UNIVERZITA SV. CYRILA A METODA V TRNAVE
FAKULTA PRÍRODNÝCH VIED
Katedra aplikovanej informatiky

VÝVOJ 2D PC HRY – FANTASY CAR RACE V UNITY

Návody na cvičenia

Jana Jurinová

UNIVERZITA SV. CYRILA A METODA V TRNAVE

FAKULTA PRÍRODNÝCH VIED

Katedra aplikovanej informatiky

VÝVOJ 2D PC HRY – FANTASY CAR RACE V UNITY

Návody na cvičenia

Jana Jurinová

Trnava, 2022

Autor:

Ing. Jana Jurinová, PhD.

Recenzenti:

Ing. Miroslav Beňo, PhD.

PaedDr. Patrik Voštinár, PhD.

Skriptá boli schválené Edičnou radou Univerzity sv. Cyrila a Metoda v Trnave a vedením Fakulty prírodných vied UCM.

Vydané s podporou grantu KEGA 017UCM-4/2022 „Vývoj interaktívneho e-kurzu s využitím „SMART“ technológií na rozvoj algoritmického myslenia a programátorských zručností“.

Za odbornú a jazykovú stránku týchto skrípt zodpovedá autorka.

Rukopis neprešiel redakčnou ani jazykovou úpravou.

Vydavateľ: Univerzita sv. Cyrila a Metoda v Trnave, 2022

1. vydanie

ISBN 978-80-572-0237-0

Predhovor

Tieto skriptá sú určené pre študentov študijného odboru Aplikovaná informatika a pre všetkých, ktorí majú záujem rozvíjať svoje vedomosti v oblasti vývoja a programovania 2D hier pomocou herného enginu Unity.

Problematika spracovaná v tejto publikácii predpokladá vedomosti čitateľa z oblasti objektovo orientovaného programovania, algoritmického riešenia problémov, ako aj grafického spracovania. Snahou je predstaviť problematiku vývoja 2D hry v Unity na konkrétnom type hry, pričom skriptá povedú čitateľa krok za krokom celým vývojom tejto hry – od návrhu až po export. Je teda zrejmé, že tieto skriptá nepokrývajú celú komplexnú problematiku vývoja hier a ani všetky možností herného enginu Unity. Obsah nadväzuje na autorkine skriptá "Vývoj 2D PC hry Arkanoid v Unity". Preto sa v časti týchto skrípt odvolávame na už publikované informácie formou odkazov, aby sme zabránili duplicite.

Štruktúra a radenie jednotlivých kapitol vyplýva z logiky budovania herného mechanizmu hry. Tieto skriptá vychádzajú z oficiálnej dokumentácie k Unity. Obsah je doplnený o ilustrácie a obrázky zachytávajúce nastavenia priamo v editore Unity, alebo vo Visual Studio s cieľom podporiť vnímanie a chápanie praktickej aplikácie. Hra bola vyvíjaná v Unity vo verzii 2020.1.1f1. Kapitoly sú doplnené o kontrolné otázky a úlohy na overenie vedomostí čitateľa a motiváciu na ďalšie rozširovanie hry. Na trhu existujú desiatky, ak nie stovky publikácií venujúcich sa vývoju hier a objasňovaniu konceptov v Unity. Zoznam použitej a odporúčanej literatúry slúži pre čitateľov ako potenciálny zdroj na rozšírenie poznatkov o skúmanej problematike.

Skriptá vychádzajú v elektronickej podobe na základe dvoch faktorov. Grafická reprezentácia niektorých ilustrácií ani pri použitom formáte A4 nie je dostatočne čitateľná a vyžaduje potrebu zväčšenia. Druhým dôvodom je snaha uľahčiť čitateľom prácu, preto súčasťou skrípt sú zdrojové kódy a súbory riešeného projektu, vďaka ktorým si môžu okamžite overiť ich funkčnosť a priamo s nimi

pracovať. V skriptách sa tiež nachádzajú hypertextové odkazy, ktoré odkazujú priamo na dokumentáciu Unity s cieľom poskytnúť komplexné informácie.

Terminológia v informačných technológiách a informatike z veľkej časti pochádza z angličtiny a používatelia využívajú originálne anglické výrazy. Angličtina sa stala odborným žargónom IT špecialistov. Pre vývojára akéhokoľvek softvéru je znalosť anglického jazyka prakticky nevyhnutná, aspoň na úrovni čítania a porozumenia dokumentácií. Preto väčšina slovenských kľúčových termínov použitých v skriptách je kvôli lepšiemu pochopeniu interpretovaná aj v anglickom jazyku. Pre niektoré výrazy, ktoré sú použité v publikácii neexistujú zodpovedajúce a zaužívané ekvivalenty v slovenskom jazyku, preto používame v tejto práci pôvodnú anglickú terminológiu. Každý anglický termín je však vysvetlený pri prvom výskyte v texte publikácie.

Ak by ste objavili nejaké chyby, alebo by vám nejaké informácie v týchto skriptách chýbali, neváhajte ma kontaktovať na jana.jurinova@ucm.sk. Privítam akékoľvek pripomienky a námety na vylepšenie obsahu.

Rada by som na tomto mieste poďakovala recenzentom Ing. Miroslavovi Beňovi, PhD. a PaedDr. Patrikovi Voštinárovi, PhD. za ich prínosné odborné komentáre, poznámky a námety, ktoré prispeli k skvalitneniu týchto skript.

Marec 2022 Ing. Jana Jurinová, PhD., autorka učebnice

Obsah

1 Fantasy Car Race = Top-down Shooter Game 7

1.1 Fantasy Car Race Game Design (GDD – Game Design Document) 8

1.2 Kontrolné otázky a úlohy 9

2 Vytvorenie projektu a základného herného mechanizmu 10

2.1 Pohyb hráča 15

2.1.1 Zabezpečenie *frame-rate independent* hry 18

2.2 Snímanie sveta s využitím metódy *ViewportToWorldPoint()* 21

2.3 Kontrolné otázky a úlohy 26

3 Ako postupovať pri vývoji hry 27

3.1 Zabezpečenie streľby hráčom 29

3.1.1 Použitie korutín (*Coroutines*) 36

3.1.2 *FireContinously() Coroutine* 38

3.2 Zničenie objektu mimo herného priestoru 41

3.3 Kontrolné otázky a úlohy 45

4 Vytvorenie zhlukov pohybujúcich sa nepriateľov 46

4.1 Riešenie pohybu nepriateľa po definovanej ceste 49

4.2 Použitie *ScriptableObject* 51

4.2.1 Použitie cyklu *foreach* pre získanie zoznamu bodov danej cesty 55

4.3 Vytvorenie viacerých nepriateľov na jednej ceste 59

4.4 Riešenie rôznorodosti zhlukov nepriateľov 63

4.5 Kontrolné otázky a úlohy 71

5 Vývoj obranného a útočiaceho herného mechanizmu 72

5.1 Riešenie streľby nepriateľov 79

5.2 Ovplyvňovanie života postavy hráča a jeho smrť 84

5.3 Vyladenie procesu útoku 87

5.4 Kontrolné otázky a úlohy 92

6 Dolaďovanie vizuálnej stránky hry 94

6.1 Skrolovateľné pozadie (*scrolling background*) 94

6.2 Pridanie efektov - *Particle systems* 100

6.3 Pridanie zvukových záznamov 106

6.4 Kontrolné otázky a úlohy 112

7 Finálne úpravy hry 114

7.1 Prechod medzi scénami 114

7.2 Vytvorenie úvodnej a záverečnej scény 116

7.3 Ukončenie hry 131

7.4 Využitie návrhového vzoru *Singleton Pattern* 135

7.5 Pridanie skóre do hry 139

7.6 Zobrazenie života postavy hráča 148

7.7 Vytvorenie systému bonusov 151

7.8 Doladenie posledných vecí a publikovanie hry 156

7.9 Kontrolné otázky a úlohy 158

8 Zoznam použitej literatúry a informačných zdrojov 160

1 Fantasy Car Race = Top-down Shooter Game

Hra typu *Top-down Shooter Game* (*TDS*) známa tiež ako *Overhead shooter* je podkategóriou digitálnych hier známych pod skratkou *STG* (*shooting games*). Vyznačuje sa množstvom nepriateľov, ktorí útočia na hráča streľbou. Pohyb hráča je väčšinou zľava doprava. Avatarom hráča je spravidla buď vesmírna loď alebo vozidlo. Typickým prvkom týchto hier je skrolovateľné pozadie (*scrolling background*), pričom perspektíva hry bez možnosti rotácie sa realizuje dvomi spôsobmi:

- pohľad hráča na herné prostredie smerom zhora (*top-down perspective*),
- pohľad hráča na herné prostredie zboku (*side-view perspective*).

Niektoré hry sú doplnené aj o elementy simulujúce náraz a efekty demolácie. Hlavným cieľom takýchto hier je rýchla reakcia hráča a zmysel pre zapamätanie si útočných stratégií nepriateľov. Herné mechanizmy útokov protihráčov sú zväčša veľmi jednoduché a predvídateľné. Len veľmi zriedka sa tu uplatňujú reálne fyzikálne zákonitosti. Pohyb hráča, ako aj nepriateľov je väčšinou realizovaný priamočiaro (t. j. pohyb v rovnej línii vo vymedzenom priestore), ako aj ich streľba. V tomto kontexte sa môžeme stretnúť s pojmom „*inertia*“, ktorý hovorí o tom, že objekt je rezistentný voči všetkým fyzikálnym zákonitostiam a nedochádza k zmene vlastnosti *Velocity*, ktorou ovplyvňujeme správanie sa pohybu objektu. Základný model pohybu je, že sa objekt pohybuje rovno, buď konštantnou alebo meniacou sa rýchlosťou pohybu. Objektu sa dá pridať možnosť pohybovať sa späť. Hráč môže počas hry zbierať rôzne bonusy, ktorými môže vylepšovať schopnosti hráča (zvýšenie počtu životov, rýchlejší pohyb, väčšia možnosť obrany, väčšia rozmanitosť zbraní, sila útoku a pod.). Prvé základy pre tento charakter hier položila hra – arkáda *Space Invaders* z roku 1978 (Wikipedia, 2021).

Aj naša verzia stavia na týchto prvkoch, pričom vizuálna reprezentácia hry je vsadená do tematiky pretekov automobilu voči útoku nepriateľov. Grafické prevedenie hry však nie je primárnym aspektom týchto skrípt.

V súvislosti s Unity budeme pri vývoji tejto hry pracovať s fyzikou, korutinami a *Scriptable Objectom*. Použijeme *Unity physics engine*, budeme riešiť kolízie pevných objektov. Tiež využijeme návrhový vzor *Singleton*, postaráme sa o efekt skrolovateľného pozadia a použijeme *Particle Effect* na pridanie jednoduchej animácie. V neposlednom rade pridáme aj nejaké zvukové efekty.

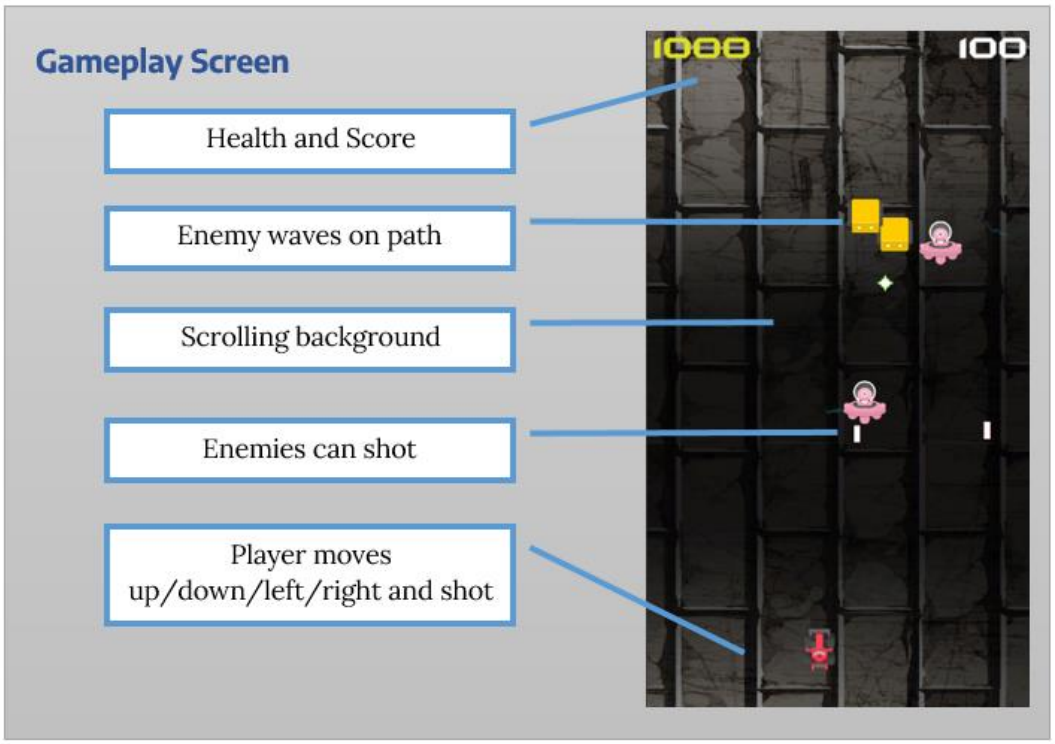
1.1 Fantasy Car Race Game Design (GDD – Game Design Document)

Základným stavebným prvkom pri každom vývoji hry je vytvorenie *Game Designu* pre vyvíjanú hru. Jeho podstatou je zachytiť návrh hry od prvotnej myšlienky cez vytvorenie témy, príbehu a pravidiel až po detailný princíp hry.

Ako prvé sa pokúsime zachytiť základný cieľ a princíp hry, ktorý je súčasťou *Game designu* (Obr. 1).

Obr. 1 *Game Design* hry.

Obrázok 2 predstavuje základné stavebné prvky hry:



Obr. 2 Základné stavebné prvky hry.

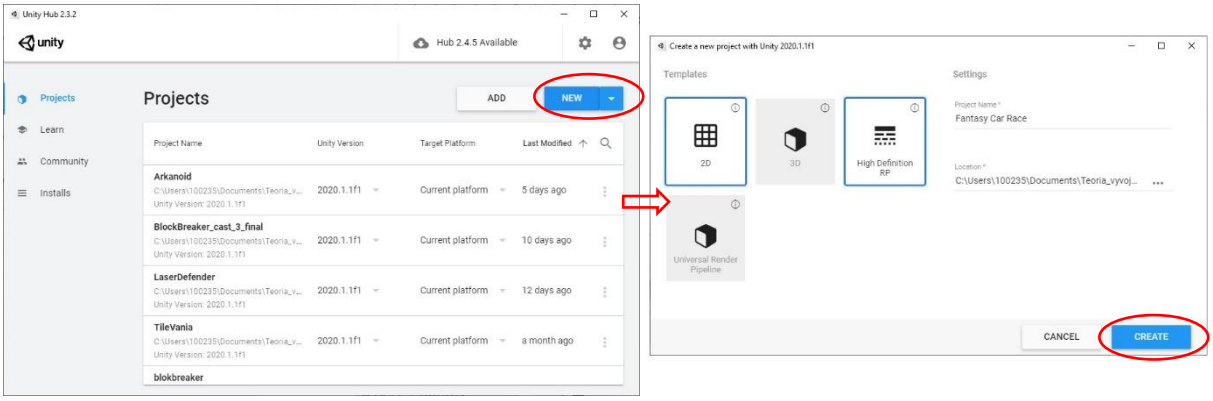
1.2 Kontrolné otázky a úlohy

1. Objasnite skratky *TDS* a *STG*.
2. Čo vyjadruje termín *Game Design*?
3. Premyslite si tému vašej hry, samozrejme môžete kopírovať moju.
4. Nájdite si vhodné obrázky pre základné stavebné prvky hry.
5. Premyslite si názov hry.

Tip autora: Bez ohľadu na použité grafické prvky, je vhodné dodržiavať pomenovania uvedené v skriptách z dôvodu, že tieto názvy budeme často používať v kódoch a pomocou nich sa na ne budeme odkazovať.

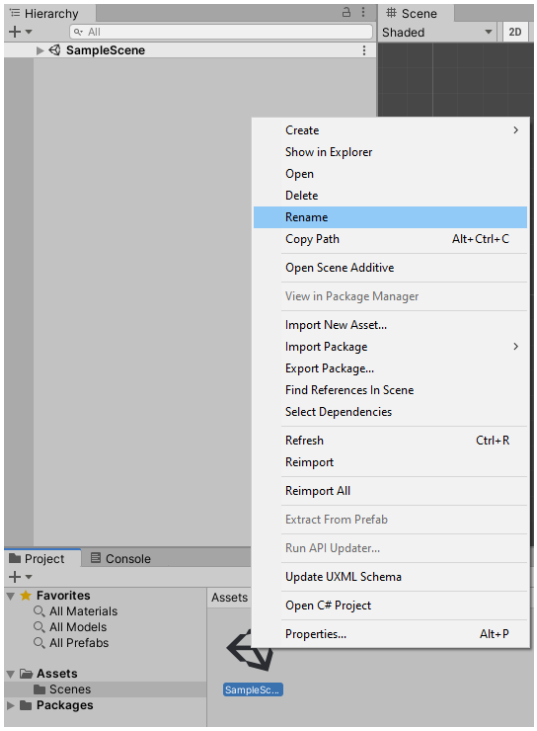
2 Vytvorenie projektu a základného herného mechanizmu

Nový projekt založíme pomocou Unity Hub kliknutím na tlačidlo NEW. Zvolíme 2D typ hry, určíme jej názov → *Fantasy Car Race* a zvolíme umiestnenie. Následne potvrdíme kliknutím na tlačidlo CREATE (Obr. 3).



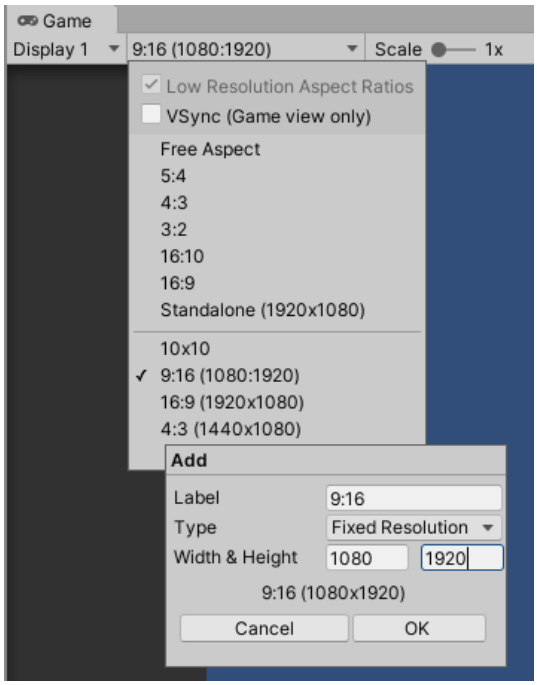
Obr. 3 Vytvorenie nového projektu.

Aktuálnu scénu, ktorej názov je *SampleScene* premenujeme na *Game*. Urobíme to tak, že v okne *Project*, v priečinku *Assets* a podpriečinku *Scenes* nájdeme túto scénu a pomocou kontextovej ponuky a možnosti *Rename* ju premenujeme (Obr. 4).



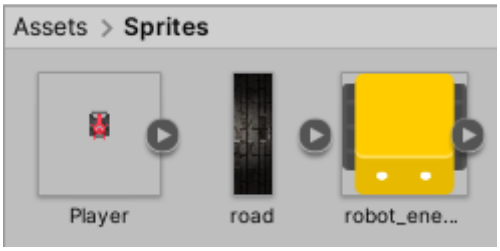
Obr. 4 Premenovanie existujúcej scény.

Pre túto hru odporúčame nastaviť *Aspect Ratio* na 9:16 a rozlíšenie (*resolution*) nastaviť na 1080 x 1920 (Obr. 5). To ovplyvní rozlíšenie hry, ktorá bude vizualizovaná vertikálne. Podľa Stewarta (2020) je nastavenie pre *Aspect Ratio* na 16:9 s rozlíšením 1920 x 1080 jedným z najčastejšie používaných.



Obr. 5 Nastavenie *Aspect Ratio* a rozlíšenia.

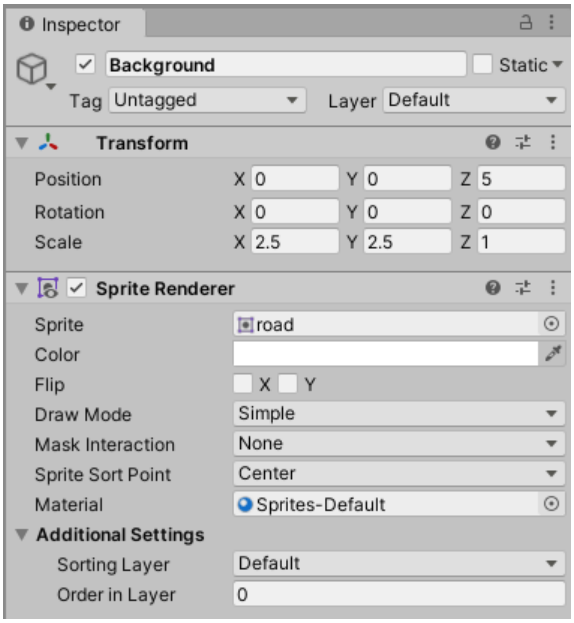
Vytvoríme priečinok *Sprites*, do ktorého budeme dávať používané aktíva (*assets*). *Assets*, ktorý používame v tejto hre, nájdete na https://kenney.nl. Linky na konkrétne použité zdroje nájde čitateľ v zozname použitej literatúry. Pre základné prvky – pozadie, hráč a nepriateľ, zvolíme vizuálne reprezentácie (Obr. 6).



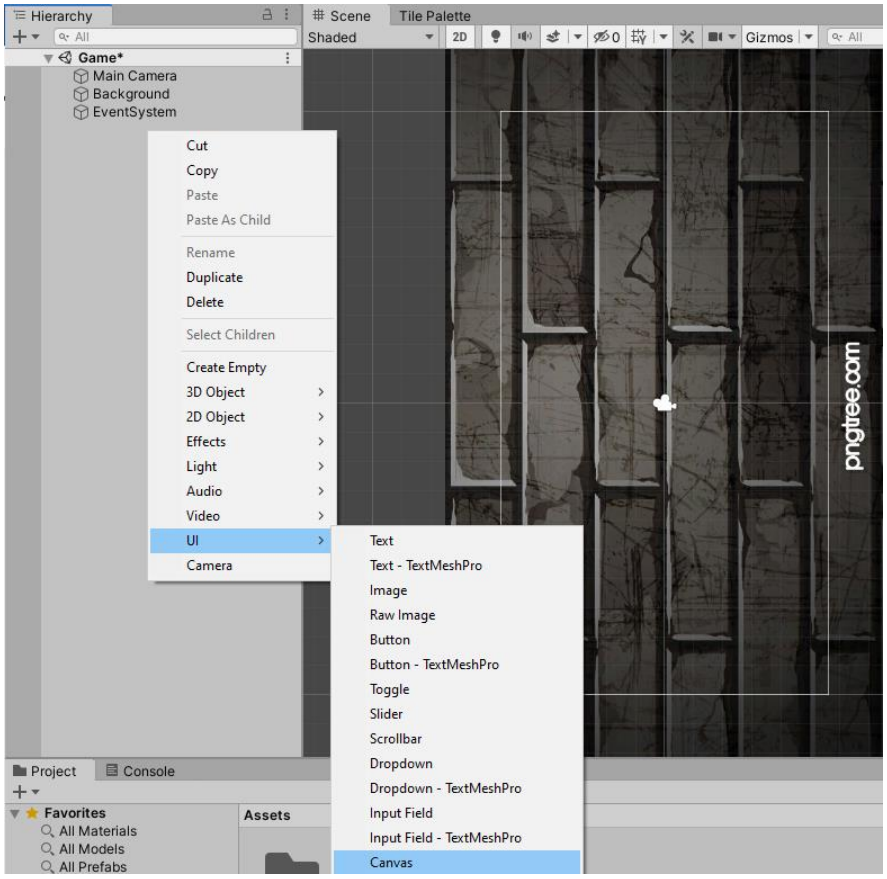
Obr. 6 Vizuálne reprezentácie pre základné prvky hry.

Do scény pridáme pozadie – *Background*, ktorému zmeníme veľkosť tak, aby vhodne ilustrovalo cestu (Obr. 7) a vytvoríme si *Canvas*, aby sme vedeli do neho

následne umiestňovať prvky nášho GUI (*Graphical User Interface*), vyvolaním kontextovej ponuky v okne *Hierarchy* UI → Canvas (Obr. 8).

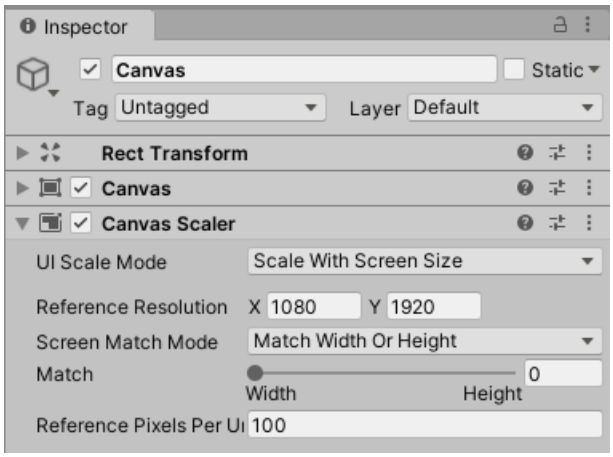


Obr. 7 Vlastnosti objektu *Background*.



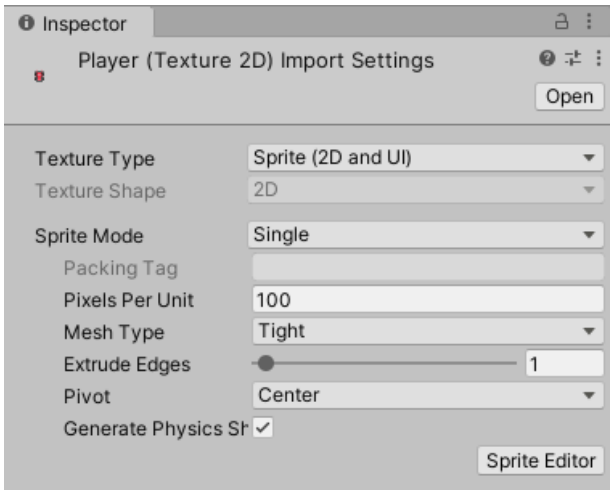
Obr. 8 Vytvorenie *Canvas* so zachytením vizuálu pozadia.

Aby sa jednotlivé objekty prispôsobovali podľa zmeny veľkosti obrazovky, nastavíme vlastnosť *UI Scale Mode* na *Scale With Screen Size* a *Reference Resolution* nastavíme na 1080 x 1920, ako sme uviedli vyššie (Obr. 9).



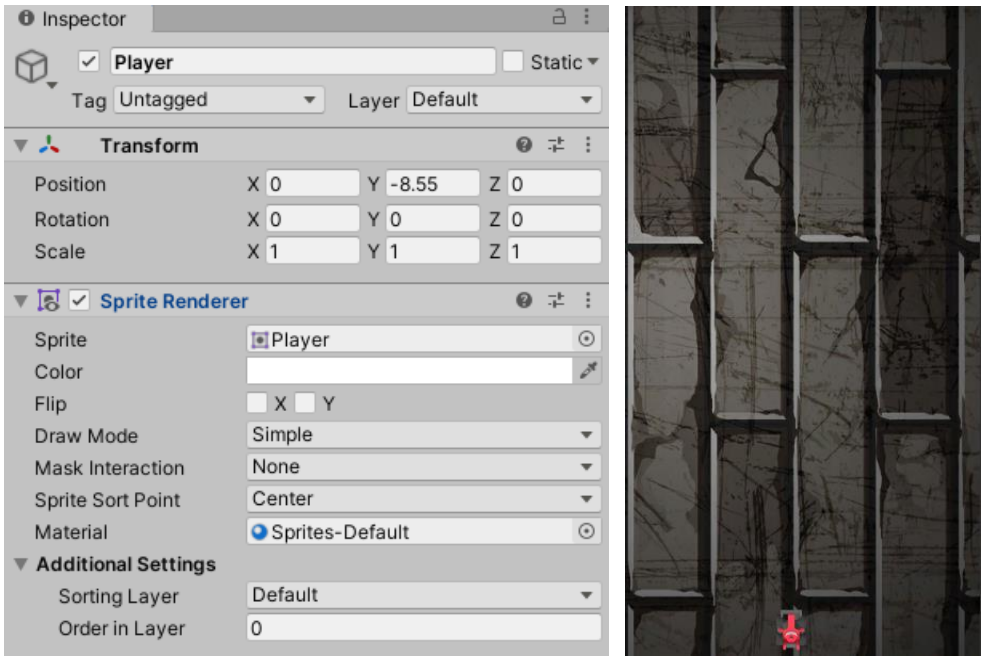
Obr. 9 Nastavenie *UI Scale Mode* a *Reference Resolution*.

Hráča pridáme tak, že v okne *Hierarchy* vytvoríme prázdny *Game Object*, ktorý pomenujeme *Player*. Pridáme mu komponent *Sprite Renderer*, kde vo vlastnosti *Sprite* nastavíme obrázok predstavujúci hráča z *Assets*, konkrétne z priečinka *Sprites*. V prípade potreby upravíme veľkosť obrázka tak, aby zapadal do herného sveta. Veľkosť objektu upravujeme pomocou vlastnosti *Pixel Per Unit* používanej textúry (Obr. 10).



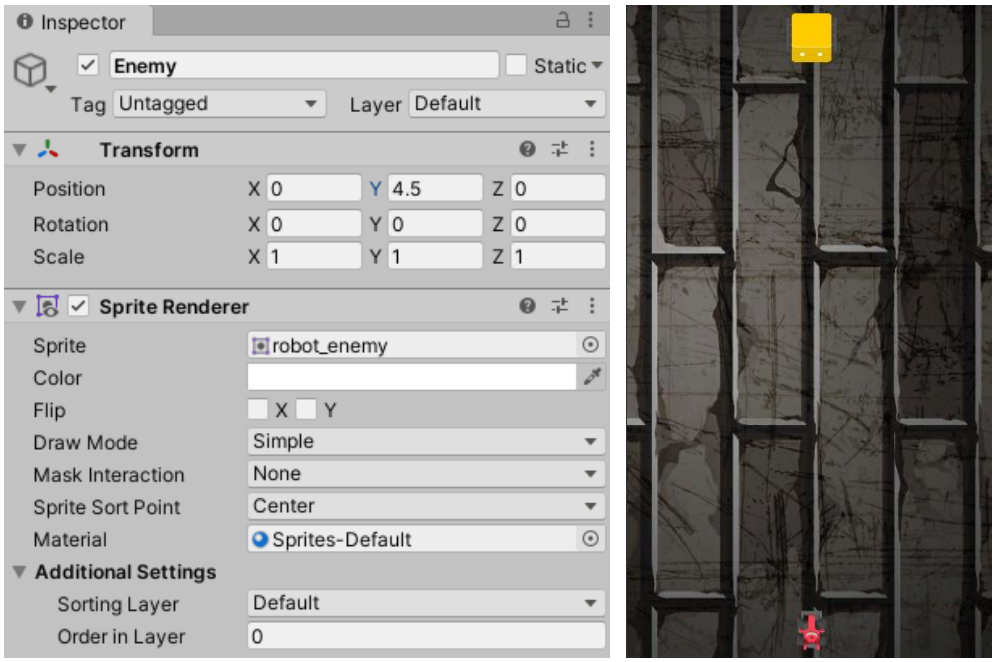
Obr. 10 Vlastnosti *spritu Player*.

Objekt umiestnime do spodnej časti obrazovky a jeho zobrazenie v smere osi z nastavíme na hodnotu inú ako má pozadie, napr. pozadie nastavíme na hodnotu 5 (Obr. 8) a objekt hráča ponecháme na hodnote 0 (Obr. 11).



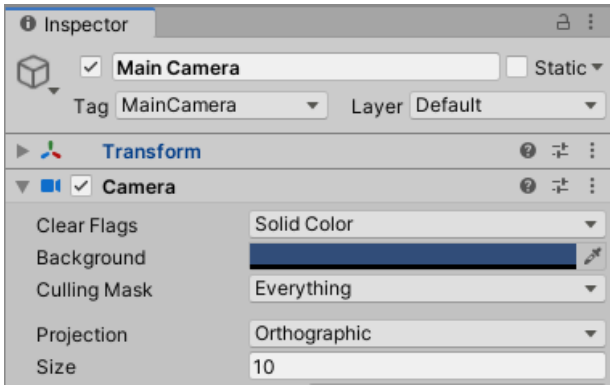
Obr. 11 Nastavenia pre objekt *Player* a jeho vizuál v scéne.

Obdobným spôsobom pridáme aj objekt predstavujúci nepriateľa – *Enemy* (Obr. 12).



Obr. 12 Nastavenia pre objekt *Enemy* a jeho vizuál v scéne.

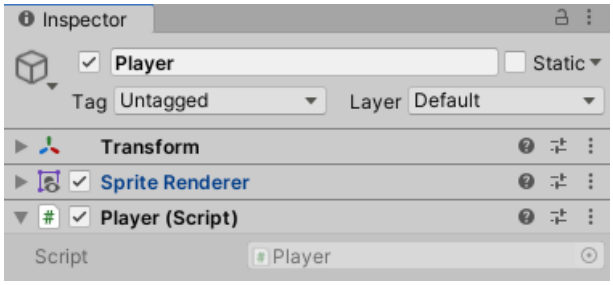
Zmenou veľkosti kamery vieme ovplyvniť veľkosť objektov v scéne, t. j. ako ich vidíme. Nastavením vlastnosti *Size* na 10 docielime zväčšenie herného priestoru (čím väčšie číslo nastavíme, tým sú objekty menšie a naopak). Dodatočne však musíme upraviť veľkosť pozadia, ako aj polohu hráča a nepriateľa.



Obr. 13 Zmena vlastnosti *Size* objektu kamera.

2.1 Pohyb hráča

Cieľom je riešiť pohyb hráča. Celú funkcionalitu budeme riešiť v skripte *Player.cs*, ktorý vytvoríme v priečinku *Scripts*. Skript pridáme ako komponent objektu hráča (Obr. 14).



Obr. 14 Objekt *Player* a jeho komponenty.

Pohyb je nutné riešiť vo funkcii *Update()*, ktorá je volaná každú snímku (*frame*) z dôvodu toho, aby sme vedeli modifikovať pohyb objektu reprezentujúceho hráča vzhľadom na vstup používateľa.

V skripte vytvoríme súkromnú metódu *Move()*. V nej definujeme dve lokálne premenné. Premenná *deltaX* reprezentuje zmenu/rozdiel v smere osi *x* na základe

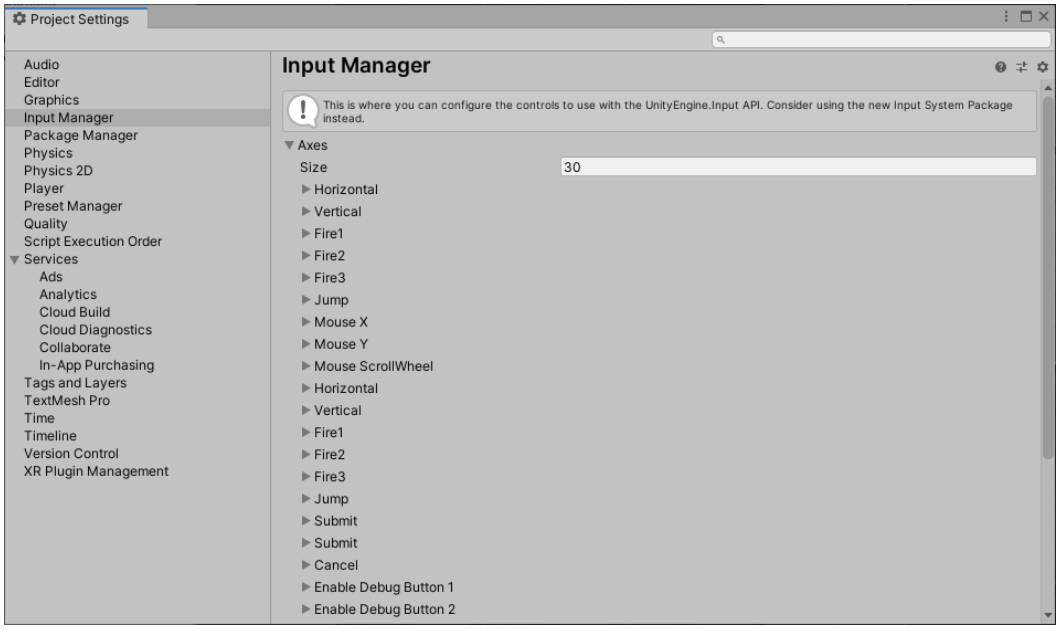
vstupu hráča, ktorá identifikuje rozdiel medzi tým, kde sa aktuálne objekt nachádza a kde chceme aby bol. Premenná *newXPos* bude reprezentovať aktuálnu pozíciu objektu v horizontálnom smere (t. j. v smere osi *x*) upravenú o hodnotu premennej *deltaX*. Obe premenné definujeme pomocou kľúčového slova *var*, ktorý sa častokrát používa pri deklarácii lokálnych premenných v momentoch, kedy vývojové prostredie (*Visual Studio*) jednoznačne vie identifikovať dátový typ premennej. V tomto prípade ide o *float*. Jednoznačné určenie je možné na základe skutočnosti, že návratovou hodnotou metódy *GetAxis()* triedy *Input* je *float*.

```
var deltaX = Input.GetAxis("Horizontal");
var newXPos=transform.position.x + deltaX;
```

Metóda *GetAxis*() vracia hodnotu v rozsahu ⟨-1, 1⟩ v smere osi, ktorá je zadefinovaná ako parameter metódy. V prípade joysticku hodnota 1 znamená jeho tlačenie doprava, hodnota -1 doľava a hodnota 0 znamená, že joystick je v neutrálnej pozícii. V prípade ovládania pomocou myši je táto hodnota určovaná trochu inak. Každopádne, pozitívnu hodnotu dostávame v prípade pohybu myši doprava (aj v prípade ak sa myš pohybuje smerom dolu) a negatívnu hodnotu v prípade pohybu doľava (aj v prípade ak sa myš pohybuje smerom hore).

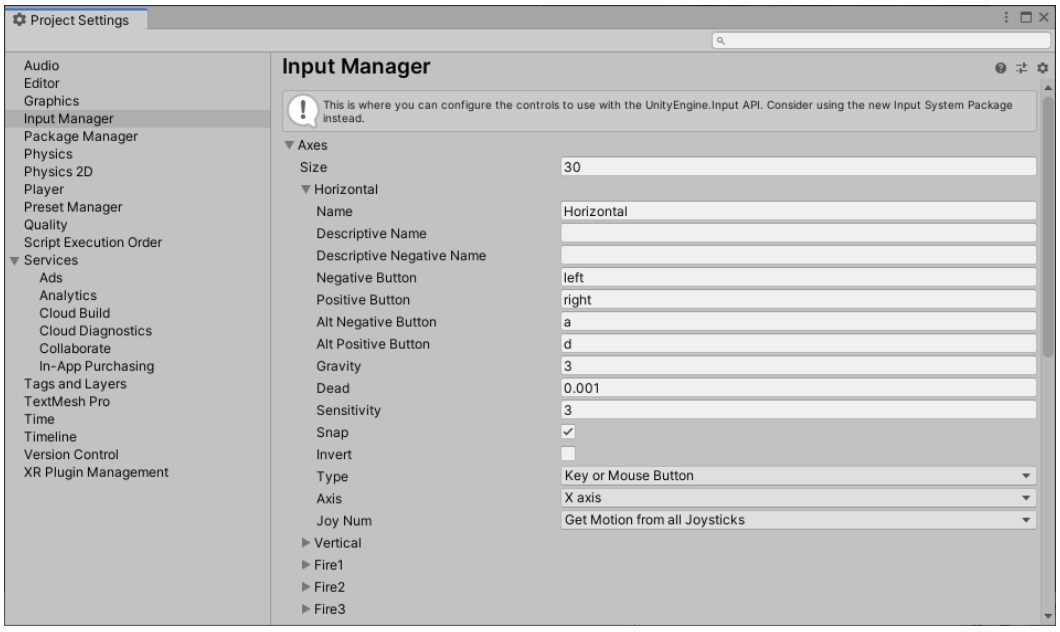
Pomocou možnosti *Edit* → *Project Settings* → *Input Manager* je možné otvoriť okno *Input Manager*. Tu je možné nastaviť, alebo kontrolovať vyhovujúce nastavenia pre možnosti vstupu hráča (*input*). Tieto zahŕňajú nastavenia pre klávesy klávesnice (*key*), tlačidlá fyzických zariadení, napr. *gamepads* (*button*), ako aj virtuálne osi (*virtual axes*), ktoré slúžia pre ovládanie pohybu (Unity, 2021a).

Po rozbalení ponuky *Axes* je možné vidieť všetky aktuálne vstupy (Obr. 15).



Obr. 15 Okno *Input Manager*.

My využijeme pohyb v smere osi *x*, preto pracujeme so vstupom s názvom *Horizontal*. Okrem vlastnosti *Name* si môžeme všimnúť aj iné vlastnosti. Z obrázka 16 je vidieť, že tento pohyb je možné ovládať aj pomocou kláves šípok, alebo klávesmi *a*, *d* reprezentované vlastnosťami *Negative/Positive Button*, resp. *Alt Negative/Positive Button*, kde Alt = *Alternative*.



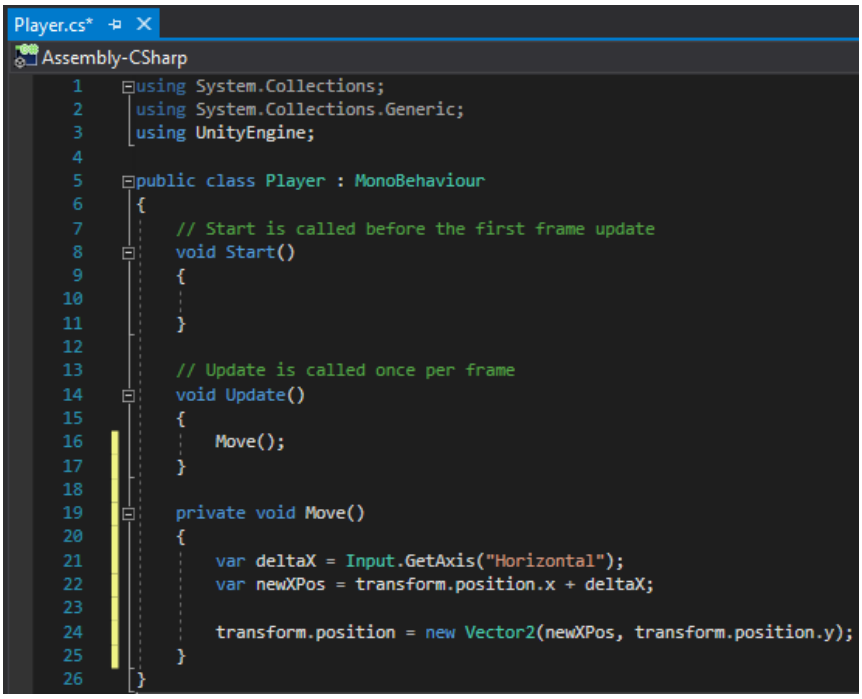
Obr. 16 Okno *Input Manager – Horizontal*.

Pozn. autora: V tomto okne je možné nájsť časť s rovnakým názvom, ale tá obsahuje nastavenia pre joystick. Toto je určené vlastnosťou *Type*, ktorá v nastaveniach na obrázku 16 jednoznačne hovorí o tom, že tieto sú pre klávesnicu a myš.

Samotnú zmenu pohybu zabezpečíme príkazom:

```
transform.position = new Vector2(newXPos, transform.position.y);
```

ktorým meníme cez vlastnosť *Position*, komponentu *Transform* súradnice na hodnoty premennej typu *Vector2*.



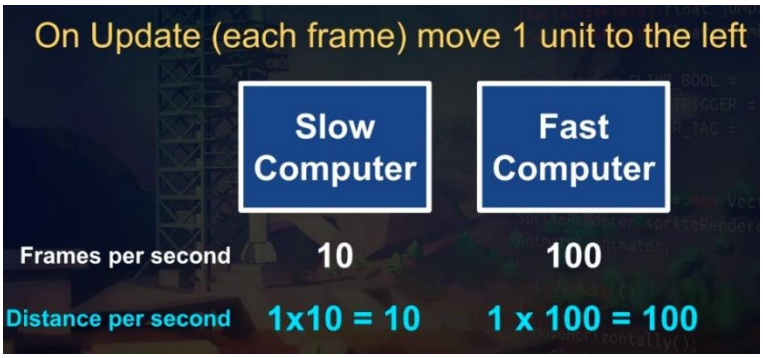
Obr. 17 Vizuál skriptu *Player.cs*.

Ak otestujeme funkčnosť kódu v editore Unity, zistíme, že tento pohyb je príliš rýchly (pri jednom stlačení tlačidla prejde objekt mimo herný priestor).

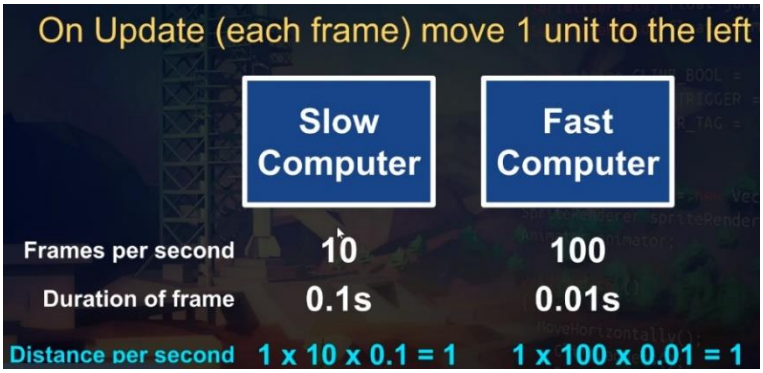
2.1.1 Zabezpečenie *frame-rate independent* hry

Vyššie identifikované nežiadúce správanie môže byť spôsobené tým, že máme vysoký FPS (*Frame Per Second*). Cieľom je teda pohyb nastaviť tak, aby bol nezávislý od FPS zariadenia, t. j. zabezpečiť aby hra bola *frame-rate independent*. Tým

dosiahneme, že herný zážitok z danej hry bude jednotný na každom zariadení, bez ohľadu na FPS. V podstate v tomto kontexte platí, že čím je vyššia hodnota FPS, tým hráč pri pohybe prejde väčšiu vzdialenosť (Obr. 18). Obrázok 19 reprezentuje túto závislosť aj s uvedením času trvania prehratia (*execute*) medzi poslednou a aktuálnou snímkou (*Duration of frame*), t. j. v čase vykonania jednej snímky (*frame*). Nie každý *frame* sa musí vykonať za ten istý čas. Túto hodnotu vieme zistiť pomocou vlastnosti *deltaTime* triedy *Time*.



Obr. 18 Objasnenie správania pri rôznych FPS (Davidson, Tristem. 2019).



Obr. 19 Objasnenie správania pri rôznych FPS (Davidson, Tristem. 2019).

Z tohto dôvodu upravíme hodnotu premennej *deltaX* tak, že ju vynásobíme *Time.deltaTime*:

```
var deltaX = Input.GetAxis("Horizontal") * Time.deltaTime;
```

Ak otestujeme funkčnosť pohybu po týchto nastaveniach, tak sa nám môže zdať, že hráč sa pohybuje pomaly. Avšak presne to bolo našim cieľom, aby sme vedeli plynulosť pohybu nastaviť aj na zariadeniach s nižším alebo vyšším FPS. Vytvoríme preto premennú *moveSpeed*, vďaka ktorej budeme nastavovať rýchlosť pohybu.

Aby sme mohli otestovať správne nastavenie tejto premennej je vhodné vytvoriť ju ako *SerializeField*.

```
[SerializeField] float moveSpeed = 10f;
```

Je nutné upraviť výpočet pre premennú *deltaX* nasledovne:

```
var deltaX = Input.GetAxis("Horizontal") * Time.deltaTime * moveSpeed;
```

Inicializácia na hodnotu 10 sa javí byť vhodnou, čo sme zistili pri testovaní pohybu v editore Unity.

Pozn. autora: Atribút *SerializeField* používame pri premenných, ktoré sú *private*, ale zároveň chceme aby sa táto premenná zobrazila v editore Unity s cieľom jej jednoduchej modifikácie pri testovaní.

Okrem zabezpečenia pohybu zľava doprava, chceme umožniť hráčovi pohybovať sa aj vo vertikálnom smere. Študujúcemu odporúčame samostatne implementovať danú funkcionalitu a výsledok skontrolovať podľa časti uvedenej nižšie. V skripte *Player.cs* definujeme dve nové premenné *deltaY* a *newYPos* pri implementovaní obdobnej funkcionality ako sme spravili v prípade horizontálneho pohybu (Obr. 20).

Player.cs*

Assembly-CSharp

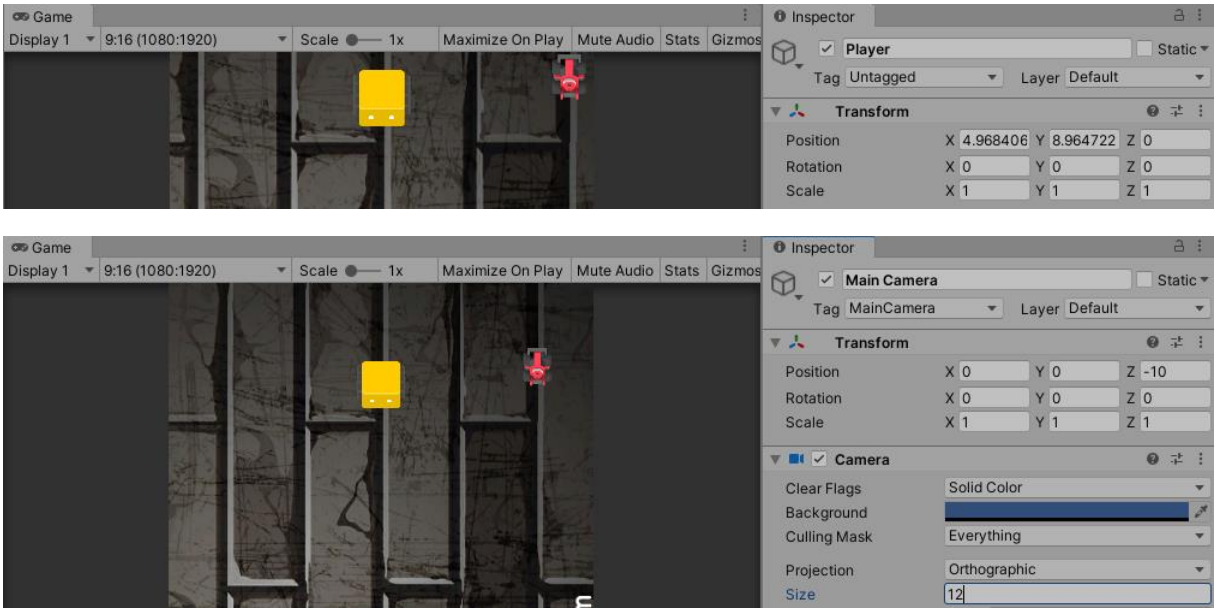
```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Player : MonoBehaviour
6  {
7      [SerializeField] float moveSpeed = 10f;
8      // Start is called before the first frame update
9      void Start()
10     {
11
12     }
13
14     // Update is called once per frame
15     void Update()
16     {
17         Move();
18     }
19
20     private void Move()
21     {
22         var deltaX = Input.GetAxis("Horizontal") * Time.deltaTime * moveSpeed;
23         var newXPos = transform.position.x + deltaX;
24
25         var deltaY = Input.GetAxis("Vertical") * Time.deltaTime * moveSpeed;
26         var newYPos = transform.position.y + deltaY;
27
28         transform.position = new Vector2(newXPos, newYPos);
29     }
30 }
```

Obr. 20 Vizuál skriptu *Player.cs*.

Testovaním implementovanej funkcionality v editore Unity by sme mali zistiť, že pohyb hráča je možný vo všetkých smeroch. Samozrejme, že v tomto štádiu môže hráč prekročiť hranice herného priestoru. Toto vyriešime následne.

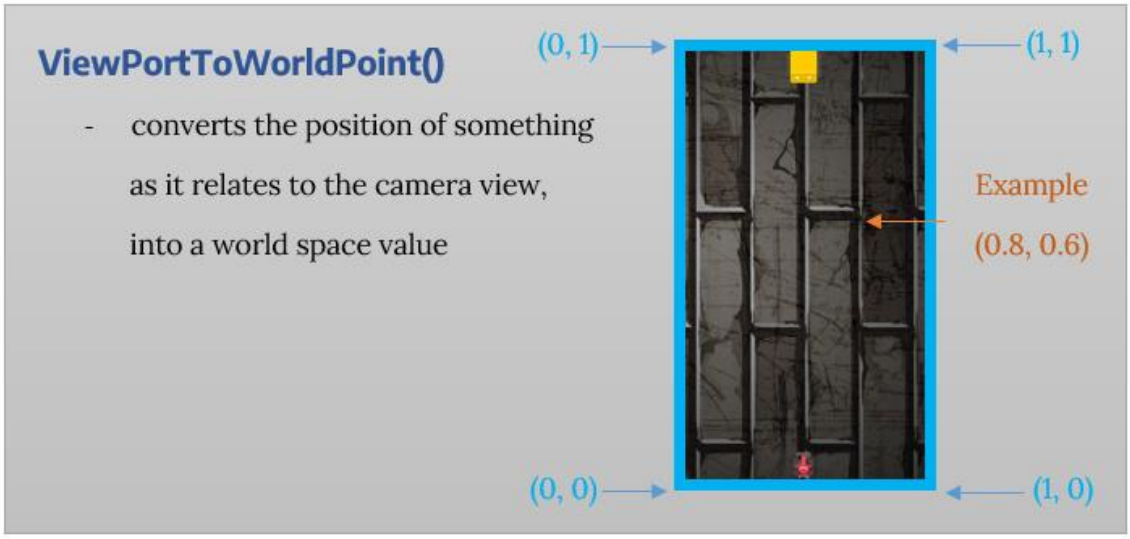
2.2 Snímanie sveta s využitím metódy *ViewportToWorldPoint()*

Pohybom hráča po hernom priestore a sledovaním zmien vo vlastnosti *Position* komponentu *Transform* objektu *Player* môžeme sledovať, ako sa tieto hodnoty menia (Obr. 21). Tieto čísla sú relatívne, t. j. pri zmene veľkosti kamery (*Size*) sa zmení zorné pole. Napríklad zmenou veľkosti kamery z hodnoty 10 na 12 si môžeme všimnúť, že hráč sa už nenachádza v pravom hornom rohu, ale že prišlo k zdanlivému zväčšeniu herného priestoru (Obr. 21). To vyvolá nutnosť zmeny pozície v smere osi x a y, ak sa chceme opätovne dostať do pravého horného rohu.



Obr. 21 Pozície objektu *Player* pred a po zmene veľkosti kamery.

Z dôvodu takéhoto správania sa preto javí ako vhodné, prekonvertovať tieto hodnoty podobne ako na obrázku 22, t. j. pohybovať sa len v rozsahu $\langle 0, 1\rangle$ v smere osi x a y.



Obr. 22 Obrázok transformácie súradníc *x* a *y*.

Predstavenú funkcionalitu zabezpečíme vytvorením metódy *SetUpMoveBoundaries*(), ktorou zabezpečíme obmedzenie pohybu hráča v hernom priestore. Metódu voláme z funkcie *Start*().

```
void Start()
{
    SetUpMoveBoudaries();
}
```

V metóde deklarujeme lokálnu premennú *gameCamera* pre identifikáciu kamery. Inicializujeme ju objektom našej kamery pomocou vlastnosti main triedy *Camera*:

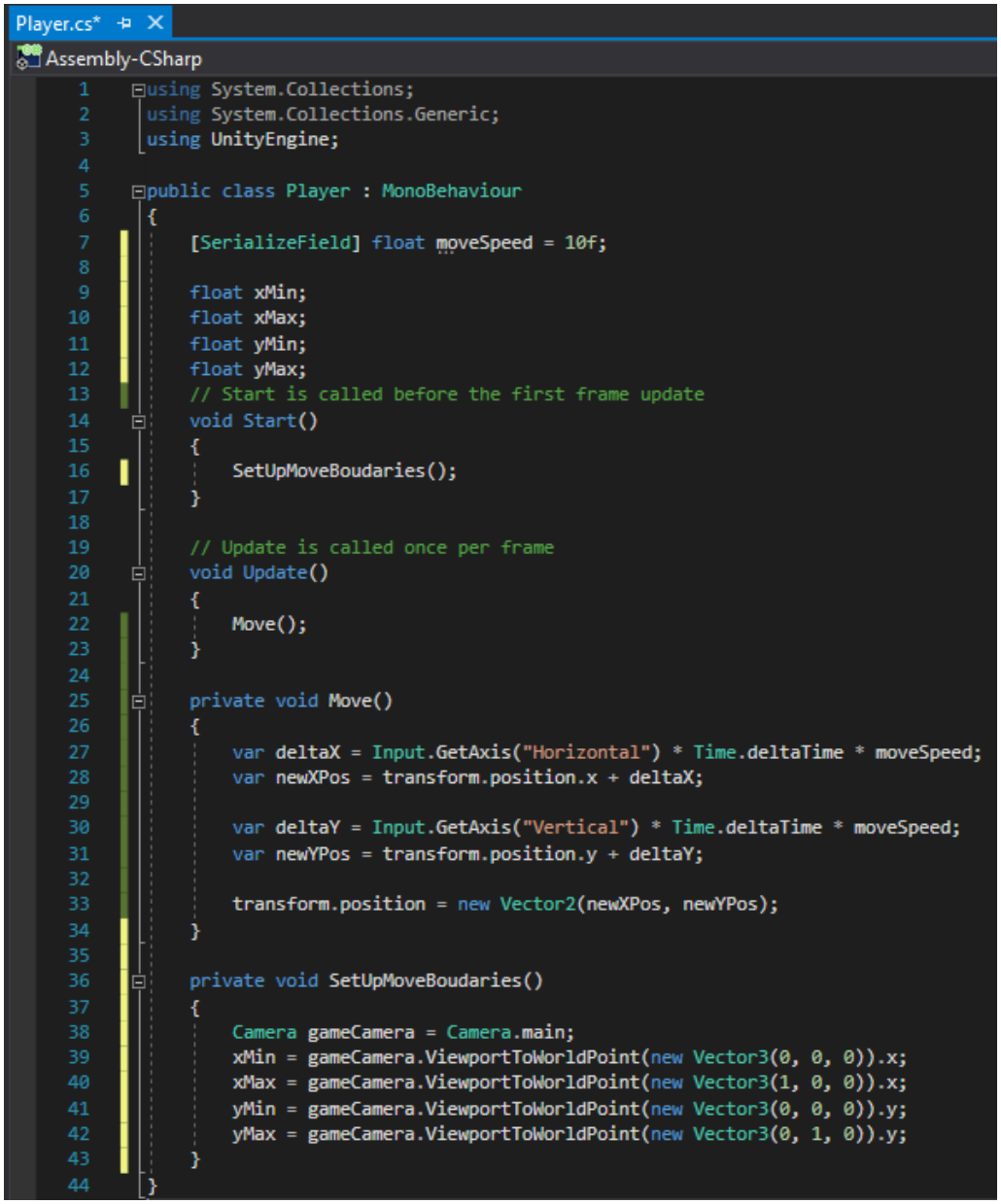
```
Camera gameCamera = Camera.main;
```

V triede *Player* zadeklarujeme štyri premenné typu *float* určujúce hranice pohybu hráča v smere osi *x*: *xMin* a *xMax* a v smere osi *y*: *yMin* a *yMax*. O ich naplnenie sa postaráme v metóde *SetUpMoveBoundaries*() metódou *ViewportToWorldPoint*(), ktorá ako parameter očakáva premennú typu *Vector3*. Z uvedené prepočtu vnímania herného priestoru ilustrovaného obrázkom 22 je zrejmé, že *xMin* = 0 a *xMax* = 1. Prepočítavanie nám zabezpečí práve metóda *ViewportToWorldPoint*(), kde na nastavenie pozície v smere osi *x* použijeme tieto príkazy:

```
xMin = gameCamera.ViewportToWorldPoint(new Vector3(0, 0, 0)).x;
xMax = gameCamera.ViewportToWorldPoint(new Vector3(1, 0, 0)).x;
```

Nastavenia v smere osi *y* zrealizujeme príkazmi:

```
yMin = gameCamera.ViewportToWorldPoint(new Vector3(0, 0, 0)).y;
yMax = gameCamera.ViewportToWorldPoint(new Vector3(0, 1, 0)).y;
```



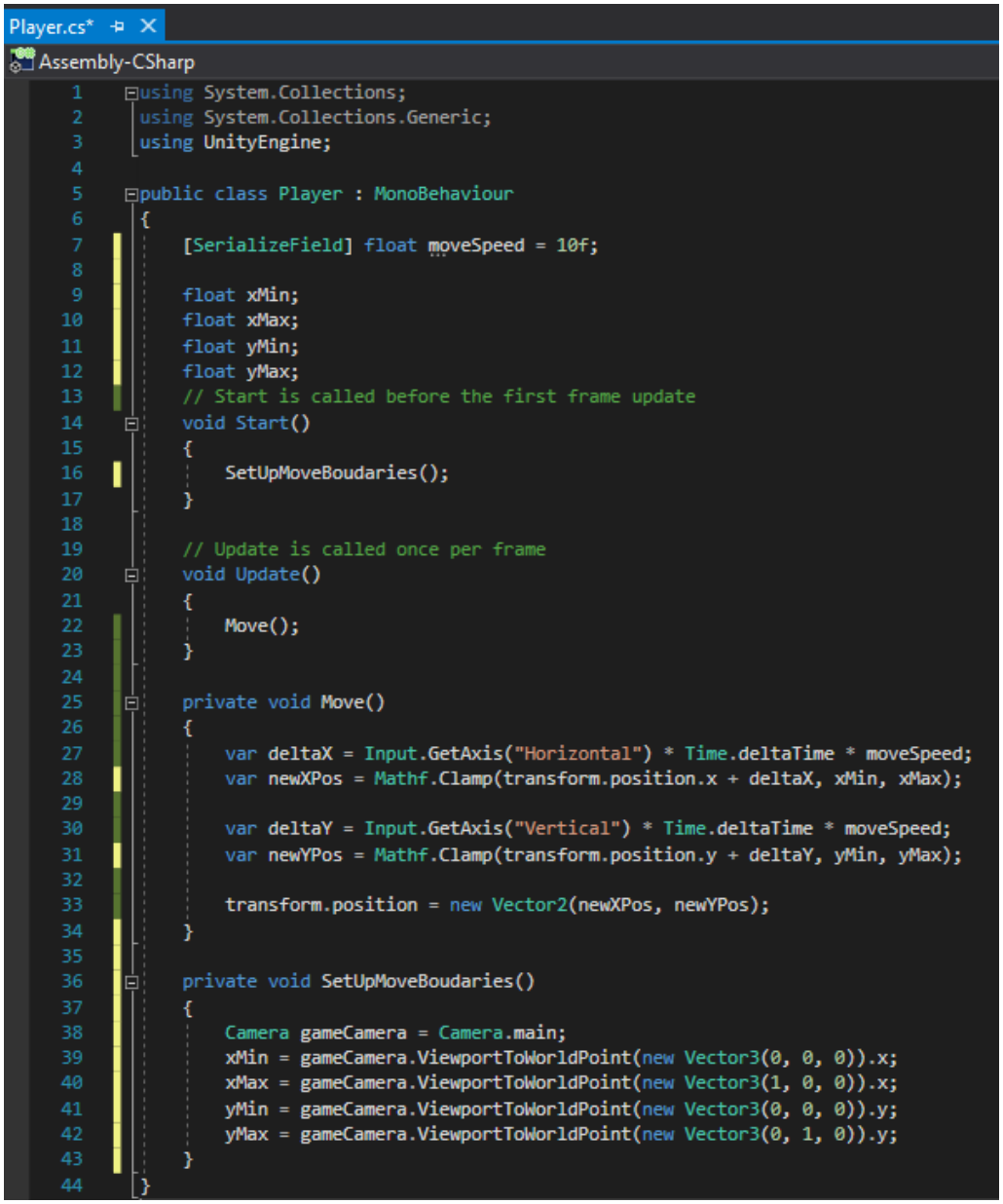
Obr. 23 Vizuál skriptu *Player.cs*.

Tieto zmeny sa musia odzrkadliť v prípade určovania pozície hráča v smere osi *x* a *y*. Cieľom je zabezpečiť, aby sa hráč mohol pohybovať len vo vymedzenom priestore. K tomu využijeme metódu *Clamp()* triedy *Mathf*, ktorou vieme zadefinovať požadované obmedzenia.

Upravujeme teda nastavenia premenných *new*XPos a *new*YPos deklarovaných v metóde *Move*() nasledovne:

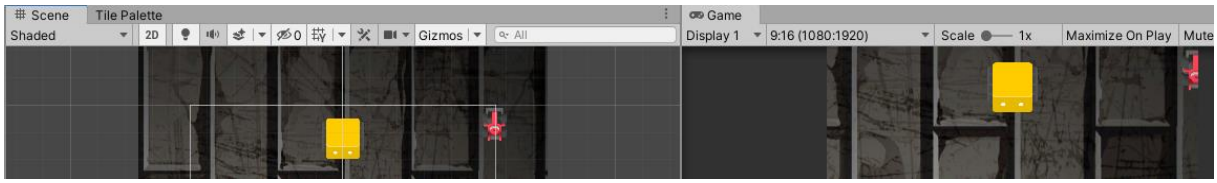
```
var newXPos=Mathf.Clamp(transform.position.x + deltaX, xMin, xMax);
var newYPos=Mathf.Clamp(transform.position.y + deltaY, yMin, yMax);
```

Obrázkom 24 ilustrujeme celý skript *Player*.cs po zrealizovaných zmenách.



Obr. 24 Vizuál skriptu *Player*.cs.

Po návrate do editora Unity otestujeme implementovanú funkcionalitu. Môžeme si všimnúť, že hranice sveta fungujú korektne. Avšak pri pohybe do rohov, resp. hrán si môžeme všimnúť, že objekt hráča ako keby sa dostal mimo hraníc herného sveta, ilustrované obrázkom 25.



Obr. 25 Zachytenie pozície hráča v pravom hornom rohu.

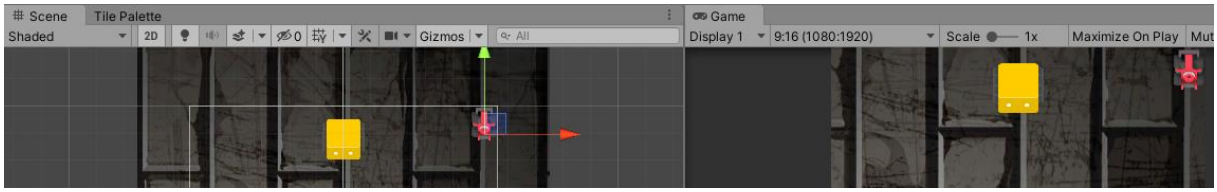
Tento jav je predvídateľný, pretože pivot objektu je v strede a implementovaná funkcionalita reaguje vzhľadom na tento bod. V skripte *Player.cs* definujeme premennú *padding*, ktorú inicializujeme na hodnotu 1. S touto hodnotou budeme experimentovať pri testovaní hry, aby sme našli jej vhodné nastavenie.

```
[SerializeField] float padding = 1f;
```

V metóde *SetUpMoveBoundaries*() upravíme inicializáciu premenných nasledovne:

```
xMin = gameCamera.ViewportToWorldPoint(new Vector3(0, 0, 0)).x + padding;
xMax = gameCamera.ViewportToWorldPoint(new Vector3(1, 0, 0)).x - padding;
yMin = gameCamera.ViewportToWorldPoint(new Vector3(0, 0, 0)).y + padding;
yMax = gameCamera.ViewportToWorldPoint(new Vector3(0, 1, 0)).y - padding;
```

Testovaním funkčnosti zistíme, že premennú *padding* je ideálne nastaviť na hodnotu 0,5. Výsledok našich nastavení ilustruje obrázok 26.



Obr. 26 Vizualizácia pohybu hráča s obmedzením pohybu v hernom svete.

2.3 Kontrolné otázky a úlohy

1. Definujte objekt *Canvas*. Aká je jeho základná funkcionalita?
2. Čo vyjadruje termín *Aspect Ratio*?
3. Čo ovplyvňuje a ako funguje vlastnosť obrázka (*spritu*) *Pixel Per Unit*?
4. Ako vieme nastavovať vstupy používateľa na ovládanie jednotlivých udalostí v hrách?
5. V akom prípade vieme využiť kľúčové slovo *var* pri deklarácii premenných?
6. Z akého dôvodu využívame vlastnosť *deltaTime* triedy *Time*?
7. Čo znamená, ak premenné deklarujeme ako *SerializeField*?
8. Definujte metódu *ViewportToWorldPoint*(). V akom kontexte ju vieme využiť?
9. Definujte metódu *Clamp*() triedy *Mathf*.

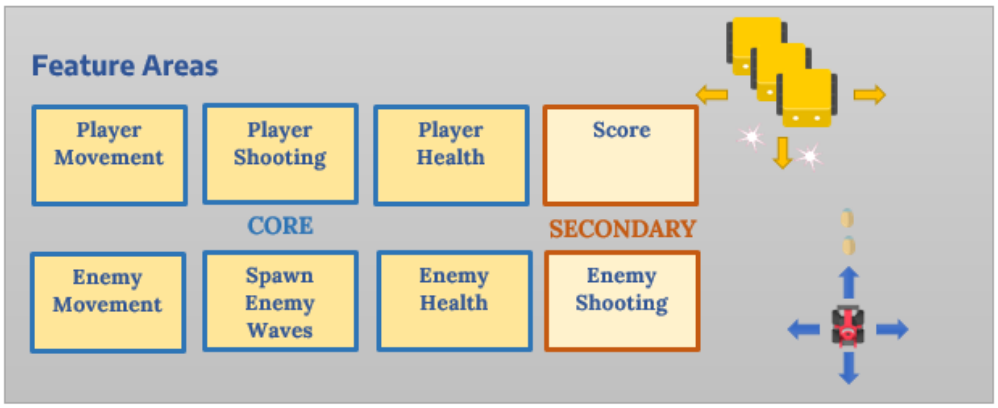
3 Ako postupovať pri vývoji hry

Pri každom riešení komplexného problému je vhodné tento dekomponovať na menšie, a tieto následne riešiť. V prípade vývoja hry sa vo všeobecnosti vždy začína implementáciou, prípadne spracovaním vstupov pomocou ktorých bude hráč ovládať hru. V našom prípade je to funkcionalita riešenia pohybu hráča, ktorú sme zrealizovali v kapitole 2. Obrázkom 27 ilustrujeme hlavné oblasti (*core feature areas*), ktoré je v hre nutné riešiť. Môžeme konštatovať, že oblasť *Player Movement* máme vyriešenú. Obrázok ilustruje pohyb hráča aj nepriateľa, ako i možnosti ich streľby. Preto ďalšími oblasťami na riešenie sú:

- *Player Shooting* – hráč môže strieľať,
- *Player/Enemy Health* – systém podľa ktorého budeme vedieť reagovať na to, či má hráč/nepriateľ zomrieť,
- *Enemy Movement* – cieľom je zabezpečiť pohyb nepriateľa po vopred definovanej ceste (*path*),
- *Spawn Enemy Waves* – cieľom je vytvoriť rôzne zhluky nepriateľov, ktorí sa budú vedieť pohybovať po rôznych cestách.

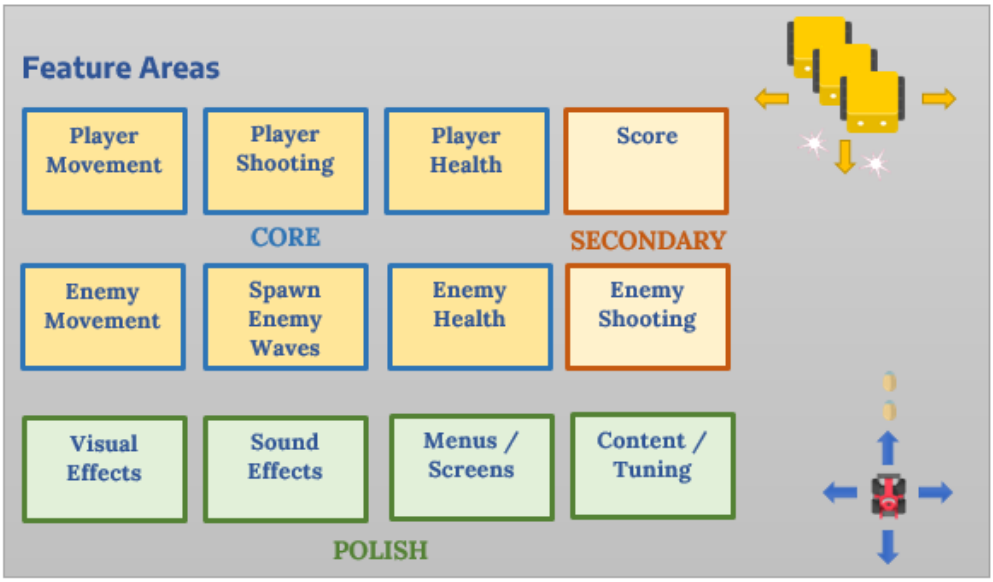
Obr. 27 Hlavné oblasti riešené pri vývoji hry.

Pri vývoji sa bežne definujú aj sekundárne ciele (oblasti). Tieto predstavujú také funkcionality, že hra by bola hrateľná aj bez ich implementácie (Obr. 28).



Obr. 28 Hlavné a vedľajšie oblasti riešené pri vývoji hry.

Okrem toho sa pri vývoji snažíme zvýšiť aj pôžitok z danej hry. Tieto oblasti zväčša zahŕňajú doplnenie hry o rôzne vizuálne a zvukové efekty, ponuku (*menu*) pre hráča, aby hráč vedel interagovať s hrou, prerušiť ju a pod. Veľká pozornosť sa venuje testovaniu a ladeniu hry, aby priniesla čo najlepší herný zážitok. V našom prípade môže ísť napríklad o nastavenie rôznych ciest nepriateľov, ovplyvňovanie ich rýchlosti pohybu, rýchlosti a sile ich útoku, možnosti ich znovuzrodenia, zmene dizajnu a pod. Táto časť je reprezentovaná na obrázku 29 kategóriou *Polish*, ktorú by sme mohli voľne preložiť ako fáza dolaďovania hry.



Obr. 29 Hlavné, vedľajšie a dolaďovacie oblasti riešené pri vývoji hry.

Otázkou by teda mohlo byť, ktorú ďalšiu funkcionalitu budeme ďalej implementovať. Je zrejmé, že budeme pokračovať oblasťami z hlavných cieľov. Len ťažko by sme sa mohli venovať správe životov hráča, ak nemáme ešte implementovaný systém útoku (*damage system*) a hráč ešte nevie strieľať. Okrem toho streľba hráča sa javí ako jedna zo základných funkcionalít. Zdá sa byť viac než logické implementovať funkcionalitu streľby hráča, ktorej logiku neskôr využijeme aj v prípade riešenia streľby nepriateľov. Vedľajšie ciele, ako aj dolaďovacie záležitosti sa zväčšia riešia až neskôr.

Pozn. autora: Viacerí zastávajú názor, že častiam z *Polish* kategórie sa venujú až na konci. Avšak ja to robím priebežne, aby som už počas vývoja videla, ako sa prvky v hre budú správať a vyzerať. Potom je aj priebežné testovanie zábavnejšie. Položte si otázku: nie je krajšie mať hneď od začiatku nastavené pozadia, požadované vizualizácie pre hráča a nepriateľa, ako na ich vizualizáciu použiť napríklad len základné geometrické objekty? Podľa odpovede je zrejmé ku ktorej skupine vývojárov sa radíte.

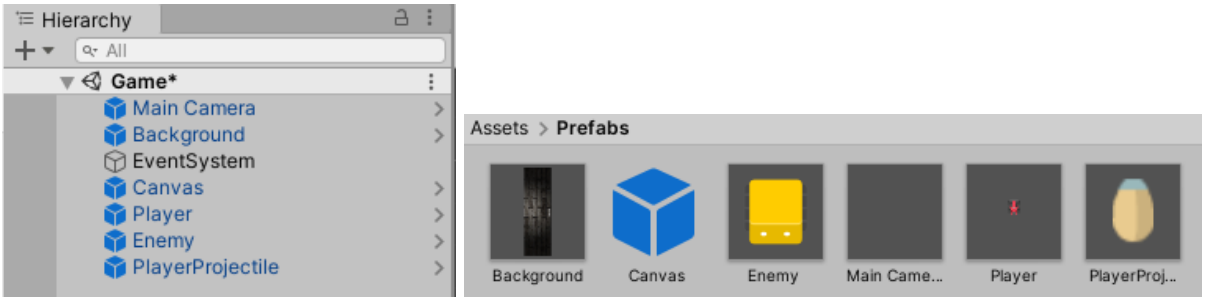
3.1 Zabezpečenie streľby hráčom

Ako prvé si vyberieme vhodnú vizuálnu reprezentáciu projektilu a pridáme ho do priečinku *Sprites*. Vytvoríme z neho inštanciu tak, že ho presunieme do okna *Hierarchy* a pomenujeme *PlayerProjectile* (Obr. 30).

Obr. 30 Pridanie *spritu* pre projektil hráča.

V *Assets* vytvoríme nový priečinok s názvom *Prefabs*, kde presunieme všetky objekty zo scény ako ilustruje *obrázok 31*. V okne *Hierarchy* je táto skutočnosť

zvýraznená zmenou farby objektov na modrú. Robíme to z dôvodu, že ak budeme chcieť jednotne modifikovať vlastnosti jednotlivých inštancií použitých v scéne, bude sa nám komfortnejšie pracovať. Viac sa študujúci o problematike *prefabov* dočíta v skriptách autorky (Jurinová, 2022).

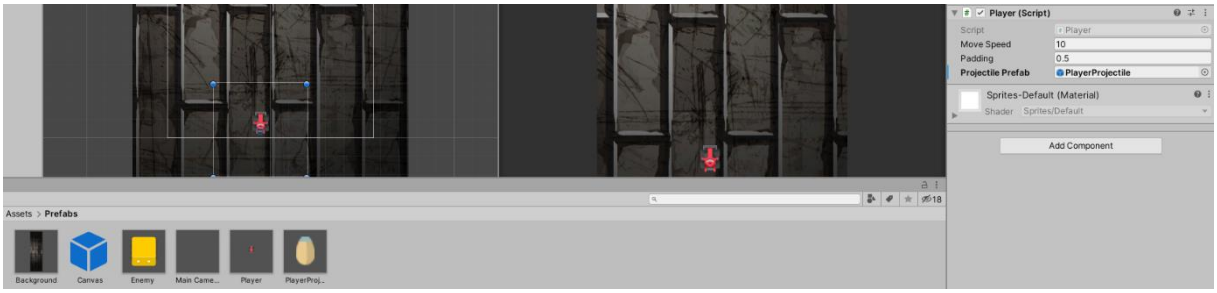


Obr. 31 Vytvorenie *prefabov* z objektov scény.

Následne v skripte *Player.cs* vytvoríme novú konfiguračnú premennú pre našu strelu. Premenná prirodzene bude dátového typu *GameObject*.

```
[SerializeField] GameObject projectilePrefab;
```

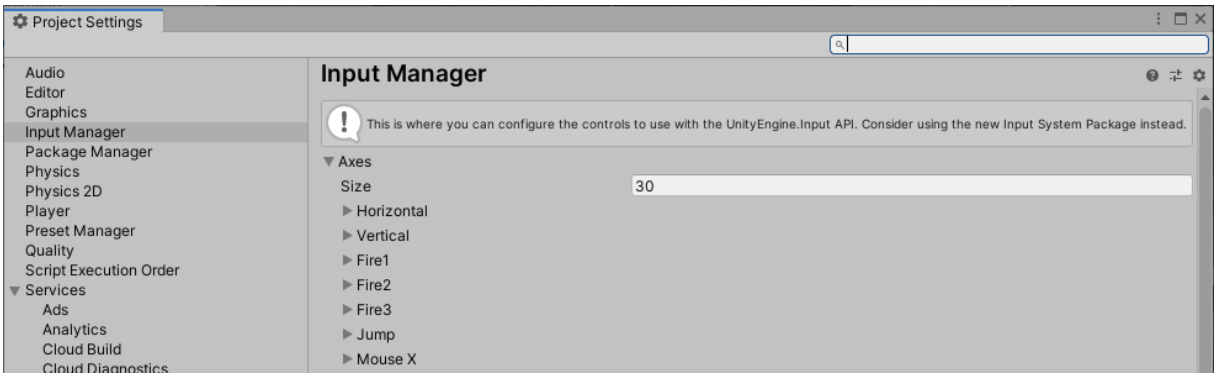
Po uložení zmien môžeme v editore Unity v okne *Hierarchy* vymazať objekt *PlayerProjectile*, pretože zabezpečíme jeho zrodenie (*spawn*) pomocou kódu. Inicializáciu premennej *projectilePrefab* zrealizujeme presunutím *prefabu PlayerProjectile* do premennej v komponente *Player* (Obr. 32).



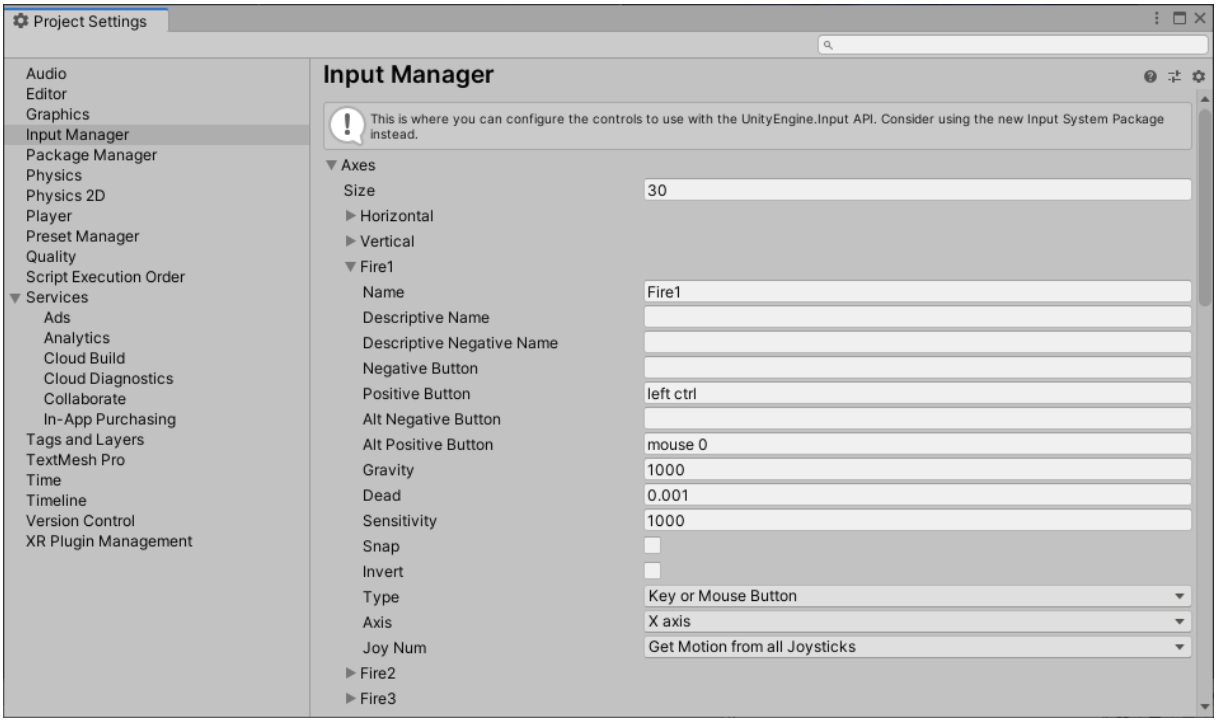
Obr. 32 Zachytenie nastavení premennej *projectilePrefab* vo vlastnostiach komponentu.

Vytvorenie strely riešime dynamicky pomocou kódu. Vo funkcii *Update*() zavoláme metódu *Fire*(), ktorú následne definujeme. Vystrelenie zabezpečíme na stlačenie medzery (pre prípad ak hráč používa pri hre klávesnicu), alebo na stlačenie ľavého tlačidla myši (ak preferuje ovládanie streľby myšou). Je nutné upraviť vstupy pre možnosť *Fire1*. V okne *Project Settings* otvoreného cez možnosť *Edit*, si môžeme

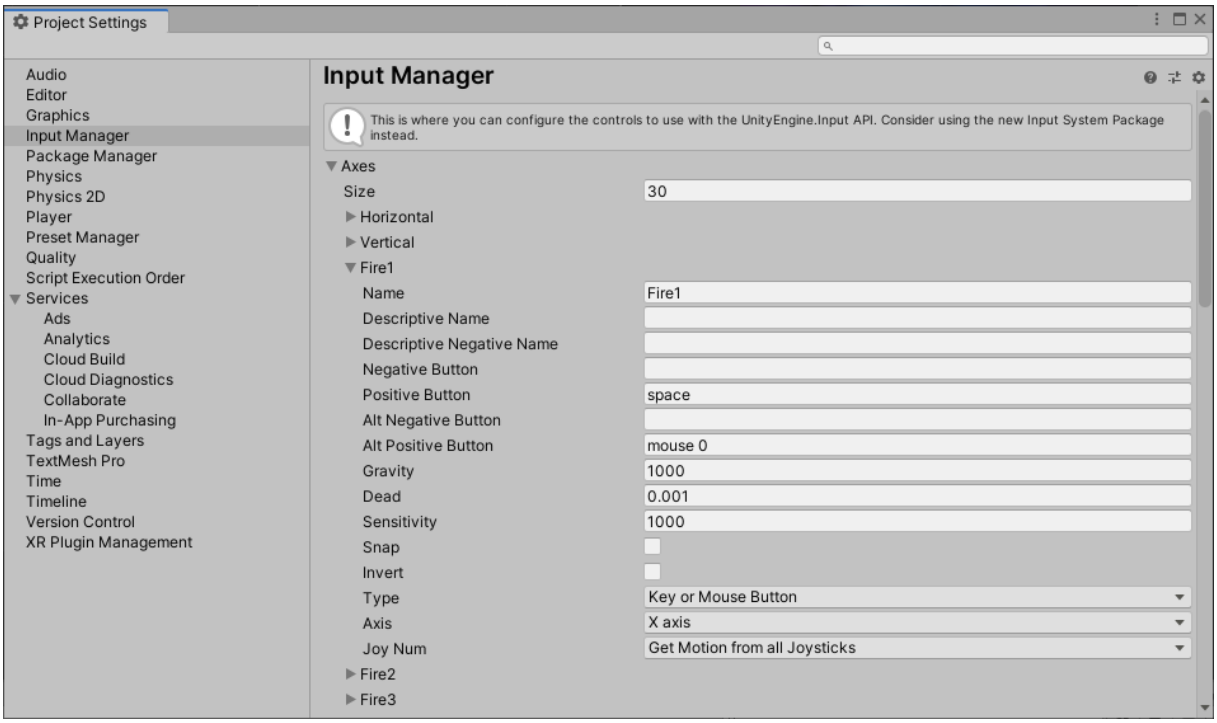
všimnúť aj možnosti *Fire2* a *Fire3* (Obr. 33). Je teda zrejmé, že je možné implementovať viacero rôznych možností streľby. My si vystačíme s možnosťou *Fire1*, kde štandardné nastavenie pre streľbu v prípade použitia klávesnice je pomocou ľavej klávesy Ctrl (Obr. 34), ktorú zmeníme na medzeru (*Space*) a pri použití myši (štandardne ľavé tlačidlo myši) ponecháme bezo zmeny (Obr. 35).



Obr. 33 Okno *Input Manager*.



Obr. 34 Pôvodné nastavenia pre *Fire1*.



Obr. 35 Upravené nastavenia pre *Fire1*.

Pozn. autora: V tomto okne je možné nájsť ešte jednu možnosť *Fire1*, kde sa nachádzajú nastavenia pre joystick.

Následne implementujeme metódu *Fire*(), kde sa postaráme o vytvorenie inštancie strely pomocou metódy *Instantiate()* triedy Object, pri stlačení medzery na klávesnici, alebo ľavého tlačidla myši. Zachytenie vstupu robíme pomocou metódy *GetButtonDown()* triedy Input.

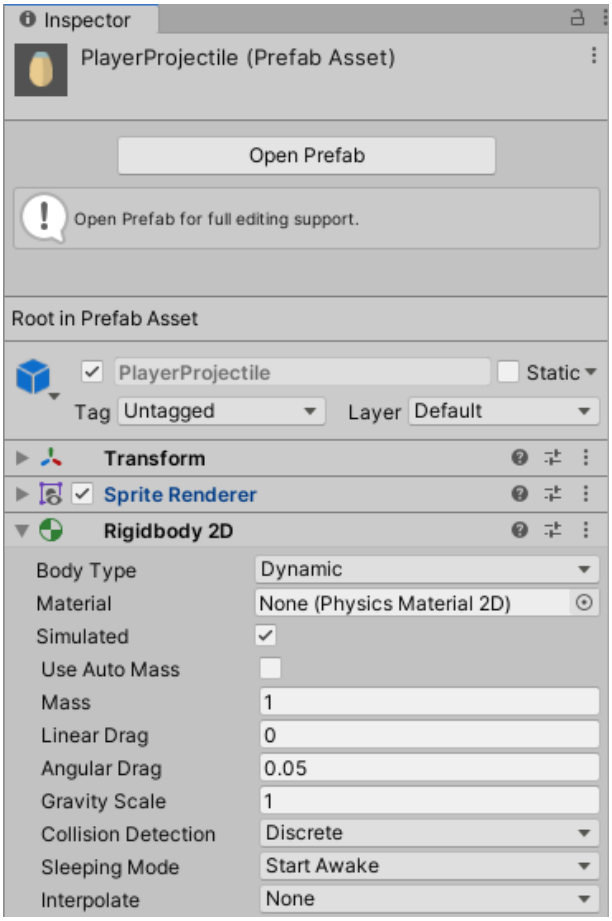
Pri metóde *Instantiate*() používame túto variantu:

```
public static Object Instantiate
       (Object original, Vector3 position, Quaternion rotation);
```

kde prvý parameter reprezentuje objekt, v našom prípade *prefab PlayerProjectile*. Druhý parameter špecifikuje pozíciu objektu, ktorý vznikne presne tam, kde je hráč. Posledný parameter určuje rotáciu objektu, ktorú nastavíme bez rotácie pomocou vlastnosti *identity* štruktúry *Quaternion*. *Quaternion* sa v Unity využíva pre reprezentáciu rotácie objektov.

```
private void Fire()
    {
        if (Input.GetButtonDown("Fire1"))
        {
            Instantiate(projectilePrefab, transform.position, Quaternion.identity);
        }
    }
```

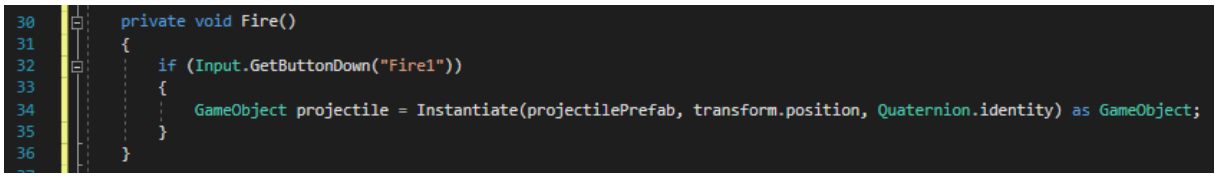
Funkcionalita vzniku strely je zabezpečená. Potrebujeme však zabezpečiť aj pohyb tejto strely. *Prefabu PlayerProjectile* pridáme komponent *RigidBody2D* (Obr. 36). Pridanie komponentu robíme z toho dôvodu, že pre riešenie pohybu strely využijeme jeho vlastnosť *Velocity*.



Obr. 36 Nastavenia *prefabu PlayerProjectile*.

V skripte *Player.cs* upravíme proces vytvorenia strely v metóde *Fire()* tak, že definujeme lokálnu premennú *projectile* takto:

```
GameObject projectile =
   Instantiate(projectilePrefab, transform.position, Quaternion.identity) as GameObject;
```



Obr. 37 Vizuál upravenej metódy *Fire()*.

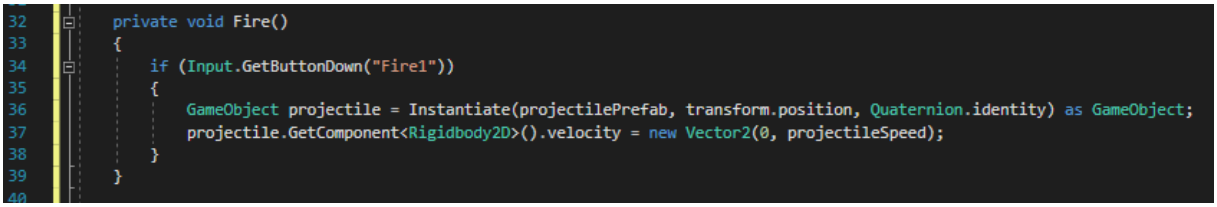
V triede vytvoríme novú premennú *projectileSpeed*, pomocou ktorej budeme ovládať rýchlosť pohybu strely.

```
[SerializeField] float projectileSpeed = 10f;
```

Pomocou vlastnosti *velocity* a použitiu komponentu *Rigidbody2D* zabezpečíme pohyb strely v smere osi *y* pri každom vystrelení. Z toho dôvodu doplníme do metódy *Fire()* príkaz:

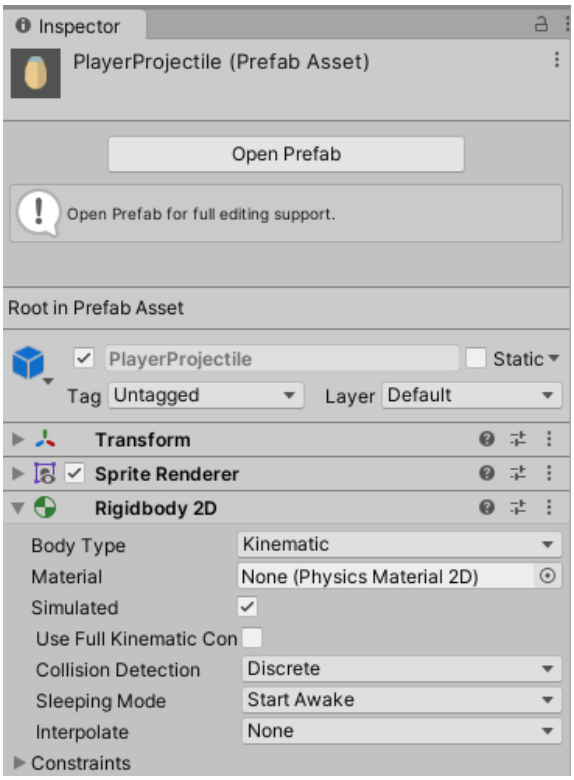
```
projectile.GetComponent<Rigidbody2D>().velocity = new Vector2(0, projectileSpeed);
```

kde nastavenie pre *Vector2* je v smere osi *x* nulové, a v smere osi *y* využijeme premennú *projectileSpeed*, ktorej vhodnú inicializáciu otestujeme v editore Unity.



Obr. 38 Vizuál upravenej metódy *Fire()*.

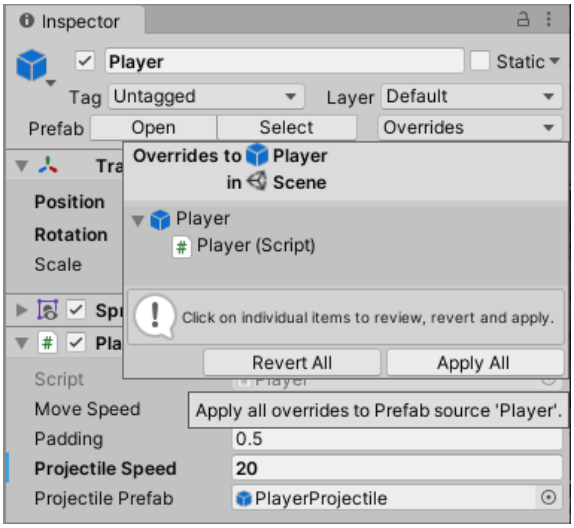
Pri testovaní funkčnosti môžeme pozorovať, že na vystrelené strely pôsobia fyzikálne zákonitosti. Je vhodné nastaviť vlastnosť *Body Type* v komponente *Rigidbody 2D* na *Kinematic* (Obr. 39). To spôsobí, že sa na neho nebudú aplikovať žiadne fyzikálne zákonitosti.



Obr. 39 Nastavenie vlastnosti *Body Type* na prefabe *PlayerProjectile*.

Ak sa vám zdá, že rýchlosť streľby je pomalá, môžete zmeniť hodnotu premennej *projectileSpeed*.

Pozn. autora: V prípade modifikácie premennej *projectileSpeed* v editore Unity treba zmeny aplikovať na *prefab* pomocou *Apply All*. Kým je zmena zvýraznená tučným písmom (*boldom*), znamená to, že sa zmeny ešte neaplikovali na *prefab*.



Obr. 40 Úprava hodnoty premennej *projectileSpeed* v editore Unity.

Pri testovaní si môžeme všimnúť aj skutočnosť, že pri držaní klávesy *Space*, alebo ľavého tlačidla myši, je generovaná len jedna strela. Túto funkcionalitu upravíme tak, aby sa generovala postupne vždy len jedna strela s nastaveným časovým rozostupom medzi vzniknutými streľami v prípade držania vstupu pre streľbu.

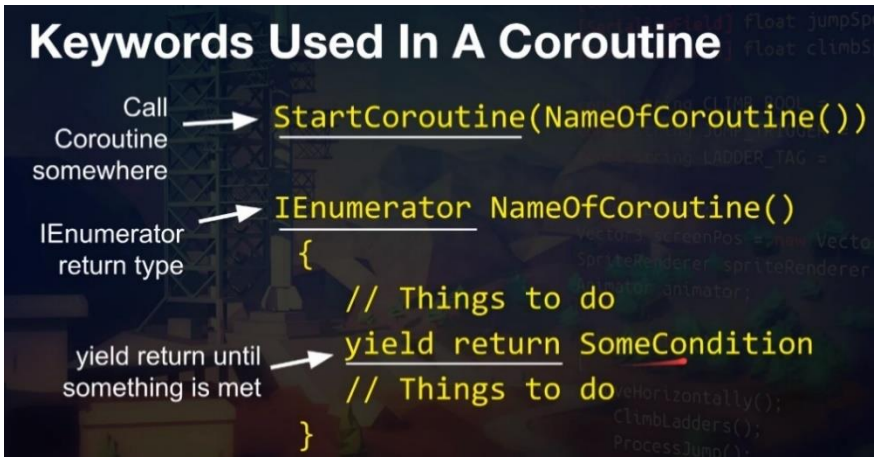
3.1.1 Použitie korutín (*Coroutines*)

Korutina (*coroutine*) je v podstate metóda, ktorá umožňuje pozastaviť vykonávanie a vrátiť obsluhu Unity, ktorý môže prejsť k prehratiu ďalšej snímky (*frame*). Korutina v podstate dovoľuje rozšíriť spracovávanú úlohu aj cez viacero snímok. Vo väčšine prípadov volania funkcií sa činnosť v nich obsiahnutá, musí vykonať za jednu snímku (*single frame update*). Ak ale potrebujeme zavolať metódu, ktorá obsahuje napríklad prehratie animácie (jej prehratie trvá viac ako jeden *frame*) je nutné použiť korutinu. Je však dôležité si uvedomiť, že korutina nie je to isté ako vlákno (*threads*), ktoré poznáme aj z iných programovacích jazykov. Všetky operácie vykonávajúce sa v korutine sú stále vykonávané hlavným vláknom (*main thread*) (Unity, 2021b).

V kontexte hier korutinu využívame ak má dôjsť k nejakej udalosti (procesu), kedy je nutné prerušiť hru a vykonať túto udalosť. Napríklad, ak by hráč dosiahol nulové skóre, tak zavoláme korutinu *KillPlayer*(), ktorá sa postará o prehratie animácie smrti hráča. Počká 3 sekundy, čo je dĺžka/doba prehratia animácie, a pokračuje sa napríklad zobrazením ponuky pre znovu hranie hry (levelu). Prípadne by sme mohli zrealizovať pozastavenie vykonávania hry, kým sa neprehrá animácia smrti hráča, a následne v rámci tejto korutiny reštartujeme daný level. Ak by sme nepoužili na riešenie korutinu, tak by sme potrebovali prehrať danú animáciu smrti v tom istom *frame* v ktorom by dochádzalo k reštartovaniu hry, čo by spôsobilo, že by sme jej prehratie vlastne nikdy nevideli.

V prípade záujmu o využívanie viacvláknového programovania odporúčame v Unity využívať C# Job System. Táto problematika je však nad rámec týchto skrípt.

Aby sme vedeli s korutinou pracovať, musíme vedieť použiť niekoľko kľúčových slov ilustrovaných obrázkom 41.



Obr. 41 Kľúčové slová korutiny (Davidson, Rick and Ben Tristen. 2019).

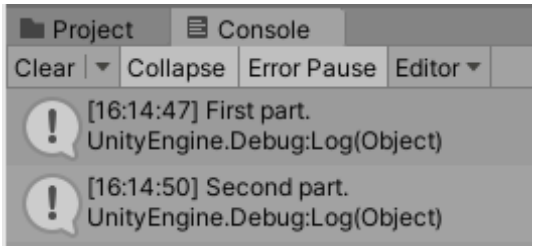
Korutinu je možné vyvolať pomocou funkcie StartCoroutine(). Zavolanie korutiny *KillPlayer()* by bolo v tvare: *StartCoroutine(KillPlayer())* a došlo by k nemu napríklad v prípade dosiahnutia nulového skóre hráča. Každá korutina musí mať návratovú hodnotu typu IEnumerator, a v jej tele musí byť príkaz uvedený pomocou kľúčových slov *yield return*.

Objasníme si to na veľmi jednoduchom príklade kedy pomocou korutiny vypíšeme do konzoly nejakú informáciu, počkáme 3 sekundy (čo ilustruje vykonanie nejakej akcie) a následne vypíšeme do konzoly inú informáciu. Pozastavenie vykonávania zrealizujeme pomocou metódy WaitForSeconds(). Korutinu zavoláme z funkcie *Start()*.

```
void Start()
    {
        SetUpMoveBoundaries();
        StartCoroutine(TestCoroutine());
    }
```

```
IEnumerator TestCoroutine()
    {
        Debug.Log("First part.");
        yield return new WaitForSeconds(3);
        Debug.Log("Second part.");
    }
```

Pri testovaní funkčnosti v editore Unity môžeme sledovať výpisy priamo v konzole (Obr. 42).



Obr. 42 Výpis do konzoly zrealizovaný pomocou korutiny.

Pozn. autora: Ak by sme korutinu volali vo funkcii *Update*(), tak by sme v konzole mohli sledovať, ako sa tieto výpisy po 3 sekundách menia. K zavolaniu korutiny by dochádzalo v každom jednom *frame*.

Tento koncept teraz použijeme pre zabezpečenie funkcionality, že hráč v prípade držania vstupu pre streľbu bude mať opakovane generované strely, a v prípade len stlačenia vstupu nastane vygenerovanie jednej strely.

3.1.2 *FireContinously() Coroutine*

Implementovanú metódu *Fire*() v skripte *Player.cs* prerobíme tak, že v nej budeme volať korutinu *FireContinously*(). Premennú *projectileFiringPeriod* použijeme pre určovanie časových rozostupov medzi generovaním jednotlivých striel. Celú funkcionalitu zapúzdrime do nekonečného cyklu. Zastavenie korutiny zrealizujeme pomocou metódy StopAllCoroutines() triedy MonoBehavior.

```
    [SerializeField] float projectileFiringPeriod = 0.1f;
```

Korutina:

```
IEnumerator FireContinously()
{
    while(true)
    {
        GameObject projectile =
          Instantiate(projectilePrefab, transform.position, transform.rotation)
          as GameObject;
        projectile.GetComponent<Rigidbody2D>().velocity =
          new Vector2(0, projectileSpeed);
        yield return new WaitForSeconds(projectileFiringPeriod);
    }
}
```

Volanie a zastavenie korutiny vzhľadom na vstup používateľa:

```
private void Fire()
{
    if (Input.GetButtonDown("Fire1"))
    {
        StartCoroutine(FireContinously());
    }
    if (Input.GetButtonUp("Fire1"))
    {
        StopAllCoroutines();
    }
}
```

Pozn. autora: V tomto prípade je možné použiť metódu *StopAllCoroutines*(), pretože využívame len jednu korutinu. Pozor však v prípade využívania viacerých korutín. Mohlo by to spôsobiť nežiadúce správanie.

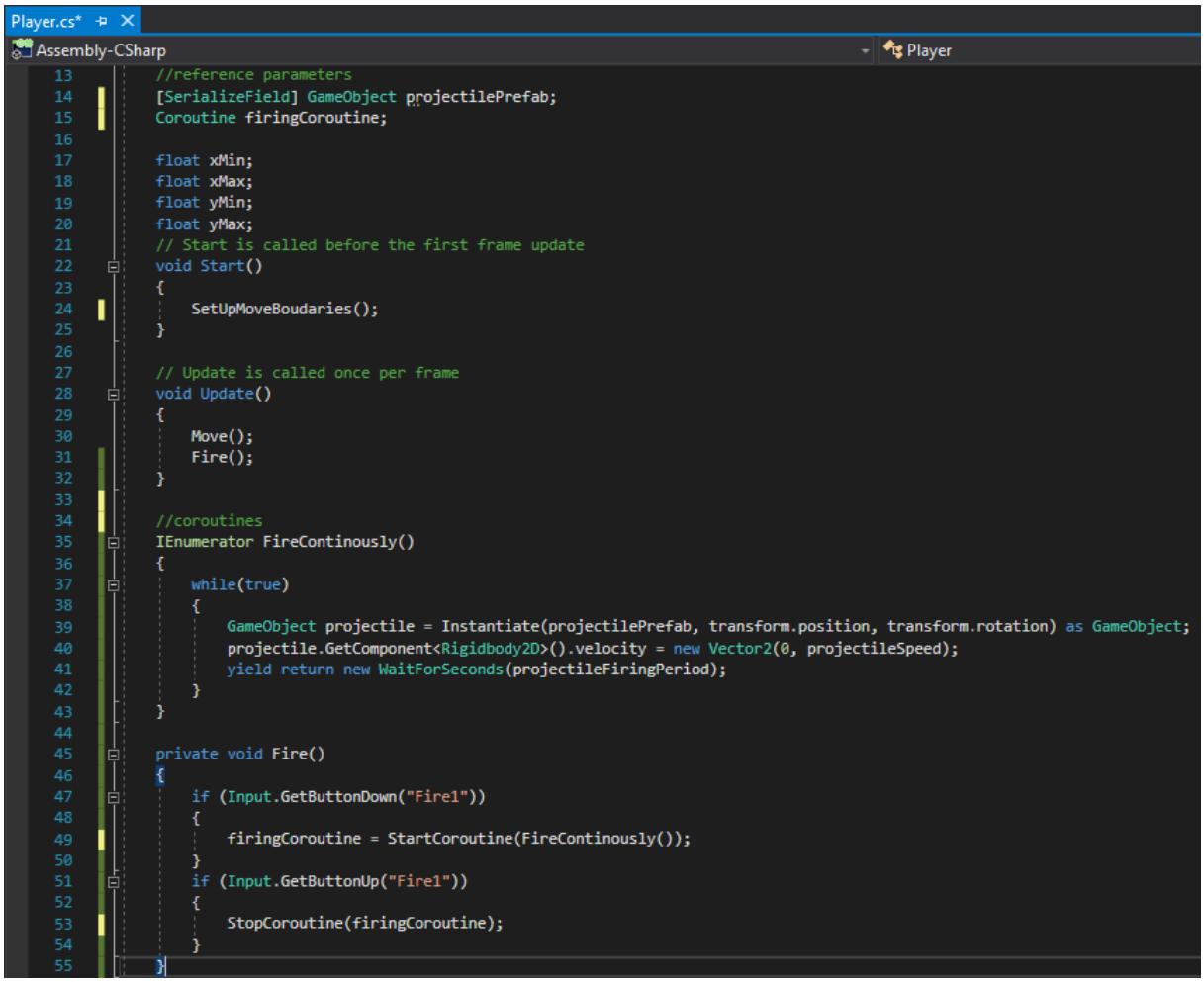
Vhodnejšie by bolo deklarovať premennú *firingCoroutine* typu *Coroutine*.

```
Coroutine firingCoroutine;
```

Potom treba metódu *Fire*() upraviť nasledovne.

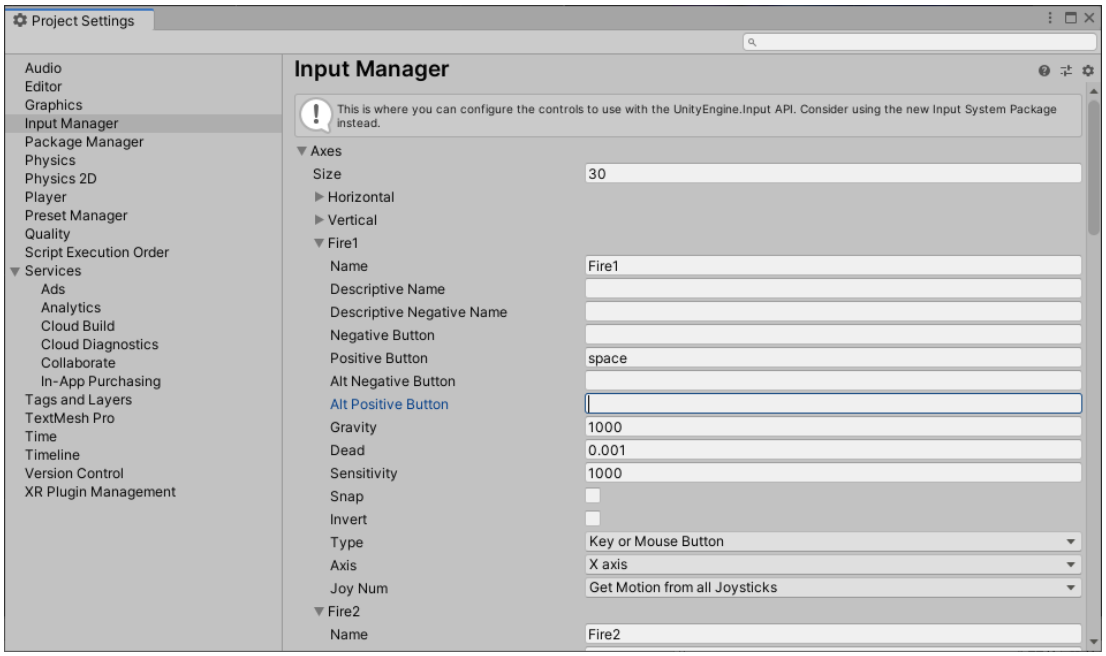
```
private void Fire()
{
    if (Input.GetButtonDown("Fire1"))
    {
        firingCoroutine = StartCoroutine(FireContinously());
    }
    if (Input.GetButtonUp("Fire1"))
    {
        StopCoroutine(firingCoroutine);
    }
}
```

Je zrejmé, že teraz dochádza k zastaveniu konkrétnej korutiny podľa jej názvu využitím metódy *StopCoroutine()*.



Obr. 43 Vizuál implementovanej funkcionality s využitím korutiny.

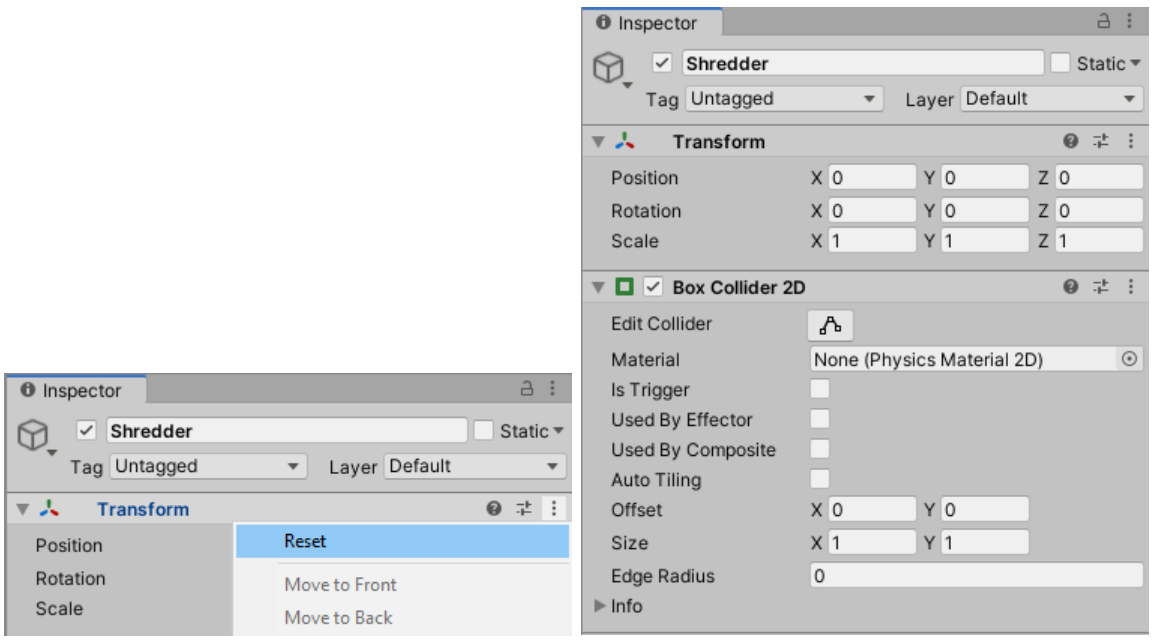
Pri testovaní funkcionality sa javí, že všetko funguje podľa požiadaviek. Avšak v prípade strieľania s využitím klávesnice a aj myši sa začnú strely generovať na rôznych vrstvách. Tento problém vyriešime nastavením len jedného vstupu pre streľbu a to klávesou medzery (*Space*).



Obr. 44 Upravené nastavenia pre *Fire1*.

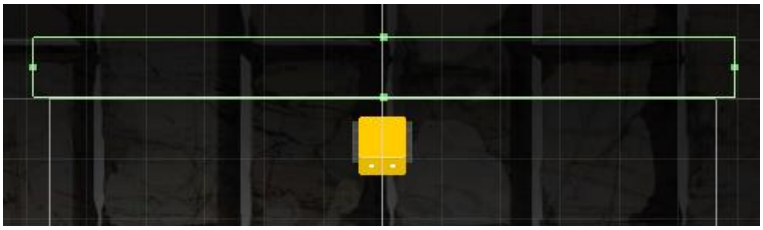
3.2 Zničenie objektu mimo herného priestoru

Cieľom je zabezpečiť zničenie striel v prípade ak sa dostanú mimo herný svet. Vytvoríme prázdny *Game Object*, ktorý pomenujeme *Shredder* resetneme mu nastavenia v komponente *Transform* pomocou možnosti *Reset*. Pridáme mu komponent *Box Colider 2D* (Obr. 45).



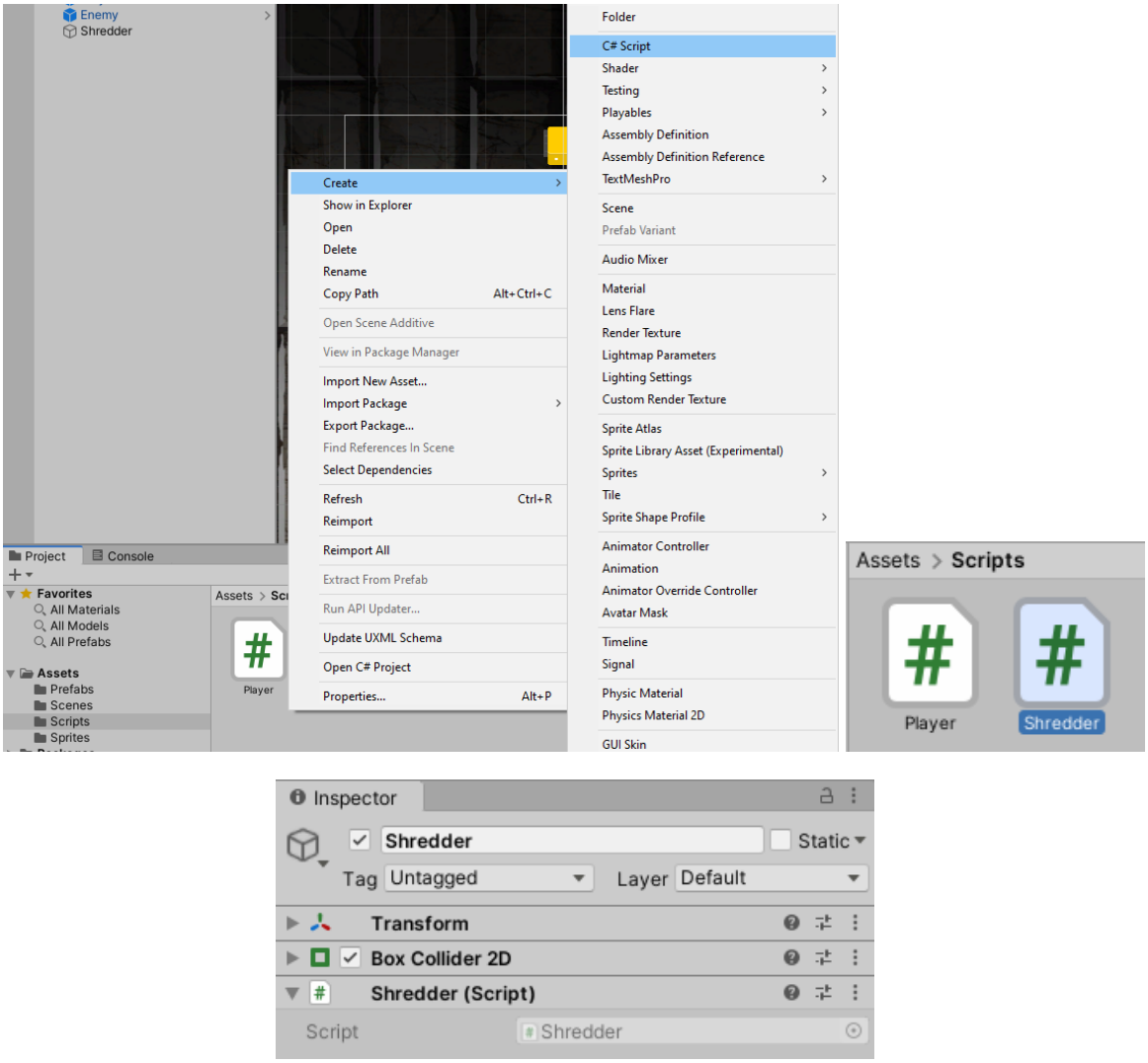
Obr. 45 Vytvorenie prázdneho *Game Objectu – Shredder*.

Objekt umiestnime do hornej časti obrazovky a pomocou *Edit Collider* mu zmeníme veľkosť *collideru* tak, aby prekrýval šírku herného sveta (Obr. 46).



Obr. 46 Vizuál a umiestnenie objektu *Shredder*.

Vytvoríme nový skript s názvom *Shredder* a pridáme ho objektu ako komponent (Obr. 47).



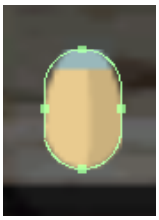
Obr. 47 Vytvorenie C# skriptu a jeho pridanie objektu *Shredder*.

V skripte sa postaráme o zničenie každého objektu, s ktorým nastane kolízia. Funkcionalitu riešime pomocou metódy *OnTriggerEnter2D()* triedy *MonoBehavior*, v ktorej zrealizujeme zničenie objektu zavolaním metódy *Destroy()* triedy *Object* (Obr. 48). Metóda *Destroy()* umožňuje zničiť objekt, komponent ale aj *asset*. Metóda má dva parametre. Prvý hovorí o tom, čo chceme zničiť a druhý parameter za koľko to chceme zničiť. V prípade zničenia objektu dôjde k jeho zničeniu aj so všetkými potomkami, ak ich objekt má. V našom prípade stačí ako prvý parameter použiť *collision.gameObject*, keďže chceme, aby bol objekt zničený okamžite.

```
Shredder.cs
Assembly-CSharp
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class Shredder : MonoBehaviour
6   {
7       private void OnTriggerEnter2D(Collider2D collision)
8       {
9           Destroy(collision.gameObject);
10      }
11  }
```

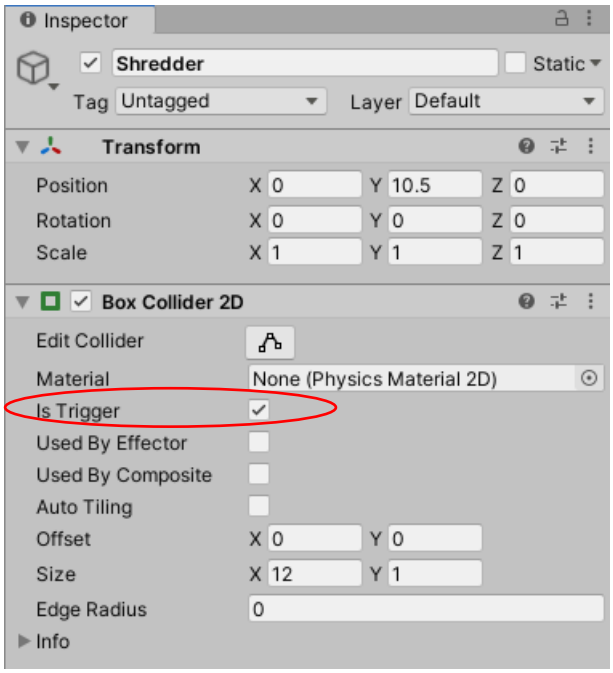
Obr. 48 Vizuál skriptu *Shredder.cs*.

Aby to ale fungovalo, musíme pridať *collider* aj našej strele. Pridáme *Capsule Collider 2D*. Ak chceme skontrolovať ako sa aplikoval tento *collider* na objekt, stačí ak na chvíľku umiestnime do scény jednu inštanciu, aby sme videli, či je nutné tvar *collideru* upraviť.



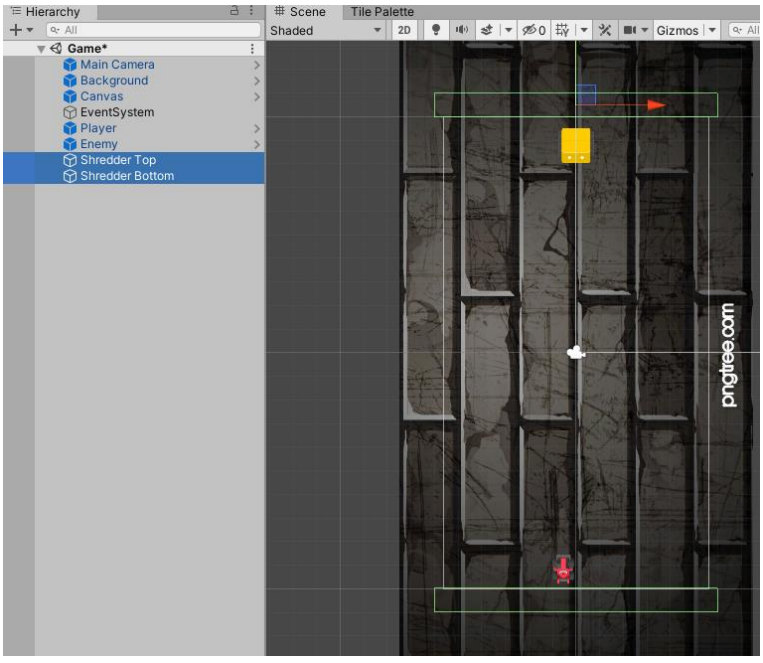
Obr. 49 *Capsule Collider 2D* objektu *PlayerProjectile*.

Pred samotným otestovaním funkcionality v editore Unity je nutné zaškrtnúť v komponente *Box Collider 2D* objektu *Shredder* vlastnosť *Is Trigger* (Obr. 50). Tým aktivujeme túto vlastnosť a pri kolízii bude reakcia taká, akú sme implementovali v skripte *Shredder.cs*.



Obr. 50 Vlastnosť *Is Trigger*.

Je vhodné mať takýto objekt aj v spodnej časti herného priestoru, ktorý bude v budúcnosti ničiť strely nepriateľov. Vytvoríme ho jednoduchým duplikovaním Ctrl + D, premenovaním na *Shredder Bottom* a správnym umiestnením v scéne, ako ilustruje obrázok 51. Zároveň sme upravili názov pôvodného objektu na *Shredder Top*.



Obr. 51 Vizuál objektov *ShredderTop* a *ShredderBottom*.

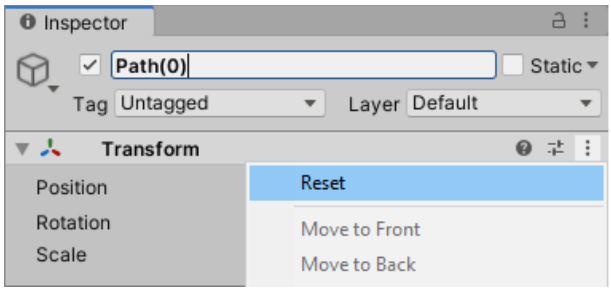
3.3 Kontrolné otázky a úlohy

1. Definujte proces vývoja hry a hlavné oblasti vývoja (*core*, *secondary* a *polish features*).
2. Čo vyjadruje termín *prefab*? Pre aké objekty je vhodné ho používať?
3. Pomocou akej metódy vieme dynamicky vytvoriť inštanciu konkrétneho objektu? Definujte túto metódu.
4. Ako vieme zabezpečiť funkcionalitu strieľania?
5. Definujte, aké možnosti nastavenia máme v prípade vlastnosti *Body Type* komponentu *Rigidbody 2D*.
6. Definujte, čo je korutina a v akom kontexte ju vieme pri vývoji hier využiť.
7. Definujte kľúčové slová, ktoré je nutné použiť v prípade definovania korutiny.
8. Aké metódy používame pre zastavenie činnosti korutiny?
9. Ako sa volajú metódy na obsluhu udalostí, ktoré sú volané v prípade kolízie dvoch objektov?
10. Čo vyjadruje termín *collider*? Aké typy *colliderov* poznáte?
11. Aká je funkcionalita vlastnosti *Is Trigger*?

4 Vytvorenie zhlukov pohybujúcich sa nepriateľov

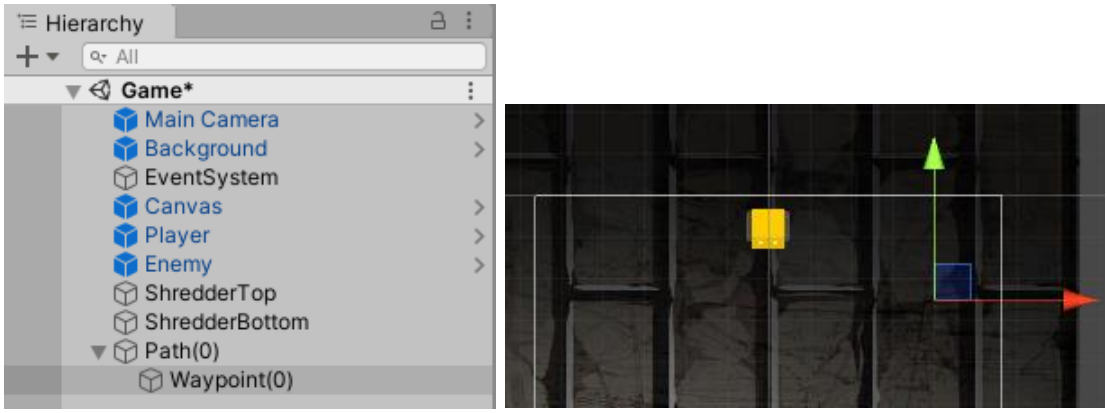
Cieľom tejto kapitoly je zabezpečiť vytvorenie zhlukov nepriateľov pohybujúcich sa po vopred definovanej ceste. Základnou myšlienkou zabezpečenia tejto funkcionality je vytvorenie zoznamu, tzv. *listu* rôznych trasových bodov (*waypoints*). Z týchto vytvoríme cestu, po ktorej sa budú nepriatelia pohybovať.

Ako prvé vytvoríme v okne *Hierarchy* prázdny *Game Object*, pomenujeme ho *Path(0)* a resetneme nastavenia v komponente *Transform* pomocou možnosti *Reset* (Obr. 52).



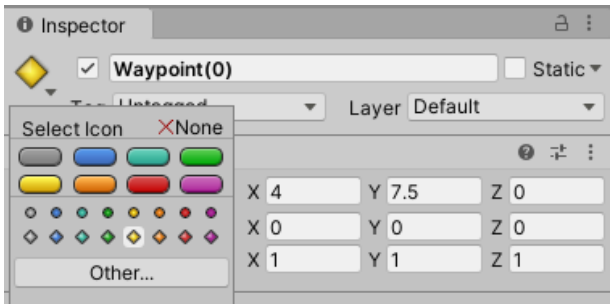
Obr. 52 Vytvorenie objektu *Path(0)* a resetnutie jeho pozície a rotácie.

Následne vytvoríme ďalší prázdny *Game Object* ako potomka objektu *Path(0)* a pomenujeme ho *Waypoint(0)*, ktorý v hernom priestore umiestnime napríklad vpravo hore (Obr. 53).



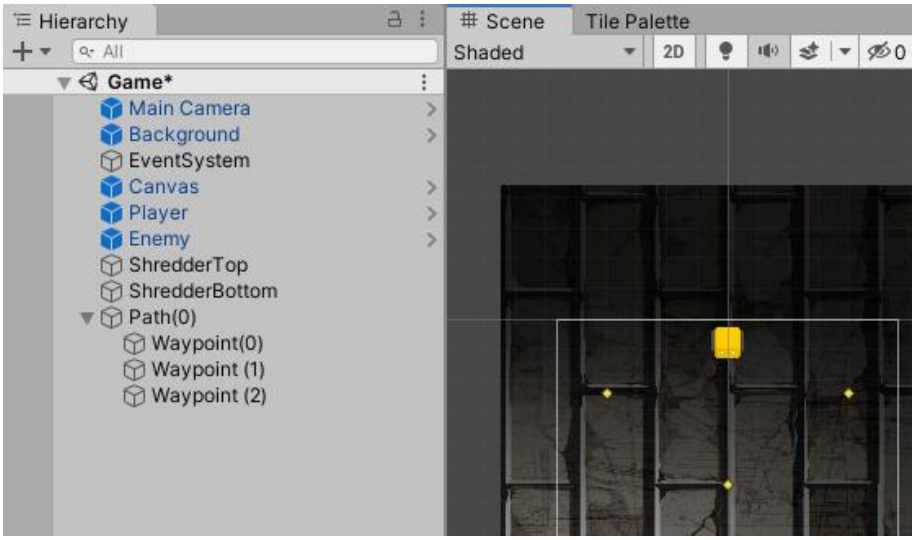
Obr. 53 Vytvorenie objektu *Waypoint(0)* a jeho umiestnenie.

Menším diskomfortom pri práci s prázdnymi *Game Objectmi* sa môže zdať ich neviditeľnosť v scéne. Jednoduchý trik, ktorým si vieme pomôcť, je zmena ikony objektu v okne *Inspector* (Obr. 54). My sme tu využili ikonu žltého kosoštvorca.



Obr. 54 Zmena ikony objektu *Waypoint(0)*.

Duplikovaním tohto objektu (Ctrl + D) vytvoríme ďalšie dva objekty a vytvoríme z nich jednoduchú cestu ako ilustruje obrázok 55.



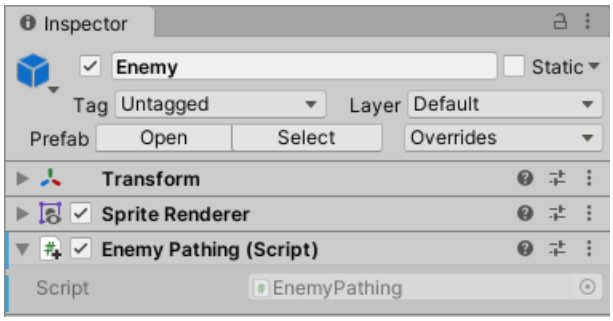
Obr. 55 Vytvorenie cesty.

Pole (*array*) v jazyku C# je statickou homogénnou dátovou štruktúrou, čo znamená že má pevnú veľkosť a umožňuje uchovávať viacero prvkov rovnakého dátového typu. Základný rozdiel medzi zoznamom (*listom*) a poľom je ten, že *list* je dynamická dátová štruktúra, ktorá môže meniť svoju veľkosť počas činnosti programu. Ide teda o flexibilnejšiu dátovú štruktúru. Nespornou výhodou využívania poľa je rýchlosť prístupu k jednotlivým prvkom, ku ktorým vieme pristupovať s konštantnou časovou zložitosťou, keďže sú indexované. Prvky poľa zvykneme indexovať od 0. V prípade *listu* musíme rátať v najlepšom prípade s konštantnou, v tom najhoršom prípade s lineárnou časovou zložitosťou. Je to závislé od spôsobu ako sú prvky v *liste* uložené a ako s nimi pracujeme.

Ukážka inicializácie poľa a *listu* v jazyku C#:

```
int[] evenNumbers = { 2, 4, 5, 8, 10 };
List<int> evenNumbers = { 2, 4, 5, 8, 10 };
```

Objektu *Enemy* pridáme nový komponent v podobe C# skriptu, ktorý pomenujeme *EnemyPathing.cs* (Obr. 56).

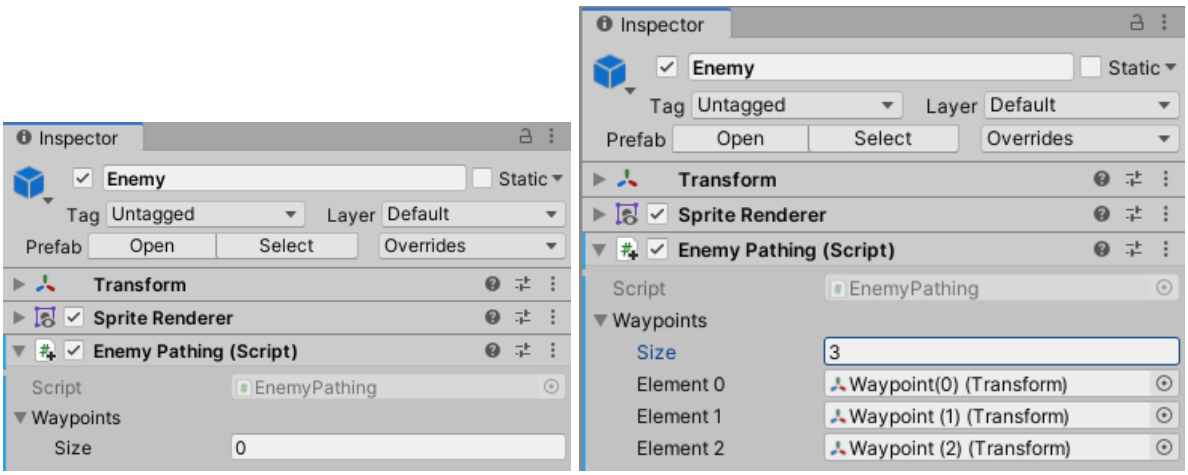


Obr. 56 Vytvorenie a pridanie C# skriptu objektu *Enemy*.

V skripte *EnemyPathning.cs* deklarujeme premennú *waypoints* ako *list* s typom *Transform*, pomocou ktorej budeme vedieť upravovať pozíciu jednotlivých bodov predstavujúcich cestu.

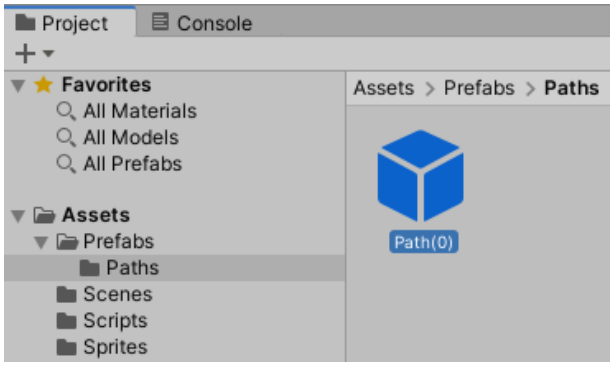
```
[SerializeField] List<Transform> waypoints;
```

Uložíme zmeny a vrátime sa do editora Unity, kde sa postaráme o inicializáciu prvkov tohto *listu*. *Size* určuje veľkosť *listu*. V našom prípade ju nastavíme na hodnotu 3 a jednotlivé prvky inicializujeme priamo objektmi *Waypoint*(0), *Waypoint*(1) a *Waypoint*(2) (Obr. 57).



Obr. 57 Inicializácia *listu waypoints*.

Z cesty, ktorú reprezentuje objekt *Path*(0) vytvoríme *prefab*, ktorý uložíme do priečinka *Paths* (Obr. 58).



Obr. 58 Vytvorenie *prefabu* z objektu *Path*(0).

4.1 Riešenie pohybu nepriateľa po definovanej ceste

Pohyb nepriateľa po definovanej ceste zabezpečíme pomocou metódy *MoveTowards()* štruktúry Vector2. Metóda má tri parametre, ktoré reprezentujú aktuálnu pozíciu, cieľovú pozíciu a rýchlosť pohybu objektu. Celý algoritmus pohybu si môžeme predstaviť ako cyklus, ktorý sa bude opakovať, kým nebude dosiahnutý posledný bod cesty. V cykle budeme zabezpečovať pohyb objektu z aktuálnej pozície do cieľovej pozície a v prípade úspechu sa postaráme o inkrementáciu premennej, pomocou ktorej budeme pristupovať k jednotlivým bodom cesty. V prípade, ak dosiahneme posledný bod cesty, postaráme sa o zničenie objektu nepriateľa.

V skripte *EnemyPathing.cs* definujeme premennú *waypointIndex*, ktorá bude určovať index jednotlivých bodov predstavujúcich cestu.

```
int waypointIndex = 0;
```

Premenná *moveSpeed* bude určovať rýchlosť pohybu nepriateľa. Premennú definujeme ako *SerializeField* s cieľom jej jednoduchej modifikácie priamo v editore Unity:

```
[SerializeField] float moveSpeed = 2f;
```

Vo funkcii *Start()* určíme počiatočnú pozíciu nepriateľa pomocou priradenia:

```
void Start()
{
    transform.position = waypoints[waypointIndex].transform.position;
}
```

kde ľavá časť priradenia hovorí o pozícii nepriateľa (skript je komponent objektu *Enemy*, preto nie je nutné špecifikovať akému objektu tieto vlastnosti patria) a pravá časť reprezentuje pozíciu prvého bodu definovanej cesty.

Vo funkcii *Update()* zavoláme metódu *Move()*, v ktorej riešime funkcionalitu predstaveným spôsobom. Pomocou vlastnosti *Count* použitého *listu* vieme zistiť jeho veľkosť (v našom prípade je táto hodnota 3). Tým dosiahneme opakovanie pohybu, kým sa nedostaneme na koniec cesty. Hodnotu dekrementujeme o 1 z dôvodu, že prvky *listu* sú tiež indexované od nuly.

```
private void Move()
{
    if (waypointIndex <= waypoints.Count - 1)
    {
        //what is target
        var targetPosition = waypoints[waypointIndex].transform.position;
        //how fast I want to move
        var movementThisFrame = moveSpeed * Time.deltaTime;
        transform.position = Vector2.MoveTowards
                (transform.position, targetPosition, movementThisFrame);

        if (transform.position == targetPosition)
        {
            waypointIndex++;
        }
    }
    else
    {
        Destroy(gameObject);
    }
}
```

Pozn. autora: Nečakajte v metóde *Move()* implementovaný žiadny cyklus. *Move()* je metóda, ktorá sa volá v *Update()*, pričom vieme, že táto funkcia je volaná každý jeden *frame*, čo v konečnom dôsledku zabezpečuje opakovateľnosť deja.

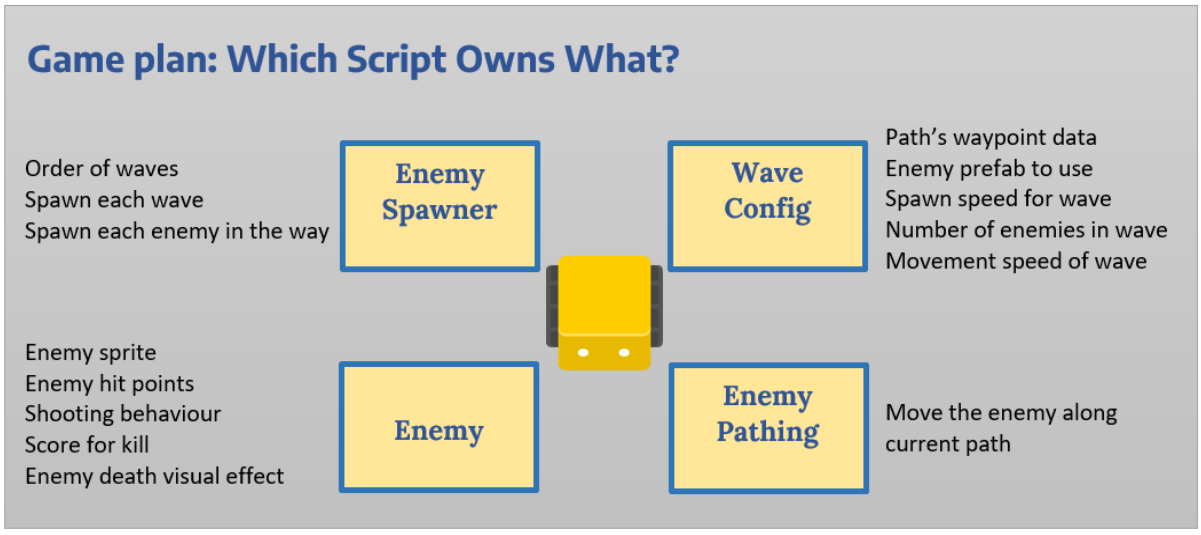
Po uložení zmien a návrate do editora Unity otestujeme implementovanú funkcionalitu. Objekt nepriateľa sa pohybuje od jedného bodu k druhému, až dôjde k jeho zničeniu. S pozíciou bodov cesty, ako aj zo zmenou rýchlosti pohybu nepriateľa, môžeme experimentovať podľa svojich preferencií.

4.2 Použitie *ScriptableObject*

ScriptableObject je dátové úložisko, ktoré môžeme použiť pre uloženie väčšieho množstva dát nezávisle od inštancie triedy. Jedným z typických použití je snaha minimalizovať priestorové nároky, ktoré vznikajú vytváraním kópií hodnôt pre jednotlivé inštancie triedy. Užitočné je to v prípade ak máme *prefab*, ktorý uchováva väčšie množstvo nemenných údajov (Unity, 2021c).

My *scriptableObject* využijeme práve v kontexte s vytváraním ciest (*waves*), po ktorých sa budú pohybovať nepriatelia. Je prirodzené evidovať pre každú cestu informácie o type nepriateľa, ktorý sa bude pohybovať po danej ceste spolu s uvedením jeho rýchlosti pohybu. Tiež má zmysel evidovať počet inštancií nepriateľov, ktorý sa na ceste vytvoria, spolu s informáciou ako často majú tieto inštancie vznikať.

Zabezpečenie predstaveného herného mechanizmu vyžaduje vytvorenie ďalších skriptov tak ako ilustruje obrázok 59.



Obr. 59 Zodpovednosť jednotlivých skriptov.

V skripte *Enemy.cs* budeme riešiť:

- vizuálnu podobu nepriateľa (*Enemy sprite*),
- koľko krát treba nepriateľa zasiahnuť aby sme ho zničili (*Enemy hit points*),

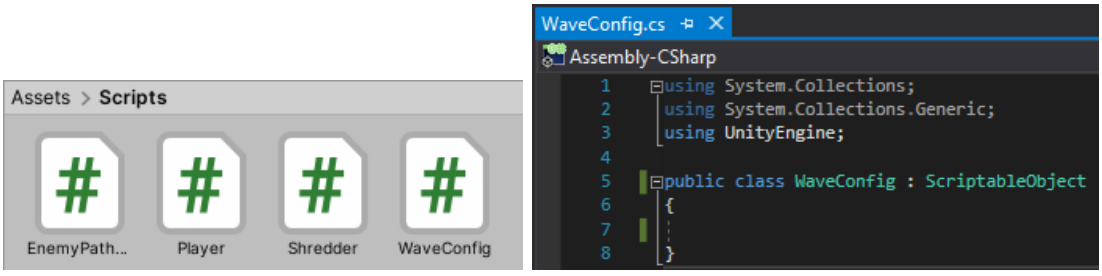
- akou rýchlosťou, akou silou, respektíve ako často strieľa nepriateľ (*Shooting behaviour*),
- aké skóre sa pripíše hráčovi za zničenie nepriateľa (*Score for kill*),
- aká animácia, prípadne aký zvukový záznam sa prehrá v prípade smrti nepriateľa (*Enemy death visual effect*).

Skript *WaveConfig* bude vytvorený ako *ScriptableObject*. V ňom budeme udržiavať:

- informácie o jednotlivých bodoch, ktoré tvoria cestu (*Path's waypoint data*),
- ktorý *prefab* reprezentuje nepriateľa (*Enemy prefab to use*),
- v akých časových rozostupoch majú vznikať jednotlivé inštancie nepriateľov (*Spawn speed for wave*),
- aký je cieľový počet nepriateľov (*Number of enemies in wave*),
- akou rýchlosťou sa budú pohybovať nepriatelia po ceste (*Movement speed of wave*).

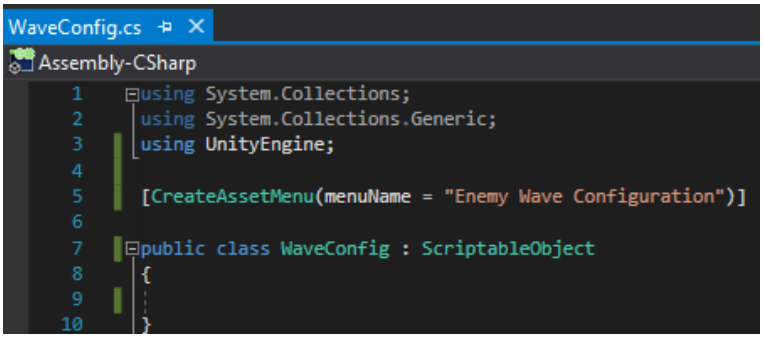
Skript *EnemyPathing.cs* sme vytvorili v predchádzajúcej kapitole. V ňom zabezpečujeme pohyb konkrétnej inštancie nepriateľa po vopred definovanej ceste. Po akej ceste sa má nepriateľ pohybovať budeme vedieť podľa skriptu *EnemySpawner.cs*, ktorý bude zodpovedný za vznik cesty a nepriateľov na nej. V neposlednom rade tu bude riešená logika poradia pohybujúcich sa nepriateľov po jednotlivých cestách. Je predpoklad, že ciest a rôznych nepriateľov budeme mať v hre viac. Informácie bude tento skript získavať z *WaveConfig*, pričom zo skriptu *EnemySpawner.cs* budeme posielať informácie do skriptu *EnemyPathing.cs*.

Ako prvé vytvoríme skript *WaveConfig.cs*, ktorému zmeníme typ z *MonoBehaviour* na *ScriptableObject* (Obr. 60).

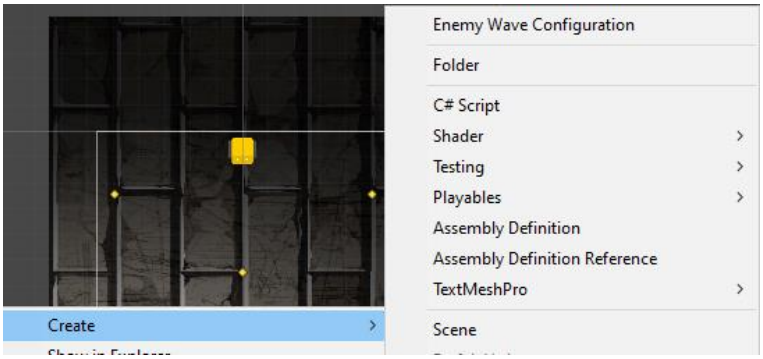


Obr. 60 Vytvorenie skriptu *WaveConfig* ako *ScriptableObject*.

Pomocou atribútu *CreateAssetMenu* triedy *UnityEngine* (Obr. 61) rozšírime ponuku *Assets/Create submenu*, aby sme jednoducho vedeli vytvoriť *ScriptableObject*, uchovávaný v projekte ako ďalší *asset* (Obr. 62).

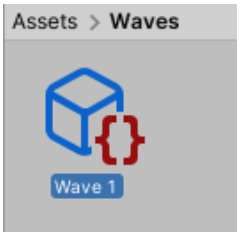


Obr. 61 Rozšírenie ponuky *Create*.



Obr. 62 Rozšírená ponuka *Create*.

V projekte vytvoríme nový priečinok *Waves*, do ktorého pomocou tejto rozšírenej ponuky vytvoríme *ScriptableObject Wave 1*.



Obr. 63 Vytvorenie *ScriptableObject Wave 1*.

V skripte *WaveConfig.cs* vytvoríme nasledovné premenné:

```
[SerializeField] GameObject enemyPrefab;
[SerializeField] GameObject pathPrefab;
[SerializeField] float timeBetweenSpawns = 0.5f;
[SerializeField] float spawnRandomFactor = 0.3f; //for random behavior
[SerializeField] int numberOfEnemies = 5;
[SerializeField] float moveSpeed = 2f;
```

Aby sme umožnili ostatným skriptom získavať tieto informácie, vytvoríme si verejné prístupové metódy, tzv. *gettery* pre každú premennú podľa prototypu:

```
public GameObject GetEnemyPrefab() { return enemyPrefab; }
```

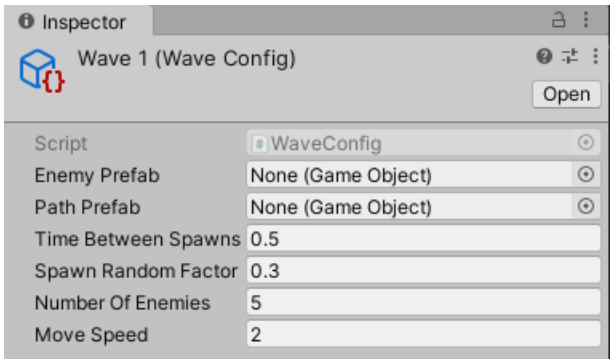
WaveConfig.cs

Assembly-CSharp

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  [CreateAssetMenu(menuName = "Enemy Wave Configuration")]
6
7  public class WaveConfig : ScriptableObject
8  {
9      [SerializeField] GameObject enemyPrefab;
10     [SerializeField] GameObject pathPrefab;
11     [SerializeField] float timeBetweenSpawns = 0.5f;
12     [SerializeField] float spawnRandomFactor = 0.3f; //for random behavior
13     [SerializeField] int numberOfEnemies = 5;
14     [SerializeField] float moveSpeed = 2f;
15
16     //getters
17     public GameObject GetEnemyPrefab() { return enemyPrefab; }
18     public GameObject GetPathPrefab() { return pathPrefab; }
19     public float GetTimeBetweenSpawns() { return timeBetweenSpawns; }
20     public float GetSpawnRandomFactor() { return spawnRandomFactor; }
21     public int GetNumberOfEnemies() { return numberOfEnemies; }
22     public float GetMoveSpeed() { return moveSpeed; }
23 }
```

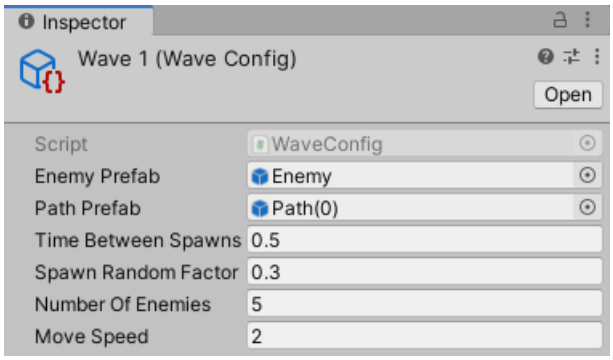
Obr. 64 Vizuál skriptu *WaveConfig.cs*.

Po uložení zmien a návrate do editora Unity, pri zobrazení vlastností objektu *Wave 1* v okne *Inspector*, vidíme všetky dáta, ktoré sme pridali.



Obr. 65 Vlastnosti objektu *Wave 1*.

Následne inicializujeme premenné *enemyPrefab* a *pathPrefab* objektmi, ktoré máme zatiaľ vytvorené (Obr. 66).



Obr. 66 Inicializácia premenných *enemyPrefab* a *pathPrefab*.

4.2.1 Použitie cyklu *foreach* pre získanie zoznamu bodov danej cesty

V predchádzajúcej kapitole sme v skripte *WaveConfig.cs* definovali *getter GetPathPrefab()*, ktorý je schopný vrátiť cestu ako jeden objekt. My však vieme, že cesta pozostáva z jednotlivých bodov, preto túto metódu upravíme tak, aby vracala zoznam týchto bodov, k čomu použijeme opätovne dátovú štruktúru *list*. Existujúcu metódu nahradíme touto:

```
public List<Transform> GetWaypoints()
{
    var waveWaypoints = new List<Transform>();
    foreach (Transform child in pathPrefab.transform)
    {
        waveWaypoints.Add(child);
    }
    return waveWaypoints;
}
```

Zmeníme jej prototyp tak, že návratovou hodnotou bude *list* jednotlivých bodov. Tento *list* si pomocou *foreach* cyklu naplníme všetkými bodmi, z ktorých daná cesta pozostáva. Označenie *child* sme použili preto, lebo jednotlivé *Waypoint* sú potomkami objektu *Path*. Tým sme dosiahli, že máme prístup ku všetkým bodom danej cesty jednotlivo, a nielen k celému objektu.

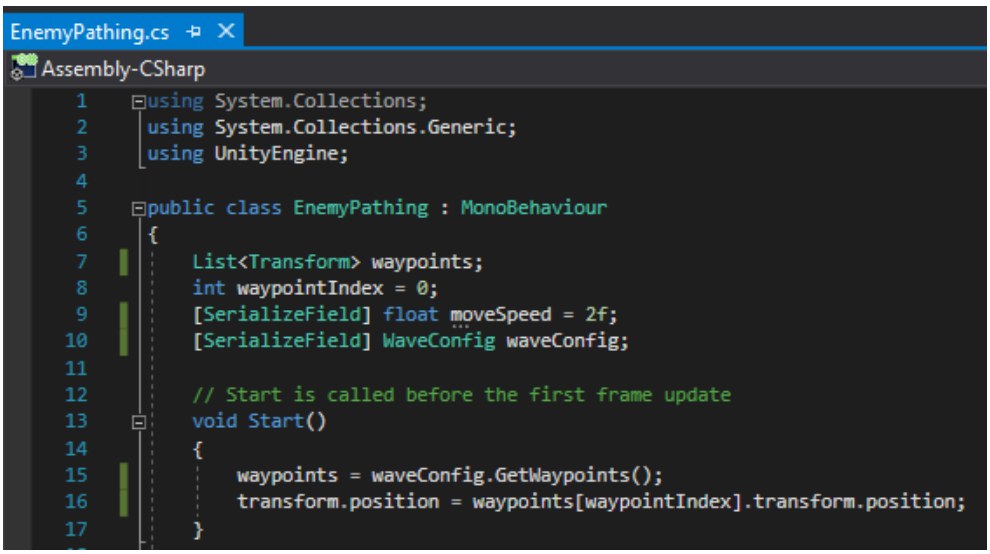
Ďalšie úpravy pred otestovaním funkčnosti v editore Unity robíme v skripte *EnemyPathing.cs*. Deklarujeme premennú *waveConfig*, ktorá bude reprezentovať náš *ScriptableObject WaveConfig*.

```
[SerializeField] WaveConfig waveConfig;
```

Ako vieme, v tomto skripte máme definovaný *list* našich bodov, ktorý stále potrebujeme. Nepotrebujeme však, aby bol *SerializeField*, z dôvodu že informácie o bodoch danej cesty máme k dispozícii a vieme ich ľubovoľne modifikovať vďaka vytvorenému *ScriptableObject*. Z tohto dôvodu upravujeme deklaráciu premennej *waypoints* nasledovne:

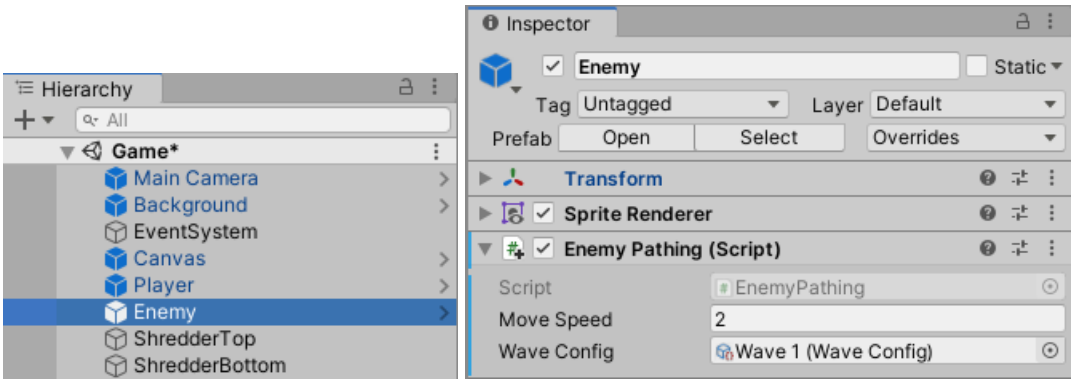
```
List<Transform> waypoints;
```

Vo funkcii *Start*() pred inicializáciou aktuálnej pozície nepriateľa doplníme zavolanie *gettera GetWaypoints*(), čím zabezpečíme inicializáciu premennej *waypoints* (Obr. 67).



Obr. 67 Vizuál fragmentu skriptu *EnemyPathing.cs*.

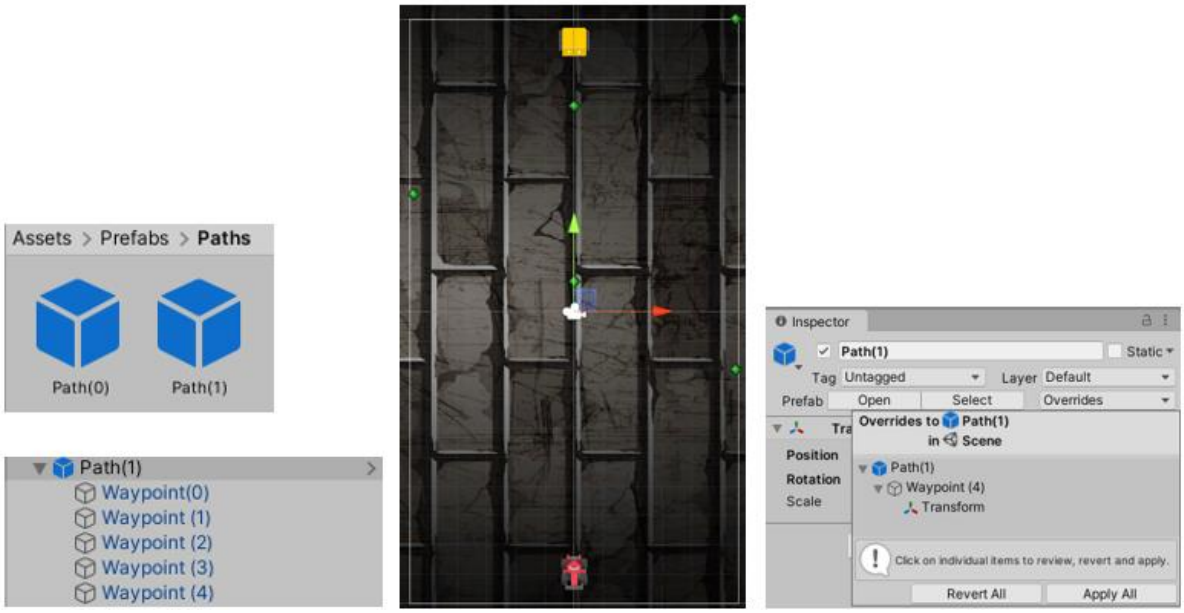
Po návrate do editora Unity môžeme v okne *Hierarchy* vymazať objekt *Path*(0). Nebudeme ho už potrebovať, keďže sme zabezpečili funkcionalitu, ktorou vieme ku každej konkrétnej inštancii nepriateľa priradiť konkrétnu cestu po ktorej sa bude pohybovať. Pred testovaním funkčnosti je však nutné inicializovať premennú *waveConfig* vytvorením *ScriptableObjectom Wave 1* (Obr. 68).



Obr. 68 Vlastnosti objektu *Enemy*.

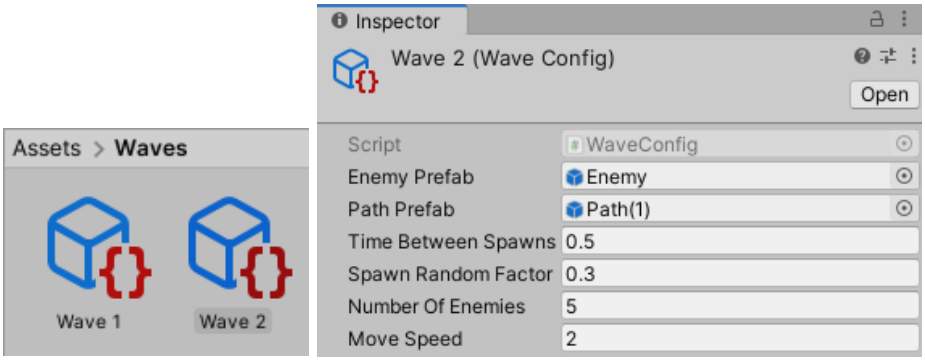
Pre upevnenie vedomostí implementovanej funkcionality, ako aj zabezpečenie rôznorodosti prvkov hry, vytvoríme novú cestu a zabezpečíme, aby sa nepriateľ pohyboval po nej. Odporúčame študujúcemu, aby implementoval samostatne a následne si svoj postup skontroloval podľa časti uvedenej nižšie.

Najjednoduchší spôsob, ako si vytvoriť novú cestu, je duplikovaním *prefabu* objektu *Path*(0). Novú cestu sme vyskladali z piatich bodov, ktoré sme vhodne rozmiestnili v hernom priestore. Pre ich rôznu vizuálnu reprezentáciu od cesty *Path*(0) sme použili zelené diamanty. Po zrealizovaní úprav nezabúdame zmeny aplikovať na *prefab* pomocou možnosti *Apply All* (Obr. 69).



Obr. 69 Vytvorenie *prefabu Path*(1).

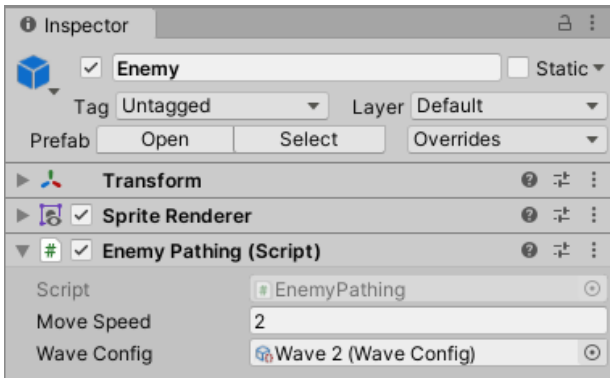
Duplikovaním si tiež môžeme vytvoriť nový *ScriptableObject* *Wave* 2, kde premennú *pathPrefab* inicializujeme novo vytvoreným *prefabom* *Path*(1) (Obr. 70).



Obr. 70 Vytvorenie objektu *Wave* 2 a úprava jeho nastavení.

Pozn. autora: Ostatné hodnoty premenných je zatiaľ zbytočné upravovať, pretože ich funkcionalitu sme ešte neimplementovali.

Poslednou úlohou je priradiť túto novú cestu nepriateľovi tak, že inicializujeme v editore Unity premennú *waveConfig*. Momentálne nie je podstatné, či zmeny vykonávate na konkrétnej inštancii objektu *Enemy* alebo na *prefabe*.



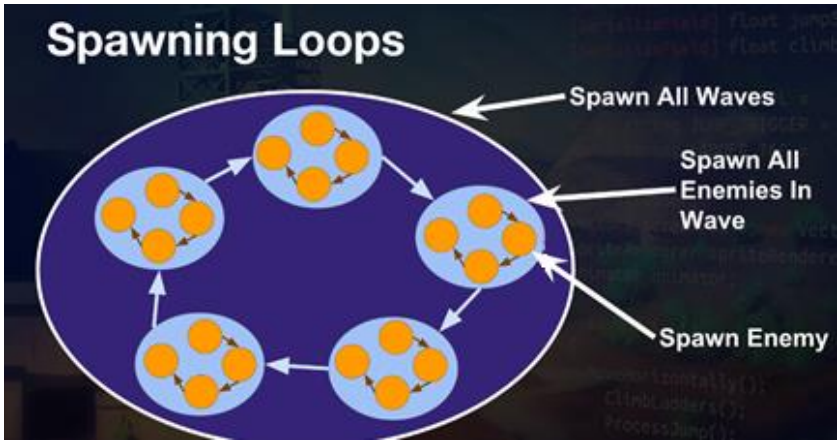
Obr. 71 Inicializácia premennej *waveConfig* objektu *Enemy*.

Po otestovaní funkčnosti vidíme, že sa nepriateľ pohybuje po novo vytvorenej ceste. Nespornou výhodou tejto implementácie je, že pracujeme s cestou ako s jedným celkom, pričom dáta sú sústredené na jednom mieste.

4.3 Vytvorenie viacerých nepriateľov na jednej ceste

V tejto kapitole budeme riešiť tvorbu viacerých nepriateľov (*spawn multiple enemies*) pohybujúcich sa po priradenej ceste. Využijeme k tomu vnorené korutiny (*nested coroutine*). Skôr ako prejdeme k samotnej implementácii predstavíme koncept tejto funkcionality:

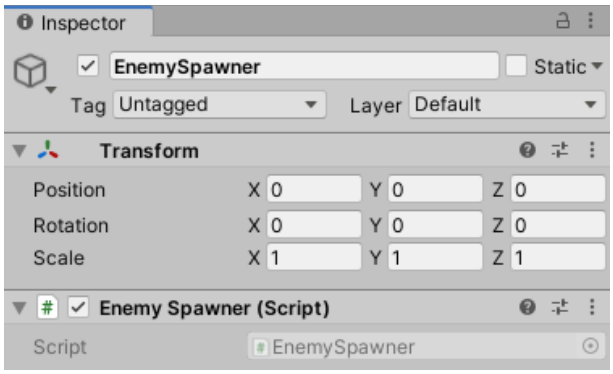
1. Postaráme sa o vytvorenie jednej inštancie nepriateľa (*spawn enemy*).
2. Následne sa postaráme o vytvorenie požadovaného počtu nepriateľov (určeného premennou *numberOfEnemies*) na jednej ceste s určeným časovým rozostupom vzniku (premenná *timeBetweenSpawns*). Všetky tieto dáta máme prístupné vďaka vytvorenému *ScriptableObject WaveConfig*.
3. V momente, ako nastane vytvorenie posledného nepriateľa na danej ceste, postaráme sa o vytvorenie novej cesty. Tento proces opakujeme, aby sme zabezpečili vznik požadovaného zhluku pohybujúcich sa nepriateľov.



Obr. 72 Vytvorenie zhlukov nepriateľov (Davidson, Rick and Ben Tristen. 2019).

Na vytvorenie tejto funkcionality je vhodné použiť korutiny. Potrebujeme čakať kým sa vytvoria všetci nepriatelia na danej ceste a až následne budeme realizovať vytvorenie ďalšej cesty.

Ako prvé vytvoríme v editore Unity v okne *Hierarchy* prázdny *Game Object* s názvom *EnemySpawner*, resetneme jeho nastavenia v komponente *Transform* a pridáme mu skript s rovnakým názvom (Obr. 73).

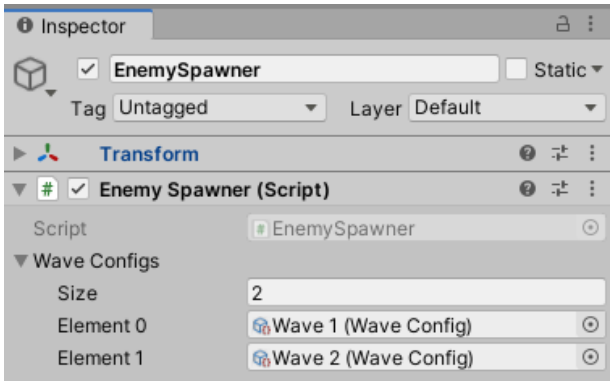


Obr. 73 Vytvorenie objektu *EnemySpawner*.

V skripte *EnemySpawner.cs* deklarujeme premennú *waveConfigs*, v ktorej budeme uchovávať dáta, aby sme vedeli, aké cesty s akými nepriateľmi chceme vytvoriť. Z tohto dôvodu je premenná typu *list*.

```
[SerializeField] List<WaveConfig> waveConfigs;
```

V projekte máme zatiaľ vytvorené len dva objekty typu *WaveConfig*, preto inicializujeme túto premennú v editore Unity týmito objektmi. V budúcnosti môžeme tento zoznam ľubovoľne rozšíriť.



Obr. 74 Inicializácia premennej *waveConfig*.

V skripte pokračujeme definíciou premennej *startingWave*, ktorá nám bude slúžiť na špecifikáciu toho, ktorý objekt typu *WaveConfig* budeme chcieť použiť. Inicializujeme ju na hodnotu 0.

```
int startingWave = 0;
```

Vo funkcii *Start*() definujeme premennú *currentWave*, ktorá bude slúžiť na identifikáciu aktuálneho objektu.

```
var currentWave = waveConfigs[startingWave];
```

Vo funkcii sa tiež postaráme o volanie korutiny *SpawnAllEnemiesInWave*() s parametrom reprezentujúcim aktuálny objekt.

```
StartCoroutine(SpawnAllEnemiesInWave(currentWave));
```

Definícia korutiny:

```
private IEnumerator SpawnAllEnemiesInWave(WaveConfig waveConfig)
{
    Instantiate(waveConfig.GetEnemyPrefab(),
          waveConfig.GetWaypoints()[0].transform.position, Quaternion.identity);
    yield return new WaitForSeconds(waveConfig.GetTimeBetweenSpawns());
}
```

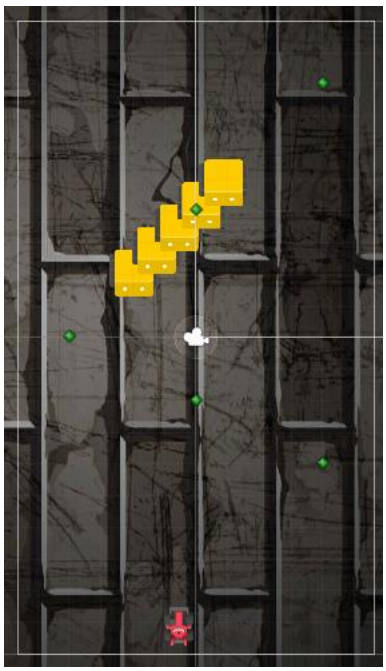
Pomocou parametra korutiny, ktorý je typu *WaveConfig* a predstavuje náš *ScriptableObject*, vieme pristupovať ku všetkým dátam objektu pomocou *getterov*, ktoré sme v ňom definovali. V korutine sa postaráme o vytvorenie inštancie nepriateľa pomocou metódy *Instantiate()* triedy *Object*. Prvý parameter reprezentuje objekt, ktorý chceme vytvoriť. Využijeme na to *getter GetEnemyPrefab*(). Druhý parameter určuje pozíciu, kde treba inštanciu objektu vytvoriť. Našim cieľom je vytvoriť ju na prvom bode požadovanej cesty. Pre získanie zoznamu bodov cesty využijeme *getter GetWaypoints*(). K prvému bodu pristúpime cez index 0. Rotáciu objektu nechávame bez zmeny, čo definujeme vlastnosťou *identity* štruktúry *Quaternion*. V príkaze *yield return* pozastavíme vykonanie do doby, ktorú určuje premenná *timeBetweenSpawns*. Hodnotu premennej opätovne získame pomocou definovaného *gettera*.

Uložíme zmeny a v editore Unity môžeme otestovať túto čiastočnú implementáciu. V okne *Hierarchy* môžeme vymazať objekt *Enemy*, pretože inštancie jednotlivých nepriateľov vytvárame už dynamicky pomocou kódu. Zatiaľ nemôžeme očakávať, že sa budú vytvárať viacerí nepriatelia na ceste, pretože to sme zatiaľ neimplementovali.

Aby sme vedeli vytvoriť viacerých nepriateľov, je nutné do korutiny doplniť cyklus, ktorý sa bude opakovať podľa hodnoty premennej *numberOfEnemies*. Hodnotu získame pomocou *gettera GetNumberOfEnemies*() a túto uložíme do pomocnej premennej *tmp* (*temporary*).

```
private IEnumerator SpawnAllEnemiesInWave(WaveConfig waveConfig)
{
    int tmp = waveConfig.GetNumberOfEnemies();
    for (int enemyCount = 0; enemyCount < tmp ; enemyCount++)
    {
        Instantiate(waveConfig.GetEnemyPrefab(),
            waveConfig.GetWaypoints()[0].transform.position, Quaternion.identity);
        yield return new WaitForSeconds(waveConfig.GetTimeBetweenSpawns());
    }
}
```

Zmeny uložíme a otestujeme funkčnosť v editore Unity (Obr. 75).



Obr. 75 Vizuál pohybujúcich sa nepriateľov po definovanej ceste.

Pozn. autora: Ak vám uvedené nastavenia premenných nevyhovujú môžete ich ľubovoľne pomeniť.

4.4 Riešenie rôznorodosti zhlukov nepriateľov

Cieľom je vyriešiť možnosť pohybovať sa s rôznou rýchlosťou, prípadne po rôznych cestách, aby útok nepriateľov nebol prvoplánový.

Premennú *waveConfig*, deklarovanú v skripte *EnemyPathing.cs*, už nebudeme potrebovať upravovať v editore Unity, lebo celú funkcionalitu riešime pomocou kódu. Preto nepotrebujeme aby bola premenná definovaná ako *SerializeField*. Premennú *moveSpeed* už tiež nepotrebujeme, keďže tieto dáta uchovávame v *ScriptableObject WaveConfig*. Namiesto toho, pre získanie tejto hodnoty na riadku 30 obrázku 76, využijeme *getter GetMoveSpeed()*.

EnemyPathing.cs

Assembly-CSharp — EnemyPathing

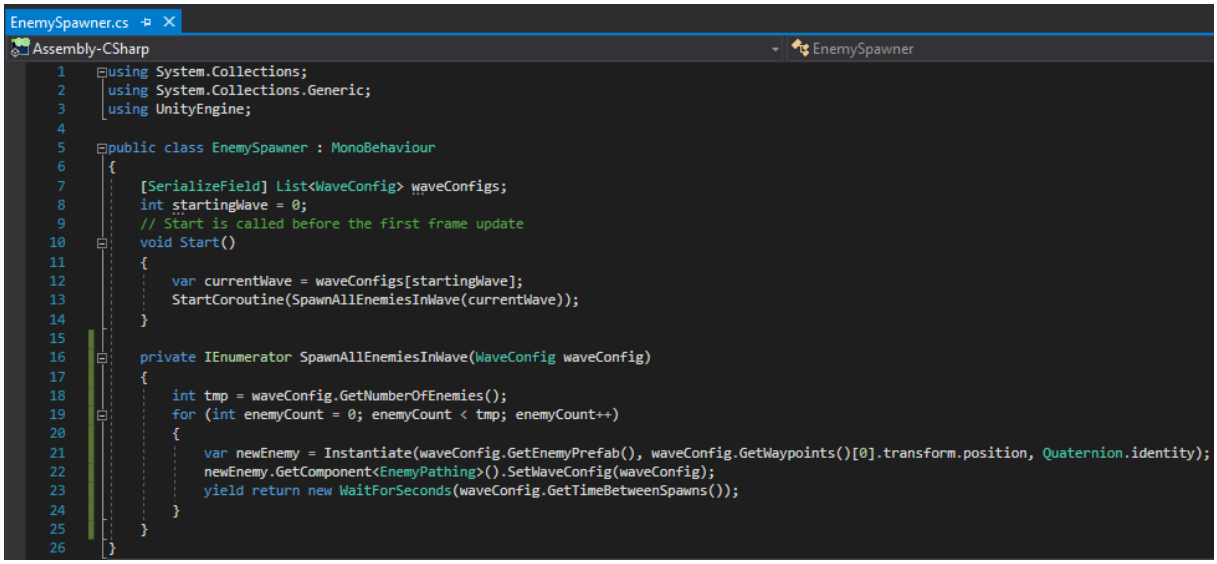
```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class EnemyPathing : MonoBehaviour
6  {
7      List<Transform> waypoints;
8      int waypointIndex = 0;
9      WaveConfig waveConfig;
10
11     // Start is called before the first frame update
12     void Start()
13     {
14         waypoints = waveConfig.GetWaypoints();
15         transform.position = waypoints[waypointIndex].transform.position;
16     }
17     // Update is called once per frame
18     void Update()
19     {
20         Move();
21     }
22
23     private void Move()
24     {
25         if (waypointIndex <= waypoints.Count - 1)
26         {
27             //what is target
28             var targetPosition = waypoints[waypointIndex].transform.position;
29             //how fast I want to move
30             var movementThisFrame = waveConfig.GetMoveSpeed() * Time.deltaTime;
31             transform.position = Vector2.MoveTowards(transform.position, targetPosition, movementThisFrame);
32
33             if (transform.position == targetPosition)
34             {
35                 waypointIndex++;
36             }
37         }
38         else
39         {
40             Destroy(gameObject);
41         }
42     }
43 }
```

Obr. 76 Vizuál skriptu *EnemyPathing.cs*.

V skripte zadefinujeme *setter SetWaveConfig()*.

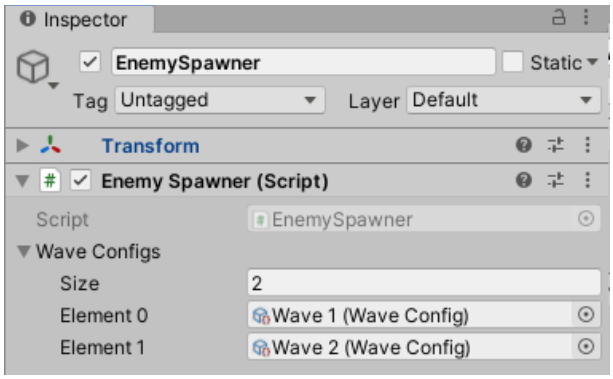
```
public void SetWaveConfig(WaveConfig waveConfig)
{
    this.waveConfig = waveConfig;
}
```

Takto definovaný *setter* využijeme v skripte *EnemySpawner.cs*, kde vytvoríme novú premennú *newEnemy*, ako výsledok procesu inštanciácie. Pomocou metódy *GetComponent()* triedy *GameObject* (pričom v < > špecifikujeme ku ktorému komponentu chceme pristúpiť, v tomto prípade k skriptu *EnemyPathing.cs*) zavoláme *setter SetWaveConfig()*, čím zabezpečujeme prenos dát (Obr. 77).



Obr. 77 Vizuál skriptu *EnemySpawner.cs*.

Po týchto úpravách je očakávané správanie také, že zhluk nepriateľov sa vytvorí na ceste určenej objektom *Wave 1*, pretože je to prvý prvok *listu waveConfings* objektu *EnemySpawner* (Obr. 78).



Obr. 78 Vlastnosti objektu *EnemySpawner*.

Aby zhluky nepriateľov vznikali na rôznych cestách, budeme potrebovať ďalšiu korutinu. V skripte *EnemySpawner.cs* vytvoríme korutinu *SpawnAllWaves()*, kde príkaz *yield return* bude viazaný na ukončenie procesu korutiny *SpawnAllEnemiesInWave()*, ktorú sme už definovali.

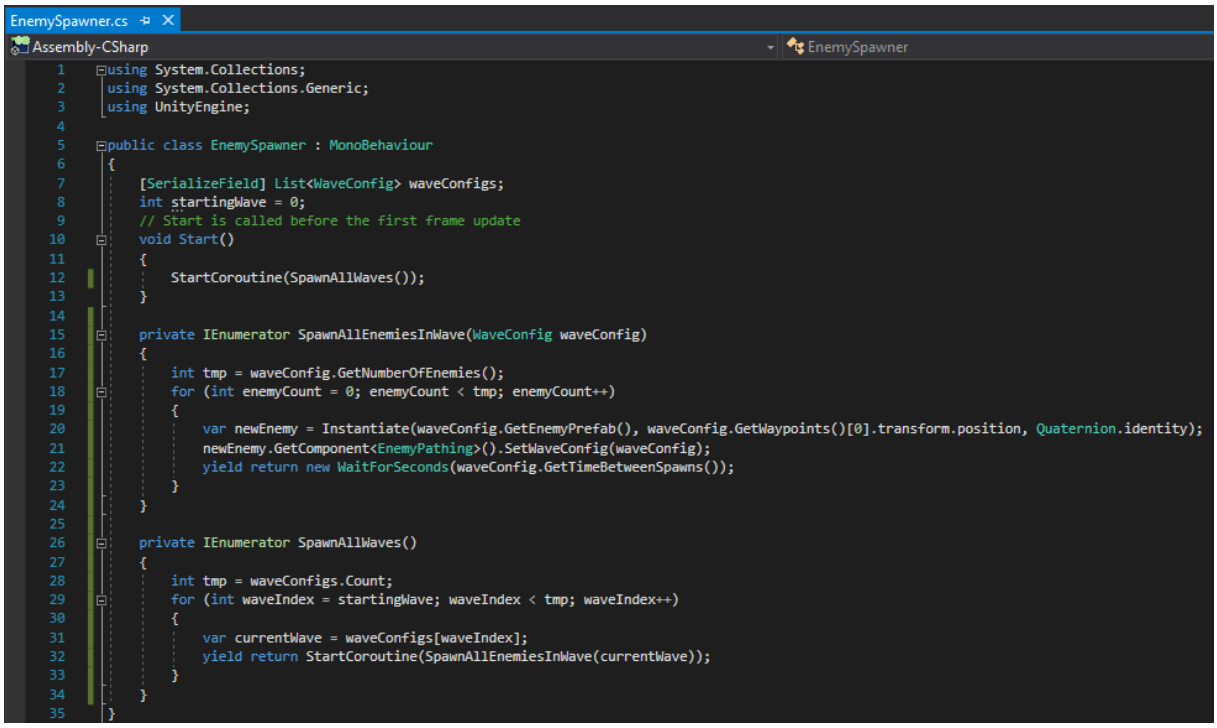
Definícia korutiny:

```
private IEnumerator SpawnAllWaves()
{
    int tmp = waveConfigs.Count;
    for (int waveIndex = startingWave; waveIndex < tmp; waveIndex++)
    {
        var currentWave = waveConfigs[waveIndex];
        yield return StartCoroutine(SpawnAllEnemiesInWave(currentWave));
    }
}
```

Do pomocnej premennej *tmp* uložíme počet prvkov *listu waveConfings*. Pomocou cyklu *for* zabezpečíme prechod cez všetky prvky, ktoré sa nachádzajú v *liste*. Prvý príkaz v tele cyklu definuje premennú *currentWave*, ktorá identifikuje aktuálny objekt typu *WaveConfig*. Premenná *waveIndex* reprezentuje riadiacu premennú cyklu, pomocou ktorej zabezpečíme postupne vznik nepriateľov na všetkých definovaných objektoch. V príkaze *yield return* uvádzame spustenie korutiny *SpawnAllEnemiesInWave()*, ktorá sa stará o vytvorenie nepriateľov na danej

ceste. Posledné čo treba v tomto skripte upraviť, je volanie správnej korutiny vo funkcii *Start()*:

```
StartCoroutine(SpawnAllWaves());
```



Obr. 79 Vizuál skriptu *EnemySpawner.cs*.

Po uložení zmien a návrate do editora Unity môžeme sledovať, že sme schopní vytvoriť pohybujúce sa zhluky nepriateľov, ktoré vznikajú na rôznych cestách. Aby sme však umožnili opakovanie tohto deja, v skripte *EnemySpawner.cs* definujeme logickú premennú *looping*.

```
[SerializeField] bool looping = false;
```

Z funkcie *Start()* spravíme korutinu, ktorá bude riešiť volanie korutiny *SpawnAllWaves()* v závislosti od hodnoty premennej *looping*.

```
IEnumerator Start()
{
    do
    {
        yield return StartCoroutine(SpawnAllWaves());
    } while (looping);
}
```

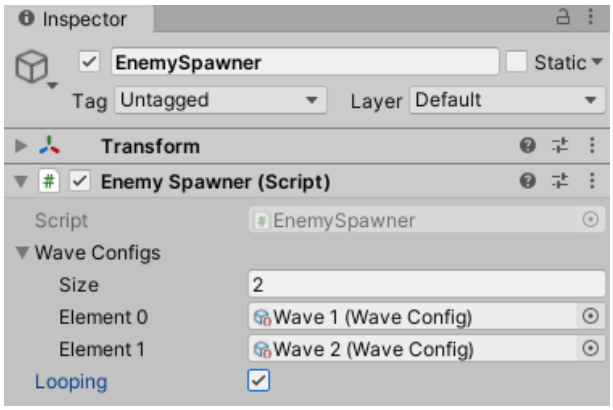
EnemySpawner.cs

Assembly-CSharp — EnemySpawner

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class EnemySpawner : MonoBehaviour
{
    [SerializeField] List<WaveConfig> waveConfigs;
    [SerializeField] bool looping = false;
    int startingWave = 0;

    IEnumerator Start()
    {
        do
        {
            yield return StartCoroutine(SpawnAllWaves());
        } while (looping);
    }

    private IEnumerator SpawnAllEnemiesInWave(WaveConfig waveConfig)
    {
        int tmp = waveConfig.GetNumberOfEnemies();
        for (int enemyCount = 0; enemyCount < tmp; enemyCount++)
        {
            var newEnemy = Instantiate(waveConfig.GetEnemyPrefab(), waveConfig.GetWaypoints()[0].transform.position, Quaternion.identity);
            newEnemy.GetComponent<EnemyPathing>().SetWaveConfig(waveConfig);
            yield return new WaitForSeconds(waveConfig.GetTimeBetweenSpawns());
        }
    }

    private IEnumerator SpawnAllWaves()
    {
        int tmp = waveConfigs.Count;
        for (int waveIndex = startingWave; waveIndex < tmp; waveIndex++)
        {
            var currentWave = waveConfigs[waveIndex];
            yield return StartCoroutine(SpawnAllEnemiesInWave(currentWave));
        }
    }
}
```

Obr. 80 Vizuál skriptu *EnemySpawner.cs*.

Po návrate do editora Unity nesmieme zabudnúť zaškrtnúť premennú *looping*, čím ju nastavíme na hodnotu *true* a riešime tak opätovný vznik rôznych zhlukov nepriateľov, pohybujúcich sa po rôznych cestách (Obr. 81).

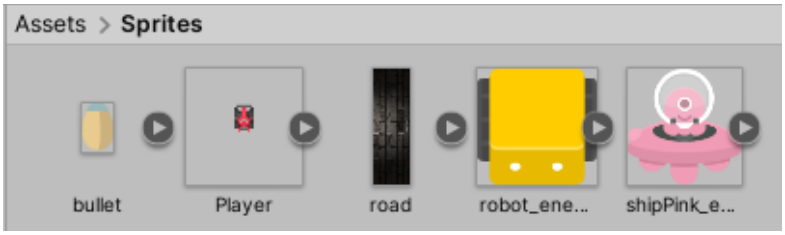


Obr. 81 Nastavenie premennej *looping*.

S využitím tohto hernému mechanizmu nám postačí vytvoriť desiatky ciest s rôznymi nastaveniami a hra tým získa zaujímavý a nepredvídateľný charakter.

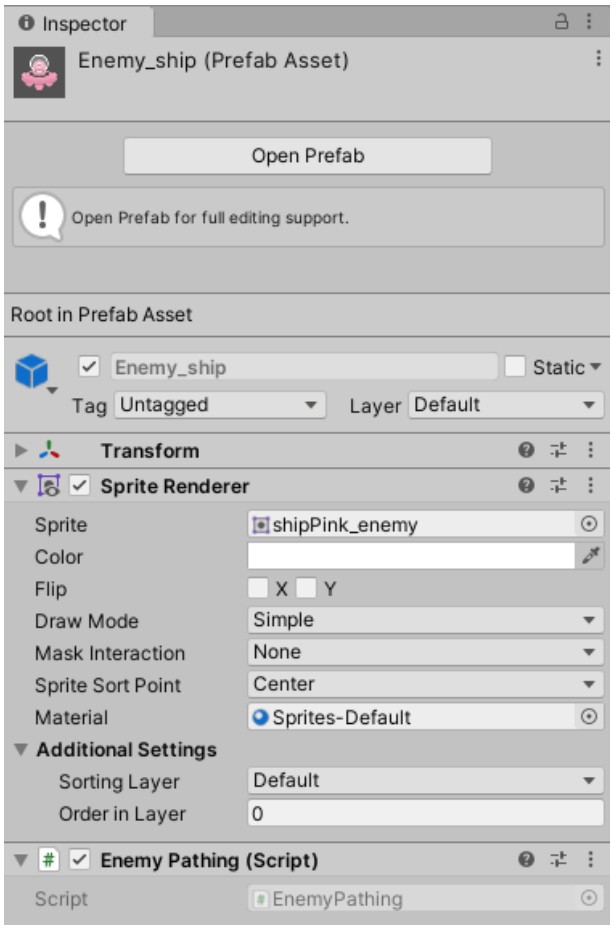
Implementovanú funkcionalitu aplikujeme aj na ďalšieho nepriateľa ktorého vytvoríme. Cieľom je mať dvoch nepriateľov, ktorí sa budú vedieť pohybovať po ceste *Path(0)* alebo *Path(1)*. Študujúcemu odporúčame danú funkcionalitu implementovať samostatne a výsledok skontrolovať podľa časti uvedenej nižšie.

V priečinku *Sprites* už máme vložený *sprite* pre ďalšieho nepriateľa (Obr. 82).



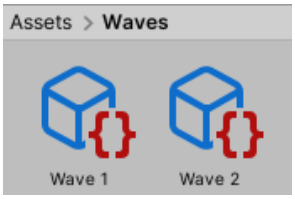
Obr. 82 Vizuálna reprezentácia nepriateľa.

Nového nepriateľa je možné vytvoriť viacerými spôsobmi. My volíme možnosť duplikácie existujúceho *prefabu Enemy*. Pôvodný *prefab* premenujeme na *Enemy_robot* a nový na *Enemy_ship* (z dôvodu jednoduchšej identifikácie). Pri takomto spôsobe vytvorenia nového *prefabu* stačí len zmeniť vizuálnu reprezentáciu objektu pomocou vlastnosti *Sprite* (Obr. 83).



Obr. 83 Vlastnosti *prefabu Enemy*.

Zatiaľ máme v projekte definované len dva *ScriptableObjecty* určujúce vznik zhlukov nepriateľov (Obr. 84).

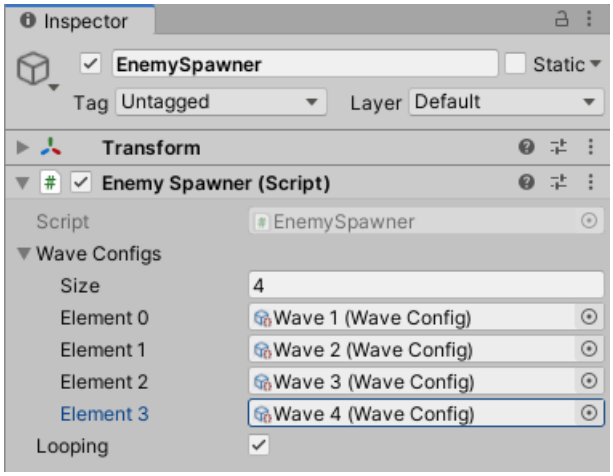


Obr. 84 Cesty definované v projekte.

Vytvoríme si ich duplikáty, ktoré použijeme pre nového nepriateľa. Nastavenia sú plne vo vašej kompetencii. My sme pristúpili k modifikácii, kde *Wave* 1 bude obsahovať jedného nepriateľa a *Wave* 4 ich bude mať štyroch, aby sme vedeli aj vizuálne odlíšiť dané objekty podľa počtu nepriateľov (Obr. 85).

Obr. 85 Nastavenia objektov *Wave* 1 ÷ *Wave* 4.

Nezabudneme tieto rozšírenia aplikovať aj na objekt *EnemySpawner*, kde je nutné zmeniť veľkosť *listu* a inicializovať nové prvky (Obr. 86).



Obr. 86 Nastavenia objektu *EnemySpawner*.

Výsledkom je, že pomocou objektu *EnemySpawner* vieme vytvoriť štyri rôzne zhluky pohybujúcich sa nepriateľov, pričom každý objekt typu *WaveConfig*

môžeme nastavovať podľa preferencií (môžeme určovať počet nepriateľov, ich rýchlosť, ako rýchlo vznikajú, na akej ceste a pod.). Z objektu *EnemySpawner* vytvoríme *prefab*.

4.5 Kontrolné otázky a úlohy

1. Definujte rozdiely medzi dátovými štruktúrami pole (*array*) a zoznam (*list*) v jazyku C#.
2. Pomocou akej metódy vieme zabezpečiť pohyb objektu medzi dvomi pozíciami? Definujte túto metódu.
3. Definujte termín *ScriptableObject*. V akom kontexte používame takýto typ objektu?
4. Objasnite implementovanú hernú mechaniku útoku nepriateľov. Ako by sme vedeli tento herný mechanizmus vylepšiť?
5. Definujte čo je korutina. V akom kontexte sme korutiny využili v našom projekte?
6. Definujte metódy typu *setter* a *getter*.
7. Rozšírte ponuku rôznych typov nepriateľov pohybujúcich sa po rôznych cestách.
8. Implementujte univerzálny mechanizmus vzniku ciest, ktorý zabezpečí automatickú tvorbu ciest a ich bodov.

5 Vývoj obranného a útočiaceho herného mechanizmu

V tejto kapitole sa pokúsime vytvoriť univerzálny *damage* mechanizmus. Základnou myšlienkou je mať jednu triedu, ktorá bude použitá na viacerých miestach (pre rôzne typy nepriateľov, ako aj hráča).

Ako prvé vytvoríme skript *DamageDealer.cs*. V ňom definujeme celočíselnú premennú *damage*, ktorá bude predstavovať zranenie. Skript budeme pridávať ako komponent rôznym *prefabom*. Z tohto dôvodu premennú definujeme ako *SerializeField*, aby sme ju mohli jednoducho editovať v editore Unity.

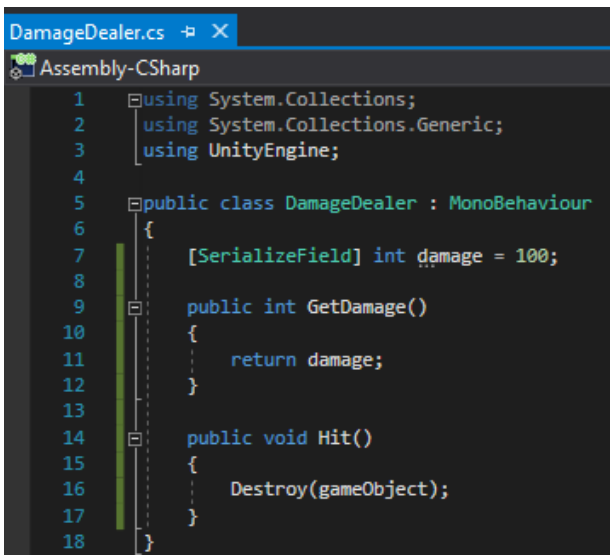
```
[SerializeField] int damage = 100;
```

Pre premennú definujeme *getter*:

```
public int GetDamage()
{
    return damage;
}
```

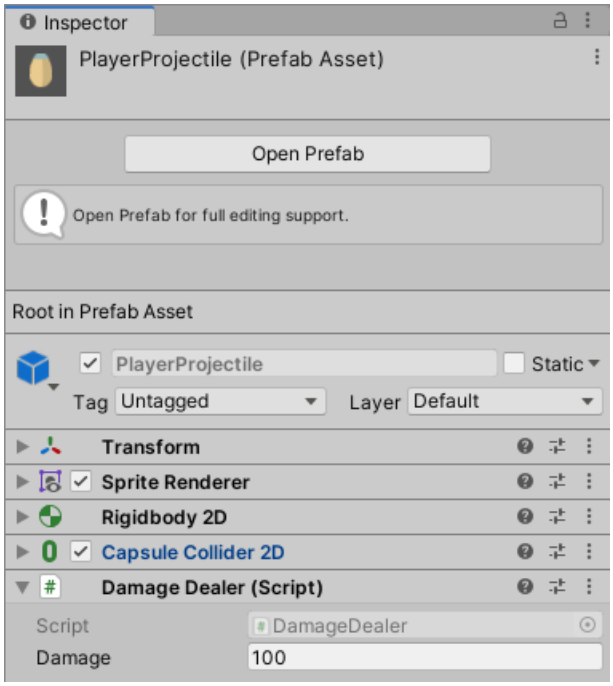
Následne definujeme metódu *Hit()*, ktorá sa nám postará o zrušenie zasiahnutého objektu:

```
public void Hit()
{
    Destroy(gameObject);
}
```



Obr. 87 Vizuál skriptu *DamageDealer.cs*.

Po uložení zmien sa vrátime do editora Unity, kde tento skript pridáme *prefabu PlayerProjectile* (Obr. 88).

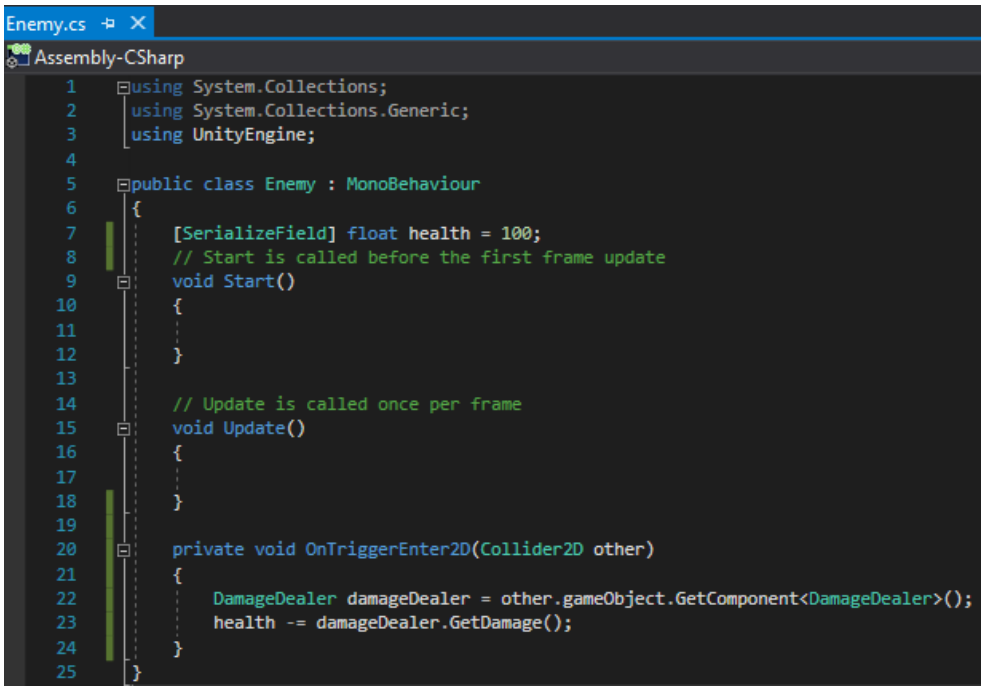


Obr. 88 Pridanie skriptu objektu *PlayerProjectile*.

Následne potrebujeme vytvoriť systém, ktorý bude kontrolovať, či má dôjsť k zničeniu nepriateľa na základe obsahu premennej *health*. V tejto premennej budeme udržiavať informáciu o tom, koľko kvázi bodov má daný nepriateľ, t. j. až po znížení tejto hodnoty na 0 dôjde k zničeniu nepriateľa, pričom sila útoku jednotlivých striel môže byť rôzna. Vytvárame nový skript s názvom *Enemy.cs*, v ktorom definujeme premennú, ktorá bude reprezentovať životnosť (*health*).

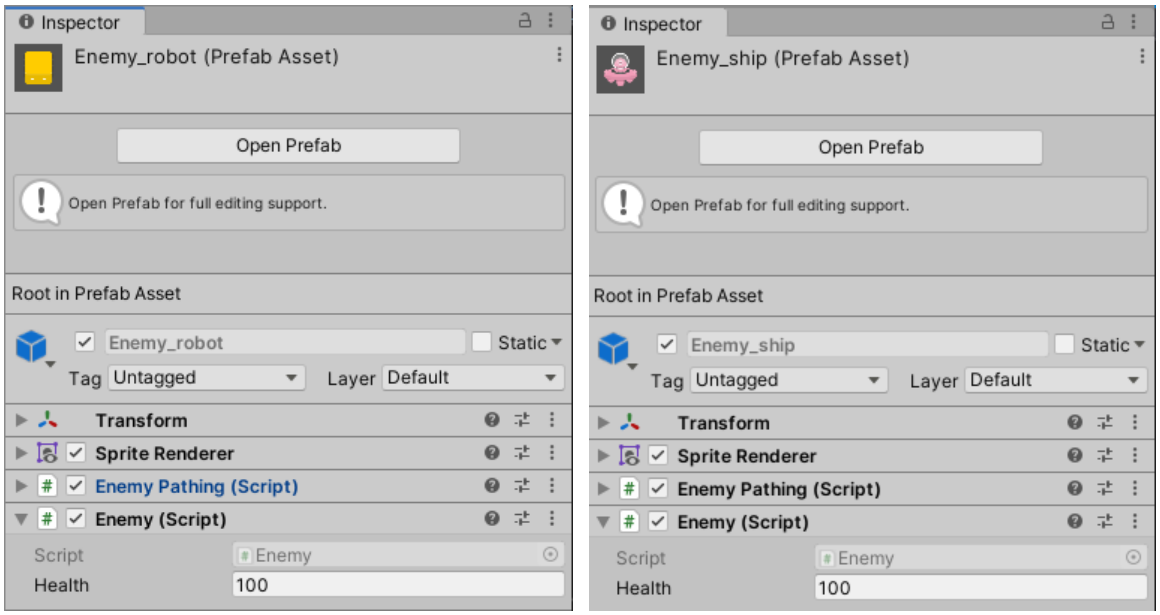
```
[SerializeField] float health = 100;
```

Je zrejmé, že tento mechanizmus budeme riešiť pri kolízii objektov, keď dochádza k vyvolaniu *OnTriggerEnter2D()*. Zadefinujeme lokálnu premennú typu *DamageDealer*, aby sme vedeli pristúpiť k hodnote premennej *damage* definovanej v skripte *DamageDealer.cs*. S akým objektom prišlo ku kolízii vieme podľa parametra *OnTriggerEnter2D()*. Týmto spôsobom môže byť *damage* rôznych objektov nastavený na rôzne hodnoty. V neposlednom rade sa postaráme o aktualizáciu premennej *health* (Obr. 89).



Obr. 89 Vizuál skriptu *Enemy.cs*.

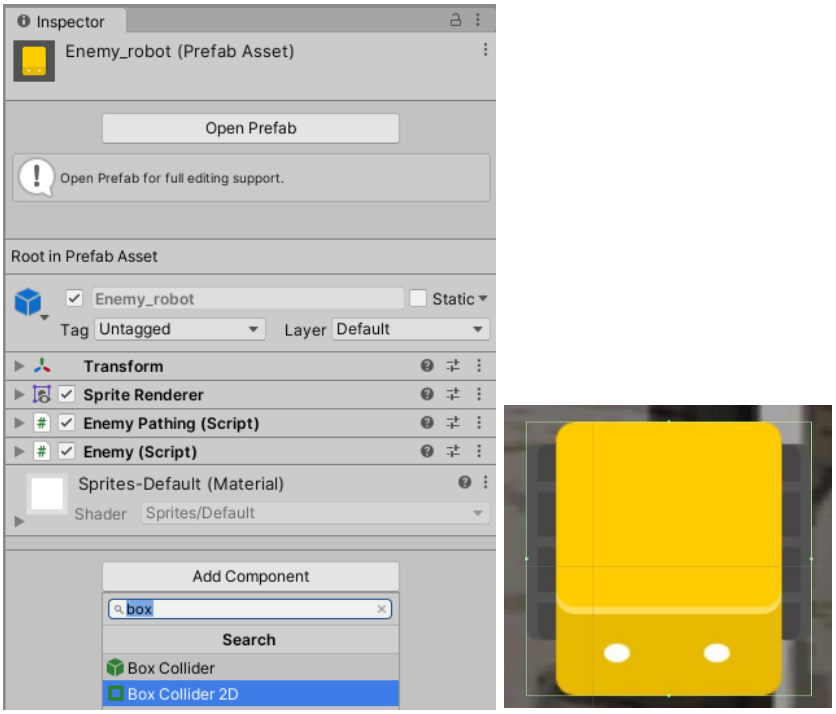
Uložíme zmeny a vrátime sa späť do editora Unity, kde skript *Enemy.cs* pridáme ako komponent *prefabom* nepriateľov (Obr. 90).



Obr. 90 Pridanie skriptu *Enemy.cs* *prefabom* nepriateľov.

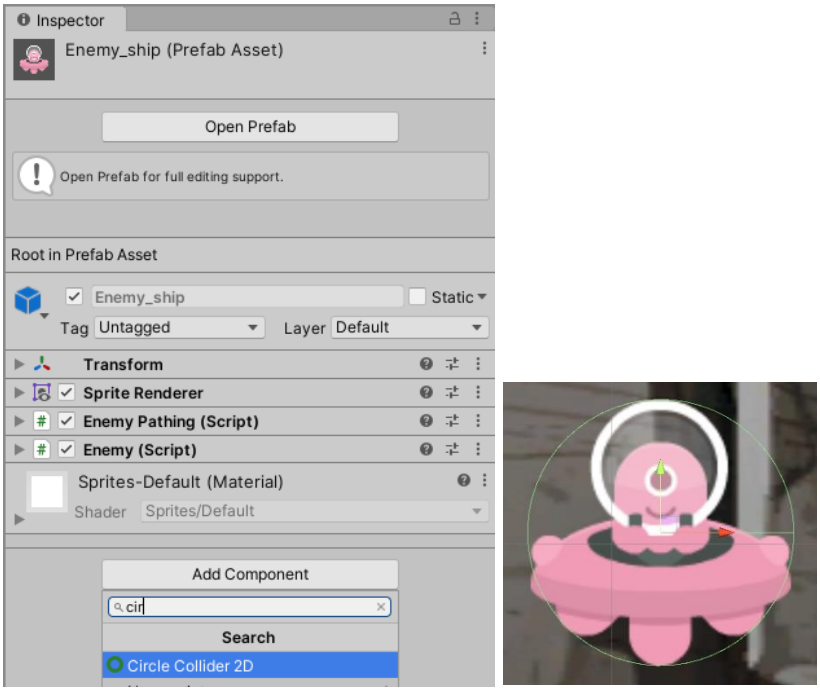
Aby sme mohli detegovať kolízie medzi objektmi, musíme pridať *collider* našim nepriateľom. Viac o problematike kolízie objektov sa môže študujúci dočítať

v skriptách autorky (Jurinová, 2022), konkrétne v kapitole 4. *Prefabu Enemy_robot* pridáme *BoxCollider 2D* (Obr. 91).



Obr. 91 Pridanie *BoxCollideru 2D* objektu *Enemy_robot*.

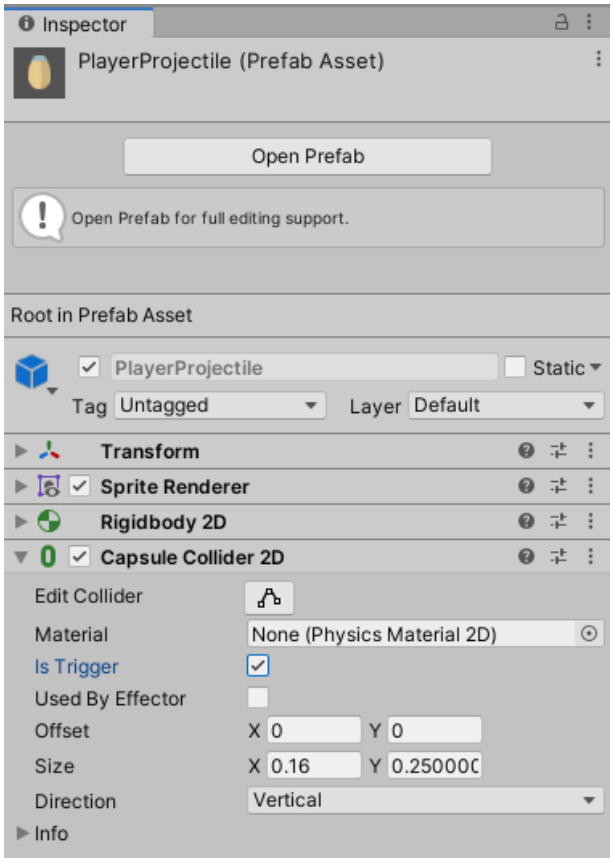
A *prefabu Enemy_Ship* pridáme *Circle Collider 2D* (Obr. 92).



Obr. 92 Pridanie *Circle Collideru 2D* objektu *Enemy_ship*.

Pozn. autora: Ak nerobíte tieto zmeny priamo na *prefabe*, ale na konkrétnej inštancii objektu, nezabudnite zmeny aplikovať pomocou možnosti *Apply All* aby sa odzrkadlili aj na *prefabe*.

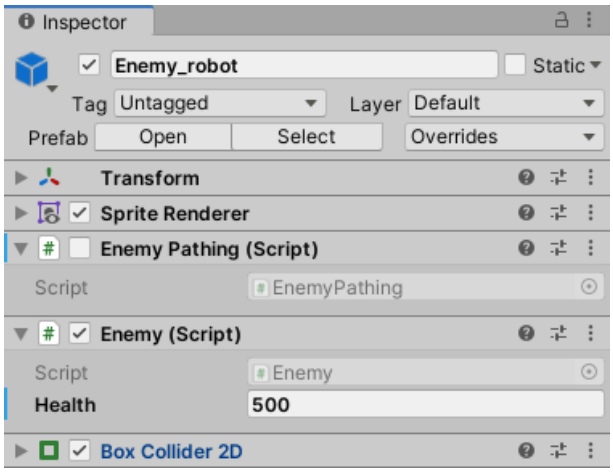
Aby sme vedeli pridať nami požadovanú funkcionalitu pri kolízii dvoch objektov v okne *Inspector* pre *prefab PlayerProjectile*, musíme vlastnosť *Is Trigger* nastaviť na aktívny stav (východiskové nastavenie pre vlastnosť *Is Trigger* je neaktívny stav), čo spravíme zaškrtnutím *checkboxu* (Obr. 93).



Obr. 93 Vlastnosť *Is Trigger*.

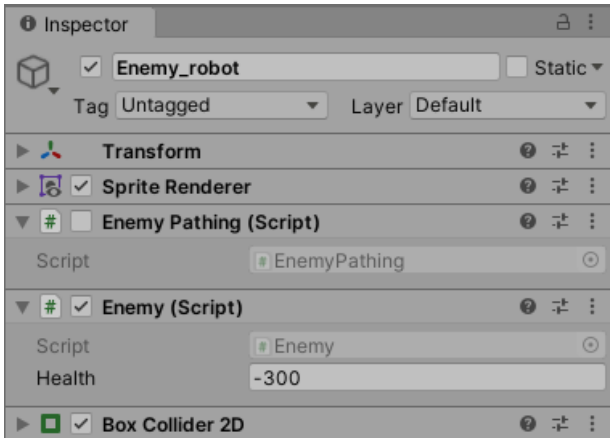
Implementovanú čiastočnú funkcionalitu znižovania životnosti (*health*) nepriateľa vieme jednoducho otestovať tak, že do scény pridáme inštanciu zvoleného nepriateľa, ktorému nastavíme *health* napr. na 500. Je dôležité deaktivovať skript *EnemyPathing* pridaný ako komponent z dôvodu, že sme dynamickým spôsobom implementovali vytváranie zhlukov nepriateľov pohybujúcich sa po definovaných cestách, ktorý teraz popierame. Chceme docieliť v scéne jeden nepohybujúci sa

objekt typu nepriateľa, aby sme jednoducho otestovali implementovanú funkcionalitu. Komponent deaktivujeme odškrtnutím *checkboxu* pri danom komponente (Obr. 94). V opačnom prípade sa môžeme stretnúť s chybovým hlásením *NullReferenceException*.



Obr. 94 Zmena premennej *health* a zneaktívnenie skriptu.

V hernom móde potom sledujeme v okne *Inspector* objektu *Enemy_robot*, či dochádza k znižovaniu hodnoty premennej *health* pri jeho zasiahnutí strelou hráča. Môžeme pozorovať, že v prípade opakovaného zasiahnutia nepriateľa sa hodnota premennej *health* dostáva do záporných hodnôt (Obr. 95), čo je prirodzené, keďže túto časť sme ešte nedoriešili. Cieľom bude zabezpečiť zničenie objektu v momente, keď sa *health* zníži na hodnotu nuly.

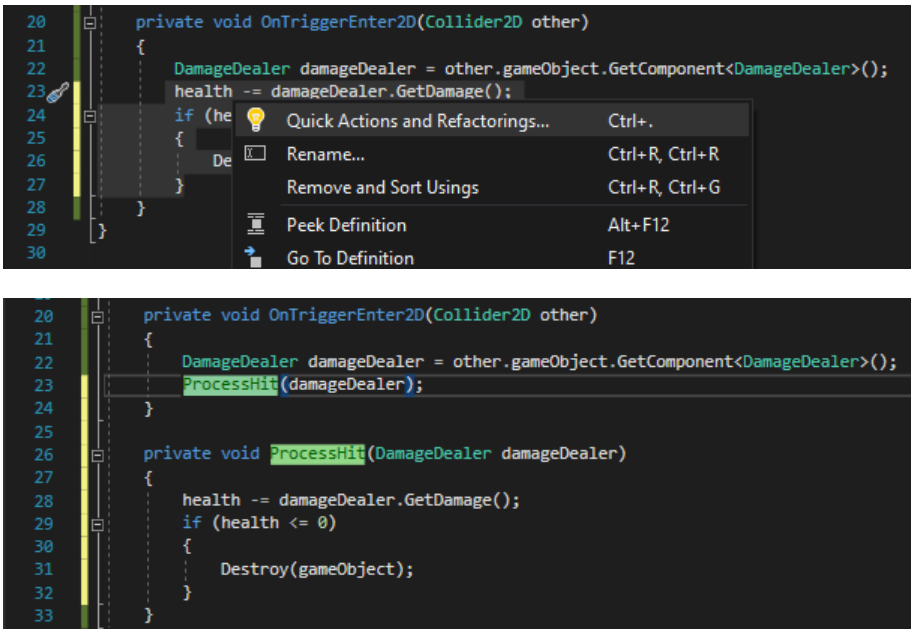


Obr. 95 Sledovanie zmeny premennej *health*.

Zničenie objektu vyriešime doplnením jednoduchej podmienky do *OnTriggerEnter2D()* v skripte *Enemy.cs*:

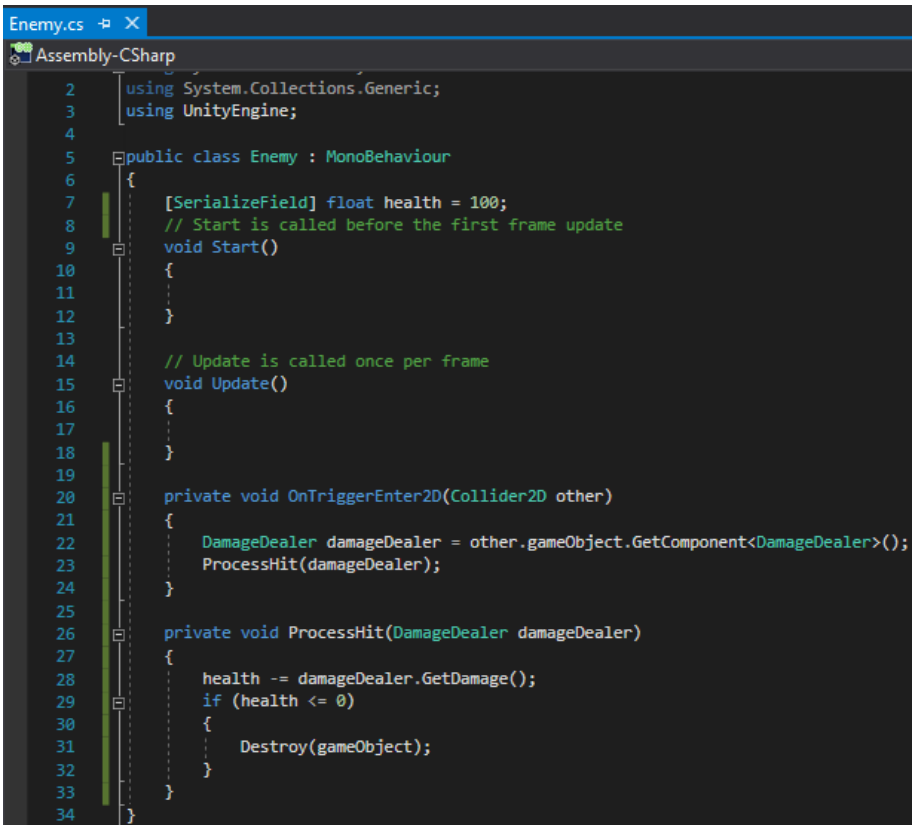
```
private void OnTriggerEnter2D(Collider2D other)
{
    DamageDealer damageDealer = other.gameObject.GetComponent<DamageDealer>();
    health -= damageDealer.GetDamage();
    if (health <= 0)
    {
        Destroy(gameObject);
    }
}
```

Časť, ktorá sa stará o úpravu premennej *health* a následnú kontrolu či netreba zničiť inštanciu objektu, zapúzdrime do novej metódy *ProcessHit()* využitím kontextovej ponuky *Quick Actions and Refactoring* vyvolanej stlačením pravého tlačila myši (Obr. 96).



Obr. 96 Vytvorenie metódy *ProcessHit()*.

Pozn. autora: Všimnite si, že *VisualStudio* hodnotne identifikuje, že v tejto časti kódu sa pracuje s objektom, ktorý táto metóda potrebuje pre správnu funkčnosť a vytvorí teda metódu s parametrom.



Obr. 97 Vizuál skriptu *Enemy.cs*.

5.1 Riešenie streľby nepriateľov

V kapitole 3.1 sme riešili funkcionalitu streľby hráča. Základný rozdiel streľby nepriateľov voči hráčovi je ten, že pri hráčovi streľbu podmieňujeme stlačením tlačidla. Nepriateľ bude schopný strieľať automaticky v určitých časových intervaloch. Tu sa budeme snažiť pridať aj určitú náhodnosť tejto doby. Z tohto dôvodu budeme potrebovať nejaký mechanizmus pre určenie časových intervalov. Preto v skripte *Enemy.cs* deklarujeme premennú *shotCounter*. Táto premenná by nemusela byť nutne *SerializeField*, pretože ju nebudeme chcieť meniť, ale pre účely testovania bude vhodné, ak budeme môcť sledovať jej zmenu v editore Unity.

```
[SerializeField] float shotCounter;
```

Potrebujeme ďalšie dve premenné pre určenie hraníc časového intervalu medzi generovanými strelami.

```
[SerializeField] float minTimeBetweenShot = 0.2f;
[SerializeField] float maxTimeBetweenShot = 3f;
```

Pri zrode každého nepriateľa určíme hodnotou premennej *shotCounter* pomocou metódy *Range()* triedy *Random*, aby sme zabezpečili istú náhodnosť. Túto inicializáciu robíme vo funkcii *Start*().

```
void Start()
{
    shotCounter = Random.Range(minTimeBetweenShot, maxTimeBetweenShot);
}
```

Pozn. autora: Ak by ste mali problém s metódu *Range*(), stačí daný riadok upraviť nasledovne:

```
shotCounter = UnityEngine.Random.Range(minTimeBetweenShot, maxTimeBetweenShot);
```

Vo funkcii *Update*() sa postaráme o zabezpečenie strieľania pomocou novo definovanej metódy *Fire*(). Túto funkcionalitu zapúzdrime do metódy *CountDownAndShoot*(), ktorej základná logika spočíva v zmene počítadla *shotCounter*, ktorého hodnota sa každý *frame* znižuje a v momente keď dosiahne hodnotu 0, dôjde k vypáleniu strely.

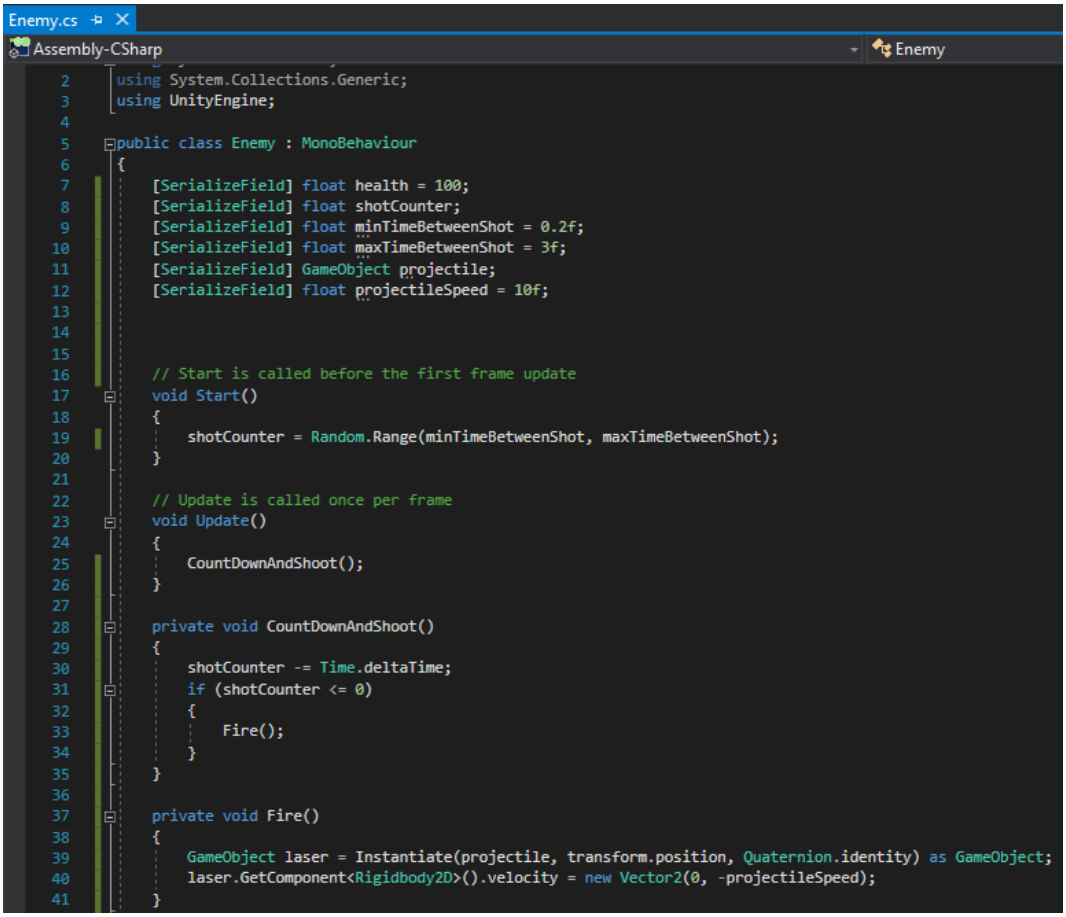
```
private void CountDownAndShoot()
{
    shotCounter -= Time.deltaTime;
    if (shotCounter <= 0)
    {
        Fire();
    }
}
```

V metóde *Fire*() sa postaráme o vytvorenie inštancie strely a zabezpečíme jej pohyb obdobným spôsobom, ako sme realizovali pri hráčovi. Pre správnu funkčnosť metódy je dôležité vytvoriť dve premenené. Premenná *projectile* bude predstavovať samotný objekt strely a *projectileSpeed* bude určovať rýchlosť pohybu strely.

```
[SerializeField] GameObject projectile;
[SerializeField] float projectileSpeed = 10f;
```

V metóde *Fire()* sa postaráme o vznik novej inštancie strely a zabezpečíme jej pohyb pomocou vlastnosti *Velocity*. Hodnota *y* premennej typu *Vector2* nadobúda zápornú hodnotu z dôvodu zabezpečenia pohybu strely zhora nadol (Obr. 98).

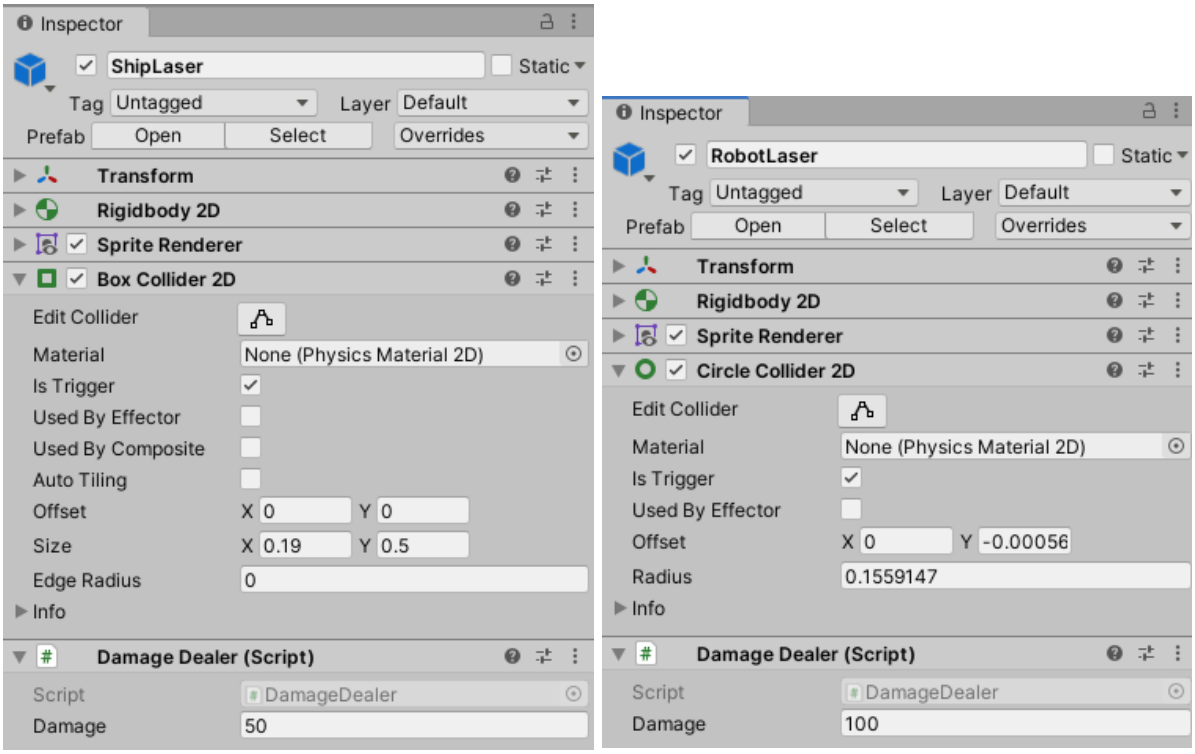
```
private void Fire()
{
    GameObject laser = Instantiate(projectile, transform.position,
          Quaternion.identity) as GameObject;
    laser.GetComponent<Rigidbody2D>().velocity = new Vector2(0, -projectileSpeed);
}
```



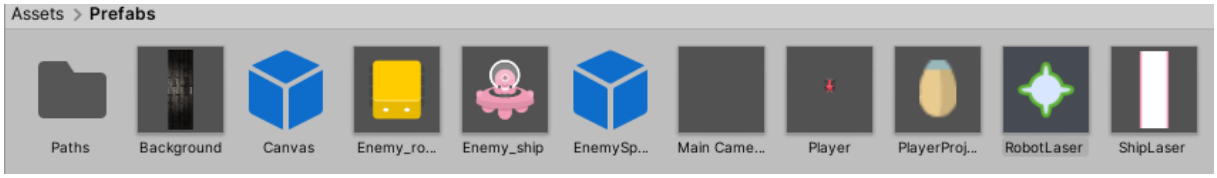
Obr. 98 Vizuál fragmentu kódu skriptu *Enemy.cs*.

Uložíme zmeny a vrátime sa do editora Unity, kde si vytvoríme nové *prefaby*, ktoré budú reprezentovať strely nepriateľov. Tieto môžeme vytvoriť na základe existujúceho *prefabu PlayerProjectile* a jeho úpravou, alebo si vytvoríme nový prázdny *Game Object*, ktorý pomenujeme *ShipLaser*. Pridáme mu komponent *RigidBody2D*, kde nastavujeme *Body Type* na *Kinematic* a *Sprite Renderer*, kde vo vlastnosti *Sprite* zvolíme vhodnú vizuálnu reprezentáciu strely (použili sme

laserPink). Objektu pridáme *Box Collider 2D* a vlastnosť *Is Trigger* nastavíme na aktívnu. Robíme to z dôvodu, že v prípade kolízie objektov chceme, aby nastala nami implementovaná funkcionalita. Skript *DamageDealer.cs* pridáme ako komponent a môžeme v ňom nastaviť silu útoku napríklad na 50. Z objektu nezabudneme vytvoriť *prefab*. Obdobným spôsobom vytvoríme aj *prefab RobotLaser*, v ktorom využijeme *Circle Collider 2D* vzhľadom na vizuálnu reprezentáciu strely (Obr. 99).

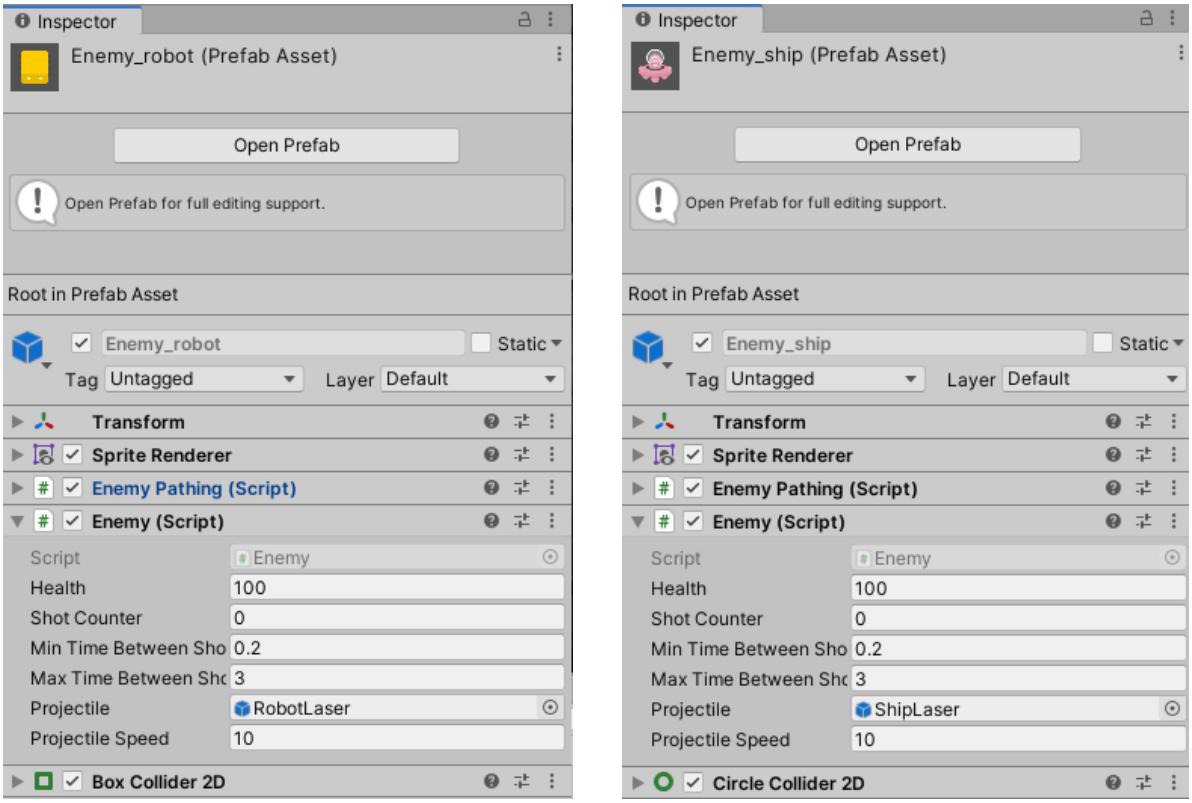


Obr. 99 Nastavenia objektov *ShipLaser* a *RobotLaser*.



Obr. 100 Vizuál vytvorených *prefabov*.

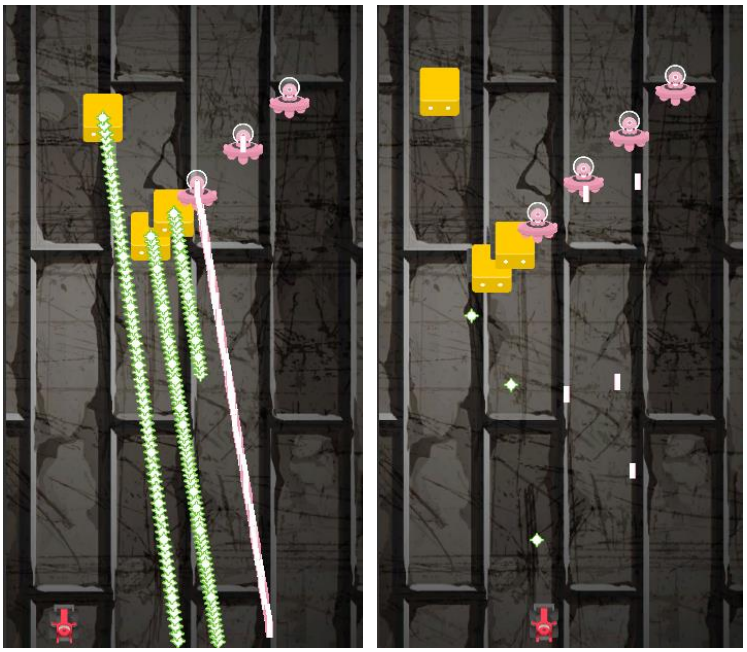
Vytvorené *prefaby* striel priradíme zodpovedajúcim nepriateľom (Obr. 101).



Obr. 101 Inicializácia premennej *projectile* vytvorenými *prefabmi* striel.

Po otestovaní funkčnosti si môžeme všimnúť, že to nefunguje podľa našich predstáv. Jeden z problémov je, že neresetujeme premennú *shotCounter*, čo spôsobuje, že strely sú produkované ako keby neustále (Obr. 102). Z tohto dôvodu pridáme nastavenie premennej obdobným spôsobom, ako pri jej inicializácii ihneď po volaní metódy *Fire*(), čím zabezpečíme očakávané správanie.

```
private void CountDownAndShoot()
{
    shotCounter -= Time.deltaTime;
    if (shotCounter <= 0)
    {
        Fire();
        shotCounter = Random.Range(minTimeBetweenShot, maxTimeBetweenShot);
    }
}
```



Obr. 102 Obrazovka ilustrujúca streľbu nepriateľov pred a po úprave.

5.2 Ovplyvňovanie života postavy hráča a jeho smrť

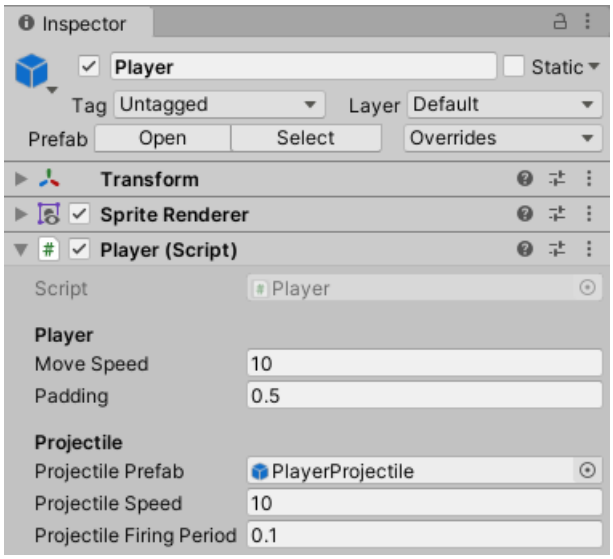
V tejto kapitole sa postaráme o znižovanie života (*health*) hráča po zasiahnutí strelou nepriateľa, prípadne jeho smrť. Budeme pracovať so skriptom *Player.cs*, kde najprv vizuálne sprehľadníme časť definície premenných. Pomocou atribútu *HeaderAttribute* doplníme jednoduchú kategorizáciu premenných (Obr. 103), ktorá sa prejaví v okne *Inspector* ako ilustruje obrázok 104.

Player.cs

Assembly-CSharp

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class Player : MonoBehaviour
6   {
7       //configuration parameters
8       [Header("Player")]
9       [SerializeField] float moveSpeed = 10f;
10      [SerializeField] float padding = 1f;
11
12      [Header("Projectile")]
13      [SerializeField] GameObject projectilePrefab;
14      [SerializeField] float projectileSpeed = 10f;
15      [SerializeField] float projectileFiringPeriod = 0.1f;
16
17      //reference parameters
18      Coroutine firingCoroutine;
```

Obr. 103 Vizuál fragmentu kódu skriptu *Player.cs*.



Obr. 104 Okno *Inspector* po doplnení atribútu *Header*.

V skripte *Player.cs* v časti „Player“ definujeme novú premennú *health*, ktorá bude predstavovať život hráča.

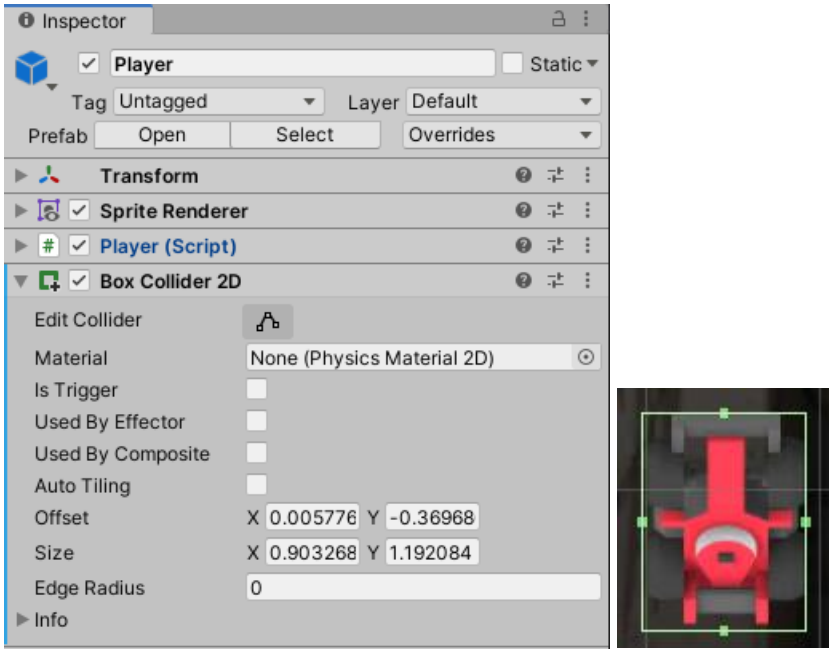
```
[SerializeField] int health = 200;
```

Obdobnú funkcionalitu znižovania *health* sme riešili aj v skripte *Enemy.cs*. Na prvý pohľad by sa mohlo zdať, že potrebujeme rovnakú funkcionalitu riešiť na dvoch miestach. To by nás mohlo priviesť k nápadu vytvoriť si skript, v ktorom by sme tento mechanizmus implementovali, a pridať ho ako komponent objektu *Enemy* aj *Player*. Každopádne po aplikovaní rôznych efektov a ďalších rozšírení hry uvidíme, že veci sa začnú mierne odlišovať a je preto vhodnejšie riešiť to individuálne. Napríklad ak postava hráča zomrie, malo by dôjsť k načítaniu záverečnej scény, čo pri usmrtení nepriateľa nenastáva.

V skripte *Player.cs* definujeme metódu *ProcessHit()* v ktorej sa postaráme o znižovanie života (*health*) hráča a v prípade, ak klesne táto na 0, postaráme sa o zničenie inštancie hráča. Metódu potrebujeme zavolať v momente ako nastane kolízia projektilu nepriateľa s objektom hráča, preto ju voláme z *OnTriggerEnter2D()*.

```
private void ProcessHit(DamageDealer damageDealer)
{
    health -= damageDealer.GetDamage();

    if (health <= 0)
    {
        Destroy(gameObject);
    }
}

private void OnTriggerEnter2D(Collider2D other)
{
    DamageDealer damageDealer = other.gameObject.GetComponent<DamageDealer>();
    ProcessHit(damageDealer);
}
```

Uložíme zmeny a vrátime sa do editora Unity, kde objektu *Player* pridáme *Box Collider 2D* (Obr. 105). Nezabudneme zmeny aplikovať na *prefab*.



Obr. 105 Pridanie *Box Collideru 2D* objektu *Player*.

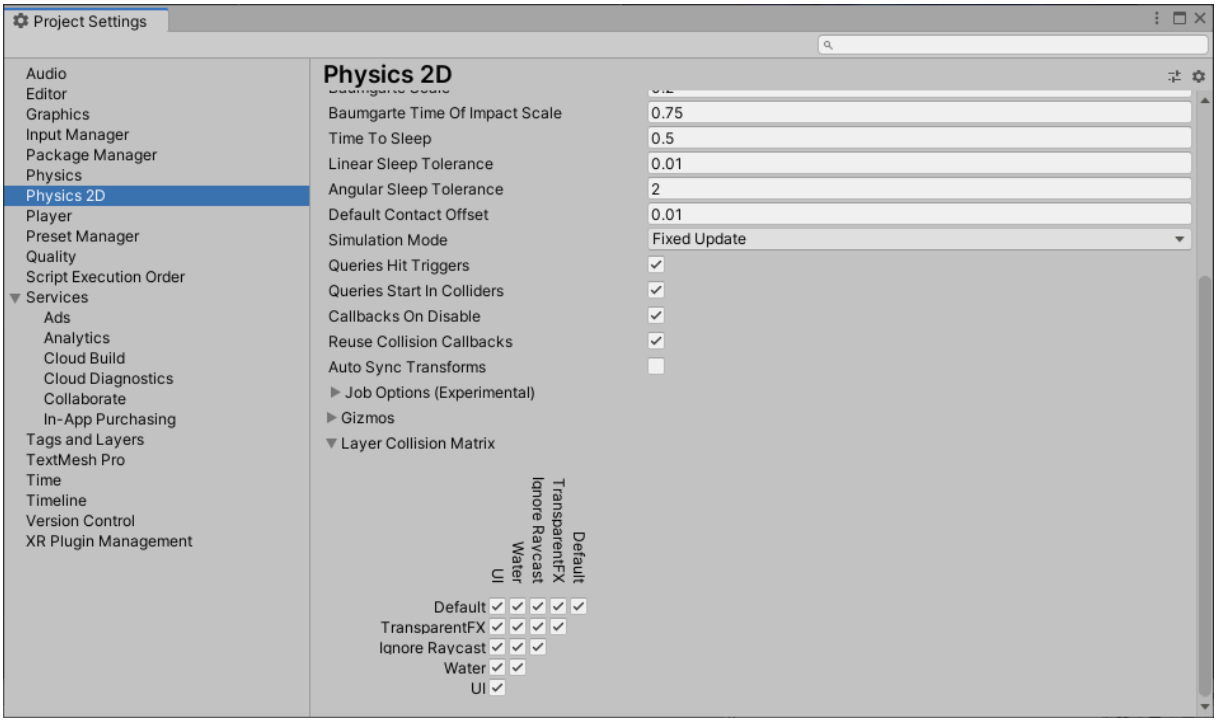
Ak teraz otestujeme funkčnosť pri sledovaní premennej *health* objektu *Player* v okne *Inspector*, môžeme vidieť, že skutočne po zasiahnutí strelou sa táto hodnota aktualizuje. V prípade dosiahnutia hodnoty 0 dôjde k zničeniu inštancie hráča. Hra zdanlivo pokračuje ďalej. Týmito a ďalšími detailami riešení funkcionality hry sa zaoberáme v nasledujúcej kapitole.

5.3 Vyladenie procesu útoku

V prípade seriózneho sledovania zatiaľ implementovanej funkcionality ste si mohli všimnúť, že nepriatelia sa ničia navzájom, strely nepriateľov pri zásahu hráča prechádzajú cez neho a nedochádza k ich zničeniu.

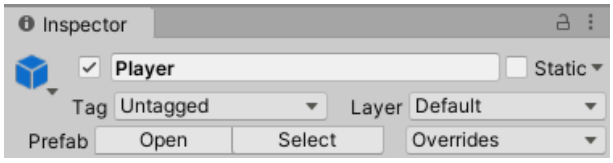
Cieľom je zabezpečiť, aby strely nepriateľov reagovali znížením *health* hráča, a nie aby útočili sami na seba, prípadne aby jeden typ nepriateľa likvidoval iný typ.

Túto funkcionalitu zabezpečíme nastaveniami v *Layer Collision Matrix*, ktorú nájdeme v ponuke *Edit* → *Project Settings* → *Physics 2D*. Na obrázku 106 môžeme vidieť päť základných vrstiev (*layer*), ktoré Unity ponúka: *Default*, *TransparentFX*, *Ignore Raycast*, *Water* and *UI*. Tieto vrstvy je možné dopĺňať vlastnými a nastavovať potom vzájomné interakcie objektov (Unity, 2021d).



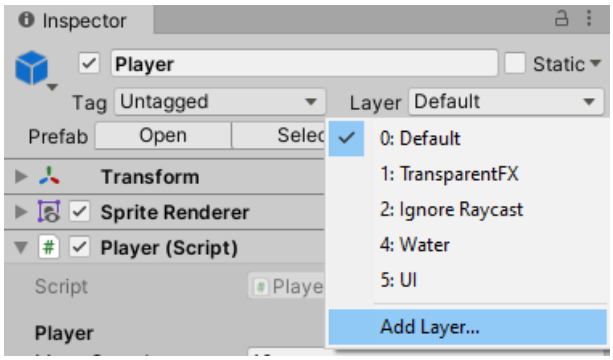
Obr. 106 *Layer Collision Matrix*.

Pri zobrazení vlastností objektu *Player* v okne *Inspector* si môžeme všimnúť, že tento objekt je umiestení vo vrstve *Default* (Obr. 107).



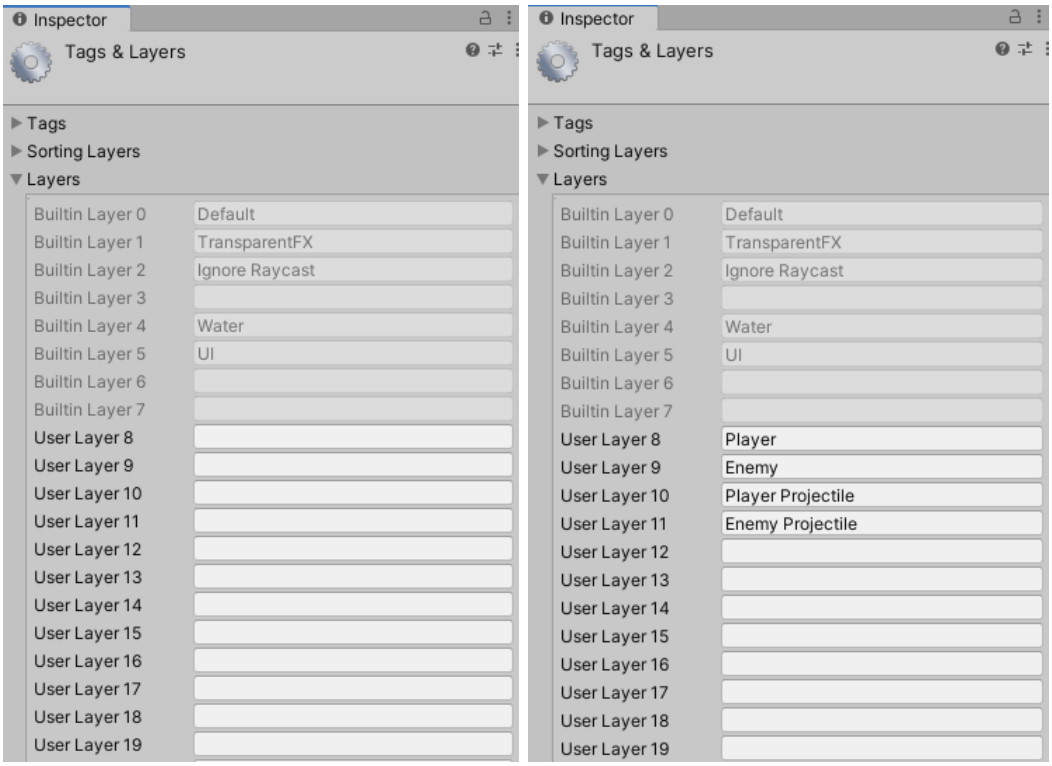
Obr. 107 Objekt *Player* vo vrstve *Default*.

Rozbalením ponuky pomocou možnosti *Add Layer* vieme pridávať nové vrstvy (Obr. 108).



Obr. 108 Pridanie nových vrstiev.

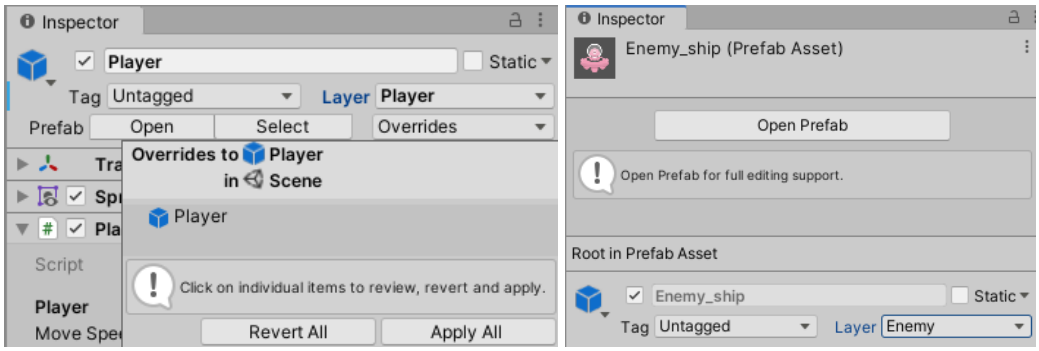
Prvých osem vrstiev je vyhradených, preto začíname nami vytvorené vrstvy definovať až následne, ako to ilustruje obrázok 109. Pridané vrstvy reprezentujú všetky objekty našej hry, medzi ktorými by mohlo prichádzať k vzájomnej interakcii.



Obr. 109 Východiskové a upravené nastavenia pre vrstvy.

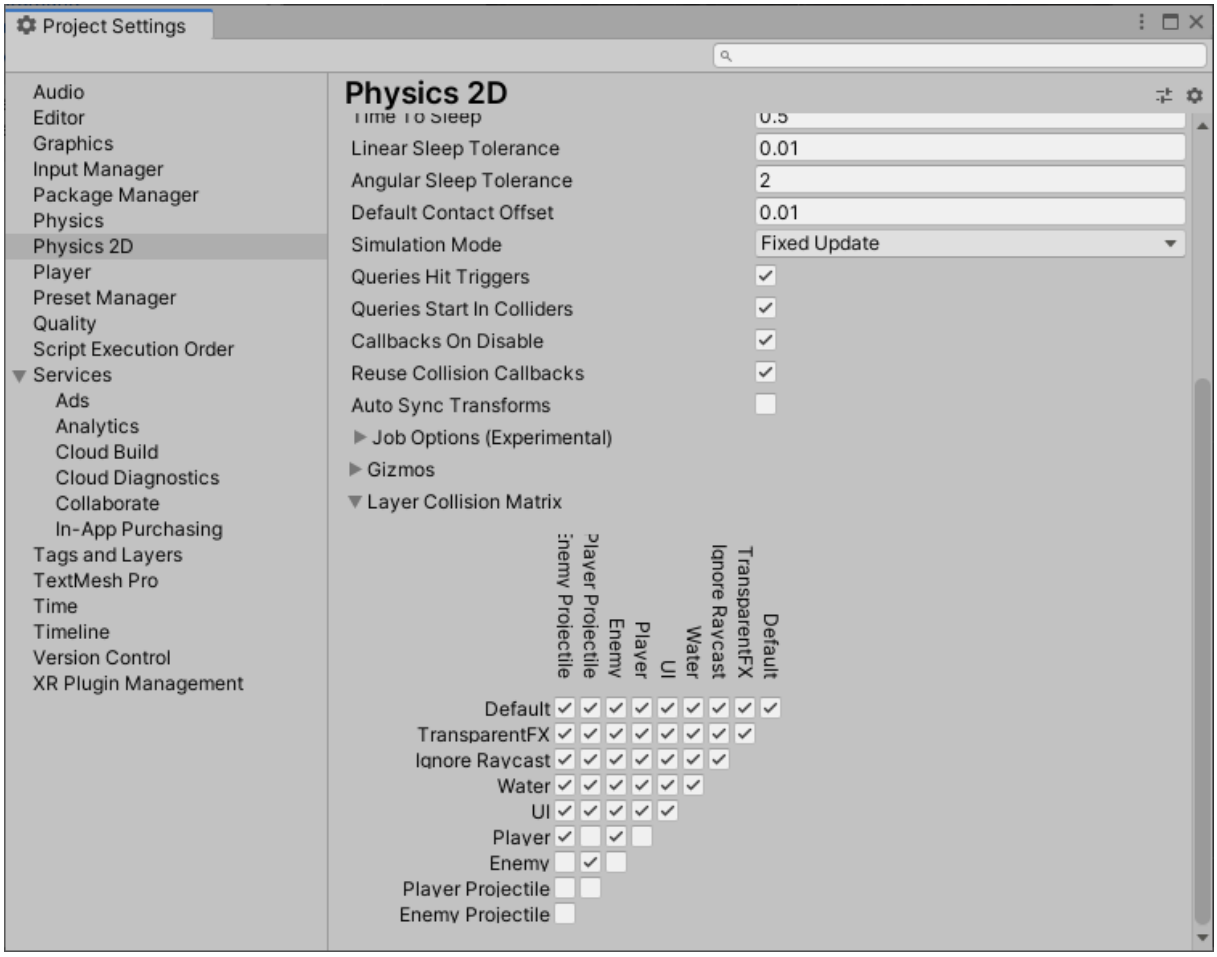
Po pridaní vrstiev realizujeme nasledovné úpravy:

- objekt *Player* umiestnime do vrstvy *Player* (Obr. 110) a vykonané zmeny aplikujeme na *prefab*,
- *prefab Enemy_ship* umiestnime do vrstvy *Enemy* (Obr. 110),
- *prefab Enemy_robot* umiestnime do vrstvy *Enemy*,
- *prefab RobotLaser* umiestnime do vrstvy *Enemy Projectile*,
- *prefab ShipLaser* umiestnime do vrstvy *Enemy Projectile*,
- *prefab PlayerProjectile* umiestnime do vrstvy *Player Projectile*.



Obr. 110 Ukážka priradenia definovanej vrstvy objektua *prefabu*.

Teraz sa vrátime opätovne do nastavení *Layer Collision Matrix* cez možnosť *Edit → Project Settings → Phycis 2D*, kde upravíme vzájomné interakcie. Zaškrtnutý *checkbox* znamená áno (má dochádzať k interakcii), nezaškrtnutý nie (nemá dochádzať k interakcii). Cieľom je zabezpečiť, aby dochádzalo k vzájomnej interakcii hráča a projektilov nepriateľov a obrátene, aby projektily hráča interagovali s objektmi nepriateľov. Tu môžeme reagovať aj na interakciu samotných objektov nepriateľov a hráča (Obr. 111). Z nastavení je zrejmé, že môžeme spôsobiť vzájomnú interakciu aj medzi nepriateľmi (napr. ich vzájomné odrazenie sa a pod.), resp. hráča samého so sebou alebo s vlastnými projektilmi (napr. v prípade odrazu hráčovho projektilu a jeho zasiahnutí by mohol poškodiť sám seba).



Obr. 111 Nastavenia *Layer Collision Matrix*.

Teraz sa postaráme o zničenie striel v prípade zásahu. V skripte *DamageDealer.cs* máme definovanú metódu *Hit()*, ktorá sa postará o zničenie objektu. Zničenie striel je potrebné upraviť na dvoch miestach: v prípade zásahu nepriateľa v skripte *Enemy.cs* alebo zásahu hráča v skripte *Player.cs*. Alternatívnym riešením je mať definovanú triedu *Health*, ktorá by riešila túto implementáciu na jednom mieste.

V skripte *Player.cs* doplníme volanie metódy *Hit()* triedy *DamageDealer* v mieste, kde dochádza k úprave životnosti postavy hráča, t. j. v momente kedy je hráč zasiahnutý.

```
private void ProcessHit(DamageDealer damageDealer)
{
    health -= damageDealer.GetDamage();
    damageDealer.Hit();

    if (health <= 0)
    {
        Destroy(gameObject);
    }
}
```

Obdobne to spravíme aj v skripte *Enemy.cs*. Po uložení zmien a otestovaní funkčnosti v editore Unity môžeme vidieť, že to funguje korektne. Dôjde k odstráneniu strely ak zasiahneme nepriateľa, ako aj opačne, odstráni sa strela, ak nepriateľ zasiahne hráča.

V implementovanom kóde máme jednu potencionálne nebezpečnú časť, ktorá by mohla vyvolať chybové hlásenie „*NullReferenceExeceptions*“. Ide o časť v skripte *Player.cs* a *Enemy.cs*, kde kontrolujeme kolízie a kde má nastať zavolanie metódy *ProcessHit()*. Metóda očakáva parameter typu *DamageDealer* a ak nastane interakcia s objektom, ktorý tento komponent nemá, metóda vlastne nemá aký objekt odovzdať. Ošetrenie zabezpečíme jednoducho otestovaním obsahu premennej *damageDealer*, či je táto hodnota rôzna od null (nulový pointer). V prípade ak áno, nastane zavolanie metódy *ProcessHit()*, ak nie, postaráme sa o ukončenie metódy.

Pôvodná verzia *OnTriggerEnter2D*():

```
private void OnTriggerEnter2D(Collider2D other)
{
    DamageDealer damageDealer = other.gameObject.GetComponent<DamageDealer>();
    ProcessHit(damageDealer);
}
```

Upravená verzia *OnTriggerEnter2D*():

```
private void OnTriggerEnter2D(Collider2D other)
{
    DamageDealer damageDealer = other.gameObject.GetComponent<DamageDealer>();
    if (!damageDealer) { return; }
    ProcessHit(damageDealer);
}
```

Pozn. autora: Daný problém sme označili ako potencionálny, pretože ak pracujeme sofistikovane s nastaveniami v *Layer Collision Matrix*, nemalo by k takémuto stavu dôjsť. Je však vhodné túto situáciu vždy ošetriť.

5.4 Kontrolné otázky a úlohy

1. Akú funkcionalitu riešime v skripte *DamageDealer.cs*? Aké objekty tento skript obsahuje? V akých ďalších situáciách ho vieme použiť?
2. Čo vyjadruje termín *collision*?
3. Z akého dôvodu pridávame objektom komponent *Rigidbody 2D*?
4. Podľa vlastnosti *Body Type*, aké možnosti nastavenia *Rigidbody* poznáme?
5. Čo vyjadruje termín *collider*? Aké typy *colliderov* poznáte?
6. Aká je funkcionalita vlastnosti *Is Trigger*?
7. Ako sa volajú funkcie na obsluhu udalostí, ktoré sú volané v prípade kolízie dvoch objektov?
8. Z akého dôvodu sme nastavili premennú *health* ako *SerializeField*, keďže nemá logické opodstatnenie modifikovať ju manuálne v editore Unity?
9. Ako vieme zistiť interval medzi poslednou a aktuálnou snímkou (*frame*)?
10. Ako sme zabezpečili streľbu nepriateľov?

11. Identifikáciu kolidujúcich objektov vieme riešiť použitím *tagov*, alebo umiestnením objektov do vrstiev (*layers*). Ako a kde potom nastavujeme želanú interakciu objektov?
12. Kedy má prísť k odstráneniu strely nepriateľa? Kde implementujeme požadovanú funkcionalitu?
13. Rozšírte funkcionalitu hry tak, že v prípade stretu hráča s nepriateľom dôjde k ich vzájomného zápasu a nie k smrti postavy hráča ako teraz.
14. Rozšírte funkcionalitu hry tak, že pri vzájomnom strete nepriateľov nastane zmena ich pohybu (napríklad sa odrazia od seba).
15. Rozšírte funkcionalitu hry tak, že zabezpečíte odrazenie striel hráča, ktoré môžu spôsobiť zranenie aj samotnému hráčovi.
16. Vytvorte triedu *Health* v ktorej zapúzdrite všetky požadované funkcionality tejto triedy.

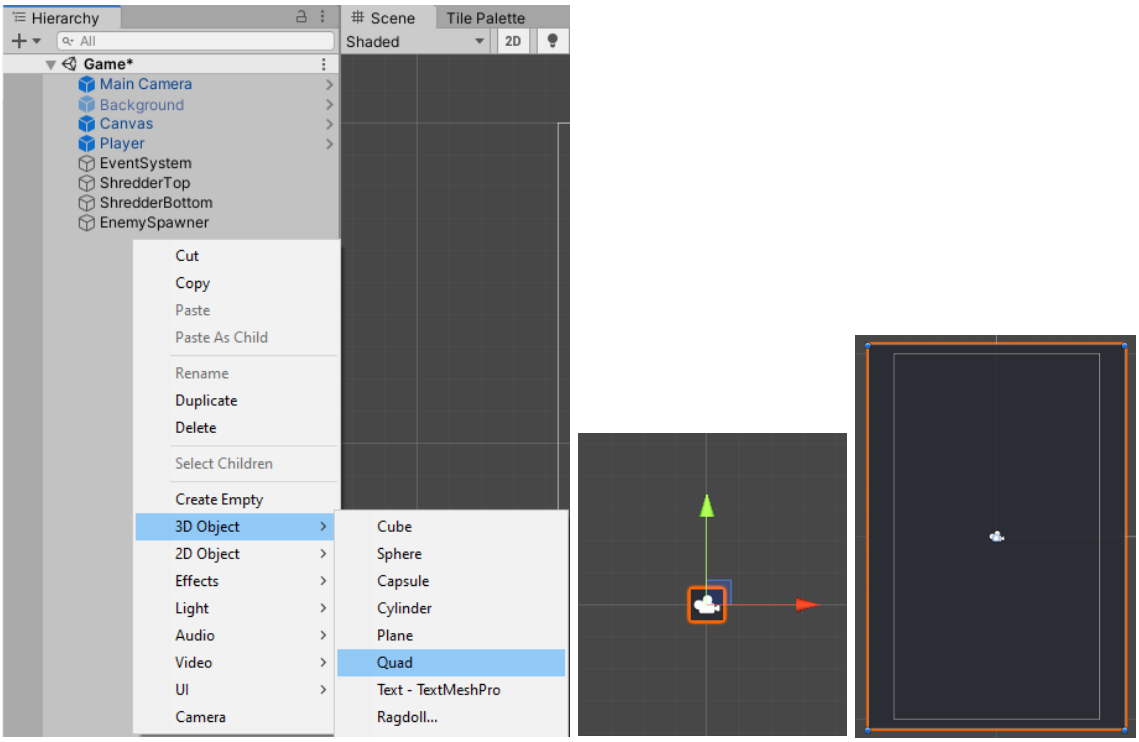
6 Dolaďovanie vizuálnej stránky hry

Typickým znakom hier typu *Top-down Shooter Game* (TDS) je tzv. skrolovateľné pozadie. V tejto kapitole sa postaráme o jeho implementáciu. Okrem toho pozornosť venujeme pridaniu efektov pomocou *Particle System*, ako aj zvukových stôp.

6.1 Skrolovateľné pozadie (*scrolling background*)

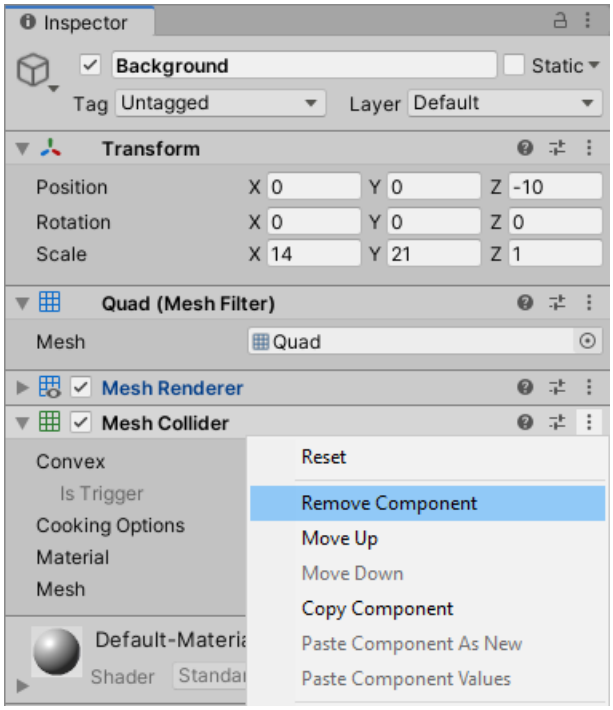
Aktuálne je naše pozadie vytvorené jednoducho pomocou 2D textúry (*sprite*), čo nám obmedzuje kreatívne možnosti. Cieľom je zabezpečiť efekt, že hráč bude mať pocit ako keby sa pozadie pohybovalo. V Unity práve trieda *Mesh* poskytuje možnosť tvorby a úpravy geometrického modelu objektu počas vykonávania aplikácie (*runtime*). Veľmi často sa to využíva pri grafických efektoch, kedy potrebujeme modifikovať model objektu, napr. ilustrácia demolácie auta a pod. Táto problematika práce s 3D modelmi je však nad rámec zámeru týchto skrípt, preto k tejto časti pristupujeme veľmi elementárne vzhľadom na naše potreby (Meshes, 2021e).

My v tomto kontexte využijeme vytvorenie 3D objektu *quad*, ktorý reprezentuje jednu stranu kocky (*flat surface*). Na rozdiel od *sprite* nám to umožní aplikovať materiál, pomocou ktorého vytvoríme efekt posúvajúceho pozadia. V okne *Hierarchy* sa postaráme o vytvorenie takéhoto typu objektu a pomenujeme ho *Background*. Môžeme si všimnúť, že novo vytvorený objekt je ohraničený oranžovým lemom. Jeho veľkosť prispôsobíme veľkosti herného sveta (Obr. 112). Pôvodný objekt *Background* môžeme zo scény vymazať, nebudeme ho už potrebovať.



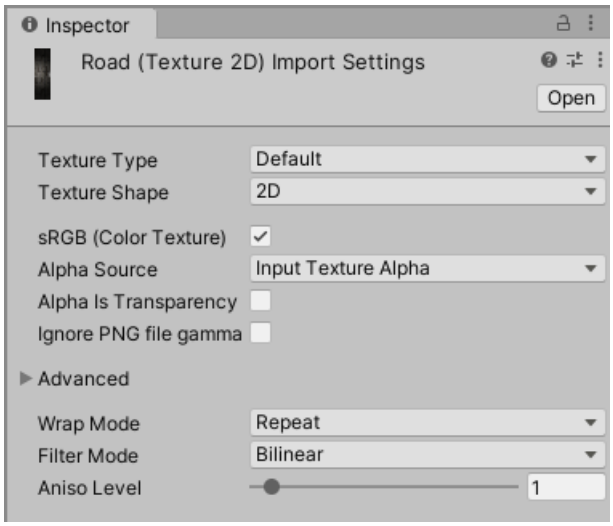
Obr. 112 Vytvorenie objektu *quad* a jeho nastavenia.

Objekt tohto typu má viacero komponentov, čo vidíme v okne *Inspector*. Momentálne sa postaráme o odstránenie komponentu *Mesh Collider*, ktorý nepotrebujeme (Obr. 113).



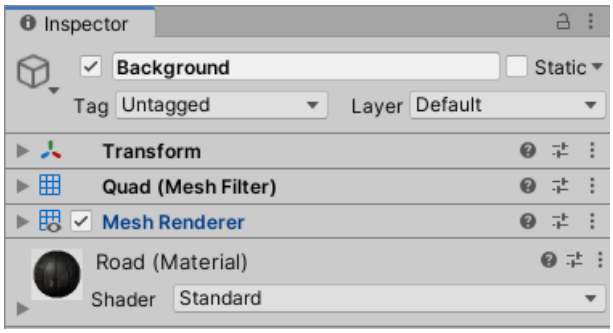
Obr. 113 Odstránenie komponentu.

Ako prvé upravíme vlastnosti *spritu road*, ktorý predstavuje naše pozadie. Vlastnosť *Texture Type* nastavujeme na *Default*. Robíme to z toho dôvodu, že ide o najbežnejší typ používaný pri textúrach, ktorý poskytuje prístup k väčšine vlastností, ktorými vieme modifikovať správanie. Wrap Mode nastavíme na *Repeat* (Obr. 114). Zmeny nezabudneme aplikovať.



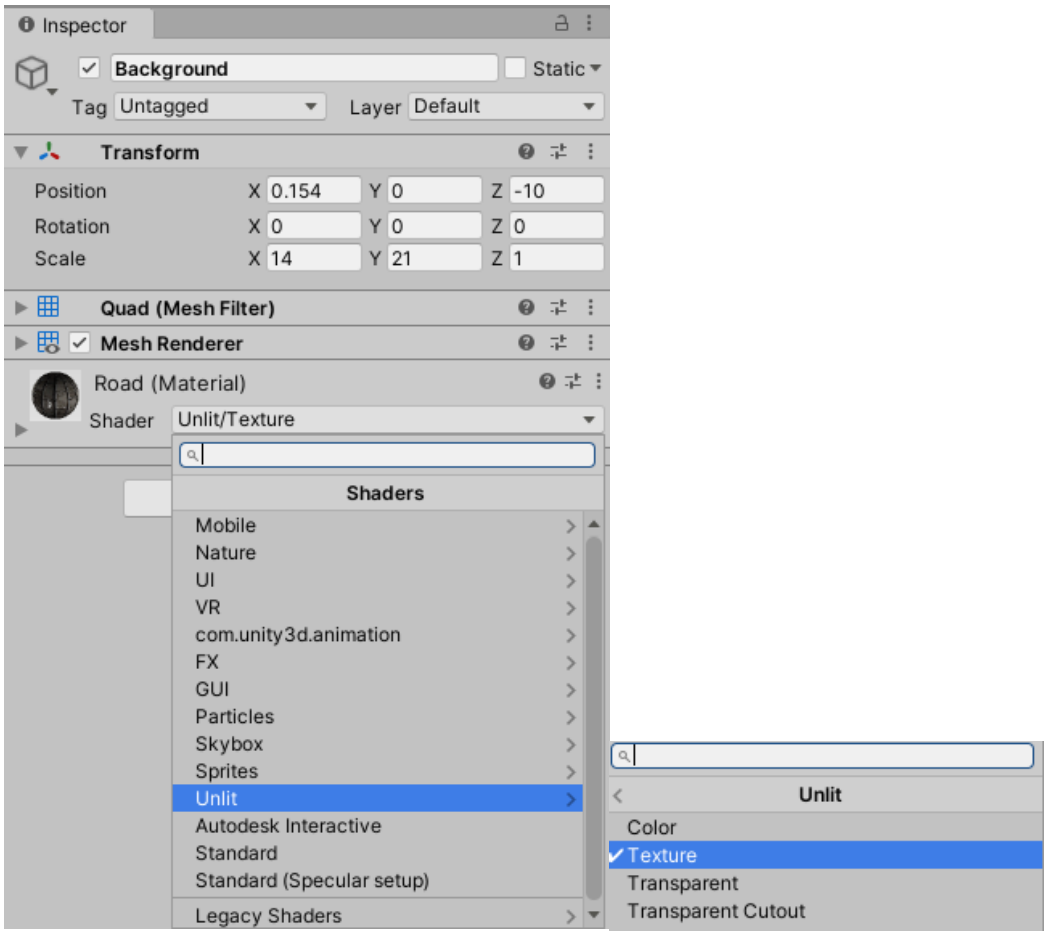
Obr. 114 Nastavenia *spritu road*.

Jednoduchým presunutím tejto textúry na *quad* v scéne ho aplikujeme. V okne *Inspector* môžeme sledovať zmeny, že sa táto textúra aplikovala na objekt ako materiál (Obr. 115). Úpravy nastavení môžeme realizovať v komponente *Mesh Renderer*.



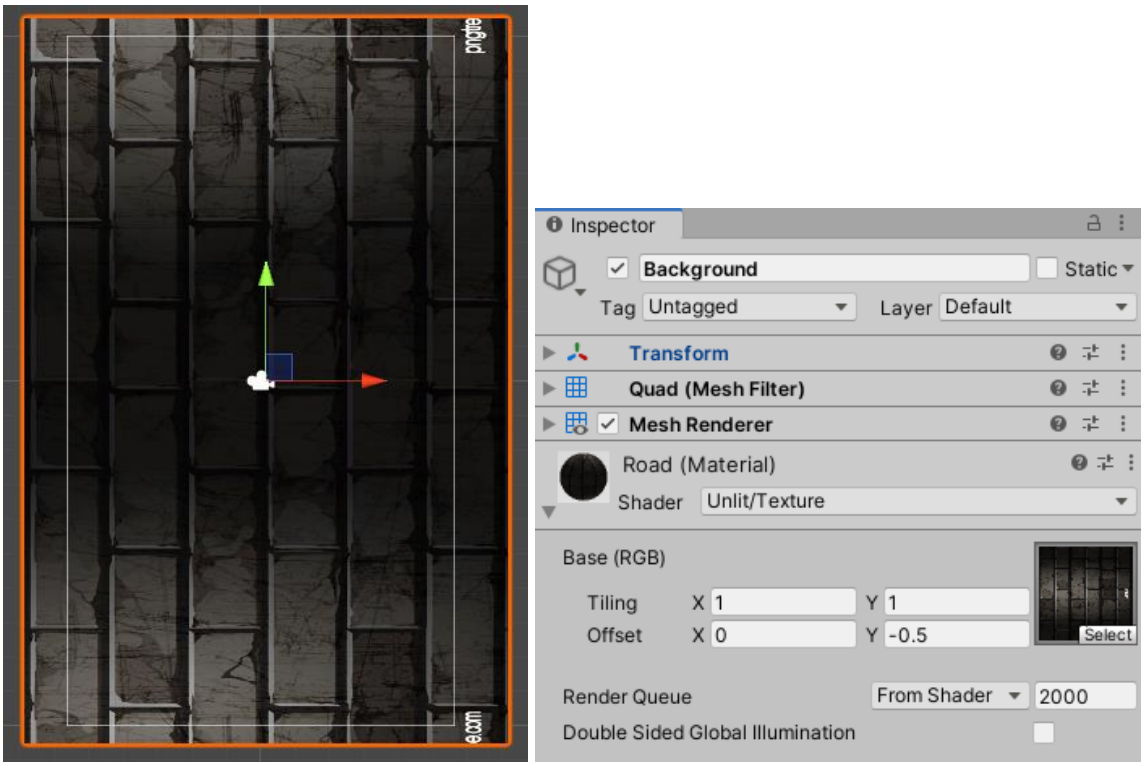
Obr. 115 Aplikovanie textúry na *quad*.

Vlastnosť *Shader* nastavíme na *Unlit/Texture*, čo je jeden zo základných *shaderov*, ktoré na textúru neaplikujú žiadne osvetlenie. Tým dosiahne také isté farebné pozadie, ako sme mali predtým (Obr. 116).



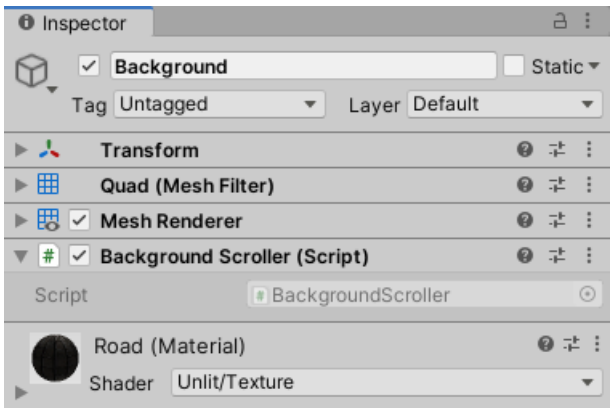
Obr. 116 Nastavenie vlastnosti *Shader*.

Pomocou vlastnosti *Offset* zabezpečíme pohyb pozadia. Experimentovaním s touto hodnotou v smere osi *y* si môžeme všimnúť, že pozadie sa pohybuje presne tak, ako si predstavujeme (Obr. 117).



Obr. 117 Zmena hodnoty *Offset* v smere osi *y* a jej vplyv na zmenu pozadia.

Túto zmenu zabezpečíme dynamicky, preto vytvárame nový skript s názvom *BackgroundScroller.cs*, ktorý pridáme ako komponent objektu *Background* (Obr. 118).



Obr. 118 Pridanie skriptu k objektu *Background*.

V skripte *BackgroundScroller.cs* deklarujeme nasledovné premenné:

```
[SerializeField] float backgroundScrollSpeed = 0.5f;
Material myMaterial;
Vector2 offSet;
```

Pomocou premennej *myMaterial*, ktorá je typu *Material* zabezpečíme pohyb pozadia zmenou vlastnosti *Offset*. Vo funkcii *Start()* zabezpečíme inicializáciu tejto premennej použitým materiálom pomocou vlastnosti *material* triedy *Renderer*. Premennú *offSet*, ktorá je typu štruktúry *Vector2*, nastavíme tak, že súradnicu *x* nechávame bezo zmeny a hodnotu *y* nastavujeme pomocou premennej *backgroundScrollSpeed*.

```
void Start()
{
    myMaterial = GetComponent<Renderer>().material;
    offSet = new Vector2(0f, backgroundScrollSpeed);
}
```

Vo funkcii *Update()* zabezpečíme posun, ktorý je nezávislý od frekvencie prehrávania snímok pomocou vlastnosti *mainTextureOffest* triedy *Material*.

```
void Update()
{
    myMaterial.mainTextureOffset += offSet * Time.deltaTime;
}
```

BackgroundScroller.cs

Assembly-CSharp

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class BackgroundScroller : MonoBehaviour
6  {
7      [SerializeField] float backgroundScrollSpeed = 0.5f;
8      Material myMaterial;
9      Vector2 offSet;
10
11     // Start is called before the first frame update
12     void Start()
13     {
14         myMaterial = GetComponent<Renderer>().material;
15         offSet = new Vector2(0f, backgroundScrollSpeed);
16     }
17
18     // Update is called once per frame
19     void Update()
20     {
21         myMaterial.mainTextureOffset += offSet * Time.deltaTime;
22     }
23 }
```

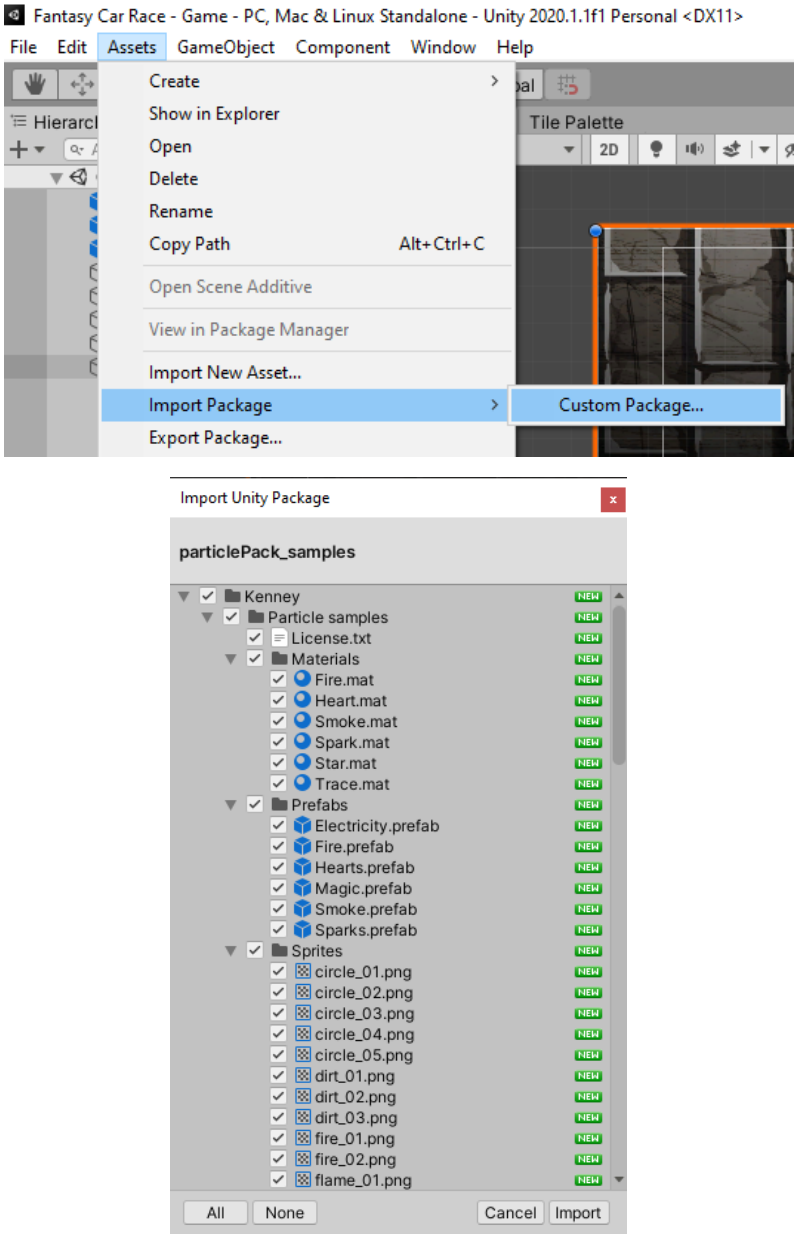
Obr. 119 Vizuál skriptu *BackgroundScroller.cs*.

Po uložení zmien otestujeme funkčnosť v editore Unity.

Pozn. autora: Ak nevidíte pozadie v hernom priestore, stačí mu nastaviť pozíciu v smere osi *z* na 0.

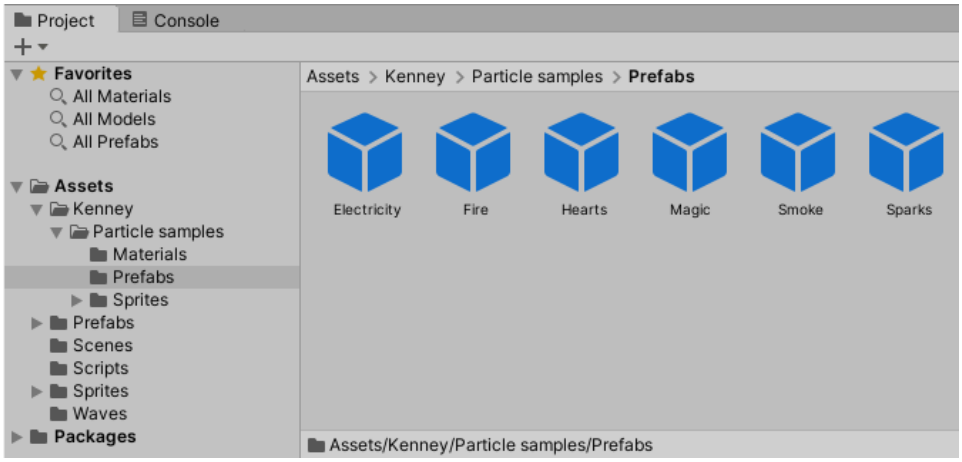
6.2 Pridanie efektov - *Particle systems*

Problematiku tvorby efektov vytvorením *Particle system* sme podrobnejšie rozobrali v kapitole 9 v skriptách autorky (Jurinová, 2022). V tomto prípade nebudeme vytvárať vlastné efekty, ale do projektu si importujeme hotový balíček efektov (*package*) „Particle Pack“. Tento je voľne dostupný na webe www.kenney.nl. *Package* je možné importovať pomocou možnosti Assets → *Import Package* (Obr. 120).



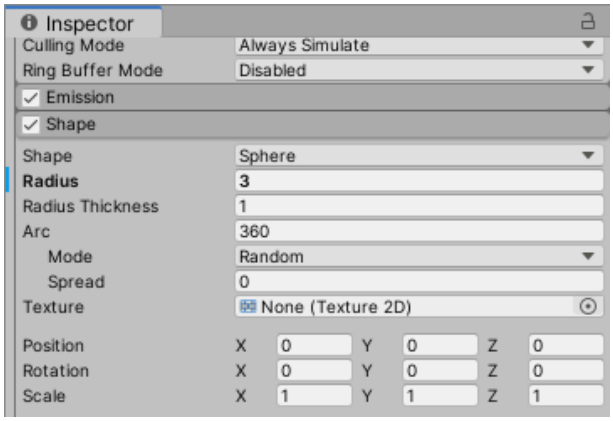
Obr. 120 Importovanie balíčka.

Po naimportovaní môžeme vidieť v okne *Project* v časti Assets nový priečinok s názvom *Kenney* s podpriečinkami, kde je možné nájsť jednotlivé materiály, *sprity*, ako aj hotové *prefaby* efektov.



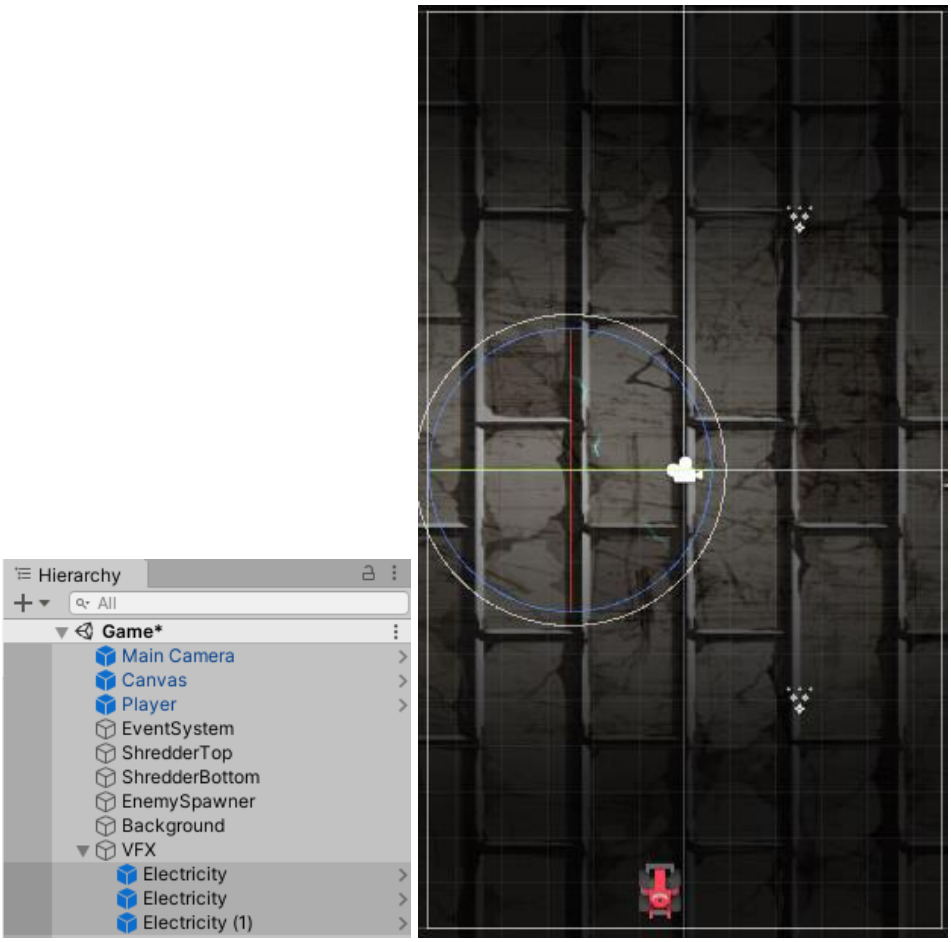
Obr. 121 Importovaný balíček *Kenney*.

Herný priestor ozvláštnime pridaním efektu *Electricity*. V *prefabe Electricity* v module *Shape* zmeníme nastavenie vlastnosti *Radius*, ktorá hovorí o veľkosti rozptylu na hodnotu 3 (Obr. 122).



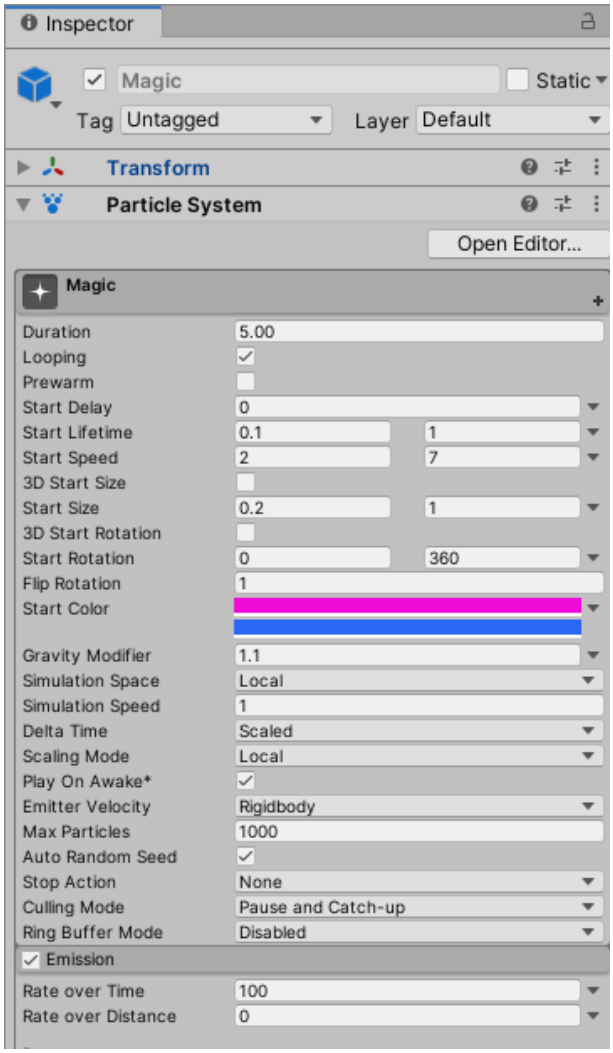
Obr. 122 Zmena vlastnosti *Radius prefabu Electricity*.

V okne *Hierarchy* vytvoríme prázdny *Game Object*, ktorý pomenujeme VFX = *Visual Effects*. Do neho umiestnime tri inštancie *prefabu Electricity* a vhodne ich umiestnime v hernom priestore (Obr. 123).



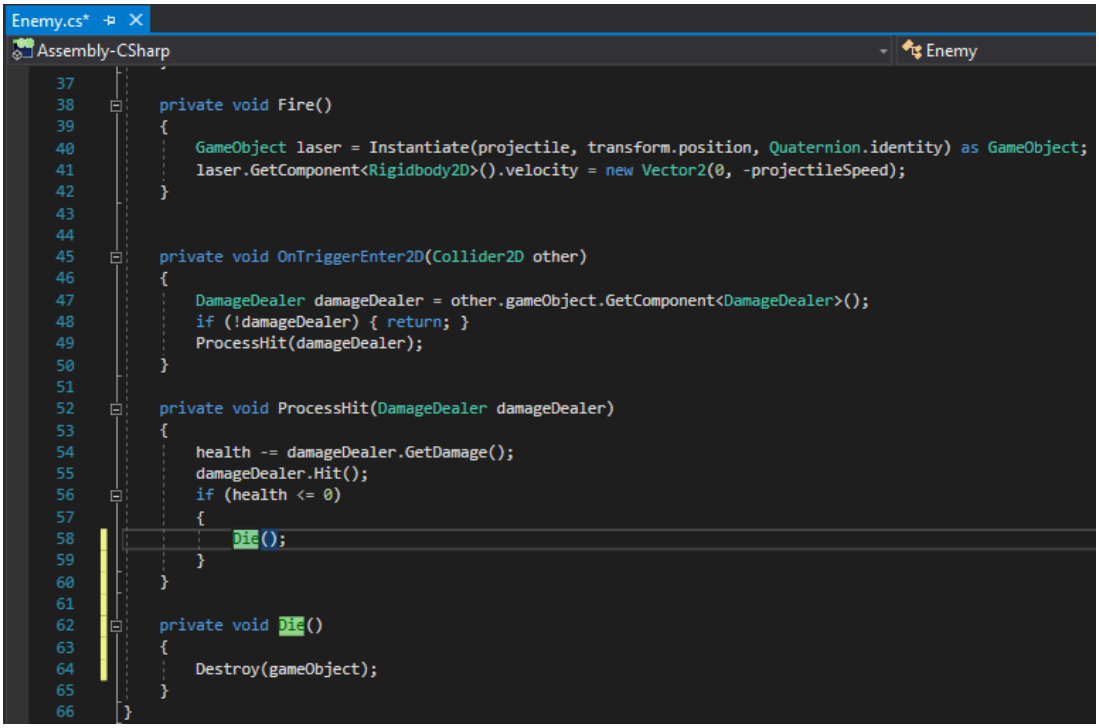
Obr. 123 Ilustrácia umiestnenia inštancií *Electricity*.

Efekt výbuchu (*prefab Magic*) využijeme v prípade keď nastane smrť nepriateľa. *Prefabu* upravujeme nastavenia vlastnosti *Start Size* na rozsah hodnôt 0,2 a 1, čím docielime zväčšenie emitujúcich častíc. Vlastnosť *Rate over Time* v module *Emission* upravíme na hodnotu 100, čím zvýšime počet emitovaných častíc (Obr. 124).



Obr. 124 Úprava nastavení *prefabu Magic*.

V skripte *Enemy.cs* v metóde *ProcessHit()* sa staráme o zničenie inštancie v prípade, ak hodnota premennej *health* klesne na 0. Namiesto príkazu `Destroy(gameObject);` zavoláme novú metódu *Die()*, v ktorej sa postaráme o zničenie inštancie, ako aj o prehratie vizuálneho efektu (Obr. 125).



Obr. 125 Vytvorenie metódy *Die()*.

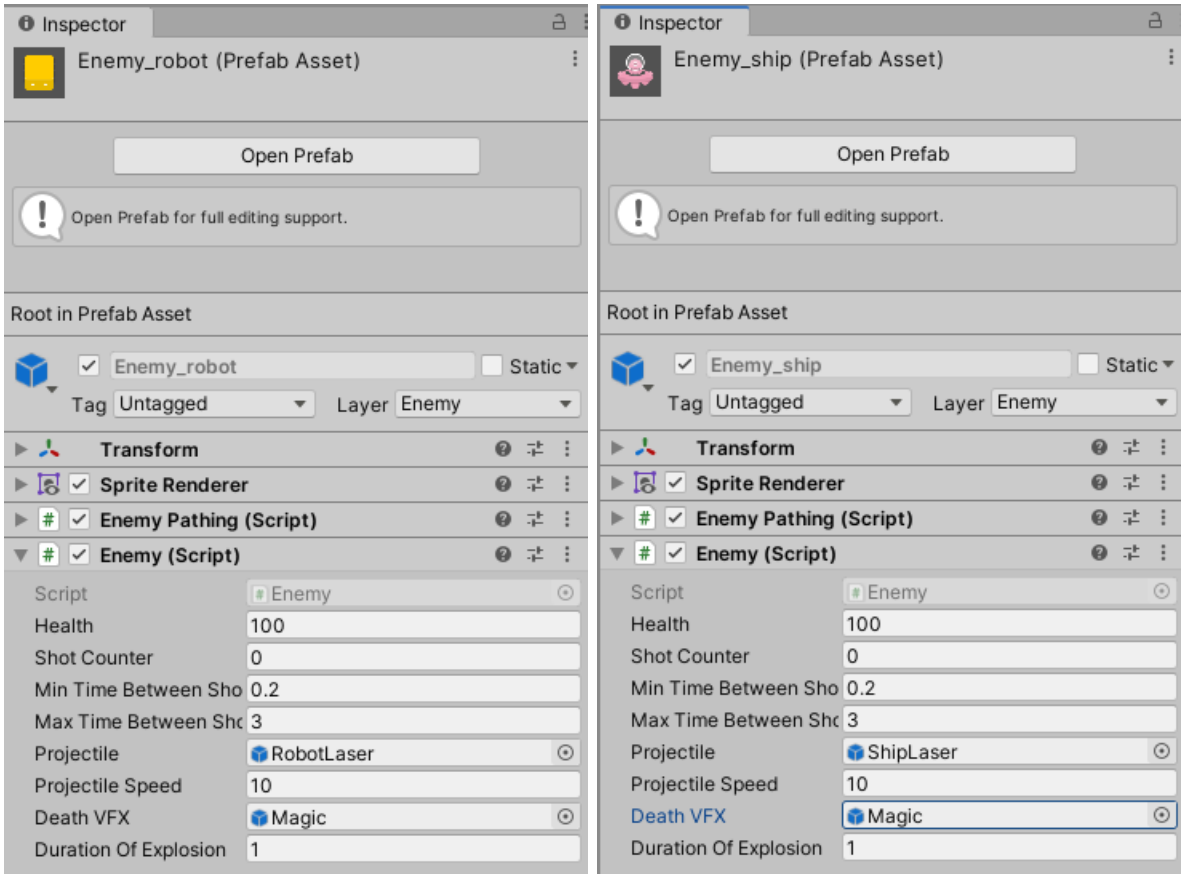
Particle system musíme vytvoriť ako samostatný objekt. Ak by bol súčasťou inštancie nepriateľa, nikdy by sme nevideli prehratie tohto efektu z dôvodu, že pri odstránení inštancie nepriateľa by nastalo aj odstránenie *particle* systému. Z tohto dôvodu deklarujeme dve premenné. Premenná *deathVFX* bude predstavovať samotný efekt a premenná *durationOfExplosion* čas trvania prehratia efektu.

```
[SerializeField] GameObject deathVFX;
[SerializeField] float durationOfExplosion = 1f;
```

V metóde *Die()* sa postaráme o zničenie inštancie nepriateľa. Následne vytvoríme nový objekt *explosion*, ktorý predstavuje efekt. Tento vznikne na pozícii, kde sa nachádza nepriateľ. Nakoniec sa postaráme aj o jeho zničenie v časovom intervale určenom premennou *durationOfExplosion*.

```
private void Die()
{
    Destroy(gameObject);
    GameObject explosion =
          Instantiate(deathVFX, transform.position, transform.rotation);
    Destroy(explosion, durationOfExplosion);
}
```

Uložíme zmeny, a po návrate do editora Unity nezabudneme inicializovať premennú *deathVFX* efektom *Magic* v oboch *prefaboch* reprezentujúcich nepriateľov (Obr. 125).



Obr. 126 Inicializácia premennej *deathVFX*.

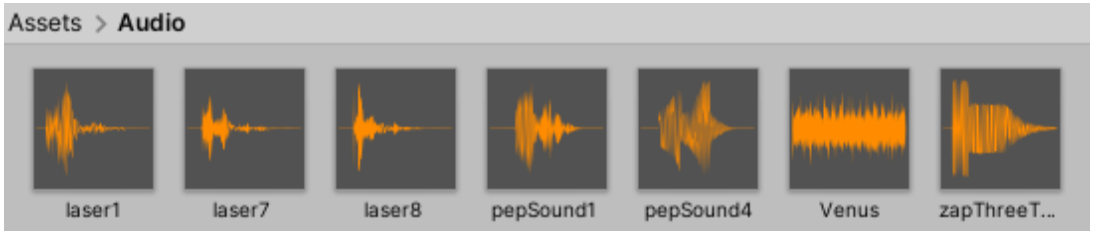
Otestovaním funkčnosti by sme mali vidieť prehratie efektu pri zničení nepriateľa hráčom.

6.3 Pridanie zvukových záznamov

Hra bez použitia zvukových stôp by nebola kompletnou. Zvuk na pozadí, alebo ako efekt pri akciách, je bežnou súčasťou hier. V prvom rade by sme sa mali zamyslieť, kde je žiaduce a nie príliš rušivé pridať zvukový záznam. Pridať zvukový záznam sa zdá byť logické pri týchto udalostiach:

- smrť postavy nepriateľa,
- smrť postavy hráča,
- streľba hráča,
- streľba nepriateľa.

V *Assets* vytvoríme nový priečinok *Audio* do ktorého presunieme používané audio záznamy, ktoré sme stiahli z webu https://www.kenney.nl/assets/digital-audio (Obr. 127).



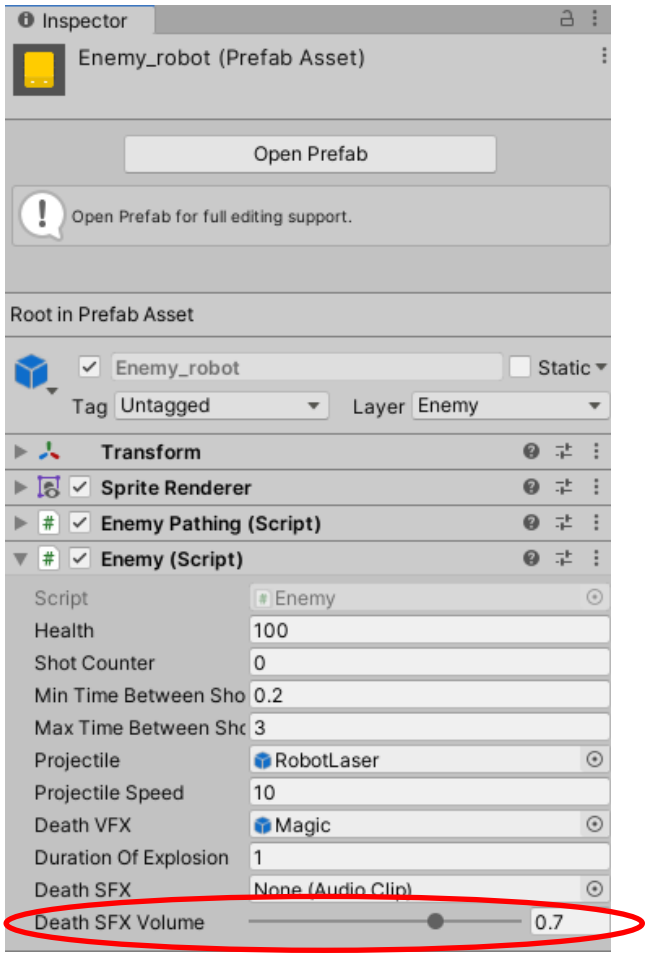
Obr. 127 Vloženie zvukových záznamov do projektu.

V skripte *Enemy.cs* vytvoríme premennú *deathSFX* typu *AudioClip*, ktorá bude reprezentovať použitý zvukový záznam. K prehratiu tohto záznamu príde v prípade zničenia nepriateľa. Premenná *deathSFXVolume* určuje hlasitosť prehratia tohto záznamu.

```
[SerializeField] AudioClip deathSFX;
[SerializeField] float deathSFXVolume = 0.7f;   //range 0-1
```

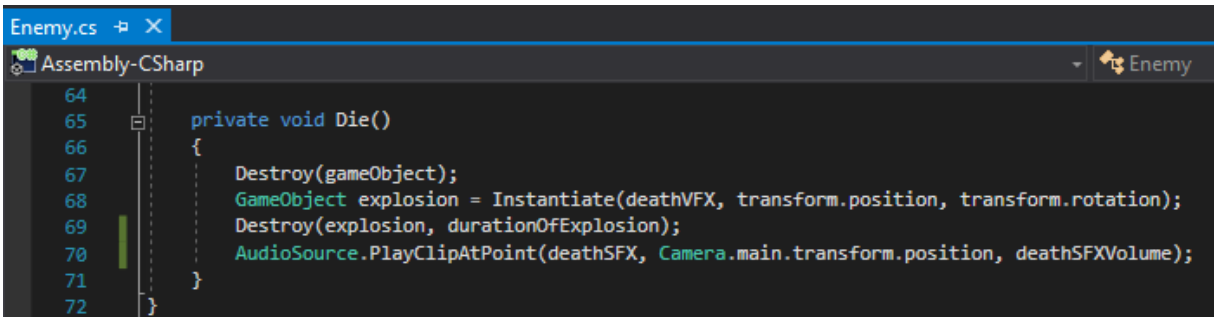
Ak chceme obmedziť nastavenie rozsahu hodnoty premennej, je možné použiť triedu *RangeAttribute*. Tým dosiahneme v okne *Inspector* možnosť meniť hodnotu premennej v danom intervale pomocou jazdca (*slider*) (Obr. 128).

```
[SerializeField] [Range(0, 1)] float deathSFXVolume = 0.7f;   //range 0-1
```



Obr. 128 *Slider* pre zmenu hodnoty premennej *deathSFXVolume*.

Na prehratie zvukového záznamu použijeme metódu *PlayClipAtPoint()* triedy AudioSource. Metóda očakáva tri parametre. Prvý parameter špecifikuje zvukový záznam, k čomu sme deklarovali premennú *deathSFX*. Druhý parameter hovorí o pozícii umiestnenia v hernom svete. Pre tento parameter využijeme objekt kamery, aby sme zabezpečili dostatočnú a rovnakú hlasitosť prehratia všetkých zvukových záznamov. Vieme, že jednotlivé objekty v scéne sa môžu nachádzať v rôznej hĺbke, čo by mohlo mať vplyv na hlasitosť prehratia jednotlivých zvukových záznamov. Tretí parameter určuje hlasitosť prehratia, ktorú nastavujeme pomocou premennej *deathSFXVolume*. Volanie metódy umiestnime do metódy *Die()* (Obr. 129).

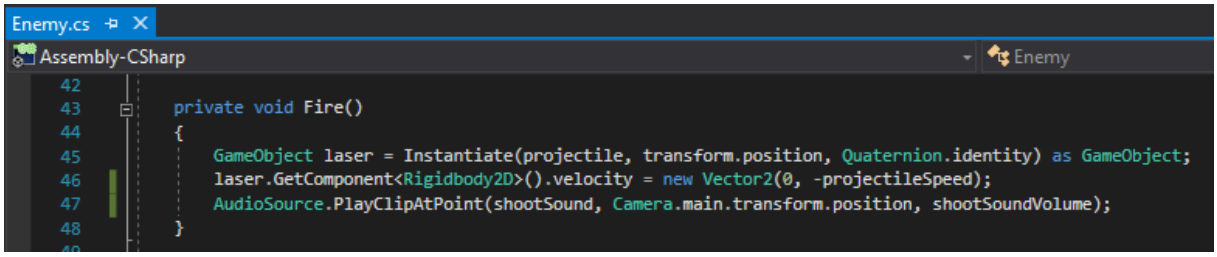


Obr. 129 Fragment kódu skriptu *Enemy.cs*.

Obdobné premenné potrebujeme aj pre prehratie zvukového záznamu v prípade streľby.

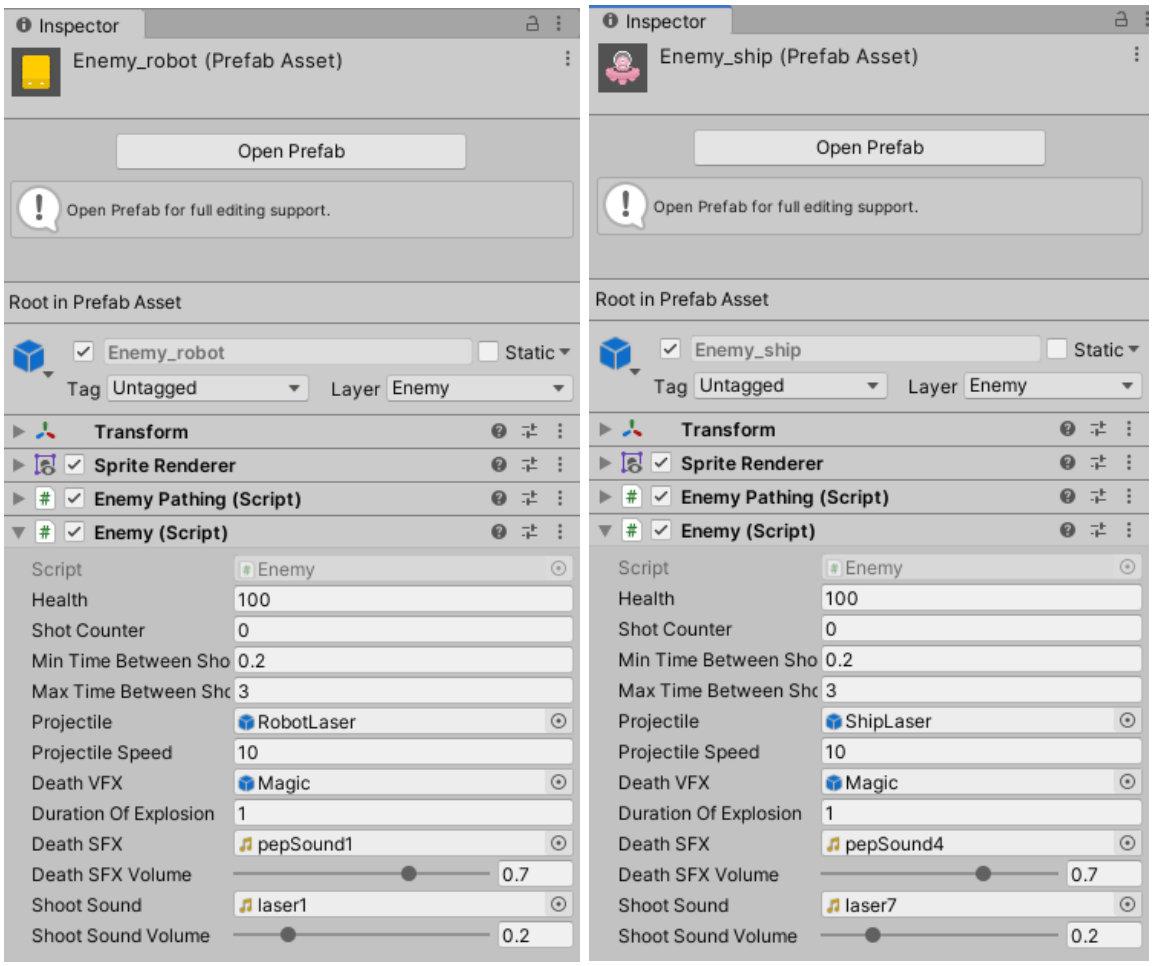
```
[SerializeField] AudioClip shootSound;
[SerializeField] [Range(0, 1)] float shootSoundVolume = 0.2f;   //range 0-1
```

Prehratie zvukového záznamu doplníme do metódy *Fire()* (Obr. 130).



Obr. 130 Fragment kódu skriptu *Enemy.cs*.

Uložíme zmeny a vrátime sa späť do editora Unity, kde inicializujeme premenné *deathSFX* a *shootSound* požadovanými zvukovými záznamami. My sme zvolili pre každý typ nepriateľa iný zvukový záznam (Obr. 131).

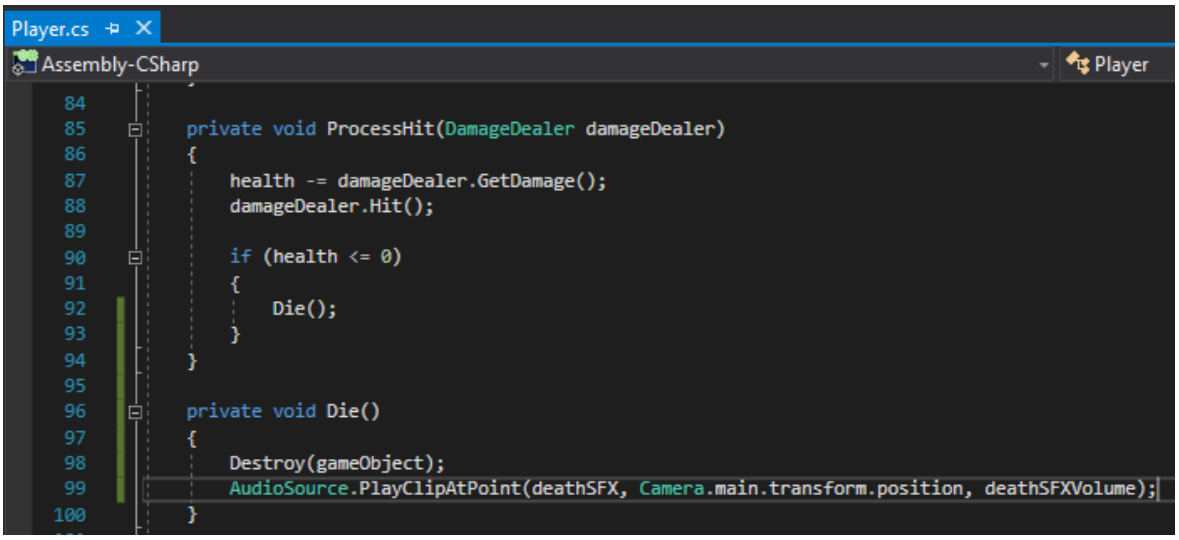


Obr. 131 Inicializácia premennej *deathSFX* a *shootSound* v prefaboch nepriateľov.

Obdobnú funkcionalitu zabezpečíme aj pre hráča v skripte *Player.cs*, kde ako prvé deklarujeme potrebné premenné.

```
[SerializeField] AudioClip deathSFX;
[SerializeField] [Range(0, 1)] float deathSFXVolume = 0.7f;    //range 0-1
```

V metóde *ProcessHit()*, v prípade zníženia premennej *health* na 0, nastáva priamo zničenie objektu. Toto zničenie objektu doplnené o prehratie zvukového záznamu zapúzdríme do metódy *Die()* (Obr. 132).

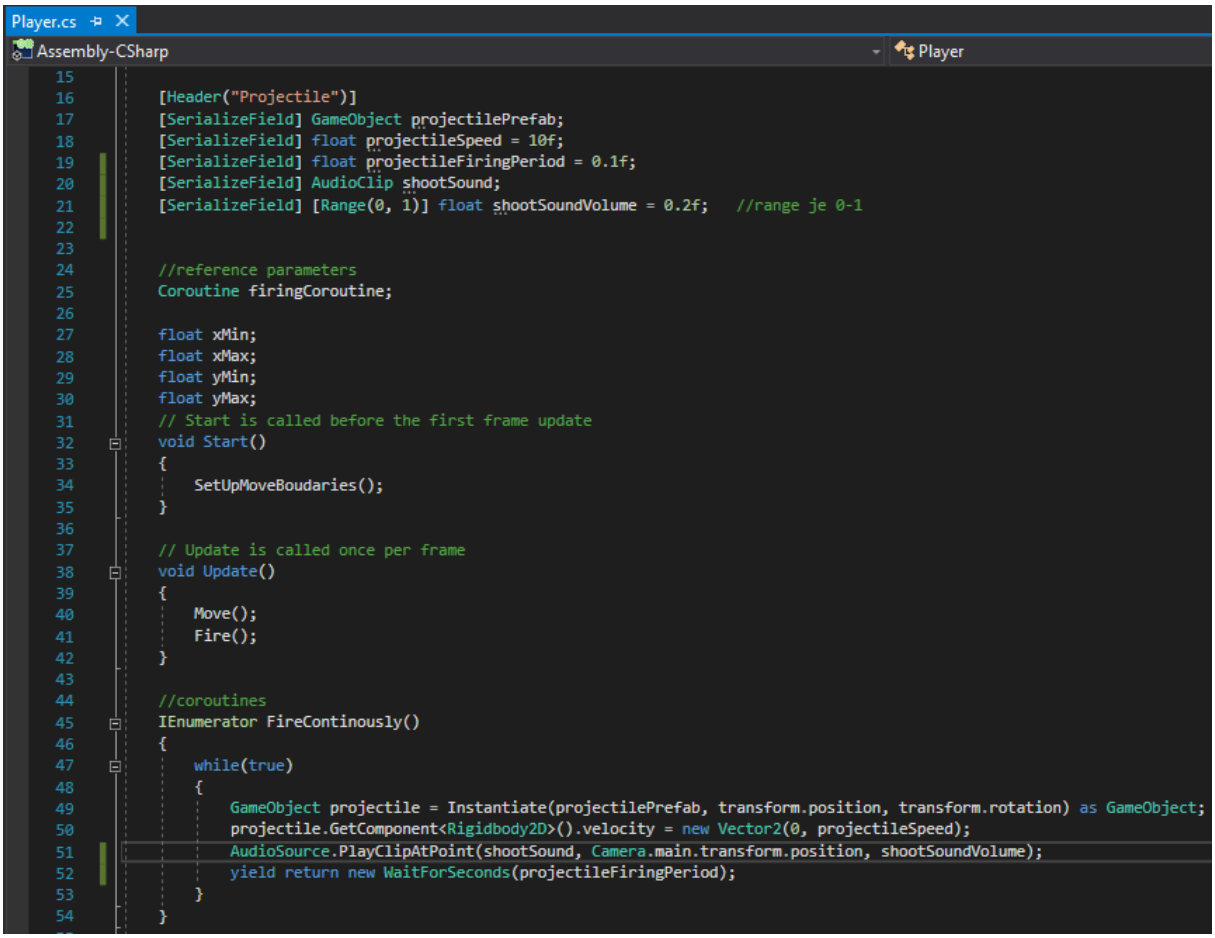


Obr. 132 Fragment kódu skriptu *Player.cs*.

V tomto skripte sa postaráme aj o pridanie zvukovej stopy pri streľbe, k čomu obdobne potrebujeme dve premenné:

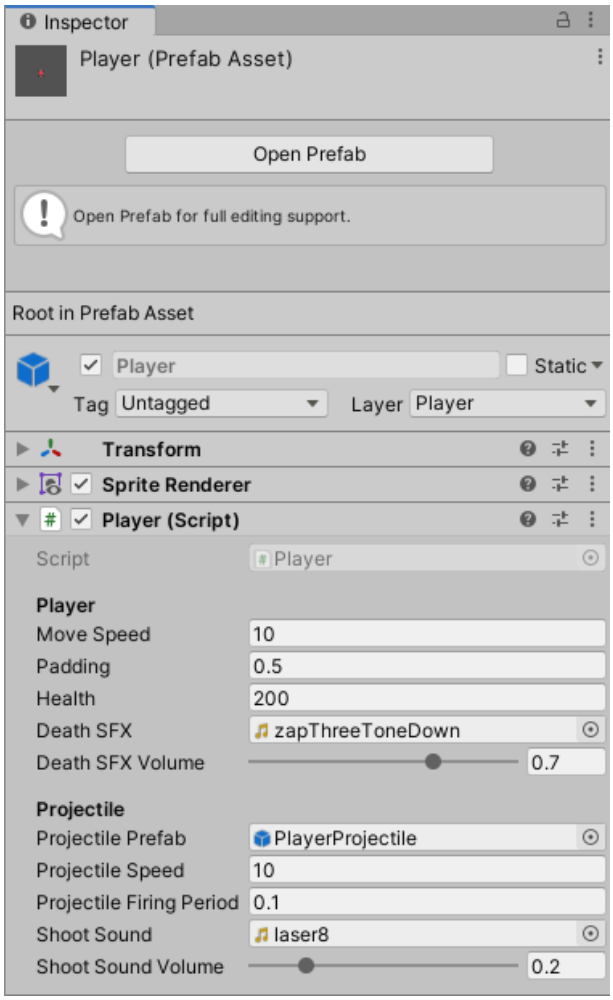
```
[SerializeField] AudioClip shootSound;
[SerializeField] [Range(0, 1)] float shootSoundVolume = 0.2f;   //range 0-1
```

Funkcionalitu vytvorenia, ako aj zmenu pozície strely, riešime v korutine *FireContinously()*, kde sa postaráme aj o volanie metódy *PlayClipAtPoint()* (Obr. 133).



Obr. 133 Fragment kódu skriptu *Player.cs*.

Uložíme zmeny a vrátime sa späť do editora Unity, kde inicializujeme premenné *deathSFX* a *shootSound* požadovanými zvukovými záznamami pre *prefab Player* (Obr. 134).



Obr. 134 Inicializácia premennej *deathSFX* a *shootSound* v *prefabe Player*.

6.4 Kontrolné otázky a úlohy

1. Objasnite spôsob, akým sme zabezpečili funkcionalitu skrolovania pozadia.
2. Ako je možné v Unity vytvoriť nejaký vizuálny efekt?
3. Definujte pojmy *Particle System* a *Particle Effect panel*.
4. Akými modulmi disponuje *Particle System*? A akými vlastnosťami jednotlivé moduly?

5. Vytvorte si vlastný efekt výbuchu a experimentujte s jednotlivými nastaveniami vlastností v moduloch *Particle System*.
6. Vytvorte vlastný efekt, ktorý sa prehrá napríklad pri smrti postavy hráča, výhre, či prehre hráča, pri postupe do ďalšieho levelu a pod.
7. Pomocou akého atribútu vieme obmedziť nastavenie hodnoty premennej v požadovanom vopred definovanom intervale?
8. Definujte pojmy *audio listener*, *audio source* a *audio clip*.
9. Pomocou akej metódy/metód je možné zabezpečiť prehratie zvukovej stopy? Definujte dané metódy.

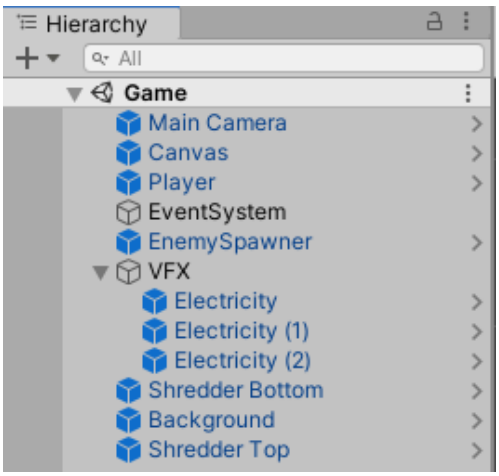
7 Finálne úpravy hry

V tejto kapitole sa postaráme o vytvorenie úvodnej a záverečnej obrazovky hry. Okrem toho budeme riešiť prechod medzi jednotlivými obrazovkami (scénami) pomocou skriptu *Level.cs*, ktorý bude obsahovať metódy pre načítanie jednotlivých scén, ako aj metódu pre ukončenie hry.

7.1 Prechod medzi scénami

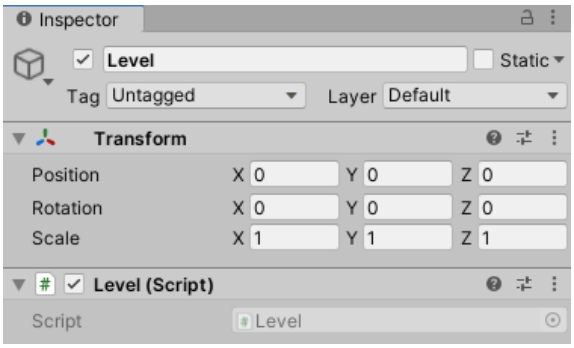
Scéna *StartMenu* bude obsahovať dve tlačidlá: *Start* a *Quit*. Scéna *GameOver* bude obsahovať tlačidlá: *Play Again* a *Main Menu*. V prípade, ak má študujúci dostatočné vedomosti, môže si túto funkcionalitu implementovať individuálne a skontrolovať svoj výsledok podľa časti uvedenej nižšie. Problematiku prechodu scénami sme riešili v skriptách autorky (Jurinová, 2022).

Pred vytvorením novej scény skontrolujeme, či máme zo všetkých objektov v scéne *Game* vytvorené *prefaby* (Obr. 135).



Obr. 135 Štruktúra scény *Game*.

V okne *Hierarchy* vytvoríme nový *Game Object*, ktorý pomenujeme *Level* a resetneme mu nastavenia v okne *Inspector* v komponente *Transform*. Objektu ako komponent pridáme skript s rovnakým názvom (Obr. 136).

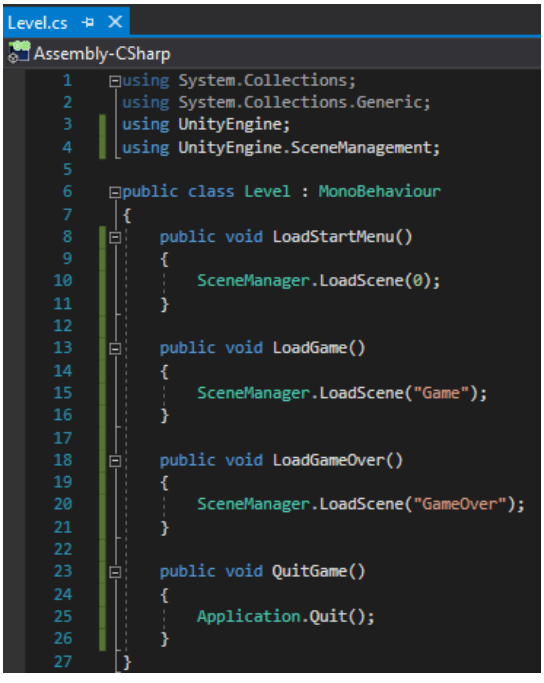


Obr. 136 Vytvorenie objektu *Level*.

V skripte *Level.cs* pridáme *namespace SceneManagement*, aby sme mohli využívať metódy triedy *SceneManager*.

```
using UnityEngine.SceneManagement;
```

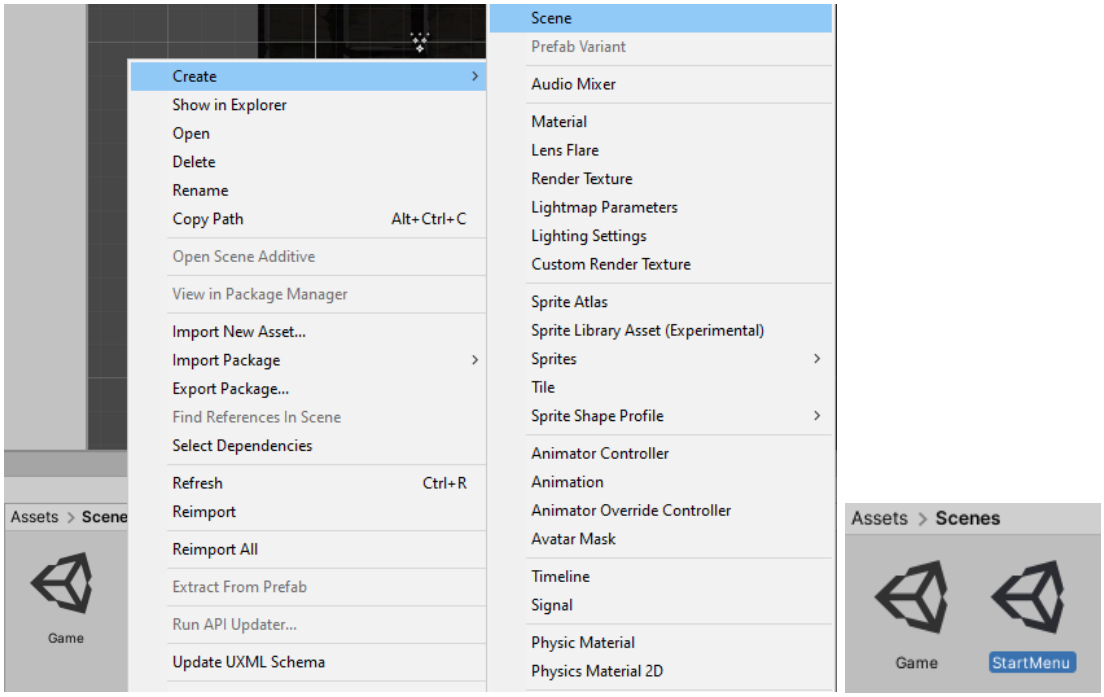
Definujeme tu štyri verejné metódy pomocou ktorých sa budeme starať o načítavanie jednotlivých scén a ukončenie aplikácie (Obr. 137). Pri načítavaní scén pomocou metódy *LoadScene()* triedy SceneManager môžeme ako parameter odovzdať buď index scény, alebo jej názov. Aplikáciu ukončujeme metódou *Quit()* triedy *Application*. Pozor, volanie tejto metódy je ignorované v prípade spustenia hry v editore. To znamená nečakajte ukončenie aplikácie. Z objektu *Level* urobíme *prefab*.



Obr. 137 Vizuál skriptu *Level.cs*.

7.2 Vytvorenie úvodnej a záverečnej scény

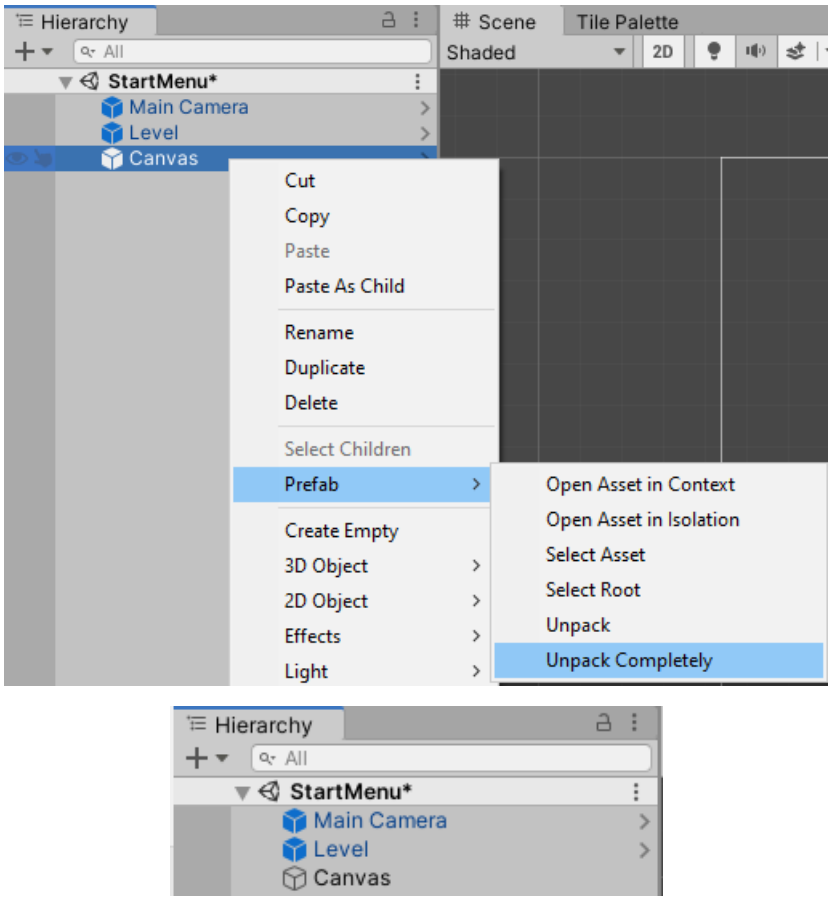
V tejto kapitole sa zameriame na vytvorenie scény *StartMenu* a *GameOver* v priečinku *Scenes* (Obr. 138).



Obr. 138 Vytvorenie scény *StartMenu*.

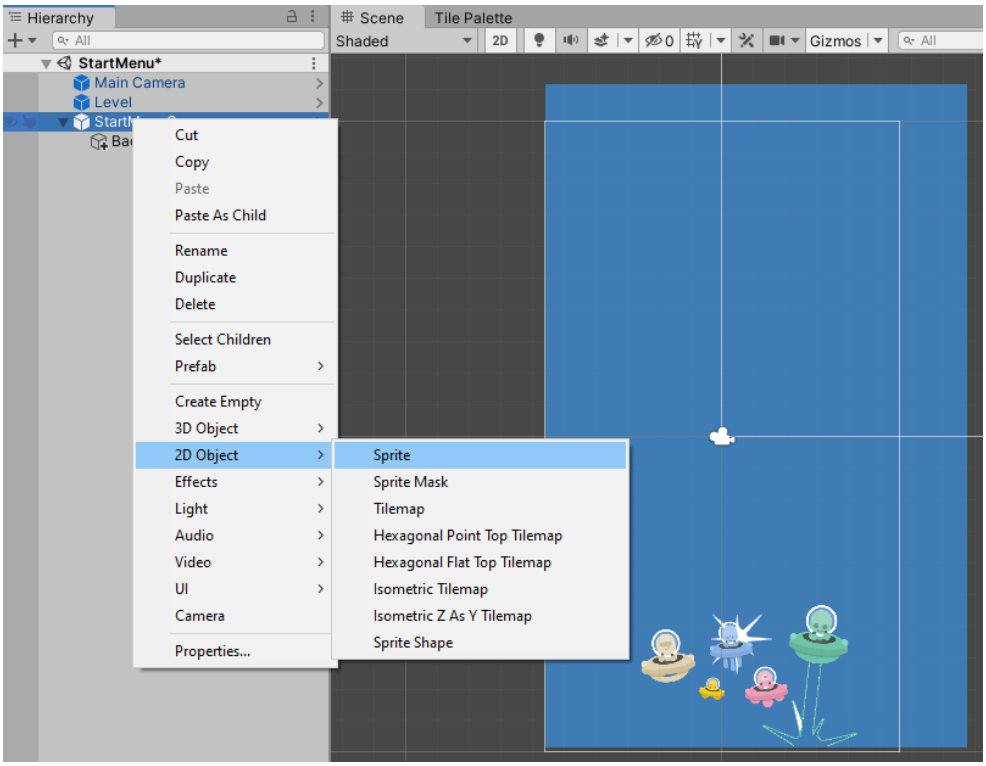
V scéne vymažeme aktuálnu kameru a nahradíme ju objektom *Main Camera* z priečinka *Prefabs*. Do scény tiež pridáme *prefab Level* a *Canvas*.

Prefab Canvas je určený pre jadro hry, ale v tomto momente ho môžeme využiť na to, aby sme nemuseli nanovo nastavovať základné nastavenia. Ak nechceme, aby sa zmeny implementované v tejto scéne aplikovali na *prefab*, ktorý slúži pre jadro hry, tak využitím kontextovej ponuky a možnosti *Prefab → Unpack Completely* z *prefabu* urobíme opäť len obyčajný objekt (Obr. 139).

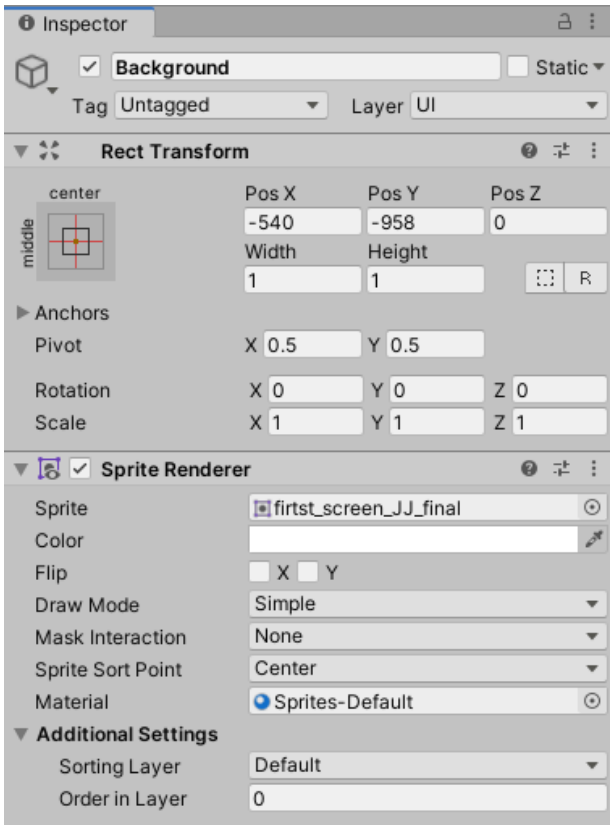


Obr. 139 Zrušenie *prefabu*.

Premenujeme ho na *StartMenuCanvas* a po jeho presunutí do priečinku *Prefabs* budeme mať separátny *prefab* pre túto scénu. Ako potomka pridáme 2D objekt *Sprite*, ktorý pomenujeme *Background* a vlastnosť *Sprite* komponentu *Sprite Render* nastavíme na vhodnú vizualizáciu, ktorú sme si vopred pripravili (Obr. 140). Vlastnosť *Pixel Per Unit* použitej textúry sme zmenili na 30. V hernom priestore sme mu vhodne upravili pozíciu. Vlastnosti objektu ilustruje obrázok 141.

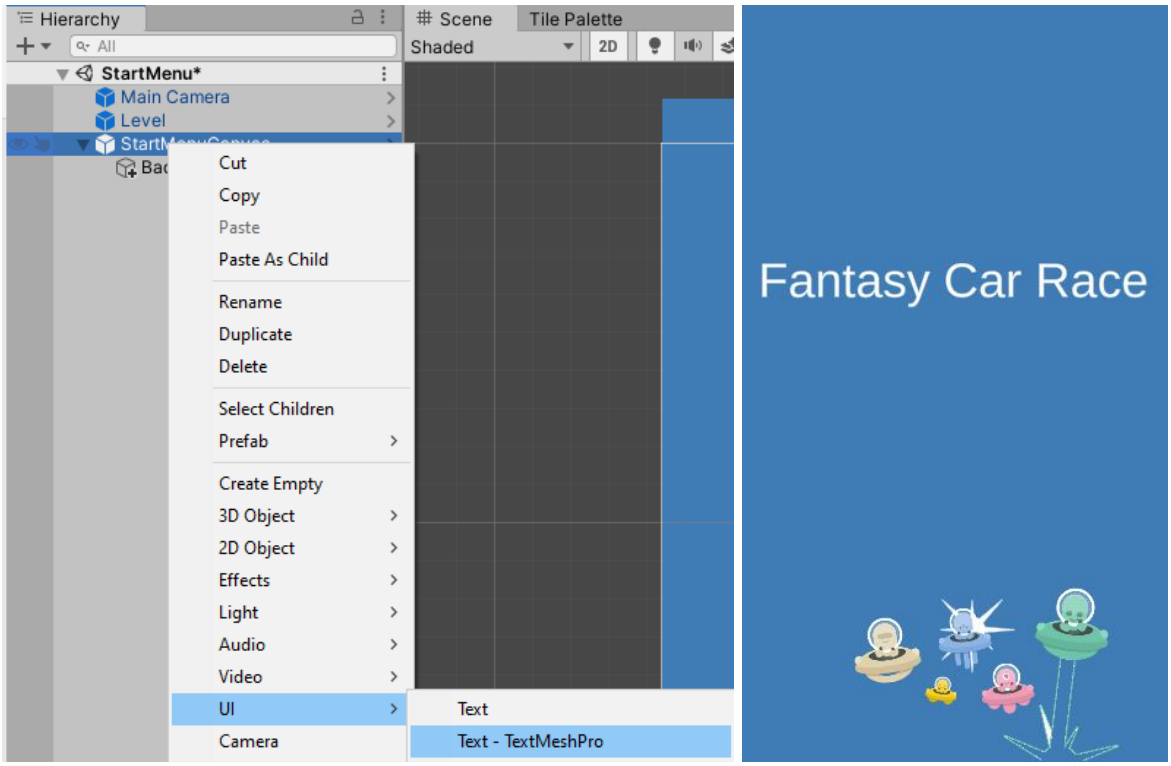


Obr. 140 Vytvorenie pozadia.



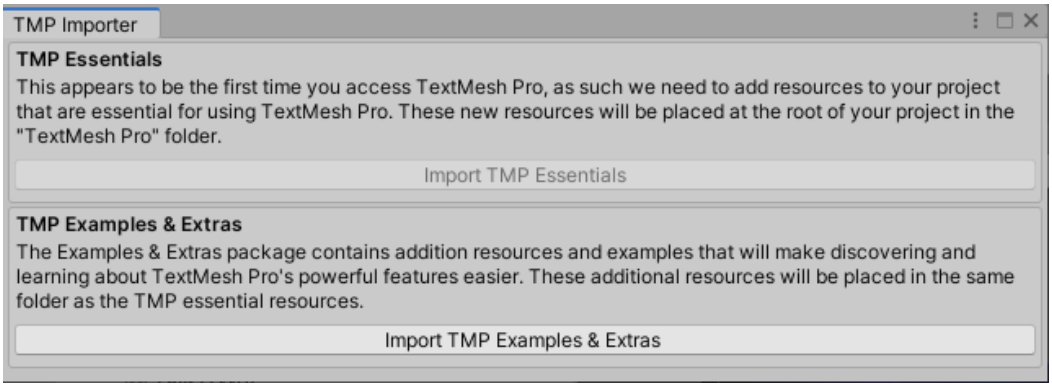
Obr. 141 Nastavenie vlastností objektu *Background*.

Ako ďalšieho potomka objektu *StartMenuCanvas* pridáme UI element *TextMeshPro* (Obr. 142). Pomenujeme ho ako *HeadingText*. Do poľa *Text Input* vložíme názov našej hry „*Fantasy Car Race*“. Veľkosť textu (*font size*) zmeníme na 120 a zmeníme veľkosť objektu tak, aby bol text celý viditeľný v jednom riadku. Vhodne ho umiestnime v hernom svete.



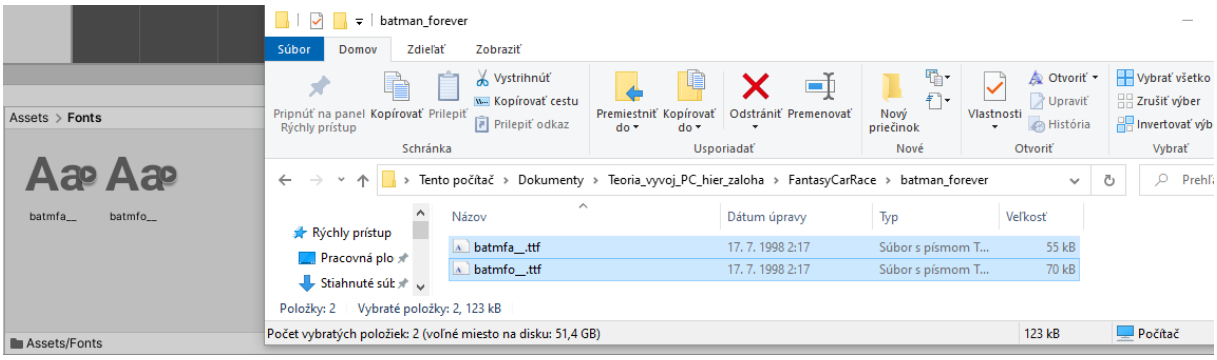
Obr. 142 Pridanie objektu *HeadingText* a jeho vizuál.

Pozn. autora: Ak je nutné pri pridaní objektu *TextMeshPro* importovať súčasti, (Obr. 143) importujte ich a okno zatvorte.



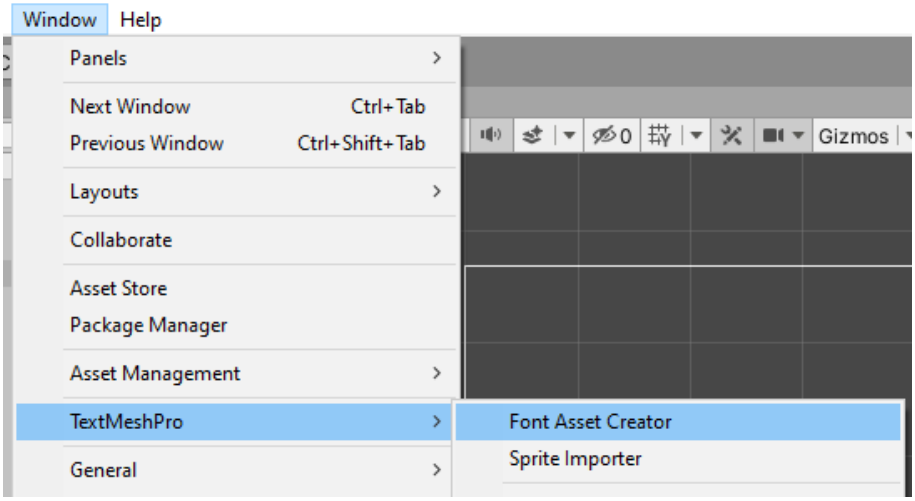
Obr. 143 Importovanie súčasti pre objekt typu *TextMeshPro*.

V prípade použitia vlastného preferovaného fontu tieto môžeme voľne sťahovať z webu www.datafont.com. My sme v tomto prípade využili font *batman forever*. V *Assets* si vytvoríme nový priečinok s názvom *Fonts* do ktorého presunieme požadovaný zdrojový súbor pre font (Obr. 144).



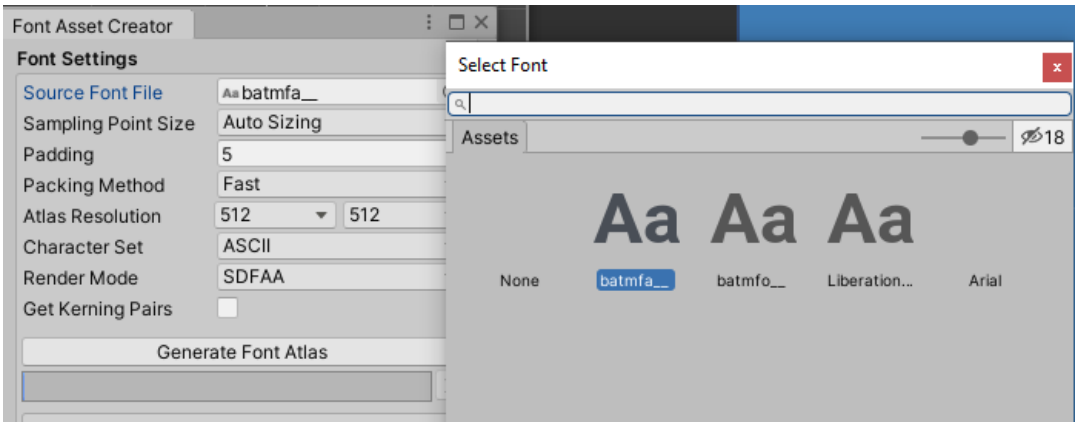
Obr. 144 Pridanie fontu do projektu.

Aby sme mohli daný font použiť, musíme ho ešte vytvoriť pomocou ponuky *Window* → *TextMeshPro* → *Font Asset Creator* (Obr. 145).



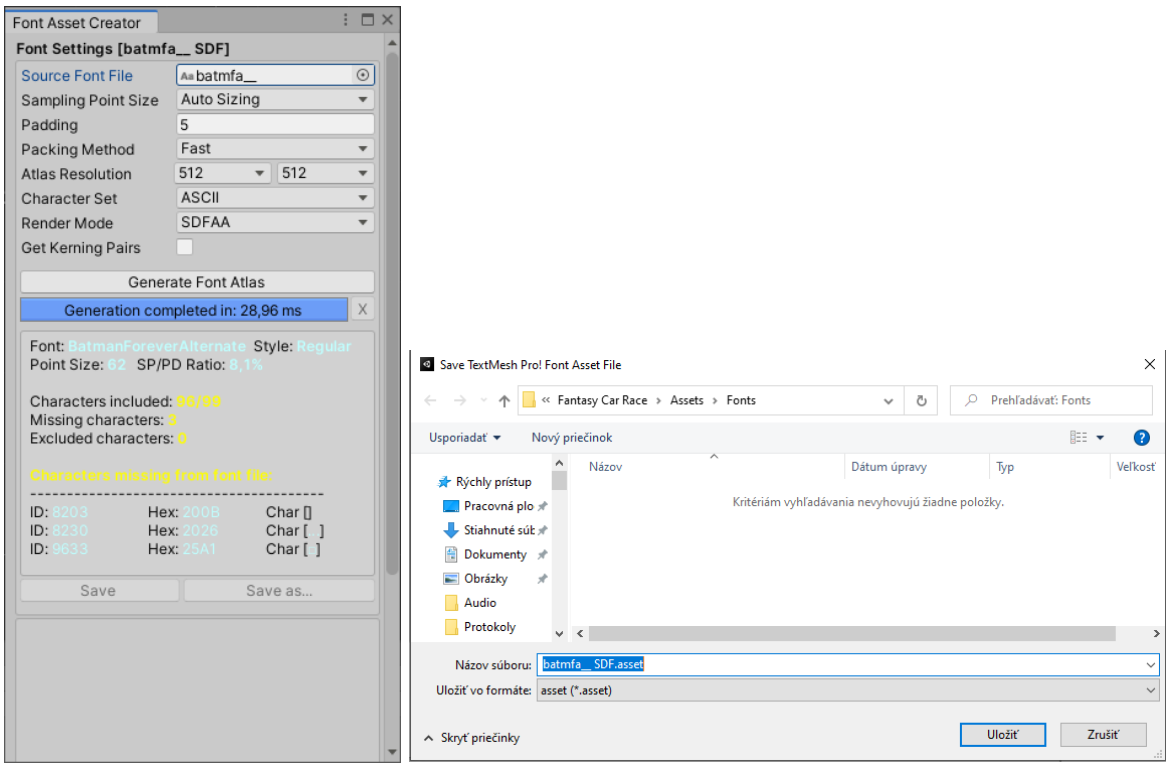
Obr. 145 Vytvorenie nového fontu.

Ako prvý krok v možnosti *Source Font File* zvolíme font, ktorý chceme použiť (Obr. 146).



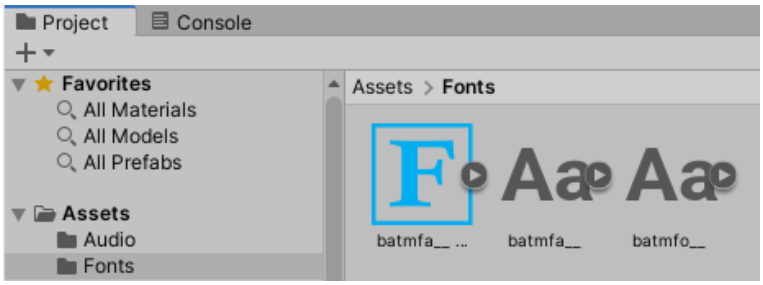
Obr. 146 Voľba fontu.

Následne ho necháme vygenerovať pomocou možnosti *Generate Font Atlas* a uložíme do pripraveného priečinku s názvom *Fonts* (Obr. 147).



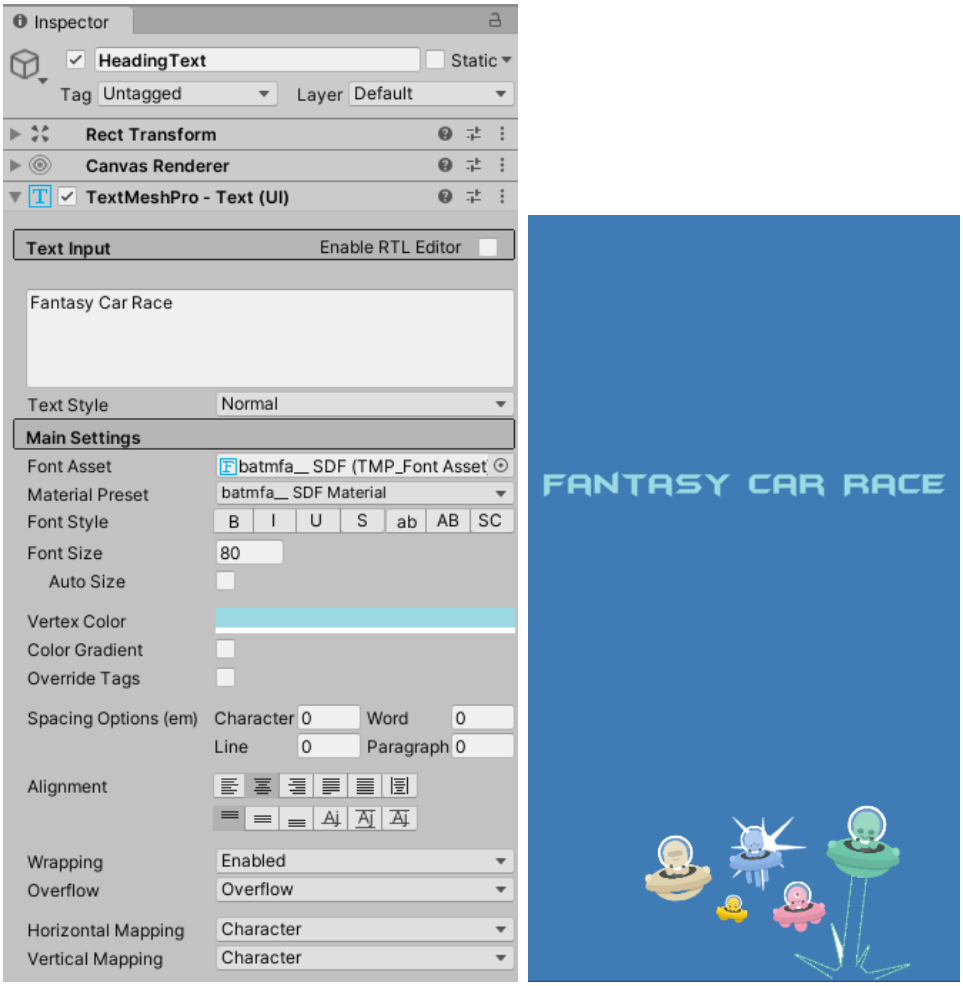
Obr. 147 Vygenerovanie a uloženie fontu.

Výsledkom je nový font pridaný do priečinku *Fonts* v našom projekte, ktorý už môžeme používať (Obr. 148).



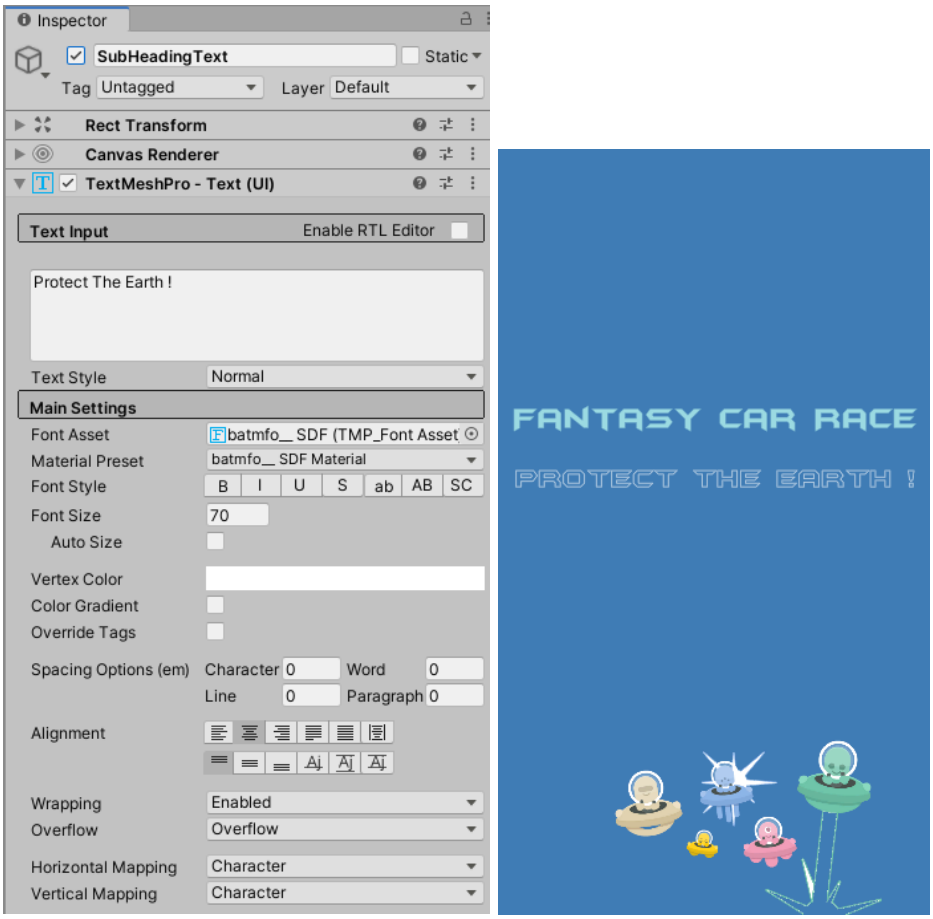
Obr. 148 Výsledok po pridaní nového fontu.

Obdobný spôsob použijeme aj pre druhý font. Zmena pre náš *HeadingText* na novo pridaný font sa prejaví vizuálne tak, ako ilustruje obrázok 149. Bolo nutné zmenšiť veľkosť textu na 80, text sme vycentrovali a zmenili mu farbu pomocou vlastnosti *Vertex Color*.



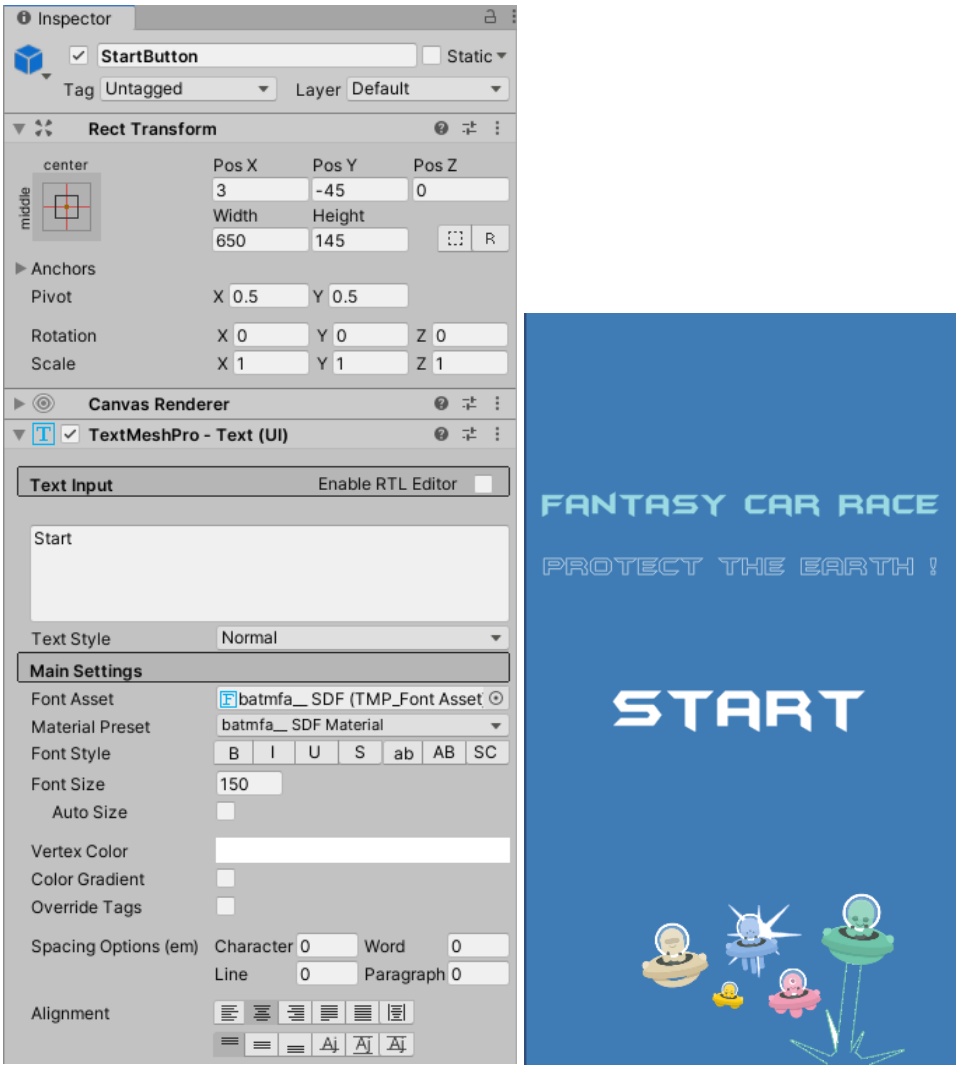
Obr. 149 Nastavenie vlastností objektu *HeadingText* a jeho vizuál.

Následne si vytvoríme jeho kópiu (Ctrl + D) pre účely podnázvu, kde zmeníme text, font, ako aj veľkosť (Obr. 150).



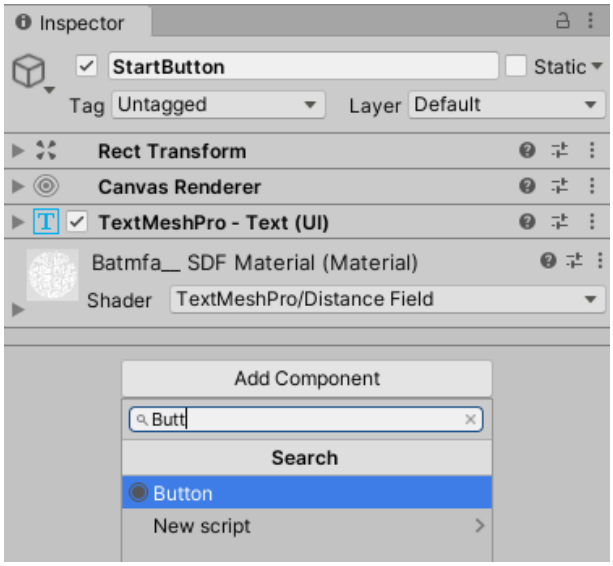
Obr. 150 Nastavenie vlastností objektu *SubHeadingText* a jeho vizuál.

Teraz sa postaráme o vytvorenie tlačidiel. Jednou z možností je vložiť do scény objekt z kategórie *UI*: *Button* alebo *Button – TextMeshPro*. My však nechceme okolo tlačidla žiadny vizuál pripomínajúci tlačidlo, želáme si tlačidlo v podobe textu a preto postupujeme takým spôsobom, že vložíme objekt typu *TextMeshPro* z kategórie *UI*. Pomenujeme ho *StartButton*. Objektu pridáme text „Start“ nastavíme požadovaný font, veľkosť, zarovnanie a jeho vhodné umiestnenie v scéne (Obr. 151).



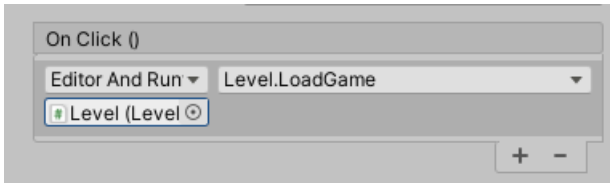
Obr. 151 Nastavenie vlastností objektu *StartButton* a jeho vizuál.

Objektu pridáme komponent *Button* (Obr. 152).



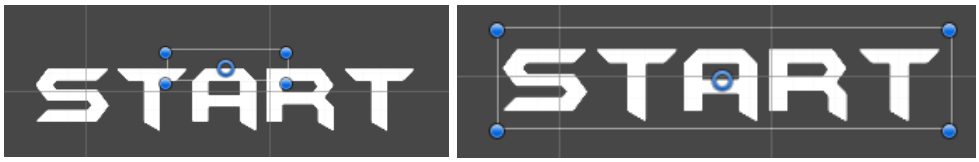
Obr. 152 Pridanie komponentu *Button* objektu *StartButton*.

Správnu funkcionalitu zabezpečíme, ak v okne *Inspector* tlačidla *StartButton* v možnosti *On Click*() pridáme objekt *Level* a využijeme vytvorenú metódu *LoadGame*(), ktorá zabezpečí načítanie scény predstavujúcej jadro hry (Obr. 153).



Obr. 153 Pridanie objektu *Level* a vyvolanie metódy *LoadGame*().

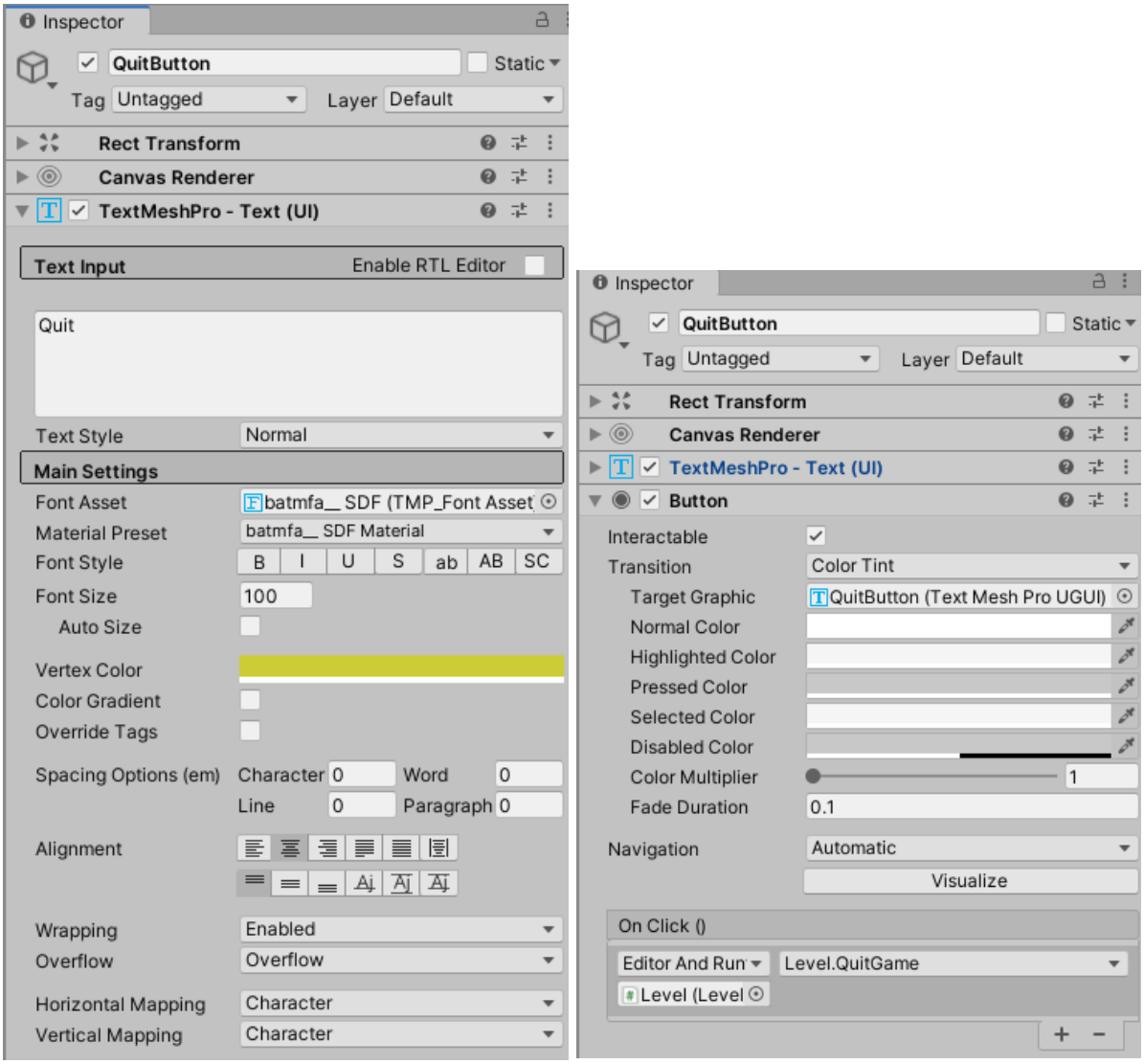
Dôležitým krokom je skontrolovanie veľkosti aktívnej plochy tlačidla (*hit area*), ktorá je reprezentovaná obdĺžnikom ako ilustruje obrázok 154. Je potrebné túto plochu zväčšiť napríklad tak, ako ilustruje obrázok.



Obr. 154 Zmena aktívnej plochy tlačidla.

Vytvoríme kópiu tohto tlačidla, premenujeme ho na *QuitButton*, zmeníme text, veľkosť textu aj farbu. Objekt vhodne umiestnime do herného sveta. My sme ho

umiestnili pod tlačidlo *StartButton*. Jeho funkcionalitu zabezpečíme obdobne, avšak v tomto prípade použijeme metódu *QuitGame()* (Obr. 155).

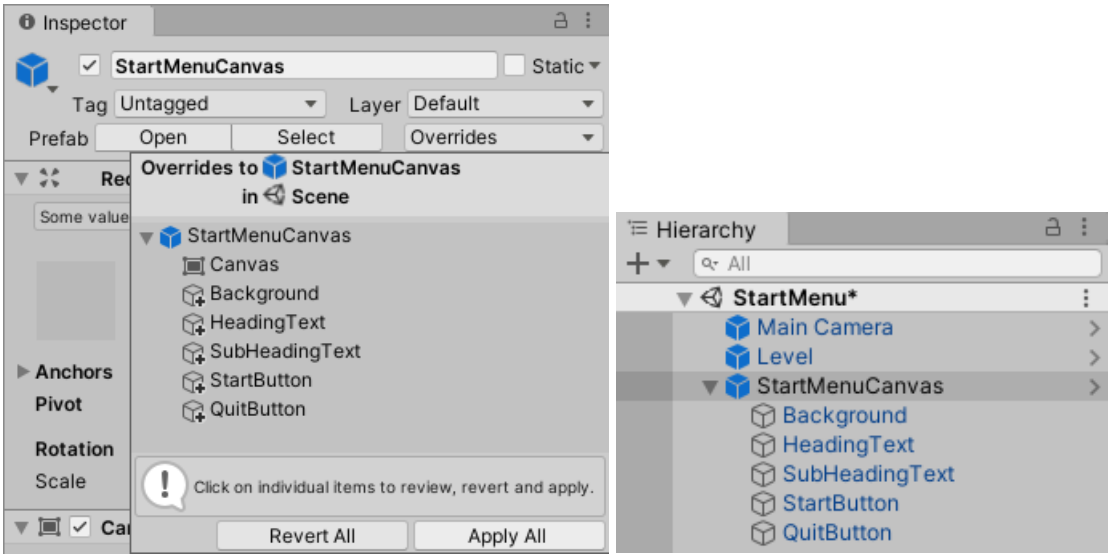


Obr. 155 Nastavenie vlastností objektu *QuitButton*.

Tiež nezabudneme upraviť aktívnu plochu tlačidla (Obr. 156).

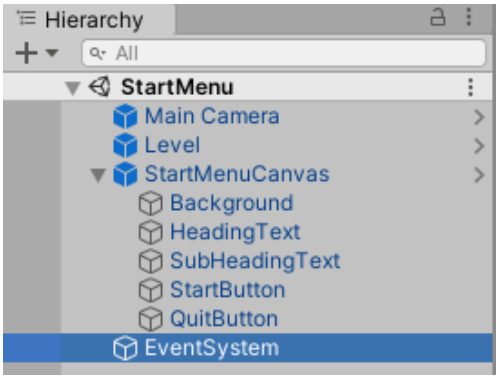
Obr. 156 Úprava aktívnej plochy tlačidla.

Všetky zmeny nezabudneme aplikovať na *prefab StartMenuCanvas* (Obr. 157).



Obr. 157 Aplikovanie zmien na *prefab StartMenuCanvas*.

Do scény pridáme kopírovaním zo scény *Game* objekt *EventSystem* (Obr. 158), ktorý je zodpovedný za spracovanie a obsluhu udalostí v scéne. Bez neho by toto nefungovalo (Unity, 2021f).

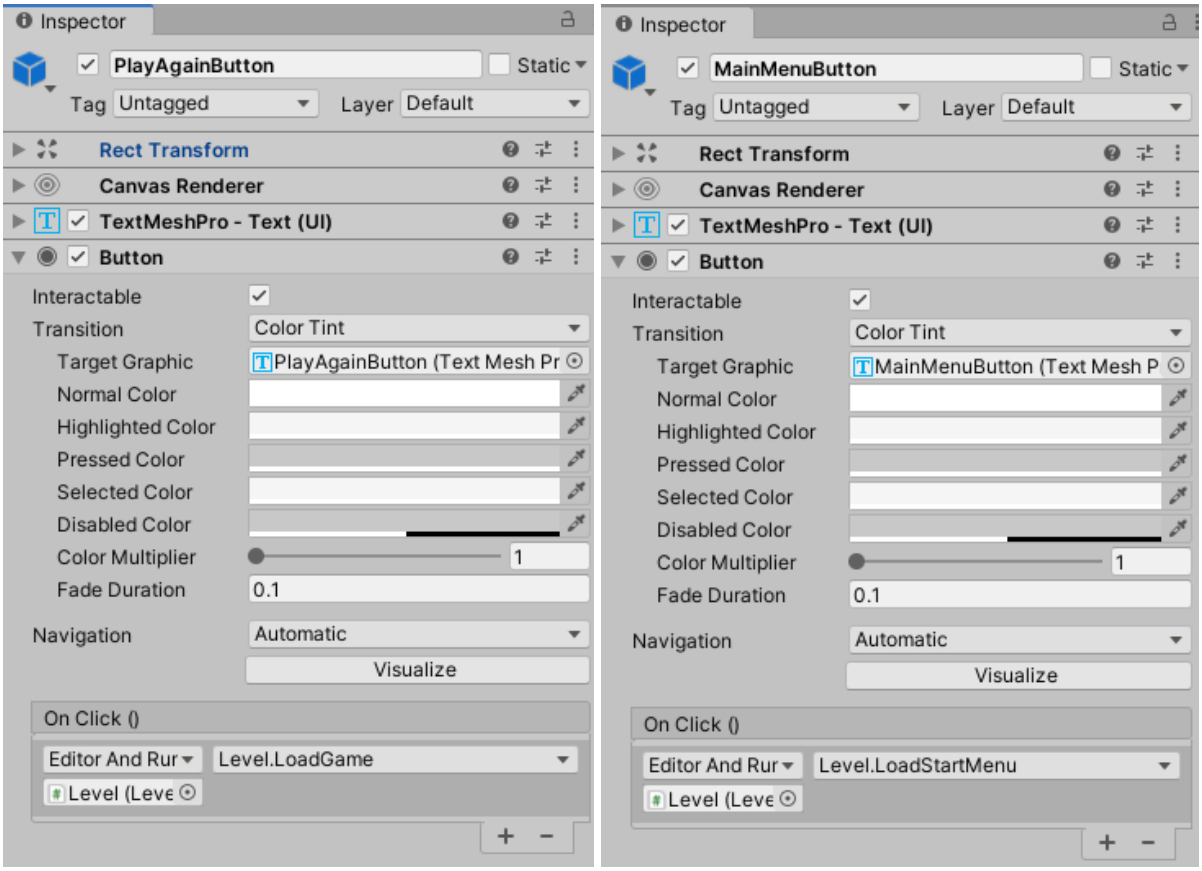


Obr. 158 Pridanie objektu *EventSystem*.

Následne túto scénu duplikujeme a vytvoríme novú s názvom *GameOver*. V scéne upravíme príslušné náležitosti týkajúce sa vizuálnej stránky, ako aj funkčnosti podľa obrázku 159. Postupujeme obdobne ako sme robili pri scéne *StartMenu*.

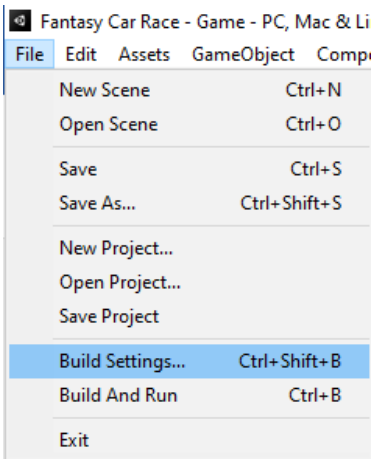
Obr. 159 Štruktúra a vizuálna reprezentácia scény *GameOver*.

Funkcionalitu tlačidla *PlayAgainButton* zabezpečíme pomocou objektu *Level* a metódy *LoadGame*(). Pomocou tlačidla *MenuMainButton* zavoláme metódu *LoadStartMenu*() (Obr. 160).

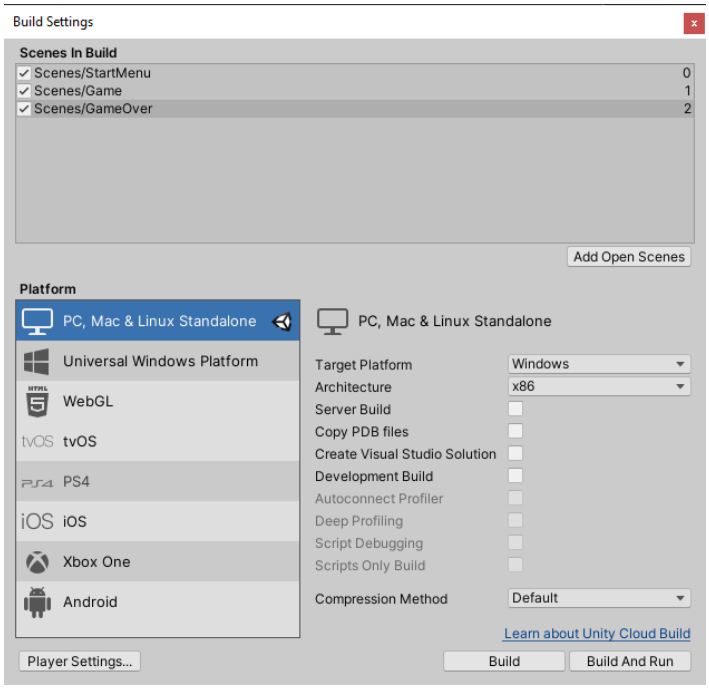


Obr. 160 Nastavenie funkcionality pre tlačidlá *PlayAgainButton* a *MainMenuButton*.

Aby celé načítavanie scén fungovalo korektne, je potrebné v Unity v možnosti *File → Build Settings* (Obr. 161) pridať všetky scény s ktorými pracujeme, v takom poradí ako potrebujeme, bez nutnosti exportovania výslednej hry (*Build And Run*). Pridávame ich jednoducho presunutím požadovanej scény z *Assets* (Obr. 162). Po pridaní všetkých scén stačí okno zatvoriť.



Obr. 161 Ako sprístupniť okno pre pridanie scén.

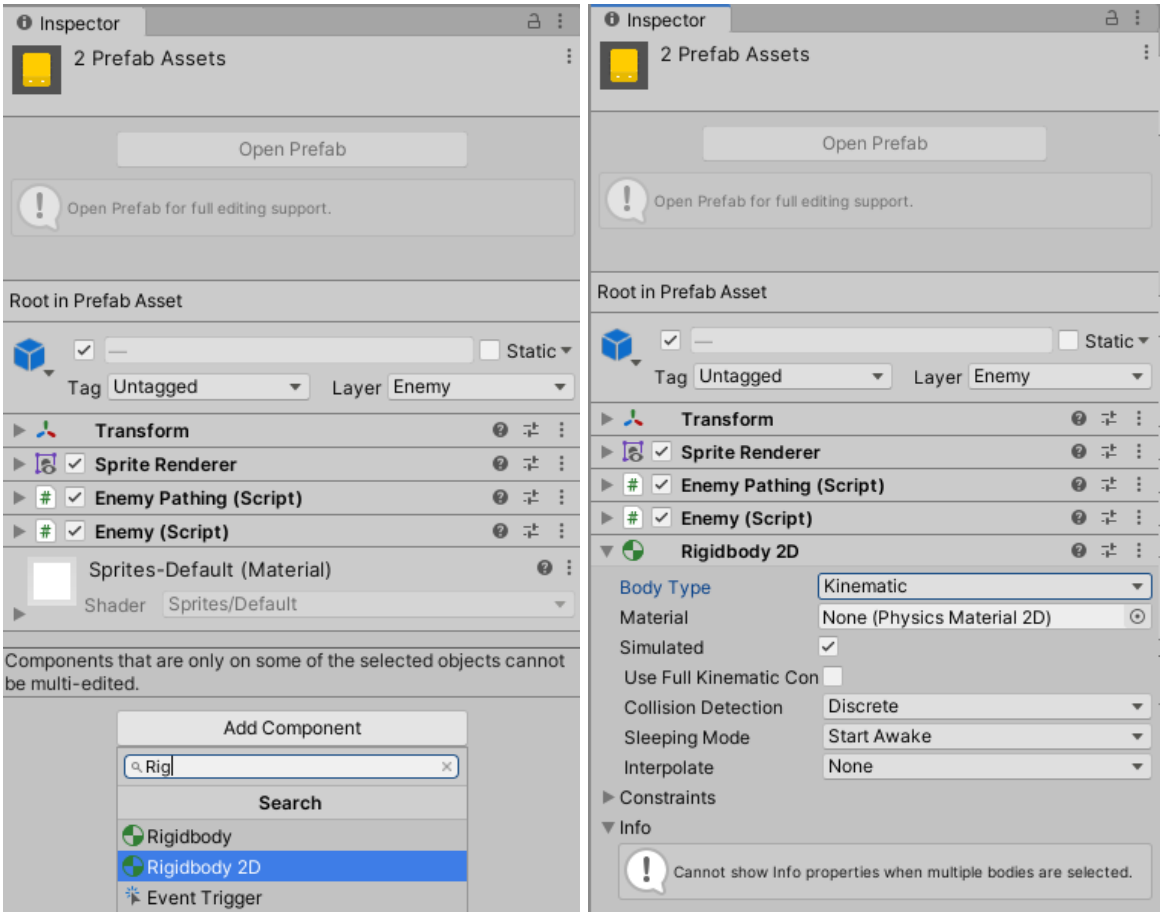


Obr. 162 Okno *Build Settings* po otvorení a po pridaní želaných scén.

Pozn. autora: Zatiaľ sa nevieme pohybovať cez scény, pretože nemáme ošetrené kedy nastane ukončenie hry a načítanie scény *GameOver*. To vyriešime v nasledujúcej kapitole. Je teda možné overiť funkčnosť presunu z úvodnej obrazovky do hry, prípadne zo záverečnej obrazovky späť do hry alebo na úvodnú obrazovku.

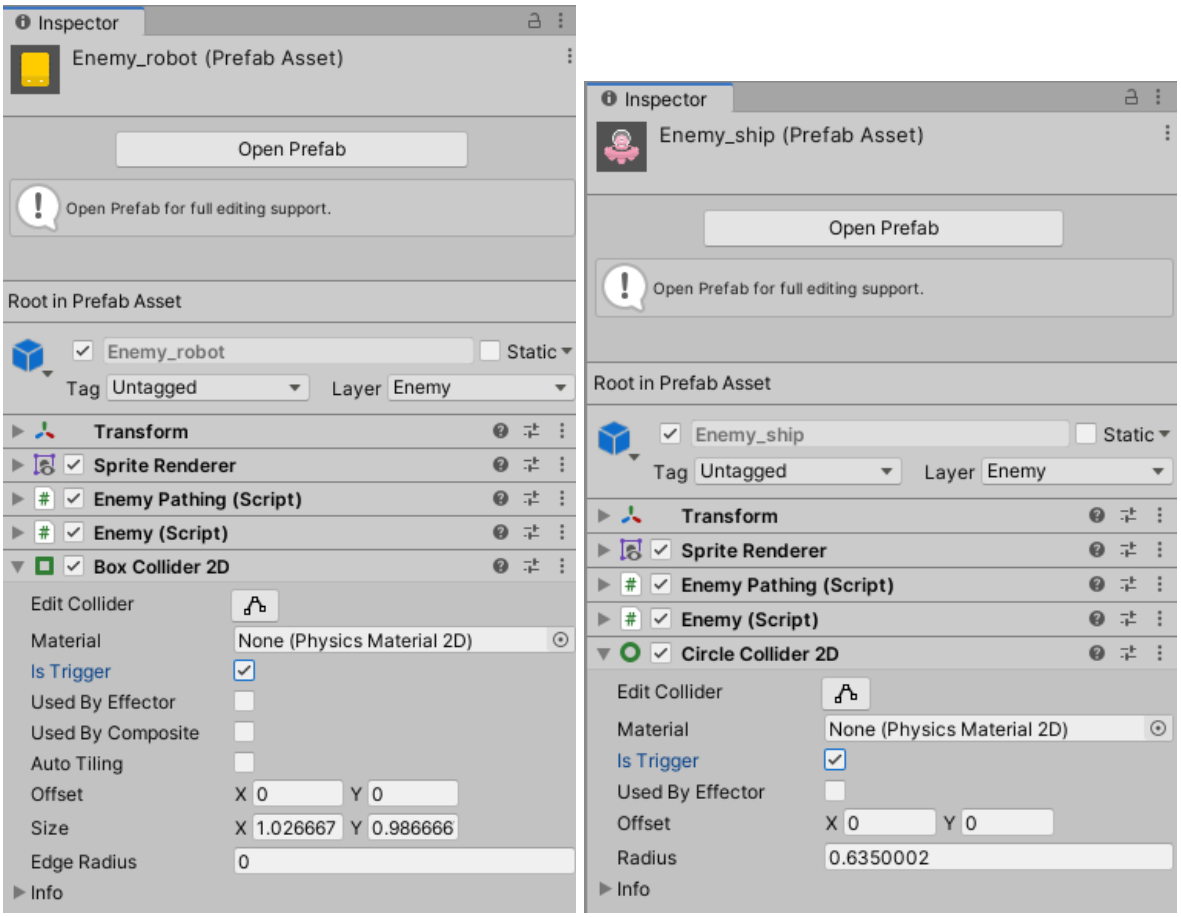
7.3 Ukončenie hry

Hru sme navrhli tak, že k ukončeniu hry príde v prípade, ak hodnota premennej *health* klesne na nulu. V tejto časti ešte zabezpečíme ukončenie hry, ak nastane kolízia hráča s nepriateľom. V priečinku *Prefabs* označíme oboch nepriateľov súčasne pomocou klávesy Ctrl a pridáme im komponent *Rigid Body 2D*, kde vlastnosť *Body Type* nastavíme na *Kinematic* (Obr. 163).



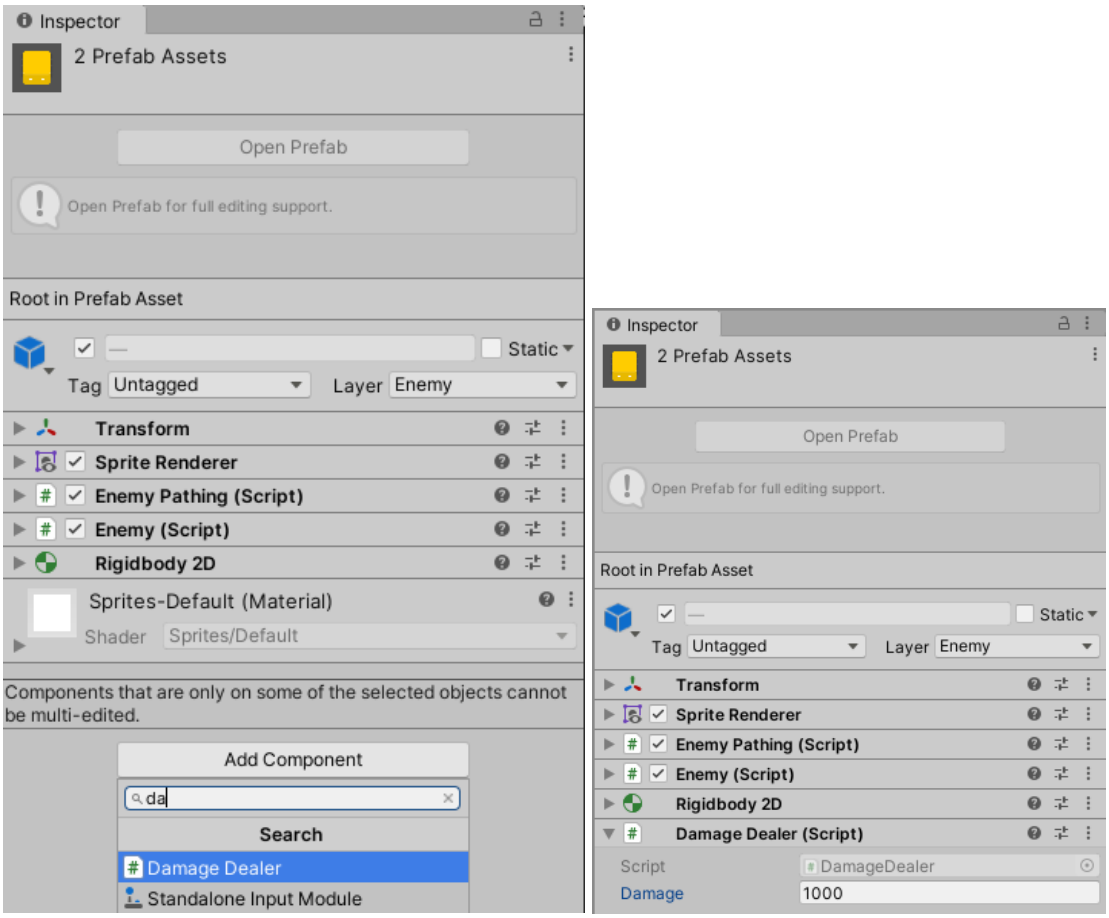
Obr. 163 Pridanie komponentu *Rigidbody 2D* a nastavenie vlastnosti *Body Type*.

U oboch, v príslušných *collideroch*, zmeníme vlastnosť *Is Trigger* na aktívnu (zaškrtnutím *checkboxu*), čím dosiahneme možnosť aplikovať požadované správanie pri kolízii týchto objektov s iným objektom (Obr. 164). V našom prípade nastane ukončenie hry pri strete hráča s nepriateľom.



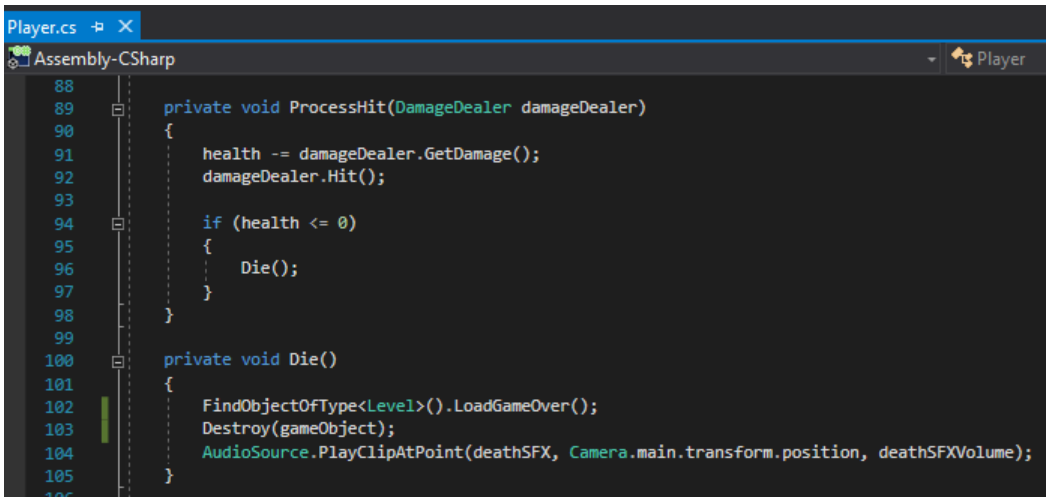
Obr. 164 Nastavenie vlastnosti *Is Trigger*.

Obom *prefabom* pridáme ako komponent skript *DamageDealer*, kde hodnotu premennej *damage* nastavíme na 1 000 (Obr. 165). Hodnota je určená podľa nastavení objektu *Player* v premennej *health*. Ak dôjde k stretu postavy hráča s nepriateľom, smrť nastáva okamžite.



Obr. 165 Pridanie skriptu *prefabom* nepriateľov a zmena premennej *damage*.

V skripte *Player.cs* v časti kódu kde dochádza k jeho úmrtiu, sa postaráme o vyvolanie verejnej metódy *LoadGameOver()*, ktorú sme definovali v *Level.cs*. Tým nastane ukončenie hry. Ako prvé musíme zabezpečiť prepojenie na tento skript, čo je možné pomocou metódy *FindObjectOfType()* triedy *Object* (Obr. 166), prípadne si môžeme vytvoriť referenčnú premennú.



Obr. 166 Fragment kódu skriptu *Player.cs*.

Uložíme zmeny a vrátime sa do editora Unity, kde môžeme otestovať implementovanú funkcionalitu.

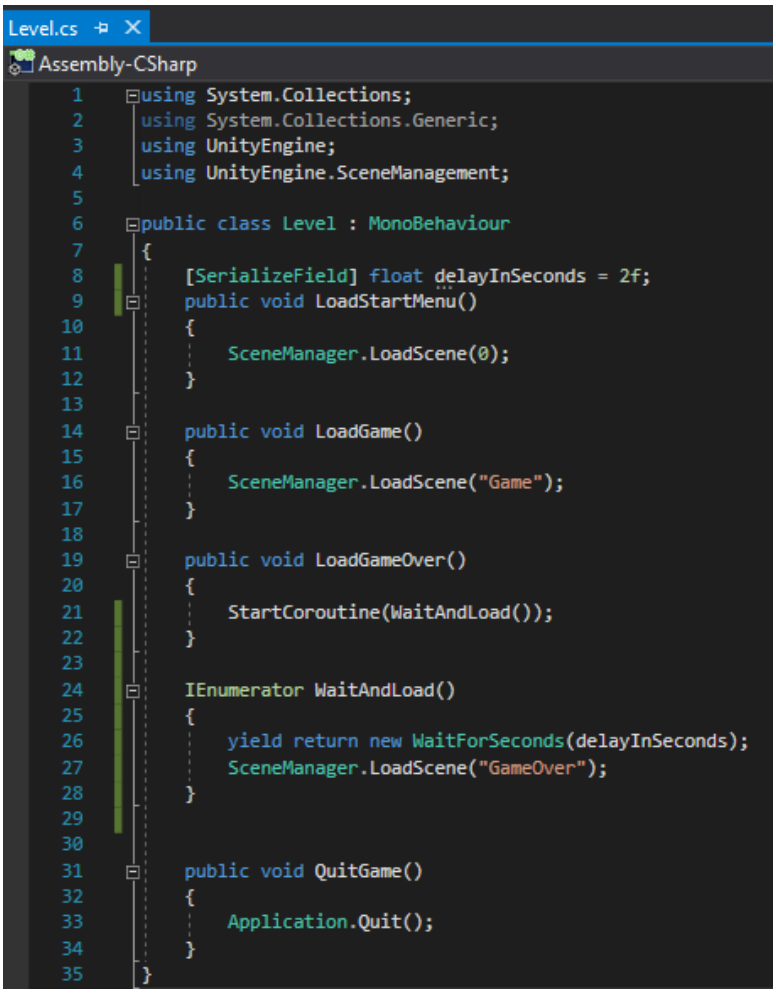
V prípade smrti postavy hráča je cieľom zabezpečiť krátke pozastavenie (*delay*), napríklad pre prípad začlenenia prehratia animácie jeho úmrtia, skôr ako nastane načítanie scény *GameOver*. Zabezpečíme to korutinou *WaitAndLoad*(), ktorú vytvoríme v skripte *Level.cs*. Je prirodzené, že zmeny robíme práve v tomto skripte, pretože našim cieľom je pozastaviť volanie metódy *LoadGameOver*(), ktorá zabezpečuje prehratie ukončovacej scény. Ako prvé v skripte *Level.cs* zavoláme na správnom mieste korutinu, ktorá sa o toto postará. Toto pozastavenie má nastať v prípade keď bude volaná metóda *LoadGameOver*(), preto korutinu voláme z jej tela.

```
public void LoadGameOver()
{
    StartCoroutine(WaitAndLoad());
}
```

Samotná korutina sa postará o nahratie scény *GameOver*, ale až po uplynutí času, ktorý udáva definovaná premenná *delayInSeconds*.

```
[SerializeField] float delayInSeconds = 2f;

IEnumerator WaitAndLoad()
{
    yield return new WaitForSeconds(delayInSeconds);
    SceneManager.LoadScene("Game Over");
}
```

Implementovanú funkcionalitu zachytáva obrázok 167.

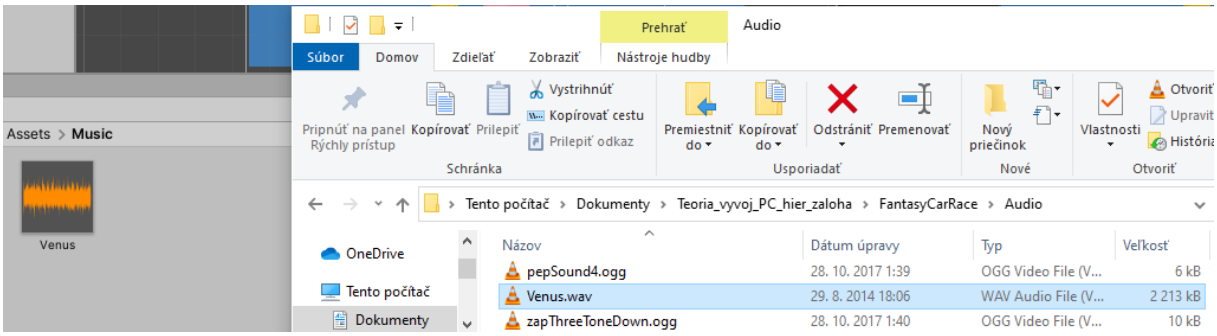


Obr. 167 Vizuál skriptu *Level.cs*.

7.4 Využitie návrhového vzoru *Singleton Pattern*

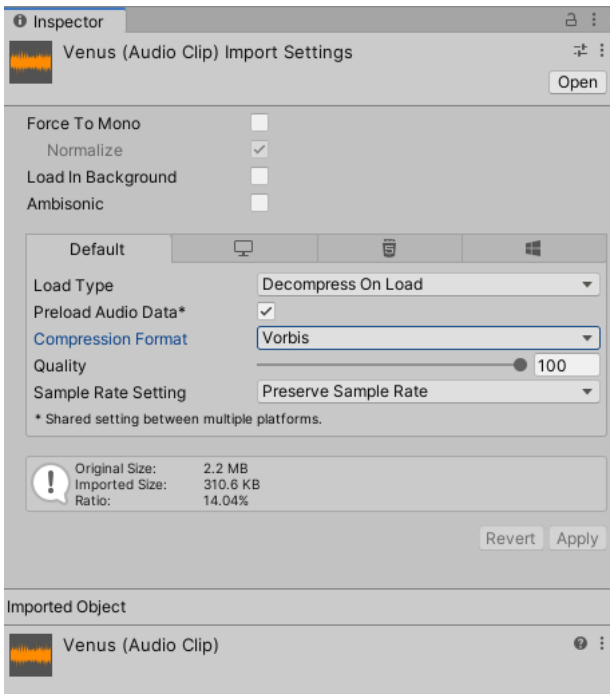
Cieľom je zakomponovať prehrávanie hudby do celej hry, t. j. od načítania úvodnej obrazovky až po ukončovaciu scénu. Aby sme nemuseli objekt, ktorý rieši prehratie zvukového záznamu dávať do každej scény, čo by okrem iného spôsobilo pri každom prechode prehratie zvukovej stopy vždy od začiatku, využijeme návrhový vzor *Singleton Pattern*. Tým zabezpečíme vytvorenie len jednej inštancie príslušného objektu. Problematiku tohto návrhového vzoru sme rozoberali aj v skriptách autorky (Jurinová, 2022).

Začneme vytvorením priečinku *Music*, do ktorého presunieme zdrojový súbor, ktorý sme si stiahli z webu https://opengameart.org.



Obr. 168 Pridanie zvukového záznamu do projektu.

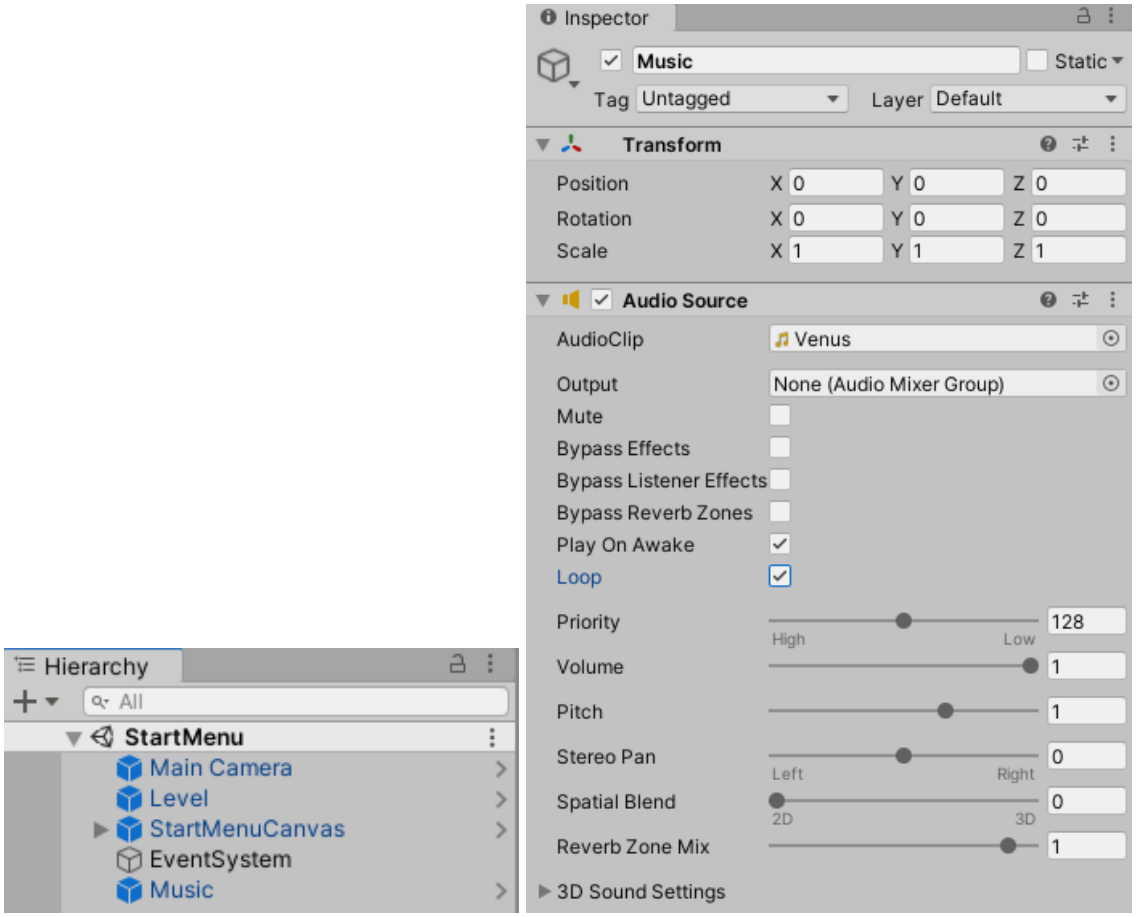
Z vlastností zvukového záznamu (*audio clip*) je zaujímavé si všimnúť, že Unity používa *Vorbis compression format* a umožňuje optimalizovať veľkosť súboru nastavením vlastnosti *Quality* (Obr. 169). V tomto prípade má originálna zvuková stopa veľkosť 2,2 MB a importovaná verzia 310,6 KB, čo je celkom dobrý výsledok, takže sú viac menej zbytočné dodatočné úpravy vzhľadom k veľkosti.



Obr. 169 Vlastnosti *audio clipu*.

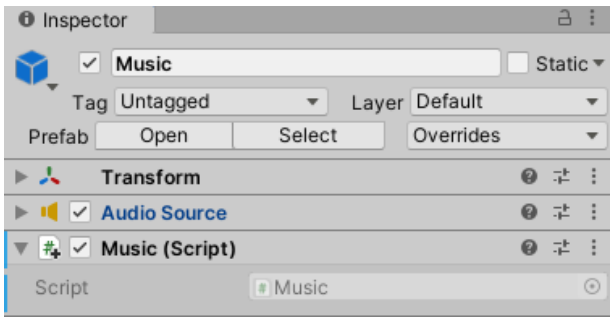
V scéne *StartMenu* v okne *Hierarchy* vytvoríme nový *Game Object*, ktorý pomenujeme *Music* a resetneme jeho nastavenia v komponente *Transform*.

Následne presunieme zvukovú stopu *Venus* priamo do *Inspectoru* tohto objektu, čo automaticky vytvorí *Audio Source* komponent a priradí túto zvukovú stopu ako zdrojový súbor pre vlastnosť *AudioClip*. Aby sa prehratie zvukového záznamu opakovalo, zaškrtneme vlastnosť Loop (Obr. 170). Z objektu vytvoríme *prefab*.



Obr. 170 Vytvorenie objektu *Music* a jeho nastavenie vlastností.

V scéne *StartMenu* pridáme objektu *Music* skript s rovnakým názvom ako komponent (Obr. 171).



Obr. 171 Pridanie skriptu ako komponent.

Definujeme funkciu *Awake*(). V nej zavoláme súkromnú metódu *SetUpSingleton*(), v ktorej sa postaráme buď o zničenie, ak existuje viacero inštancií typu *Music*, alebo o načítanie už existujúcej inštancie.

Táto metóda funguje tak, že v prípade existencie viacerých inštancií triedy *Music* sa postará o ich deaktiváciu, a v prípade existencie práve jednej inštancie zabráni jej zničeniu. Pomocou metódy *FindObjetsOfType()*, ktorá je schopná vrátiť pole všetkých objektov špecifikovaného typu v projekte, a vďaka metóde *GetType*(), ktorá zistí, aký typ hľadáme pomocou vlastnosti *length* triedy *Array*, spočítame všetky objekty typu *Music*. V prípade existencie viacerých inštancií tieto najprv zneaktívnime, a následne zničíme. Ak existuje len jedna inštancia, postaráme sa o to, aby nedošlo k jej zničeniu (Obr. 172).

Music.cs

Assembly-CSharp

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class Music : MonoBehaviour
6   {
7       void Awake()
8       {
9           SetUpSingleton();
10      }
11
12      private void SetUpSingleton()
13      {
14          if (FindObjectsOfType(GetType()).Length > 1)
15          {
16              gameObject.SetActive(false);
17              Destroy(gameObject);
18          }
19          else
20          {
21              DontDestroyOnLoad(gameObject);
22          }
23      }
24  }
```

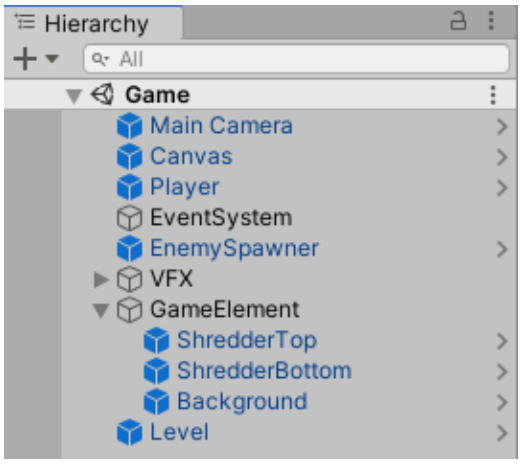
Obr. 172 Vizuál skriptu *Music.cs*.

Uložíme zmeny a vrátime sa otestovať funkčnosť do editora Unity. Nezabudneme aplikovať tieto zmeny na *prefab*.

7.5 Pridanie skóre do hry

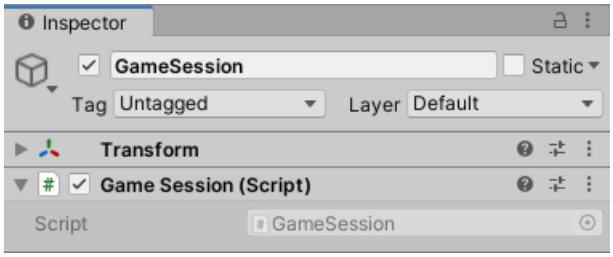
Každý z nepriateľov bude mať pomocou premennej *scoreValue* definovanú hodnotu, podľa ktorej sa bude navyšovať skóre hráča. Okrem zobrazenia skóre v scéne *Game* ho budeme chcieť zobraziť aj v prípade ukončenia hry.

Skôr než začneme, v scéne *Game* mierne upravíme štruktúru v okne *Hierarchy*, kde vytvoríme prázdny *Game Object* s názvom *GameElements* a pod neho umiestnime objekty, ako ilustruje obrázok 173, z dôvodu prehľadnosti.



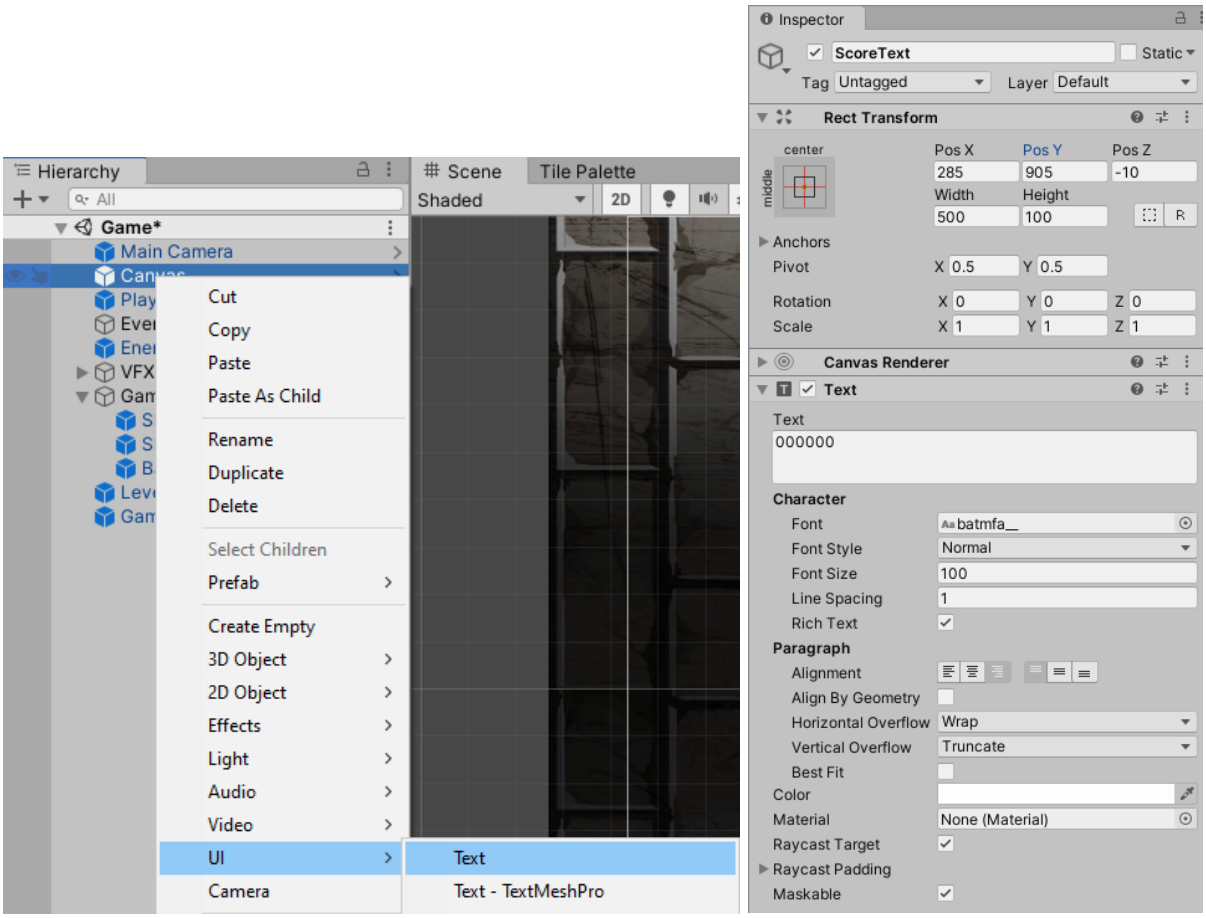
Obr. 173 Štruktúra scény *Game*.

Následne vytvoríme nový *Game Object* s názvom *GameSession*, ktorému priradíme skript s rovnakým názvom a urobíme z neho *prefab* (Obr. 174). Pre účely tejto implementácie vytvoríme ešte jeden skript s názvom *ScoreDisplay*.



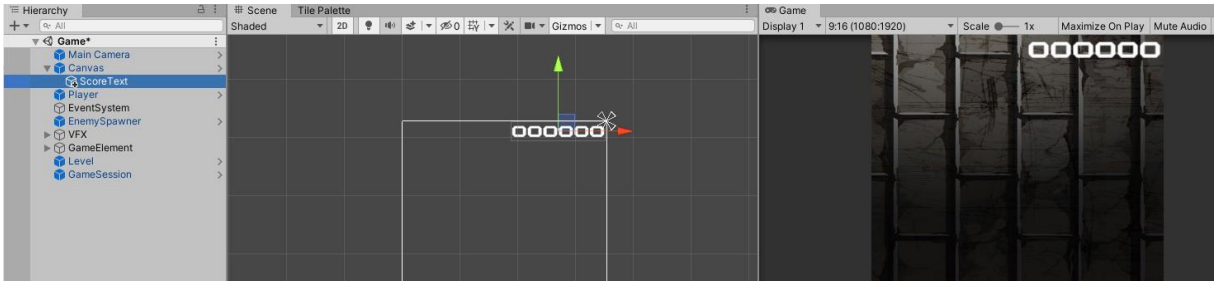
Obr. 174 Vytvorenie objektu *GameSession*.

Objektu *Canvas* pridáme potomka z kategórie *UI Text* a pomenujeme ho *ScoreText*. Zmeníme mu font, veľkosť písma, farbu a do textu vložíme 6 núl ako ilustráciu toho, či bude priestor dostatočne veľký pre zobrazenie skóre. Zarovnanie použijeme doprava (Obr. 175).



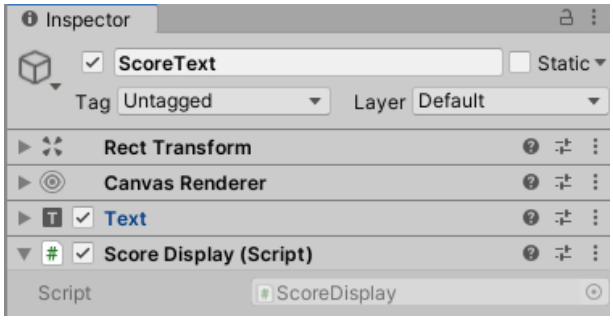
Obr. 175 Pridanie objektu *ScoreText* a jeho vlastnosti.

Následne v okne *Hierarchy* vyberieme objekt *ScoreText* a umiestnime ho do pravého horného rohu. Upravíme jeho veľkosť tak, aby bolo všetko viditeľné (Obr. 176). Dôležitý krok je nastaviť *Anchors*. Nastavujeme na pravý horný roh, čo zabezpečí, že aj pri zmene veľkosti hry, resp. rozlíšenia a pomeru (*screen resolution* and *aspect ratio*), objekt ostane prichytený na správnom mieste.



Obr. 176 Umiestnenie objektu *ScoreText*.

Skript *ScoreDisplay.cs*, ktorý sme už vytvorili pridáme ako komponent tomuto objektu (Obr. 177). Zmeny nezabudneme aplikovať na *prefab*.



Obr. 177 Pridanie skriptu ako komponent.

V skripte *GameSession.cs* sa postaráme o funkcionalitu udržiavania skóre. Je zrejmé, že pri implementácii opäť využijeme návrhový vzor *Singleton*. Je to z dôvodu, že v prípade zobrazenia skóre v scéne *GameOver* chceme zobraziť obsah inštancie, ktorá vznikla v scéne *Game*. Ako prvé definujeme v skripte premennú *score*, ktorú inicializujeme na 0.

```
int score = 0;
```

Funkcie *Start()* a *Update()* nebudeme potrebovať. Definujeme *Awake()*, v ktorej zavoláme metódu *SetUpSingleton()*, ktorú následne definujeme.

```
private void Awake()
{
    SetUpSingleton();
}
```

V metóde definujeme lokálnu premennú *numberGameSession*, ktorá bude reprezentovať počet existujúcich inštancií typu *GameSession* v projekte.

```
int numberGameSession = FindObjectsOfType<GameSession>().Length;
```

Pri existencii viacerých inštancií sa postaráme o ich zničenie. V opačnom prípade zabezpečíme, aby nedošlo k zničeniu tejto inštancie pri načítaní novej scény.

```
private void SetUpSingleton()
{
    int numberGameSession = FindObjectsOfType<GameSession>().Length;
    if (numberGameSession > 1)
    {
        gameObject.SetActive(false);
        Destroy(gameObject);
    }
    else
    {
        DontDestroyOnLoad(gameObject);
    }
}
```

V skripte *GameSession.cs* vytvoríme ďalšie tri verejné metódy, ktoré budú volané z iných skriptov. Pre premennú *score* definujeme verejné prístupové metódy, tzv. *setter* a *getter*. Definujeme metódu *ResetGame()*, ktorá sa postará o zničenie objektu v prípade začatia novej hry.

```
public int GetScore()
{
    return score;
}

public void AddToScore(int scoreValue)
{
    score += scoreValue;
}

public void ResetGame()
{
    Destroy(gameObject);
}
```

Obr. 178 Vizuál skriptu *GameSession.cs*.

Ďalšie zmeny realizujeme v skripte *ScoreDisplay.cs*, kde ako prvé pridáme *namespace*: `using UnityEngine.UI;` aby sme mohli pracovať s objektom *ScoreText*. Ak by sme do scény vložili objekt typu *TextMeshPro*, museli by sme pridať iný namespace: `using TMPro;` aby sme mohli pracovať s objektom typu *TextMeshPro*.

Deklarujeme premennú *scoreText* typu *Text* reprezentujúcu obsah, ktorý budeme zobrazovať v objekte *ScoreText*.

```
Text scoreText;
```

Vo funkcii *Start()* nadviažeme komunikáciu pomocou metódy *GetComponet()* triedy *GameObject*, ktorá vráti inštanciu daného typu.

```
void Start()
{
    scoreText = GetComponent<Text>();
}
```

Vieme, že skóre udržiavame v skripte *GameSession.cs*, preto potrebujeme definovať ďalšiu premennú tohto typu, aby sme zaistili prepojenie medzi týmito dvomi triedami a mohli volať príslušné metódy.

```
GameSession gameSession;
```

Premennú *gameSession* vo funkcii *Start()* inicializujeme existujúcou inštanciou, ktorú vyhľadáme využitím metódy *FindObjectOfType()* triedy *Object*.

```
gameSession = FindObjectOfType<GameSession>();
```

Vo funkcii *Update()* sa postaráme o zobrazenie aktuálneho skóre. Hodnotu skóre získame zavolaním metódy *GetScore()* a obsah celočíselnej premennej prekonvertujeme do reťazca využitím metódy *ToString()* triedy *Object*.

```
void Update()
{
    scoreText.text = gameSession.GetScore().ToString();
}
```

ScoreDisplay.cs

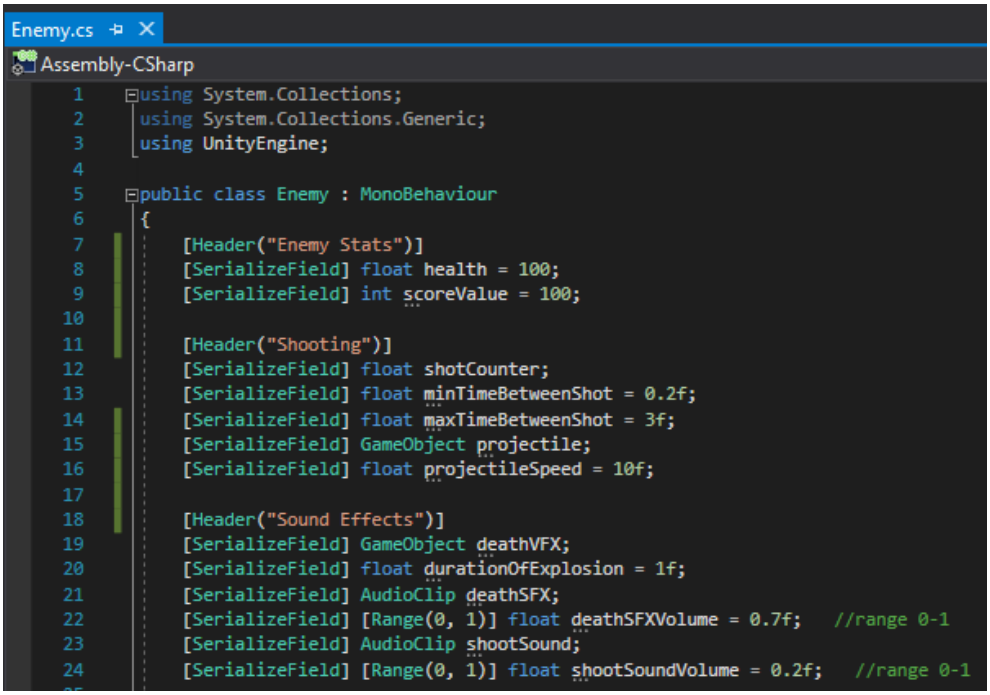
Assembly-CSharp

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.UI;
5
6  public class ScoreDisplay : MonoBehaviour
7  {
8      Text scoreText;
9      GameSession gameSession;
10
11     // Start is called before the first frame update
12     void Start()
13     {
14         scoreText = GetComponent<Text>();
15         gameSession = FindObjectOfType<GameSession>();
16     }
17
18     // Update is called once per frame
19     void Update()
20     {
21         scoreText.text = gameSession.GetScore().ToString();
22     }
23 }
```

Obr. 179 Vizuál skriptu *ScoreDisplay.cs*.

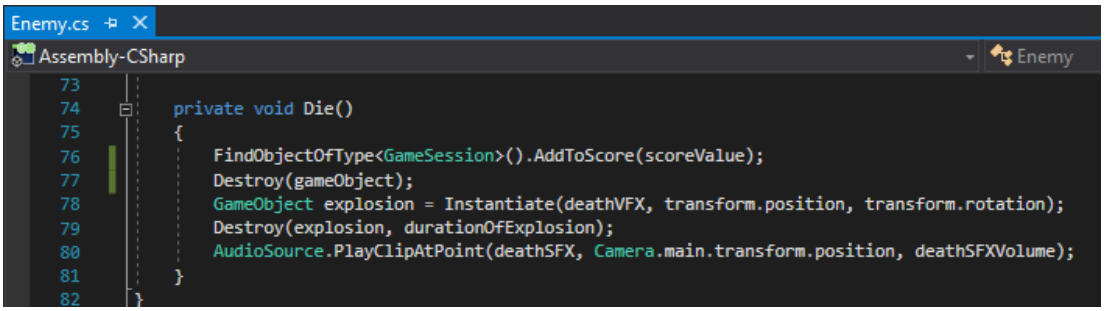
Po uložení zmien a návrate do editora Unity môžeme pri testovaní implementovanej funkcionality vidieť, že sa na obrazovke zobrazí skóre s hodnotou 0. Žiadny mechanizmus jej navyšovania zatiaľ nemôžeme pozorovať, pretože toto sme zatiaľ neimplementovali.

Skóre je nutné modifikovať v momente, keď zasiahneme nepriateľa. Preto v skripte *Enemy.cs* definujeme premennú *scoreValue*, ktorú inicializujeme napríklad hodnotou 100. Premennú definujeme ako *SerializeField*, takže v prípade potreby je možné túto premennú modifikovať priamo v editore Unity. Zároveň v tomto skripte vizuálne sprehľadníme definíciu premenných pomocou atribútu *HeaderAttribute*, čím doplníme jednoduchú kategorizáciu premenných tak, ako sme urobili v kapitole 5.2 v skripte *Player.cs* (Obr. 180).



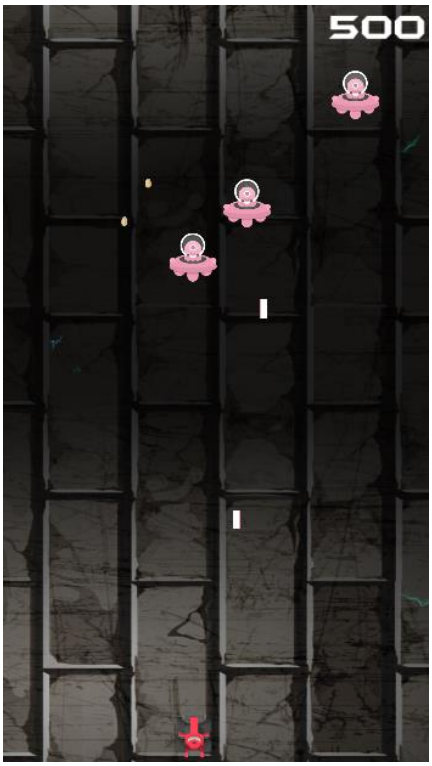
Obr. 180 Použitie atribútu *HeaderAttribute* v skripte *Enemy.cs*.

Úpravu skóre realizujeme zavolaním metódy *AddToScore()* v metóde *Die()*. Vytvoríme spojenie s objektom typu *GameSession* a metóde ako parameter odovzdáme hodnotu premennej *scoreValue* (Obr. 181).



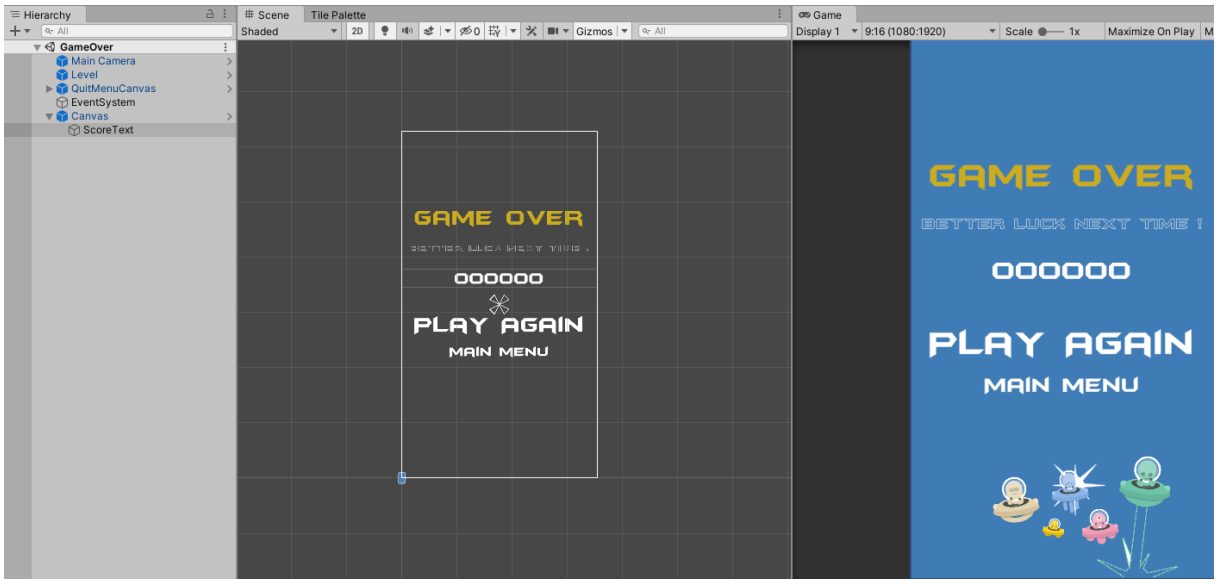
Obr. 181 Metóda *Die()* v skripte *Enemy.cs*.

Po uložení zmien a otestovaní funkčnosti v editore Unity môžeme sledovať, že pri zasiahnutí nepriateľa sa hodnota skóre navyšuje o hodnotu 100 (Obr. 182). Nastavenia hodnoty premennej *scoreValue* je plne v kompetencii vývojára.



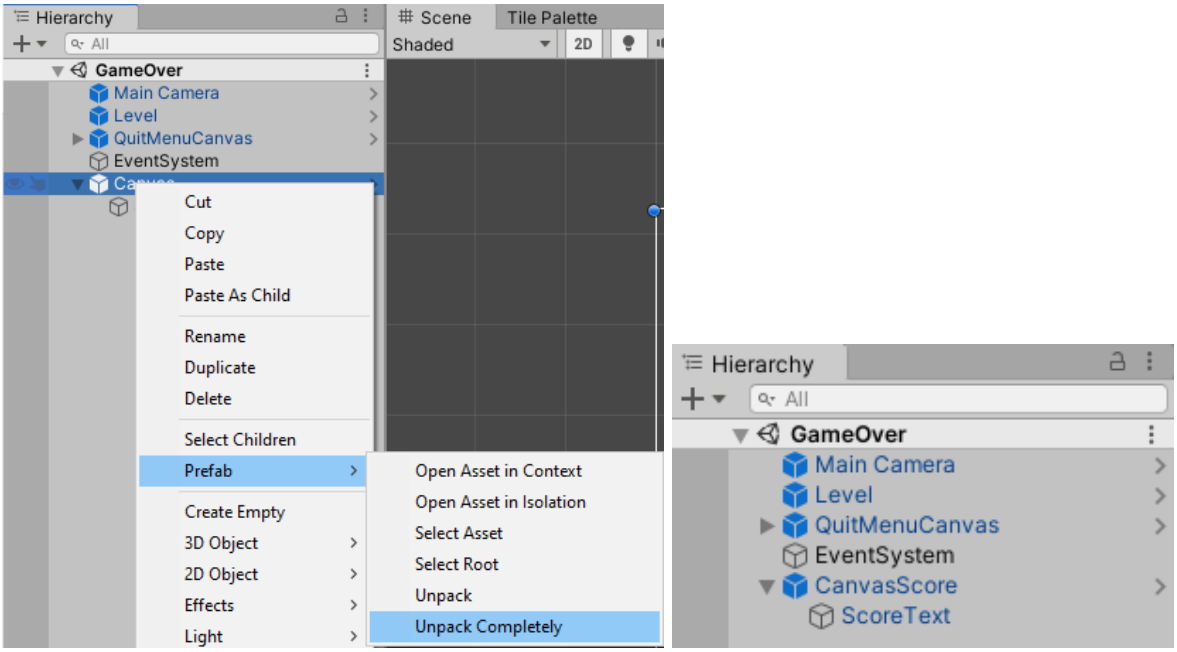
Obr. 182 Ilustrácia navyšovanie premennej *score*.

Objekt *GameSession* je implementovaný ako *Singleton*, preto s existenciou tejto inštancie môžeme počítať aj po prechode do scény *GameOver*. Musíme sa len postarať o zobrazenie nahratého skóre v tejto scéne. Využijeme na to *prefab Canvas* používaný v scéne *Game*. Objekt *ScoreText* umiestnime do stredu obrazovky, zmeníme mu veľkosť tak, aby jeho šírka korešpondovala so šírkou herného sveta (uľahčí nám to jeho centrovanie voči hernému svetu) a text vycentrujeme. Jeho zarovnanie vzhľadom na svet nastavíme na stred (*middle center anchor*) (Obr. 183).



Obr. 183 Pridanie *prefabu Canvas* do scény *GameOver*.

Z *prefabu Canvas* si vytvoríme nový *prefab*, ktorý bude slúžiť pre účely zobrazenia skóre v scéne *GameOver*. Využitím možnosti *Prefab → UnpackCompletly* zrušíme *prefab*, objekt premenujeme na *CanvasScore* a z objektu opätovne vytvoríme *prefab* (Obr. 184).



Obr. 184 Vytvorenie *prefabu CanvasScore*.

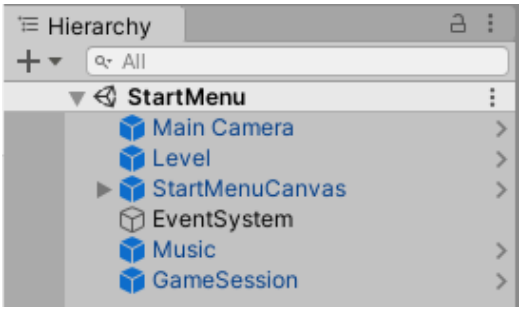
Po uložení zmien a otestovaní funkčnosti zistíme, že skóre sa na poslednej obrazovke zobrazí, avšak pri spustení novej hry sa neaktualizuje správnym

spôsobom. Je to z dôvodu, že sme nie korektne resetli naše načítanie. V skripte *GameSession.cs* sme vytvorili metódu *ResetGame()*, ale sme ju nezavolali. Treba ju zavolať v skripte *Level.cs* v metóde *LoadGame()* pomocou ktorej načítavame hru (Obr. 185).

```
Level.cs
Assembly-CSharp
4   using UnityEngine.SceneManagement;
5
6   public class Level : MonoBehaviour
7   {
8       [SerializeField] float delayInSeconds = 2f;
9       public void LoadStartMenu()
10      {
11          SceneManager.LoadScene(0);
12      }
13
14      public void LoadGame()
15      {
16          SceneManager.LoadScene("Game");
17          FindObjectOfType<GameSession>().ResetGame();
18      }
```

Obr. 185 Fragment kódu skriptu *Level.cs*.

Po uložení zmien a testovaní funkčnosti v editore Unity nastane pravdepodobne chyba *NullReferenceException*. Je to spôsobené tým, že inštanciu objektu *GameSession* treba pridať aj do scény *StartMenu* (Obr. 186).



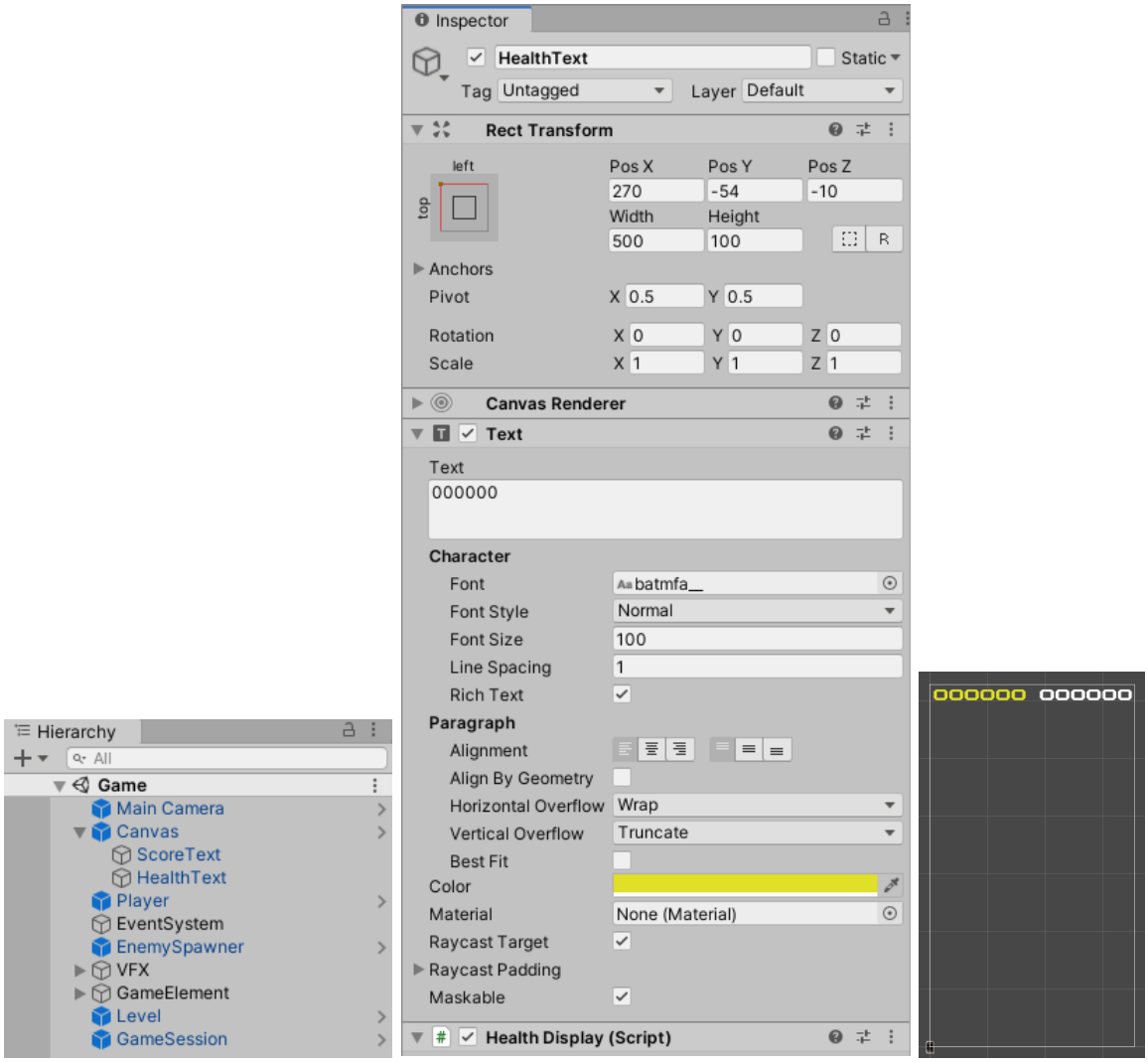
Obr. 186 Pridanie objektu *GameSession* do scény *StartMenu*.

7.6 Zobrazenie života postavy hráča

Cieľom je v scéne *Game* zobraziť okrem nahratého skóre aj život postavy hráča. Tento uchovávame v premennej *health*. Logika je obdobná, ako sme robili v prípade zobrazenia skóre. Preto odporúčame študujúcemu implementovať túto

funkcionalitu individuálne a následne si skontrolovať výsledok podľa časti uvedenej nižšie.

Zobrazenie životnosti hráča umiestnime do ľavej hornej časti obrazovky. Ako prvé zduplikujeme objekt *ScoreText* a premenujeme ho na *HealthScore*. Umiestnime ho na správne miesto v hernom priestore, zmeníme mu zarovnanie doprava a môžeme mu zmeniť aj farbu. Ukotvenie nastavíme na ľavý horný roh (*top left anchor*). Priradený skript *ScoreDisplay* nahradíme novým skriptom s názvom *HealthDisplay*. Zmeny aplikujeme na *prefab* (Obr. 187).

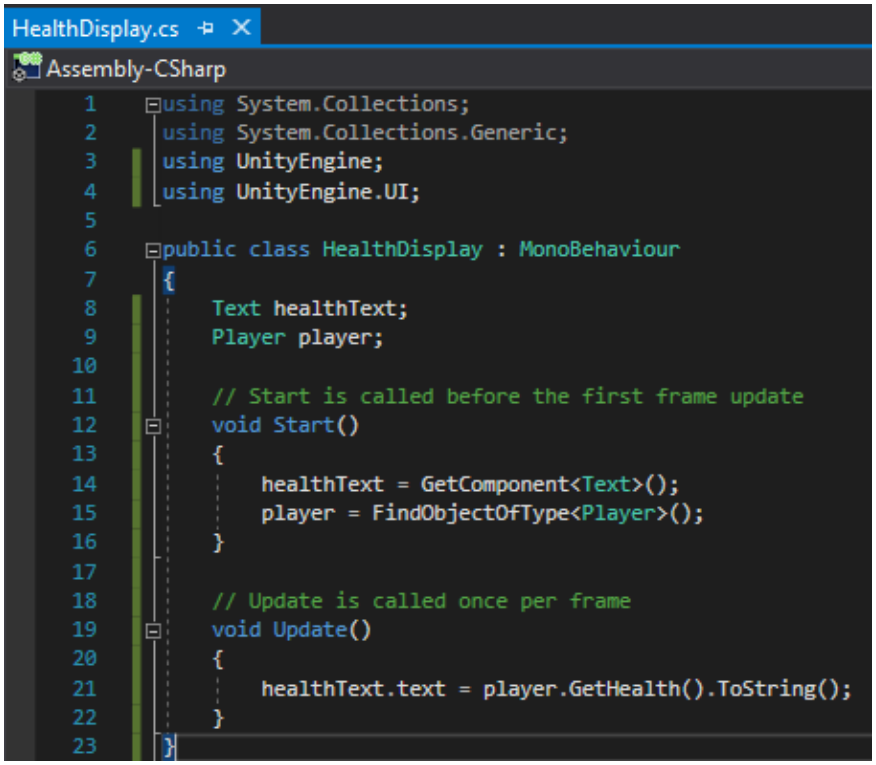


Obr. 187 Pridanie objektu *HealthText*, jeho nastavenia a vizuál.

V skripte *Player.cs* treba definovať novú verejnú metódu, tzv. *getter*, ktorý vráti obsah premennej *health*.

```
public int GetHealth()
{
    return health;
}
```

Ďalšie úpravy urobíme v skripte *HealthDisplay.cs*, kde bude implementovaná logika veľmi podobná ako v skripte *ScoreDisplay.cs*. Jediný rozdiel bude v tom, že nevytvárame spojenie s objektom typu *GameSession*, ale s objektom *Player* (Obr. 188).



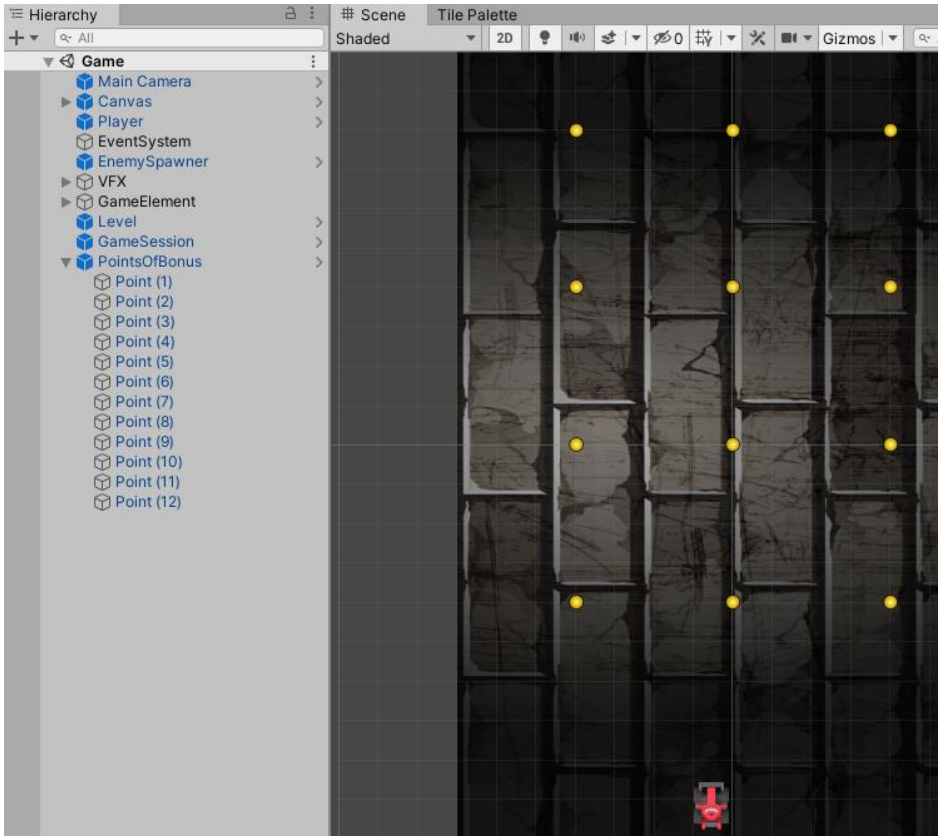
Obr. 188 Vizuál skriptu *HealthDisplay.cs*.

Po uložení zmien a návrate do editora Unity môžeme otestovať implementovanú funkcionalitu. Môžeme sledovať znižovanie životnosti postavy hráča pri zásahu nepriateľom, ako ja pri priamom strete s nepriateľom.

7.7 Vytvorenie systému bonusov

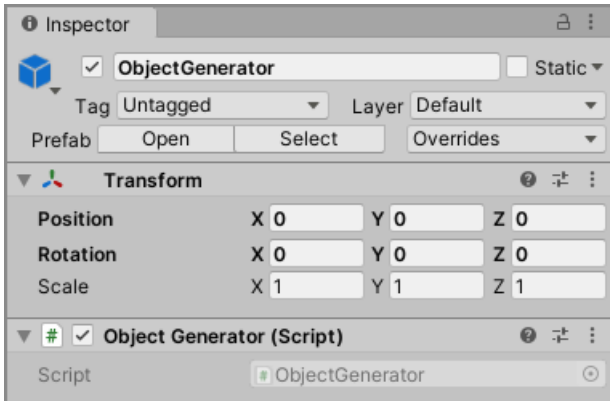
V tejto kapitole vytvoríme jednoduchý herný mechanizmus, ktorý bude zodpovedný za vznik bonusov v hernom priestore. Bonusy budú vznikať náhodne na vopred definovaných miestach. V prípade, že hráč tento bonus zoberie, budú sa na neho aplikovať požadované zmeny, napr. zvýšenie premennej *health* hráča, modifikácia rýchlosti hráča alebo strely, modifikácia času ako rýchlo je schopný strieľať a podobne.

Ako prvé vytvoríme v hernom priestore miesta, kde bude môcť objekt bonusu vzniknúť. Vytvoríme prázdny *Game Object* s názvom *PointsOfBonus*, ktorému resetneme nastavenia v komponente *Transform*. Ďalšie prázdne *Game Objecty* vytvoríme ako potomkov s názvom *Point (1)* až *Point (12)* a vhodne ich umiestnime v hernom svete. Pre ich vizuálne rozlíšenie v scéne herného priestoru sme použili žlté ikony kruhu (Obr. 189). Z objektu môžeme vytvoriť *prefab*.



Obr. 189 Vytvorenie objektu *PointsOfBonus*.

V okne *Hierarchy* vytvoríme ďalší prázdny *Game Object* s názvom *ObjectGenerator*, ktorému resetneme nastavenia v komponente *Transform* a pridáme mu skript s rovnakým názvom (Obr. 190). Z objektu tiež môžeme vytvoriť *prefab*. Tento objekt bude zodpovedný za vznik inštancií bonusov.

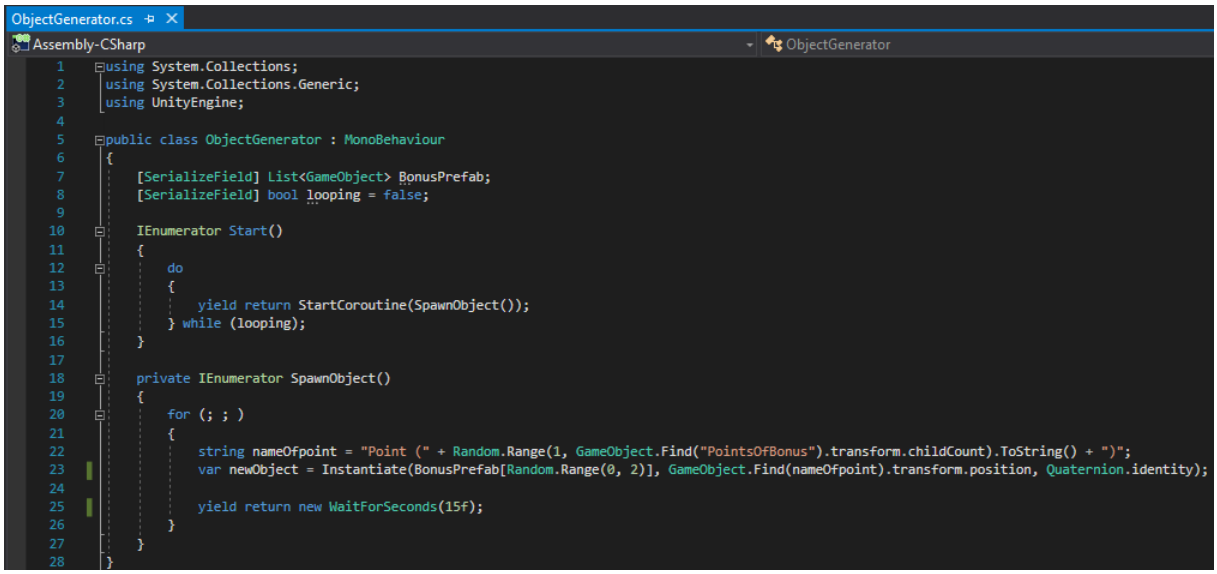


Obr. 190 Vytvorenie objektu *ObjectGenerator*.

V skripte *ObjectGenerator.cs* vytvoríme dve premenné. Premenná *BonusPrefab* typu zoznam (*list*) bude udržiavať jednotlivé *prefaby* bonusov a logická premenná *looping* zabezpečí ich opakované generovanie.

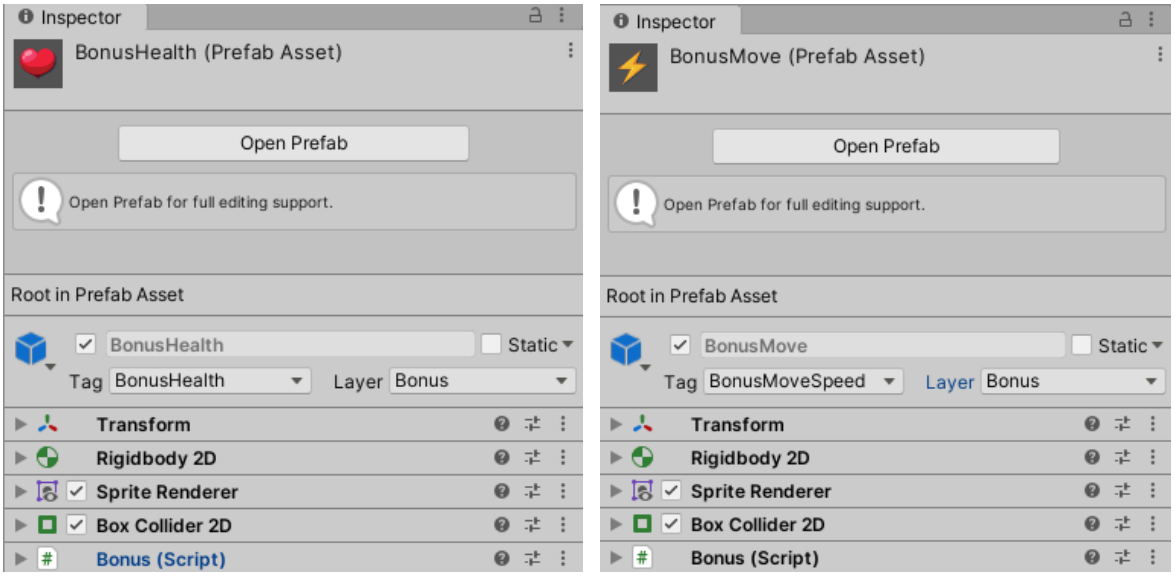
```
[SerializeField] List<GameObject> BonusPrefab;
[SerializeField] bool looping = false;
```

Z funkcie *Start*() spravíme korutinu, ktorá zabezpečí volanie ďalšej korutiny *SpawnObject*(). Táto bude zodpovedná za vznik jednotlivých inštancií bonusov každých 15 sekúnd vo vymedzenom hernom priestore reprezentovanom objektom *PointsOfBonus*. Tvorba inštancií bude náhodná. Náhodnosť udalostí sme zabezpečili použitím metódy *Range()* triedy *Random* (Obr. 191).



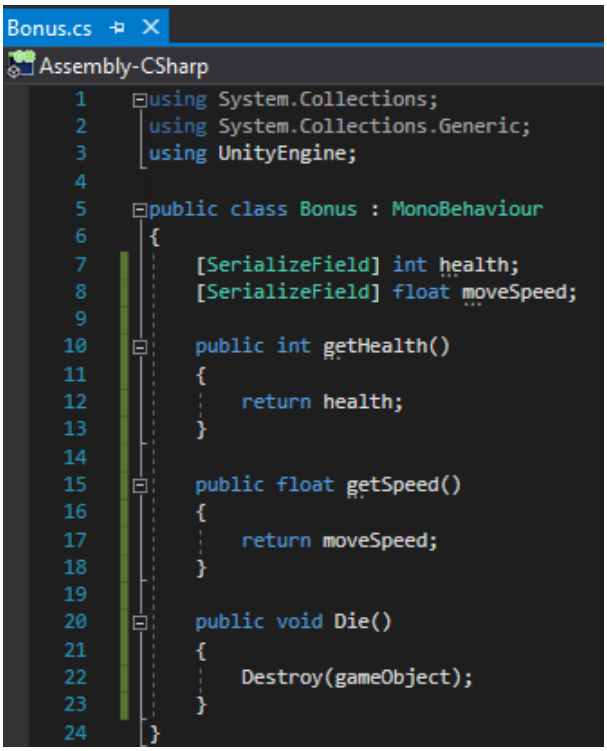
Obr. 191 Vizuál skriptu *ObjectGenerator.cs*.

Aby sme mohli inicializovať premennú *BonusPrefab*, vytvoríme objekty, ktoré budú reprezentovať bonusy. Objekty budú slúžiť na navýšenie premennej *health* a navýšenie rýchlosti pohybu hráča. Z dôvodu ich funkcionality zvolíme vhodné vizuálne reprezentácie. Objekty môžeme vytvoriť na základe už vytvorených *prefabov*, napr. *ShipLaser*. Okrem zmeny vizuálnej podoby vo vlastnosti *Sprite* je nutné prispôsobiť *collidery* objektov. Skript *DamageDealer* pridaný ako komponent odstránime a nahradíme ho novým skriptom *Bonus*. *Prefaby* objektov umiestnime do novej vrstvy *Bonus* a každému z objektov priradíme *tag* (Obr. 192).



Obr. 192 Nastavenia vlastností nových objektov pre bonusy.

V skripte *Bonus.cs* deklarujeme dve premenné, pomocou ktorých budeme ovplyvňovať zmenu premenných *health* a *moveSpeed* hráča. Pre každú premennú definujeme verejnú prístupovú metódu, tzv. *getter*. Vytvoríme metódu *Die()*, ktorá zabezpečí zničenie objektu. Túto metódu vyvoláme pri kolízii hráča s inštanciou objektu bonusu (Obr. 193).



Obr. 193 Vizuál skriptu *Bonus.cs*.

Posledné zmeny zrealizujeme v skripte *Player.cs*. V *OnTriggerEnter2D()* zabezpečíme požadovanú funkcionalitu. Inštancie bonusov sme umiestnili do vrstvy *Bonus*. Ide o 12–tú vrstvu, preto ak nastane kolízia s objektmi na tejto vrstve, postaráme sa o inicializáciu premennej *bonus* a túto odovzdáme ako parameter metóde *GetBonus()*.

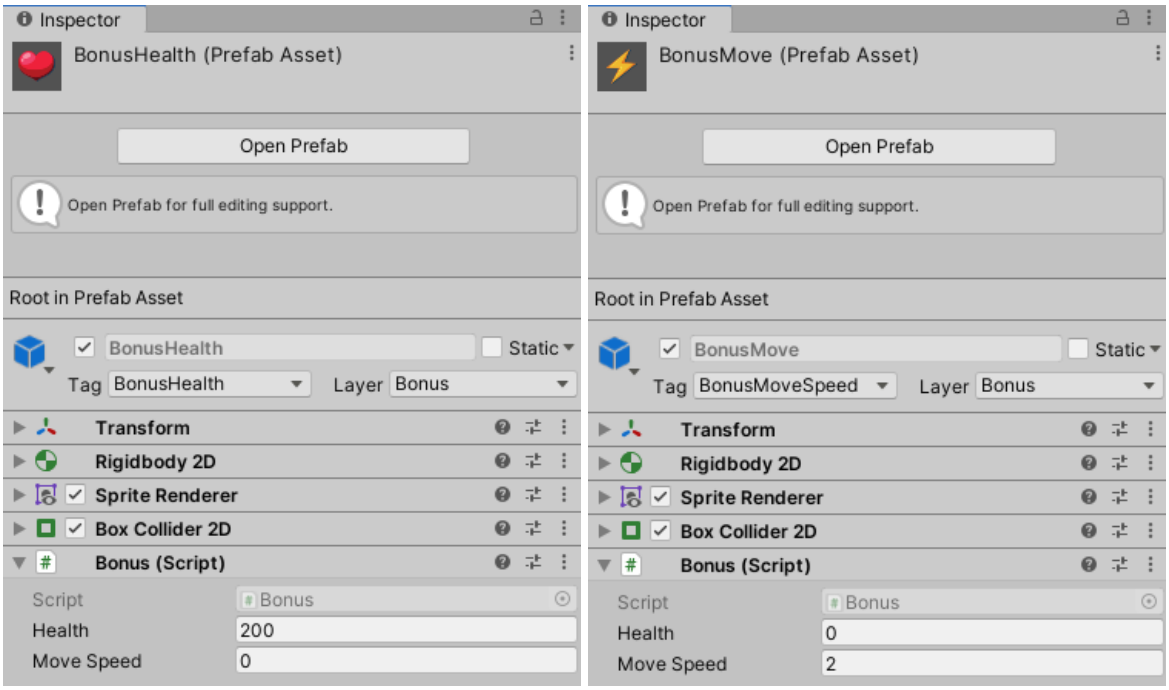
```
private void OnTriggerEnter2D(Collider2D other)
{
    if (!other.gameObject.layer.Equals(12))
    {
        DamageDealer damageDealer = other.gameObject.GetComponent<DamageDealer>();
        if (!damageDealer) { return; }
        ProcessHit(damageDealer);
    }
    else
    {
        Bonus bonus = other.gameObject.GetComponent<Bonus>();
```

```
            if (!bonus) { return; }
            GetBonus(bonus);
        }
    }
```

V metóde *GetBonus()* na základe *tagu* objektu odlíšime jednotlivé bonusy a postaráme sa o zmenu hodnôt príslušných premenných objektu *Player*. Nakoniec zavolaním metódy *Die()* sa postaráme o zánik danej inštancie.

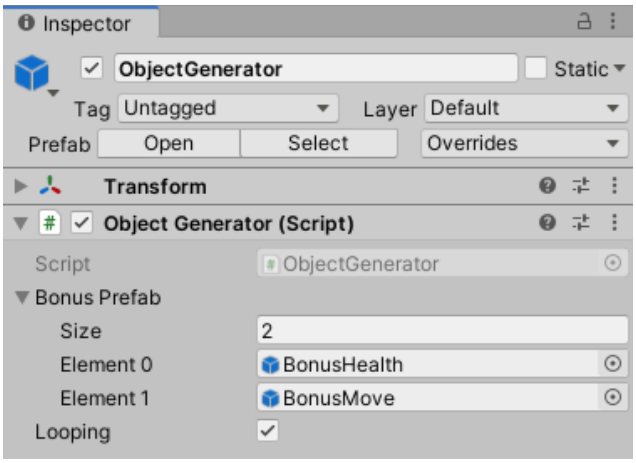
```
    private void GetBonus(Bonus bonus)
    {
        if (bonus.tag == "BonusHealth")
        {
            health += bonus.getHealth();

        }
        else if (bonus.tag == "BonusMoveSpeed")
        {
            moveSpeed += bonus.getSpeed();
        }
        bonus.Die();
    }
```

Po návrate do editora Unity pred samotným testovaním funkcionality sa postaráme o inicializáciu premenných podľa typu bonusu priamo na *prefaboch* daných bonusov (Obr. 194).



Obr. 194 Nastavenia premenných pre bonusy.

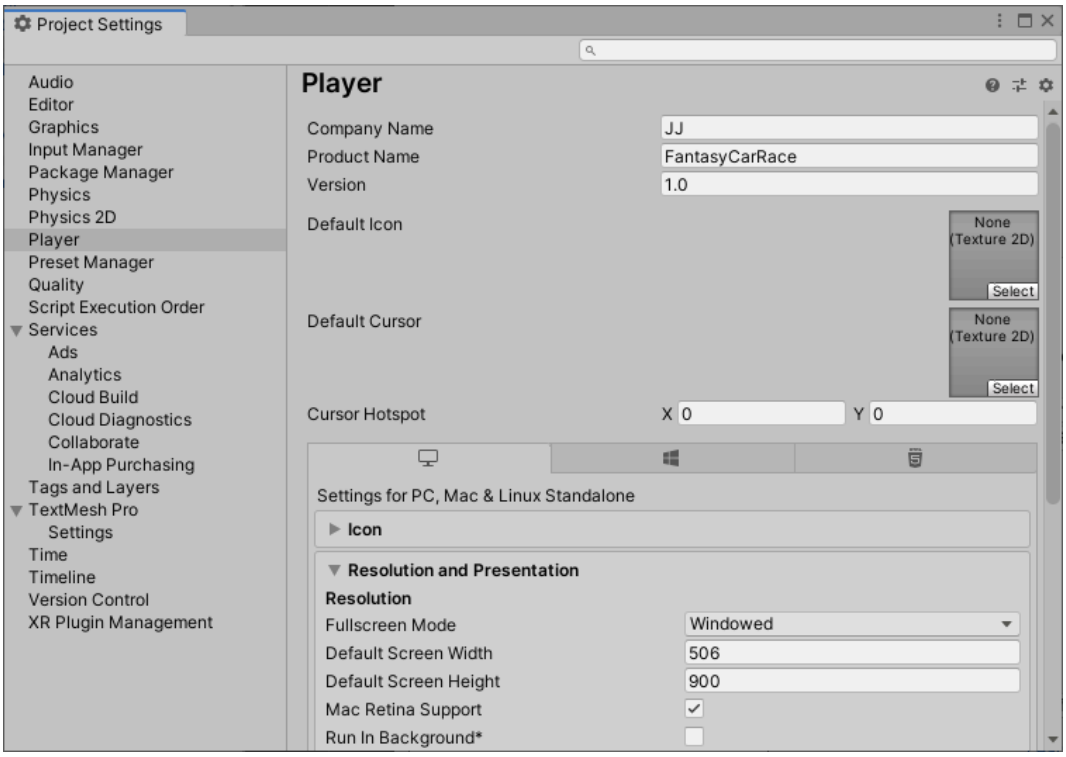
Teraz už len treba inicializovať premennú *bonusPrefab* objektu *ObjectGenerator* *prefabmi* bonusov a nastaviť premennú *looping* na *true* (Obr. 195) a môžeme otestovať implementovanú funkcionalitu.



Obr. 195 Inicializácia premennej *bonusPrefab*.

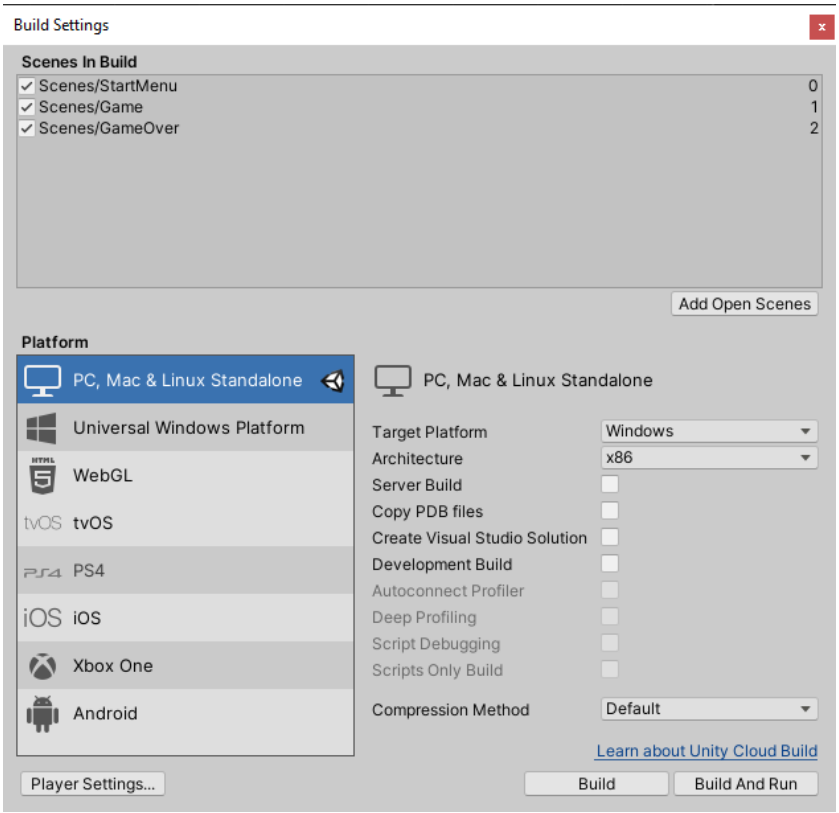
7.8 Doladenie posledných vecí a publikovanie hry

Pre výsledný export hry využijeme možnosť *File* → *Build Settings*, kde je možné okrem nastavenia cieľovej platformy nastaviť aj rôzne iné nastavenia. My exportujeme hru pre Windows a pomocou možnosti *Player Settings* v časti *Player* zvolíme *FullscreenMode* ako *Windoved*, pričom veľkosť výsledného okna je nutné nastaviť v pomere v ktorom sme hru vyvíjali. *Aspect Ratio*, ktorý sme pri vývoji hry používali bol 9:16 a rozlíšenie (*resolution*) 1080 x 1920. Ak chceme aby výška okna (*Default Screen Height*) bola 900, jeho šírka (*Default Screen Width*) musí byť 506 (hodnotu sme získali na základe výpočtu (900/16) * 9 = 506,25) (Obr. 196). Samozrejme, tieto, ako aj iné nastavenia exportu, sú plne na študujúcom.



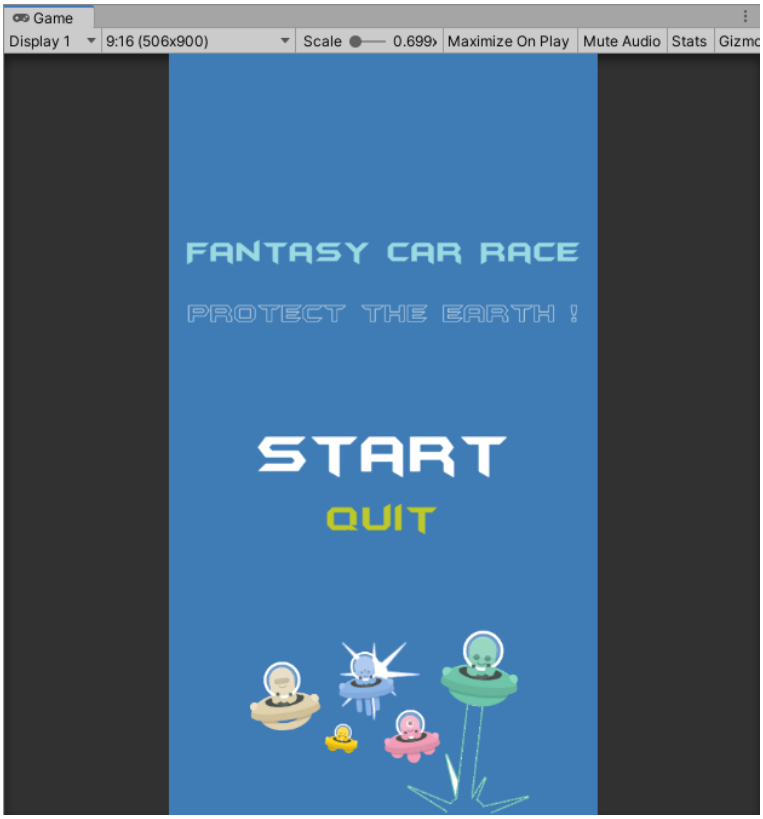
Obr. 196 Nastavenia v časti *Player* pred výsledným exportovaním.

Potom už stačí využiť možnosť *Build* (Obr. 197).



Obr. 197 Finálne exportovanie hry.

Pozn. autora: Po exportovaní hry sa vám môže stať, že obrázok pozadia, ktorý je použitý v scéne *StartMenu* a *GameOver*, nie presne kopíruje herný svet. Je to z dôvodu, že sme zmenili rozlíšenie exportovanej hry. Najjednoduchšie doladenie týchto rozmerov spočíva v pridaní rozlíšenia 506x900 v pomere 9:16 v *okne Game view* a prispôsobenie veľkosti obrázka pozadia pre toto rozlíšenie (Obr. 198).



Obr. 198 Prispôsobenie veľkosti pozadia vzhľadom na rozlíšenie exportu.

7.9 Kontrolné otázky a úlohy

1. Aký je rozdiel medzi objektmi *Text* a *TextMeshPro*?
2. Pomocou akého komponentu vieme z objektu vytvoriť tlačidlo?
3. Čo vyjadruje termín *Scene Index*?
4. Ako sa volá metóda, pomocou ktorej vieme načítať scénu? Aká je jej deklarácia? Je vhodnejšie používať index scény alebo jej názov?
5. Kde určujeme, ktoré scény a v akom poradí sa budú prehrávať?

6. Pomocou akej metódy vieme zatvoriť (skončiť) aplikáciu?
7. Aký je rozdiel medzi použitím *GetComponent<>* a *GameObject.FindObjectOfType<>*?
8. Definujte návrhový vzor *Singleton* a vymenujte jeho typické použitia v hrách.
9. Čo vyjadruje termín *UI Anchors*? V akom kontexte, prípadne s akým cieľom toto používame?
10. Ako viete zobraziť obsah číselnej premennej v objekte typu text?
11. Rozšírte systém zbierania bonusov, tieto môžu mať pozitívne, ako aj negatívne účinky. Resp. dopracujte mechanizmus zberu mincí na základe ktorých bude môcť hráč kupovať rôzne rozšírenia svojich schopností. Mince môžu byť produkované napríklad na mieste zabitia nepriateľa hráčom.
12. Dopracujte herný mechanizmus, ktorý bude v hernom priestore náhodne generovať rôzne objekty tvoriace prekážky v pohybe hráča, či nepriateľov.
13. Personalizujte hru hráčovi. Vytvorte v úvodnej snímke jednoduché rozhranie (*user interface*), kde umožníte hráčovi zadať svoj *nick*, zvoliť zapnutie, prípadne vypnutie prehrávania hudobného záznamu na pozadí, alebo aj iných zvukových efektov. Rozšírte ponuku zvukových klipov. Navrhnite rôzne úrovne náročnosti danej hry.
14. Dopracujte do hry ukladanie dát tak, aby ste mohli implementovať pauzu, návrat k rozohratej hre pri ukončení hry v procese, resp. rebríček hodnotenia (*leaderboard*).
15. Dopracujte viacero levelov danej hry a modifikujte priebeh prechodu jednotlivými scénami, napríklad na základe získania určeného skóre, resp. rozšírte hru o časové hľadisko a podmieňte prechod jednotlivými levelmi na základe času.
16. Časové hľadisko môžete zohľadniť aj v možnosti udeľovania skóre, ak sa hráčovi podarí ukončiť level v kratšom čase ako bolo nastavené, prípadne zaviesť rebríček hodnotenia jednotlivých levelov.

8 Zoznam použitej literatúry a informačných zdrojov

Datafont, 2021. Batman Forever. [online]. [cit. 18.7.2021]. Dostupné na: ⟨https://www.dafont.com/search.php?q=batman+forever⟩.

Davidson, Rick and Ben Tristem. 2019. Complete C# Unity Game Developer 2D. [online]. [cit. 13.4.2020]. Dostupné na: ⟨https://www.udemy.com/course/⟩.

FREEPIK. 2021. Space galaxy game template Free Vector. © 2010-2021. [online]. [cit. 18.4.2021]. Dostupné na: ⟨https://www.freepik.com/free-vector/space-galaxy-game-template_4307061.htm#query=game%20icon&position=28&from_view=search⟩.

JURINOVÁ, Jana, 2022. VÝVOJ 2D PC HRY – ARKANOID V UNITY. Návody na cvičenia; Trnava : Univerzita sv. Cyrila a Metoda v Trnave, 2022. ISBN 978-80-571-0225-7. Dostupné na: https://www.ucm.sk/sk/ucebne-texty-k-stiahnutiu/

Kenney, 2021a. Particle Pack. [online]. [cit. 21.7.2021]. Dostupné na: ⟨https://www.kenney.nl/assets/particle-pack⟩.

Kenney, 2021b. Digital Audio. [online]. [cit. 21.7.2021]. Dostupné na: ⟨https://www.kenney.nl/assets/digital-audio⟩.

Kenney, 2021c. Robot pack. [online]. [cit. 18.4.2021]. Dostupné na: ⟨https://www.kenney.nl/assets/robot-pack⟩.

Kenney, 2021d. Space Shooter Redux. [online]. [cit. 18.4.2021]. Dostupné na: ⟨https://www.kenney.nl/assets/space-shooter-redux⟩.

Kenney, 2021e. Alien UFO pack. [online]. [cit. 18.4.2021]. Dostupné na: ⟨https://www.kenney.nl/assets/alien-ufo-pack⟩.

Microsoft, 2021. IEnumerator Interface. [online]. [cit. 21.5.2021]. Dostupné na: ⟨https://docs.microsoft.com/en-us/dotnet/api/system.collections.ienumerator?view=net-6.0⟩.

SketchyLogic. 2014. OpenGameArt.org, Venus. [online]. [cit. 8.8.2021]. Dostupné na: ⟨https://opengameart.org/content/nes-shooter-music-5-tracks-3-jingles⟩.

STEWART, Samuel, 2020. Best Aspect Ratio For Gaming – Which Should I Choose? [online]. [cit. 8.5.2021]. Dostupné na: ⟨https://www.gamingscan.com/best-aspect-ratio-for-gaming/⟩.

Unity, 2021a. Input Manager. [online]. [cit. 18.4.2021]. Dostupné na: ⟨https://docs.unity3d.com/Manual/class-InputManager.html⟩.

Unity, 2021b. Coroutines. [online]. [cit. 21.5.2021]. Dostupné na: ⟨https://docs.unity3d.com/Manual/Coroutines.html⟩.

Unity, 2021c. ScriptableObject. [online]. [cit. 3.6.2021]. Dostupné na: ⟨https://docs.unity3d.com/Manual/class-ScriptableObject.html⟩.

Unity, 2021d. Layer-based collision detection. [online]. [cit. 18.6.2021]. Dostupné na: ⟨https://docs.unity3d.com/Manual/LayerBasedCollision.html⟩.

Unity, 2021e. Meshes. [online]. [cit. 8.8.2021]. Dostupné na: ⟨https://docs.unity3d.com/Manual/mesh.html⟩.

Unity, 2021f. EventSystem. [online]. [cit. 23.8.2021]. Dostupné na: ⟨https://docs.unity3d.com/2018.2/Documentation/ScriptReference/EventSystems.EventSystem.html⟩.

WIKIPEDIA, 2021. Shoot ‘em up. [online]. [cit. 5.10.2021]. Dostupné na: ⟨https://en.wikipedia.org/wiki/Shoot_%27em_up⟩.

Názov:	VÝVOJ 2D PC HRY –FANTASY CAR RACE V UNITY Návody na cvičenia (Skriptá a učebné texty)
Autor:	Ing. Jana Jurinová, PhD.
Recenzenti:	Ing. Miroslav Beňo, PhD. PaedDr. Patrik Voštinár, PhD.
Vydavateľ:	Univerzita sv. Cyrila a Metoda v Trnave
Rok:	2022
Rozsah:	162 strán / 11,11 AH
Náklad:	50 ks
Grafická úprava obálky:	Ing. Jana Jurinová, PhD.
Tlač:	elektronická verzia