VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Absolvování individuální odborné praxe

Individual Professional Practice in the Company

2015

Pavel Drábek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Pavel Drábek**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Absolvování individuální odborné praxe
Individual Professional Practice in the Company

Zásady pro vypracování:

1. Student vykoná individuální praxi ve firmě: Craneballs s.r.o.
2. Struktura závěrečné zprávy:
   a) Popis odborného zaměření firmy, u které student vykonal odbornou praxi a popis pracovního zařazení studenta.
   b) Seznam úkolů zadaných studentovi v průběhu odborné praxe s vyjádřením jejich časové náročnosti.
   c) Zvolený postup řešení zadaných úkolů.
   d) Teoretické a praktické znalosti a dovednosti získané v průběhu studia uplatněné studentem v průběhu odborné praxe.
   e) Znalosti či dovednosti scházející studentovi v průběhu odborné praxe.
   f) Dosažené výsledky v průběhu odborné praxe a její celkové zhodnocení.

Seznam doporučené odborné literatury:

Podle pokynů konzultanta, který vede odbornou praxi studenta.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **doc. Mgr. Jiří Dvorský, Ph.D.**

Konzultant bakalářské práce: Ing. Matěj Rejnoch

Datum zadání: 01.09.2014

Datum odevzdání: 07.05.2015

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.*

V Ostravě 24. června 2015 .....

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 24. června 2015 .....

Na tomto místě bych chtěl poděkovat vedení firmy za umožnění vykonání bakalářské práce formou odborné praxe a zejména svému konzultantovi, panu Ing. Matěji Rejnochovi. Dále bych chtěl poděkovat doc. Mgr. Jiřímu Dvorskému, Ph.D za vedení bakalářské práce a všem, kteří mi pomohli k vypracování a dokončení této práce.

Abstrakt

Práce se zabývá popisem individuální odborné praxe, kterou jsem vykonával ve firmě Craneballs s.r.o. [1] Konrétně vývojem multiplayerové online hry. Po krátkém úvodu je čtenář seznámen s firmou, ve které byla praxe vykonávána. Dále je ve zkratce popsán game design a využívané technologie při vývoji. Následuje jednotlivé zadání úkolů včetně postupu řešení a hodnocení časové náročnosti. Závěrem práce je shrnuto, kterých znalostí získaných během studia jsem využil, které mi naopak chyběly a jaké jsem během praxe získal. V poslední řadě celkové zhodnocení praxe.

Klíčová slova: Hra, Unity, střílečka, online, klient, server, optimalizace, Craneballs, multiplatformnost

Abstract

This article details author's work experience developing a multiplayer online game in a company Craneballs s.r.o. [1] The company's history and structure is introduced in the beginning, followed by a short introduction of the specific game design and the technologies used for development and backend. The work then covers specific tasks needed to be solved in more detail including the applied solutions and evaluation of time requirements. The conclusion summarizes which knowledge bits gained during author's studies were utilized, which were missing and which were gained. The professional experience is evaluated in the end.

Keywords: Game, Unity, shooter, online, client, server, optimization, Craneballs, multiplatform

Seznam použitých zkratek a symbolů

2D	–	2 dimensions (dvou rozměrný)
3D	–	3 dimensions (troj rozměrný)
MP	–	multiplayer, hra více hráčů
SW	–	software (program; sada programů)
OS	–	operační systém
FPS	–	frames per second (počet snímků za vteřinu)
FPS	–	first person shooter (střílečka z první osoby; vidíme to co vidí naše ovládaná postava)
3PS	–	third person shooter (střílečka ze třetí osoby; ovládanou postavu vidíme celou zezadu)
ID	–	identifikační číslo
XML	–	Extensible Markup Language (rozšiřitelný značkovací jazyk); formát souboru

Obsah

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Cílem této bakalářské práce je popis mého působení ve firmě Craneballs s.r.o. (dále jen Craneballs), která se zabývá tvorbou her pro mobilní platformy. Ve firmě již 2 roky pracuju jako programátor a proto jsem si vybral vypracování bakalářské práce formou praxe. Zaměřím se pouze na úkoly zadané během 5. a 6. semestru studia. Během své praxe jsem pracoval na multiplayerové hře se zaměřením na mobilní platformy iOS a Android. Později během vývoje přibyly platformy Windows a OSX, avšak o těch psát nebudu.

Na začátku práce popíšu, čím se firma Craneballs zabývá, popíšu základní game design hry, která je náplní mé práce. Dále se zaměřím na konkrétní úkoly, které jsem během praxe vykonával. Nastíním jejich problematiku a vysvětlím mnou navržené a implementované řešení.

Většina úkolů znamenala práci jak na serverové, tak na klientské části zároveň. Jednotlivé úkoly nejsou seřazeny podle data zadání, jelikož se občas úkoly časově překrývaly, jindy jsem dělal na úkolech nesouvisejicíh se zadáním mé práce nebo jsem pořadí zaměnil z důvodu zachování souvislostí textu.

2 Popis odborného zaměření firmy a pracovního zařazení

2.1 Odborné zaměření

Poměrně mladá firma Craneballs vyvíjí hry na mobilní platformy od roku 2008. Nejdříve se zaměřovala na v té době novou platformu iOS od společnosti Apple Inc. V roce 2009 přišla se svou první hrou Blimp: The Flying Adventures. Stejného roku studio vydalo stealth akci z druhé světové války - 33rd Division, kterou brzy stáhli a znova vydali roku 2013. Roku 2010 vychází hry Monorace a SuperRope.

Největšího rozmachu dosáhla v roce 2011, kdy se se svou arkádovou 2D střílečkou z první osoby Overkill dostali na první místo v americkém App Store žebříčku a zaznamenali přes 10 milionů stažení. Od této doby se Craneballs začíná věnovat i mobilní plaformě Android. Firma se této značce věnuje dodnes. Pokračovním Overkill 2 svůj úspěch zopakovali a v době absolvování praxe vyšlo třetí pokračování v enginu Unity 3D [2].

2.2 Popis pracovního zařazení

Pro bakalářskou praxi jsem si vybral firmu Craneballs s.r.o., pro kterou pracuji již dva roky jako programátor. Pracoval jsem na prototypu zatím nezveřejněné nové hry, kterou si chtěla firma rozšířit obzory a pole působnosti. Byl jsem součástí malého týmu o 4 lidech. Nejdříve na prototypu pracovali 2 programátoři, později se tým rozšířil o jednoho 2D grafika a jednoho 3D grafika.

Firma pro řízení projektů používá agilní metodou Scrum, která říká, že se tým schází pravidelně za určitou dobu (sprint), kde vyhodnocuje dokončené úkoly. V případě nedostatečného nebo chybného vyhotovení se úkol přehodnotí a přesune se do dalšího sprintu. Na tomto setkání týmu se také rozdělují nové úkoly. Pro náš tým trval jeden sprint 1 až 3 týdny v závislosti na fázi vývoje.

3 Seznam úkolů zadaných studentovi v průběhu odborné praxe s vyjádřením jejich časové náročnosti

3.1 Návrh a implementace logiky serveru pro FPS (First Person Shooter)

Najít možné serverové řešení. Server musí běžet jako samostatná jednotka na fyzickém serveru poskytovaného třetí stranou. Navrhnout a implementovat logiku komunikace mezi serverem a klienty v reálném čase. Akci (střelba, pohyb, smrt, zvednutí power upu), kterou provede kterýkoliv klient, se musí ihned zaregistrovat na všech dalších klientech. Je potřeba vzít v potaz odezvu serveru i klientů a nenarušit spravedlivost hry.

3.2 Parsování položek do hry z XML

Zobrazit v menu hry předměty, kterými si hráč bude vylepšovat svou postavu. Vymyslet způsob, jakým se budou předměty editovat a přidávat.

3.3 Skóre tabulka

Zobrazovat ve hře tabulku s aktuálními výsledky. Tabulka bude obsahovat přezdívku hráče, počet zabití, asistencí, úmrtí a celkové skóre. Výsledná tabulka bude vypadat stejně jako předloha dodaná grafickým oddělením.

3.4 Optimalizace

Optimalizovat kód a komunikaci mezi serverem a klientem.

3.5 Audio pulser

Rozpohybovat reproduktory nacházející se na mapě do rytmu hudby.

3.6 Refaktorizace z Unity 4.6 do Unity 5

Přepsat projekt do nově vyšlé verze Unity 5 včetně uživatelského rozhraní z NGUI do UGUI se zachováním funkčnosti.

3.7 Rozpohybování kamery za běhu hrdiny

Rozpohybovat kameru, která sleduje hráče, aby vznikl lepší a dynamičtější pocit ze hry.

3.8 Shader průhlednosti překážek

Vyřešit problém, kdy objekt na mapě zakryje kameru.

4 Využité technologie

4.1 Klientská část

Pro klienta jsme využili engine Unity 3D [2] ve verzi 4.6.3 (dále jen Unity). Díky grafickému rozhraní a implementované logice pro tvorbu her jsme se nemuseli zdržovat vývojem vlastního enginu. Unity také do jisté míry zajišťuje vývoj jednoho klienta pro více platforem. Oproti konkurenci (CryEngine a Unreal Engine) je Unity mladší a méně zdokumentovaný SW. Má ovšem obrovskou komunitu a v době výběru vývojového prostředí byla levnější i pořizovací cena. Pro vývoj jsme v týmu používali jazyk C# a verzování kódu probíhalo přes gitu (Github [3] a Sourcetree [4]).

4.2 Serverová část

Server musí běžet jako samostatná jednotka na fyzickém serveru poskytovaného třetí stranou. Pro vývoj však používal každý programátor vlastní server běžící na jeho počítači. Komunikaci s klientem a základní logikou multiplayeru nám zajišťovala knihovna Photon Server SDK [5] (dále jen Photon). Tento software byl vytvořen právě pro online hry, zajišťuje load balancing a nabízí i poskytnutí vlastních serverů v rozumné cenové kategorii. V době mé praxe mohl Photon server běžet pouze na platformě Windows. Programovací jazyk a verzování jsou stejné jako u klientské části, C# a git.

5 Základní game design hry

Jedná se o střílečku z pohledu třetí osoby pro více hráčů přes internet. Hra se ovládá na platformách iOS a Android pomocí dotykového displeje. Platformy OS X a Windows využívají klávesnici a myš. Aplikace je rozdělena do dvou částí - menu a samotná hra (gameplay). V menu mají hráči možnost výběru ze 3 postav. Každou postavu či její vybavení mohou do jisté míry vylepšovat a tím zvýšit své šance na výhru. Při vstupu do samotné hry se klient automaticky připojí do už probíhající hry nebo mu bude automaticky vytvořena hra nová. Která situace nastane nemůže hráč ovlivnit.

Ve hře se hráč snaží spolu se svým týmem prostřílet k nepřátelské základně, kterou následně musí zničit. V momentě kdy jeden z týmů nemá základnu, prohrává a hra končí. Aby hráč své šance na úspěch zvýšil, může sbírat vylepšení (power up) v podobě beden vyskytujících se různě na mapě. Tato vylepšení se skládají na sebe a hráč je může kdykoliv během hry aktivovat. Mezi power upy patří např. Thunderstorm (mrak plující nad hráčem zraňující nepřátelé v určité vzdálenosti od hráče), Turret (automatické zařízení střílející po hráčích na dohled), Missile Rocket (automatická raketa sledující označeného hráče) apod.

Obrázek 1: Screenshot ze hry během vývoje

6 Zvolený způsob řešení zadaných úkolů

6.1 Návrh a implementace logiky serveru pro FPS (First Person Shooter)

I když výše popisuji použité technologie, v této chvíli jsme ještě nevěděli jakou technologii použít pro serverovou část. Jisté bylo pouze to, že hra poběží v Unity a bude pro různé mobilní platformy.

Nejdříve jsme tedy museli s týmem prozkoumat možné řešení. Jedna z možností byla napsat si vlastní server. Ta ovšem byla zavrhnuta hned z několika důvodů. Největší důvody byly, že by server obsluhoval v reálném čase několik tisíc hráčů, takže naše řešení bude muset umět load balancing a nikdo ve firmě nemá s takovým projektem zkušenost. Za druhé, firma by musela přidělit do týmu více lidí. Zbývalo tedy použít software třetí strany.

Na Unity Store sice existuje několik řešení, ale žádné z nich není pro naše účely dostačující. Nakonec se kolegovi podařilo objevit Photon od Exit Games. Nabízí load balancing, je navržený speciálně pro multiplayerové hry a cenově přijatelný. V té chvíli to pro nás byla jasná volba. Kolega v týmu zprovoznil základní propojení serveru a klienta a mohli jsme se pustit do vývoje. Celkem rychle ale nastaly problémy v síťové komunikaci, převážně pokud měl klient špatné internetové připojení. Byl jsem pověřen, abych tyto problémy vyřešil.

Photon nabízí odesílání a přijímání operací nebo eventů. Operaci posílá klient serveru nebo server jednomu klientovi. Event je posílán serverem nebo klientem více klientům (kromě sebe). Jak jsem brzy objevil, chyba spočívala ve špatné komunikaci se serverem a nekonzistenci dat. Pokud chtěl klient poslat informaci o své pozici, rozeslal event všem klientům a serveru.

Při špatném spojení se tak mohlo stát, že hráč byl již dávno mrtvý, ale než tuto zprávu dostal, stihl ještě nějakou dobu střílet a eventuelně někoho zabít. Klienti s dobrým připojením tak mohli vidět střílet mrtvého hráče, což narušilo jejich hru. Situace s podobnou podstatou problému, kdy hráč střílel do pohybujícího se hráče. Hráč viděl, že ho zasáhl, ale přesto se nic nestalo. Než totiž informace s pozicí hráče A dorazila hráči B, byl hráč A už někde jinde. Proto se do něj nemohl nikdy trefit. Tento problém se zvyšoval, pokud byla odezva mezi klienty příliš vysoká. Další problémová situace nastala, když někdo položil něco na zem případně zničil překážku. Hráč, který se nově připojil tyto informace nedostal a tak neviděl stejnou scénu jako všichni ostatní. Jednotlivým řešením těchto situací se budu věnovat později.

Po důkladném rozebrání všech problémů vyplynulo, že server bude muset být centrem všeho dění a veškerá komunikace půjde přes něj. Aby v budoucnu nenastaly podobné potíže, musel jsem vypracovat základní pravidla, kterými se budou programátoři řídit, pokud budou zasahovat do serverové komunikace. Vytvořil jsem proto nádstavbu, jakousi komunikační vrstvu, která funkce Photonu (operace a eventy) zapouzdří.

```
Dictionary<byte, object> parameters = new Dictionary<byte, object>();
parameters.Add(0, ownerViewID);
parameters.Add(1, powerUpViewID);
parameters.Add(2, timestamp);
parameters.Add(3, actionCode);
```

```
parameters.Add(4, data);

PhotonNetwork.MyCustomOp(CustomOperationCodes.PowerUpAction, parameters, true);
```

Výpis 1: Příklad volání požadavku z klienta na server

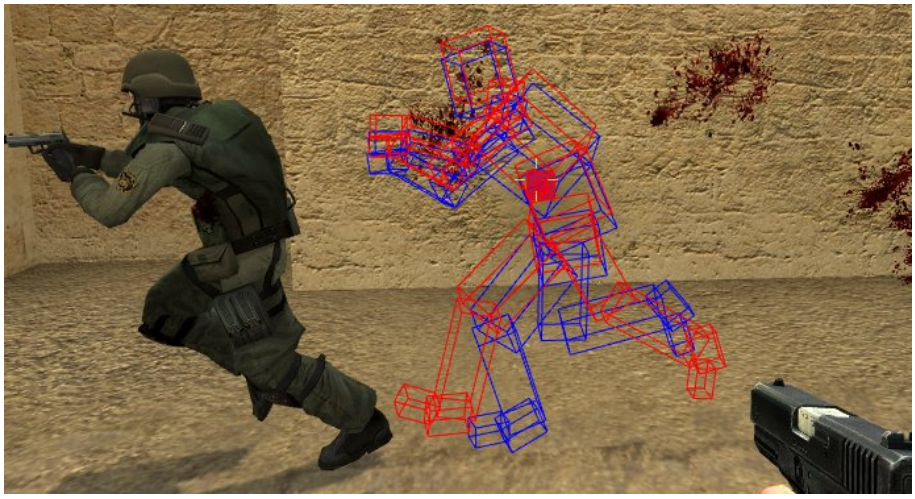
Klient vždy pošle na server požadavek (ve formě operace) jakou akci si přeje vykonat. Server požadavek vyhodnotí a rozešle příkazy na provedení akce všem klientům. Díky tomu můžeme zabránit např. cheaterům, aby narušovaly chod hry. Dejme tomu, že klient požádá server o použití lékárničky i když žádnou nemá (např. podvržením packetů). Server jednoduše zjistí, že hráč žádnou lékárničku nemá a požadavek ignoruje. Stejně tak můžeme jednoduše odhalit použití zamčených funkcí atd. Ve výsledku může také hráče zapsat do databáze, abychom ho mohli později prověřit. Jednoduše řečeno: Server má všechny data, serverové data jsou vždy správná a server rozhoduje o všem.

Díky tomu, že server má všechna aktuální data, může je poslat nově připojenému hráči, který konečně vidí stejnou scénu jako zbytek hráčů. Také se tímto vyřešil problém, kdy mrtvý hráč střílel. Server vidí, že je hráč mrtvý a proto požadavek na střelbu zahodí.

Když jsem nastavil pravidla komunikací se serverem, byla na řadě poslední chyba s nemožností zasáhnout hráče. Musím tedy předpokládat, že každý hráč má mít jinou rychlost a kvalitu připojení. V praxi to znamená, že informace které server rozešle dorazí každému klientovi v jinou dobu. Pokud se nepřítel pohybuje, dorazí hráči informace o jeho poloze zastaralá (obvykle v rámci 30-90ms, což může pro výsledek FPS hry hrát ohromnou roli).

Tušil jsem, že všechny online multiplayrové hry řešily stejný problém, proto jsem nechtěl znova vynalézat kolo a rozhodl se nastudovat postup jiných společností. Z těch nejznámnějších společností umožnuje nahlédnout do svých řešení jen hrstka. Nejvíce přínosné pro mě byly články a videa studií Bungie (Halo) a Valve Software (Counter-strike). Dozvěděl jsem se, že problém nejde úplně odstranit, ale metodou Lag compensation (odškodnění pomalé odezvy) můžu problém co nejvíce minimalizovat.

Na obrázku níže můžeme vidět co způsobí 200ti milisekundová odezva v již zmíněné hře Counter-strike. Jednotlivé kvádry jsou collidery detekující zásah připevněné na hráči.



Obrázek 2: Problém pozdního příchodu dat ve hře Counter-strike [6]

Červené collidery ukazují místo, kde byl hráč na obrázku (oběť), když byl zasažen hráčem z výhledu (agresor). Než tato informace dorazila na server (100ms), hráč se stihl posunout doleva, zcela mimo pozici, kterou server bude kontrolovat. Modré collidery ukazují místo vypočítané pomocí lag compensation. Metoda spočívá v tom, že server zpětně vypočítá pozici oběti, na které ji agresor v době výstřelu viděl. Pokud hráče dosadí na tuhle pozici a výstřel by ho trefil, počítá střelu jako zásah.

Pro implementaci jsem potřeboval uchovávat historii pozic hráče. Nejen pro tento účel vytvářím pro každého hráče na serveru objekt uchovávající si mimo ID hráče i aktuální a maximální počet životů, nasazenou zbraň apod. také seznam obsahující časové razítko s pozicí hráče. Jeden záznam má 16 bajtů (4 bajty časové razítko, 12 bajtů pozice), pokud síťová komunikace probíhá 60x za vteřinu, znamená to, že jeden hráč na serveru alokuje za vteřinu 960 bajtů, 1000 hráčů potom přibližně 1MB paměti za vteřinu a to je byla obrovská a zbytečná zátěž na paměť. Historii tedy nemůžu uchovávat na vždy a budu ji muset mazat. Na základě testů jsme se rozhodli uchovávat historii pozice po dobu 2 vteřin. Abych ušetřil i neustálou alokaci, vytvořil jsem pro každého hráče pole s konstantní velikostí 120 hodnot (2 * 60). V praxi se může stát, že hráč má pomalou odezvu a tak mezi prvním a posledním záznamem bude rozdíl více než 2 vteřin, to je ale spíše k užitku.

Výpočet času, ke kterému budeme získávat pozici je jednoduchý. Určím jej odečtením aktuální odezvy hráče od aktuálního času serveru. V tomto čase poté hledáme pozici. S největší pravděpodobností v historii pozic vypočtené časové razítko neexistuje. Většinou bude vždy menší nebo větší než některý z uložených časů. V takovém případě vypočítaný čas výstřelu je interpolací mezi dvěma nejbližšími časy v historii. Zjistím koeficient interpolace a použiju jej pro výpočet konkrétní pozice.

Ačkoliv jsem věděl, že se pravděpodobně jedná o nejlepší řešení, byl jsem nejprve skeptický a myslel si, že tohle řešení nebude fungovat. Po implementaci a následném celodenním testování však veškeré mé obavy odpadly. Pokud jsme nešli do extrému a

nesimulovali lag nad 500ms, kdy hráč různě mění směr pohybu, zásah bude vždy vyhodnocen správně.

Celý tento úkol byl velmi náročný. Nejenom, že jsme s týmem začínali na prázdném projektu, ale s realtime komunikací se serverem neměl nikdo ve firmě zkušenosti a tak všechno stálo na pokusech a omylech. Celý úkol včetně výběru technologie, nastudování cizího kódu, výzkumu problematiky, implementace a testování mi zabral přibližně 280 hodin.

6.2 Parsování položek do hry z XML

Tento úkol byl jeden z prvních, který jsem v Unity dostal za úkol. Dříve jsem totiž ve firmě pracoval v jiných enginech napsané pro jazyk C++ a Java. Hráč si může zvolit jednu ze tří postav (tříd). Každá postava má jiné předměty. Předměty se dělí do 3 kategorií: obleky, zbraně a power upy. Každý předmět může a nemusí vylepšit jednu z hráčových vlastností: životy, rychlost, zranění. Jako nejjednodušší mi přišlo uchovávat data o jednotlivých předmětech v XML. Dobře se čte a edituje, všechny programovací jazyky, se kterými jsem se setkal, s ním umí pracovat. Stromová struktura je pro uchování dat tohoto druhu ideální.

Celkem dlouho jsem přemýšlel, jestli mít zvlášť XML pro kategorii předmětu nebo zvlášť pro každou třídu. V době implementace totiž nebylo stoprocentně rozhodnuto jaké předměty ve hře budou a co budou umět. Nakonec jsem se rozhodl vše napsat do jednoho souboru s tím, že základním elementem je třída postavy, následují elementy jednotlivých kategorií a jednotlivé předměty mají atributy s vlastnostmi.

Následovalo parsování v unity a zobrazování v menu. Samotné GUI jsem vytvořil celkem rychle. Potíže jsem měl s orientací ve struktuře Unity. Musel jsem si nastudovat k čemu slouží jednotlivé komponenty, co jsou "game objects" apod. Každá třída v Unity musí dědit ze třídy MonoBehavior, což způsobuje, že nemůžu instancovat objekt třídy, ale vždy musím vytvořit game object, na který třídu přidám jako komponentu. Ve firmě již bylo několik lidí, kteří v Unity půl roku až rok pracovali, takže když mi něco nebylo jasné, proškolili mě.

Vzhledem k tomu, že to byla moje první práce v Unity, zabral mi úkol asi 50 hodin. Dnes si myslím, že bych vše zvládl do 10 hodin.

6.3 Skóre tabulka

Obrázek 3: Zadáná předloha GUI pro tabulku výsledků

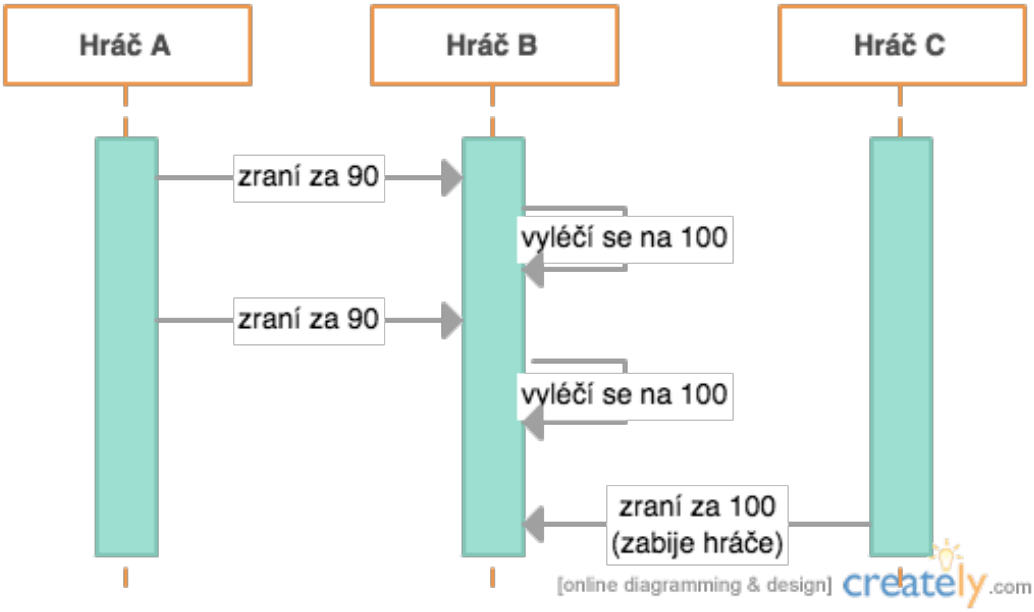
Nejprve jsem si s týmem ujasnil za co všechno budeme přičítat hráčům skóre a jakým způsobem budeme určovat asistenci zabití. Z diskuze vyplynulo, že se skóre bude započítávat nejen za zabití nebo asistenci, ale také za různé mini úspěchy jako např. počet zabití v řadě, poškození nepřátelské základny, použití power upu apod. Proto jsem se rozhodl vypracovat zvlášť rozhraní pro počítání zabití, asistencí a smrtí a zvlášť rozhraní pro počítátní skóre.

Jako první jsem si připravil třídu pro práci a uchování skóre hráče. Jelikož klient nesmí o podobných věcech rozhodovat, provedl jsem implementaci na serverové části. Díky tomu, kdykoliv budeme potřebovat hráči přičíst skóre, stačí přistoupit k jeho objektu na serveru a zavolat metodu AddScore s parametry ScoreType a celočíselnou hodnotou skóre, které chceme přičíst. Parametr hodnoty skóre není povinný, v takovém případě se dosadí defaultní hodnota podle zadaného typu. Uvnitř metody se zavolá příslušný manager, který hodnotu přičte, aktualizuje tabulku a rozešle eventy s novými hodnotami všem klientům.

Poté jsem začal implementovat rozhraní pro počítání zabití, asistencí a smrtí (statistiky). V průběhu implementace jsem si však uvědomil, že zde mám mnoho duplicitních volání a proto jsem rozhraní spojil s výše zmíněným skóre rozhraním. V případě, že hráč někoho zabije, asistuje u zabití nebo umře, zavolá se také metoda AddScore se stejnými parametry a podle ScoreType uvnitř rozhodnu, kterou statistiku inkrementuju.

Samotná detekce zranění a zabití byla již vypracována (její fungování jsem nastínil v předchozí části), takže jsem volání metod pouze doplnil na příslušná místa. Detekce asistence však chyběla, proto jsem ji musel vytvořit.

Jako nejlepším řešením se mi zdálo, aby si každý hráč uchovával seznam (hitList) ID hráčů, kteří ho zranili spolu s celkovou hodnotou zranění. Pokud hráč umře, vyberu maximální hodnotu z hitListu a dostanu ID hráče, který si započítá asistenci. Avšak nesmí to být hráč, který si připisuje zabití, v takovém případě vyberu hráče s druhou maximální hodnotou nebo žádného, pokud takový neexistuje. Při následném testování se ukázalo, že jsem nebral v úvahu, kdy se hráč vyléčí ze svých zranění. Jednu ze situací se pokusím vysvětlit na následujicím sekvenčním diagramu.



Obrázek 4: Problém, který může nastat v první verzi hitListu

V momentě, kdy se hráč B (nejen) zcela vyléčí, měl by být hitList prázdný, protože hráč A už hráči C při zabití nepomáhal. Data jsou také nekonzistentní, protože hráč má pouze 100 životů a bylo mu ubráno 280. Proto jsem logiku hitListu přepracoval na historii zranění. Atributy ID hráče a hodnota zranění zůstaly. Zůstal i způsob vyhodnocení asistence, pouze jsem musel celkovou hodnotu zranění sčítat až při vyhodnocování. Přidal jsem však mazání z této hitorie. Pokaždé když se hráč vyléčí, smažou se nejstarší záznamy o zranění v celkové hodnotě vyléčených bodů.

```
public void RemoveDamageFromHistory(float damageToRemove)
{
    for (int i = 0; i < hitList.Count; i++) {
        int viewID = hitList[i].shooterViewID;
        float damage = hitList[i].damageGiven;
        damage -= damageToRemove;
        if (damage <= 0) {
            damageToRemove = -damage;
            hitSummaryHistory[viewID] -= hitList[i].damageGiven;
            hitList.RemoveAt(i);
            i--;
        } else {
            hitSummaryHistory[viewID] -= damageToRemove;
            StatiscticsHitInfo tmp = hitList[i];
            tmp.damageGiven = -damage;
            hitList[i] = tmp;
            break;
        }
    }
}
```

Výpis 2: Mazání historie zranění z hitListu

Poslední fáze bylo nasazení samotného GUI. Kromě odchytávání události o aktualizaci skóre, je to jediná část tohoto zadání, kterou jsem řešil na klientské části. Od grafika jsem si vzal potřebné podklady a domluvili jsme se jakým stylem se grafika bude chovat při různých rozlišeních a poměrech obrazovky. Samotná implementace, která probíhala v NGUI, byla celkem rychlá. Během práce jsem zachoval konzistentní hierarchii, kterou jsme si určili pro práci s uživatelským rozhraním. To znamená, že jsem použil nový GameObject (základní objekt v Unity), který byl rodičem celé tabulky výsledků a byl nezávislý na jiných objektech. V praxi to znamená, že pokud někdo chce tabulku zobrazit, stačilo tento GameObject zapnout a o nic dalšího se programátor/skriptér starat nemusel.

Během vývoje jsem narazil na problém, že zranit hráče nemusí jenom jiný hráč, ale i překážka (např. vybuchující barel) nebo power up jiného hráče. Po domluvě v týmu jsme si řekli, že pokud hráče zabije power up, chceme to v hitListu rozlišovat, abychom mohli hráči přičíst více bodů. Proto je potřeba ukládat ID power upu a ne hráče. Power upy ale používaly zcela odlišné číslování než hráči a přijatelný způsob oba typy ID uchovávat neexistoval, proto bylo potřeba ID sjednotit. S kolegou jsme společně vymysleli řešení, že hráči budou mít ID číslované po tisíci (1000, 2000, ...) a veškeré power upy, které vyvolají bude číslované jako ID hráče + pořadí (2001, 2002, ..., 2999). Pokud nenastane neočekávaná chyba, hráč nemůže stihnout vyvolat více než 100 power upů za hru, proto by rozsah měl být dostatečný. ID v rozsahu 0 - 999 byly vyhrazeny pro objekty na mapě, které musí být řízeny ze serveru (zničitelné překážky a barely). Implementace byla přidělena kolegovi.

Vyhledem k tomu, že bylo potřeba doimplementovávat většinu funkcionality na serverové části si myslím, že jsem úkol splnil celkem rychle. Celý úkol včetně porad a samotného testování mi zabral asi 60 hodin.

6.4 Optimalizace

Nejedná se o jednorázové zadání. Během vývoje jsme se opakovaně po pár týdnech vrátili ke "starému" kódu (někdy cizímu), abychom jej mohli vylepšit. Obzvláště pokud to byla nějaké základní mechanika v jádru systému. Ne vždy se jednalo o složité a zdlouhavé optimalizování. Některé optimalizace zabraly pár desítek minut jiné pár hodin. Objevily se ale i takové, u kterých jsem musel spoustu věcí reimplementovat a zabraly i několik dnů. Vybral jsem si proto 2 optimalizace, které mi zabraly nejvíce času (jedná se o pracovní názvy). Optimalizace multihit a optimalizace explosion tree

Nejdříve začnu optimalizací multihit. Ve hře jde zranit najednou nejenom více hráčů, ale i více objektů a aktivních power upů. Pokud bylo způsobeno zranění více objektům najednou, jedná se z 99% o výbuch. Například raketa má při výbuchu plošné zranění a teoreticky může v jednu chvíli zranit všech zbylých 7 hráčů, zároveň také přibližně 3 překážky (závisí na lokaci hráče) a několik aktivních power upů. Ve výsledku se tak v jednu chvíli zavolá např. 12 operací s žádostí o zranění. Server každou vyhodnotí a 12

pošle aktualizovanou tabulku. Na klientovi se potom tabulka 12x aktualizuje. 11 volání je zde teda zcela zbytečných a zbytečně zatěžují komunikaci. Jedna aktualizace tabulky může být poslání až 128 bajtů, rozeslání všem osmi hráčům ve hře je tedy 1kB. Zbytečně se tedy posílá 11kB dat při výbuchu.

Dalším problémem bylo, že při vytvoření nového objektu způsobující zranění více zranění najednou, musel si naprogramovat řízení celé záležitosti sám. Většinou tak docházelo ke kopírování starého kódu a vznikaly chyby z nepozornosti. Tomu jsem chtěl také zamezit.

Vycházel jsem z aktuálně naprogramovaných metod posílající žádosti na zranění zvlášť pro hráče, překážky a power upy. Každá žádost musí obsahovat ID agresora (může být hráč, ale také power up či překážka) a časovou známku žádosti o request, tyto informace se tedy posílaly duplicitně. I z hlediska programování se volaly stále stejné metody pořád dokola, proto jsem vytvořil na klientské části statickou třídu Multihit, která by měla pomoct. Pokud programátor chce vyvolat jakýkoliv typ exploze nebo jiného plošného zranění, stačí zavolat ve třídě Multihit metodu GetData, kde do parametrů předá pozici výbuchu, jeho radius a ID agresora (hráč nebo power up). Metoda vrátí veškerá data, které stačí odeslat na server s požadavkem na multihit. Na serveru jsem naprogramoval metodu, která si data přebere, zparsuje, ověří zda objekty v přijatých datech opravdu byly na těchto místech a rozdá zranění objektům a nakonec aktualizuje a rozešle aktualizované výsledky. Pokud tedy není potřeba psát speciální ošetření, nemusí se do serverové části v tomto ohledu dále zasahovat.

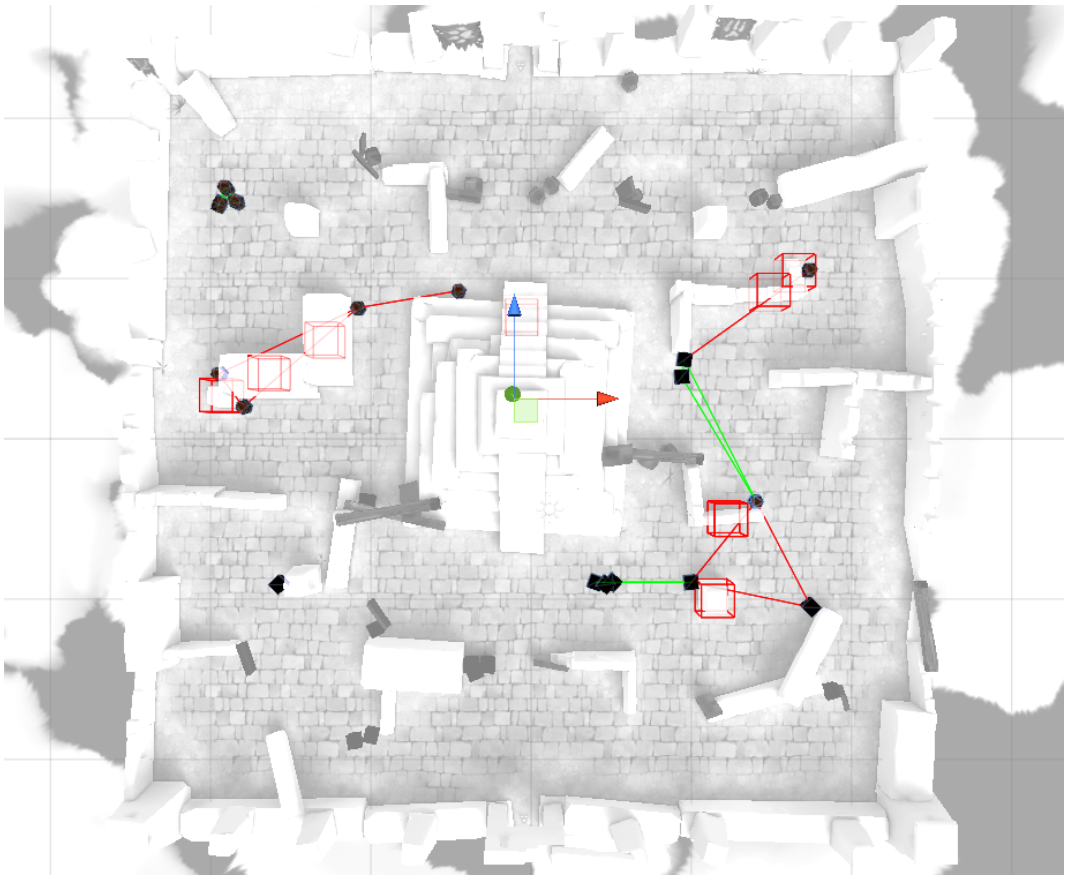
Přestože jsem musel přepisovat značnou část kódu, ušetřil jsem v této části přes 70% datového přenosu a díky jednoduchému volání metod a možnosti nezasahovat na server jsem zmenšil pravděpodobnost zbytečných lidských chyb při dalším rozšiřování hry, které zbytečně prodlužovaly další vývoj i o několik dnů.

Nyní nastíním problém, kdy jsem musel pro řešení použít explosion tree. Ve hře jsou na mapě rozmístěny barely, které po zásahu výbuchem zraní hráče. Pokud je v blízkosti a přímé viditelnosti jiný barel, musí vybuchnout také. Dosavadní řešení bylo, že barely bouchaly i když byly za zdí, protože server nemá informace o tom, kde se nacházejí statické překážky. I kdyby tyto informace měl, neumí určit přímou viditelnost narozdíl od klienta, který běžel na Unity. Spoléhat se, že klient pošle vždy správné informace jsme nechtěli, poučili jsme se z předchozích chyb.

Mnou navrženým řešením bylo, aby server u každé mapy věděl, které barely jsou navzájem mezi sebou v přímé viditelnosti již předem. Jelikož tyto barely se objevují vždy na stejných místech, můžeme tyto data získat již při dodání mapy od Level designera. Pro automatizaci jsem vytvořil skript, který se pustí na každé mapě, která ve hře bude. Ten vyhledá na mapě všechny vybuchující objekty (pomocí speciálního tagu). Posléze porovnává každý objekt s každým na přímou viditelnost. Vyšle mezi těmito dvěmi pozicemi raycast a prochází všechny objekty, které zasáhl. Pokud narazí na statickou překážku, znamená to, že přímá viditelnost neexistuje a proto jde na další objekt. Pokud se žádná překážka mezi objekty nevyskytuje nebo se jedná o zničitelnou překážku, barel musí být při výbuchu toho druhého zničen a proto si jej uloží do seznamu.

Pro rychlou kontrolu jsem jednotlivé raycasty zobrazil. Designer tak může okamžitě

vidět, kterou výbušninu posunout dál, aby vybuchla, nebo který objekt kvůli špatnému umístění nevybuchne. Pro lepší představu přikládám obrázek ukázkové mapy, kdy jsem zvýraznil pouze raycasty. Zelené čáry znázorňují, že se dané 2 objekty výbuchem ovlivní, zelené potom, že jim stojí v cestě překážka. Konkretní místo střetu s překážkou je znázorněno červenou krychlí. Pokud mezi objekty není žádná čára, výbuch objektu nedošáhne na druhý objekt.



Obrázek 5: Grafické znázornění vygenerovaného XML pro explosion tree

Jakmile projde všechny objekty, vytvoří ze zjištěných dat XML, které stačí nahrát na server a přiřadit ke konrétní mapě. Server si při spuštění nové mapy načte příslušné XML a pokud dojde k výbuchu jednoho z objektů, zjistí si ze seznamu všechny další objekty, na kterých má zavolat výbuch. Serverovou implementaci už dělal kolega.

Celkově jsem na optimalizacích strávil přibližně 100 hodin. Multihit mi trval přibližně 40 hodin hlavně z důvodu přepisování kódu hluboko v jádru aplikace. Explosion tree mi zabral asi 8 hodin.

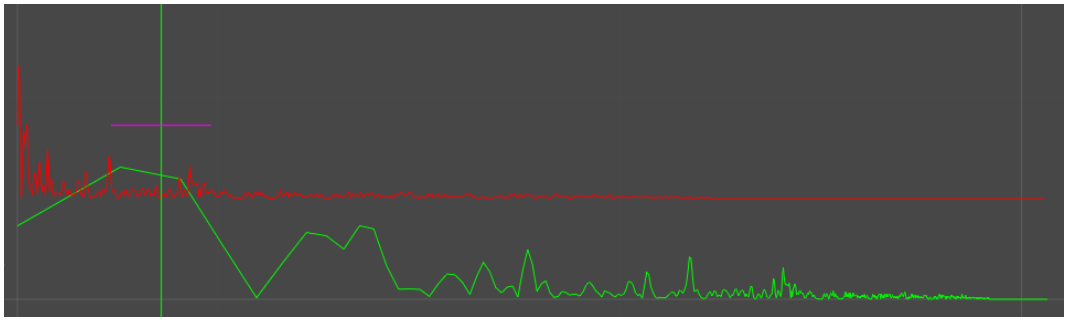
6.5 Audio pulser

Když reproduktor vydává zvuk (převážně nízké tóny - basy), dochází k charakteristické vibraci membrány. Ve skutečnosti je to však naopak, protože tato membrána zvuk vytváří, ve hře musím však postupovat obráceně. Mám zvuk a potřebuju zjistit kdy se má membrána zvětšit. Mohl bych určit, aby se membrána pohybovala pravidelně do nějakého rytmu, ale řekl jsem si, že takový výsledek hráči chtít nebudou.

Existují grafické ekvalizéry, které aktuálně přehrávaný zvuk získávají a převádí je do grafu (viz obrázek níže). Když bych získal tento graf, mohl bych sledovat hodnotu funkce v bodě X a podle výstupu měnit velikost membrány. Po chvilce pátrání na internetu jsem našel, že Unity mi tyto data umí vrátit v poli o velikosti mocniny 2. V dokumentaci získávali pole velikosti 1024, rozhodl jsem se tuto velikost zachovat.

Pro debug jsem se tyto data rozhodl vykreslit do křivky (viz obrázek níže, červená křivka). To umožní někomu, kdo bude reproduktory nastavovat, podívat se, která část spektra se pohybuje v rytmu jakém požaduje pohyb membrány. Všiml jsem si, že body pulzují pouze na prvních přibližně 10% křivky a zbylých 90% je téměř nehybných. To příliš komplikovalo správný výběr bodu X na křivce, abych dostal požadovanou hodnotu na ose Y. Rozhodl jsem se funkci zlogaritmovat (zelená křivka). To mi umožnilo spektrum více rozložit a lépe se orientovat ve výstupních datech.

Nakonec jsem přidal svislý zelený ukazatel, který znázorňuje vybraný bod X v grafu a fialovou vodorovnou linku, která ukazuje nejvyšší hodnotu za poslední půl vteřinu ve vybraném bodě.



Obrázek 6: Spektrum zvukove stopy v daném čase

Podle návrhového vzoru Observer jsem vytvořil objekt, který tyto data každý snímek obrazovky získal a uchoval. Následně měl v poli uloženy všechny membrány, které informoval o změně a ty se aktualizovaly. Každá membrána mohla sledovat jinou část spektra a měla nastaveno o kolik se posune a zvětší v závislosti na hodnotě ze sledovaného spektra. Hodnota posunu a zvětšení se ještě násobila intenzitou (také pro každou membránu zvlášť), jelikož hodnoty z výstupu funkce byly příliš malé.

Celý úkol mi zabral celý jeden den, tedy 8 hodin.

6.6 Refaktorizace z Unity 4.6 do Unity 5

Unity 5 slibuje větší výkon a lepší optimalizaci pro mobilní zařízení a také přídává vlastní nádstavbu pro uživatelské přostředí - UGUI. Do této doby jsme používali NGUI od výrobce třetí strany, které obsahovalo několik chyb a tak znepříjemňovaly vývoj. Kvůli novému GUI

Nejdříve jsem si tedy spolu se stáhnutím nového Unity prošel oficiální videotutoriály UGUI. Na první pohled to vypadalo, že refaktorizace relativně snadno. Bohužel pro nás Unity 5 zavedlo nový systém práce s Eventy a práce v UGUI se až příliš lišila od stávajícího NGUI. Po dvou dnech stráveným refaktorizací jsem týmu a vedení přednesl mé poznatky a rozhodli jsme se, že projekt dokončíme v Unity 4.6.

6.7 Rozpohybování kamery za běhu hrdiny

Nedostal jsem zadání jak konkrétně by se měla kamera pohybovat, jelikož to nikdo ve firmě nedokázal předem říct. Měl jsem tedy postupovat metodou pokus omyl. Rozhodl jsem se, že nebude dobré psát instrukce pro pohyb přímo do kódu, jelikož by později při požadavku na jakoukoliv změnu bylo potřeba rozluštit napsaný algoritmus a i přes sebelepší dokumentaci by práce byla příliš dlouhá a složitá.

Řekl jsem si, že vytvořím v unity nějaké grafické rozhraní, ke kterému si bude moct někdo sednout, pohrát si s hodnotami a určit nejlepší výsledek. Při tomto úkolu jsem poprvé použil křivky v Unity. Vytvořil jsem křivku pro osu X a zvlášť pro osu Y, která v čase určovala o jakou hodnotu (offset) se kamera posune z původní pozice. Jednotlivé hodnoty jsem násobil další křivkou, která měla zajistit, aby se kamera nebýbala ve smyčce pořád stejně a zajišťovala drobné odchylky v prvních dvou křivkách. Abych toho dosáhl, byla křivka dělší oproti prvním dvoum v násobku 1,31.

Hotové rozhraní jsem předal grafikům, kteří mají pro takové věci lepší cítění, aby zkusili nastavit správné hodnoty. Ukazálo se však, že rozpohybování kamery způsobilo problém při přesném míření během hry, jelikož se tím zároveň pohyboval i zaměřovač. Zkusil jsem přeprogramovat míření, aby nezáviselo na kameře. To ovšem znamenalo, že se musí pohybovat zaměřovač, což vadí uplně stejně. Požadavek na rozpohybování kamery tak byl přehodnocen a zrušil se.

Návrh a implementace rozhraní pro nastavování kamery jsem měl hotové přibližně za 10 hodin, úkol se ale protáhl o další experimentování kvůli vzniklým problémům a proto jsem na tomto zadání strávil asi 40 hodin.

6.8 Shader průhlednosti překážek

Pokud si hráč stoupnul zády k překážce, kamera vlezla za tuto překážku a hráč jednak nešel vidět, ale hlavně neviděl, kam míří, jelikož překážka zakryla celou kameru. Ve hrách se to často řeší “inteligentní” kamerou, která když narazí na překážku, která by ji zakryla, posune se mimo svou běžnou pozici, aby zajistila hráči viditelnost. Většinou je ale vývoj takové kamery příliš dlouhý a nákladný.

Přednesl jsem návrch, že by se mohla překážka, která brání ve výhledu zneviditelnit, případně alespoň trochu zprůhlednit. Odezva byla pozitivní s tím, že nezmizí celá překážka, ale zprůhlední se pouze oválná část okolo hráče, aby šlo vidět kam míří. Měl jsem za úkol vypracovat shader, který takto překážku upraví. Detekce, kdy a na jaké překážce se má shader zapnout dostal na starost kolega.

Implementace byla celkem snadná, jelikož hráč je téměř uprostřed obrazovky, takže jsem v shaderu pouze zjistil na jakou pozici na obrazovce se pixel vykresluje a pokud se blížil středu tak jsem mu snižoval alfu až byl zcela průhledný.

Na řešení úkolu jsem potřeboval přibližně 3 hodiny.

7 Závěr

7.1 Uplatnění teoretické a praktické znalosti

Během absolvování odborné praxe jsem uplatnil znalosti získané během studia z několika předmětů. Bylo zapotřebí znát jazyk C# (PJ2), základy uživatelského rozhraní (URO), práci s grafikou (APPS, ZPG) a návrh algoritmů (ALG1, ALG2, SWI). Během studia jsem se o vývoj her zajímal samostudiem teoreticky i prakticky, takže jsem pochytil základní principy fungování těchto systémů. Nejvíce zkušeností jsem získal na konci prvního ročníku, když jsem dostal možnost pro firmu Craneballs pracovat.

7.2 Scházejicí znalosti

Přestože pro firmu pracuji již delší dobu, během odborné praxe jsem narazil na oblasti vývoje, se kterými jsem doposud žádné zkušenosti neměl. Konrétně vývoj v Unity byl pro mě zcela novou a velmi osvěžující zkušeností. Nesetkal jsem se také se serverovou komunikací v reálném čase s několika klienty zároveň. Možnost pracovat na klientovi a zároveň serveru a zajistit vzájemnou komunikaci je pro mě také velmi cenná zkušenost.

7.3 Dosažené výsledky v průběhu praxe a jejich zhodnocení

Absolvování bakalářské práce formou odborné praxe jsem si zvolil převážně z důvodu, že si myslím, že zkušenosti se praxí získávají mnohem rychleji, lépe a jsou v budoucnu více ceněny. Jakmile bude projekt, na kterém jsem se podílel, dokončen, mnou vyvíjenou hru budou hrát tisíce až miliony lidí. Právě hráči jsou skupina uživatelů softwaru, kteří jsou nekompromisní a velmi rádi ukáží svou nespokojenost s produktem.

V novém týmu jsem měl možnost vyzkoušet si v omezené míře i vedení týmu, převážně zadávání práce grafikům a plánování porad. Právě v oblasti vedení týmu bych chtěl v budoucnu ještě hodně zapracovat.

Pro firmu dále pracuji a odbornou praxi hodnotím zcela kladně. Jsem velmi rád za možnost, které se mi dostalo.

8 Reference

[1] Craneballs [online]. © 2012 [cit. 2015-06-22].
Dostupné z: http://www.craneballs.com

[2] Unity - Game engine [online]. © 2015 [cit. 2015-06-22].
Dostupné z: https://unity3d.com

[3] About · GitHub [online]. © 2015 [cit. 2015-06-22].
Dostupné z: https://github.com/about

[4] Atlassian SourceTree [online]. © 2014 [cit. 2015-06-22].
Dostupné z: https://www.sourcetreeapp.com

[5] Photon Server SDK. *Exit Games* [online]. © 2003-2015 [cit. 2015-06-22].
Dostupné z: http://doc.exitgames.com/en/onpremise/current/sdks-and-api/sdk-photon-server

[6] Lag Compensation. *Source Multiplayer Networking* [online]. 27.6.2014 [cit. 2015-05-22].
Dostupné z: https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking