Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Vývoj 2D hry v Unity

Bakalářská práce

Autor: Jakub Lásko
Studijní obor: Aplikovaná informatika

Vedoucí práce: Malý Filip, doc, Ing., Ph. D.

Hradec Králové duben 2018

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 16.4.2018 Jakub Lásko

Poděkování:

Děkuji vedoucímu bakalářské práce Ing. Malý Filip za metodické vedení práce.

Anotace

Bakalářská práce se soustředí na vývoj 2D hry v prostředí Unity a na popis celého procesu spolu se základní funkcionalitu Unity. V práci je prolínána teoretická a praktická část. Máme vždy teorii o nějaké funkcionalitě, kde pak následuje praktická ukázka.

Z pohledu teoretické části je kladen důraz na seznámení uživatele se základními prvky a principy Unity, které se uplatnily v rámci vývoje hry anebo jsou velice často využívány všeobecně.

Praktická část bakalářské práce tvoří popisy jednotlivých funkcí a různé části kódu z naší hry, které bývají přirovnány k obecným postupům v Unity. Návrh grafického rozhraní, implementace algoritmů, tvorba skriptů, vestavěné funkce, tvorba animací a zvukových efektů. Často je zde využit alternativní přístup při vývoji hry v porovnání s běžnými postupy.

Annotation

Title: 2D Game development in Unity

Bachelor thesis focuses on 2D game development in Unity and description of the whole process with basic functionality in Unity. In thesis both theoretical and practical parts are interleaved. We always have a theory about some functionality and then practical example follows.

From theoretical perspective there is emphasis on introducing the user to the basic features and principals in Unity, which were applied during game development or are very often generally used.

Practical part of the thesis is made up by description of individual functions and different parts of code from our game, which are usually compared to the common practices in Unity. Design of graphical interface, algorithm implementation, script making, built-in functions, animation and sound making. Often there is used an alternative approach during the game development compared to common approaches.

Obsah

Seznam obrázků

Seznam kódu

1 Úvod

Jedná se o seznámení s vývojovým prostředím Unity. Ukážeme si, co nám toto prostředí nabízí a jaké máme možnosti při vývoji her. Budeme se zaměřovat na 2D aspekt tvorby s tím, že pokryjeme veškeré kroky, které by se měly vyskytovat při vývoji každé hry.

Cílem teoretické části bakalářské práce je seznámení uživatele s vývojovým prostředím Unity a veškerých jeho základních funkcionalit. Důraz bude kladen na ukázku veškerých způsobů a nástrojů, které by uživatel mohl potřebovat při tvorbě 2D her v Unity. Bude zde popsáno samotné uživatelské prostředí, dále práce s obrázky, skripty, jak na animace, hudbu a také širší popsání kolizí a fyzikálních aspektů vývoje.

V praktické části je hlavním cílem znázornění a samotná implementace poznatků, které jsme si ukázali ve hře. Jedná se o 2D hru vytvořenou pomocí Unity, kde byly demonstrovány veškeré základní techniky, které jsou při vývoji her potřeba. Je zde vše, ať už od implementace uživatelského rozhraní, přes animace, fyziku, kolize a hudbu.

Při tvorbě hry byly využity vlastní praktické znalosti. Zároveň bylo využito zdrojů Live Training, které nabízí Unity. Díky tomuto zdroji je možné si osvojit různé oblasti vývoje her v Unity za pomoci odborníků, kteří živě ukazují řešení. Pro ostatní potřeby byla využita dokumentace, která je zároveň častým zdrojem v práci samotné.

2 Základy Unity z pohledu 2D

Unity je volně dostupný engine pro vývoj her, ale lze ho využít i při projektování a simulacích. Nabízí funkce jak 2D tak 3D a spousty pokročilých technik. Lze získat i verzi Unity Pro, která je placená, ale otevírá nové možnosti jako vlastní shadery a spolupráci více vývojářů na jednom projektu.

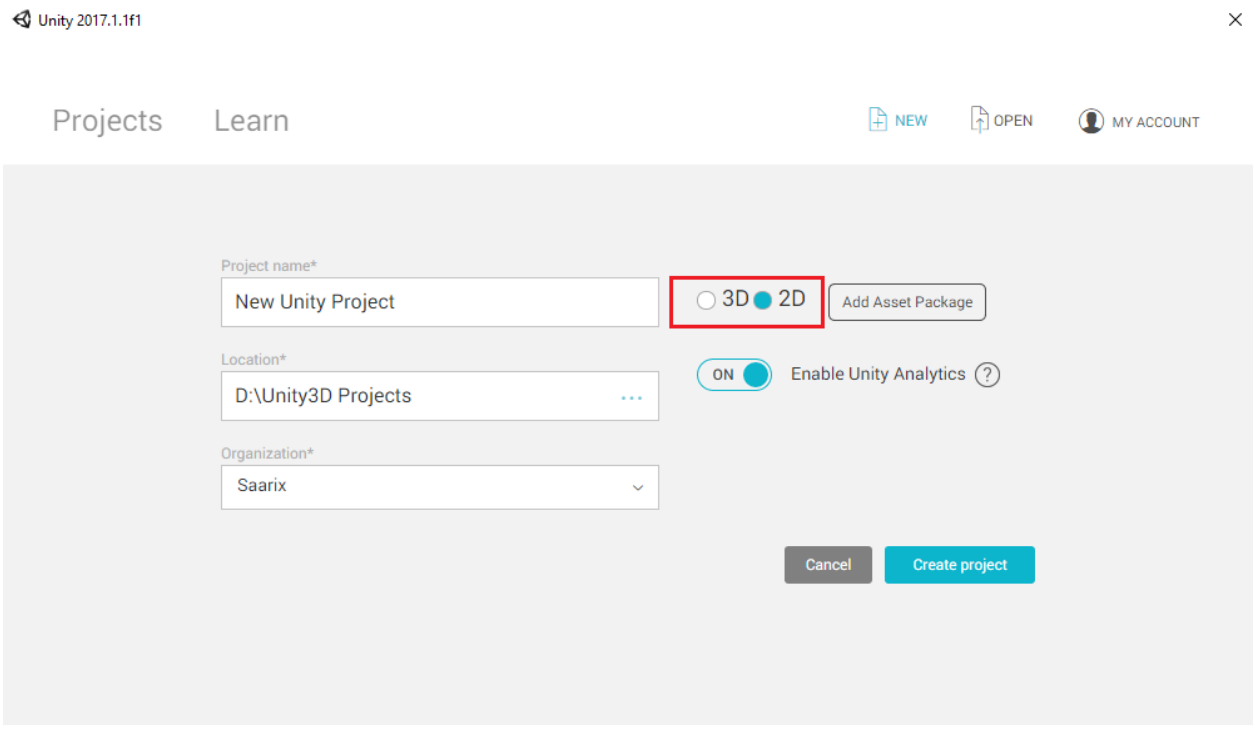
Dále se podíváme na základní aspekty Unity, které využijeme při vývoji her. Například založení projektu, techniky tvorby v 2D, přidání nových objektu a tak dále.

2.1 Nový projekt a nastavení

Unity podporuje vývoj her jak v 3D tak 2D prostoru. Pro náš případ využijeme 2D a proto je nutné projekt řádně nastavit.

Změnu lze provést buď v editoru pro již založený projekt anebo přímo při zakládání nového projektu. Při změně v editoru je třeba jít do okna Editor Settings Inspector, na které se dostanete skrze Edit>Project Settings>Editor menu.

Obrázek 1: Nastavení módu editoru (převzato z https://docs.unity3d.com/Manual/2DAnd3DModeSettings.html)



Obrázek 2: Založení nového projektu (zdrojem autor)

Toto nastavení ať už 2D nebo 3D nám v Unity editoru přímo ovlivní několik nastavení a upraví určité funkcionality. Nastavení, která jsou provedena můžeme vidět níže.

„V 2D módu projektu:

- *Jakékoliv obrázky, které importujete jsou považovány za 2D (Sprites) a jsou nastaveny na Sprite mód.*
- *Sprite Packer je povolen.*
- *Scene View je nastaven na 2D.*
- *Výchozí herní objekt nemá přímé osvětlení v reálném čase.*
- *Výchozí pozice kamery je 0,0,-10. (Což je 0,1,-10 v 3D módu.)*
- *Kamera je nastavena na Orthographic. (V 3D je nastavena na Prespective.)*
- *V okně Osvětlení:*
    - *Skybox je vypnut pro nové scény.*
    - *Ambient Source je nastaven na Color. (Spolu s barvou nastavenou na tmavě šedou: RGB: 54,58,66.)*
    - *Precomputed Realtime GI je vypnuto.*

- *Baked GI je vypnuto.*
- *Auto-Building je vypnuto.“*

„V 3D módu projektu:

- *Jakékoliv obrázky, které importujete nejsou považovány za 2D (Sprites).*
- *Sprite Packer je vypnut.*
- *Scene View je nastaven na 3D.*
- *Výchozí herní objekt má přímé osvětlení v reálném čase.*
- *Výchozí pozice kamery je 0,1,-10. (Což je 0,0,-10 v 2D módu.)*
- *Kamera je nastavena na Prespective. (V 2D je nastavena na Orthographics.)*
- *V okně Osvětlení*
  - *Skybox je přednastavený výchozí Skybox Material.*
  - *Ambient Source je nastaven na Skybox.*
  - *Precomputed Realtime GI je zapnuto.*
  - *Baked GI je zapnuto.*
  - *Auto-Building je zapnuto.*“ (převzato z [1], volně přeloženo)

2.2 Scene view a Game view

V editoru se vždy doporučuje mít obě okna zobrazená vedle sebe v dostatečné velikosti, abychom mohli přehledně pracovat na projektu. Nyní si popíšeme, k čemu okna slouží.

Scene view – *„Je místo, kde strávíme nejvíce času při práci na hře. Budeme zde stavět a manipulovat s našimi herními úrovněmi. Některé z akcí, které se zde dělají jsou například přidání a modifikace charakterů, osvětlení, kamery a ostatní objekty spojené s hrou“* (převzato z [2], volně přeloženo)

Game view – *„Umožnuje nám náhled na hru mezitím co na ni pracujeme v editoru. Neupravujeme zde hru, ale pouze se na ni díváme.*

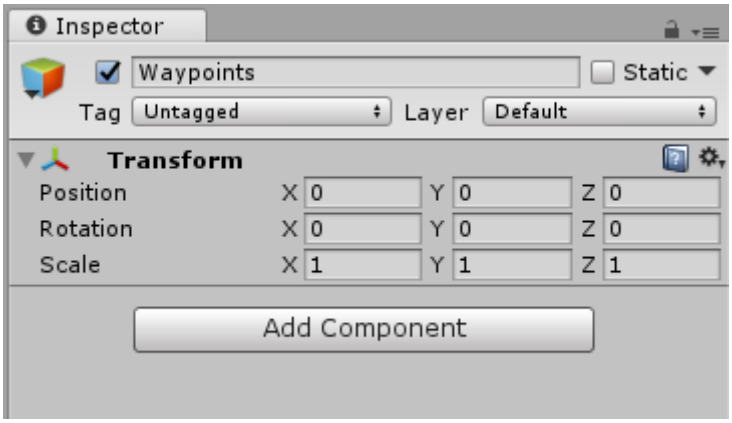
Můžeme hru také spustit. Ke spuštění herního módu, stačí kliknout na tlačítko play, které se nachází nahoře uprostřed v nabídce.

Lze provádět změny na hře i pokud jsme v herním módu. Ale všechny změny, které uděláme, tak jsou pouze dočasné a po vypnutí herního módu se automaticky vrátí zpět.“ (převzato z [2], volně přeloženo)

2.3 Založení herního objektu

Nový herní objekt lze v Unity přidat velice intuitivně. Stačí jít v horním menu GameObject>Create Empty. Tímto vytvoříme nový prázdný herní objekt. Nyní po pravé straně můžete v Inspektoru vidět jednotlivé položky objektu. Jsou zde vidět jednotlivé skripty přirazené k objektu, jeho `Transform` komponenta, která určuje pozici objektu v prostoru, a tak dále.

Přidat nové komponenty k objektu lze jednoduše jejich přetažením do Inspektoru vybraného objektu, nebo pomocí tlačítka Add Component.



Obrázek 3: Náhled inspektoru na objekt ve scéně (zdrojem autor)

Další zajímavostí je vlastnost `Tag`, kterou má každý objekt. Skrze `Tag` se dají velice dobře rozlišovat určité typy, nebo seskupení objektů. `Tag` si můžete vytvořit jakýkoliv, je to vlastně pouze textové označení objektu, skrze které pak můžeme objekt či objekty najít.

Slouží k tomu následující metoda: `GameObject.FindGameObjectsWithTag`(string tag). Jak zde můžeme vidět, tak se jedná o statickou metodu dostupnou na jakémkoliv objektu typu `GameObject`, takže ji můžeme volat z jakéhokoliv skriptu, který dědí od `MonoBehaviour` třídy. Za zmínku také stojí návratový typ metody: `GameObject[]`. [3]

2.4 Assets

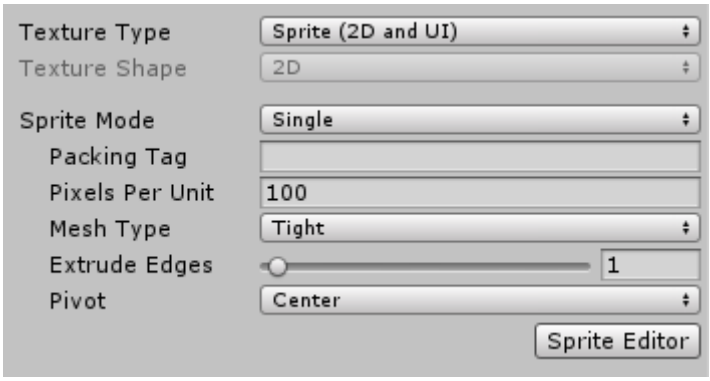
Základ každého Unity projektu. Jedná se o řadu souborů jako například textury ve formě obrázků, modely a zvukové stopy.
Při vytvoření nového projektu v Unity bude vždy vygenerována složka `Assets`, kam následně budeme vkládat veškeré soubory týkající se hry. Veškeré soubory v této složce jsou během kompilace zabaleny do souborů s příponou .assets [4]

2.5 Sprite – základní 2D objekt

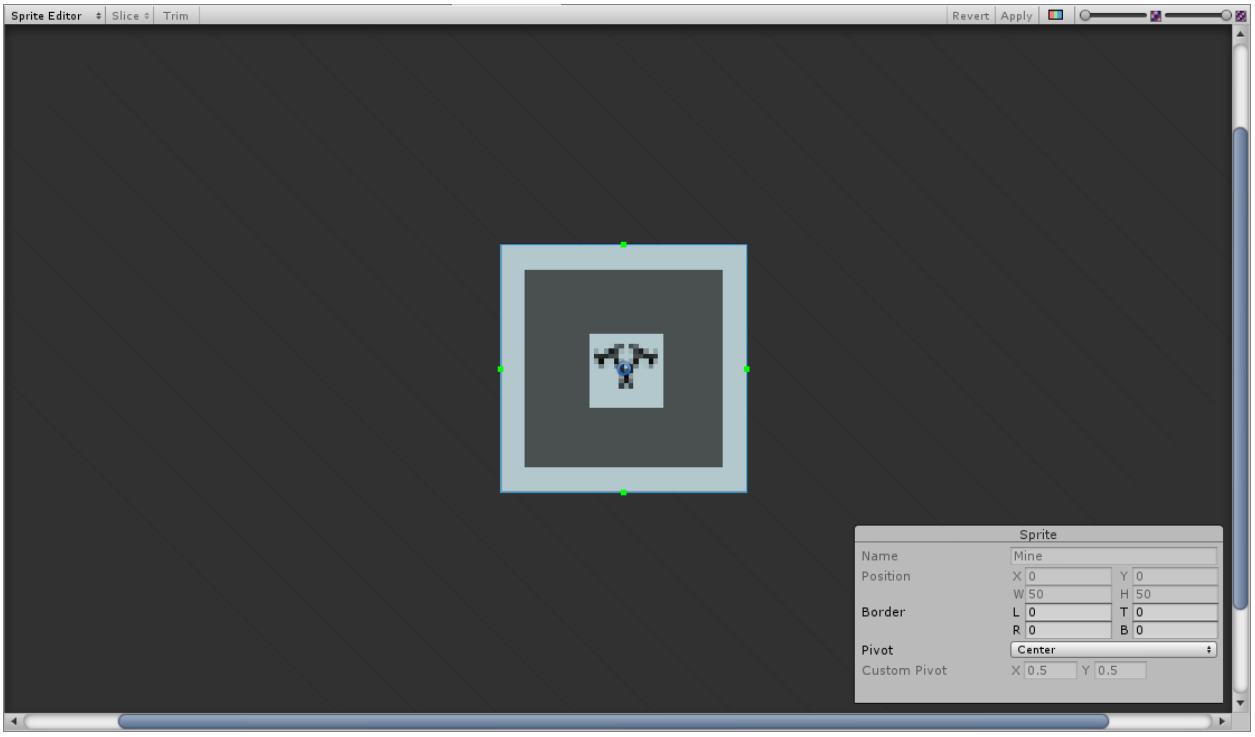
`Sprite` je grafický 2D objekt. Je to ten nejzákladnější objekt, který při vývoji 2D hry využijeme. Unity nám poskytuje hned několik nástrojů pro práci se `Sprite` objekty. Sprite Creator je první nástroj, který nám umožňuje vytvořit základní tvary jako Sprite, které můžeme využít jako placeholder v naší hře. Podporované tvary jsou:

- Čtverec
- Trojúhelník
- Diamant
- Hexagon
- Kruh
- Polygon

Sprite Editor je nejzásadnější a poskytuje nám možnost jak upravovat obrázek. Umožňuje nastavení pivotů, změnu velikosti a jiné funkce. Další velice užitečnou věcí je možnost nastavení sekce pro opakování, které se uplatní, pokud obrázek zvětšíme nad základní rozměr. Ostatní funkce např. automatické ořezávání, mřížkové rozdělení na jednotlivé segmenty, automatické rozdělení podle barev. [5]

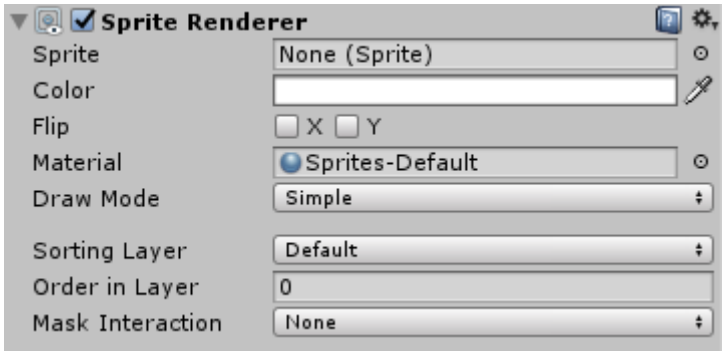


Obrázek 4: Náhled inspektoru při výběru obrázku (zdrojem autor)



Obrázek 5: Sprite Editor (zdrojem autor)

Sprite Renderer je vlastně ona komponenta, kterou vkládáme do čistého objektu, pokud chceme, aby ve scéně vykresloval náš 2D obrázek (`Sprite`). Máme k dispozici hned několik vlastností, které zde můžeme nastavit, jak můžeme vidět na obrázku níže.



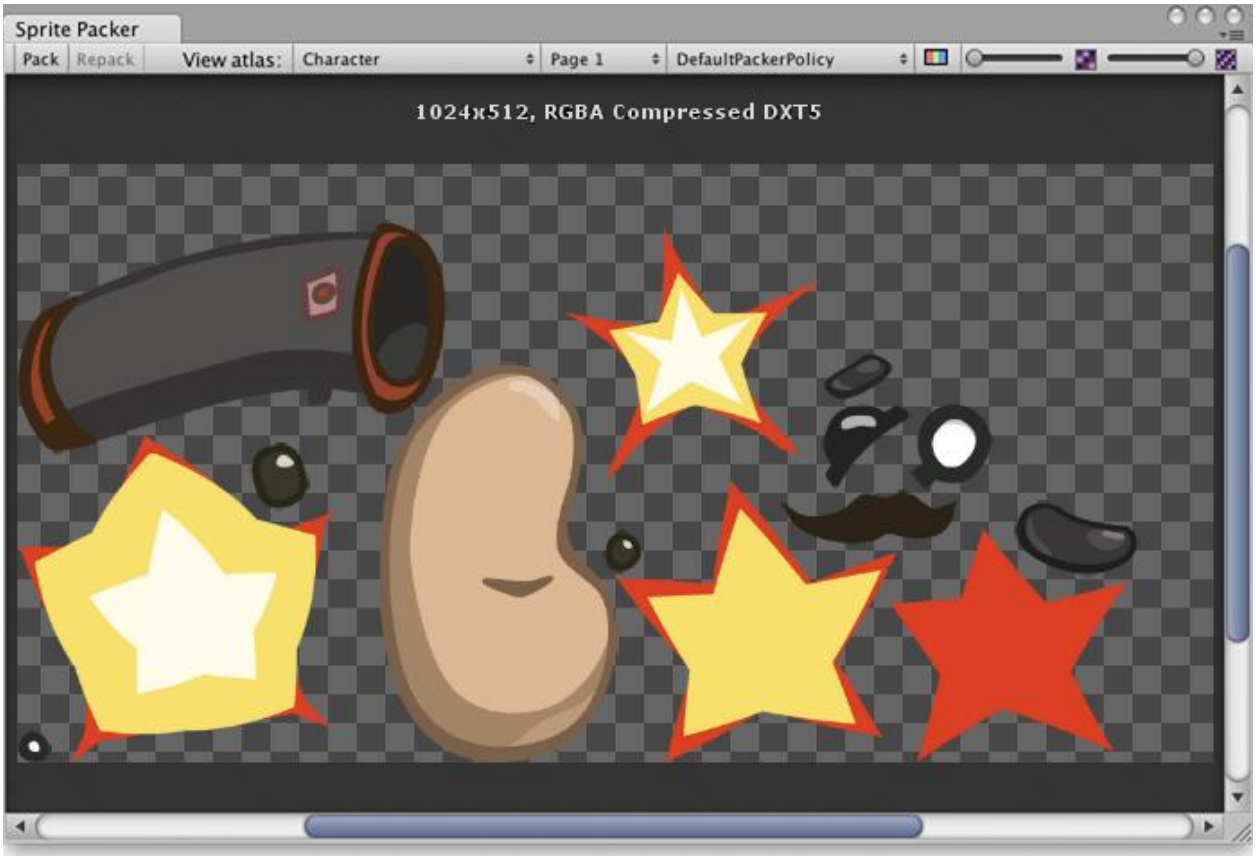
Obrázek 6: Sprite Renderer (zdrojem autor)

Sprite a Color vlastnosti není třeba vysvětlovat, ale bude nás zajímat Sorting Layer a Order in Layer vlastnost, které si popíšeme dále.

Sprite Packer je poslední nástroj, který je vhodné zmínit. Tento nástroj využijeme hlavně, když budeme chtít optimalizovat hru a slouží k tomu, abychom seskupili více Spritů do jednoho velkého.

„Když vytváříme sprite grafiku, tak je vhodné pracovat se separátním texturovým souborem pro každý charakter. Nicméně značná část sprite textury často obsahuje prázdné místo mezi jednotlivými elementy, a to pak plýtvá video pamětí při běhu. Pro optimální výkon je nejlepší seskupovat grafiku do několika sprite textur pevně k sobě uvnitř jedné textury známé jako atlas. Unity poskytuje Sprite Packer utilitu k automatizaci toho procesu, který generuje atlasy z jednotlivých sprite textur.“ (převzato z [4], volně přeloženo)

Sprite Packer je základně vypnutý, ale můžeme ho nastavit z Editor Settings (menu: Edit>Project Settings>Editor). Mód může být změněn z Disabled na Enabled for Builds (to znamená, že se seskupovaní neděje v Play módu, ale pouze při kompilaci), nebo Always enabled (seskupování je povoleno pro Play mód i kompilaci) [6]



Obrázek 7: Sprite Packer (převzato z https://docs.unity3d.com/Manual/SpritePacker.html)

2.6 Řazení obrázků ve scéně

Jelikož se pohybujeme v 2D prostoru, tak zde máme pouze X, Y souřadnice, to ale není v Unity tak úplně pravda a pokud byste si přepnuli Scene View do 3D módu, tak byste mohli vidět, že všechny obrázky mají stejnou souřadnici Z, což vypadá tak, že jednotlivé obrázky se překrývají.

Řazení dle jistých pravidel nám zajišťuje Sorting Layer a poté číselný údaj Order in Layer.

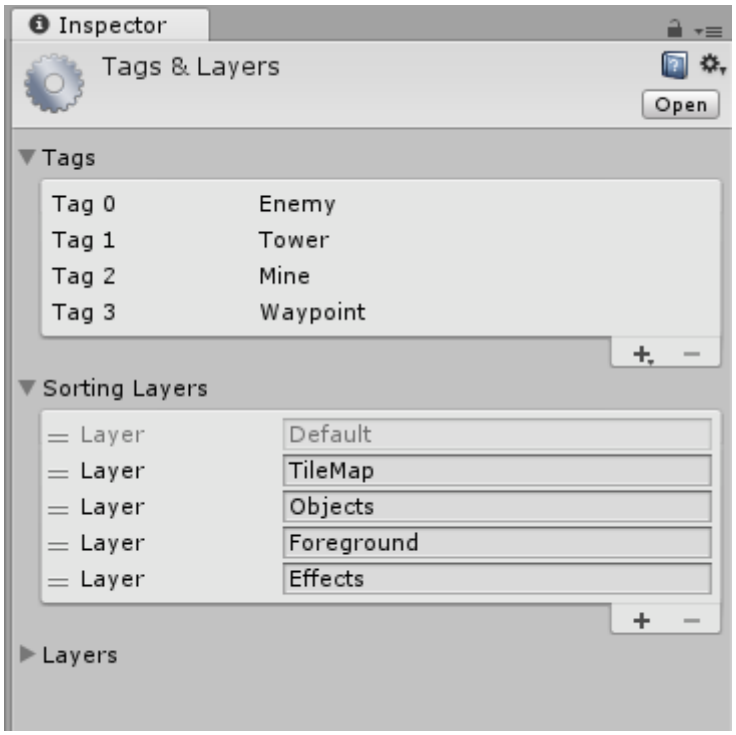
Základního rozdělení dosáhneme tím, že si v editoru nadefinujeme vlastní Sorting Layers, které si pojmenujeme tak, aby bylo z prvního pohledu jasné, které objekty do nich spadají, to by nám ale nemuselo stačit, a proto je zde další vlastnost.

Order in Layer definuje podřadné řazení již v rámci dané Sorting Layer, zde lze zadat jak záporné, tak kladné číslo a řazení pak probíhá od nejmenšího čísla k největšímu.

Někdy je ale třeba sáhnout pro řazení pomocí samotného skriptu. Můžeme řazení ovládat pomocí skriptu pro každou kameru zvlášť. K tomu je potřeba nastavit 2 proměnné u kamery, viz kód níže.

```
var camera = GetComponent<Camera>();
camera.transparencySortMode = TransparencySortMode.CustomAxis;
camera.transparencySortAxis = new Vector3(0.0f, 1.0f, 0.0f);
```

Kód 1: Změna řazení při vykreslování (převzato z https://docs.unity3d.com/Manual/Sprites.html)



Obrázek 8: Náhled inspektoru na nastavení tagů a vrstev (zdrojem autor)

3 Skriptování, tvorba UI

Zřejmě nejvyužívanější věcí v Unity jsou skripty. Jsou potřebné k tomu, pokud chceme objektu přidat vlastní chování. V rámci inspektoru lze objektu přiřadit jakékoliv množství skriptů, které pak bude využívat.

Defaultně každý skript dědí ze třídy `MonoBehaviour`, která implementuje základní herní smyčku. Herní smyčka se skládá z následujících kroků (uvedená smyčka je zjednodušená):

„Initialization

- *Awake*
- *Start*

Physics

- *FixedUpdate*
- *OnTrigger*
- *OnCollision*

Input events

- *OnMouse*

Game logic

- *Update*
- *LateUpdate*

GUI rendering

- *OnGUI*

Decommissioning

- *OnApplicationQuit*
- *OnDestroy“* (převzato z [7], volně přeloženo)

3.1 Tvorba úvodního menu

Nyní si vytvoříme hlavní menu pro naši hru. Budeme pro to potřebovat novou scénu, kterou pojmenujeme „Menu“. Jako první by se mělo objevit navigační menu s určitými standartními tlačítky (např. New Game, Options, Quit). V naší hře jsme pro navigační menu zvolili pouze tlačítka New Game a Quit.

Od Unity verze 5.6 prošla tvorba UI ohromnou změnou, nyní si níže popíšeme postup tvorby UI s jednotlivými prvky a ukázkami.

3.1.1 Canvas

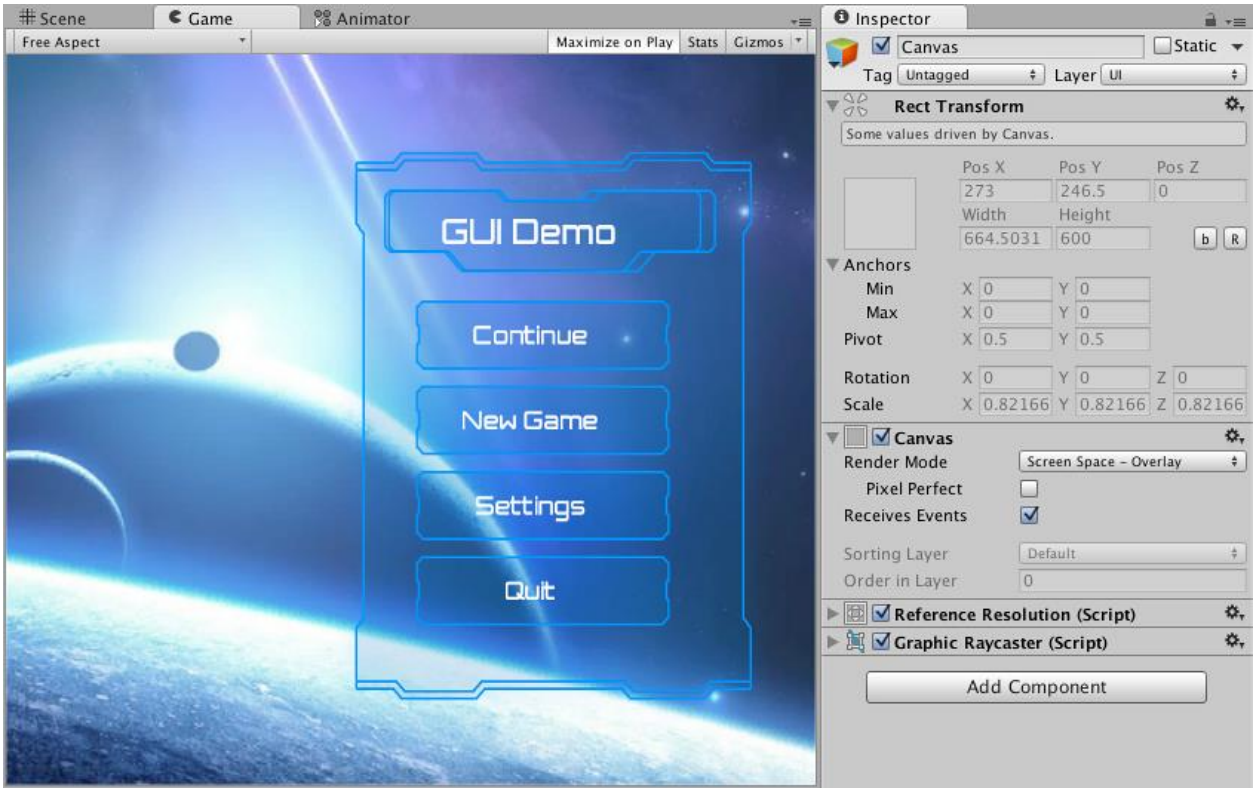
„Canvas je oblast uvnitř které by měli být veškeré UI elementy. Canvas je herní objekt s Canvas komponentou na něm a všechny UI elementy musí být potomky takového Canvasu.

Vytvořením nového UI elementu, jako je Image použitím menu GameObject>UI>Image, se automaticky vytvoří Canvas, pokud tady už nějaký Canvas není ve scéně. UI element je vytvořen jako potomek tohoto Canvasu.

Canvas oblast je zobrazena jako obdélník v Scene View. Toto umožňuje lehké pozicování UI elementů bez potřeby mít Game View neustále viditelný.

Canvas využívá EvenSystem objekt, aby pomohl Messanging Systému.

UI elementy v Canvasu jsou vykreslovány ve stejném pořadí jako se objeví v Hierarchy. První potomek je vykreslen první, druhý jako další a tak dále. Pokud se dva UI elementy překrývají, ten pozdější se objeví nad tím dřívějším.“ (převzato z [8], volně přeloženo)



Obrázek 9: UI Canvas (převzato z https://docs.unity3d.com/Manual/UICanvas.html)

3.1.2 Jednotlivé komponenty

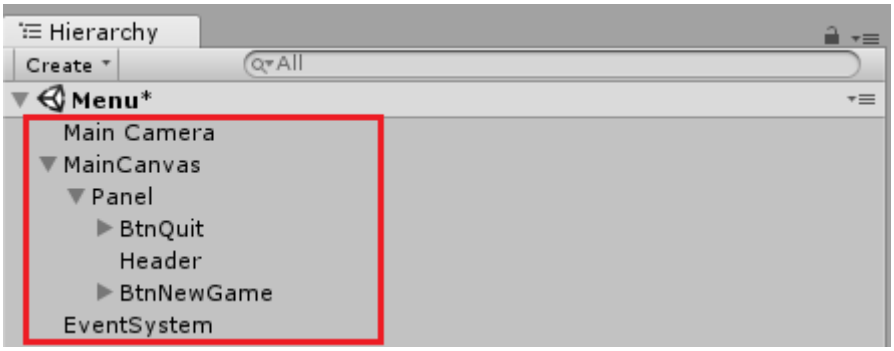
Unity nabízí poměrně velké množství základních UI komponent, které můžeme ve hrách využít. Níže si uvedeme výčet oněch komponent. Ke komponentám se dostaneme skrze menu: GameObject>UI

- Text
- Image
- Raw Image
- Button
- Toggle
- Slider
- Scrollbar
- Dropdown
- Input Field
- Canvas
- Panel
- Scroll View
- Event Systém

Podrobným popisem každé komponenty se zde zabývat nebudeme, protože většina z nich je velice intuitivních.

3.1.3 Hlavní menu

Nyní si do scény přidáme nový `Canvas` a pojmenujeme ho „MainCanvas“, dále si přidáme `Panel`, který bude jako potomek komponenty `Canvas`.
Do panelu nyní přidáme text s názvem „Header“ a 2 tlačítka („New Game“ a „Quit“). Výsledek scény by měl vypadat jako na obrázku 10.



Obrázek 10: Hierarchie objektů ve scéně (zdrojem autor)

3.1.4 Logika menu

Pokud teď zkusíme kliknout na tlačítka, tak nic nebudou dělat. Proto si vytvoříme jednoduchý skript, skrze který budeme ovládat menu a propagovat zde metody, které budou tlačítka využívat.

Skript můžeme pojmenovat např. „MenuBehaviour“ což je běžná konvence pojmenovávání v Unity, pokud se jedná o skript, který řídí chování, tak se na konec píše Behaviour. Níže můžete vidět příkladový skript pro jednoduché menu.

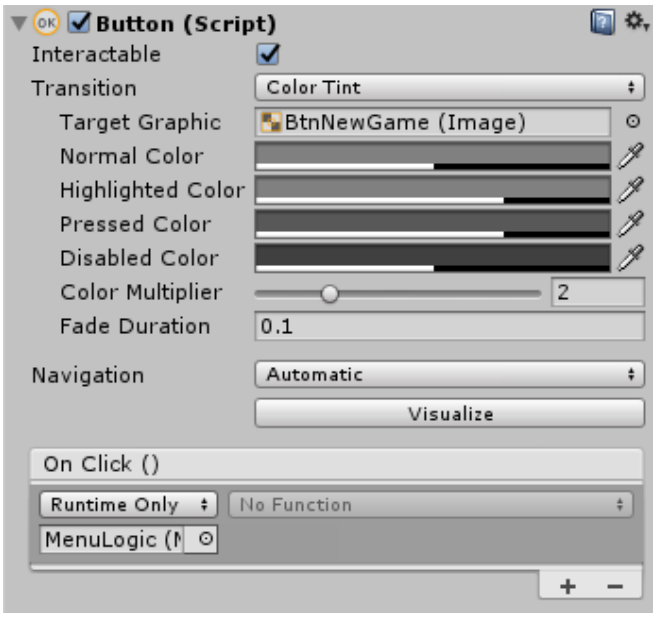
```
public class MenuBehaviour : MonoBehaviour
{
    private Canvas mainMenu;

    void Start()
    {
        mainMenu = GameObject.Find("MainCanvas").GetComponent<Canvas>();
    }

    public void StartGame()
    {
        SceneManager.LoadScene("Map");
    }

    public void Quit()
    {
        #if UNITY_EDITOR
        UnityEditor.EditorApplication.isPlaying = false;
        #endif

        Application.Quit();
    }
}
```

Kód 2: Základní chování pro menu (zdrojem autor)

Můžeme zde vidět, že při načtení tohoto skriptu si vytvoříme referenci na komponentu `Canvas`, která se nachází na objektu pojmenovaném „MainCanvas“. Dále 2 metody, které napojíme na tlačítka, jednu pro načtení nové scény, která obsahuje samotnou hru a druhá pro vypnutí aplikace nebo přehrávání v editoru.

Napojení metody na tlačítko není nijak složité. Stačí v okně Hierarchy najít požadované tlačítko a vybrat ho. Nyní v inspektoru uvidíme komponentu Button a dole je pole `On Click()`, což je vlastně list akcí, který se mají stát po kliknutí na tlačítko. Stačí kliknout na tlačítko „+“ a poté v poli `GameObject` vybrat herní objekt, který má na sobě napojen onen skript MenuBehaviour, nejlepší bude mít ho na objektu, kde je `Canvas`. A jako poslední krok vybereme metodu, která se má provést,

vybereme na objektu požadovaný skript, který metodu obsahuje a pak metodu samotnou.



Obrázek 11: Button skript (zdrojem autor)

3.1.4 Permanentní herní objekt

Často se v Unity setkáme se situací, že potřebujeme předat nějaká data mezi scénami, nebo si udržet určitá data na jednom místě po celou dobu hry. To je případ, kdy se nám nejvíce bude hodit vytvořit objekt, který se nesmaže při načtení jiné scény, ale přenese se do oné nové scény.

V našem projektu takovýto objekt využíváme již v menu, kde se nastavují vstupní proměnné pro hru, které určí její obtížnost a tak dále. Základní konvence je, že takovýto skript by měl být na samotném prázdném herním objektu, který pojmenujeme s „_“ na začátku, abychom jasně odlišili, že se jedná o permanentní objekt.

Permanentní objekt vytvoříme tak, že v metodě `Start()` zavoláme metodu `DontDestroyOnLoad(Object target)` a jako parametr nastavíme objekt, ke kterému je skript přiřazen, viz. ukázka.

```
public class GameInfo : MonoBehaviour
{
    void Start ()
    {
        DontDestroyOnLoad(gameObject);
    }
}
```

Kód 3: Zamezení destrukce objektu (zdrojem autor)

`gameObject` je vlastnost, která je definována ve třídě `MonoBehaviour` a odkazuje právě na herní objekt, ke kterému je skript přiřazen.

Pro podrobnější pochopení toho, co vše za vlastnosti a metody třída `GameObject` poskytuje se můžeme podívat dokumentaci Unity.

3.2 Základní skript a metoda Instantiate

Níže si popíšeme, jak vypadá skript po vytvoření a rozebereme si jednotlivé důležité metody skriptů, které budeme využívat.

Také zmíníme metodu `Instantiate`, které se využívá k tvorbě nových objektů, ale je část využívána specifickými způsoby kvůli výkonové náročnosti, které je při prostém volání metody.

3.2.1 Základní skript a dostupné metody

Jakmile v Unity vytvoříme nový skript, tak bude automaticky dědit ze třídy `MonoBehaviour`, což je základní třída pro všechny skripty v Unity.

Tato třída nám poskytuje spousty metody a vlastností, které máme k dispozici. Zde si vyjmenujeme pouze ty nejzákladnější, pro podrobný seznam se nachází v dokumentaci. [9]

Metody

- Start()
- Awake()
- Update()
- Destroy()
- Instantiate()

Vlastnosti

- enabled
- gameObject
- tag
- transform
- name

Metoda Start

„Start se zavolá na frame, pokud je skript enabled těsně předtím, než jakákoliv z metod Update je poprvé zavolána.

Stejně jako Awake metoda, Start je zavolána přesně jednou v životnosti skriptu. Nicméně, Awake je zavolána, když je skript objekt inicializován, nehledě na to, zda je skript enabled. Start nemůže být zavolán na stejném frame jako Awake, pokud skript není enabled v době inicializace.“ (převzato z [10], volně přeloženo)

```
public class PreviewScript : MonoBehaviour
{
    private GameObject player;

    void Start()
    {
        player = GameObject.Find("Player");
    }
}
```

Kód 4: Ukázka metody Start (zdrojem autor)

Metoda Awake

„Awake je zavolána, když je instance skriptu načítána.

Awake je využíván k inicializaci jakýchkoliv proměnných, nebo herních stavů, než začne hra. Awake je zavolána pouze jednou za dobu životnosti instance skriptu. Awake je zavolána, jakmile všechny objekty jsou inicializovány, takže můžeme bezpečně komunikovat s ostatními objekty.

Poznámka pro C# uživatele: používejte Awake místo konstruktoru pro inicializaci, protože serializovaný stav komponent je nedefinovaný během času konstruktoru.“ (převzato z [11], volně přeloženo)

```
public class PreviewScript : MonoBehaviour
{
    private GameObject player;

    void Awake()
    {
        player = GameObject.Find("Player");
    }
}
```

Kód 5: Ukázka metody Awake (zdrojem autor)

Metoda Update

„Update je zavolána každý frame, pokud je MonoBehaviour enabled.

Update je nejčastěji využívaná metoda pro implementaci jakékoliv herní logiky.“

(převzato z [12], volně přeloženo)

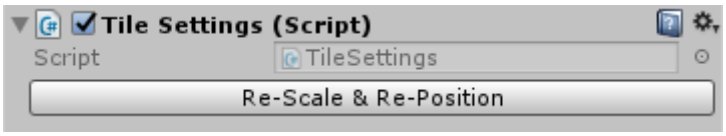
```
public class PreviewScript : MonoBehaviour
{
    void Update()
    {
        transform.Translate(Time.deltaTime * 1, 0, 0);
    }
}
```

Kód 6: Ukázka metody Update (zdrojem autor)

3.2.2 Vlastní properties v editoru

Nyní si vysvětlíme, jak si v editoru v rámci nějakého skriptu propagovat naše vlastní properties a ostatní prvky. Tento přístup se hodí například v situacích, kdy potřebujeme často měnit nějaké aspekty scény a chceme vidět změny přímo v editoru bez potřeby vždy hru spouštět.

Pro názornost si ukážeme jednoduchý příklad tohoto přístupu, který jsme použili v naší hře. Výsledný náhled skriptu v inspektoru můžeme vidět na obrázku níže.

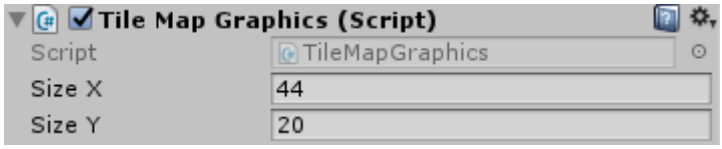


Obrázek 12: Button v editoru (zdrojem autor)

Zde jsme potřebovali na skriptu zobrazit tlačítko v rámci editoru a jak můžeme vidět, tak se zobrazí hned pod umístěním skriptu. Samozřejmě, jak napovídá nadpis, tak lze zobrazit i všechny ostatní properties ať už primitivního nebo vlastního typu. Pro propagaci běžných properties stačí, pokud jim nastavíme viditelnost public, ale pozor nemůžeme zde použít gettery a settery tak jak je známe z C#. Užití můžeme vidět na příkladu níže.

```
public int sizeX = 100;
public int sizeY = 50;
public TileMapData Map { get; set; }
```

Kód 7: Veřejné vlastnosti (zdrojem autor)



Obrázek 13: Properties v editoru (zdrojem autor)

Jak můžeme vidět, tak properties sizeX a sizeY jsou v inspektoru viditelné, ale Map nikoliv. Toto má velice dobré využití v případě, že chceme část veřejných properties vidět v inspektoru, ale zbytek pouze v rámci kódu na instanci třídy, tak toho docílíme snadno pomocí definováním getteru/setteru.

Nyní k příkladu, jak zobrazit tlačítko v rámci skriptu. Není to tak, jak by si mohla většina myslet, budeme totiž muset logiku pro zobrazení tlačítka naprogramovat v jiném skriptu než v samotném `TileSettings`.

Existuje zde konvence pojmenovací, že se vezme název existujícího skriptu, u kterého chceme zobrazovat a na konec přidáme Inspector, takže náš nový skript se bude jmenovat `TileSettingsInspector`. Teď si ukážeme kód a popíšeme základní prvky.

```
[CustomEditor(typeof(TileSettings))]
public class TileSettingsInspector : Editor
{
    public override void OnInspectorGUI()
    {
        base.OnInspectorGUI();

        if (GUILayout.Button("Re-Scale & Re-Position"))
        {
            TileSettings tile = (TileSettings)target;
            tile.SetUp();
        }
    }
}
```

Kód 8: Vlastní náhled v editoru (zdrojem autor)

Jak můžeme vidět, tak skript dědí ze třídy `Editor` a ne `MonoBehaviour`, ale nejpodstatnější věc je ta anotace `[CustomEditor()]`, která určuje, pro který typ tento editor bude, tím tedy určíme, že tato logika se zobrazí v inspektoru u skriptu `TileSettings`.

Poté si přepíšeme logiku metody `OnInspectorGUI()`, která řeší, co se v inspektoru vykreslí a zde pomocí programového volání nadefinujeme tlačítko a uvnitř

podmínky určíme co se stane po kliku. V našem případe zavoláme metodu SetUp() na cílovém skriptu.

Díky takovýmto voláním si můžeme například testovat generování map přímo ve scéně bez potřeby hru spouštět.

3.2.3 Metoda Instantiate

Naklonuje originální objekt, který předáme jako parametr a vrátí nám klon.

„Tato funkce dělá kopii objektu podobně jako Duplicate command v editoru. Pokud klonujeme GameObject, pak můžeme také specifikovat jeho pozici a rotaci (jinak jsou použity originální pozice a rotace z GameObject). Pokud klonujeme Component, pak ten GameObject ke kterému je přidělena bude také naklonován, opět s volitelnou pozicí a rotací.

Když klonujeme GameObject nebo Component, všechny podražené objekty a komponenty (children) budou také naklonovány spolu s jejich vlastnostmi nastavenými jako v originálním objektu.

Aktivní status herního objektu v čase klonování bude předán, takže pokud je originál neaktivní, pak bude klon vyroben v neaktivním stavu.

Instantiate je nejčastěji používáno k vytváření projektilů, AI nepřátel, explozí projektilů nebo nahrazení zničených objektů. Příklad instancování projektilů můžeme vidět níže.“ (převzato z [13], volně přeloženo)

```
public class ExampleClass : MonoBehaviour
{
    public Rigidbody projectile;
    void Update()
    {
        if (Input.GetButtonDown("Fire1"))
        {
            Rigidbody clone;
            clone = Instantiate(projectile, transform.position,
                transform.rotation) as Rigidbody;
            clone.velocity = transform.TransformDirection(Vector3.forward * 10);
        }
    }
}
```

Kód 9: Ukázka tvorby objektů (převzato z https://docs.unity3d.com/ScriptReference/Object.Instantiate.html)

Zde si ale musíme dávat pozor, protože obecně využití `Instantiate()` je velice výkonově náročné. Pokud bychom si představili hru, kde například generujeme

herní pole o velikosti 100x100, tak by se to mohlo stávat relativně pomalým procesem, proto se v takových případech využívají jiné přístupy.

Jedním z těchto přístupů je object pooling. Metoda, při které si ve hře předinstancujeme optimální počet objektů ve hře, které budeme využívat. Například si předem nainstancujeme 20 meteoritů. Díky tomu za běhu hry už nebude třeba instancovat a poté už jen nastavujeme meteory do původního stavu a jakmile splní svou funkci, například vyletí z obrazovky, tak jim opět resetujeme stav do výchozí podoby a zařadíme je do fronty na znovupoužití. To by mohl řešit například nějaký generátor, který jim bude nastavovat velocity a tím je „vystřelí“ do obrazovky.

Object pooling je tedy dobrý v případě častého měnění herních stavů, ale v případě, když máme například hru z žánru Tower Defese, kde můžeme generovat herní pole 100x100, jak jsem zmínil výše. Tak je třeba zvolit jiný přístup.

V naší hře jsme zvolili přístup, kde místo toho abychom využili `Instantiate` a nagenerovali tak 10 000 prefabů (kopií objektů z předlohy), tak jsme sáhli po programovém generovaný textury a vytvoření vlastní struktury pro udržení informací o jednotlivých dlaždicích na textuře.

Nejprve je třeba vytvořit objekt, který bude reprezentovat jednotlivé herní dlaždice vygenerované na textuře (mapě), také musí mít rozlišené typy, zda se jedná o cestu, volné místo, nebo obsazené. Zde využijeme i `Enum` jak můžeme vidět na skriptech níže.

```
public class Tile
{
    public Vector2 Position { get; private set; }

    private bool canBuildOn;
    public bool CanBuildOn { get { return canBuildOn; } }
    public bool IsOccupied { get; set; }

    private eTileType.Texture type;
    public eTileType.Texture Type
    {
        get { return type; }
        set
        {
              type = value;
              switch (type)
              {
                    case eTileType.Texture.Empty:
                          canBuildOn = true;
                          break;
                    case eTileType.Texture.Tree:
                          canBuildOn = false;
```

```
                        break;
                    case eTileType.Texture.GoldVein:
                        canBuildOn = true;
                        break;
                    case eTileType.Texture.Path:
                        canBuildOn = false;
                        break;
                    case eTileType.Texture.Obstacle:
                        canBuildOn = false;
                        break;
                    case eTileType.Texture.Ruins:
                        canBuildOn = true;
                        break;
            }
        }
    }

    public Tile(Vector2 position)
    {
        Position   = position;
        Type       = eTileType.Texture.Empty;
        IsOccupied = false;
    }

    public Tile(Vector2 position, eTileType.Texture tileType)
    {
        Position   = position;
        Type       = tileType;
        IsOccupied = false;
    }
}
```

Kód 10: Třida reprezentující dlaždici ve hře (zdrojem autor)

Zde použitý `Enum` obsahuje pouze výčet možných dlaždic co se mohou vygenerovat na herní ploše, proto ho zde nebudeme dále popisovat.

Jak můžeme vidět, tak si v tomto objektu držíme jeho pozici v souřadnicích x,y, zda zde lze stavět a jestli je dlaždice již obsazena (něco na ní stojí).

Hlavní logiku nám obslouží třídy `TileMapData` a `TileMapGraphics`. Nejprve si vytvoříme programovou strukturu mapy v `TileMapData` za pomoci našich Tile objektů a poté ve třídě `TileMapGraphics` proběhne samotné vytvoření textury na základě map data.

```
public List<Vector2> WaypointPositions { get; private set; }
public float TileSize { get; private set; }

private List<Tile> tiles;
private int sizeX, sizeY;
private SpawnerBehaviour spawner;

public TileMapData(int sizeX, int sizeY)
{
    tiles = new List<Tile>();
```

```
    WaypointPositions = new List<Vector2>();
    this.sizeX = sizeX;
    this.sizeY = sizeY;

    TileSize =
       GameObject.Find("TileMap").GetComponent<TileMapGraphics>().tileSize;
    spawner = GameObject.Find("Spawner").GetComponent<SpawnerBehaviour>();

    Debug.Log("Generating Tiles...");
    GenerateTiles();

    Debug.Log("Generating Path...");
    GeneratePath();
}

public Tile GetTile(int x, int y)
{
    return tiles.Find(t => t.Position == new Vector2(x, y));
}

public List<Tile> GetTiles(eTileType.Texture type)
{
    return tiles.FindAll(x => x.Type == type);
}

public Tile FindTile(Predicate<Tile> match)
{
    return tiles.Find(match);
}
```

Kód 11: Základní vlastnosti a metody třídy TileMapData (zdrojem autor)

Toto je základní tělo třídy `TileMapData`, máme zde konstruktor a metody pro vyhledání jednotlivých dlaždic, ale to nejdůležitější jsou následující 3 generovací metody, které uchovávají veškerou logiku generování.

První metodou je `GenerateTiles()`, kde se v rámci 2 for cyklů, které projdou přes každou pozici na mapě se vybere náhodné číslo, které určí typ dlaždice, výchozí stav je prázdná a pokud číslo odpovídá jinému typu, tak se přepíše typ dlaždice. Poté se se dlaždice na dané pozici přidá do kolekce tiles.

```
private void GenerateTiles()
{
    for (int y = 0; y < sizeY; y++)
    {
        for (int x = 0; x < sizeX; x++)
        {
              int number = Random.Range(0, 101);
              Tile tile = new Tile(new Vector2(x, y), eTileType.Texture.Empty);

              if (number > 90)
                  tile.Type = eTileType.Texture.Tree;
              if (number > 95)
                  tile.Type = eTileType.Texture.Obstacle;
```

```
                if (number > 97)
                    tile.Type = eTileType.Texture.Ruins;
                if (number > 99)
                    tile.Type = eTileType.Texture.GoldVein;

                tiles.Add(tile);
            }
        }
    }
```

Kód 12: Metoda pro generování dlaždic (zdrojem autor)

Další metodou je `GeneratePath()`. Toto je nejobsáhlejší metoda, řídí samotné generovaní cesty. Je navržena tak, aby cesta vedla vždy zleva doprava a náhodně se roztahuje nahoru a dolů jako had, ale vždy, pokud se cesta stočí shora dolů, tak se zároveň posune doprava, aby vždy bylo mezi cestou 1 pole volné.

```
private void GeneratePath()
{
    int startingPoint = Random.Range(0, sizeY);
    int currentX = 0;
    int prevVertDirection = 0;
    int currentY = startingPoint;

    tiles.Find(x => x.Position == new Vector2(0, startingPoint)).Type =
       eTileType.Texture.Path;

    WaypointPositions.Add(new Vector2(0, -startingPoint * TileSize));
    spawner.transform.position = new Vector3(0 + (TileSize / 2),
       (-startingPoint       * TileSize) - (TileSize / 2), 0);

    while (currentX < sizeX - 1)
    {
       if (Random.Range(0, 4) == 1)
       {
              currentX++;

              if (prevVertDirection != 0)
              {
                     // ... logic for adding tiles and waypoints
              }

                prevVertDirection = 0;
       }
       else
       {
              switch (prevVertDirection)
              {
                     case 1:
                     if (currentY < sizeY - 1)
                     {
                            currentY++;
                            prevVertDirection = 1;
                     }
                     break;
                     case -1:
```

```
                    if (currentY > 0)
                    {
                        currentY--;
                        prevVertDirection = -1;
                    }
                    break;
                    case 0:
                    if (Random.Range(0, 2) == 1)
                    {
                        if (currentY < sizeY - 1)
                        {
                            // ... incrementation and adding waypoint
                        }
                        else
                        {
                            // ... decrementation and adding waypoint
                        }
                    }
                    else
                    {
                        if (currentY > 0)
                        {
                            // ... decrementation and adding waypoint
                        }
                        else
                        {
                            // ... incrementation and adding waypoint
                        }
                    }
                    break;
                }
            }

            tiles.Find(x => x.Position == new Vector2(currentX, currentY)).Type =
                eTileType.Texture.Path;
        }

        Vector2 finishTilePos =
            tiles.Find(x => x.Position.x == sizeX - 1
            && x.Type ==  eTileType.Texture.Path).Position;

        WaypointPositions.Add(new Vector2(finishTilePos.x * TileSize,
             -finishTilePos.y * TileSize));

        MakeWaypoints();
    }
```

Kód 13: Metoda pro generování cesty na mapě (zdrojem autor)

Poslední metodou je `MakeWaypoints()`, která už vytváří jednotlivé waypointy jako instance objektů do scény. Tyto waypointy pak slouží k navigování nepřátel po hrací ploše až k jádru.

```
private void MakeWaypoints()
{
    foreach (GameObject obj in GameObject.FindGameObjectsWithTag("Waypoint"))
        GameObject.DestroyImmediate(obj);

    GameObject waypointHolder = GameObject.Find("Waypoints");
    foreach (Vector2 waypoint in WaypointPositions)
    {
        GameObject newWaypoint = new GameObject("Waypoint");
        newWaypoint.tag = "Waypoint";
        newWaypoint.transform.parent = waypointHolder.transform;
        newWaypoint.transform.localPosition = new Vector3(
            waypoint.x + TileSize / 2, waypoint.y - TileSize / 2, 0);
    }
}
```

Kód 14: Metoda k vytvoření waypointů pro AI (zdrojem autor)

Tímto bychom měli připravenou datovou strukturu herní plochy a nyní se podíváme na logiku třídy `TileMapGraphics`, která nám vytvoří a vykreslí texturu.

Opět si nyní ukážeme základní tělo třídy a poté si popíšeme její zásadní metody, které ovládají logiku.

```
public int sizeX = 100;
public int sizeY = 50;
public float tileSize = 1.0f;

public Texture2D terrainTiles;
public int tileResolution = 50;
public TileMapData Map { get; private set; }

void Start ()
{
    BuildMesh();
    GameObject.Find("_GameMaster").GetComponent<GameMaster>().GenerateCore();
}
```

Kód 15: Základní vlastnosti a metody třídy TileMapGraphics (zdrojem autor)

První metodou je `BuildMesh()`, tato metoda se stará o samotné sestavení `Mesh`, což je objekt na který poté naneseme texturu. Nejprve si necháme vytvořit `TileMapData` a poté vygenerujeme mesh data, což je vlastně sada verticies, normals a uv vektorů.

```
public void BuildMesh()
{
    // creating map data
    Map = new TileMapData(sizeX, sizeY);

    // number of tiles
    int numTiles = sizeX * sizeY;
    // number of triangles
    int numTris = numTiles * 2;
```

```
    // verticies size, number
    int vSizeX = sizeX + 1;
    int vSizeY = sizeY + 1;
    int numVerts = vSizeX * vSizeY;

    // generate mesh data
    Vector3[] verticies = new Vector3[numVerts];
    Vector3[] normals = new Vector3[numVerts];
    Vector2[] uv = new Vector2[numVerts];
    int[] triangles = new int[numTris * 3];

    int x, y;
    for (y = 0; y < vSizeY; y++)
       for (x = 0; x < vSizeX; x++)
       {
           verticies[y * vSizeX + x] =
               new Vector3(x * tileSize, -y * tileSize, 0);
           normals[y * vSizeX + x] = Vector3.up;
           uv[y * vSizeX + x] = new Vector2((float)x / sizeX, (float)y / sizeY);
       }

    int squareIndex, triOffset;
    for (y = 0; y < sizeY; y++)
       for (x = 0; x < sizeX; x++)
       {
           squareIndex = y * sizeX + x;
           triOffset = squareIndex * 6;

           triangles[triOffset + 0] = y * vSizeX + x + 0;
           triangles[triOffset + 1] = y * vSizeX + x + vSizeX + 1;
           triangles[triOffset + 2] = y * vSizeX + x + vSizeX + 0;

           triangles[triOffset + 3] = y * vSizeX + x + 0;
           triangles[triOffset + 4] = y * vSizeX + x + 1;
           triangles[triOffset + 5] = y * vSizeX + x + vSizeX + 1;
       }

    //create new mesh and populate with data
    Mesh mesh = new Mesh();
    mesh.vertices = verticies;
    mesh.triangles = triangles;
    mesh.normals = normals;
    mesh.uv = uv;

    //assign our mesh to our filter,renderer,collider
    MeshFilter meshFilter = GetComponent<MeshFilter>();
    MeshCollider meshCollider = GetComponent<MeshCollider>();

    meshFilter.mesh = mesh;
    meshCollider.sharedMesh = mesh;

    BuildTexture();
}
```

Kód 16: Metoda k sestrojení Mesh objektu pro texturu (zdrojem autor)

Druhá metoda `BuildTexture()` vytvoří 2D texturu, která pokryje celou herní plochu, poté se textura naseká na kusy, které odpovídají jednotlivým dlaždicím a na základe `TileMapData` se nanesou na jednotlivé dlaždice textury, které odpovídají jejich typu. Na konci tuto texturu aplikujeme na `Mesh`, který jsme vytvořili v předchozí metodě.

```
void BuildTexture()
{
    // size of a texture
    int texWidth = sizeX * tileResolution;
    int texHeight = sizeY * tileResolution;
    Texture2D texture = new Texture2D(texWidth, texHeight);

    Color[][] tiles = ChopUpTiles();

    for (int y = 0; y < sizeY; y++)
    {
       for (int x = 0; x < sizeX; x++)
       {
           // needs the index (number) of a tile in sprite sheet
           Color[] p = tiles[(int)Map.GetTile(x, y).Type];

           texture.SetPixels(x * tileResolution, y * tileResolution,
              tileResolution, tileResolution, p);
       }
    }

    texture.filterMode = FilterMode.Point;
    texture.wrapMode = TextureWrapMode.Clamp;
    texture.Apply();

    MeshRenderer meshRenderer = GetComponent<MeshRenderer>();
    meshRenderer.sharedMaterials[0].mainTexture = texture;
}
```

Kód 17: Metoda sestavující samotnou texturu (zdrojem autor)

Poslední metoda je `ChopUpTiles()`, které udělá ono rozdělení textury na jednotlivé dlaždice. Její využití můžeme vidět v předchozí metodě. Návratový typ je 2 dimenzionální pole pixelů textury.

```
Color[][] ChopUpTiles()
{
    int numTilesPerRow = terrainTiles.width / tileResolution;
    int numRows = terrainTiles.height / tileResolution;

    // two dimensional field (in 1st dimension is each tile sprite (1, 2, 3))
    // (in 2nd dimension is pixel data of that tile)
    Color[][] tiles = new Color[numTilesPerRow * numRows][];

    for (int y = 0; y < numRows; y++)
    {
       for (int x = 0; x < numTilesPerRow; x++)
       {
```

```
            tiles[y * numTilesPerRow + x] = terrainTiles.GetPixels(
                x * tileResolution, y *    tileResolution, tileResolution,
                tileResolution);
        }
    }

    for (int i = 0; i < tiles.Length; i++)
        tiles[i] = tiles[i].Reverse().ToArray();

    return tiles;
}
```

Kód 18: Metoda, které vrátí jednotlivé dlaždice jako sady pixelů k obarvení (zdrojem autor)

Díky těmto třídám máme nyní k dispozici vygenerované libovolně velké hrací pole, kde nebylo třeba využít ani jedno volání metody `Instantiate()` k vytvoření samotné dlaždice.

Nyní se můžeme na jakoukoliv dlaždici dotázat pomocí get metod, které jsme si nadefinovali a například detekce myši na herní ploše je řešena pomocí Raycastingu, protože na této komponentě není žádný trigger box, který by mohl snímat jednotlivé dlaždice. Nicméně implementace Raycastingu pro detekci myši nad dlaždicemi je poměrně jednoduchá a efektivní. Názorný příklad si ukážeme v další kapitole.

3.3 Raycasting

„Raycasting je běžně používaný ve vývoji her pro věci jako určení line of sight u hráče či AI, kam poletí projektil, tvorbu laserů a další. Raycast je v podstatě paprsek, který je vyslán z určité pozice v 3D nebo 2D prostoru and pohybuje se ve specifickém směru. Unity3D má vestavěné funkce, které mohou být použity k implementaci Raycastu v naší hře.“ (převzato z [14], volně přeloženo)

Když chceme využít Raycasting, tak objekty musím mít `collider` nebo `rigidbody`, aby se dali zaznamenat při protínání paprsku.

Máme zde 2 rozdílné přístupy:

Physics.Raycast – Zde je vržen paprsek z výchozí pozice v jistém směru o určité délce oproti všem colliderům v současné scéně. Tento přístup je vhodný, pokud neděláme detekci v rámci statického prostředí, kde by byl střet pouze s jedním colliderem, ale máme dynamické prostředí, kde například detekujeme stěny, hráče, atd… a to vždy s jiným výsledkem. [15]

Collider.Raycast – Zde je vržen paprsek s určenou výchozí pozicí a směrem, také lze určit délku, ale ignoruje všechny collidery, kromě tohoto. Tento přístup jsme využili při tvorbě hry, při zjišťování, na kterou dlaždici herní plochy právě hráč ukazuje myší. Jak to implementujeme si popíšeme níže. [16]

Poslední věc, co stojí za zmínku jsou 2 typy (třídy), které v rámci Raycastingu využijeme a jsou to `Ray` a `RaycastHit`.

`Ray` je reprezentací paprsků samotných, určují se u něj počátek a směr.

`RaycastHit` je struktura využívaná pro získání informací zpět od raycast. Vlastně tím získáme výsledek raycastu. Hlavní proměnné, co zde využijeme jsou collider, rigidbody a point, v proměnné collider je uložený samotný `Collider` co byl zasažen a nebo naopak v rigidbody je uložený `Rigidbody` objektu co byl zasažen, nakonec v proměnné point je bod v rámci world space, kde ray zasáhl onen collider.

3.3.1 Vlastní implementace

Zde si popíšeme, jak jsme ve hře implementovali zjišťování vybrané dlaždice v rámci herní plochy díky raycastu od kamery z aktuální pozice myši směrem na herní plochu.

Skript jsme nazvali `TileMapMouse` a je třeba ho mít napojený přímo na herní ploše, abych se snadno dostal ke collideru herní plochy. Jádro logiky se bude odehrávat v metodě `Update()`, protože musíme myš sledovat neustále, a ne pouze na základě nějakého eventu. Nyní si ukážeme metodu update a popíšeme její funkce.

```
void Update ()
{
    Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
    RaycastHit hitInfo;

    if (GetComponent<Collider>().Raycast(ray, out hitInfo, Mathf.Infinity))
    {
       int x = Mathf.FloorToInt(hitInfo.point.x / tgMap.tileSize);
       int y = Mathf.FloorToInt(hitInfo.point.y / tgMap.tileSize);

       highlightTile.SetActive(true);

       currentTileCoord.x = x;
       currentTileCoord.y = y + 1;
       highlightTile.transform.position = new Vector3(
           currentTileCoord.x * tgMap.tileSize,
           currentTileCoord.y * tgMap.tileSize, 0);
    }
```

```
    else
    {
       highlightTile.SetActive(false);
    }

    if (highlightTile.activeSelf)
    {
       // Currently hovered tile
       tile = tgMap.Map.GetTile((int)currentTileCoord.x,
           (int)-currentTileCoord.y);
       towerGUI.OnHover(tile);

       // right mouse click
       if (Input.GetMouseButtonDown(1))
       {
           buildGUI.Show(tile);
       }

       // left mouse click
       if (Input.GetMouseButtonDown(0))
       {
           towerGUI.OnLeftClick(tile);
       }
    }
}
```

Kód 19: Detekce Raycastingu vůči herní ploše (zdrojem autor)

Nejprve si určíme `Ray`, který budeme vysílat vůči herní ploše. Využili jsme k tomu funkci kamery `ScreenPointToRay`, která vrátí `ray` procházející od kamery skrze daný point. `SceenPoint` je v pixelech pozice na obrazovce, kterou získáme pomocí `Input.mousePosition`. Tím máme vytvořený `Ray`, který vychází z pozice naší myši směrem do scény.

Poté si vytvoříme `RaycastHit`, kam si uložíme výsledek a v rámci následující podmínky „vystřelíme“ paprsek vůči collideru naší herní plochy.

Pokud jsme zasáhli, tak je třeba přepočítat získané souřadnice na souřadnice x,y, které využíváme v rámci mapy dlaždic. Jak můžeme vidět využívá se proměnná point z `RaycastHit`, která obsahuje pozici protnutí paprsku s colliderem.

Zaktivujeme tzv. highlight tile, která slouží ke grafickému znázornění, aby hráč viděl přesně na které dlaždici je „najetý“. Nastavíme souřadnice stávající dlaždici a přesuneme tam highlight tile. Pokud jsme ale nic nezasáhli, tak ji deaktivujeme.

Nakonec, pokud je higlight tile aktivní, což znamená, že máme označenou nějakou herní dlaždici, tak odchytáváme, zda uživatel nezmáčkl levé, či pravé tlačítko myši. Pokud ano, tak mu buď ukážeme měnu pro stavění věže anebo informace, o již postavené věži, na kterou se kliklo.

4 Fyzika, hudba a animace

Tyto 3 sekce jsou nedílnou součástí vývoje každé hry, jinak bychom dostali pouze nedodělaný polotovar. V každé sekci se budeme soustředit na pohled z 2D, který uplatníme v naší hře.

Je nutno podotknout, že je určitý rozdíl mezi každou sekcí a její workflow v 2D a 3D. Podle toho, v jaké světě hru vytváříme, tak budeme muset postupy přizpůsobit dané problematice.

4.1 Fyzika

Unity obsahuje 2 separátní physics enginy, jeden pro 2D a druhý pro 3D. Aby hra vypadala realisticky a dodala nám správný pocit ponoření, tak je potřeba zpracovat fyzikální interakce. Například gravitace, masa objektu, zrychlování a různé další chování.

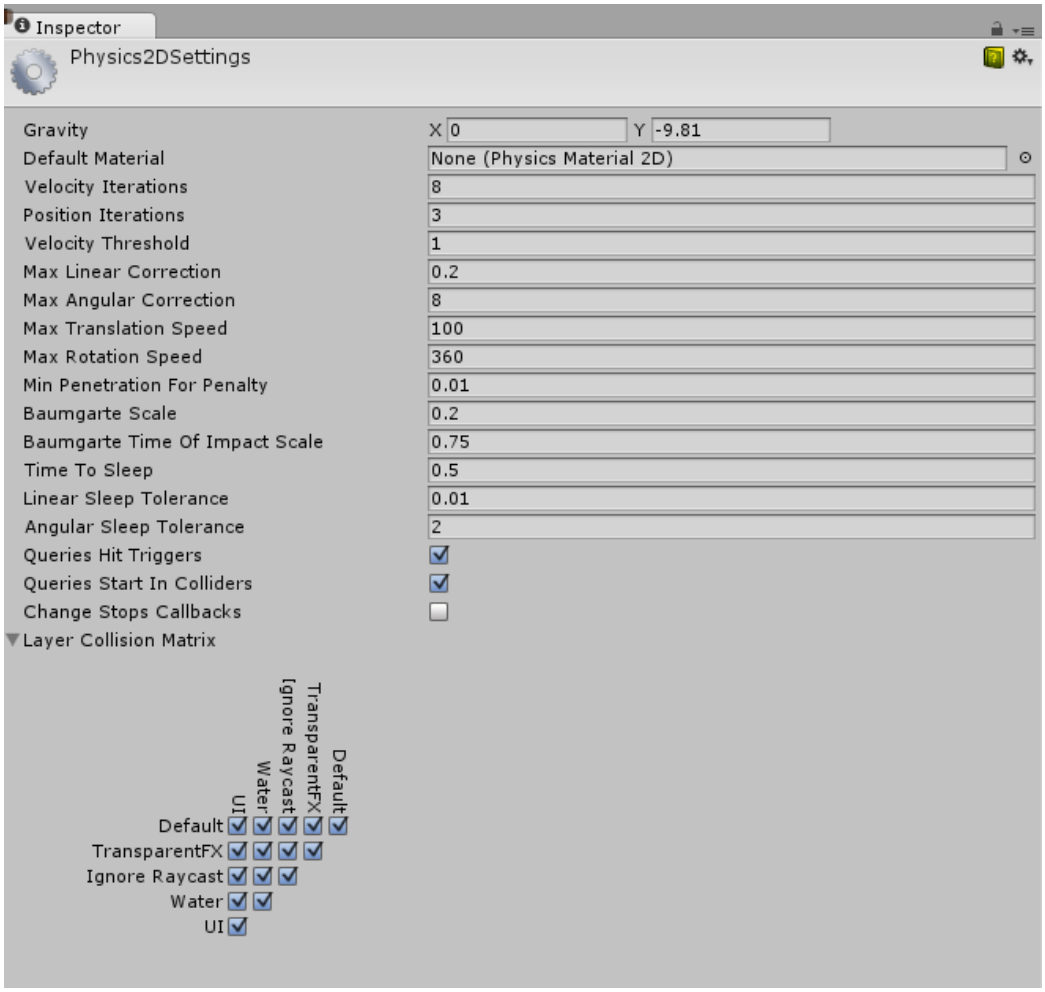
V rámci 2D budeme využívat `Physics2D`, který je určen pro 2D scény, jak už lze z názvu říci. Většina komponent sdílí podobné názvy jako jejich protějšky z 3D, i funkcionalita je dosti podobná, například v 3D máme `Rigidbody` a v 2D je `Rigidbody2D`.

Pokud chceme v Unity implementovat nějakou fyziku, tak rozdělíme úlohy na 2 kroky. Zaprvé budeme muset ve scéně vybrat požadované komponenty, které budou sloužit jako vstupní/výstupní efektory pro interakci a nastavit jim požadované vlastnosti. Výčet jednotlivých dostupných komponent a jejich detailnější popis si uvedeme níže. Zadruhé budeme třeba vytvořit či modifikovat skripty, které budou ovládat danou fyzikální interakci. K těmto účelům nám Unity nabízí řadu metod, které můžeme ve skriptu zavolat a zpracovat tak různé události, jako například událost, že objekt vstoupil do vymezené oblasti jiného objektu, a tak dále. Opět si výčet oněch metod a základních konstrukcí uvedeme níže i s příklady. [17]

4.1.1 2D Komponenty

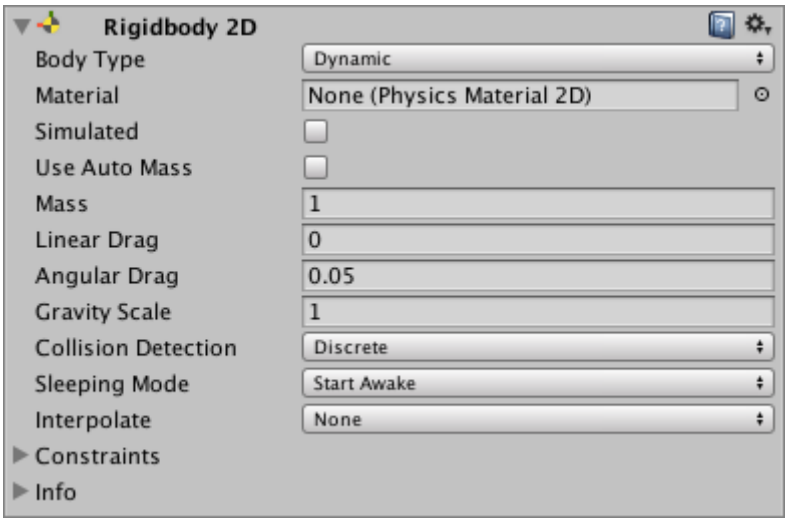
V rámci fyziky v 2D se zároveň řeší obecné nastavení editoru.

„Physics 2D nastavení poskytuje globální nastavení pro 2D fyziku (menu: Edit > Project Settings > Physics 2D). Je zde zároveň obdobné nastavení Physics Manageru pro 3D projekty.“ (převzato z [17], volně přeloženo)



Obrázek 14: Nastavení fyziky (převzato z https://docs.unity3d.com/Manual/class-Physics2DManager.html)

Rigidbody 2D – *„Tato komponenta umisťuje objekt pod kontrolu psyhics enginu. Mnoho konceptů známých ze standardního Rigidbody je přeneseno do Rigidbody 2D,*



rozdíl je, že v 2D objekty se mohou pohybovat pouze po rovině XY a mohou rotovat pouze vůči daným rovinám.“ (převzato z [18], volně přeloženo)

Obrázek 15: Rigidbody 2D (převzato z https://docs.unity3d.com/Manual/class-Rigidbody2D.html)

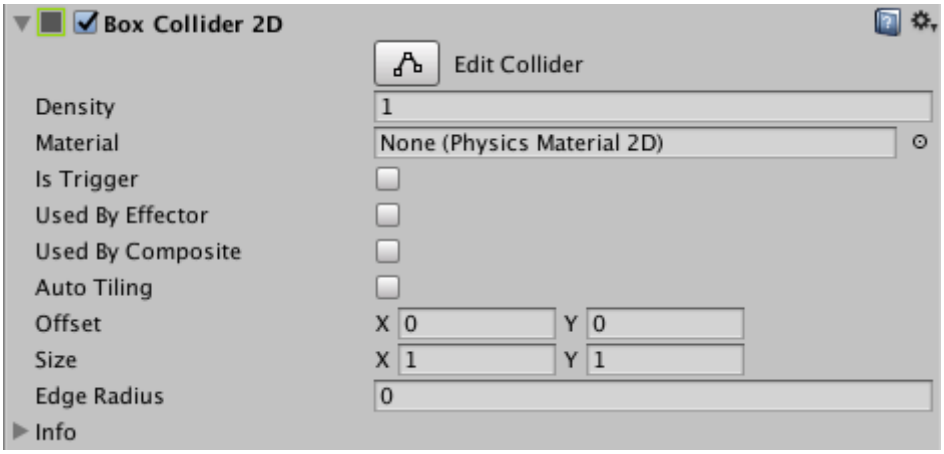
Hlavní užití této komponenty je simulace gravitace. Jak můžeme vidět na obrázku, tak se objektu nastaví hmotnost a škála gravitace, kterou bude ovlivňován a další dodatečná nastavení co doplní chování.

Collider 2D – *„Je komponenta, která definuje tvar 2D GameObjectu pro účely kolizí. Collider, který je neviditelný nemusí být přesně stejného tvaru jako Mesh onoho GameObjectu. Vlastně hrubá aproximace je často více efektivní a rozlišitelná ve hře.*“ (převzato z [19], volně přeloženo)

Máme k dispozici více typů colliderů, které se liší dle tvaru a následně jejich mapování na daný herní objekt.

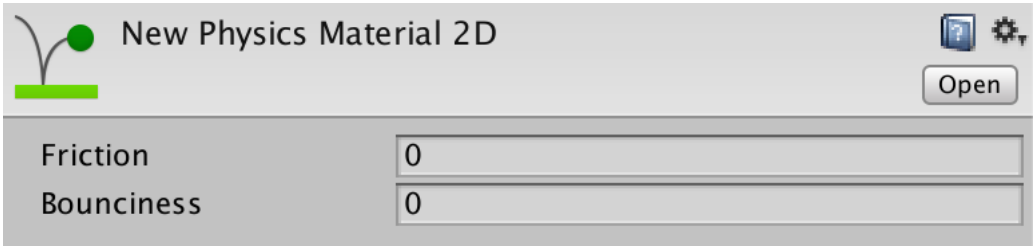
- **Circle Collider 2D** – pro kruhové kolizní zóny.
- **Box Collider 2D** – pro obdélníkové a čtvercové kolizní zóny.
- **Plygon Collider 2D** – pro volné kolizní zóny. Definují se hrany, které jsou spojeny úsečkami a tvoří kolizní polygon.
- **Edge Collider 2D** – pro volné kolizní zóny a zóny, které nejsou zcela uzavřeny.
- **Capsule Collider 2D** – pro kruhové či kulové kolizní zóny.
- **Composite Collider 2D** – pro seskupení Box a Plygon collideru.

Na obrázku níže lze vidět náhled na `BoxCollider2D`, což je jeden z nejpoužívanějších. Jeho nastavení je velice snadné. Za zmínku stojí vlastnost `Is Trigger`, které nám umožní, aby se collider choval jako trigger. To lze uplatnit dále ve skriptech, kde se dají využívat trigger metody. [19]



Obrázek 16: Box Collider 2D (převzato z https://docs.unity3d.com/Manual/class-BoxCollider2D.html)

Physics Material 2D – *„Využívá se k upravení tření a odrazivosti, která se vyskytuje mezi 2D objekty, když spolu kolidují. Lze vytvořit nový Physics Material 2D skrze Asset menu (Assets > Create > Physics Material 2D).“* (převzato z [20], volně přeloženo)



Obrázek 17: Physics Material 2D (převzato z https://docs.unity3d.com/Manual/class-PhysicsMaterial2D.html)

2D Joints – *„Jak naznačuje jméno, klouby pojí GameObjecty dohromady. Můžeme spojit pouze 2D klouby ke GameObjectu, který má Rigidbody 2D komponentu připojenou, nebo k fixní pozici ve světě.“* (převzato z [21], volně přeloženo)

Opět zde máme více typů kloubů. Liší se hlavně v možném užití, ale zde pouze zmíníme možné volby, samotnému popisu se vyhneme. Máme k dispozici Distance, Fixed, Friction, Hinge, Relative, Slider, Spring, Target a Wheel klouby.

4.1.2 Metody využitelné v rámci fyziky

Metody, které se dají v rámci skriptu v Unity volat se nazývají Messages, tzv. zprávy. My pouze danou metodu ve skriptu deklarujeme (vždy jsou návratového typu void, ale někdy mohou obsahovat vstupní parametry) a Unity nám už samo zařídí spouštění oněch metod na základě zpráv.

Nyní si popíšeme jednotlivé messages, které jsou dostupné pod třídou `Collider2D`, která je součástí `UnityEngine`.

OnCollisionEnter2D – Tato metoda má jako vstupní parametr `Collision2D` instanci. Z toho parametru můžeme získat veškerá potřebná data o kolizi. Jedním z hlavních atributů, na který se budeme dotazovat bude odkaz na kolidující `gameObject` anebo jeho transform. Jak lze vyčíst z názvu metody, tak je tato message zaslána ve chvíli, kdyby nějaký jiný collider zasáhne do collideru našeho objektu. [22]

OnCollisionExit2D – Tato message je zaslána, jakmile se collider jiného objektu přestane dotýkat collideru našeho objektu. Vstupní parametry jsou stejné jako u metody `OnCollisionEnter2D`. [22]

OnCollisionStay2D – Tato message je zasílána každý frame po dobu, kdy se collider jiného objektu dotýká collideru našeho objektu. Vstupní parametr je opět jeden a typu `Collision2D`. [22]

OnTriggerEnter/Exit/Stay2D – Opět podobná skupina tří metod, ale mají základní rozdíl. `Collider` zde musí mít zapnutou vlastnost `IsTriggered`, to znamená, že objekt přestane zpracovávat fyzickou kolizi, tzn. šlo by skrze objekt projít, ale pouze registruje, zda collideru našeho objektu vstoupí jiný objekt a odešle zprávu. Jako vstupní parametr metody se využívá jeden parametr typu `Collider2D`.

Jako příklad si nyní ukážeme zpracování kolize jednotlivý projektilů, které jsme využili při tvorbě hry. [22]

```
void OnTriggerEnter2D(Collider2D coll)
{
    if (coll.gameObject.tag == "Enemy")
    {
       Instantiate(ExplosionAnimation, coll.gameObject.transform.position,
           Quaternion.identity);

       EnemyBehaviour enemyBehaviour =
           coll.gameObject.GetComponent<EnemyBehaviour>();
       enemyBehaviour.Enemy.Health -= damage;
       enemyBehaviour.OnEnemyHitted();

       Destroy(gameObject);
    }
}
```

Kód 20: Metoda OnTriggerEnter2D, která zachycuje kolize typu Enemy (zdrojem autor)

Využíváme zde `Trigger`, protože se tím zamezí případné nárazy projektilů do nepřátel a jejich možné posunutí. Jakmile tedy `Trigger` zaznamená vstup jiného collider, tak se odešle zpráva, a jednoduše se podíváme na tag objektu, který máme pro nepřátele nastavený na `Enemy` a pokud to souhlasí, tak se vytvoří instance animace exploze, upraví se životy nepřítele, vyvolá se na něm událost, že byl zasažen a zničíme tento projektil.

4.2 Hudba

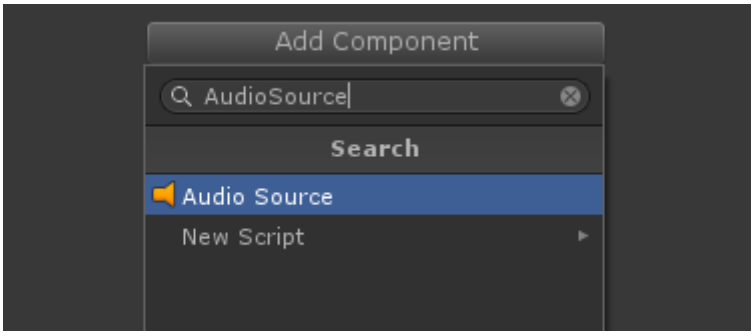
Unity nabízí mnoho funkcionalit a prostředků pro realizaci hudby a zvukových efektů ve hrách. V poslední době se ve hrách často dělají tzv. ambientní zvuky, které přidávají na úrovni ponořený do hry díky prostorové simulaci zvuku. K dispozici je více formátů zvuku, se kterými můžeme jako se zdroji pracovat, např. AIFF, WAV, MPC anebo Ogg. Je tu také podpora takzvaných tracker modules, což jsou krátké nahrávky například různých nástrojů, které se pak seskupují k vytvoření hudby. Podporované typy tracker modulů jsou .xm, .mod, .it a .s3m.

Existuje i přístup k mikrofonu počítače skrze skript a vytvoření zvukové nahrávky přímo pomocí mikrofonu. `Microphone` třída k tomuto účelu nabízí snadno použitelnou API, díky které můžeme ve hrách zpracovat různé zvukové vstupy od uživatele.

Nyní si ukážeme postup jako do hry přidat naše první zvukové efekty.

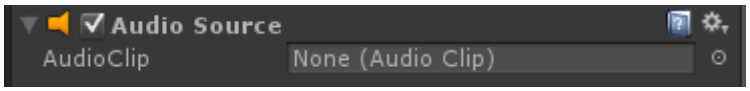
Prvním krokem bude vybrat nějaký prefab, na který chceme přidat nějaký zvukový zdroj. V našem projektu se jedná například o prefab projektilů, protože každý má

zvukový efekt při kolizi. Potom pouze přidáme novou komponentu `Audio Source` na náš prefab.



Obrázek 18: Audio Source, přidání komponenty (převzato z https://www.raywenderlich.com/132145/introduction-unity-sound)

Teď máme připravenou komponentu pro zdroj zvuku, ale ta potřebuje nějaký `AudioClip`, který poté bude přehrávat. `AudioClip` bude jeden ze zvukových souborů, které si přidáme do `Assets`.



Obrázek 19: Audio Source – Audio Clip (převzato z https://www.raywenderlich.com/132145/introduction-unity-sound)

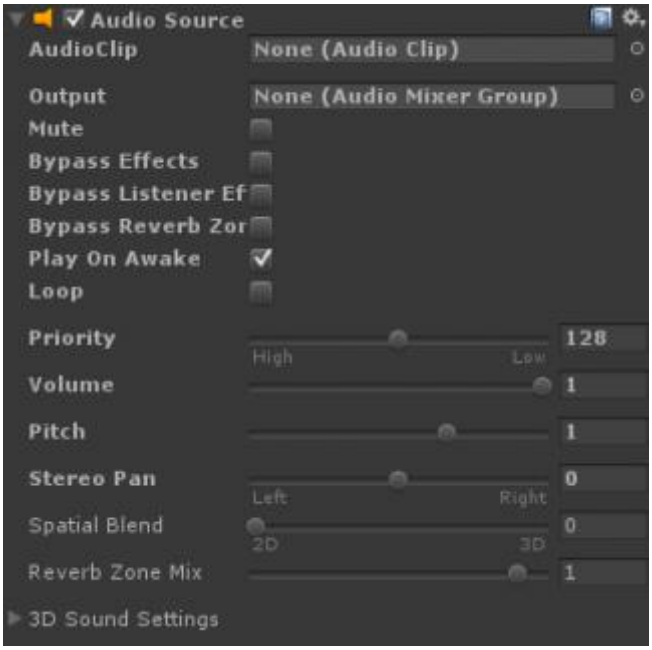
4.2.1 Jak fungují zvukové efekty v Unity

„K přehrání zvuku v Unity využíváme AudioSource a AudioClip.

AudioSource je to co vlastně bude přehrávat zvuk v 2D nebo 3D prostoru. V 3D prostoru se může úroveň hlasitosti zvuku měnit na základě vzdálenosti AudiouSource od objketu, který mu naslouchá (AudiouListener, který je ve hře na kameře).

Můžeme nastavit AudioSource, aby přehrával zvuk v 2D prostoru, což znamená, že bude přehrávat na konzistentní úrovni hlasitosti nehledě na vzdálenost od AudioListener.

AudioClip je samotný zvukový soubor, který AudioSource bude přehrávat.“ (převzato z [23], volně přeloženo)



Obrázek 20: Audio Source komponenta (převzato z https://www.raywenderlich.com/132145/introduction-unity-sound)

„Další důležitou věcí je, že AudioSource je komponenta. Což znamená, že je to objekt, který dědí z MonoBehaviour třídy a může být přiřazen přímo k jakémukoliv Unity GameObject.

AudioClip je proměnná, která se vyskytuje u AudioSource (ztn. Každá AudioSource bude mít jeden AudioClip). Můžeme přidat komponenty skrze editor nebo kód, ale zde se zabýváme pouze přidáním pomocí editoru.“ (převzato z [23], volně přeloženo)

Důležitou částí přehrávání zvuku je nastavení triggeru, který spustí přehrávání. `Trigger` je pouze ta podmínka, která řekne `AudioSource`, aby přehrálo zvuk.

Pokud chceme například vytvořit zvukovou smyčku, která by se spustila, pokud jsem v menu a bude se přehrávat dokola. Tak stačí pouze přidat `AudioSource` komponentu s požadovaným `AudioClip` zdrojem. Musíme však nastavit 2 proměnné, a to `Play On Awake`, což nám spustí hudbu hned po načtení komponenty a proměnnou `Loop`, aby se hudba opakovala. Obě proměnné můžeme vidět na obrázku výše. [23]

Nakonec ukázka krátkého skriptu, který můžeme použít jako trigger ke spuštění zvuku.

```
void Start()
{
    AudioSource audioSource = GetComponent<AudioSource>();
    audioSource.pitch = Random.Range(0.8f, 1.5f);
    audioSource.Play();
}
```

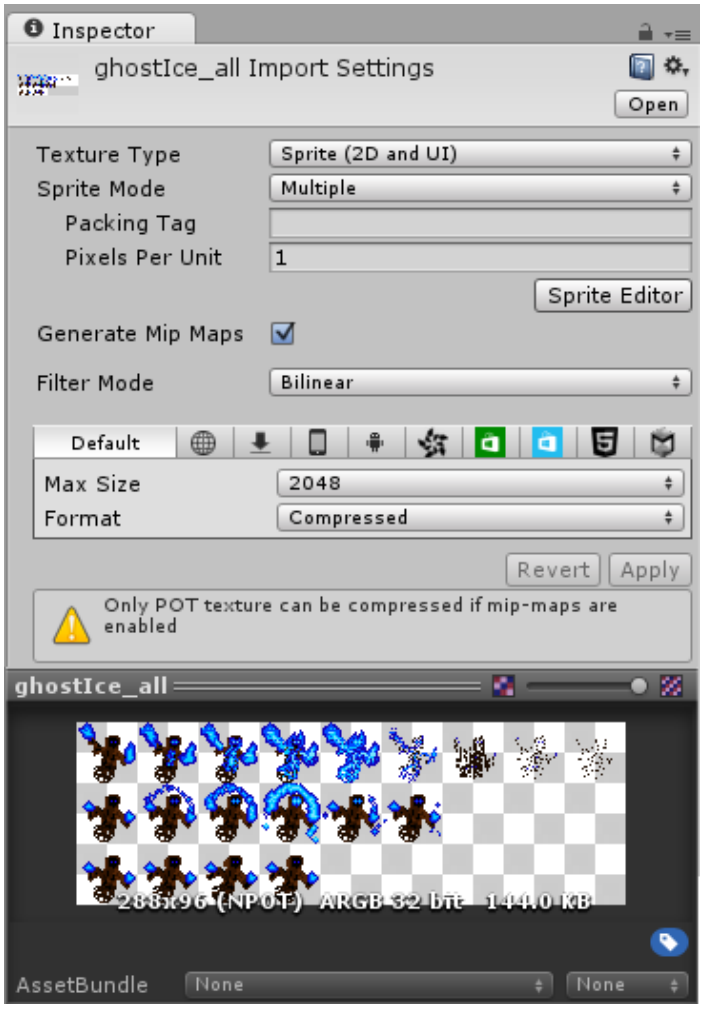
Kód 21: Přehrání audia (převzato z https://www.raywenderlich.com/132145/introduction-unity-sound)

4.3 Animace

Animace v Unity může být poměrně komplexní záležitost, ale zde se budeme věnovat pouze 2D animacím za pomoci kolekce spritů, nebo tzv. sprite atlasu.

„Unity podporuje přehrávání vícero animací ve stejný čas na vícero vrstvách, nebo jedné animace na základní vrstvě. Díky tomu je Unity extrémně flexibilní v přístupu, jak chceme vytvořit svou hru.“ (převzato z [24], volně přeloženo)

Začneme tedy importem spritů, které dále využijeme jako zdroj pro vytvoření animace. Důležité je, že musíme nastavit Texture Type na Sprite(2D and UI), SpriteMode na Multiple, protože budeme sprite muset „rozsekat“ na jednotlivé segmenty animace. Pixels Per Unit nastavíme na 1 místo 100, Generate Mip Maps na zaškrtnuto, Filter Mode na Bilinear, Max Size na 2048 a nakonec Format na compressed. Nesmíme zapomenout kliknout na tlačítko Apply, aby se změny projevily.



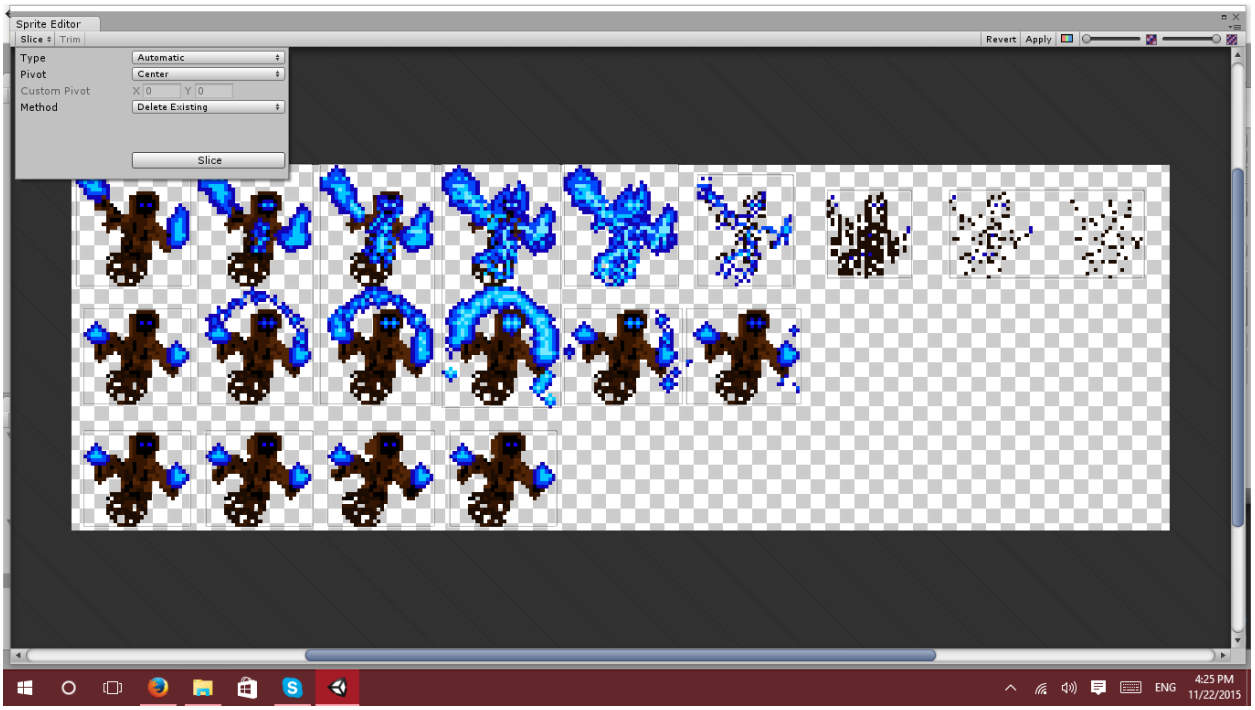
Obrázek 21: Sprite inspektor (převzato z https://gamedevacademy.org/unity-3d-animations-tutorial/)

4.3.1 Příprava segmentů animace

Teď klikneme na Sprite Editor pod nastavením Pixels per unit. Dostaneme se tím do editoru, kde budeme upravovat naše sprity. Lze zde například měnit pivot, a hlavně rozřezat sprite na pod obrázky, které budou tvořit segmenty animace.

„Navrchu ve Sprite Editoru vybereme Slice. Typ by měl být Automatic, Pivot center a metoda delete existing. Klikneme na tlačítko Slice.“ (převzato z [24], volně přeloženo)

Měli bychom vidět vybrání jednotlivých obrázků a pokud vypadá výběr v pořádku, tak potvrdíme tlačítkem Apply.



Obrázek 22: Sprite Edtior (převzato z https://gamedevacademy.org/unity-3d-animations-tutorial/)

Nyní v Assetech pod naším spritem, pokud ho rozklikneme bychom měli vidět jednotlivé segmenty obrázků. To znamená, že se spritem jsme hotovi a můžeme se přesunout k samotné animaci. [24]

4.3.2 Vytvoření animce

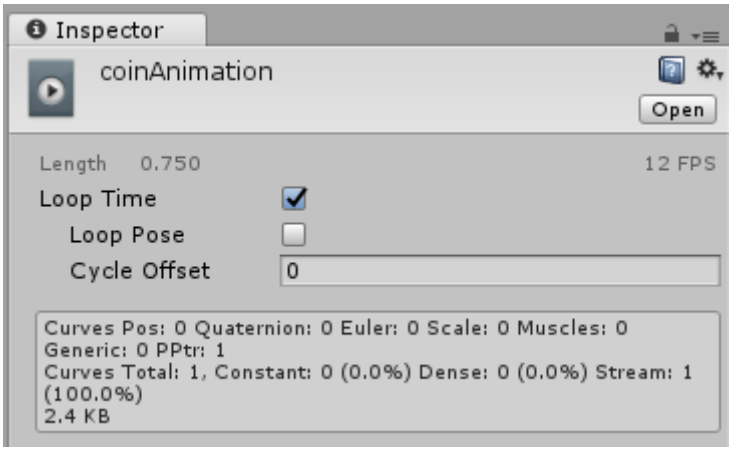
Tvorba samotné animace je poměrně přímočarý postup. Stačí vybrat všechny segmenty, které se nám vytvoří „rozsekáním“ spritu, po jejich vybrání je stačí přetáhnout do jakékoliv scény a editor z nich automaticky vytvoří animaci.

Jak jsi můžeme všimnout, tak se vytvoří 2 komponenty, jedna bude samotná animace označená .anim a druhou bude kontrolér s označením .controller, který bude řídit průběh a chování animace.

Teď bude žádoucí animaci nastavit, aby její animování probíhalo hladce a vyhovovalo našim potřebám, a nakonec přidáme k objektu, který se má animovat komponentu `Animator`, která zastřešuje celé chování animace a popíšeme si ji níže.

4.3.3 Animation

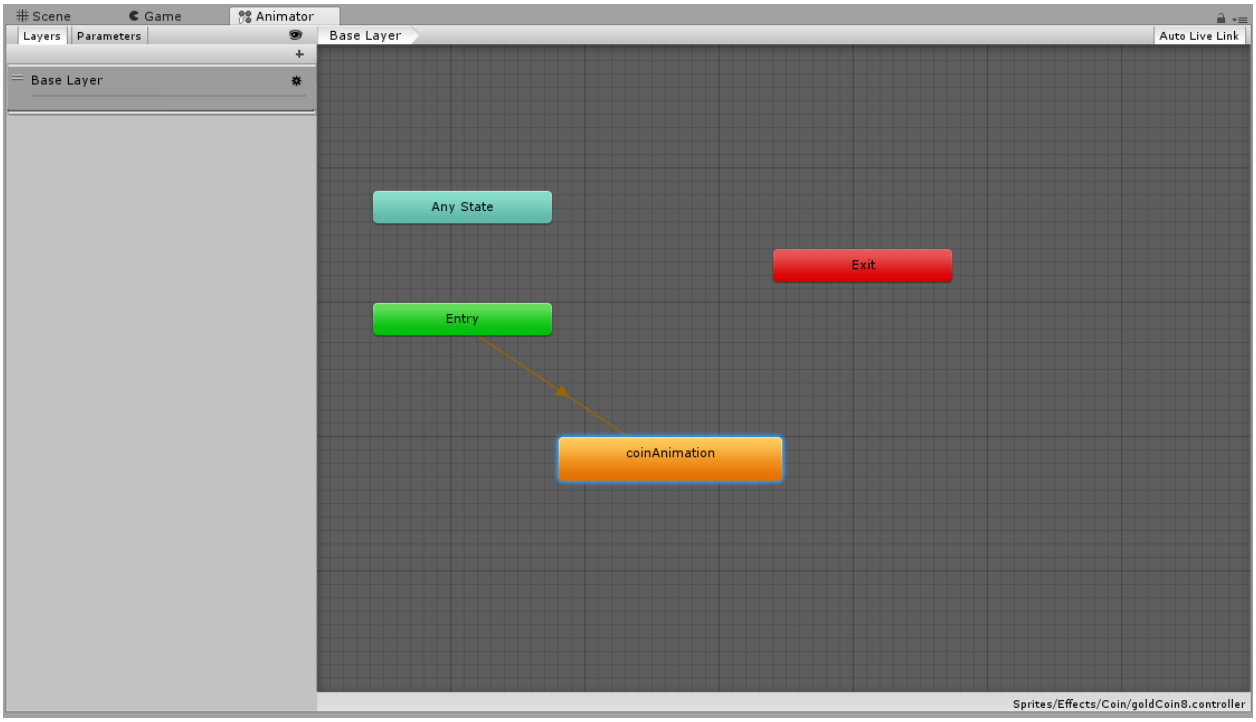
`Animation` je tedy soubor se samotnou animací, kterou budeme moct přehrávat, ale stále je zde pár proměnných, které lze nastavit a radikálně mění chování. Hlavní proměnnou, kterou budeme ve většině případů nastavovat bude `Loop Time`, která se zaškrtne, pokud chceme mít nekonečně se opakující animaci, což jsme využili v mém případě u otáčející se mince.



Obrázek 23: Animation soubor v inspektoru (zdrojem autor)

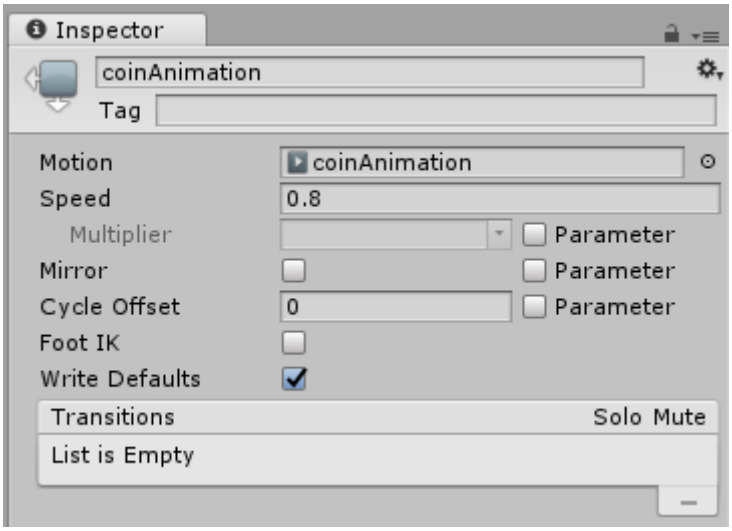
4.3.4 Animation Controller – Animator

„Je to zjednodušeně stavový stroj. Systém, který ukládá stavy a přechody mezi nimi, které jsou využity k řízení, kdy má být animace přehrána a jaké podmínky řídí přechod mezi stavy. Animation Crontrollers jsou uloženy s příponou .controller. Animace nemůže být přehrána bez Animation Controller.“ (převzato z [25], volně přeloženo)
Využít tento komplexní systém lze už například při tvorbě animace postavy, kde budeme mít animaci pro běžný stav kdy postava stojí, animaci chůze a animaci běhu. V animatoru lze pak nastavit jednotlivé přechody mezi animacemi se zvyšující se rychlostí postavy. Možností je zde velké množství a záleží pouze na konkrétní případu.



Obrázek 24: Animator, náhled průběhu animace (zdrojem autor)

My budeme mít v animátoru pouze jednu animaci, na kterou se přejde ihned ze stavu Entry. Je ale důležité poznamenat, že zde lze opět dospecifikovat chování jednotlivých animací. Pokud klikneme na danou animaci, tak uvidíme sadu proměnných a ta co nás zajímá je Speed, která určuje rychlost přehrávání a bude muset být vždy upravena, aby odpovídala našim představám.

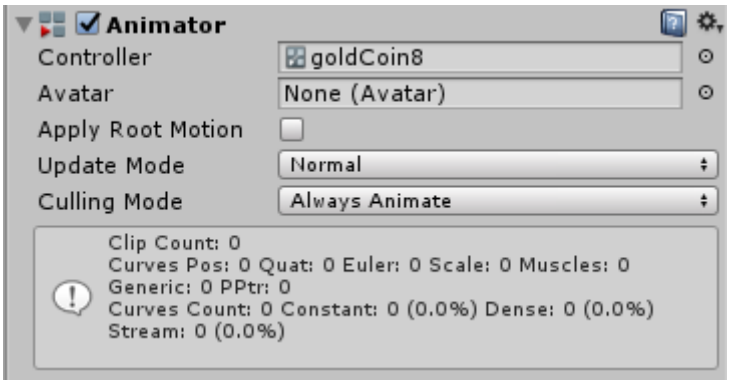


Obrázek 25: Animation State v Animatoru (zdrojem autor)

4.3.5 Animator komponenta

„K využití animace, co jsme si připravili v controlleru ji musíme aplikovat na herní objekt ve scéně. Toho docílíme skrze Animator komponentu, kterou lze vidět na obrázku níže.“ (převzato z [25], volně přeloženo)

Jak lze vidět, tak je nutné napojit referenci na požadovaný controller, který chceme využít.



Obrázek 26: Animator, zobrazení v inspektoru (zdrojem autor)

5 Shrnutí výsledků

V práci jsme pokryli základní kapitoly potřebné při každém vývoji hry. Popsané kapitoly byly striktně vztaženy na 2D prostor, není tedy možné veškeré poznatky aplikovat i pro 3D prostor, neboť se jedná o odlišné přístupy.

Začali jsme obecným popisem vývojového prostředí Unity a postupně se propracovali skrze manipulaci a assety až ke psaní skriptů, tvorbě animací, řešení fyziky a hudby.

Na daná témata lze na webu najít velké množství různých zdrojů a návodů, ať už je to oficiální dokumentace Unity či soukromé blogy vývojářů, tak lze téměř na každý problém najít ukázkové řešení.

Samotný engine Unity a jeho dokumentace je neustále vyvíjen a je třeba si držet přehled o nových nástrojích, či změnách těch starých.

Možné rozšíření problematiky týkající se této práce:

- Osvetlení a Lightmapper v Unity pro 2D
- Tilemap a tvorba levelů
- Scriptable Brushes pro dynamickou změnu prostředí
- Timeline výkonný nástroj pro tvorbu cutscenes a cinematics
- Multiplayer a Networking rozšíření o síťovou podporu

6 Závěry a doporučení

V práci nebyla z hlediska komplexnosti pokryta část databázová, která je zároveň částí většiny her. Unity podporuje napojení různých databázových systémů, ale nikdy ne přímo. Je to navržené tak, že se musí udělat mezivrstva, která zprostředkuje komunikaci s databází. Lze využít PHP server na kterém se budou zpracovávat SQL dotazy a na straně Unity se budou pouze generovat WWW requesty, které budou využívat PHP server pro získání dat z databáze.

Také jsme nemluvili o žádných prvcích 3D, ale to nebylo záměrem práce, jelikož je to moc obsáhlé. Ovšem Unity plně podporuje jak 2D, tak 3D a i různé platformy. Více se dá dohledat na webu Unity, nebo přímo v jejich dokumentaci.

Z pohledu 2D vývoje Unity zatím postrádá plnou podporu pro vývoj her založených na isometrickém pohledu. Nyní již s poslední verzí byla vydána podpora pro vývoj tzvn. Tile games, kde je herní plocha složena z 2D dlaždic.

Unity také trochu zaostává při řešení low-level optimizací, na které například narazí větší týmy, jako je custom occlusion culling či úprava rendering pipeline nebo práce se shadery. V těchto případech si často týmy vývojářů musí psát vlastní řešení, která obchází engine. Unity na tomto pracuje. V nynější beta verzi je již např. implementovaná programmable rendering pipeline, která umožňuje vývojářům zasáhnout do procesu renderování na nejnižší úrovni a upravit ji, jak je třeba.

7 Seznam zdrojů

[1] Unity Technologies. 2D and 3D Mode Settings [online]. Unity Technologies: © 2017 [cit. 20.11.2017]. Dostupné z: https://docs.unity3d.com/Manual/2DAnd3DModeSettings.html

[2] LAVIERI, Edward. Getting Started with Unity 2018. Packt, 3. vyd., 2018, s. 134-137 [cit. 14.4.2018]

[3] Unity Technologies. GameObject [online]. Unity Technologies: © 2017 [cit. 20.11.2017]. Dostupné z: https://docs.unity3d.com/ScriptReference/GameObject.html

[4] LINTRAMI, Tommaso. Unity 2017 Game Development Essentials. Packt, 3. vyd., 2018, s. 106 [cit. 14.4.2018]

[5] Unity Technologies. Sprites [online]. Unity Technologies: © 2017 [cit. 20.11.2017]. Dostupné z: https://docs.unity3d.com/Manual/Sprites.html

[6] Unity Technologies. Sprite Packer [online]. Unity Technologies: © 2017 [cit. 20.11.2017]. Dostupné z: https://docs.unity3d.com/Manual/SpritePacker.html

[7] Unity Technologies. Execution order [online]. Unity Technologies: © 2017 [cit. 7.1.2018]. Dostupné z: https://docs.unity3d.com/Manual/ExecutionOrder.html

[8] Unity Technologies. Canvas [online]. Unity Technologies: © 2017 [cit. 20.11.2017]. Dostupné z: https://docs.unity3d.com/Manual/UICanvas.html

[9] Unity Technologies. MonoBehaviour [online]. Unity Technologies: © 2017 [cit. 20.11.2017]. Dostupné z: https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

[10] Unity Technologies. MonoBehaviour.Start() [online]. Unity Technologies: © 2017 [cit. 20.11.2017]. Dostupné z: https://docs.unity3d.com/ScriptReference/MonoBehaviour.Start.html

[11] Unity Technologies. MonoBehaviour.Awake() [online]. Unity Technologies: © 2017 [cit. 20.11.2017]. Dostupné z: https://docs.unity3d.com/ScriptReference/MonoBehaviour.Awake.html

[12] Unity Technologies. MonoBehaviour.Update() [online]. Unity Technologies: © 2017 [cit. 20.11.2017]. Dostupné z: https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html

[13] Unity Technologies. Object.Instantiate() [online]. Unity Technologies: © 2017 [cit. 7.1.2018]. Dostupné z: https://docs.unity3d.com/ScriptReference/Object.Instantiate.html

[14] Mark Phillipp, Application Engineer at Studica. How to create a raycast in Unity3D [online]. Studica: © 2016 [cit. 7.1.2018] Dostupné z: https://www.studica.com/blog/how-to-create-a-raycast-in-unity-3d

[15] Unity Technologies. Physics.Raycast [online]. Unity Technologies: © 2017 [cit. 7.1.2018]. Dostupné z: https://docs.unity3d.com/ScriptReference/Physics.Raycast.html

[16] Unity Technologies. Collider.Raycast [online]. Unity Technologies: © 2017 [cit. 7.1.2018]. Dostupné z: https://docs.unity3d.com/ScriptReference/Collider.Raycast.html

[17] Unity Technologies. Physics 2D Settings [online]. Unity Technologies: © 2018 [cit. 17.3.2018]. Dostupné z: https://docs.unity3d.com/Manual/class-Physics2DManager.html

[18] Unity Technologies. Rigidbody 2D [online]. Unity Technologies: © 2018 [cit. 17.3.2018]. Dostupné z: https://docs.unity3d.com/Manual/class-Rigidbody2D.html

[19] Unity Technologies. Collider 2D [online]. Unity Technologies: © 2018 [cit. 17.3.2018]. Dostupné z: https://docs.unity3d.com/Manual/Collider2D.html

[20] Unity Technologies. Physics Material 2D [online]. Unity Technologies: © 2018 [cit. 17.3.2018]. Dostupné z: https://docs.unity3d.com/Manual/class-PhysicsMaterial2D.html

[21] Unity Technologies. 2D Joints [online]. Unity Technologies: © 2018 [cit. 17.3.2018]. Dostupné z: https://docs.unity3d.com/Manual/Joints2D.html

[22] Unity Technologies. Collider 2D - Script [online]. Unity Technologies: © 2018 [cit. 17.3.2018]. Dostupné z: https://docs.unity3d.com/ScriptReference/Collider2D.html

[23] Anthony Uccello. Introduction to Unity Sound [online]. Anthony Uccello: © 2016 [cit. 17.3.2018]. Dostupné z: https://www.raywenderlich.com/132145/introduction-unity-sound

[24] Jesse Glover. Understanding 2D Animations in Unity3D [online]. Zenva PTY LTD: © 2018 [cit. 17.3.2018]. Dostupné z: https://gamedevacademy.org/unity-3d-animations-tutorial/

[25] GODBOLD, Ashley. JACKSON, Simon. Mastering Unity 2D Game Development. Packt, 2. vyd., 2016, s. 71-72 [cit. 14.4.2018]

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2017/2018

Studijní program: Aplikovaná informatika
Forma: Prezenční
Obor/komb.: Aplikovaná informatika (ai3-p)

Podklad pro zadání BAKALÁŘSKÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Lásko Jakub	Smetanova 1245, Choceň	I14098

TÉMA ČESKY:

Vývoj 2D hry v Unity

TÉMA ANGLICKY:

2D game development in Unity

VEDOUCÍ PRÁCE:

doc. Ing. Filip Malý, Ph.D. - KIKM

ZÁSADY PRO VYPRACOVÁNÍ:

Vypracovat postup tvorby 2D hry v Unity. Zahrnout běžně využívané postupy, které budou rozšířeny o konkrétní aplikace a případná vylepšení např. z hlediska výkonu. Dále vytvořit praktickou ukázku 2D hry v Unity, na které budou demonstrovány popsané postupy v praxi.

1. Úvod
2. Základy Unity z pohledu 2D
3. Skriptování, tvorba UI
4. Fyzika, hudba a animace
5. Závěr
6. Literatura

SEZNAM DOPORUČENÉ LITERATURY:

Oficiální dokumentace Unity
Oficiální dokumentace C#

Podpis studenta: Lásko ..... Datum: 11.10.2017

Podpis vedoucího práce: mg ..... Datum: 11.10.2017