Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Herní engine pro 2D plošinové hry

2024

Jakub Stráněl

Vedoucí práce:
Mgr. Petr Osička, Ph.D.

Studijní program: Informatika,
Specializace: Programování a vývoj software

Bibliografické údaje

Autor:	Jakub Stráněl
Název práce:	Herní engine pro 2D plošinové hry
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2024
Studijní program:	Informatika, Specializace: Programování a vývoj software
Vedoucí práce:	Mgr. Petr Osička, Ph.D.
Počet stran:	33
Přílohy:	elektronická data v úložišti katedry informatiky
Jazyk práce:	český

Bibliographic info

Author:	Jakub Stráněl
Title:	Engine for 2D platformer games
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2024
Study program:	Computer Science, Specialization: Programming and Software Development
Supervisor:	Mgr. Petr Osička, Ph.D.
Page count:	33
Supplements:	electronic data in the storage of department of computer science
Thesis language:	Czech

Anotace

Využitím příslušných algoritmů a postupů, zpracovaných pomocí objektově orientovaného programování, vznikl engine pro 2D plošinové hry, který umožňuje běh hry, vykreslování a správu herních prvků.

Synopsis

By using appropriate algorithms and procedures, processed using object oriented programming, an engine for 2D platform games has been created, which allows the game to run, render and manage game elements.

Klíčová slova: herní engine; algoritmus; objektově orientované programování

Keywords: game engine; algorithm; object oriented programming

Odevzdáním tohoto textu jeho autor/ka místopřísežně prohlašuje, že celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

Obsah

Seznam obrázků

Seznam zdrojových kódů

1 Úvod

Videoherní průmysl světově každým rokem rychle roste. V roce 2022 měl dokonce větší obrat než kombinace hudebního a filmového průmyslu. Herní enginy jsou softwarové platformy, které jsou velmi komplikované a jejichž vývoj je náročný jak finančně, tak i časově. Jejich úkolem je poskytnutí prostředků na vývoj her. Engine se stará o herní logiku, správné vykreslování, přesné chování prvků v herním světě podle předem daných fyzikálních zákonů a mnoho dalšího.

Existuje několik populárních enginů, které jsou dostupné pro vývojářskou komunitu. Patří sem například Unity, Unreal Engine, CryEngine a mnoho dalších. Tyto enginy umožňují tvorbu her pro různé platformy, včetně počítačů, konzolí, mobilních zařízení a VR (virtuální reality).

Téma pro mě bylo zajímavé kvůli už zmíněné náročnosti. Chtěl jsem vytvořit ne příliš složitý engine, na kterém bude snadné ukázat princip fungování herních enginů a jejich klíčových komponent. Během vývoje jsem si uvědomil, kolik práce je potřeba odvést a množství detailů, které se musí zohlednit při simulaci herního světa a interakci s ním.

Herní enginy se ale nepoužívají jen na tvorbu her, mají široké uplatnění i v mnoha jiných oblastech. Jejich výkonnost, schopnost simulace a vizualizace, a flexibilita je činí atraktivními pro různé aplikace. V oblasti simulací a výcviku slouží k vytváření realistických prostředí pro vojenský výcvik, letecké simulátory nebo lékařská cvičení. V architektuře pomáhají při tvorbě realistických vizualizací budov a prostorů a ve světě virtuální a rozšířené reality umožňují vytváření věrohodných zážitků a interaktivních průvodců. Filmový průmysl také využívá herní enginy pro tvorbu vizuálních efektů a digitálních scén, výtvarníci je využívají pro podobné účely. V průmyslu se s nimi můžeme setkat například při tvorbě a návrhu produktů nebo simulaci výrobního provozu. Celkově tak herní enginy překračují rámec her a stávají se nezbytným nástrojem v různých kreativních a profesionálních oblastech.

Výsledkem vývoje by měl být celek komponent, vytvořený pomocí objektové orientovaného programování, který je schopen vykreslovat herní grafiku, přehrávat zvukové efekty, chovat se podle předem daných fyzikálních pravidel, rozpoznávat a řešit kolize, nahrávat soubory do paměti a pracovat s nimi, ukládat prvky z herního světa do souborů, reagovat na vstupy od uživatele a pomocí všech těchto zmíněných věcí tvořit simulaci herního světa.

2 Metodologie

V této kapitole si podrobněji rozebereme jakým způsobem jsem můj engine tvořil, co vše jsem k tomu potřeboval a co vše jsem využíval. Hlavním stavebním kamenem mého engine je externí knihovna Simple DirectMedia Layer (SDL).

2.1 Simple DirectMedia Layer

SDL je multimediální knihovna, která poskytuje rozhraní pro přístup k různým multimediálním funkcím na nízké úrovni. Vytvořená primárně pro podporu her a multimediálních aplikací, SDL slouží jako prostředek pro správu okna, obsluhu událostí, manipulaci s grafikou a zvukem, a mnoho dalších multimediálních operací. SDL je kompatibilní s řadou programovacích jazyků, jako je C, C++, Python, Rust a další. SDL může být také snadno přenášeno mezi různými platformami včetně Windows, Linuxu a macOS. V mém enginu využívám SDL k obsluze událostí, vykreslování grafiky a přehrávání zvuků.

2.2 Entity Component System

Entity Component System (ECS) je architektonický vzor , který se často používá ve vývoji her a simulací. Tento vzor se zaměřuje na vytvoření systému, který se zaměřuje na znovupoužitelnost modulů, je snadno rozšiřitelný a efektivní v oblasti správy a manipulace s daty. Základní myšlenka ECS spočívá ve vytvoření tří hlavních komponent: entit, komponent a systémů.

Entita je základním stavebním blokem v ECS. Je to obecný objekt, který slouží jako unikátní identifikátor pro soubor komponent. Entita sama o sobě neobsahuje žádné datové ani logické informace.

Komponenty jsou jednotlivé části, které obsahují data nebo stav související s entitou. Každá komponenta je obvykle malá datová struktura, která obsahuje informace o určitém aspektu entity, například pozici, rychlost, zdraví nebo grafické reprezentace. Komponenty nemají žádnou funkcionalitu, slouží jako jednoduché úložiště dat.

Systémy jsou místa, kde se provádí veškerá logika a funkcionalita. Každý systém je zodpovědný za určitou část hry nebo aplikace, jako je například vykreslování, fyzika, AI, atd. Systémy pracují s entitami prostřednictvím jejich komponent. Provádějí operace nad komponentami a mohou tak měnit stav a chování entit.

ECS přináší řadu výhod oproti tradičním objektově orientovaným systémům. Jednou z hlavních výhod je oddělení dat od logiky, což umožňuje větší flexibilitu a snadnější testování. Dále je potřeba zdůraznit možnost znovupoužitelnosti, kde jednu komponentu, můžeme přiřadit k několika entitám, například každá postava (entita) ve hře bude mít své zdraví (komponenta). Můžeme také snadno modifikovat entity, komponenty i systémy, bez razantního zásahu do kódu nebo změny kódu na mnoha místech.

V ECS hraje důležitou roli Entity Manager, který, jak už název napovídá, spravuje entity. Poskytuje mechanismus pro získání existujících entit na základě jejich identifikátoru.

Já si z tohoto návrhu beru převážně styl jakým se píše, protože je podle mě přehledný.

2.3 C#

C# jsem si vybral protože to je moderní jazyk, který je objektově orientován, je výkonný a také zároveň snadno použitelný. V tomto jazyku můžeme tvořit v bohatém vývojovém prostředí Visual Studio, kde je možné využít různé nástroje pro ladění a analýzu kódu. Jelikož je jazyk oblíbený a má velkou komunitu, máme k dispozici velké množství zdrojů a dokumentace. Jeden z důvodů volby tohoto jazyka je taky podpora a možnost integrace s právě knihovnou SDL, která je zmíněna výše. Výzvou byla implementace ECS v C#, protože ECS se snaží jít proti stromu dědičnosti v OOP.

2.4 Newtonsoft JSON

Newtonsoft je populární knihovnou pro práci s formátem JSON v C#, která vystupuje svou jednoduchostí použití. Hojně se využívá pro komunikaci se síťovými API. Umožňuje převod .NET objektů do JSON a naopak. Tyto převody se nazývají Serializace a Deserializace. V mém enginu tuto knihovnu využívám při ukládání levelů, kde všechny herní objekty převedu do formátu JSON a zapíšu do souboru. Při načítání tento proces provedu pozpátku.

3 Implementace

Následuje popis implementace herního enginu, který jsem navrhl speciálně pro 2D plošinové hry. Podíváme se na jednotlivé části enginu, popíšeme, jak byly navrženy a realizovány a ukážeme konkrétní příklady kódu, které nám pomohly dosáhnout funkčního výsledku. Věc, kterou jsem začal, je reprezentace prvků v herním světě.

3.1 Implementace ECS

Jak už jsem zmínil výše, myšlenkou ECS je implementovat ho tak, aby nevnikl strom dědičnosti, protože ve velkém programu, může tento strom nabýt velkých rozměrů a stát se tak nepřehledným. V mé implementaci má strom dědičnosti maximální výšku 1, to znamená, že strom má jen kořen a kořen má potomky. Kořenem jsou základní třídy pro entitu a komponentu.

```
1 public class Component {}
```

Zdrojový kód 1: Třída pro základní komponentu

Jak už bylo řečeno, tak komponenty jsou v ECS pouze datovými nosiči a nemají v sobě žádnou logiku, proto je třída prázdná. Tato třída slouží jen k tomu abych mohl v aplikaci rozlišit s jakým typem objektu pracuji.

```
1  public class Entity
2     {
3        public int Id { get; set; }
4        public List<Component> Components = new List<Component>();
5
6        public Entity()
7        {
8           Id = EcsId.GetId();
9        }
10
11       public bool HasComponent<T>()
12       {
13          foreach (Component component in Components)
14          {
15             if (component.GetType().Equals(typeof(T)))
16             {
17                return true;
18             }
19          }
20          return false;
21       }
22  //další kód třídy
```

Zdrojový kód 2: Třída pro základní entitu

Třída pro entitu už vypadá zajímavěji. Každý prvek v herním světě musí být jednoznačně identifikovatelný, tím pádem jsem každé entitě přiřadil unikátní Id. Dostupnost unikátního Id zajišťuje třída EcsId, která přiřazuje Id od 0. Má zaroveň list Id entit, které byly smazány a může je znovu přiřadit. Takhle bude zajištěno, že bude vždy dostupné tolik Id, jaká je velikost datového typu integer. Entita může mít přiřazeno libovolné množství komponent, které jsou uloženy v listu a můžeme zjistit zda je daná komponenta přítomna v entitě pomocí metody HasComponent. Všechny entity jsou při inicializaci programu uloženy do globálního pole a každá část programu s ním může pracovat. Ve třídě je mnoho dalších metod, které si postupně odhalíme v dalších částech tohoto textu.

Další věc, kterou bych rád představil je Entity Manager. Momentálně Manager umí 2 věci, získat hráče a získat pole entit stejného typu, například pole prvků typu Fruit. Třída se dá pochopitelně rozšiřovat o další metody, které budou při růstu engine potřeba.

Metoda projde pole všech entit a u každé zkontroluje, zda není typu T, pokud ano uloží ji do pole a pokračuje dále. Až projde všechny entity, vrátí pole entit, které mají typ T.

Nyní už jsme schopni dát obecnému objektu entita smysl a přiřadit mu komponenty, které ho budou charakterizovat. Například hráč se skládá z komponent name, solid, hitbox, velocity, mass, moveableObject, physicsObject a sprite. Některé z nich jsou primitivní datové typy jako například name (string) nebo solid (boolean), ale i komplexní datové objekty jako třeba sprite (obsahuje informace

```
1  public static List<Entity> GetAll<T>()
2        {
3           List<Entity> list = new List<Entity>();
4           foreach (Entity entity in Variables.Entities)
5           {
6              if (entity.GetType() == typeof(T))
7              {
8                 list.Add(entity);
9              }
10          }
11          return list;
12       }
```

Zdrojový kód 3: Získaní pole entit typu T

o pozici, výšce, délce, rychlostí přehrávání atd.). Teď máme herní objekty, které už mají kontext a každý má svou úlohu ve hře, ale pořád s nimi nemůžeme interagovat. Tuhle interakci, ať už s hráčem nebo mezi sebou, zajistí právě systémy. V enginu mám tyto systémy:

1. grafický,
2. systém spravující načtené multimediální soubory,
3. animační,
4. fyzikální,
5. zvukový.

3.2 Implementace grafického systému

Samotné vykreslování a okno do kterého se vykresluje, obstarává knihovna SDL. K dispozici je velké množství funkcí, já používám převážně vykreslování z textury, která je vytvořena z obrázku. Tato funkce se nazývá SDL_RenderCopyF. Abychom měli kam vykreslovat, musíme inicializovat okno knihovny SDL

Abychom mohli pro každou entitu vykreslit její obrázek, potřebujeme ho mít uložený a musíme vědět, jaká entita má jaký obrázek. Na to jsem vytvořil speciální komponentu s názvem sprite. Sprite ukládá informace o takzvaném sprite sheetu, což je obrázek, na kterém jsou po řádcích uloženy animace různých úkonů jednotlivých postav nebo herních prvků, o tom ale více v kapitole o animacích. Pro teď nám postačí vědět, že sprite komponenta obsahuje assetsIndex, což je index obrázku dané entity v poli Assets, kde jsou načteny multimediální soubory.

Engine celý běží v takzvaném game loopu, v podstatě je to jeden velký while cyklus. V jednom běhu tímto cyklem všechny zmíněné systémy nějakým způsobem upraví herní svět. To se stane za sekundu několikrát a vzniká iluze simulace. Rychlost se měří ve snímcích za sekundu (Frames Per Second). Jeden snímek

```
public class SpriteComponent : Component
   {
      public int assetsIndex;
      // další proměnné
   }
```

Zdrojový kód 4: Sprite komponenta

značí jeden průběh cyklem. Grafický systém přichází na řadu jako poslední a všechny změny v aktuálním cyklu vykreslí na obrazovku. Ještě než vykreslím obrázek, tak pomocí metody GetFacing, která je definována na třídě Entity, zjistím, zda se herní postava dívá doleva (metoda vrátí 0) nebo doprava (metoda vrátí 1) a rozhodnu, jakou funkci na vykreslování použiji. Konstruktor okna přijímá řadu argumentů. Pro nás jsou zajímavé výška a šířka okna.

```
if (entity.GetFacing() == 1)
{
   SDL.SDL_RenderCopyF(Variables.Renderer, t, ref s, ref h);
 }
   else
{
    SDL.SDL_RenderCopyExF(Variables.Renderer, t, ref s, ref h, 0,
       IntPtr.Zero, SDL.SDL_RendererFlip.SDL_FLIP_HORIZONTAL);
}
```

Zdrojový kód 5: Vykreslování do okna

V tuto chvíli už máme herní objekty, které můžou nést libovolná data a pomocí grafického systému je můžeme vykreslit do okna.

3.3 Implementace načítání multimediálních souborů

Všechna logika ohledně načítání souborů se odehrává ve třídě Assets. Třída Assets má tři atributy. První z nich je pole AssetsArray, kde jsou uložené už načtené textury z obrázků, z textů a zvukové soubory. Další je pole AssetsPaths, což není nic jiného než pole cest k multimediálním souborům. Posledním atributem je Index, který slouží jako index v poli AssetsArray a po každém přidání nového souboru se inkrementuje.

Ukládáním načtených textur a souborů do AssetsArray se zlepšuje výkon aplikace, protože se předejde opakovanému načítání stejných souborů. Toto je obzvláště důležité v herním vývoji, kde rychlý přístup k multimediálním souborům a jejich efektivní využití jsou klíčové pro plynulý běh hry.

Soubory se nahrávají vždy při inicializaci programu pomocí metody AddAsset. Tato metoda bere jako argument cestu k souboru. Uvnitř metody zjistíme příponu souboru a podle ní se rozhodneme jakou funkci pro vytvoření textury

nebo nahrátí zvukového souboru použijeme. Tyto funkce mi poskytuje knihovna SDL. Funkce nám vrátí texturu a tu uložíme do pole AssetsArray na index Index a inkrementujeme ho. Metoda AddAsset vrací index v poli AssetsArray, na kterém jsme soubor uložili.

```
1  public static int AddAsset(string path)
2        {
3           AssetsPaths.Add(path);
4           var extension = path.Split('.').Last();
5           if(extension == "jpg" || extension == "png")
6           {
7              IntPtr asset = SDL_image.IMG_Load(path);
8              IntPtr texture = SDL.SDL_CreateTextureFromSurface(
                  Variables.Renderer, asset);
9              if (texture == IntPtr.Zero)
10                return -1;
11             AssetsArray[Index] = texture;
12
13             return Index++;
14          }
15          // další kód
```

Zdrojový kód 6: Nahrání jpg nebo png obrázku

Ve třídě Assets je ještě funkce na přidání textu. Text se nedá vykreslit jen tak jednoduše. Nejprve musíme načíst soubor s fontem, který chceme použít a vygenerujeme textovou plochu s požadovaným textem a barvou. Plochu musíme opět převést na texturu a až potom můžeme text vykreslit do okna. Uložení do AssetsArray už je stejné jako u předchozí metody. Všechny pomocné funkce opět poskytuje SDL.

Pokud chceme, aby byla entita viditelná v herním světě, musí mít svou texturu. Tu ale, jak už bylo řečeno, musíme vytvořit a uložit. Toho docílíme metodou AddAsset, která nám vrátí index do pole AssetsArray, které je dostupné globálně. Index, který dostaneme uložíme do sprite komponenty, aby byl vždy dostupný. Tohle vše se odehrává při inicializaci programu, pro všechny komponenty, které jsou v herním světě. Grafický systém poté v každém průběhu cyklem projde pole všech entit, u každé entity se podívá na index ve sprite komponentě, a vykreslí texturu, která je na daném indexu v poli souborů. Přesněji se vykresluje do takzvaného hitboxu. Hitbox je neviditelná oblast, která je používána k detekci kolizí nebo interakcí mezi objekty, jako jsou postavy, nepřátelé, projektily nebo herní prostředí. V podstatě reprezentuje fyzické hranice objektu. O hitboxu si povíme více ve fyzikálním systému.

Teď už tedy umíme vykreslit entity a víme jakým způsobem se nahrávají a ukládájí soubory, co jsou k tomu potřeba. Nyní entitám vdechneme život tím, že je rozpohybujeme.

```
1  public static int AddText(string text, SDL.SDL_Color color, string
       font)
2          {
3              IntPtr arial = SDL_ttf.TTF_OpenFont(Path.Combine(AppContext
                   .BaseDirectory, font), 26);
4              // { r = 255, g = 255, b = 255, a = 255 };
5              IntPtr surfaceText = SDL_ttf.TTF_RenderText_Solid(arial,
                   text, color);
6              IntPtr textTexture = SDL.SDL_CreateTextureFromSurface(
                   Variables.Renderer, surfaceText);
7
8              if (textTexture == IntPtr.Zero)
9                  return -1;
10             AssetsArray[Index] = textTexture;
11
12             return Index++;
13         }
```

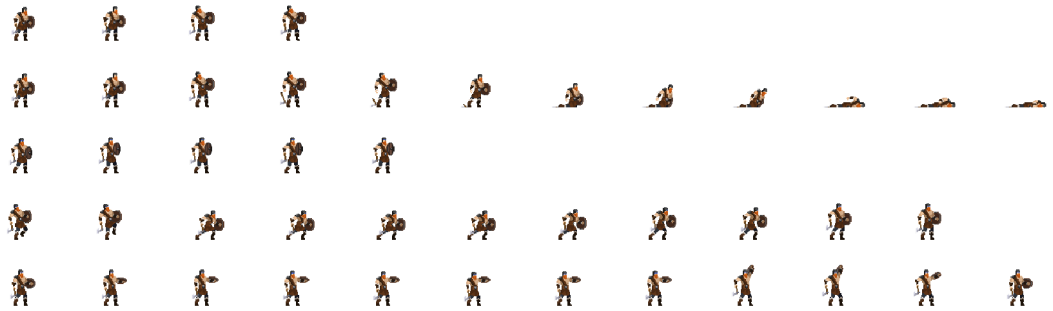
Zdrojový kód 7: Nahrání textu

3.4 Implementace animačního systému

Animace ve 2D hrách jsou klíčovým prvkem, který přináší herní svět k životu, propůjčuje postavám pohyb a vytváří dynamické herní prostředí. Typicky, animace ve 2D hrách jsou realizovány prostřednictvím sérií pečlivě vytvořených snímků (tzv. sprites), které jsou přehrávány v rychlé sekvenci, aby vytvořily iluzi pohybu. Tento proces, známý jako sprite animation, umožňuje postavám provádět různé akce, jako je chůze, běh, skákání nebo provádění útoků. S použitím moderních nástrojů a technik, jako jsou animační softwary a herní enginy, mohou vývojáři vytvářet hladké a realisticky působící animace, které zvyšují poutavost a vizuální atraktivitu hry. Při tvorbě animací se dbá na detaily jako jsou časování, plynulost pohybu a interakce s herním prostředím, což vše přispívá k celkovému dojmu a zábavě při hraní 2D her.

U sprite animation jsou důležité soubory, které se jmenují sprite sheets. Sprite sheet je soubor, který obsahuje řadu menších obrázků, neboli sprites. Tyto sprites mohou být různé - od postav a nepřátel po pozadí a herní prvky. Použitím sprite sheetu mohou vývojáři snadno organizovat a přistupovat k mnoha různým grafickým prvkům hry, přičemž je všechny načítají jako jediný obrazový soubor. To vede k výraznému snížení výpočetního výkonu potřebného k načítání a zpracování grafiky, protože se minimalizuje počet nutných načítacích operací. Ve hře se pak jednotlivé sprites vyřezávají z tohoto sheetu a používají pro animace nebo zobrazení herních objektů.

Abych mohl vykreslovat jednotlivé sprites ze sprite sheetu, musím je nějakým způsobem vyřezat. K tomu slouží source (zdrojový) a destination (cílový) obdélník. Source obdélník specifikuje oblast v rámci textury nebo sprite sheetu, která má být vykreslena. Například, pokud máte sprite sheet s různými animačními fá-



Obrázek 1: Sprite sheet

zemi postavy, source rectangle definuje, která konkrétní fáze má být použita pro aktuální snímek animace. Na druhé straně, destination rectangle určuje, kam a v jaké velikosti má být tento vybraný sprite vykreslen na obrazovce nebo v rámci herního okna. Mým destination obdélníkem je vždy již zmíněný hitbox, což není nic jiného než obdélník se souřadnicemi x a y. Source obdélník musím dopočítat podle toho, jaký snímek animace je zrovna na řadě. Všechna data, která k tomu pořebujeme, jsou ve sprite komponentě.

```
1  public class SpriteComponent : Component
2      {
3          public int assetsIndex;
4          public int spriteX;
5          public int spriteY;
6          public int spriteWidth;
7          public int spriteHeight;
8          public int spriteRow;
9          public int spriteCol;
10         public int spriteXDiff;
11         public int spriteYDiff;
12         public int speed;
13         public short facing;
14         public bool animated;
15     }
```

Zdrojový kód 8: Sprite komponenta

Vždy potřebujeme znát levý horní roh spritu, jeho výšku a šířku a vzdálenost mezi jednotlivými sprity. Typicky každý řádek sprite sheetu představuje jiný úkon postavy, například na prvním může být chůze, na druhém běh a tak dále. Proto máme zaznamenaný současný řádek a sloupec ve sprite sheetu. Velký dopad na celkový dojem z animace má rychlost přehrávání, tu nám v komponentě zachycuje atribut speed. Atribut facing může být hodnota 0 (doleva) nebo 1 (doprava), podle toho jakým směrem je postava natočená. Zbývá nám už jen atribut animated, který nám říká jestli je entita animovaná nebo ne, protože

v herním světě můžeme mít prvky, které žádnou animaci nemají a jsou pouze statické, často jsou to objekty, které modelují herní prostředí.

Nyní si konečně můžeme představit jak kompletně funguje vykreslování v grafickém systému.

```
1                var sc = entity.GetComponent<SpriteComponent>();
2                var t = Assets.AssetsArray[sc.assetsIndex];
3                var h = entity.GetHitbox();
4
5                if (sc.animated)
6                {
7                   SDL.SDL_Rect s = new SDL.SDL_Rect()
8                   {
9                      x = sc.spriteX + (sc.spriteCol * sc.spriteXDiff
                          ),
10                     y = sc.spriteY + (sc.spriteRow * sc.spriteYDiff
                          ),
11                     h = sc.spriteHeight,
12                     w = sc.spriteWidth,
13                  };
14                  if (entity.GetFacing() == 1)e
15                  {
16                     SDL.SDL_RenderCopyF(Variables.Renderer, t, ref
                          s, ref h);
17                  }
18                  else
19                  {
20                     SDL.SDL_RenderCopyExF(Variables.Renderer, t,
                          ref s, ref h, 0, IntPtr.Zero, SDL.SDL_
                          RendererFlip.SDL_FLIP_HORIZONTAL);
21                  }
22               }
23               else
24               {
25                  SDL.SDL_RenderCopyF(Variables.Renderer, t, IntPtr.
                       Zero, ref h);
26               }
```

Zdrojový kód 9: Vykreslování

Jako první si do proměnné uložíme sprite komponentu, kterou získáme pomocí generické metody GetComponent. Z této proměnné získáme index, na kterém je v poli AssetsArray uložená textura sprite sheetu nebo obrázku entity a opět si ji uložíme do proměnné. Jako poslední získáme hitbox pomocí metody GetHitbox. Díky atributu animated zjistíme, jestli je entita statická nebo naopak. Pokud je statická, tak je to jednoduché a pouze ji pomocí SDL vykreslíme do okna na souřadnice hitboxu, který slouží jako destination obdélník. Source obdélník v tomto případě nepotřebujeme, protože se textura vykresluje celá. V opačném případě musíme dopočítat source obdélník. Toho docílíme tak, že si

vytvoříme novou instanci obdélníku a jako argumenty ji předáme výšku spritu, délku spritu a pomocí dat ze sprite komponenty vypočítáme x a y souřadnici na sprite sheetu. Výpočet x-ové souřadnice spritu ve sprite sheetu nám vyjadřuje rovnice 1.

$$xSouřadniceSpritu + (sloupecSpritu \times xVzdálenostMeziSprity) \quad (1)$$

Jakmile máme vytvořený source obdélník, stačí jen zjistit, jestli je entita natočená doleva nebo doprava a vykreslit. Jako argument vykreslovací funkci předáváme SDL Renderer, který se inicializuje na začátku programu, texturu a source a destination obdélník. U komplikovanější funkce, kde je potřeba otočit texturu ještě nastavujeme, jak přesně se má otočit.

Animace už umíme správně vykreslovat, teď je potřeba vytvořit logiku přepínání logiku snímků animace. Assets třída má ještě metodu Animate.

```
1 public static void Animate(double elapsed)
2       {
3          MainProgram.Animations(elapsed);
4       }
```

Zdrojový kód 10: Animate metoda

Metoda Animate volá další metodu v MainProgram a to Animations. O této třídě si povíme dále v tomto textu. Uvnitř metody Animations máme tuto animační logiku.

```
1  public static void Animate(double elapsed)
2        var player = EntityManager.GetPlayer();
3           var arr = Variables.KeyArray;
4           var grounded = player.Grounded();
5           SpriteComponent sc = player.GetComponent<SpriteComponent>()
               ;
6
7        if (grounded && (arr[(int)SDL.SDL_Scancode.SDL_SCANCODE_LEFT]
            == 1 || arr[(int)SDL.SDL_Scancode.SDL_SCANCODE_RIGHT] ==
            1))
8           {
9              sc.spriteRow = 2;
10             sc.spriteCol = (int)(SDL.SDL_GetTicks() / sc.speed) % 8;
11             player.SetSprite(sc);
12          }
```

Zdrojový kód 11: Animační logika hráče

Napřed získáme data, která potřebujeme a to je entita hráče, pole keyArray, které má jako indexy kódy kláves a na indexu je vždy uloženo, zda je daná klávesa

stisknuta. Dále si uložíme do proměnné grounded výsledek metody Grounded, který je true nebo false, podle toho jestli se postava hráče dotýká země nebo ne. Program dojde k podmínce a rozhodne, zda-li je hráč na zemi a zároveň jestli jsou stisknuty šipka doleva nebo doprava. Představme si, že podmínka je pravdivá. Nastavíme si řádek ve sprite sheetu na 2, protože na třetím řádku se nachází animace běhu hráče a nyní vypočítáme, jaký snímek by měl být momentálně přehrán. Toho dosáhneme tak, že zavoláme funkci GetTicks, která nám vrátí čas uplynulý od inicializace SDL, vydělíme ji rychlostí přehrávání a nad tímhle vším provedeme operaci modulo osmi, protože na daném řádku je osm snímků animace. Zahrnout uplynulý čas do této rovnice je velmi důležité, protože kdybychom animaci upravovali v každém průběhu cyklem, tak by se animace měnila podle toho, kolik by měl uživatel na svém počítači ve hře snímků za sekundu, což je nežádoucí chování. Výsledkem jsou měnící se obrázky za daný časový interval, utvářející iluzi pohybu.

Jsme ve stavu, kdy máme vykreslené entity , které mají své animace, pořád ale stojí na stejném místě a neprobíhá mezi nimi žádná interakce. Na řadu přichází fyzikální systém, aby tohle napravil.

3.5 Implementace fyzikálního systému

Fyzikální systém je v mém herním enginu největším systémem. Je zásadní pro interakci objektů ve hře. Implementuje algoritmy na detekci kolizí a jejich nápravu, obsahuje gravitační logiku, řeší zrychlení a další. Aby mohl fyzikální systém upravovat entity, budeme muset přidat několik komponent.

První z nich bude komponenta PhysicsObjectComponent, která není nic jiného, než boolean hodnota vyjadřující zda entita podléhá fyzikálním zákonům v herním světě. Ve fyzikálním systému musíme vědět, jestli se entita může pohybovat, k tomu nám slouží komponenta MoveableObjectComponent. Pokud se entita pohybuje, musíme někam ukládat její rychlost a to do komponenty VelocityComponent, kde je uložena v metrech za sekundu a má typ double. Entita už se může pohybovat nějakou rychlostí, ale pořád neznáme směr. Směr a fyzikální sílu vyjadřuji vektorem, kde délka vektoru je velikost síly v newtonech. Pro vektor jsem vytvořil třídu, která obsahuje x-ovou, y-ovou souřadnici a metodu na výpočet délky vektoru. Potom působení dvou sil je jednoduché sečtení dvou vektorů, kde výsledný vektor je výsledná síla. Tento vektor je tedy uložen ve ForceComponent. Další vlastnost, kterou má každá entita, na kterou platí fyzikální zákony je hmotnost a ta je uložena v MassComponent. Budeme také potřebovat příznak, který nám řekne, jestli je entita pevná a může interagovat s dalšími pevnými objekty. Pevným objektem je myšlen třeba hráč nebo překážka a opakem potom například strom v pozadí. Pokud bude objekt pevný, a může narážet do pevných objektů, bude zapotřebí možná nejdůležitější komponenty v celém fyzikálním systému a tou je hitbox.

Hitbox je komponenta, která definuje hranici okolo entity, kde může dojít k interakcím, jako jsou kolize. Obvykle se jedná o jednoduchý geometrický tvar,

```
1  public class Vector
2     {
3        public double x { get; set; }
4        public double y { get; set; }
5
6        public Vector(double x, double y)
7        {
8           this.x = x;
9           this.y = y;
10       }
11
12       public double Length()
13       {
14          double result = 0;
15          result = Math.Sqrt(x * x + y * y);
16          return result;
17       }
18    }
```

Zdrojový kód 12: Třída vektoru

jako je obdélník, kruh, nebo dokonce komplexnější polygon, který je mapován na vizuální reprezentaci entity ve hře. Hitboxy jsou klíčové pro detekci kolizí mezi objekty, protože můžeme jednoznačně říct, zda se hitboxy překrývají, či nikoli. K reprezentaci hitboxu využívám obdélník z kniihovny SDL (SDL Rect). Takže je to objekt, který má x-ovou souřadnici, y-ovou souřadnici, výšku a délku. Pomocí tohoto obdélníku můžu určit fyzickou pozici a velikost entity v herním světě. Výše jsem zmínil, že hitbox využívá i animační systém, který do něj vykresluje textury entit.

Teď se podíváme na metodu, kterou každá entita v každém snímku projde. Je to metoda Update ve třídě Physics. Jako první se vyfiltrují entity, které nejsou pohyblivé. Dále pokud je entita na zemi, nastavíme rychlost na nula a resetujeme sílu pouze na sílu gravitační.

```
1   if (isGrounded)
2           {
3              entity.SetForce(new Vector(
4              0, Variables.GravitationalPull * entity.GetMass()
5              ));
6              entity.SetVelocity(0);
7           }
```

Zdrojový kód 13: Entita je na zemi

V else větvi přepokládáme, že je entita ve vzduchu. Podle toho musíme upravit síly. Pokud člověk vyskočí v reálném světě, stoupá vzhůru do chvíle, než se dostane do bodu, kdy síla vynaložená při výskoku postupně vyprchá, protože je

přetlačena gravitační silou a člověk znovu padá k zemi. Takže každou sekundu se síla obrací. To simuluji tak, že za určitý časový interval vždy k síle přičtu nějaký zlomek síly (síla mířena dolů je kladná a naopak) a tím se síla postupně otočí zpět dolů. Tato část síly je uložena v proměnné GravitationalDiff. Maximální síla, která může působit dolů je ve výši té gravitační (GravitationalPull). Sílá působící na ose x se odečítá obdobně jako na té y-ové, akorát tam není žádná maximální hranice.

```
1
2
3  else
4          {
5
6              var yForce = Variables.GravitationalPull * entity.
                   GetMass();
7              if (force.y < yForce)
8              {
9                  force.y += Variables.GravitationalDiff * elapsed;
10             }
11             else if (force.y > yForce)
12                 force.y = yForce;
13
14             if(force.x < xForce)
15             {
16                 force.x += GravitationalDiff * elapsed;
17             } else if(force.x > xForce) {
18                 force.x -= GravitationalDiff * elapsed;
19             }
20             entity.SetForce(force);
21         }
```

Zdrojový kód 14: Entita je ve vzduchu

Další fáze, kterou entita projde, je výpočet rychlosti. Nejprve musíme vypočítat zrychlení, čehož docílíme podílem síly a hmotnosti entity. K rychlosti potom přičteme součin akcelerace a uplynulého časového úseku a vyjde nám momentální rychlost entity.

```
1  double acceleration = forceLength / mass;
2  velocity += elapsed * acceleration;
```

Zdrojový kód 15: Výpočet momentální rychlosti

Rychlost potom musíme promítnout do pozice entity. Toho docílíme tak, že získáme znaménko síly pomocí Math.Sign (tím zjistíme, jestli směr síly je nahoru, dolů, doleva nebo doprava), to potom vynásobíme momentální rychlostí a uplynutým časem, výsledek tohoto ještě vynásobíme herními metry a vyjde nám

o kolik pixelů máme entitu posunout. Zapíšeme entitě rychlost do komponenty, se zápisem souřadnic ještě počkáme a předáme je jako argument třídě Main, metodě Input. O třídě Main si povíme dále v textu. Následně entitu předáme do metody CheckCollision, která nám vrátí seznam všech entit, se kterými koliduje a poté se znovu předá s každou touto kolidující entitou do metody SolveCollision, která kolize vyřeší.

```
1  x = (float)(Math.Sign(force.x) * velocity * elapsed) * Variables.
       Meter; // 1m/s = 20px/s
2  y = (float)(Math.Sign(force.y) * velocity * elapsed) * Variables.
       Meter; // 1m/s = 20px/s
3  entity.SetVelocity(velocity);
4
5  MainProgram.Input(keyArray, entity, elapsed, isGrounded, x, y);
6
7  List<Entity> list;
8  list = CheckCollision(entity);
9  if(list.Count() > 0)
10     {
11        foreach(Entity collider in list)
12        {
13            SolveCollision(entity, collider);
14         }
15     }
```

Zdrojový kód 16: Výpočet posunu x-ové a y-ové souřadnice

Jako první se podíváme, jak funguje metoda pro detekci kolizí. Metoda nejprve zkontroluje, zda první entita v seznamu Variables.Entities není null. Pokud ano, metoda se ukončí a vrátí prázdný seznam, protože neexistují žádné entity k ověření kolize. Metoda poté vstupuje do while cyklu, kteráýiteruje přes všechny entity v seznamu Variables.Entities. Pokud entita není pevná nebo je to stejná entita jako ta, která je testována na kolizi, smyčka pokračuje na další iteraci. Pro každou zbylou entitu získá jejich hitboxy (hitbox pro testovanou entitu a collider pro entitu ze seznamu). Metoda vypočítává čtyři hodnoty (x1, y1, x2, y2), které reprezentují vzdálenosti mezi hranami hitboxů obou entit. Pokud je kterákoli z těchto hodnot větší než 0, znamená to, že mezi hitboxy není žádné překrytí, a tedy nedošlo ke kolizi. Smyčka se poté posune na další entitu. Jestliže žádná z hodnot (x1, y1, x2, y2) není větší než 0 a současně není ani rovna 0, je to indikace, že došlo ke kolizi. V takovém případě je aktuálně procházená entita přidána do seznamu. Po dokončení iterace přes všechny entity, metoda vrací seznam, který obsahuje všechny entity, s nimiž testovaná entita koliduje.

Teď už máme seznam entit, se kterými předaná entita koliduje a potřebujeme tyto kolize vyřešit. Na řadu přichází metoda SolveCollision. Metoda má přes 70 řádků kódu, takže si ukážeme jen části z ní a celou si ji popíšeme. Metoda vypočítá středy obou entit (eMid pro entitu a cMid pro collider) a vypočítá jejich horizontální (dx) a vertikální (dy) rozdíl. Tento rozdíl je normalizován

```
1           while (i < Variables.Entities.Count)
2           {
3               if (!Variables.Entities[i].GetSolid() || entity ==
                    Variables.Entities[i])
4               {
5                   i++;
6                   continue;
7               }
8
9               var hitbox = entity.GetHitbox();
10              var collider = Variables.Entities[i].GetHitbox();
11
12              float x1 = collider.x - (hitbox.x + hitbox.w);
13              float y1 = collider.y - (hitbox.y + hitbox.h);
14              float x2 = hitbox.x - (collider.x + collider.w);
15              float y2 = hitbox.y - (collider.y + collider.h);
16
17              if (x1 > 0 || y1 > 0 || x2 > 0 || y2 > 0)
18              {
19                  i++;
20                  continue;
21              }
22
23              if (x1 == 0 || y1 == 0 || x2 == 0 || y2 == 0)
24              {
25                  i++;
26                  continue;
27              }
28              list.Add(Variables.Entities[i]);
29              i++;
30          }
31          return list;
```

Zdrojový kód 17: While cyklus metody CheckCollison

podle polovin šířek a výšek collider, což naznačuje, jak blízko středy jsou k sobě ve vztahu k velikosti collideru. Metoda pak používá absolutní hodnoty těchto rozdílů (absDx, absDy) k určení typu kolize. Pokud jsou rozdíly absDx a absDy podobné (rozdíl menší než 0.1), je interpretováno, že kolize nastala na rohu collideru. Dále pokud je absDx větší než absDy, kolize je považována za horizontální (na levé nebo pravé straně). Jestliže je absDy větší, kolize je považována za vertikální (na horní nebo spodní straně). Na základě předchozího rozhodnutí metoda upravuje hitbox entity. Pro rohové kolize se upravuje hitbox entity tak, aby se posunul pryč od collideru jak v horizontálním, tak ve vertikálním směru. U horizontálních kolízí se hitbox entity posune pouze horizontálně. Pro vertikální kolize: Hitbox entity se posune vertikálně. V případě, že kolize je na spodní straně, je také nastavena vertikální rychlost entity na 0, což naznačuje zastavení pohybu když entita narazí při výskoku například do stropu. Posun se provádí výpočtem rozdílu

mezi hranami entity a collideru. Tento rozdíl je poté použit k posunutí entity tak, aby se vyřešila kolize. Směr posunu závisí na tom, zda jsou dx nebo dy pozitivní nebo negativní, což určuje, z jaké strany entita a collider kolidují.

```
1           else if (absDx > absDy)
2           {
3               //positive x
4               if (dx < 0)
5               {
6                   var length = collider.GetRight() - entity.GetLeft();
7                   entity.SetHitbox(length, 0);
8               }
9               else //negative x
10              {
11                  var length = entity.GetRight() - collider.GetLeft();
12                  entity.SetHitbox(-length, 0);
13              }
```

Zdrojový kód 18: Vyřešení horizontální kolize

Jak můžeme vidět v ukázce kódu, pokud se entity srazily ze strany, tak je posuneme o tolik pixelů o kolik se překrývají a to na tu stranu, na které se srazily.

V tuto chvíli už máme herní svět, který se hýbe, objekty v něm spolu kolidují, působí na ně gravitace a mají různé animace.

3.6 Implementace zvukového systému

Poslední zásádní systém, který si představíme je systém zvukový. Určitě se všichni shodnou, že trefné zvukové efekty a výstižný hudební podklad jsou pro kvalitní hry klíčové. Můj zvukový systém je velmi jednoduchý a opět využívám dostupné funkce knihovny SDL. Každému zvuku, který chci přehrát, musím přiřadit kanál, na kterém bude přehráván. Vytvořil jsem třídu Audio a v ní dvě metody.

První metoda PlayEffect slouží k přehrávání zvukových efektů, které neběží po velký časový interval. První parametr -1 značí, že SDL_mixer může vybrat jakýkoli dostupný kanál a tím pádem mě od této povinnosti odstiňuje. Druhý parametr effect je ukazatel na zvukový soubor. Zvukové soubory je nutno nahrát a uložit do paměti, to dělá má třída Assets. Poslední číselný parametr vyznačuje, zda se bude hudba přehrávat pořád dokola (-1) či nikoli (0). Metoda PlayMusic slouží k opakovanému přehrávání zvukové nahrávky. Typicky se používá pro hudební podklad, který nás doprovází herním světem. Metody ze zvukového systému jsou globální a můžeme je použít napříč celým enginem v libovolné části kódu.

```
1  public static class Audio
2      {
3          public static void PlayEffect(IntPtr effect)
4          {
5              SDL_mixer.Mix_PlayChannel(-1, effect, 0);
6          }
7
8          public static void PlayMusic(IntPtr music)
9          {
10             SDL_mixer.Mix_PlayChannel(-1, music, -1);
11         }
12     }
```

Zdrojový kód 19: Třída Audio

3.7 Implementace ukládání, nahrávání a mazání levelů

Díky ukládání, nahrávání a mazaní částí herního světa, jsme schopni hráči vytvořit nespočet unikátních zkušeností a zážitků. Můžeme hru rozdělit na mnoho fází a tvořit nové výzvy. Máme k dispozici třídu Level a v ní šest metod a atribut LevelNumberFile. Tento atribut obsahuje cestu k souboru s číslem uložených levelů. Pracujeme s ním pomocí trojice metod a to SaveNumberToFile, ReadNumberFromFile a GetLevelNumber. SaveNumberToFile bere jako argument číslo a to uloží do zmíněného souboru. Metoda ReadNumberFromFIle přečte číslo, které je v souboru a použije ho jako návratovou hodnotu. A jako poslední metoda GelLevelNumber, která se používá při ukládání levelu, využije obě předchozí metody, takže vrátí současné číslo uložené v souboru a do souboru zapíše současné číslo zvětšeno o jedna.

Nyní se nabízí metoda SaveLevel. Metoda začíná zavoláním funkce GetLevelNumber, takže získá číslo levelu a zapíše nové. Název souboru je vytvořen spojením cesty, slova level a čísla levelu, s příponou .txt. Výsledný řetězec ukazuje na umístění souboru, kam bude level uložen. Metoda pak serializuje seznam entit (Variables.Entities) do formátu JSON pomocí knihovny NewtonSoft. Serializovaný JSON řetězec je uložen do souboru pomocí File.WriteAllText. Máme uložené entity, ale ne cesty k souborům, které tyto entity vyžadují, to musíme napravit. Proto vytvoříme nový soubor assets spojený s číslem levelu a do něj zapíšeme obsah pole Assets.AssetsPaths, který obsahuje cesty k momentálně načteným souborům. Teď už máme uložené všechny potřebné informace.

Entity a soubory chceme načíst zpět a to pomocí metody LoadLevel. Postup této metody je pochopitelně opačný, než u metody předchozí. Pomocí argumentu, který obsahuje název souboru najdeme potřebné soubory, přečteme je, deserializujeme a načteme do polí Entities a Assets.

Metoda DeleteLevel jen jednoduše vyhledá soubory podle parametru, kde je uložen název souboru a soubory smaže.

Nyní už máme vše co potřebujeme pro správu levelů.

```
1  public static void SaveLevel()
2        {
3           int number = GetLevelNumber();
4           string fileTitle = "./levels/level" + number + ".txt";
5
6           string json = JsonConvert.SerializeObject(Variables.
                Entities, Formatting.Indented,
7           new JsonSerializerSettings
8           {
9              ReferenceLoopHandling = Newtonsoft.Json.
                   ReferenceLoopHandling.Ignore,
10             TypeNameHandling = TypeNameHandling.Auto,
11          });
12
13          if (!System.IO.Directory.Exists("levels"))
14             System.IO.Directory.CreateDirectory("levels");
15
16          File.WriteAllText(fileTitle, json);
17
18          string assetsTitle = "./levels/assetslevel" + number + ".
                txt";
19
20          json = JsonConvert.SerializeObject(Assets.AssetsPaths,
                Formatting.Indented,
21          new JsonSerializerSettings
22          {
23             ReferenceLoopHandling = Newtonsoft.Json.
                   ReferenceLoopHandling.Ignore,
24             TypeNameHandling = TypeNameHandling.Auto,
25          });
26
27          File.WriteAllText(assetsTitle, json);
28       }
```

Zdrojový kód 20: Metoda SaveLevel

3.8 Implementace jednoduchého uživatelského rozhraní

Pro implementaci uživatelského rozhraní jsem si vytvořil třídu GUI a Button. Třída Button poskytuje předpis pro objekt tlačítka, který obsahuje velikost, x-ovou a y-ovou souřadnici, index v poli Assets pro texturu textu, index pro texturu pozadí a příznak, jestli je tlačítko zakliknuto.

Ve třídě GUI je metoda ButtonClicked, která využívá SDL události a zjišťuje zda-li bylo kliknuto na tlačítko. Testuje jen jestli jsou souřadnice kliku uvnitř tlačítka. Všechna tlačítka jsou uložena v poli Buttons a pokud je hráč v menu, tak se vykreslují pomocí metody DrawMenu, která v cyklu volá metodu DrawButton.

```
1  public static void DrawButton(Button b)
2        {
3           SDL.SDL_RenderCopy(Variables.Renderer, Assets.AssetsArray[b
                .buttonTexture], IntPtr.Zero, ref b.rect);
4
5           var textRect = new SDL.SDL_Rect
6           {
7              x = b.rect.x + (b.rect.w / 5) / 2,
8              y = b.rect.y + (b.rect.h / 8) / 2,
9              h = b.rect.h - b.rect.h / 8,
10             w = b.rect.w - b.rect.w / 5
11          };
12          SDL.SDL_RenderCopy(Variables.Renderer, Assets.AssetsArray[b
                .textTexture], IntPtr.Zero, ref textRect);
13       }
```

Zdrojový kód 21: Metoda DrawButton

3.9 Konfigurační soubor

V enginu se mi začaly množit konstanty, které značným způsobem ovlivňují chování herního světa, proto jsem se rozhodl vytvořit soubor s těmito konstantami, které si bude uživatel moci libovolně nastavit. Soubor obsahuje třídu Variables, kde jsou atributy, které nastavují například sílu gravitačního zrychlení, rychlost hráče, délku metru v pixelech, délka jednoho snímku, pole entit a další.

3.10 Třída Main

V této podkapitole si konečně odhalíme k čemu slouží třída Main. Tato třída vznikla, protože jsem chtěl uživatele odstínit od implementace enginu, aby se mohl soustředit jenom na herní vývoj, postupně jsem ale zjišťoval, že zásah do enginu jako celku bude často nevyhnutelný, ale třídu Main jsem ponechal a představím co v ní je. Ve třídě si uživatel může například nastavovat globální proměnné, které chce používat napříč celým programem. Dále může implementovat i své vlastní entity a komponenty. Hlavně ale obsahuje 8 metod, které popíšu.

První metoda je Init. Do těla metody uživatel vloží vše, co by chtěl, aby se vykonalo ještě před prvním vstupem do herního cyklu. Typicky to bude vytvoření entit a načtení multimediálních souborů.

Další v pořadí je metoda Input, která přijímá 6 argumentů a to pole stisknutých kláves, entitu, uplynutý čas od minulého snímku, příznak jestli je entita na zemi, posun x-ové a y-ové souřadnice. Tato metoda se volá ve fyzikálním enginu, proto jsou jako argument souřadnice, které může uživatel dále upravovat, pokud chce reagovat na vstupy nebo implementovat umělou inteligenci. Takže v této metodě sesbírá všechny vstupy za daný snímek a rozhodne se, co se stane.

CollisionResolve slouží k vlastnímu řešení kolizí, volá se vždy jako první věc ve fyzikálním systému v metodě SolveCollision. Pokud volání CollisionResolve s

argumentem entita a collider vrátí true, znamená to, že uživatel vyřešil kolizi po svém a tělo SolveCollision se neprovede, pokud vrátí false, znamená to, že pro danou kombinaci entity a collideru není implementováno žádné vlastní řešení kolize a tělo se provede.

```
1   if (entity.GetType() == typeof(Player) && collider.GetType() ==
        typeof(Frog))
2           {
3               Variables.Running = false;
4               Variables.Score = 0;
5               Level.LoadLevel(Variables.CurrentLevel);
6               Variables.GameOver = true;
7
8               return true
9           }
```

Zdrojový kód 22: Vyřešení kolize hráče s žábou v metodě CollisionResolve

V této ukázce kódu můžeme vidět, že když hráč koliduje s žábou tak hra končí a body se resetují na nulu.

V metodě Animatons uživatel upravuje animace v jednotlivých snímcích, k dispozici má čas uplynulý od minulého snímku. Například pokud je hráč ve vzduchu a animace pádu je ve sprite sheetu na devatenáctém řádku, tak bude kód vypadat následovně:

```
1   if(!grounded)
2           {
3               sc.spriteRow = 18;
4               sc.spriteCol = (int)(SDL.SDL_GetTicks() / sc.speed) % 6;
5               player.SetSprite(sc);
6           }
```

Zdrojový kód 23: Animace pádu

V další metodě MenuMouseInput může uživatel rozhodovat, co se stane pokud je v menu a klikne na určité tlačítko.

Posledni 2 metody jsou GameLoop a MenuLoop, které se volají v jejich hlavních ekvivalentech a uživatel v nich může napsat cokoli, co chce, aby se vykonávalo při každé iteraci těchto cyklů.

4 Ukázková hra

V poslední kapitole si představíme hru, kterou jsem tvořil postupně s enginem. Jelikož jsem tvořil engine a hru současně, mohl jsem hned na hře otestovat funkčnost nové vyvinuté části enginu. V této hře byly využity všechny části enginu, takže je ideální pro demonstraci.

Jako první si vyjmenujeme všechny entity, které jsem pro hru vytvořil. Tyto entity jsou: Player, Obstacle, Fruit, Frog, Texture. Všechny tyto entity sdílí nějaké komponenty a ty jsou následující: NameComponent, SolidComponent, HitboxComponent, SpriteComponent, MoveableObjectComponent. U entity Obstacle bude navíc ještě komponenta PhysicsObject, aby s ní ostatní entity mohly kolidovat. Potom entity, které se můžou pohybovat v herním světě (Player, Frog, Fruit) mají navíc ještě komponenty VelocityComponent, MassComponent a ForceComponent.

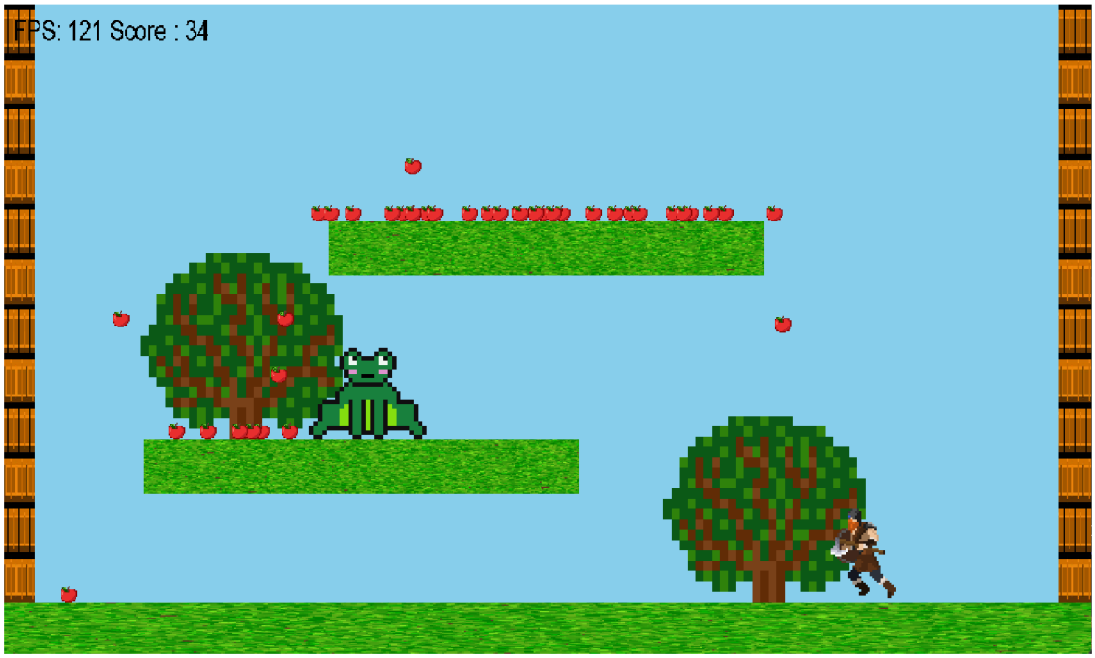
Entita hráče se může libovolně pohybovat po herním světě. Má k dispozici pohyby doleva, doprava a skok. Dále máme v herním světě entitu Fruit, která je vizualizována jako jablko. Pozice jablek je náhodně generována. Pokud se hráč dostane do kolize s jablkem, jablko zmizí, objeví se nové na náhodném místě a hráči se připíše jeden bod. Svět je složen pomocí entity Obstacle. Obstacle používám jako stěny, podlahy a platformy. Výhodou je, že každá instance Obstacle může mít jinou texturu, to platí pro každou entitu. Opakem Obstacle je Texture, která se světem žádným způsobem neinteraguje a pouze se vykresluje její textura. Tato entita slouží pouze pro dekoraci. Jako poslední je entita Frog, což je jedovatá žába, která se ve hře pohybuje směrem k hráči a když se s ním srazí, tak hra končí. Cílem této hry je nasbírat 100 jablek a nedotknout se přitom žáby.

```
1             var frog = EntityManager.GetAll<Frog>()[0];
2             if(frog.Grounded()) {
3                var playerLeft = EntityManager.GetPlayer().GetLeft();
4                var frogLeft = frog.GetLeft();
5                if (playerLeft < frogLeft)
6                {
7                   frog.SetFacing(0);
8                   frog.SetForce(-10, (float)(frog.GetMass() *
                       Variables.GravitationalPull) * -10);
9                } else
10               {
11                  frog.SetFacing(1);
12                  frog.SetForce(10, (float)(frog.GetMass() *
                       Variables.GravitationalPull) * -10);
13               }
14               frog.SetVelocity(40);
15               frog.SetHitbox(0, -1);
16               Audio.PlayEffect((Assets.AssetsArray[JumpSoundIndex])
                    );
17            }
```

Zdrojový kód 24: Pohyb žáby

Když se hra spustí, hráč se dostane do menu a může si vybrat jestli chce hrát, uložit level nebo hru ukončit. Pokud klikne na tlačítko Play, tak program skočí do hlavního cyklu a renderuje se herní svět, po stisknutí klávesy escape se znovu zobrazí menu a hra se pozastaví. Po stisknutí tlačítka Save Level se

uloží aktuální level, který je načten v paměti. Stiskem tlačítka Quit se program ukončí.



Obrázek 2: Ukázková hra

Závěr

Při vývoji tohoto enginu jsem prošel řadou výzev a naučil se nejednu věc o herním vývoji, architektuře enginu a mnoho dalšího. Uvědomil, že i když můj herní engine dosáhl určité úrovně funkčnosti a nabízí základní možnosti pro tvorbu her, stále existuje mnoho oblastí, ve kterých může být vylepšen, optimalizován a rozšířen. Ať už jde o výkonnostní optimalizace, rozšíření podpory pro různé grafické efekty, nebo zlepšení uživatelského rozhraní, možnosti pro další vývoj a inovace jsou prakticky neomezené.

Conclusions

During the development of this engine I went through many challenges and learned a lot about game development, engine architecture and much more. It has made me realize that even though my game engine has reached a certain level of functionality and offers basic capabilities for game development, there are still many areas where it can be improved, optimized and extended. Whether it's performance optimizations, expanding support for various graphical effects, or improving the user interface, the possibilities for further development and innovation are virtually limitless.

5 Obsah elektronických dat

SDL_2_TEST/

- README.txt - informační soubor
- SDL_2_Test.sln- Visual Studio solution

  SDL_2_TEST/

  Main.cs - hlavní soubor, kam uživatel píše kód

  text/
  - adresář se soubory k textu práce

  engine/
  - adresář se soubory, které tvoří jádro enginu

  bin/

  Debug/
  - adresář s debug soubory

  Release/
  - adresář se spustitelným souborem

Bibliografie

[1] NYSTROM Robert. Game programming patterns. Genever Benning, 2014, 456 s. ISBN: 978-0-9905829-2-2.

[2] GREGORY Jason. Game Engine Architecture, Third Edition. Taylor & Francis Ltd, 2018, 1221 s. ISBN: 9781138035454.

[3] SUNG Kelvin, PAVLEAS Jebediah, MUNSON Matthew, PACE Jason. Build your own 2D game engine and create great web games. Apress Media, LLC, 2015, 741 s. ISBN: 978-1-4842-7376-0.