Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Bakalářská práce

Hra ve frameworku Monogame

Jakub Krejdl

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Jakub Krejdl

Informatika

Název práce

Hra ve frameworku Monogame

Název anglicky

Game in Monogame framework

Cíle práce

Cílem práce je vytvořit dvojrozměrnou plošinovou hru za použití frameworku Monogame, ve které bude hráč s úrovněmi získávat lepší vybavení a schopnosti. Bude zde také zaměření na rozmanitost npc stejné třídy, tak aby hra získala na komplexitě. Na výběr bude k dispozici více typů hry, jako je klasický scénář a úrovně generované náhodně. Hra bude představovat plnohodnotný produkt rozšiřitelný skrze JSON soubory.

Metodika

Práce bude rozdělena na tři hlavní části, analýza, návrh a vlastní implementace. V první části budou rozebrány herní dynamiky podobného žánru, z toho budou odvozeny potřebné algoritmy a techniky. V části druhé bude navrhnut herní engine, optimalizace vykreslování a výpočtů s důrazem na znovupoužitelnost kódu a snadnou rozšiřitelnost. Vlastní implementace bude realizována v jazyce C#.

Doporučený rozsah práce

30-40 stran

Klíčová slova

Monogame, C#, Windows, Hra, HLSL, JSON, OpenGL

Doporučené zdroje informací

GOLDSHTEIN, Sasha., Dima. ZURBALEV a Ido. FLATOW. Pro .NET performance. Expert's voice in .NET. ISBN 978-1-4302-4458-5.

MAREŠ, Amadeo. 1001 tipů a triků pro C# 2010. Brno: Computer Press, 2011. ISBN 978-80-251-3250-0.

MCMILLAN, Michael. Data structures and algorithms using C#. New York: Cambridge University Press, 2007. ISBN 978-052-1876-919.

REED, Aaron. Learning XNA 4.0. Sebastopol, CA: O'Reilly, 2011. ISBN 14-493-9462-0.

SHARP, John. Microsoft Visual C# 2010: Krok za krokem. Brno: Computer Press, 2012. ISBN 978-80-251–3147-3.

Předběžný termín obhajoby

2017/18 LS – PEF

Vedoucí práce

Ing. Marek Pícka, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 11. 1. 2018

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 11. 1. 2018

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 09. 03. 2018

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Hra ve frameworku Monogame" jsem vypracoval(a) samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 13.3.2018 _____

Poděkování

Rád bych touto cestou poděkoval Panu Ing. Píckovi, Ph.D. za odborné vedení této práce.

Hra ve frameworku Monogame

Abstrakt

Práce je zaměřena na vývoj platformové dvojrozměrné hry s použitím frameworku Monogame. Hra je naprogramována v programovacím jazyce C# s použitím jazyka HLSL pro grafickou úpravu hry. K vývoji hry bylo použito vývojové prostředí Microsoft Visual Studio 2017.

Práce samotná je rozdělena na teoretickou a praktickou část. teoretická část práce se zabývá základními prvky použitými při programování hry. Je zde popsána jejich stručná historie a použití programovacích jazyků C# a HLSL a základní vlastnosti herní fyziky a těles ve hře stejné kategorie.

Praktická část se zabývá již samotným naprogramováním a je rozdělena na analýzu, návrh a samotnou implementaci. Analýza zkoumá cesty, jak dosáhnout cíle a zvažuje možná řešení. V návrhu se rozebírají vlastnosti částí programu použité při tvorbě hry. V implementaci jsou prezentovány samotné kusy kódu jako ukázka funkcionality.

V závěru jsou rozebrány výsledky a cesty jakými se musel autor ubírat při samotném vývoji.

Klíčová slova: hra, program, C#, C sharp, HLSL, platformer, Monogame, XNA, 2D

Game in the Monogame framework

Abstract

The aim of this work is to develop the two dimensional platform game with the usage of Monogame framework. The game is programmed in C# programming language with the use of HLSL for graphic adaptation. The game was created in the environment of development tool Microsoft Visual Studio 2017.

The work is divided in two parts, theoretical and practical. The theoretical part of this work describes the basic elements used while programming the game. This part also describes brief history and usage of C# and HLSL languages and basic characteristics of ingame physics and of figures within the same category.

The practical part of this work is based on the programing itself and is further divided into three parts: analysis, design and implementation. The analysis focuses on different ways to achieve the goal of this work and provides the possible solutions. The design then focuses on the program properties used in the development of the game. The implementation presents various parts of the code to show its functionality.

The paths that were used to reach the specific results are covered in the final part of the work.

Keywords: game, program, C#, C sharp, HLSL, platformer, Monogame, XNA, 2D

Obsah

Seznam obrázků

Seznam tabulek

Seznam útržků programu

Úvod

První počítačové hry se na světě objevily již s prvními počítači disponujícími grafickým rozhraním a již mnoho dekád prochází, ruku v ruce s výpočetními zařízeními, rychlým tempem vývoje. I když technologie hardware počítače výrazně pokročily, dosud není možné řešit fyzikálně přesné simulace na domácích počítačích v reálném čase. I dnešní hry využívají zjednodušený fyzikální model a pouze přibližné vykreslování zobrazované reality. Optimalizace a šetření výpočetním výkonem je stále nutností.

Tato práce je zaměřena na vývoj počítačové hry v moderním programovacím jazyce C# vyvinutého společností Microsoft v jeho poslední verzi 7.0. Hra bude stavět na open source frameworku Monogame, který je pokračovatelem XNA game studia, od společnosti Microsoft, která však tento projekt ukončila v roce 2014. Monogame respektuje, stejně jako jeho předchůdce, základní konstrukci her a dává široký prostor k přizpůsobení. Přichází však se základními konstrukty, které hry sdílí a které značně usnadňují a urychlují začátek a proces vývoje.

Cíl práce a metodika

Cíl práce

Cílem hry je naprogramovat dvojrozměrnou plošinovou hru, za pomoci jazyka C# a open-source frameworku Monogame. Hra bude obsahovat vlastní herní engine, který bude obsahovat správu obrazovek s vlastními UI elementy, nahrávání obsahu JSON souborů a vlastní jednoduchou simulaci herní fyziky. Hra samotná bude obsahovat NPC postavy. Bude také obsahovat interaktivní prvky jako výtahy, dveře, vozidla a nositelné objekty. Hráč bude moci sbírat body za poražené NPC a přes obchodní místa ve hře vylepšovat svoje vybavení.

Grafická úprava bude probíhat přes HLSL shadery a bude obsahovat jednoduché osvětlení s normálovými texturami pro přidání dojmu hloubky.

Hra by měla být ve výsledku projekt, který bude moci být dále rozšiřován a upravován s možností přidání vlastního obsahu.

Metodika

Teoretická část práce bude pojednávat o použitých technologiích, jejich stručné historii a způsobu využití. Prvním prvkem praktické části bude analýza. Budou v ní rozebrány herní mechaniky použité v podobných hrách a z toho dále odvozeny i s ohledem na dostupné technologie, se kterými se bude pracovat, možné realizace kódu.

Druhým procesem v pořadí bude návrh objektů a herní struktury, například herního enginu jako celku, hráče a objektů ve hře samotné. Dále bude navržena struktura dědičnosti a vzájemná odezva prvků hry.

Konečným stupněm vývoje, vlastní implementací, se myslí naprogramování vlastní hry, tříd a technik HLSL. Na vývoj hry bude využit open-source framework Monogame. Vývoj bude probíhat ve vývojovém prostředí Visual Studio 2017.
To bude využito i při editaci JSON souborů a souborů HLSL. Hra bude v této práci určena pouze pro platformu Windows. Pro tvorbu grafiky bude využit GIMP.

Teoretická východiska

Jazyk C#

Tým společnosti Microsoft vedený Andersem Hejlsbergem spolu s Scottem Wiltamuthem a Peterem Goldem vytvořili jazyk C#, který byl vydán roku 2000 jako jazyk pro vytváření aplikací pro běhové prostředí .NET, jehož cílem bylo být moderní, jednoduchý, všestranný, objektově orientovaný. Na svém počátku byl alternativou k jazyce Java na platformě Windows. (ECMA-334:2017, 2017)

C# je typově bezpečný jazyk, syntaxí podobný jazykům C a C++, ze kterých přímo vychází. Běhové prostředí .NET dává aplikacím naprogramovaným v tomto a dalších mnoha jazycích možnosti přenositelnosti, typové bezpečnosti, kontroly indexů u polí, správy paměti a jiné. V důsledku zajišťuje stabilnější a bezpečnější aplikace.

Kompilátor zkompiluje napsaný program nejdříve do Intermediate Language (IL), který je podobný assembleru, takto zpracovanému kódu se říká assembly. Oproti programu napsanému v assembleru má IL kód výhodu přenositelnosti, protože tvoří pouze mezikód mezi strojovým kódem a jazykem C#. V .NET framework programu se o kompilaci na strojový kód stará Common Language Runtime (CLR). Je to virtuální stroj, který využívá (JIT) just-in-time kompiler pro překlad, když spustíme program. (Stephens, 2014)

Monogame

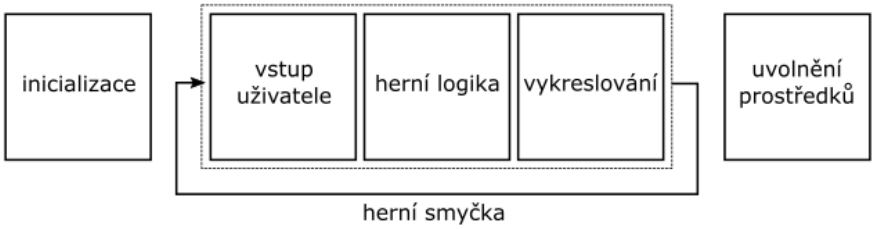
1.1.1 XNA a Monogame

Monogame (Monogame, 2018) je open source implementace XNA game studia společnosti Microsoft, která se snaží zachovat hierarchii oboru názvů a strukturu tříd XNA 4 frameworku, jehož vývoj byl oficiálně ukončen v dubnu roku 2014. Framework Monogame podporuje mnoho platforem, mimo počítačů to je široká škála herních konzolí a moderních operačních systémů pro mobilní telefony typu Android, iOS a WindowsPhone 8.

1.1.2 Herní smyčka

Každá počítačová hra obsahuje podobnou sekvenci po sobě následujících operací:

1. Inicializace a nahrání obsahu hry
2. Nekonečná herní smyčka, kde probíhá většina výpočtů a vykreslování
    i. Vstup uživatele
    ii. Herní logika
    iii. Vykreslování
3. Finalizace a uvolnění prostředků



Obrázek 1 - herní smyčka

Přičemž vykreslování probíhá stejně často nebo častěji než Update herní logiky. Pokud vykreslování probíhá častěji, je nutná v metodě vykreslování interpolace pohybu objektů mezi snímky. Nutno podotknout, že takto po sobě následující bloky příkazů nejde ve většině případů zaměňovat.

1.1.3 Realizace základní struktury hry

```
public class Game1 : Game
{
  GraphicsDeviceManager graphics;
  SpriteBatch spriteBatch;

  public Game1()
  {
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
  }

  protected override void Initialize()
  {
    base.Initialize();
  }
```

```
    protected override void LoadContent()
    {
        spriteBatch = new SpriteBatch(GraphicsDevice);
    }

    protected override void UnloadContent(){}

    protected override void Update(GameTime gameTime)
    {
        if (GamePad.GetState(PlayerIndex.One).Buttons.Back
uttonState.Pressed || Keyboard.GetState().IsKeyDown(Keys.Escape))
            Exit();

        base.Update(gameTime);
    }

    protected override void Draw(GameTime gameTime)
    {
        GraphicsDevice.Clear(Color.CornflowerBlue);

        base.Draw(gameTime);
    }
}
```

Kód 1 - monogame třída Game1

GraphicsDeviceManager je třída, která spravuje nastavení grafické jednotky. Vlastní chování lze docílit odvozením vlastní třídy. Třída *SpriteBatch* se stará o posílání dvojrozměrné grafiky na grafickou kartu po skupinách (batches), kde jednotlivé sprity sdílejí stejné nastavení. Dále je nutné podotknout že, jako víceméně každém herním enginu, osa x směřuje doprava a osa y dolů. Nulové souřadnice jsou tedy v levém horním rohu. *Content* je instancí třídy *ContentManager*, která se stará o nahrávání a životní cyklus obsahu ze souborů a třída *GameTime* obsahuje časové konstanty jako je celkový čas aplikace nebo čas mezi jednotlivými updaty.

HLSL

HLSL (HLSL, b.r.) je zkratka pro High Level Shading Language. Je to programovací jazyk pro programování způsobu zpracování grafických dat na grafické jednotce v reálném čase pro DirectX. Jeho syntaxe, funkce, typy a další se velmi podobají konstrukci jazyka C. Poprvé byl představen pro DirectX 9 a je podporován pouze platformou Microsoft Windows.

Data vstupují do pipeline jako proud grafických elementů, vertexů, které jsou dále zpracovány různými stupni shaderu. Prvním stupněm je tedy input assembler, který slouží jako vstup dat ke zpracování. Dostupnost některých stupňů závisí na verzi DirectX, ale

základními plně programovatelnými následujícími stupni jsou vždy pixel a vertex shader. Dále od verze DirectX 10 může obsahovat i geometry shader, který se stará o transformaci skupin vertexů jako celku. Posledním stupněm je output merger, jehož úkolem je z dostupných zpracovaných dat sestavit finální obraz, který bude vykreslován na obrazovku. (Jones, 2008)

Verze Direct3D	Verze shader model						
9	1	2	3				
10	1	2	3	4			
11	1	2	3	4	5		
11.3	1	2	3	4	5	5.1	
12	1	2	3	4	5	5.1	6

Tabulka 1 - verze shader modelů

```
#if OPENGL
   #define SV_POSITION POSITION
   #define VS_SHADERMODEL vs_3_0
   #define PS_SHADERMODEL ps_3_0
#else
   #define VS_SHADERMODEL vs_4_0_level_9_1
   #define PS_SHADERMODEL ps_4_0_level_9_1
#endif

matrix WorldViewProjection;

struct VertexShaderInput
{
   float4 Position : POSITION0;
   float4 Color : COLOR0;
};

struct VertexShaderOutput
{
   float4 Position : SV_POSITION;
   float4 Color : COLOR0;
};

VertexShaderOutput MainVS(in VertexShaderInput input)
{
   VertexShaderOutput output = (VertexShaderOutput)0;

   output.Position = mul(input.Position, WorldViewProjection);
   output.Color = input.Color;

   return output;
}

float4 MainPS(VertexShaderOutput input) : COLOR
{
   return input.Color;
}
```

```
technique BasicColorDrawing
{
    pass P0
    {
        VertexShader = compile VS_SHADERMODEL MainVS();
        PixelShader = compile PS_SHADERMODEL MainPS();
    }
};
```

Kód 2 - .fx soubor generovaný monogame pipeline managerem

Výstupem výše uvedeného Kódu 2, je stejná barva pixelu na výstupu jako na vstupu. Shader zde obsahuje dvě struktury, vstup a výstup vertex shaderu. Jako vstup, který představuje struktura *VertexShaderInput* jsou zde přítomny dvě proměnné, tak jak vstupují so shaderu vertexy se dvěma vlastnostmi, barva a pozice. Při zpracování 2D grafiky lze vertex shader opomenout a jako vstup pixel shaderu bude sloužit pouze proměnná typu float2 *TEXCOORD0*, která představuje souřadnice pixelu. Převedeme kód na kompaktnější formu. V obou případech je ale zásadní, aby pixel shader vracel hodnotu typu float4, zde představovanou *COLOR0*. Klíčová slova jako *COLOR0* nebo *TEXCOORD0* se nazývají sémantika a označují jakým způsobem má shader optimalizovat práci s proměnnou a na co bude proměnná použita.

```
sampler s0;

float4 PixelShaderFunkce(float2 souřadnice : TEXCOORD0) : COLOR0
{
    float4 barva = tex2D(s0, souřadnice);

    return barva;
}

technique Technika1
{
    pass Průchod
    {
        PixelShader = compile ps_1_0 PixelShaderFunkce();
    }
}
```

Kód 3 - zjednodušený shader

Shader musí obsahovat minimálně jednu techniku s jedním průchodem a musí specifikovat verzi pixel a v předchozím případě i vertex shaderu. *tex2D* je věstavěná funkce.

1.1.4 HLSL Proměnné

Základními typy hodnot mimo *float*, double a *int* jsou zabalené skupiny hodnot jako *float2*, *float4*, a matice *float2x2*, *float3x3* atd. Díky specializovanému hardwaru počítače je práce s proměnnými dobře optimalizovaná pro architekturu shaderu. (Jones, 2008)

Ke skupinám hodnot typů *float2* až *float4* můžeme přistupovat přes indexy *.rgba* nebo *.xyzw*, které jsou vzájemně zaměnitelné. Podobně se bude přistupovat i k maticím.

1.1.5 Funkce

V jazyce HLSL můžeme definovat bloky kódu, funkce, a psát tak znovupoužitelný kód. Funkce je definována podobně jako metoda jazyka C#. Jako parametr vstupuje do funkce *float4* barva a funkce vrací červenou barvu.

```
float4 Červená(float4 barva : COLOR0)
{
    return float4(1, 0, 0, 1);
}
```

Kód 4 - funkce v HLSL

HLSL také obsahuje již vestavěné funkce, angl. intristics, jako je výpočet absolutní hodnoty *abs*, funkce *sin*, *cos* a jiné.

Fyzika ve hře

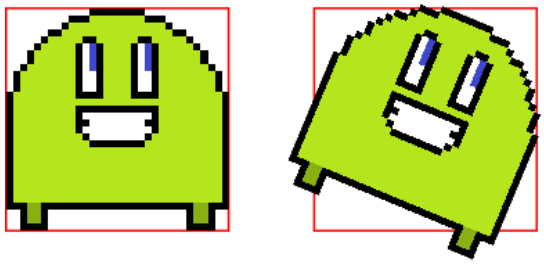
1.1.6 Teorie

Každé těleso v reálném světě je prezentováno svým tvarem, hmotou, hybností a dalšími faktory. Simulovat komplexní tělesa se všemi fyzikálními vlastnostmi, které ho tvoří, nebo silami, které na něj působí však není ve hrách žádoucí. Na tyto výpočty není v reálném čase dostatek prostoru a ani jejich prezence ve hře není ve většině případů ani vyžadována. Simulace fyzikálně přesných modelů probíhá na velkých superpočítačích a řešení komplexních problémů může zabrat i dny. V herním průmyslu, kde počítače dosahují pouze zlomku výkonu si musíme vystačit s velmi zjednodušenými modely.

Můžeme opomenout deformace některých objektů, nebo některé objekty učinit statické, neměnné.

1.1.7 AABB

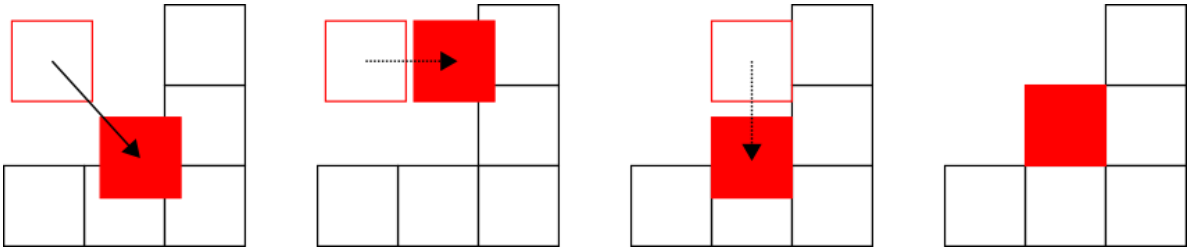
Neboli axis-aligned bounding box, česky možno přeložit jako objekt, kvádr nebo i obdélník, který je orientován rovnoběžně s osami. Tento objekt nemůže rotovat, ale může měnit svoji velikost. Je to zjednodušená prezentace komplexnějšího objektu. Může být užitečná jako detekce potencionálních kolizí, protože šetří výpočetním výkonem.
Ve dvojrozměrné hře však může AABB představovat již samotný fyzikální objekt. S takto představovanými objekty se fyzikální model ve hře velmi zjednoduší. Na rozdíl od detekce potencionálních kolizí, kde musí bounding box obsahovat celý objekt, se při použití jako fyzikálního tělesa ve hře nemusí shodovat s grafickým prvkem.



Obrázek 2 - bounding box ve dvojrozměrné hře

1.1.8 Detekce kolize a odezva

Jak již bylo víše zmíněno, v plošinové hře slouží bounding box současně i pro detekci kolizí, primárně protože se tento objekt nedá vyjádřit jednodušeji. Základem algoritmu bude detekce a řešení kolize po osách. Pohyb bude rozdělen na vodorovný a vertikální. Pro první osu aplikujeme pohyb, zjistíme kolize a posuneme objekt. Pak se provedeme totéž pro druhou osu. Na pořadí os nezáleží. Toto je osvědčený postup, při řešení obou os zároveň anebo kolize postupně s každým jednotlivým objektem se můžeme dostat do problémů, jako je zasekávání na rozích a jiné.



Obrázek 3 - zpracování vícenásobných kolizí

Vlastní práce

Analýza

Vlastní program a jeho samotná realizace na platformě Windows může následovat více cest. Samotná hra jako celek může být renderována v různých rozlišeních anebo využívat tzv. resolution independence (nezávislost na rozlišení). Program je takto možno zvětšovat v okně a udržet tak stejný poměr stran. Není vhodné upravovat velikost okna nebo bránit jeho zvětšení či zmenšení určitým způsobem. Hra je proto vykreslována s černými okraji tam kde okno přesahuje velikost vykreslované plochy.

Platformové hry často pracují s platformami jako bloky zarovnanými do mřížky. U některých her se však můžeme setkat i s řešením ve formě polygonů. Takto založené hry využívají často fyzikální engine třetích stran, jako je například VelcroPhysics. (GitHub VelcroPhysics, 2018)

Implementovat vlastní řešení fyziky založené na různorodých tvarech není nutné, protože tato řešení jsou si často velmi podobná a mají i podobnou optimalizaci. Vlastní řešení využijeme pouze u speciálních případů nebo z licenčních důvodů. Nutno podotknout, že VelcroPhysics je open-source řešení.

Vlastní řešení využijeme především u jednodušších enginů jako jsou tile-based neboli už zmíněná verze s bloky pouze ve mřížce se stejnou velikostí buněk anebo a řešení s AABB objekty jako jsou obdélníky, čtverce a kruhy. V této práci se zaměříme na vytvoření vlastního fyzikálního enginu s využitím AABB bounding boxů.

Dalším problémem však může být optimalizace detekce kolizí s terénem. Procházet všechny objekty nebo buňky mapy není žádoucí v žádné hře. Testujeme pouze terén blízký hráčovi anebo NPC postavám a aktuální fyzikálním tělesům. Okolí hráče můžeme buďto testovat jako výseč z pole, nebo využít struktury jako je quad tree.

Hráč může být realizován rozmanitými způsoby. Ve hrách se často objevů prvky těles, které představují deformující se blob, anebo měnící svůj tvar a vlastnosti v závislosti na prostředí. Dále může být hry zaměřeno na střílení, překonávání překážek a mnoho dalších cílů. Stejně tak se dají prezentovat NPC postavy a nejen postavy.

Především u NPC a různých objektů představující fyzikální tělesa anebo podobné NPC podobné třídy anebo s podobnými základy je vhodné, aby tyto třídy dědily z předků,

které skupiny těchto vlastností zaštiťují. NPC je tedy třídou která bude dědit z předka společného pro většinu NPC. Bude obsahovat životy a jiné. Tento předek může být prezentován jako potomek fyzikálního tělesa. Dědičnost se uplatí u každé třídy včetně zbraní, bonusů a jiné. Celkové využití dědičnosti je ve hře velmi využitelné.

Návrh

1.1.9 Hra jako celek

1.1.9.1 Hra

Hra má obsahovat hráče, který se bude pohybovat ve dvojrozměrném prostoru po platformách. Bude na něj působit jednoduchá fyzika prostředí. Bude moci skákat a chodit. Ovládání hráče bude probíhat skrze klávesnici pro pohyb a myší pro míření a střelbu. Hra bude obsahovat vozidla jako je robot a nositelné interaktivní objekty jako je střílející věžička a baterii do spínače. Dále bude obsahovat výtah jehož trasa bude definována uzly a dveře. Klávesy budou moci být nastaveny v menu hry. Hra bude obsahovat i zvuky.

Hra bude obsahovat nákupní místa a místa na skládání nakoupených komponent do mřížky pro získání bonusů. Komponentami půjde rotovat. Dalo by se říci, že se jedná o variaci na puzzle s „kostiček“ ze hry tetris.

1.1.9.2 Fyzika

Hra bude obsahovat statické objekty a také jednoduché kinematické, se kterými budou ostatní fyzikální tělesa představující hráče nebo bonusy reagovat a kolidovat. Při stlačení hráče, nebo některého z NPC postav bude tato entita „zabita“ a přestane s prostředím vzájemně reagovat. Především u výtahů nebudou implementovány události jako zaseknutí výtahu o jinou fyzikální entitu. Volná entita bude prostě deaktivována nebo přestane existovat. Tímto se zabrání nežádoucímu chování objektů.

Jelikož u této hry není žádoucí, aby dynamické objekty interagovali spolu navzájem a také, protože vytvořit komplexnější fyzikální engine je rozsáhlý problém sám o sobě viz. (Millington, 2007), bude stačit, když každý fyzikální prvek hry bude naimplementován jako AABB, kde ke se statickým i kinematickým objektům bude přistupovat stejně. Statický objekt tedy bude brán podobně jako kinematický, ale nebude mít žádnou rychlost.

Dalším problémem je tzv. bulleting, kde rychle pohybující se objekt pronikne v jednom updatu přes překážku. Tento problém se dá vyřešit jednoduchou iterací, která spočívá k vydělení rychlosti v jednom updatu a iterací postupně posouvat objekt po malých krocích, detekovat kolize a přímo na ně i reagovat. Rychlost by měla být snížena na polovinu šířky aabb, v ose, kde je objekt tenčí. Výsledná reálná rychlost se tedy nezmění.

1.1.9.3 UI a menu

Hra jako celek bude obsahovat jednoduché UI ve formě několika základních tříd prvků, jako jsou:

- Tlačítka
- Text boxy
- Popisky
- Drop-down menu

Pro vytváření prvků bude vytvořen vlastní kód. Prvky nebudou uloženy ve struktuře, která by spolehlivě řešila jejich překrývání, neboť se to v našem případě nevyužije. Pro UI prvky bude vytvořen jednotný obrázkový font. Ten by měl obsahovat základní znaky abecedy, číslice a znak představující chybějící znak. Bude využit UI prvky v menu hry. Ve hře samotné bude využit samostatný font obsahující pouze čísla.

Tlačítka budou obsahovat seznam metod, které se zavolají při jejich stisku. Budou reagovat stejně jako v OS Microsoft Windows. To znamená že tlačítko se aktivuje až při uvolnění tlačítka myši. Stejně tak bude využita u ostatních prvků. Tlačítko samotné může být využito i v například drop-down menu.

Textboxy budou umožňovat zadávat hodnoty do pole. Barva textu nebo pozadí se změní, pokud bude zadána nepřípustná hodnota. Bude umožnovat i text delší, než je šířka pole. Tímto textem se pak budeme orientovat šipkami. Text bude vkládán na kurzoru.

U popisků bude možnost zarovnání doleva a na střed.

V případě drop-down menu by měly být prvky na obrazovce řazeny pod sebou, tak aby se problém překrývání lépe řešil a nevyžadoval složitější přístup. Bude pracovat s enumerací, jejíž slovní prvky zobrazí jako možnosti.

Instance obrazovek menu budou složeny především z těchto prvků a uloženy do slovníku.

1.1.10 Správa obrazovek a UI

Obrazovky budou naimplementovány jako třídy, obsahující prvky UI, jako jsou tlačítka a popisky. Je vhodné jejich instance ukládat ve slovníku a deklarovat jednu statickou proměnnou, která bude držet referenci na právě aktivní obrazovku. Obrazovky se budou měnit podle toho, jaká reference je udržována v proměnné.

Vlastní prvky UI jako tlačítka by měla na mít pro vykonání svojí akce sadu metod.

1.1.11 Nahrávání obsahu

K nahrávání obsahu v Monogame primárně slouží content manager. Ten zkonvertuje soubory do binární podoby s příponou .xnb.

Data z obrázků, zvuků anebo i fontů a dalších zdrojů, je dobré definovat jako statické a přístupné z jedné třídy pro všechny ostatní. Odpadají tím problémy s definováním vlastního obsahu pro každou třídu. Vůbec není vhodné definovat tentýž obsah pro každý objekt znovu. I takto však dochází k opakování kódu a prodlužování doby práce. Proto bude seznam položek souborů uveden v jednom JSON souboru a nahrán do asociativní kolekce. V kódu se pak bude dát přistupovat ke každé položce obsahu přes klíč.

1.1.12 HLSL efekty

Shader bude naimplementován v HLSL i přes zaměření aplikace na OpenGL. Kód je poté zkonvertován do podoby, kterou může OpenGL využívat. Každý efekt bude aplikován v příslušné statické metodě s parametry.

1.1.13 ExtensionMethods

Neboli rozšiřující metody rozšiřují neinvazivním způsobem již existující typy. K těmto typům přidávají vlastní metody. Klíčové slovo *this* aplikuje metodu na určitou proměnnou přes tečkovou notaci, kde vstupním parametrem je zde přímo proměnná.

1.1.13.1 Práce s úhly

```
public static double RadianToDegree(this double angle)
{
  return angle * (180.0 / Math.PI);
}

public static double AngleDifference(double firstAngle, double secondAngle)
{
  double deg1 = firstAngle.RadianToDegree();
  double deg2 = secondAngle.RadianToDegree();

  double difference = deg2 - deg1;
  while (difference < -180) difference += 360;
  while (difference > 180) difference -= 360;
  return difference;
}
```

Kód 5 - radiány na stupně

Statická metoda rozšíření *RadianToDegree* převádí hodnotu uloženou v typu double z radiánů na stupně. Další statickou metodou je metoda *AngleDifference*, která vrací nejmenší úhel mezi dvěma hodnotami *firstAngle* a *secondAngle*.

Metoda *ToAngle* převádí směrový vektor na úhel.

```
public static float ToAngle(this Vector2 vector)
{
  return (float)Math.Atan2(vector.Y, vector.X);
}
```

Kód 6 – 2D vektor na úhel

1.1.13.2 Metoda Clamp

Tato metoda ořezává hodnoty a vrací upravený výsledek. Metoda *Clamp* je generická metoda, přijímající pouze parametry implementující rozhraní *IComparable*. Vstupní

hodnota je zde porovnávána s parametry min a max. Pokud překračuje jednu z těchto hodnot, je dalo by se říci oříznuta a vrácena je příslušná hodnota *min* nebo *max*, jako maximální možná.

```
public static T Clamp<T>(this T val, T min, T max) where T : IComparable<T>
{
  if (val.CompareTo(min) < 0) return min;
  else if (val.CompareTo(max) > 0) return max;
  else return val;
}
```

Kód 7 - ořez hodnot

1.1.13.3 Práce s dvojrozměrnými vektory

Rozšiřující metoda *Rotate* Otáčí směrový vektor o určitý úhel v radiánech.

```
public static Vector2 Rotate(this Vector2 vector, float angle)
{
  double x = Math.Cos(angle) * vector.X - Math.Sin(angle) * vector.Y;
  double y = Math.Sin(angle) * vector.X + Math.Cos(angle) * vector.Y;

  return new Vector2((float)x, (float)y);
}
```

Kód 8 - rotace 2D vektoru

ToUnitVector2 jsou metody, podobné vestavěným metodám třídy *Vector2*, které jsou však volány přes název typu. Lepší použitelnost i v řetězení metod se zde docílí klíčovím slovem this.

```
public static Vector2 ToUnitVector2(this float angle)
{
  return new Vector2((float)Math.Cos(angle), (float)Math.Sin(angle));
}

public static Vector2 NormalizeCustom(this Vector2 vector)
{
  return Vector2.Normalize(vector);
}

public static Vector2 ShiftOverDistance(this Vector2 vector, float distance, float rotation)
{
  return vector + (rotation.ToUnitVector2() * distance);
}
```

Kód 9 - 2D vektor na jednotkový

ShiftOverDistance posune směrový vektor, který zde bereme jako pozici, i když stále jako posunutí v prostoru. Směr je určen parametrem *rotation*, což je újhel v radiánech a parametrem *distance*, vzdáleností.

1.1.13.4 Práce s řetězci

využití metody *Cut* je především v textBoxech. Tato metoda má za úkol odebrat počet znaků ze začátku řetězce definovaný parametrem *min* a z jeho konce parametrem *max*.

```
public static string Cut(this string s, uint min, uint max)
{
   if (min >= max)
      throw new ArgumentOutOfRangeException();

   if (min > s.Length)
      throw new ArgumentOutOfRangeException();

   string temp = "";

   if (max < s.Length)
   {
      for (uint i = min; i < max; i++)
      {
         temp += s[(int)i];
      }

      return temp;
   }
   else
   {
      for (uint i = min; i < s.Length; i++)
      {
         temp += s[(int)i];
      }

      return temp;
   }
}
```

Kód 10 - metoda ořezu řetězce

1.1.14 **HLSL efekty**

1.1.14.1 Gaussovo rozostření

Implementace lineárního Gaussova rozostření je rozdělena na horizontální a vertikální průchod. Využívá se tak vlastnost tohoto algoritmu, nemusíme tedy vynakládat úsilí na procházení bodů ve dvojrozměrném poli.

```
float4 HorizontalBlur(float2 coords: TEXCOORD0) : COLOR0
{
   float4 color = tex2D(s0, coords);
   float4 mask = tex2D(MaskSampler, coords);
   float4 temp = float4(0, 0, 0, 0);
   float divide = 0;

   [unroll(64)]
   for (int i = -kernelSize/2; i <= kernelSize/2; i++)
   {
      float4 TextureColors = tex2D(TextureInSampler,
      float2(coords.x + (1.00 / dimensions.x) * i, coords.y));

      if ((coords.x + (1.0 / dimensions.x) * i) >=
         0.0 && (coords.x + (1.0 / dimensions.x) * i) <= 1.0)
      {
         temp += TextureColors * ((kernelSize / 2 + 1) * 2 - abs(i) * 2);
         divide += ((kernelSize/2 + 1)*2 - abs(i)*2);
      }
   }

   return float4(color + temp / divide);
}
```

Kód 11 - horizontální průchod Gaussova rozostření

Do shaderu vstupuje textura s názvem *Mask*, která je dále využita ve funkci *tex2D* pro proměnnou float4 *mask*. Tato textura bude obarvovat výsledný výstup podle zabarvení vstupní textury v druhém průchodu. *kernelSize* je celková šířka záběru jednoho průchodu, tedy celkový počet pixelů, ze kterého budeme vypočítávat průměr. Čím vyšší hodnota, tím bude výsledné rozostření silnější. Je vhodné podotknout že shader pracuje se souřadnicemi jako s desetinnými čísli od 0 do 1. Proto musí do shaderu vstupovat i velikost zpracovávané textury, aby byly správně převedeny souřadnice. Příkaz *unroll* rozvine for cyklus až do 64 iterací a tím zrychlí průběh výpočtu, za cenu zvětšení binární velikosti kódu. Výsledná maximální počet bodů, ze kterých se vypočte rozostření tedy bude roven 64. Téměř totožný kód bude následovat i pro druhý průchod.

```
public static void GaussHorizontalEffect(Texture2D texture, Texture2D mask)
{
   Game1.EffectBlur.Parameters["dimensions"]
   .SetValue(Globals.WinRenderSize);
   Game1.EffectBlur.Parameters["Mask"].SetValue(mask);
   Game1.EffectBlur.Parameters["TextureIn"].SetValue(texture);
   Game1.EffectBlur.Parameters["kernelSize"].SetValue(16);
   Game1.EffectBlur.CurrentTechnique.Passes[0].Apply();
}
```

Kód 12 - metoda použití horizontálního rozostření

Do statické metody *GaussHorizontalEffect* ve statické třídě *Effects* vstupují jako parametry dvě textury. Textura *texture*, která bude zpracována a *mask*, která slouží pro výsledné obravení.

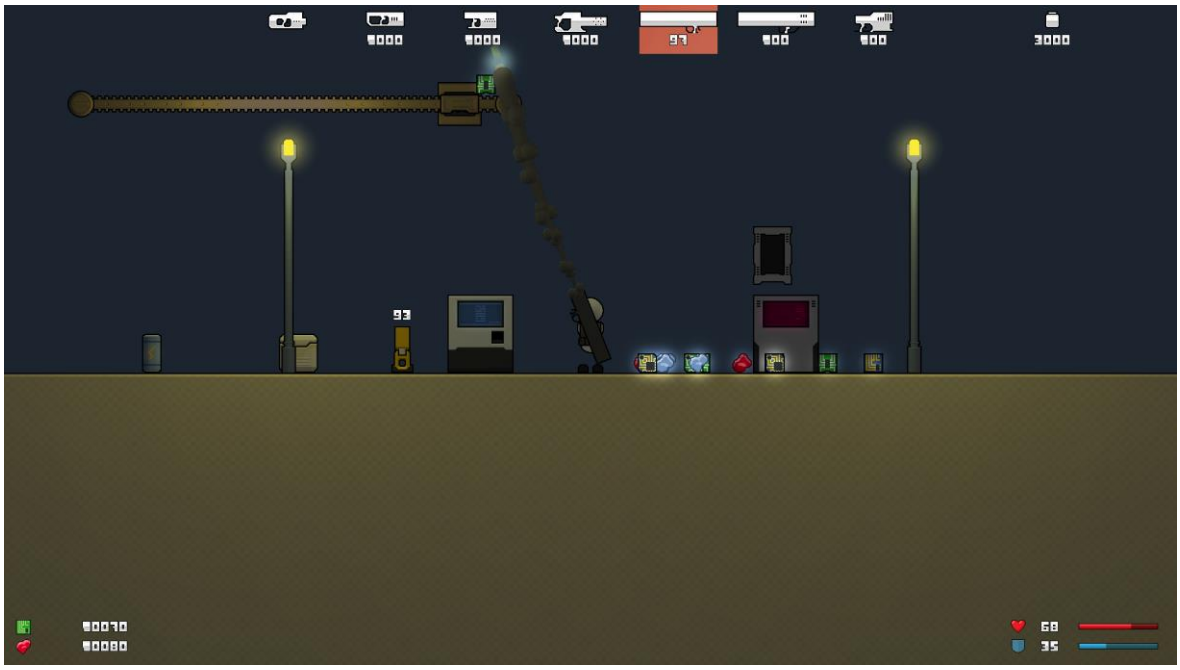
1.1.14.2 Rotace normálových map

K normálové mapě můžeme přistupovat jako poli trojrozměrných vektorů, které jsou dány intenzitami červené, zelené a modré. Barvy jsou HLSL prezentovány desetinnými čísly od 0 do 1. Abychom dostali vektor pro každou osu, který míří do všech směrů, musíme odečíst od červené a zelené 0.5f a poté mohli vektory násobit maticí *RotationMatrix* funkcí *mul*. Do funkce *RotationMatrix* vstupuje proměnná *rot*, která představuje úhel v radiánech. Poté musíme znovu přičíst 0.5f k červené a zelené. Díky této operaci získáme texturu opět se všemi barvami v rozmezí 0 až 1.

```
float2 flip;

float2x2 RotationMatrix(float rot)
{
    float c = cos(rot);
    float s = sin(rot);

    return float2x2(c, -s, s, c);
}

float angle;

float4 PixelShaderFunction(float2 coords : TEXCOORD0) : COLOR0
{
    float4 color = tex2D(s0, coords);
    float3 normalized = (color.rgb - float3(0.5f, 0.5f, 0))
    * float3(flip.x, flip.y, 1);
    normalized.rg = mul(normalized.rg, RotationMatrix(angle))
    + float2(0.5f, 0.5f);

    return float4(normalized.rgb, color.a);
}
```

Kód 13 - rotace normálové mapy

1.1.14.3 Výpočet osvětlení



Obrázek 4 - osvětlení ve hře

Do shaderu vstupují proměné *light* pro pozici světla, *lightColor* pro jeho barvu. Především stojí za zmínku *float3 normalized*, kde se jako u rotace normálové mapy musí barva převést na vektor, který míří do všech směrů. Dále proměnná *output*, která vypočítá skalární součin vektoru směru světla k aktuálnímu pixelu a normálového vektoru.

```
float4 PixelShaderFunction(float2 coords : TEXCOORD0) : COLOR0
{
   float4 color = tex2D(s0, coords);
   float4 normalColor = tex2D(NormalMap, coords);
   float2 pixelPos = float2((coords.x) *
   size.x + positionTex.x, (coords.y) * size.y + positionTex.y);

   float3 normalized = (normalColor.rgb - float3(0.5, 0.5, 0)) * float3(1, -1, 1);
   float dist = 1 / (distance(light, pixelPos) + 32);
   float DistanceLoss = (distance(light, pixelPos) + 128) / 128;
   float output = dot(float3(light.x - pixelPos.x,
   light.y - pixelPos.y, 128), normalized);

   float3 calculated = float3((color.r * output) * dist,
   (color.g * output) * dist, (color.b * output) * dist);

   float3 temp = float3(calculated.r / DistanceLoss,
   calculated.g / DistanceLoss, calculated.b / DistanceLoss) * lightColor.rgb;

   return float4(temp.rgb, normalColor.a);
}
```

Kód 14 - funkce osvětlení

1.1.15 **Práce se soubory**

1.1.15.1 Nahrávání Json souborů

Metoda nahrávání json souborů využívá *JsonConverter*, který deserializuje textová data ze souboru JSON do objektu určitého typu *T*. Pokud je vyvolána výjimka, metoda vrátí hodnotu *false*. *StreamReader* je uvozen do bloku *using*. Po jeho opuštění jsou uvolněny příslušné prostředky.

```
public static bool LoadJson<T>(ref T type, string link)
{
  string textLinks = "";
  using (StreamReader sr = new StreamReader(link))
  {
    textLinks = sr.ReadToEnd();
  }
 try
  {
    type = JsonConvert.DeserializeObject<T>(textLinks);
  }
  catch (JsonReaderException ex) { return false; }

  return true;
}
```

Kód 15 - nahrání JSON souboru

1.1.15.2 Ukládání JSON souborů

Podobně se bude postupovat i pro nahrávání obsahu

```
public static bool SaveJson<T>(T type, string link, string name)
{
  using (StreamWriter file =
    File.CreateText(Path.Combine(@link, name + ".json")))
  {
    JsonSerializer serializer = new JsonSerializer();
    serializer.Formatting = Formatting.None;

    try
    {
      serializer.Serialize(file, type);
    }
    catch (JsonWriterException ex) { return false; }

    return true;
  }
}
```

Kód 16 - uložení JSON souboru

1.1.16 **Geometrie**

1.1.16.1 CircleF

Tato třída představuje jednoduchý kruh Definovaný proměnnými *Radius* a *Center*. Tato třída obsahuje dvě metody. Obě tyto metody se používají v dalších vizuálních efektech. Metoda *DidvideCircle* rozdělí Kruh na stejně velké segmenty neboli vytvoří List Bodů *Vector2*, hrubě definující tvar kruhu. Přijímá parametr *by*. Tímto číslem se celý obvod rozdělí.

```
public List<Vector2> DidvideCircle(int by)
{
   List<Vector2> temp = new List<Vector2>();

   double angle;
 for (int i = 0; i < by; i++)
   {
      angle = i * ((Math.PI * 2) / by);
      Vector2 vector = new Vector2();
      vector.X = (float)(Center.X + Radius * Math.Cos(angle));
      vector.Y = (float)(Center.Y + Radius * Math.Sin(angle));
      temp.Add(vector);
   }

   return temp;

}
```

Kód 17 - rozdělení kruhu na body

Metoda *GenerateRandomPoint* vygeneruje náhodný bod v kruhu. Při generování za použití náhodného úhlu a vzdálenosti od středu by se počet bodů blíže ke středu stupňoval. Proto je použit tento přístup.

```
public Vector2 GenerateRandomPoint()
{
   double radius = Math.Sqrt(Globals.GlobalRandom.NextDouble()) * Radius;
   double angle = Globals.GlobalRandom.NextDouble() * Math.PI * 2;
   double x = radius * Math.Cos(angle);
   double y = radius * Math.Sin(angle);

   return new Vector2((float)x, (float)y) + Center;
}
```

Kód 18 - vytvoření náhodného bodu v kruhu

1.1.16.2 RectangleF

Protože již existují třída *Rectangle* je nedostačující, z důvodu implementace pozice v celých číslech, naimplementována byla třída *RectagleF*. Ta bude ulehčovat práci s fyzikálními objekty, protože rychlost je také definována také v desetinných číslech.

Třída bude obsahovat podobně jako *Rectangle* property typu *Position, Size, Origin* a další. Další důležitou součástí třídy je metoda *IntersectionSize*, která vrací velikost průniku dvou objektů třídy *RectangleF*. Toho bude využito ve třídě *CollisionResolver*, při řešení kolizí.

```
public Vector2 IntersectionSize(RectangleF R)
{
  if (CompareF.RectangleFVsRectangleF(this, R) == true)
  {
    float TempX = (Size.X + R.Size.X)
      / 2 - Math.Abs(Origin.X - R.Origin.X);
    float TempY = (Size.Y + R.Size.Y)
      / 2 - Math.Abs(Origin.Y - R.Origin.Y);

    return new Vector2(TempX, TempY);
  }
  return Vector2.Zero;
}
```

Kód 19 - velikost průniku dvou objektů RectangleF

1.1.16.3 LineSegmentF

Úsečka definovaná dvěma koncovými dvojrozměrnými vektory. Bude především využita u projektilů a při míření zbraní. Bude obsahovat několik pomocných metod.

Metody *Lenght* vrací délky úsečky, u druhého přetížení vzdálenost mezi dvěma body.

```
public float Lenght()
{
  return (float)Math.Sqrt(Math.Pow(End.X - Start.X, 2)
    + Math.Pow(End.Y - Start.Y, 2));
}

public static float Lenght(Vector2 start, Vector2 end)
{
  return (float)Math.Sqrt(Math.Pow(end.X - start.X, 2)
    + Math.Pow(end.Y - start.Y, 2));
}


public Vector2 NormalizedWithZeroSolution()
{
  LineSegmentF segment = new LineSegmentF(Start, End);
  Vector2 line_to_vector = segment.ToVector2();
```

```
    if (Math.Abs(line_to_vector.X) > 0 || Math.Abs(line_to_vector.Y) > 0)
        return Vector2.Normalize(line_to_vector);
    else return Vector2.UnitY;
}

public Vector2 ToVector2()
{
    LineSegmentF segment = new LineSegmentF(Start, End);
    return new Vector2(segment.End.X - segment.Start.X,
        segment.End.Y - segment.Start.Y);
}

public float ToAngle()
{
    LineSegmentF segment = new LineSegmentF(Start, End);
    return segment.ToVector2().ToAngle();
}
```

Kód 20 - metody třídy LineSegmentF

Metoda *NormalizedWithZeroSolution* vrací jednotkový vektor. Pokud má přímka nulovou délku, to znamená, pokud se začátek a konec kryjí, vrací *Vector2.UnitY*. Při použití klasické metody *normalize* třídy *Vector2* by došlo k chybě, protože z nulového vektoru nemůže být jednotkový vypočten.

Metoda *ToAngle* převede úsečku na úhel v radiánech. Na řádku za klíčovým slovem *return* jde vidět řetězení metod, kdy každá metoda vrací proměnnou a ta je použita jako vstup další.

1.1.16.4 QuadTree

Jedná se o třídu, která dělí prostor na 4 stejně velké sektory. V tomto případě má každý rodič 4 potomky. Pouze v úrovni kořene je definováno pole *Trees*, aby nečtvercová herní mapa byla efektivněji pokryta.

Pro velikosti čtverců je definována enumerace *QuadTreeSizes*, která přehledně popisuje úrovně zanoření.

```
1.  public enum QuadTreeSizes { Size_1024, Size_512, Size_256, Size_128, Size_64, Size_32 };
```

třída *MapTreeHolder* tedy obsahuje definici pole *QuadTree* objektů. Pro použití takto definované třídy tedy používáme především *mapTreeHolder*.

```
public MapTreeHolder(int x, int y)
    {
      Trees = new QuadTree[(int)(Math.Ceiling(x / 1024f)),
        (int)(Math.Ceiling(y / 1024f))];

      for (int Y = 0; Y < Trees.GetLength(1); Y++)
      {
        for (int X = 0; X < Trees.GetLength(0); X++)
        {
          Trees[X, Y] = new QuadTree(X * 1024, Y * 1024,
            QuadTreeSizes.Size_1024);
        }
      }
    }

    public void InsertRec(RectangleF rec)
    {
      if (CompareF.RectangleFVsRectangleF(Boundary, rec) == true)
      {
        if (LevelSize < QuadTreeSizes.Size_32)
        {
          if (HasChilds == false)
          {
            HasChilds = true;
            LeftTop = new QuadTree(Boundary.Position.X,
              Boundary.Position.Y, LevelSize + 1);
            RightTop = new QuadTree(Boundary.Position.X
              + (LevelSizeInt / 2), Boundary.Position.Y,
              LevelSize + 1);
            LeftBottom = new QuadTree(Boundary.Position.X,
              Boundary.Position.Y + (LevelSizeInt / 2),
              LevelSize + 1);
            RightBottom = new QuadTree(Boundary.Position.X
              + (LevelSizeInt / 2), Boundary.Position.Y
              + (LevelSizeInt / 2),
              LevelSize + 1);
          }
          LeftTop.InsertRec(rec);
          RightTop.InsertRec(rec);
          LeftBottom.InsertRec(rec);
          RightBottom.InsertRec(rec);
        }
        else
        {
          Holder = rec;
        }
      }
    }
```

Kód 21 - MapTreeHolder a vkládání RectangleF objektů

V Metodě *InsertRec* dojde k porovnání průniku *Boundary* a vkládaného objektu typu *RectangleF* do quad tree struktury. Pokud nemá větev potomky a není nejnižžší úrovně, je pro propery *LeftTop* a další tři rohy vytvořena nová úroveň. Poté je i pro tyto potomky zavolána metoda *InsertRec* a takto se rekurzivně postupuje až na nejnižší úroveň.

1.1.17 **Fyzika**

1.1.17.1 CollisionResolver

```
int iterations = (int)Math.Ceiling(Math.Max(Math.Abs(velocity.X),
  Math.Abs(velocity.Y))
  / Math.Min(boundary.Size.X, boundary.Size.Y)) * 4;
if (iterations == 0) iterations = 1;

for (int i = 0; i < iterations; i++)
{
  boundary.Position += new Vector2(0, velocity.Y.Clamp
    (-maxSpeed.Y, maxSpeed.Y) / iterations * Game1.Delta);

  RecsToCheckMap = Compare.RectangleVsMap(_map.MapTree, boundary);

  ComparableRecs.Clear();
  ComparableRecs.AddRange(RecsToCheckMap);

  if (collection != null)
  {
    List<IrecComparable> temp = new List<IrecComparable>();

    foreach (var item in collection)
    {
      if (Compare.RectangleFVsRectangleF
        (item.Boundary.Rectangle, boundary) == true)
      {
        temp.Add(item);
      }
    }

    ComparableRecs.AddRange(temp);
  } if (HorizontalPressure == false)
  {
    maxVelocityClamped = new Vector2(velocity.X.Clamp
      (-maxSpeed.X, maxSpeed.X), velocity.Y.Clamp
      (-maxSpeed.Y, maxSpeed.Y));

    ResolveWithMapYPlus(ref boundary, ref velocity, gravity,
      iterations, friction, bouncines, walking);

    maxVelocityClamped = new Vector2(velocity.X.Clamp
      (-maxSpeed.X, maxSpeed.X), velocity.Y.Clamp
      (-maxSpeed.Y, maxSpeed.Y));

    ResolveWithMapYMinus(ref boundary, ref velocity, gravity,
      iterations, friction, bouncines);
  }
```

Kód 22 – obsah třídy řešení kolizí

ResolveWithMapYPlus je jedna ze čtyř podobných metod, každá pro jeden směr. V této metodě procházíme seznamem objektů, se kterými dochází ke kolizi. Tyto objekty musí implementovat rozhraní *IrecComparable*. Pokaždý rec se poté otestuje z jaké strany došlo ke kolizi dříve. Jestli na ose x nebo y. Pokud je splněna tato podmínka, testuje se

dále, zda-li zda li bod minY opravdu představuje hranu objektu rec, která je na ose y nejvíce v záporu. Pokud není, ale je splněna předchozí podmínka, našli jsme nový objekt. Z tohoto objektu poté převezmeme jeho rychlost *rec.Velocity*. Zvýšíme také počet detekovaných objektů o jednu. Tímto procesem vybereme správný bod posunutí a rychlost které použijeme dále.

```
private void ResolveWithMapYPlus(ref RectangleF boundary, ref Vector2 velocity,
        float gravity,int iterations, Vector2 friction, float bouncines, bool walking)
    {
        int count = 0;

        float minY = float.MaxValue;

        foreach (IrecComparable rec in ComparableRecs)
        {
            if (boundary.OriginPos.Y < rec.Boundary.Rectangle.OriginPos.Y
&& rec.Velocity.Y < Math.Max(velocity.Y,velocityReceivedY))
            {
                if ((1+Math.Abs(maxVelocityClamped.X + rec.Boundary.Velocity.X))
                    / boundary.IntersectionSize(rec.Boundary.Rectangle).X
                    < (1+Math.Abs(Math.Max(maxVelocityClamped.Y, velocityReceivedY )
                    + rec.Boundary.Velocity.Y))
/ boundary.IntersectionSize(rec.Boundary.Rectangle).Y)
                {
                    if (minY > rec.Boundary.Rectangle.CornerLeftTop.Y)
                    {
                        minY = rec.Boundary.Rectangle.CornerLeftTop.Y;
                        count++;
                        velocityReceivedX = rec.Velocity.X;
                        velocityReceivedY = rec.Velocity.Y;
                    }
                }
            }
        }
        if (count > 0)
        {
            TouchTop = true;
            boundary.Position = new Vector2(boundary.Position.X,
minY - boundary.Size.Y);
            velocity = new Vector2(velocity.X, -bouncines * velocity.Y);

            if (walking == false && velocityReceivedX == 0)
                velocity *= new Vector2(1 - friction.X, 1);
        }
    }
```

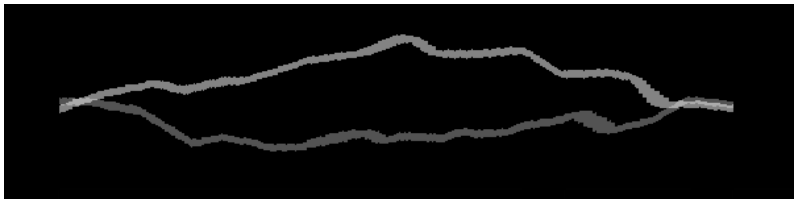
Kód 23 - testování kolize kladným směrem po y

Poté co proběhne cyklus foreach, následuje kontrola proměnné *count*. V této proměnné je uložen počet výskytů kolizí s jednotlivými objekty. Pokud došlo k minimálně jedné kolizi kladným směrem na ose y, nastaví se proměnná *TouchTop* na hodnotu true. Tato proměnná představuje dotek neboli proběhlou kolizi. Dále se objekt, který vlastní tento

CollisionResolver posune, jeho *boundary*, mimo ostatní objekty, tak aby hranou spočíval na bodu *minY*. Do proměnné *velocity*, tedy rychlosti se přes referenci zapíše nová rychlost. V případě že je odrazivost rovna 0, Výsledná rychlost získá hodnotu 0 v ose y. Tak objekt nebude pokračovat dále tam, kde kolize nastala a díky posunutí zpět z těchto míst, nebude zaseknut ve „zdi“.

1.1.18 Částice a efekty

1.1.18.1 Lightning



Obrázek 5 - efekt Lightning

Třída *Lightning* představuje vizuální efekt, blesk, který začíná a končí v určitých bodech. Implementuje rozhraní *IEffect*. Tato třída obsahuje property *VertexBuffer* a *IndexBuffer*, představující seznam vrcholů a jejich indexy. Dále obsahuje metody *Update* a *Draw*. V metodě Update se prochází uzly třídy *NodeLighting* a provádí se jejich metoda *Update*. V metodě *Draw* se pro každý uzel vytváří příslušné vrcholy ve vertex bufferu.

```
for (int i = 0; i < Nodes.Count - 1; i++)
{
   rotated = new Vector3(Nodes[i].Position.X, 0 + Nodes[i].Position.Y
      - Width / 2, 0);

   rotated = Vector3.Transform(rotated,
      Matrix.CreateRotationZ(rotation));

   vertices[i * 4 + 0] = new VertexPositionTexture
      (rotated + new Vector3(Start.Position, 0),

      new Vector2((float)((i / a)
      + ((Game1.Time+ _timeOffset) * _scroll)), 0));
```

Kód 24 - vytváření vrcholů

Pro každý uložený prvek *VertexPositionTexture* v poli *vertices* existuje index nebo indexy, které po třech vytváří vrcholy trojúhelníků, které se mají vykreslovat. Tyto indexy

jsou uloženy v poli *indices*. Každý prvek tohoto pole typů *short* představuje pořadové číslo v poli *vertices*.

```
for (int i = 0; i < (indices.GetLength(0) / 6); i++)
{
  indices[i * 6 + 0] = (short)(0 + i * 4);
  indices[i * 6 + 1] = (short)(1 + i * 4);
  indices[i * 6 + 2] = (short)(2 + i * 4);
  indices[i * 6 + 3] = (short)(0 + i * 4);
  indices[i * 6 + 4] = (short)(2 + i * 4);
  indices[i * 6 + 5] = (short)(3 + i * 4);
}
```

Kód 25 - indexy vrcholů

Data obou těchto polí jsou posléze poslána ke zpracování grafickou kartou. *IndexBuffer* a *VertexBuffer* obsahují metodu *SetData* s parametry typu příslušné pole a počet prvků ke zpracování.

```
IndexBuffer.SetData(indices, 0, (Nodes.Count * 4) + (Nodes.Count) * 4);

Game1.GraphicsGlobal.GraphicsDevice.Indices = IndexBuffer;

VertexBuffer.SetData(vertices, 0, Nodes.Count * 4);

Game1.GraphicsGlobal.GraphicsDevice.SetVertexBuffer(VertexBuffer);
```

Kód 26 - metoda SetData index a vertex bufferu

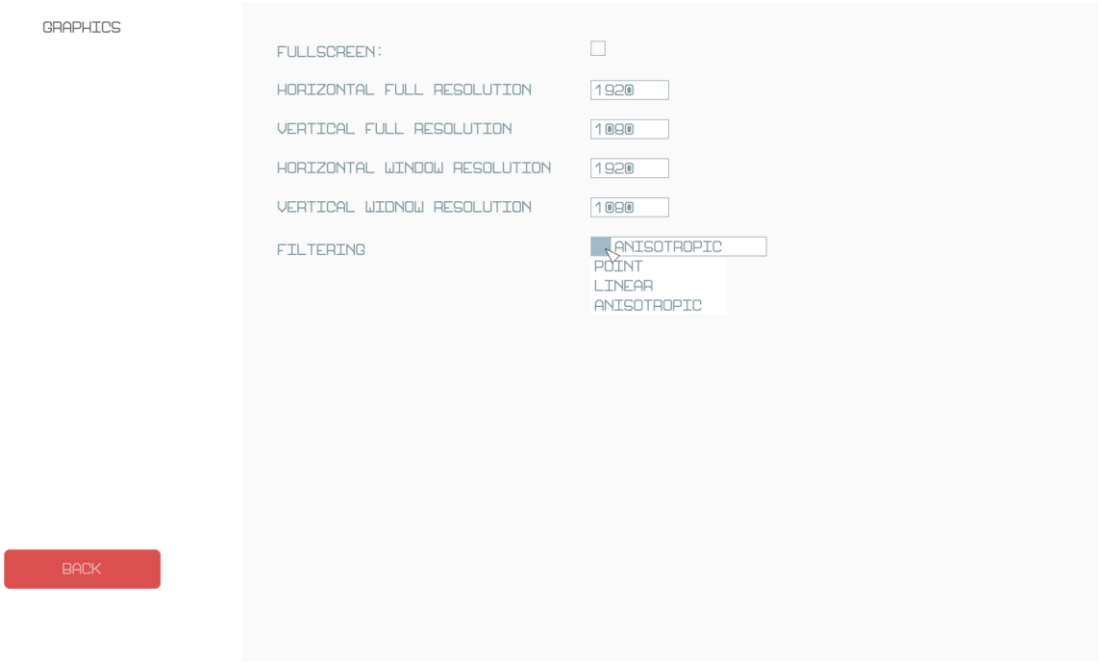
Privátní metody *interpolate* se stará o vytvoření oblouku blesku. Interpoluje mezi hodnotou mezi dvěma proměnnými, *pa* a *pb*. Víceméně představuje převzatou interpolaci, využívanou v jednorozměrném Perlinově šumu. (Procedural Generation Part 1 - 1D Perlin Noise, 2016)

```
private double interpolate(float pa, float pb, float px = 0.5f)
{
  var ft = px * Math.PI;
  var f = (1 - Math.Cos(ft)) * 0.5;
  return pa * (1 - f) + pb * f;
}
```

Kód 27 - interpolace hodnot

1.1.19 **UI prvky**

Tyto prvky představují kontrolky, které musí implementovat rozhraní *IControl*. Spolu tak tvoří ovládání hry v menu. V této části zmíním tři základná UI prvky. Další třídy jako jsou *ScrollBar*, *Radio* a mnoho dalších zde není prostor popisovat. Mohou být nalezeny v kódu projektu.



Obrázek 6 - UI

1.1.19.1 Tlačítko

Třída *Button* je UI prvek s textem reagující na klik myši. Obsahuje delegátové typy, pro přidání metod, které se mají po pokliku provést.

```
public delegate void CallMethodOnClick();
public delegate void CallMethodOnClickChangeWindow(string Key);
public delegate void CallMethodOnClickIndex(byte index);

private CallMethodOnClick _method;
private CallMethodOnClickChangeWindow _methodWindow;
private CallMethodOnClickIndex _methodIndex;
```

Kód 28 – delegáty

V metodě *Update* se provádí samotná kontrola pokliku. Tento prvek není nijak výjimečný z hlediska chování. Tlačítko je aktivováno po volnění tlačítka myši.

```
if (_clickedIn == true)
{
   if (CompareF.RectangleVsVector2(_border, MouseInput.MouseRealPosMenu()) == true
      && MouseInput.MouseStateNew.LeftButton == ButtonState.Released)
   {
      _method?.Invoke();
      _methodWindow?.Invoke(_changeWin);
      _methodIndex?.Invoke(_index);
      Game1.soundSelect.Play();
      returnClicked = true;
   }
```

Kód 29 - aktivace tlačítka a zavolaní delegátů

1.1.19.2 Dropdown menu

Třída DropDown představuje rolovací menu s tlačítkem. Pro svoji činnost využívá enumeraci, která je předána v konstruktoru jako typ. Také využívá kontrolky tlačítka.

```
public DropDown(Type e,int width, Vector2 position, OnChange change = null)
```

Kód 30 - konstruktor DropDown třídy

Poté je v konstruktoru vráceno pole řetězců metodou *Enum.GetNames* . Dále Třída obsahuje příslušné metody *Draw*, Update a další pomocné metody. Příslušný delegát je zavolán při změně volby v menu.

1.1.19.3 Zaškrtávací pole

Třída CheckBox také obsahuje prvek tlačítka po detekci pokliku. Při změně stavu se zavolají příslušné delegáty `_methodChecked a _methodUnchecked`. Jak je již patrné z názvů jeden pro označení a druhý pro odznačení.

1.1.20 **Zbraně**

Pro vytvoření zbraně bude využit návrhový vzor adapter. Zbraně může mít 3 různé způsoby, jak bude střílet. Tyto moduly mohou být vyměňovány za běhu a výrazně zjednodušují návrh a zvyšují přehlednost kódu. Zbraní bude ve hře větší počet a jejich instance budou pro hráče uloženy v jeho kolekci *Weapons*. Protože jich je větší množství, vyberu pouze specifické zástupce. Dále bude pro samostatnou klávesu existovat hození granátu.

1.1.20.1 SimpleGun

Aktuální modul je udržován v privátní proměnné *_module*. Dále obsahuje property definující bonusy, které si hráč mohl zakoupit ve hře. Tyto bonusy jsou definovány číslem, které se přičítá nebo odečítá k základním hodnotám zbraně. Bonus je tedy pro všechny moduly stejný.

```
public short Damage { get { return _module.Damage; } set { _module.Damage = value; } }

public short DamagePlus { get; set; }

public SimpleGun(float velOfProjectile, ushort maxAmmo, ushort ammo, object owner, float time, short da
mage)
{
  _module = new Simple(this,velOfProjectile, maxAmmo, ammo, owner, time, damage);
  DispersionPlus = 0;
  DamagePlus = 0;
  VelocityOfProjectilePlus = 0;
}

public void Update()
{
  _module.Update();
  CompareF.DecreaseToZero(ref Kick, Game1.Delta/16);
  CompareF.DecreaseToZero(ref Light, Game1.Delta / 128);
}

public bool Fire(LineSegmentF rayEnlonged, Vector2Object destination)
{
  if(_module.Fire(rayEnlonged, destination) == true)
  {
    Kick = 8f;
    Light = 1f;
    return true;
  }
  return false;
}
```

Kód 31 - třída zbraně

V konstruktoru třídy *SimpleGun* je dobré hlavně upozornit na předání instance třídy *SimpleGun* tříde *Simple* klíčovým slovem *this*. Tak bude zaručena komunikace mezi modulem a hlavní třídou. Modul je dále volán v metodách *Update* a *Fire*. Chová se tak jako součást třídy *SimpleGun*.

1.1.20.2 Třída Simple

Tato třída implementuje rozhraní *IWeapon* a dědí z třídy *WeaponBase*.

```
public IWeapon OwnerGun { get; set; }
```

Funkcionalita je především zjevná v metodě *Fire*. Kde se například při volání vlastnosti *OwnerGun.DamagePlus* vrací hodnota bonusu hlavní zbraně. Metoda fire je určena pro střílení projektilů nebo testování kolize s paprskem kterým se míří. Obojí má za účel vytvářet poškození na NPC třídách.

```
public bool Fire(LineSegmentF rayEnlonged, Vector2Object destination)
{
   Vector2 barrel = rayEnlonged.Start + rayEnlonged.NormalizedWithZeroSolution() * GunBarrel;

   if (GunTimer.Ready == true)
   {
      Sound.PlaySoundSimple(Game1.sound,1f, (float)(Globals.GlobalRandom.NextDouble() - 0.5f) / 2f, 0f);
      GunTimer.Reset();
      Game1.mapLive.MapProjectiles.Add(new Projectile((short)(Damage + OwnerGun.DamagePlus),
         CompareF.RotateVector2(rayEnlonged.NormalizedWithZeroSolution(),
         (float)((Globals.GlobalRandom.NextDouble() - 0.5f)
         * (Dispersion + OwnerGun.DispersionPlus)))
         * (VelocityOfProjectile+OwnerGun.VelocityOfProjectile), barrel, Owner));
      return true;
   }
   return false;
}
```

Kód 32 - metoda Fire objektu zbraně

1.1.21 **Projektily**

1.1.21.1 Třída Projectile

Tato třída představuje projektil pohybující se prostorem a působící zranění. Dědí ze třídy *ProjectileBase* a impementuje rozhraní *IProjectile*.

Projektil může vystřelit hráč anebo jiná entita. Kdo projektil vlastní se ukládá při jeho vytvoření do chráněné proměnné _from. Jelikož je tato proměnná typu object, může být jako vlastník projektilu jakýkoli jiný objekt.

1.1.21.2 Třída projectileBase

Tato třída slouží jako předek tříd představujících projektily. Implementuje základní metody, které jsou pro všechny potomky této třídy společné. Obsahuje také pole s modifikátory protected. Tímto je zpřístupníme potomkům, ale skryjeme před vnějškem třídy.Hlavní metoda, která zajišťuje funkcionalitu je metoda *UpdateLine*. Ta vytváří úsečku mezi vektory v prostoru *_oldPosition* a *_position*, které mění pozici podle stávající a předešlé pozice projektilu. Poté se testují kolize s touto úsečkou. Tím se u rychle letících projektilů zabrání přeskakování překážek.

```
protected virtual void DamageToNPC(Map map,IProjectile projectile)
protected bool OutOfmap(IProjectile projectile, out Vector2Object hitPos)
protected void UpdateLine()
```

Kód 33 - výčet několika použitých metod

1.1.22 **Hráč**

1.1.22.1 Třída Player

Tato třída je hlavním prvkem hry, díky kterému získáme ve hře možnost se pohybovat a střílet. tato třída je však složitější, než by se mohlo zdát, protože ve hře vyžadujeme možnost řídit stroje, použijeme k prezentování této třídy návrhový vzor adaptér. Reference na objekt bude uložena ve vlastnosti *CurretnObjectControl* typu *IPlayer*. Každý stroj (nebo i samotný hráč) bude prezentován jedním modulem, který se bude moci ve hře měnit. Metodám použitým pro interakci s objektem hráče, budeme předávat tento objekt referencí. Takto může modul s hlavní třídou *Player* komunikovat a ovlivňovat jí. Hráč bude dále moci nosit objekty a v tomto případě nebude moci měnit vozidla. O řešení těchto situací se stará metody *Controling*. Protože tato metoda je rozsáhlejší, uvedu pouze její část. je však patrné, že řešení není zcela triviální. Hráč musí mít požnost objekty pouštět, vozidla opouštět. Dále vozidla mohou být zničena a objekty,

které hráč nosí, nesmí ztratit svoji funkcionalitu. Tímto narážím na třídu *PlayerTurret*, která zaměřuje NPC a střílí na ně i když jí hráč drží.

Dále je patrné v metodě Controling, pří na prvním řádku předání reference na objekt hráče klíčovým slovem this.

```
CurretnObjectControl.ControlPlayer(this);

if (KeyboardInput.KeyboardStateNew.IsKeyDown(Game1.STP.ControlKeys["Grab"])
  && KeyboardInput.KeyboardStateOld.IsKeyUp(Game1.STP.ControlKeys["Grab"]))
{
  bool justLeft = false;

  if (Carry != null)
  {
    Carry.LetGo(Velocity);
    Carry = null;
  }
  else
  {
    if (Game1.PlayerInstance.InVehicle == true)
    {
      LeftVehicle();
      justLeft = true;
    }


    foreach (Inpc npc in Game1.mapLive.MapNpcs.Reverse<Inpc>())
    {
      if (justLeft == false)
      {
        if (npc is Icarry && Game1.PlayerInstance.InVehicle == false)
        {
          if (CompareF.RectangleFVsRectangleF
            (Game1.PlayerInstance.Boundary, npc.Boundary) == true)
          {
            Carry = (npc as Icarry);
            break;
          }
        }
      }
```

Kód 34 - část kódu interakce s vozidly a nositelnými objekty

Také obsahuje metody pro sbírání předmětů, které jsou aktivní i ve vozidlech a když hráč drží předmět.

```
public void AddElectronics(byte amount)
{
   Electronics += amount;
}

public void Addhealth(ushort amount)
{
   PotentialHealth += amount;
   if (PotentialHealth > MaxHealth)
      PotentialHealth = MaxHealth;
}

public void AddShield(ushort amount)
{
   PotentialShield += amount;
   if (PotentialShield > maxShield)
      PotentialShield = maxShield;
}
```

Kód 35 - metody přidání bonusu

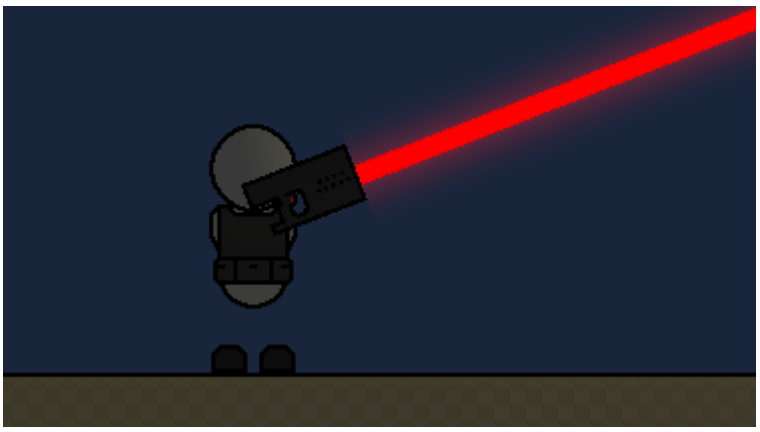
Obrázek 7 - Objekty typu *PlayerTurret* a *Battery*

1.1.22.2 BuddyModule

Tato třída tedy představuje jeden z možných modulů, prezentující klasického hráče. Obsahuje metody pro ovládání pohybu, střelbu a vykreslování postavy. Některé z těchto metody odkazují přímo na objekt *Player* a ovlivňují jeho stav. V kódu 32 lze vidět odkazování na objekt Hráče.

```
public void PhysicsMove(Player player)
{
  if (player.ControlsActive == true)
    player.Resolver.move(ref player.Velocity, new Vector2(2f),
      player.Boundary, 0f, new Vector2(0.2f, 0f), new Vector2(0.1f, 0.02f),
      new Vector2(0.3f), Game1.mapLive.MapMovables, 0, walking: player.Walking);
}

public void ShootWeapons(Player player, Vector2 realPos)
{
  ShootCalc.LineAim(realPos, player, player.WeaponOrigin,
    player.CurrentWeaponObject.GunBarrel, player.CurrentWeaponObject.Reach);

  if (player.WeaponsAvailable == true && player.ControlsActive == true
    && player.Carry == null && player.InVehicle == false)
  {
    if (MouseInput.MouseStateNew.LeftButton == ButtonState.Pressed)
    {
      if (CompareF.WeaponRayObstruction(player.Boundary,
        ShootCalc.RayBarrel, Game1.PlayerInstance).Object == null)
      {
        player.CurrentWeaponObject.Fire(ShootCalc.RaySegment,
          ShootCalc.RayDestination);
      }
      else
      {
        player.CurrentWeaponObject.SetOff();
      }
    }
  }
}
```

Kód 36 - metody pohybu a střelby



Obrázek 8 - střílející hráč

Závěr

Cílem této práce bylo vytvořit – naprogramovat platformou hru. Byly naprogramovány jednoduché NPC „postavy“ reagující s prostředím a reagujícími na hráče. Původní návrh hry počítal i s vestavěným editorem, který je přístupný ve zdrojových kódech jako samostatná část hry, kde mělo být umožněno vytvářet mapy. Mapy samotné a dále i část textur je uložena jako slovník v JSON souborech. Jsou v nich uloženy také nastavení hry a další.

Automatické neboli procedurální generování bylo naprogramováno pouze jednoduchým algoritmem. Procedurální generování samo o sobě představuje příliš rozsáhlý problém a není mu zde věnována dostatečná pozornost. Nikdy nemělo být náhradou celé hry. Toto generování však může být upraveno nebo rozšířeno upravením vhodné metody.

Menu hry a obrazovky včetně celého UI bylo v porovnáním s ostatními částmi hry jednoduchým problémem a podařilo se jej kompletně dokončit. Kód obsahuje i prvky které nebyly ve hře zpřístupněny (ve spustitelném programu), ale jsou implementovány, jako je vlastní prohlížeč souborů. Výběr barev s posuvníky a jiné. Tyto prvky byly původně určeny pro editor map.

Hráč samotný je entita, která může využívat vozidel, střílet, sbírat bonusy a používat výtahy. Dále může nakupovat vylepšení ve hře. Hráč ve hře může používat více zbraní s různými efekty střelby, může však používat vylepšení u pouze jedné zbraně. Tento problém by byl řešen vytvořením dalšího obsahu hry, který se ve výsledku ukázal jako nadbytečná činnost, která se příliš odklání od záměru této práce, protože se jedná o repetitivní činnost. Ovládání hráče může být měněno v nastavení hry.

Tato práce, protože je omezena svým rozsahem nepokrývá vysvětlení a popsaní všech tříd a algoritmů. Byly vybrány jen ty nejzajímavější a popsány společná specifika s podobnými třídami, implementujícími například stejné rozhraní nebo mající stejné použití.

Ve výsledku se vytvoření takto komplexní hry zdá jako v celku složitý problém, který zprvu nebyl úplně zřejmý. Hlavní cíl se podle mého však podařilo naplnit. Byla vytvořena hra s mnoha prvky s možností být rozšířena a s možností nahrávání vlastního obsahu.

Bibliografie

ECMA-334:2017, 2017. *Standard ECMA-334: C# Language Specification*. 5th edition. Geneva: Ecma.

GitHub VelcroPhysics [online], 2018. San Francisco: GitHub [cit. 2018-03-10]. Dostupné z: https://github.com/VelcroPhysics/VelcroPhysics

HLSL [online], b.r. Microsoft [cit. 2018-03-11]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/windows/desktop/bb509561(v=vs.85).aspx

JONES, Wendy., 2008. *Beginning DirectX 10 game programming*. Boston, MA: Thomson Course Technology. ISBN 15-986-3361-9.

MILLINGTON, Ian., 2007. *Game physics engine development*. Boston: Morgan Kaufmann Publishers. ISBN 978-012-3694-713.

Monogame [online], 2018. Monogame [cit. 2018-03-10]. Dostupné z: http://www.monogame.net/

Procedural Generation Part 1 - 1D Perlin Noise, 2016. *Codepen.io* [online]. Oregon (Spojené státy americké) [cit. 2018-03-03]. Dostupné z: https://codepen.io/Tobsta/post/procedural-generation-part-1-1d-perlin-noise

STEPHENS, Rod., 2014. *C# 5.0 programmer's reference*. 1st edition. Indianapolis, IN: John Wiley and Sons. ISBN 978-111-8847-282.

Přílohy

Zdrojový kód hry je k dispozici na https://github.com/BlueBananasaurus/BP