete(argFile);

        File.WriteAllText(argFile, directoryReport.FullName);
        Application.OpenURL(exeFile);
    }
}
// Remote report
else {
    float time = 0;
    while (time <= maxWaitTime) {
        yield return new WaitForSeconds(.1f);
        time++;
        if (File.Exists(screenFileName)) {
            screenSaved = true;
            yield break;
        }
    }
}
```

Výpis 4.4: Vytváranie lokálneho a vzdialeného reportu

Na zmenu hodnoty premennej *screenSaved* zareaguje metóda *OnUpdate* z výpisu 4.2 a spustí metódu *ZipAndSendToLinkManager*. Tá pomocou knižnice DotNetZip [10] prevedie celú zložku s reportom na zip súbor so zachovaním hierarchickej štruktúry podzložiek a súborov. Následne ho prevedie na reťazec s kódovaním base64 a predá triede HBTLLink_Manager na odoslanie do vzdialeného PC. Táto metóda je znázornená vo výpise 4.5.

Rreťazec je následne pomocou TCP protokolu odoslaný na IP adresu určenú v konfiguračnom súbore alebo vybratú z prednastavených možností vo vývojárskej konzole počas hrania. Na vzdialenom

PC musí byť spustený nástroj HoboThor a „počúvať“ na určenom porte. Dáta sú po prijatí uložené do dočasnej zložky poskytovanej operačným systémom pomocou metódy *Path.GetTempPath* v mennom priestore *System.IO* a následne rozbalené. Cesta k tejto zložke je opäť zapísaná do „Mantis argumentu“ a HoboThor je následne automaticky spustený znova. To vyústi k otvoreniu okna určeného na nahlasovanie chýb a načítaniu dát zo zložky s reportom.

```
private void ZipAndSendToLinkManager() {
    string pathToZip = directoryReport.FullName + @"\report.zip";

    using (ZipFile zip = new ZipFile()) {
        var files = Directory.EnumerateFiles(directoryReport.FullName, ".*.*",
            SearchOption.AllDirectories);
        foreach (string file in files) {
            if (file.Contains("Characters"))
                zip.AddFile(file, @"\Saves\Characters");
            else if (file.Contains("Worlds"))
                zip.AddFile(file, @"\Saves\Worlds");
            else
                zip.AddFile(file, "");
        }

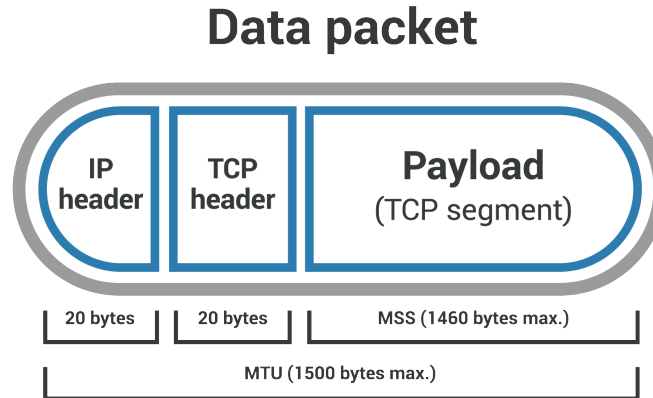
        zip.Save(pathToZip);
        Debug.Log("Zipped to " + pathToZip);
    }

    if (File.Exists(pathToZip)) {
        string base64zip = System.Convert.ToBase64String(File.ReadAllBytes(
            pathToZip));
        HBTLink_Manager.Send_MantisReport(base64zip);
        File.Delete(pathToZip);
    }
}
```

Výpis 4.5: Metóda ZipAndSendToLinkManager triedy ReportingManager

Problém tohto postupu bol spočiatku vo veľkosti vyrovnávacej pamäte na strane prijímateľa, teda nástroja HoboThor, ktorá bola poddimenzovaná. Následné testovanie ukázalo, že spoľahlivá veľkosť tejto pamäte začína až na hodnote 2^{20} bytov v závislosti primárne od veľkosti odosielanej snímky obrazovky. Vytvorenie takejto veľkej vyrovnávacej pamäte a následné uloženie všetkých

prijatých dát naraz ale nefungovalo na OS Linux. Testovacia aplikácia na tomto systéme zvládla prijať maximálne 14600 bytov. Táto hodnota odpovedá desaťnásobku MTU balíčku preneseného technológiou Ethernet v2 po odčítaní veľkostí TCP a IP hlavičiek. Veľkosť 1460 bytov sa zvykne nazývať aj MSS. Túto problematiku znázorňuje obrázok 4.2.



Obr. 4.2: Dátový balíček [11]

Riešením bolo použitie vyrovnávacej pamäte o veľkosti 2^{10} bytov a postupné prijímanie dát zo sieťového prúdu pomocou cyklu. Gro tohto postupu je zobrazené vo výpise 4.6.

```
TcpClient client = reciever_Listener.AcceptTcpClient();
NetworkStream nwStream = client.GetStream();

while ((i = nwStream.Read(buffer, 0, buffer.Length)) != 0)
{
    data += Encoding.ASCII.GetString(buffer, 0, i);
}
```

Výpis 4.6: Postupné čítanie dát zo sieťového prúdu

Mojou poslednou úlohou v časti reportovacieho systému na strane hry, ktorá prišla ako požiadavka od programátorov bolo pridať možnosť odosielať reporty priamo z editoru bez nutnosti mať spustenú hru. Typický prípad použitia bolo nahlasovanie chýb nájdených popri práci s náhľadom scény. Väčšina kódu bola prebratá z vyššie popísanej časti, museli byť však zmenené niektoré postupy.

Odchytávanie klávesovej skratky bolo tentoraz realizované natívnym postupom zabudovaným do editora Unity. Stačilo vytvoriť statickú metódu *CreateReport* a nad jej definíciu pridať atribút „MenuItem“ ako znázorňuje výpis 4.7. To vyústí k vytvoreniu zástupcu tejto metódy v hornom paneli editora. Pozícia metódy v paneli je určená zadanou cestou v parametri atribútu „MenuItem“. Za túto cestou je potom možné pridať ľubovoľnú klávesovú skratku, ktorá túto akciu vyvolá.

```
[MenuItem("Tools/HBT/Reporting/Create report #F11")] // # -> Shift
static void CreateReport()
{
    // body...
}
```

Výpis 4.7: Odchytenie stlačenia klávesovej skratky v editore Unity

Pôvodný postup vytvorenia snímky obrazovky pomocou metódy *CaptureScreenshotAsTexture* bolo nutné nahradiť metódou *ReadScreenPixel*, ktorá ale potrebuje vedieť presnú lokáciu a rozmery okna. Získať tieto údaje sa mi nakoniec podarilo až volaním C++ metód z user32.dll. Počas testovania na 4K monitore sa ukázalo, že tento prístup nevhodne vyhodnocuje veľkosť okna s nastaveným škálovaním v OS Windows. Toto bolo nutné upraviť ručne, čo ukazuje výpis 4.8.

```
[DllImport("user32.dll")]
private static extern bool GetWindowRect(IntPtr hwnd, ref Rect rectangle);
[DllImport("user32.dll")]
private static extern IntPtr GetActiveWindow();

private struct Rect {
    public int Left { get; set; }
    public int Top { get; set; }
    public int Right { get; set; }
    public int Bottom { get; set; }
}

private static Rect GetUnityBounds() {
    Rect unityWindowBounds = new Rect();
    float scale = Screen.dpi / 96;

    GetWindowRect(GetActiveWindow(), ref unityWindowBounds);

    unityWindowBounds.Right = (int)(unityWindowBounds.Right / scale);
    unityWindowBounds.Bottom = (int)(unityWindowBounds.Bottom / scale);

    return unityWindowBounds;
}
```

Výpis 4.8: Získanie veľkosti okna Unity editoru

Na strane Windows Forms aplikácie implementovanej do nástroja HoboThor bolo nutné najskôr vytvoriť sadu tried a enumov, ktoré by po serializácií/deserializácií presne kopírovali štruktúru projektu či reportu s ktorou pracuje služba Mantis Bug Tracker. Následne bolo nutné navrhnuť užívateľské rozhranie. To prechádzalo iteratívnymi zmenami na základe spätnej väzby až do svojej finálnej podoby znázornenej na obrázku 4.3.

Obr. 4.3: Užívateľské rozhranie aplikácie Mantis Report

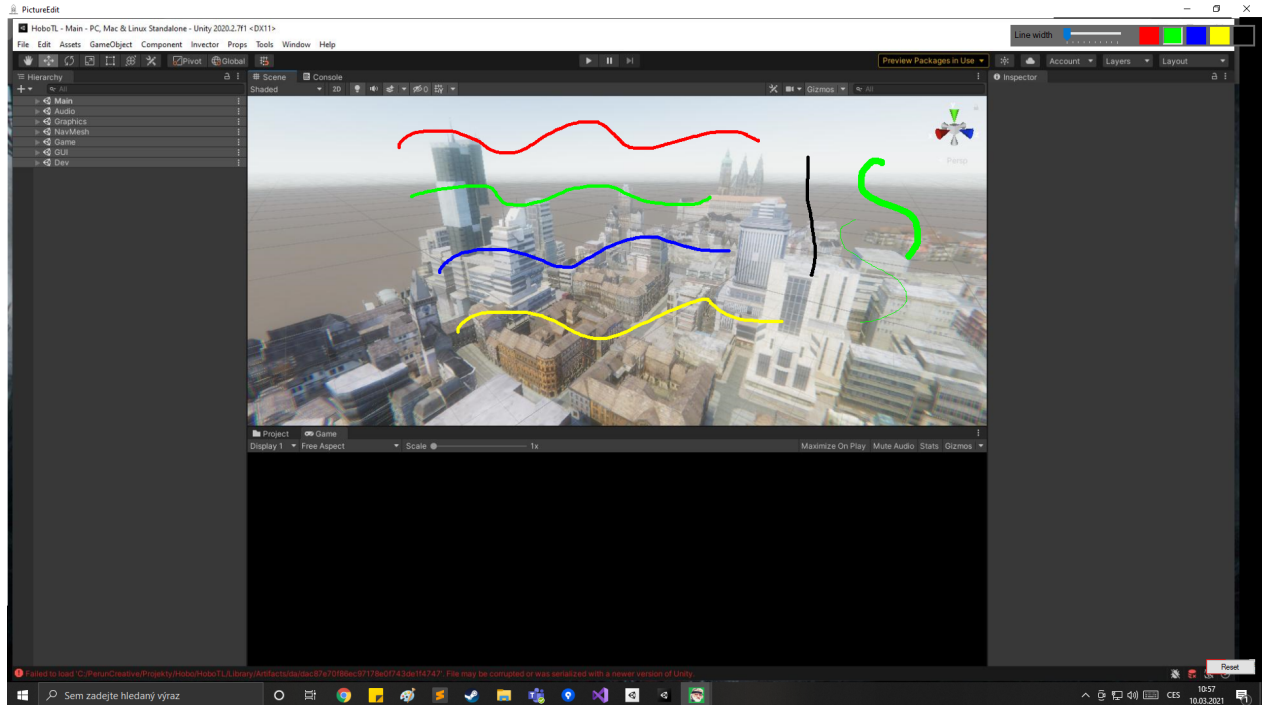
Po spustení aplikácie sa načíta uložená snímka obrazovky a zobrazí sa v náhľadovom okne na pravej strane. Vzhľadom na to, že v dobe spúšťania tento obrázok ešte nemusí byť plne k dispozícii je táto funkcionality realizované v separátnom vlákne pomocou tzv. „Background Workera“. Ten v cykle prehľadáva zložku s reportom kvôli prítomnosti obrázku a až následne, keď je k dispozícii ho zobrazí.

Po načítaní dát z report zložky do polí „Shrnutí“ a „Popis“ sú odoslané dve GET žiadosti na server. Prvá požaduje dáta o užívateľovi, ktorému náleží príslušný api kľúč odoslaný spolu so žiadosťou a druhá požaduje informácie o projekte. Medzi tieto informácie patria aj kategórie evidované pre daný projekt. Tie sú vložené do kombinovaného poľa „Kategorie“. Tento postup bol zvolený z dôvodu možných budúcich zmien v štruktúre kategórií na serveri. Ostatné kombinované polia boli vyplnené vopred pripravenými enumami. U nich sa žiadna budúca zmena nepredpokladá.

Užívateľ zadá dostupné informácie k nájdennej chybe a report odošle na server. Táto akcia zahŕňa získanie dát zo všetkých dostupných polí a následné vytvorenie inštancie triedy *Issue*, ktorá tieto dáta zapúzdruje. Všetky prílohy určené na odoslanie sú potom uložené do príslušných polí

bytov, prevedené na reťazce s kódovaním base64 obdobným postupom ako vo výpise 4.5 a predané spomínanej inštancii.

V tomto momente bola aplikácia prakticky hotová ale z QA oddelenia prišla požiadavka na možnosť úpravy obrázka pred odoslaním bez nutnosti použitia externých nástrojov. Vytvoril som teda druhé okno pre túto aplikáciu, ktoré sa otvorí po kliknutí na náhľadový obrázok a umožňuje do neho kresliť. Toto okno je znázornené na obrázku 4.4.



Obr. 4.4: Okno úpravy obrázka aplikácie Mantis Report

Postupne bola pridaná podpora viacerých hrúbok a farieb čiar či tlačítka reset, ktoré slúži na návrat k pôvodnému obrázku. Zmeny v tomto okne sú spätne reflektované aj náhľadovým obrázkom v hlavnom okne reportovacej aplikácie.

Na kreslenie bolo využité vlastné riešenie založené na udalostiach *MouseDown*, *MouseUp* a *MouseMove*. Pri pohybe myši spolu so stlačeným ľavým tlačidlom dôjde k vykresľovaniu čiary medzi dvoma bodmi, čo sa od určitej frekvencie zaznamenávania bodov javí ako súvislý ťah. Tento postup je demonštrovaný vo výpise 4.9.

```
private void pictureBoxBg_MouseDown(object sender, MouseEventArgs e) {  
    moving = true;  
    x = e.X;  
    y = e.Y;  
}
```

```

private void pictureBoxBcg_MouseMove(object sender, MouseEventArgs e) {
    if (moving && x != -1 && y != -1) {
        g.DrawLine(pen, new Point(x, y), e.Location);
        x = e.X;
        y = e.Y;
        pictureBoxBcg.Invalidate();
    }
}

private void pictureBoxBcg_MouseUp(object sender, MouseEventArgs e) {
    moving = false;
    x = -1;
    y = -1;
}

```

Výpis 4.9: Implementácia kreslenia v prostredí Windows Forms

4.1.2 Vytváranie „Patch notes“ pre QA oddelenie (GitLogger)

Časová náročnosť:

Nasadenie: 3 dni, následné úpravy: 1 deň.

Úvod do problému:

S blížiacim sa dátumom vydania hry sa zvyšuje aj frekvencia testovania a rýchlosť s akou pribúdajú opravy chýb. Medzi jednotlivými zostaveniami aplikácie býva nejaké časové obdobie - zvyčajne jeden týždeň. Toto obdobie je zakončené nahraním najnovšie zostavenej verzie hry na platformu Steam, kde ju môže QA oddelenie začať testovať. Informovanie testerov o najnovších zmenách a opravách môže byť pracné a náchylné na vynechanie niektorých dôležitých vecí. Mojou úlohou bolo tento proces pokiaľ možno čo najviac optimalizovať.

Navrhované riešenia:

Prvým navrhovaným riešením, ktoré sa už využívalo v niektorých vývojárskych nástrojoch bolo použitie knižnice, ktorá sa po zadaní správnych prihlasovacích údajov pripojí na Git server a prevedie prítomné *commity* na štruktúru tried, s ktorou je potom možné ďalej pohodlne pracovať. Po preskúmaní tohto riešenia sa ukázalo, že jednotlivé knihovne nemajú vyriešenú podporu pripojenia na server pomocou protokolu SSH ale iba HTTPS, čo sa ukázalo ako veľký problém. Prišiel som teda s riešením, ktoré by si stiahlo *commity* iba z lokálneho repozitára a to pomocou PowerShellu v OS Windows.

Realizácia:

Po založení testovacieho projektu na platforme .NET bolo mojím prvým krokom zistiť, ako programovo spustiť rôzne skripty a príkazy v programe Windows PowerShell či klasickom príkazovom

riadku. Postupne som narazil na triedu *PowerShell* určenú presne pre tento prípad použitia. Pomocou tejto triedy som si vytvoril metódu, ktorá vie spustiť ľubovoľný príkaz a vrátiť jeho výsledok ako pole reťazcov. Táto metóda je znázornená vo výpise 4.10.

Ako sa ukázalo, Windows PowerShell v predvolenom nastavení pre český jazyk nepoužíva kódovanie UTF-8, čo zapríčinilo nesprávnu interpretáciu znakov s diakritikou. Bolo nutné v nastaveniach systému v sekcii región túto voľbu ručne zapnúť, nakoľko je stále vo fáze vývoja.

```
private string[] InvokePowershellScript(string gitCmd) {
    string[] results;
    using (PowerShell powershell = PowerShell.Create()) {
        powershell.AddScript($"cd {projectDirectory}");
        powershell.AddScript(gitCmd);

        results = powershell.Invoke().Select(r => r.ToString()).ToArray();
    }

    return results;
}
```

Výpis 4.10: Metóda na spustenie skriptu v programe Powershell

Po vyskúšaní rôznych preddefinovaných variant príkazu *git log* som sa rozhodol zostaviť si vlastný príkaz, nakoľko mi žiadna preddefinovaná varianta úplne nevyhovovala. Hlavným dôvodom bola snaha čo najviac zjednodušiť následnú syntaktickú analýzu. Tento príkaz som sa rozhodol vytvárať dynamicky, aby sa sám aktualizoval po zmene premennej *mainSplitter*. Ukážka tohto postupu je zobrazená vo výpise 4.11.

```
private const char mainSplitter = '&';
private string gitLog = "git log ";

public GitLogger() {
    gitLog += string.Format("--pretty=format:\"%an%{0}%ad%{0}%s\" " +
        " --date=format:'%d.%m.%Y'", ((int)mainSplitter).ToHex());
}
```

Výpis 4.11: Dynamické vytváranie vlastného príkazu git log

Následne bolo nutné v cykle prejsť všetky výsledky od najnovšieho *commitu* až po *commit*, ktorý značil posledné zostavenie hry interne dostupné na platforme Steam. Z týchto výsledkov sa odstránili všetky zlúčenia jednotlivých vetiev a ďalšie *commity*, ktoré neobsahovali ani opravy

chýb ani novo pridané vylepšenia. Opravy boli značené tzv. „bugMarkerom“ (@b) a vylepšenia tzv. „featureMarkerom“ (@f). Tých mohlo byť v jednom *commite* hneď niekoľko, preto bol na každý *commit*, ktorý nejakú z týchto značiek obsahoval, aplikovaný regulárny výraz a jednotlivé časti boli ďalej spracovávané samostatne. Tento regulárny výraz zobrazuje výpis 4.12. Každá časť navyše mohla byť rozdelená na viacero menších častí rovnakého druhu pomocou znaku ‘;’. Jednotlivé elementárne časti boli následne štruktúrované uložené do súboru.

```
var matches = Regex.Matches(body, string.Format("{0}|{1})[~@]*", bugMarker,
    featureMarker));
```

Výpis 4.12: Regulárny výraz na rozdelenie tela *commitu*

Pôvodný prípad použitia mal zahŕňať automatické otvorenie tohto súboru v predvolenom textovom editore a jeho ručné skopírovanie do programu Microsoft Teams, ktorý sa používa na komunikáciu vo firme. Rozhodol som sa ale preskúmať možnosti automatizovaného odosielania správ v tejto službe a pokúsil sa implementovať lepšie riešenie.

Služba Microsoft Teams podporuje automatizované posielanie správ okrem iného aj pomocou tzv. „Webhooks“. Tie sa dajú nakonfigurovať priamo v užívateľskom rozhraní služby a to buď pre konkrétny kanál alebo konverzáciu. „Webhook“ vygeneruje jedinečnú URL adresu, na ktorú je možné poslať POST žiadosti, ktoré sú spomenuté aj v sekcii 4.1. Telo takejto žiadosti sa potom v aplikácii Microsoft Teams zobrazí ako správa odoslaná „Webhookom“. Dáta sú prenášané ako štruktúrovaný JSON reťazec. Okrem jednoduchých správ je možné poslať aj formátovaný text pomocou značkovacieho jazyka Markdown, prípadne tzv. karty.

Pomocou nástroja Postman som vo formáte JSON vytvoril dva návrhy takejto karty a nechal kolegov rozhodnúť o tom, ktorý sa nakoniec použije. Podľa zvoleného návrhu som následne implementoval triedu *Message* tak, aby svojou vnútornou štruktúrou odpovedala štruktúre danej karty a bolo možné ju použiť na serializáciu či deserializáciu.

Základnú kostru karty, ktorá sa počas vykonávania programu nebude meniť, som uložil do súboru. Pri každej novej POST žiadosti je tento súbor prečítaný a jeho obsah deserializovaný na novú inštanciu triedy *Message*. Do tela správy sú vložené dáta získané z *commitov* po syntaktickej analýze a následne je celá správa odoslaná do „Webhooku“ metódou uvedenou vo výpise 4.10. Tento postup demonštruje výpis 4.13.

```
Message message = Message.FromJson(File.ReadAllText(pathToJSON));
Content content = message.Attachments[0].Content;

content.Title = header;
content.Sections[0].ActivityText = PrintListToJsonString(features);
content.Sections[1].ActivityText = PrintListToJsonString(bugs);
```

```
// cmdBegin == "Invoke-RestMethod -Method post -ContentType 'Application/Json;
    charset=UTF-8' -Body ";
string command = cmdBegin + Serialize.ToJson(message) + "' -Uri " + webHook;
string result = InvokePowershellScript(command)[0];
```

Výpis 4.13: Vytvorenie a odoslanie správy do služby Microsoft Teams

Metóda *PrintListToJsonString* použitá vo výpise 4.13 pridá do dát značky jazyka Markdown, čím sa ešte upraví finálny vzhľad. Zároveň pomocou regulárneho výrazu “#d+“ nahradí všetky výskyty číselných reťazcov začínajúcich znakom ‘#’ za klikateľný odkaz na stránku konkrétneho reportu v službe Mantis Bug Tracker. To je užitočné v prípadoch keď *commit* priamo opravuje nejakú nahlásenú chybu. Vytvorená karta je znázornená na obrázku 4.5.



Obr. 4.5: Vytvorená karta v aplikácii Microsoft Teams

Poslednou pridanou funkcionalitou bolo, aby sa táto karta neposielala pri každom zostavení hry na serveri ale iba vtedy, keď je to skutočne žiadúce. Aplikácia teda prečíta obsah súboru *BuildSettings.asset* a ak v ňom nájde reťazec “sendPatchNotesToTeams: 1“ vykoná odoslanie. V opačnom prípade sú dáta len štruktúrovane zapísané na disk. Súbor *BuildSettings.asset* je podrobne popísaný v sekcii 4.1.3.

Aplikácia bola nasadená ako externý súbor, ktorý je spustený službou Gitlab CI/CD bližšie popísanou v sekcii 3.4.

4.1.3 Automatizované zostavenie hry na serveri (Build Server)

Časová náročnosť:

Nasadenie: 6 dní, následné úpravy podľa požiadaviek: 2 dni.

Úvod do problému:

Zostavenie projektu *Hobo: Tough Life* môže v závislosti od rôznych faktorov trvať aj viac ako hodinu

čistého času. Počas tejto doby samozrejme nie je možné do projektu zasahovať, čo do značnej miery obmedzuje možnosť ďalej vyvíjať. Zostavenie navyše často končí neúspechom a je nutné po vykonaní opráv proces opakovať. Mojou úlohou bolo teda preskúmať rôzne možnosti ako toto zostavenie vykonať automaticky a bez zablokovania pracovnej stanice.

Navrhované riešenia:

Prvou možnosťou, ktorá sa však hneď zavrholo bolo použitie riešenia priamo od firmy Unity Technologies a síce ich ponúkaného „Unity Cloud Buildu“. Táto možnosť je dostupná za určitý poplatok závislý od veľkosti repozitára. Tá je v našom prípade značná, čo by toto riešenie predražilo a zároveň by nahrávanie repozitára mimo lokálnu sieť bolo časovo náročné. Po preskúmaní rôznych riešení použiteľných na lokálnom serveri sa finálny výber zúžil na služby Jenkins, TeamCity a Gitlab CI/CD. Voľba nakoniec padla na Gitlab CI/CD nakoľko na firemnom serveri je využívaná služba Gitlab a očakávala sa teda najlepšia kompatibilita spomedzi ponúkaných možností.

Realizácia:

Kvôli už spomínanému rozsahu hry nie je vhodné testovať automatizované zostavenie priamo na nej, bolo teda nutné vytvoriť testovací projekt. Zostavenie tohto projektu na pracovnej stanici pomocou metódy *BuildPipeline.BuildPlayer* prebehlo úspešne. Táto metóda prijíma niekoľko nastavení ako napríklad cieľová platforma, scény určené na zostavenie či cestu k zostavenému spustiteľnému súboru a následne sama toto zostavenie vykoná. Kľúčom bolo teda túto metódu spolu s nejakou ďalšou logikou zavolať v rámci reakcie na nejakú udalosť, ktorá nastane na serveri. Táto udalosť môže byť napríklad zlúčenie zmien v rôznych vetvách či nahranie najnovších zmien v kóde do určitej vetvy vo vzdialenom repozitári. Druhú spomínanú variantu sme sa rozhodli využiť a teda spustiť zostavenie po vykonaní príkazu *push* do vetvy master, ktorá je hlavnou vetvou vo verzovacom systéme.

Po úspešnom zostavení projektu na pracovnej stanici pomocou skriptu bolo teda ďalším krokom nasadenie na server. K nasadeniu bolo nutné vytvoriť konfiguračný súbor *.gitlab-ci.yml*, v ktorom sú uložené príkazy, ktoré majú byť vykonané ako reakcia na konkrétnu udalosť. Tieto príkazy sú organizované do etáp. Každá etapa sa potom stará o určitú sadu úloh, ktoré spolu súvisia. Príkladom by mohli byť etapy ako otestuj, zostav či nasad. Na prvý pohľad je jasné, čo bude mať daná etapa na starosť.

V našom prípade bola nutná iba etapa „build“, je ale možné, že do budúcnosti sa tento počet bude zvyšovať. Obsah súboru *.gitlab-ci.yml*, ktorý bol použitý v projekte je možné vidieť vo výpise 4.14. V tomto súbore môže byť nakonfigurovaná aj pomerne zložitá logika, nakoniec sa však z rôznych dôvodov osvedčilo túto logiku presunúť do separátneho batch súboru a ten len z *.gitlab-ci.yml* spustiť. Jedným z týchto dôvodov bolo napríklad, že po neúspešnom zostavení hry sa etapa „build“ ukončila úspešne, čo je pomerne zásadný problém, ktorý sa mi nepodarilo vyriešiť. Naopak použitie batch súboru umožnilo jednoducho ukončiť etapu s hodnotou navrátenou enginu Unity. Typicky išlo o kompilačné chyby. Tento postup sa osvedčil aj v prípade chýb prejavovaných počas zostavovania projektu. V tomto prípade bolo možné nepriamo použiť hodnotu *result* zo štruktúry *BuildReport.summary*, ktorú vracia metóda *BuildPipeline.BuildPlayer*.

Príkazy zadané v súbore `.gitlab-ci.yml` vykonáva tzv „Gitlab runner“. Ten je možné stiahnuť z oficiálnych stránok ako spustiteľný súbor, je ale nutné ho zaregistrovať na konkrétny projekt. Registrácia prebieha v niekoľkých jednoduchých krokoch po spustení programu „Gitlab runner“ z príkazového riadku spolu s príkazom *register*. Jeden z krokov vyžaduje od užívateľa tzv. token, ten je možné nájsť v nastaveniach repozitára vo webovej aplikácii Gitlab. Posledným krokom je potom zvolenie tzv. „executora“, pomocou ktorého bude „Gitlab runner“ vykonávať zadané príkazy. V našom prípade bola zvolená možnosť *shell*, čo značí prosté vykonávanie príkazov v príkazovom riadku lokálneho PC.

```
variables:
  GIT_CLONE_PATH: $CI_BUILDS_DIR\
  GIT_CLEAN_FLAGS: none

unity-build:
  stage: build
  only:
    - master
  script:
    - C:\PerunCreative\Projekty\Hobo\Build\runBuild.bat
  tags:
    - unity
```

Výpis 4.14: Obsah súboru `.gitlab-ci.yml`

Registrácia neprebehla úspešne z dôvodu, že server, na ktorom je uložený náš projekt nemá doménu a ani základný SSL certifikát. Ten sa nakoniec podarilo vytvoriť a vlastným podpisom pomocou nástroja OpenSSL. Výsledný certifikát potom musel byť uložený aj na strane servera aj na strane „Gitlab runnera“ aby sa zabezpečila úspešná registrácia.

Ďalším problémom bola prednastavená *clone/fetch* zložka, do ktorej si „Gitlab runner“ sťahuje najnovšie zmeny na projekte pred zahájením zostavovania. Pri veľkosti nášho repozitára, ktorej hodnota presahuje 100GB by bolo zdĺhavé a nehospodárne mať na serveri dve takéto separátne zložky. Problém by nastal aj pri manuálnych zásahoch do repozitára, ktoré by v réžií samotného Gitlabu boli veľmi zdĺhavé, a tak by sa museli vykonávať ručne na dvoch miestach. Gitlab CI/CD neumožňuje zadanie ľubovolnej cesty do premennej *GIT_CLONE_PATH* v súbore `.gitlab-ci.yml`. Tá sa musí vždy odvíjať od nastavenia *CI_BUILD_DIR*. Hodnotu tejto premennej sa napokon po niekoľkých neúspešných pokusoch podarilo nastaviť priamo v konfiguračnom súbore „Gitlab Runnera“.

Ďalším krokom bolo vytvorenie samotného batch súboru, ktorý bude spúšťaný z `.gitlab-ci.yml` a ktorý bude zároveň spúšťať engine za účelom zostavenia projektu. Z toho vyplýva nutnosť mať

vždy k dispozícii cestu k editoru Unity. Ten je ale často aktualizovaný a cesta je teda závislá od jeho aktuálnej verzie. Za týmto účelom bol použitý textový súbor `ProjectVersion.txt`, ktorý obsahuje informácie o aktuálnej verzii projektu. Pokiaľ sa verzia projektu a editoru nebude zhodovať vyústi to do chyby, kvôli neplatnej ceste čo slúži ako impulz k aktualizovaniu editoru na serveri.

Engine Unity ponúka širokú škálu parametrov s ktorými môže byť spustený. Okrem cesty k projektu a cesty k logovaciemu súboru boli použité parametre ako *batchmode*, ktorý spustí program bez užívateľského rozhrania a parameter *quit*, ktorý editor ukončí po vykonaní všetkých naprogramovaných akcií. Nakoniec je ešte nutné uviesť statickú metódu, ktorá má byť zavolaná. Identifikátor tejto metódy musí byť uvedený vrátane všetkých menných priestorov, do ktorých metóda náleží. Návratová hodnota z editoru je následne uložená do premennej *ERRORLEVEL*. Podľa tejto premennej je ukončené vykonávanie batch súboru a zároveň podľa nej `.gitlab-ci.yml` nastaví výsledok etapy. Po úspešnom zostavení je ešte spustený súbor `GitLogger.exe`, ktorého účel bol podrobne vysvetlený v sekcii 4.1.2. Obsah celého batch súboru potom znázorňuje výpis 4.15.

```
@ECHO off
SET project_path="C:\PerunCreative\Projekty\Hobo\HoboTL"
SET path_to_log="C:\PerunCreative\Projekty\Hobo\Build\build.log"

SET /p var=<%project_path%\ProjectSettings\ProjectVersion.txt
SET version=%var:~17%
ECHO Project version is: %version%

C:\Program Files\Unity\Hub\Editor\%version%\Editor\Unity.exe -quit -batchmode -
    logfile %path_to_log% -projectPath %project_path% -executeMethod Dev.Build.
    BuildScript.Build

IF %ERRORLEVEL% GTR 0 (
    ECHO An error with errorlevel %ERRORLEVEL% occurs
    ECHO Check logfile build.txt in project folder for more details
) ELSE (
    ECHO Build succeeded
    START "" "GitLogger.exe"
)

EXIT %errorlevel%
```

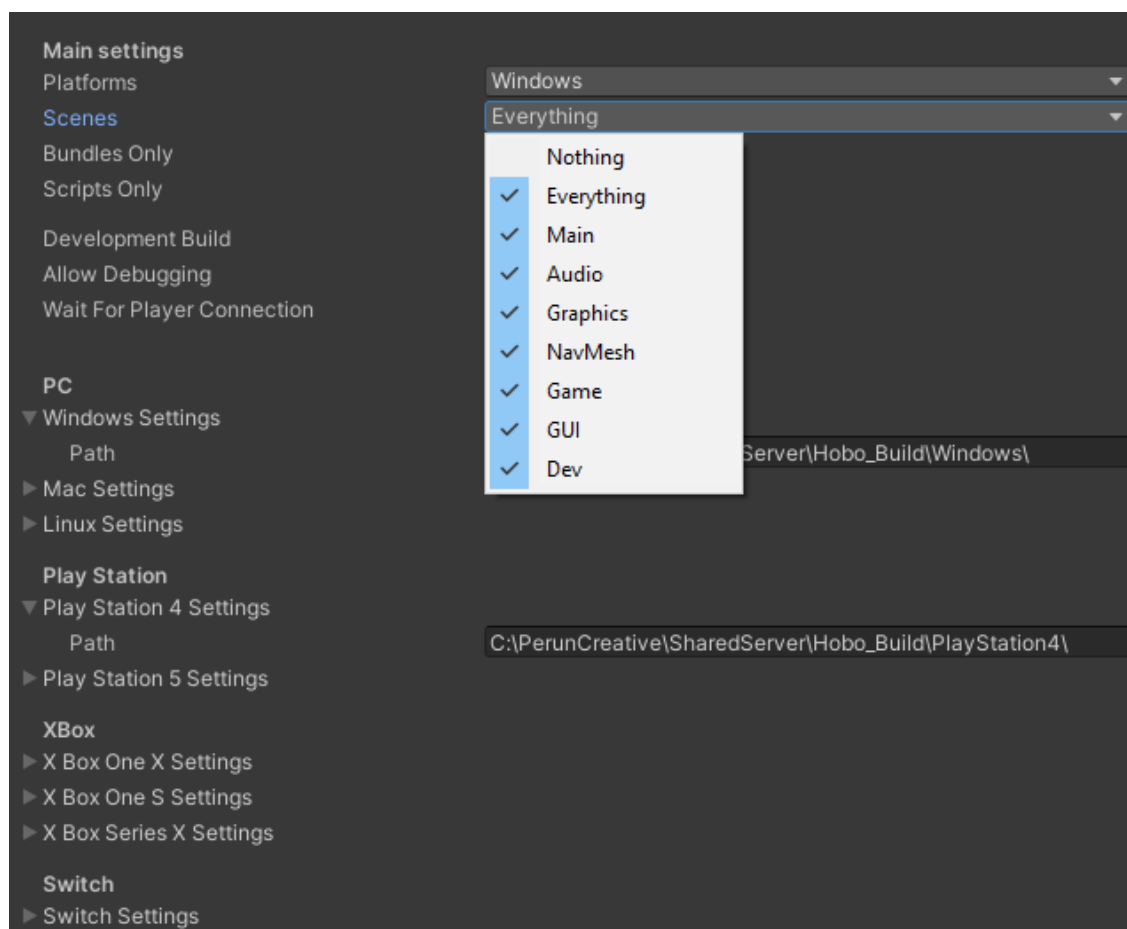
Výpis 4.15: Skript zabezpečujúci zostavenie hry na serveri

Metóda *BuildScript.Build* volá metódu *BuildScript.InternalBuild* s parametrom *closeAfterBuild* nastaveným na hodnotu *true*. Toto riešenie bolo zvolené z dôvodu, aby sa pri manuálnom spustení zostavovania pomocou skriptu, napríklad z horného menu editora Unity, editor neukončil. Úloha metódy *InternalBuild* je potom nájsť a načítať dáta z tzv. „Scriptable Objectu“, čo je uvedené vo výpise 4.16, a následne tieto dáta predať metóde *SetUpPlatforms*. Návrátová hodnota tejto metódy je potom použitá ako návratová hodnota celého programu.

```
buildSettings = AssetDatabase.LoadAssetAtPath<BuildSettings>("Assets/Settings/  
BuildSettings.asset");
```

Výpis 4.16: Načítanie dát zo scriptable objectu

„Scriptable Object“ *BuildSettings.asset* slúži ako dátové úložisko všetkých nastavení týkajúcich sa automatizovaného zostavenia. S týmto objektom je potom možné pohodlne pracovať priamo z užívateľského rozhrania engineu nakoľko som implementoval tzv. „Custom editor“. Ten je znázotnený na obrázku 4.6.



Obr. 4.6: Custom editor pre scriptable object *BuildSettings.asset*

Samotná trieda *BuildSettings* dedí z triedy *ScriptableObject* a je iba súhrnom relatívne veľkého množstva serializovaných premenných a enumov. „Custom editor“ je potom realizovaný ako trieda *BuildSettingsEditor*, ktorá je potomkom triedy *Editor*. Táto trieda sa nachádza v preddefinovanej *Assembly-CSharp-Editor*, preto je nutné potomkov triedy *Editor* presunúť do zložky *Editor* alebo ich kód obaliť do direktívy `UNITY_EDITOR` zobrazenej aj vo výpise 4.3. To zabráni vyvolaniu chyby pri pokuse o zostavenie projektu nakoľko *Assembly-CSharp-Editor* z pochopiteľných dôvodov nie je prítomná v zostavenej verzii hry. Základná kostra triedy *BuildSettings*, ktorá vykreslí iba premennú *platforms* je zobrazená vo výpise 4.17.

```
[CustomEditor(typeof(BuildSettings))]  
public class BuildSettingsEditor : Editor {  
    private BuildSettings settings;  
    private SerializedProperty platforms;  
  
    private void OnEnable() {  
        platforms = serializedObject.FindProperty("platforms");  
    }  
    public override void OnInspectorGUI() {  
        settings = AssetDatabase.LoadAssetAtPath<BuildSettings>("Assets/Settings/  
        BuildSettings.asset");  
  
        EditorGUILayout.PropertyField(platforms);  
        serializedObject.ApplyModifiedProperties();  
    }  
}
```

Výpis 4.17: Kostra triedy určenej pre vykreslenie vlastného okna v Editore Unity

Finálna trieda, použitá v projekte bola okrem pridania ostatných premenných doplnená o nadpisy, medzery medzi skupinami a v neposlednej rade aj o varovné hlásenia pri nastavení polí *Platform* či *Scenes* na hodnotu *Nothing*. Varovné hlásenie sa objaví aj pri nastavení tzv. „scripting backendu“ na hodnotu inú ako `IL2CPP`, čo je dôležité nastavenie z hľadiska výkonu finálneho produktu. Pri tomto nastavení je IL kód pred vytvorením spustiteľného súboru prevedený na kód jazyka C++ [12]. Ukážka takéhoto varovania je uvedená na obrázku 4.7.



Obr. 4.7: Varovanie v custom editore

Metóda *SetUpPlatforms* spomenutá vyššie zo získaných dát od metódy *InternalBuild* extrahuje tzv. masky. Masky platformy určuje, na ktoré platformy sa bude hra zostavovať a maska scény, ktoré scény budú v zostavenom súbore prítomné. Princíp použitia takejto masky je demonštrovaný na získavaní scén zvolených v objekte *BuildSettings.asset* vo výpise 4.18. Tieto scény sú potom predávané ako argument príslušným zostavovacím metódam.

```
private static string[] SetUpScenes() {
    Array enumScenes = Enum.GetValues(typeof(BuildSettings.Scene));
    List<string> scenesToBuild = new List<string>();

    foreach (var item in enumScenes) {
        if (IsSelected(sceneMask, (BuildSettings.Scene)item)) {
            scenesToBuild.Add("Assets/Scenes/" + item + ".unity");
        }
    }

    return scenesToBuild.ToArray();
}

private static bool IsSelected(int EnumMask, Enum enumValue) {
    int tempMask = 1 << Convert.ToInt32(enumValue);

    return (EnumMask & tempMask) == 0 ? false : true;
}
```

Výpis 4.18: Extrahovanie zvolených scén pomocou masky

Obdobne prebieha aj práca s maskou platformy. Ak metóda *IsSelected* vráti hodnotu *true* pri dotazovaní na konkrétnu platformu, je spustená príslušná zostavovacia metóda. Tá vykoná samotné zostavenie pomocou metódy *BuildPipeline.BuildPlayer* s nastaveniami špecifickými pre danú platformu. Návratová hodnota sa potom vráti späť metóde *SetUpPlatforms* a pokiaľ neznačí úspech je vyhodnená výnimka s popisom, že zostavenie na danej platforme zlyhalo. Ak všetky zvolené zostavenia skončia úspechom, je úspešne ukončený aj celý program. Gro metódy na zostavenie hry pre OS Windows je znázornený vo výpise 4.19.

Ako je možné vidieť, metóda *BuildWindows* dostane ako prvý argument nastavenia špecifické pre danú platformu, v tomto prípade je to platforma Windows. Tieto nastavenia momentálne obsahujú iba cestu k zložke do ktorej má byť zostavenie pre danú platformu uložené. Toto riešenie bolo dočasne zvolené z dôvodu, že nie je možné predikovať špecifiká, ktoré si zostavenie na niektoré konzolové platformy bude vyžadovať. Keď budú tieto informácie dostupné plánuje sa zjednotenie všetkých

zostavovacích metód do jednej. Táto metóda potom bude ako prvý argument prijímať generalizáciu nejakých konkrétnych nastavení a bude riadená dátami získanými z týchto nastavení.

```
private static bool BuildWindows(WindowsSettings settings, string[] scenes) {  
  
    // ...  
  
    BuildPlayerOptions options = new BuildPlayerOptions();  
    options.scenes = scenes;  
    options.locationPathName = path + Application.productName + ".exe";  
    options.targetGroup = BuildTargetGroup.Standalone;  
    options.target = BuildTarget.StandaloneWindows64;  
    options.options = GetBuildOptions(); // Development, AllowDebugging, ...  
  
    BuildReport report = BuildPipeline.BuildPlayer(options);  
  
    return report.summary.result == BuildResult.Succeeded ? true : false;  
}
```

Výpis 4.19: Extrahovanie zvolených scén pomocou masky

Postupne boli do projektu pridané aj ďalšie možnosti zostavenia. Možnosť „scriptsOnly“ značí, že budú nanovo prekompilované všetky skripty bez nutnosti zostaviť celý projekt nanovo. Táto možnosť ale nie je k dispozícii pre „scripting backend“ IL2CPP, takže je určená len na vývojárske využitie, kde je nutné robiť veľa iteratívnych zmien v krátkom čase a hneď ich aj testovať. Druhou implementovanou možnosťou bolo v už zostavenom projekte aktualizovať len tzv. „bundles“. „Bundles“ sú binárne súbory, ktoré môžu obsahovať napríklad nové textúry, modely či konverzácie a načítavajú sa dynamicky počas vykonávania programu [13]. Tieto možnosti teda pridávajú ďalšiu vrstvu optimalizácie tým, že kompletne zostavenie nastane až keď je to nevyhnutné.

4.2 Optimalizácia a prevod hry na ďalšie platformy

4.2.1 Optimalizácia a refaktorovanie kódu

Časová náročnosť:

4 dni.

Úvod do problému:

S rastúcim výkonom výpočtovej techniky sa neustále znižuje dôraz na optimalizáciu a efektivitu. Herný vývoj je ale jedna z disciplín, v ktorých hrá optimalizácia veľmi dôležitú úlohu aj dnes.

Rovnako ako narastá výkon tak narastajú aj požiadavky hráčov na vyššiu fotorealisticnosť grafiky či viac vykreslených snímok za sekundu. S tým ide ruku v ruku aj vývoj v ďalších oblastiach hardvéru ako sú napríklad monitory. Najmodernejšie sériovo vyrábané svojím rozlíšením presahujú rozlíšenie 4k a obnovovacou frekvenciou 240Hz. Na dosiahnutie hodnoty 60 snímok za sekundu je potom nutné vykonať všetky potrebné algoritmy v rámci 16 milisekúnd, čo nie je vždy jednoduché. Mojou úlohou bolo teda identifikovať potenciálne úzke hrdlá a pokiaľ možno ich aj optimalizovať

Navrhované riešenia:

Ako som už spomínal v sekcii 4.1 hra Hobo: Tough Life je postavená na hierarchickej štruktúre manažérov. Hlavný manažér obsahuje metódu *Update* a v nej sú postupne volané metódy *OnUpdate* všetkých ostatných manažérov. Z pohľadu výkonu je toto najkritickejšie miesto nakoľko metóda *Update* je vykonávaná počas vykreslenia každej jednej snímky. Prvou vecou, na ktorú bolo nutné sa zamerať bola práve táto hierarchická štruktúra. Druhým návrhom bolo použiť nástroj Unity Profiler, ktorého účelom je zaznamenávať rôzne údaje o výkone hry do jednotlivých grafov. Takto je napríklad možné zobraziť aj presnú metódu, ktorá bola volaná v momente, keď nastal pád snímok. Treťou a poslednou vecou, na ktorú som sa rozhodol zamerať bola celková mikrooptimalizácia založená na vyhľadávaní a oprave drobných neoptimalizovaných častí kódu pochádzajúcich prevažne z ranného štádia vývoja.

Realizácia:

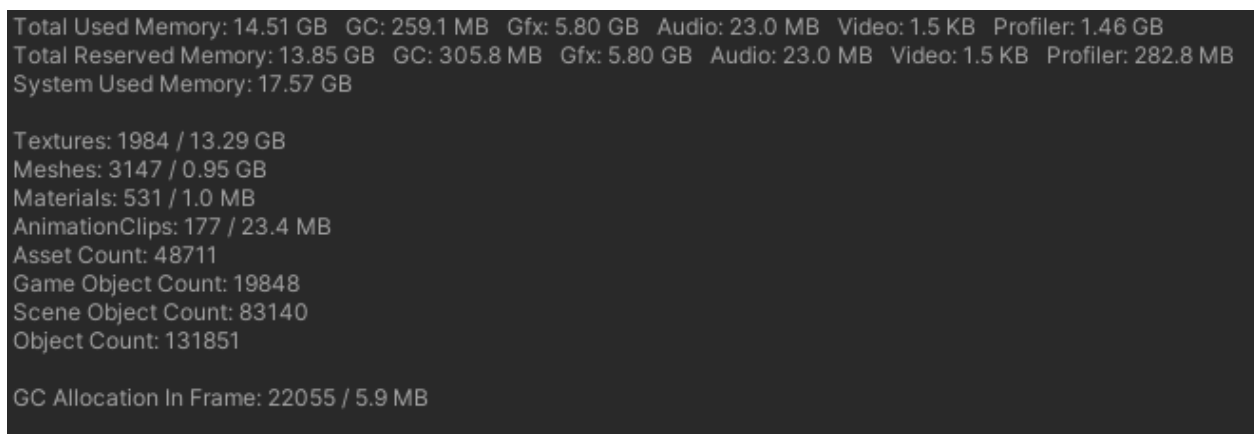
Prvým krokom bolo teda preskúmanie štruktúry metód volaných pri vykresľovaní každej snímky. Veľký prepád snímok na konkrétnom mieste, napríklad pri komunikácii s nejakým NPC je problém, pokiaľ však takáto situácia nastáva len veľmi zriedka, je výhodnejšie snažiť sa skôr o optimalizáciou často volaných metód. Tým sa zdvihne priemerná hodnota snímok za sekundu a už rozdiel v ráde nižších jednotiek býva v niektorých kritických bodoch hry signifikantný. V často volaných metódach ako napríklad už spomenutá metóda *Update* by sa prakticky nemalo vyskytovať vytváranie nových premenných referenčného typu. Tieto premenné odkazujú na pamäťový priestor vytvorený na halde a spravidla zaberajú rádovo viac pamäte ako bežné, napríklad celočíselné premenné. Táto pamäť alokovaná na halde musí byť následne uvoľnená. Narozdiel od jazykov ako C++, v jazyku C# sa o toto uvoľňovanie pamäte stará tzv „Garbage Collector“ alebo GC. Ten je volaný v pravidelných intervaloch, ale je možné ho zavolať aj predčasne, napríklad v momente načítavania novej lokácie, keď prudký pád snímok za sekundu nebude hrať žiadnu rolu. Z toho teda vyplýva, že čím menej pamäte bude nutné týmto spôsobom uvoľňovať, tým menší dopad na výkon a prepady snímok bude GC mať. Alokácia novej pamäte by teda mala nastávať pri inicializácii a v niektorých ďalších situáciách ale rozhodne nie pri vykresľovaní každej snímky. Aby sa ešte zmenšila nutnosť alokovania novej pamäte, je v projekte použitý aj tzv „Object Pool“. Ten slúži na recykláciu určitých objektov, aby ich nebolo nutné znova vytvárať.

Ďalšou vecou, ktorá by sa mala nachádzať v takýchto metódach čo najmenej sú pochopiteľne cykly. Nie vždy sa im dá vyhnúť, vždy je však vhodné zvážiť použitie iného spôsobu, napríklad korutiny. Týmto spôsobom by sa počet vykonávaní nejakého cyklu mohol zredukovať napríklad na

štvrtinu, čo by vytvorilo možnosť niektoré akcie striedať medzi sebou.

Po podrobnom preskúmaní týchto metód, som nenašiel žiadne závažné úzke hrdlo. Toto zistenie nebolo až tak prekvapivé nakoľko hra prešla počas svojho vývoja niekoľkými optimalizačnými cyklami s dôrazom práve na tieto miesta. Bol som dopredu aj upozornený, že problém optimalizácie je do veľkej miery hľadanie tzv. „ihly v kope sena“.

Druhou optimalizačnou stratégiou bolo použitie nástroja Unity Profiler. Tento nástroj pracuje v dvoch režimoch - štandardnom a hlbokom. Štandardný režim menej ovplyvňuje finálny výkon hry, v praxi sa mi ale veľmi neosvedčil. Umožňoval síce základnú detekciu problémov, od určitej hĺbky zanorenia ale nie je možné dohľadať konkrétne volané metódy. Užitočnosť tohto režimu sa ale ukázali pri hľadaní alokovanej pamäte v danom snímku. Hodnoty namerané v potenciálne problematickom mieste a síce pri otváraaní inventára znázorňuje obázok 4.8.



The image is a screenshot of the Unity Profiler interface, displaying various memory and asset statistics. The data is organized into two main sections: a top summary section and a detailed breakdown section.

Category	Value
Total Used Memory	14.51 GB
GC	259.1 MB
Gfx	5.80 GB
Audio	23.0 MB
Video	1.5 KB
Profiler	1.46 GB
Total Reserved Memory	13.85 GB
GC	305.8 MB
Gfx	5.80 GB
Audio	23.0 MB
Video	1.5 KB
Profiler	282.8 MB
System Used Memory	17.57 GB

Category	Value
Textures	1984 / 13.29 GB
Meshes	3147 / 0.95 GB
Materials	531 / 1.0 MB
AnimationClips	177 / 23.4 MB
Asset Count	48711
Game Object Count	19848
Scene Object Count	83140
Object Count	131851

Category	Value
GC Allocation In Frame	22055 / 5.9 MB

Obr. 4.8: Namerané hodnoty z nástroja Unity Profiler

Po otvorení inventára dôjde k vytvoreniu veľkého množstva objektov, ktoré sa týkajú užívateľského rozhrania. Najväčšiu časť z alokovaných objektov tvorili užívateľom získané predmety. Po porade s kolegami sme dospeli k záveru, že by to bolo možné optimalizovať len nejakou formou stránkovania, čo by ale znamenalo nutnosť prepracovať celý doterajší systém, a to sa ukázalo z časového hľadiska nerealizovateľné. Čo sa ostatných prvkov užívateľského rozhrania inventára týka, žiadny svojou náročnosťou výrazne neprevyšoval ostatné takže sa upustilo od ďalšieho skúmania. Veľká alokácia pamäte bola ešte samozrejme prítomná pri štarte hry. To sa nám podarilo z veľkej miery optimalizovať.

Následné využívanie hlbokého režimu profileru ukazovalo niekoľko potenciálnych problematických miest, všetky sa ale ukázali ako slepé uličky. Niektoré boli spôsobené spustením hry v rámci editoru a v samostatnej zostavenej verzii ich nebolo možné zreprodukovat, iné boli nevyhnutné pre samotný chod hry, prípadne zabezpečovali vykresľovanie grafiky samotnej.

Následné mikrooptimalizácie boli zamerané tzv „cachovanie premenných“, prevod *foreach* cyklov na klasické *for* cykly, prednastavenie kapacity rôznych dátových štruktúr pri ich vytváraní, pou-

žívanie lokálneho súradnicového systému na úkor globálneho pokiaľ to bolo možné či zabráneniu zbytočnej serializácií premenných, ktoré nutne serializované byť nemuseli.

„Cachovanie premenných“ je jednoduchá technika pri ktorej sa ukladajú získané dáta do nejakej dočasnej premennej aby sa s nimi mohlo ďalej pracovať bez nutnosti neustále sa na ne dotazovať znova. Efektivita takéhoto prístupu sa naviac ukazuje v spojení s *for* cyklom. Namiesto toho, aby sa kód každú iteráciu cyklu vždy dotazoval na dĺžku nejakej dátovej štruktúry je táto dĺžka uložená do premennej a vždy len prečítaná.

Foreach cyklus bol v priebehu vývoja postupne nahrádzaný *for* cyklom. Dôvodom bolo, že cyklus *for* vykazuje prakticky vždy lepšie výsledky z hľadiska výkonu. Vo väčšine kľúčových miest bol nahradený už v dobe keď som začínal s projektom pracovať, niekoľko desiatok kde ho bolo potrebné nahradiť sa ale našlo. Išlo o prípady, kde sa tento cyklus používa v rámci čistej hry pre jedného hráča. Zasahovať do cyklov v rámci vývojárskych nástrojov by z hľadiska výkonu nijak nepomohlo a nezasahovalo sa z tohto pohľadu ani do zložky hry pre viac hráčov.

Prednastavenie kapacity dátových štruktúr je taktiež veľmi užitočná optimalizácia. Obyčajné pole, ktoré je na projekte preferované, pokiaľ je ho možné použiť, sa vytvára s kapacitou takou, aby dokázalo bezpečne uložiť všetky dáta. Dátové štruktúry ako *List* či *Dictionary* majú naproti tomu výhodu, že môžu svoju kapacitu postupne zväčšovať. Časté zväčšovanie je ale veľmi náročné z hľadiska výpočtového výkonu nakoľko je nutné po presiahnutí pôvodnej kapacity vytvoriť na pozadí novú dátovú štruktúru s väčšou kapacitou a jednotlivé prvky do nej skopírovať. Pokiaľ teda programátor má aspoň približnú predstavu o počte budúcich uložených prvkov je vhodné tomu priblížiť aj prednastavenú kapacitu. V niektorých prípadoch som skutočne našiel takéto prípady a následne ich opravil.

Posledné dve vykonávané mikrooptimalizácie úzko súvisia s enginom Unity. Prvá z nich je uprednostnenie lokálneho súradnicového systému oproti globálnemu. To je z dôvodu, že lokálny súradnicový systém je relatívny k rodičovskému objektu a globálne koordináty sa musia vždy vyrátať zo všetkých rodičovských objektov až ku koreňovému objektu, čo značí omnoho vyššiu náročnosť na výpočtový výkon. Čo sa serializácie objektov týka, Unity engine automaticky serializuje všetky verejné premenné objektu. Pokiaľ nie je nutné tieto premenné serializovať nemali by byť označené ako verejné. Tento prístup je zlý aj z hľadiska zapúzdrenia. V projekte sa od ranného vývoja vyskytovali niektoré čisto verejné triedy, mojou úlohou bolo teda skontrolovať, čo je potrebné serializovať a čo nie. Následne bolo nutné upraviť modifikátor prístupu podľa potreby a prakticky všetky verejné premenné nahradiť za tzv „Properties“. Pri tomto postupe bolo nutné dbať na serializované dáta. Niektoré z týchto dát boli nastavované a následne ladené podľa potreby priamo v editore a strata týchto dát by predstavovala veľký problém. Dáta sa aj automaticky zmažu po premenovaní danej premennej a následnej kompilácii.

Po jednom nechcenom premenovaní bolo nutné ručne obnovovať niektoré dôležité dáta ako napríklad identifikátory objektov, našťastie sa to stalo iba pri jednej dátovej štruktúre, ktorá nebola veľmi obsiahla. Problémy nastali aj pri niektorých nevhodne „zacachovaných“ premenných, prípadne

pri prevode cyklov. Po ich následnej úprave a prekontrolovaní všetkých zmien bola optimalizácia a refaktorovanie ukončené z dôvodu nenájdenia ďalších veľkých problémov a nutnosti venovať sa ďalším projektom. Predtým ale došlo ešte k aplikácií automatizovaných nástrojov na formátovanie kódu či vymazanie nevyužívaných *using* príkazov.

4.2.2 Prevod hry na OS Linux

Časová náročnosť:

Nasadenie: 3 dni.

Úvod do problému:

Ako som spomenul v sekcií

Navrhované riešenia:

Nakoľko bola

Realizácia:

Pri preskúmvaní mož

4.2.3 Implementácia Epic Online Services

Časová náročnosť:

Nasadenie: 3 dni.

Úvod do problému:

Ako som spomenul v sekcií

Navrhované riešenia:

Nakoľko bola

Realizácia:

Pri preskúmvaní mož

4.3 Ostatné projekty

4.3.1 Systém líhania hráčskej postavy

Časová náročnosť:

Nasadenie: 3 dni.

Úvod do problému:

Ako som spomenul v sekcií

Navrhované riešenia:

Nakoľko bola

Realizácia:

Pri preskúmvaní mož

Literatúra

1. VISIONGAME. *Perun Creative* [online] [cit. 2021-03-06]. Dostupné z : <https://visiongame.cz/studio/perun-creative/>.
2. PERUN CREATIVE. *Hobo: Tough Life* [online] [cit. 2021-03-06]. Dostupné z : <http://hoborpg.com/>.
3. MICROSOFT. *A tour of the C# language* [online] [cit. 2021-03-07]. Dostupné z : <https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/>.
4. UNITY TECHNOLOGIES. *Unity Platform* [online] [cit. 2021-03-06]. Dostupné z : <https://unity.com/products/unity-platform>.
5. UNITY TECHNOLOGIES. *Build once, deploy anywhere* [online] [cit. 2021-03-06]. Dostupné z : <https://unity.com/features/multiplatform>.
6. CHACON, Scott; STRAUB, Ben. *Pro Git*. 2nd ed. New York: Apress, 2014. ISBN 1484200772.
7. GITLAB. *CI/CD concepts* [online] [cit. 2021-03-10]. Dostupné z : <https://docs.gitlab.com/ee/ci/introduction/index.html>.
8. EPIC GAMES. *Epic Online Services* [online] [cit. 2021-03-11]. Dostupné z : <https://dev.epicgames.com/en-US/services>.
9. MOSS, Richard. *MantisSharp* [online]. 2017 [cit. 2021-03-12]. Dostupné z : <https://github.com/cyotek/MantisSharp>.
10. FELDT, Henrik. *DotNetZip* [online] [cit. 2021-03-13]. Dostupné z : <https://github.com/haf/DotNetZip.Semverd/>.
11. CLOUDFLARE. *What is MSS (maximum segment size)?* [Online] [cit. 2021-03-20]. Dostupné z : <https://www.cloudflare.com/learning/network-layer/what-is-mss/>.
12. UNITY TECHNOLOGIES. *IL2CPP* [online] [cit. 2021-03-21]. Dostupné z : <https://docs.unity3d.com/Manual/IL2CPP.html>.
13. UNITY TECHNOLOGIES. *AssetBundles* [online] [cit. 2021-03-21]. Dostupné z : <https://docs.unity3d.com/Manual/AssetBundlesIntro.html>.