Izrada igre preživljavanja na vrijeme u programskom alatu Unity

Rošić, Ivo

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Ivo Rošić

IZRADA IGRE PREŽIVLJAVANJA NA VRIJEME U PROGRAMSKOM ALATU UNITY

ZAVRŠNI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Ivo Rošić

Matični broj: 0016151299

Studij: Informacijski i poslovni sustavi

IZRADA IGRE PREŽIVLJAVANJA NA VRIJEME U PROGRAMSKOM ALATU UNITY

ZAVRŠNI RAD

Mentor/Mentorica:

Doc. dr. sc. Mladen Konecki

Varaždin, rujan 2023.

Ivo Rošić

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Završni rad bavi se procesom izrade 2D video igre korištenjem pogona za razvoj i izradu video igara Unity, te upoznavanja sa razvojnim okruženjem. U njemu će se razmatrati programski jezik C# i mogućnosti Unity-a, te proces izrade roguelike shoot-em up video igre nazvane Gladiator arena, Igra se sastoji od scene glavnog izbornika (engl. „*Main menu*“), te jednog nivou unutar kojeg je cilj igraču preživjeti što dulje. Unutar nivoa se nalazi arena u kojoj se stvaraju 4 vrste neprijatelja (svaka vrsta ima različite statistike (engl. „*stats*“), te jedna vrsta koristi napade na daljinu), te 2 vrste šefa neprijatelja (engl. „*Boss enemies*“). Unutar nivoa igrač unapređuje svoju razinu (engl. „*level*“) skupljanjem jedinica iskustva (engl. „*experience gems*“), te svakim napretkom dobije mogućnost poboljšanja željene statistike (zdravstvena traka (engl. „*health bar*“), radijus napada (engl. „*attack range*“), štetu (engl. „*damage*“)). Također mjeri se vrijeme preživljavanja u nivou, što omogućava ponovnu igrivost s ciljem dužeg preživljavanja. Pri gubitku igre igrač je predstavljen opcijom spremanja svojeg postignuća, te odlukom spremanja se postignuće uspoređuje sa već postojećim lokalnim postignućima i sprema 5 najboljih.

Ključne riječi: 2D video igre, roguelike, shoot-em up, Unity, C#, PlayerPrefs, ScriptableObjects

Sadržaj

1. Uvod

Roguelike shoot-em up žanr video igara je žanr koji s temelji na konceptima nasumične generacije svijeta i neprijatelja, mehanika trajne smrti (engl. „*permadeath*“), raznolikost oružja i sposobnosti te progresija i nadogradnja igrača. Kombinacija ovih koncepata omogućuje veliku ponovnu igrivost, jer svaki prolazak kroz igru (engl. „*playthrough*“) može biti komplet drugačiji od prošlog. Jedni od primjera roguelike shoot-em up igara su Vampire Survivors, Dead Estate i Neon Abyss.

Razvoj video igara se postiže sa pogonom za razvoj i izradu video igara (engl. „game engine“). To su alati koji sadrže razne komponente i alate za olakšavanje izradu video igara, kao što su grafički uređivači, upravljanje fizikom i kolizijama objekata, upravljanjem korisničkog sučelja video igre (engl. „*user interface*“ - UI), upravljanjem zvukom i upravljanje scenama. Jedni od poznatijih platformi za razvoj video igara su Unity, Unreal Engine, Godot i GameMaker.[1]

Unutar ovog projekta rekreirao sam nasumičnu generaciju neprijatelja, mehaniku trajne smrti sa spremanjem najboljeg postignuća u igri, te nadogradnju statistike igrača. Video igra se sastoji od 4 neprijatelja, svaki sa svojim statistikama i jedan koji koristi napade na daljinu, te 2 šefa neprijatelja. Svaki neprijatelj ima šansu baciti jedinicu iskustva koje služe igraču za napredak razine, te prelaskom na novu razinu (engl. „*level up*“) igrač dobije mogućnost poboljšanja svoje statistike za taj prolazak kroz igru. Statistike neprijatelja se poboljšavaju nakon svaka 3 neprijateljska vala (engl. „*enemy wave*“). Na kraju svakog prolaska igre igrač ima mogućnost spremiti postignuto vrijeme, te se najboljih 5 postignuća lokalno spremaju za prikaz igraču.

2. Unity

Unity je besplatni programski okvir i pogon za razvoj video igara (engl. „*game engine*“) i aplikacija u 2D, 2.5D ili 3D formatu razvijen od strane tvrtke Unity Technologies. On omogućuje kreiranje igara pomoću koda ili svog ugrađenog uređivača (engl. „*Unity Editor*“). Također Unity omogućuje razvoj proizvoda za različite platforme kao što su operativni sustavi (Windows, macOS, Linux), mobilni uređaji (iOS i Android), igračke konzole (Playstation i Xbox), uređaji za virtualnu stvarnost (VR) i proširenu stvarnost (AR), te web aplikacije. [2]

Unity omogućava sastavljanje i uvoz resursa, pisanje koda za interakciju s objektima, kreiranje ili uvoz animacija. On podržava sve glavne formata koje izvoze 3D aplikacije i audio aplikacije, te također podržava i Photoshop-ov .psd format i sve datoteke se mogu jednostavno ubaciti u Unity projekt za daljnje korištenje pri izradi video igre ili aplikacije. Unity ima i ugrađenu trgovinu resursima (engl. „*Unity Asset Store*“) unutar koje se mogu pronaći sve potrebne komponente za izradu video igre i aplikacije, kao što su umjetnosti, 3D modeli, animacije za modele, zvučni efekti, te dodatci za Unity. Dodatci za Unity uključuju dodatke koji služe za vizualno skriptiranje kao PlayMaker i Behave, napredna sjenčala (engl. „*shaders*“) i teksture i slično. Ovo nam ukazuje da je Unity sučelje u potpunosti skriptabilno, te da omogućuje datotekama trećih strana da se integriraju izravno u Unity GUI (engl. „*graphical user interface*“). Korisnicima je omogućeno da objave i prodaju svoje resurse u Unity trgovini resursa. [2]

2.1. Programski jezik – C#

C# (izgovara se „C sharp“) je objektno orijentirani i visoko tipizirani programski jezik razvijen od strane Microsofta. On se koristi za razvoj različitih vrsta aplikacija, uključujući web aplikacija, video igara i mobilnih aplikacija. Visoka tipizacija označava da varijable, klase i objekti trebaju imati dodijeljen tip podataka kod inicijalizacije. C#, kao objektno orijentirani programski jezik, se temelje na konceptima objekata i klasa, te također podržava paradigme funkcionalnog, generičkog i komponentnog orijentiranog programiranja.

C# programi se izvode na .NET platformi koja uključuje zajednički jezik za izvođenje (engl. „*common language runtime*“ - CLR) i biblioteke klasa. C# kod se kompilira u posredni jezik (engl. „*intermediate language*“ - IL), koji se zatim izvršava u CLR-u putem kompilacije u pravo vrijeme (engl. „*Just-In-Time*“ – JIT). Osim toga, .NET podržava međujezičnu interoperabilnost, omogućujući interakciju između različitih jezika. .NET također pruža mnoge biblioteke koje podržavaju različite zadatke, uključujući manipulaciju podacima, obradu iznimki

i grafičko sučelje. Ovo omogućuje programerima da razvijaju sigurne i robusne aplikacije u različitim programskim jezicima unutar .NET okruženja. [3]

2.1.1. Varijable

Varijable su osnovne komponente u programiranju koje služe za pohranjivanje i manipulaciju podacima. Svaka varijabla ima svoje ime i vrijednost koja se može mijenjati tijekom izvođenja programa. Varijable se koriste za različite svrhe, kao što su spremanje brojeva, teksta, nizova, objekata i drugih podataka. Unutar C#-a varijable možemo deklarirati i inicijalizirati na sljedeći način:

```
// Deklaracija varijable tipa cijelog broja (int)
int broj = 42;
// Deklaracija varijable tipa znakova (string)
string ime = "John";
// Deklaracija varijable tipa decimalnog broja (float)
float iznos = 10.50f;
// Deklaracija varijable tipa boolean (bool)
bool istina = true;
```

2.1.2. Nizovi i liste

Nizovi i liste su dvije različite strukture podatak koje se koriste za pohranu i manipulaciju skupovima podataka.

Nizovi su fiksne kolekcije elemenata istog tipa podataka i veličine su im unaprijed određene, te se ona ne može mijenjati nakon inicijalizacije. Pristup elementima niza postižemo pomoću indeksa, počevši on 0 za prvi element.

Primjer deklaracije i inicijalizacije niza:

```
int[] niz = new int[5];
niz[0] = 10;
```

U navedenom primjeru deklariramo niz cjelobrojnog tipa od 5 elemenata i na prvi element postavljamo vrijednost 10.

Liste su dinamičke kolekcije elemenata istog tipa podataka, te im veličina, za razliku od nizova, nije unaprijed određena i može se mijenjati tijekom izvođenja programa. Pristup elementima se vrši pomoću indeksa, ali one pružaju dodatne funkcionalnosti, kao što je automatsko povećanje liste i metode za manipulacijom podatcima. Neke od metoda za manipulacijom podatcima su dodavanje elementa u listu „*Add*“, brisanje elementa iz liste „*Remove*“ i brisanje svih elemenata iz liste „*Clear*“.

Primjer deklaracije i inicijalizacije liste s osnovnim metodama:

```
List<int> lista = new List<int>(); // Definira praznu listu
lista.Add(10); // Dodaje element 10 u listu
lista.Add(5); // Dodaje element 10 u listu
lista.Remove(10);
lista.Clear();
```

U navedenom primjeru deklariramo praznu listu cjelobrojnog tipa, te u nju dodajemo brojeve 10 i 5. Nakon što dodamo brojeve brišemo broj 10 i ispražnjujemo cijelu list.

2.1.3.Grananja

Grananje (engl. „*branching*“) u programiranju je proces donošenja odluka o izvršavanju određenog bloka koda na temelju uvjeta ili logičkih izraza. U C# se grananje postiže pomoću izraza „*if*“ ili „*switch*“.

Izraz „*if*“ se koristi u programiranju kada želimo izvršavati određeni blok koda u određenim uvjetima. Ako je uvjet ispunjen on vraća „*true*“ te ulazi u programski blok koji se nalazi unutar izraza „*if*“, ako uvjet nije ispunjen on vraća „*false*“ i ulazi u programski blok koji se nalazi unutar izraza „*else*“ ili samo preskoči izraz „*if*“ ako „*else*“ nije definiran.

Primjer grananja sa izrazom „if“:

```
int broj = 5;
if(broj>0)
{
      Console.WriteLine(„Broj je veći od 0“);
} else
{
      Console.WriteLine(„Broj je manji od 0“);
}
```

U navedenom primjeru provjeravamo da li je broj veći od 0. Ako je broj veći od 0 „*if*“ poprima vrijednost „*true*“ te ulazi u svoj programski blok i ispisuje na konzolu poruku „Broj je veći od 0“, a ako je broj manji od 0 „*if*“ poprima vrijednost „*false*“ i ulazi u programski blok „*else*“, te ispisuje na konzolu poruku „Broj je manji od 0“.

Izraz „*switch*“ se koristi u programiranju kada želimo izvršavati različite akcije ovisno o vrijednosti varijable. Ako vrijednost varijable odgovara jednom slučaju (engl. „*case*“) tada će se izvršiti programski blok unutar tog slučaja, ako vrijednost ne odgovara ni jednom slučaju onda se izvršava programski blok unutar „*default*“ izraza. Prekid ili stajanje izvršavanja jednog slučaja se postiže sa ključnom riječi „*break*“.

Primjer grananja sa izrazom „*switch*“:

```
int dan = 3;
```

```
switch (dan)
{
    case 1:
        Console.WriteLine("Ponedjeljak");
        break;
    case 2:
        Console.WriteLine("Utorak");
        break;
    case 3:
        Console.WriteLine("Srijeda");
        break;
    case 4:
        Console.WriteLine("Četvrtak");
        break;
    case 5:
        Console.WriteLine("Petak");
        break;
    default:
        Console.WriteLine("Vikend");
        break;
}
```

U navedenom primjeru imamo varijablu dan. Ako varijabla dan ima vrijednost koja odgovara jednom slučaju, u primjeru odgovara slučaju 3, će ispisati na konzolu dan povezan sa tim slučajem. Ako varijabla ne odgovara ni jednom slučaju, na konzolu će se ispisati „Vikend“.

2.1.4. Petlje

Petlje omogućuju da program izvrši određeni programski blok više puta. One su korisne kada želimo automatizirati ponavljajuće zadatke ili obraditi veliku količinu podataka. U c# razlikujemo „*for*“, „*foreach*“, „*while*“ i „*do-while*“ petlje.

Petlje sa izrazom „*for*“ koristimo kada znamo koliko puta želimo ponoviti određeni blok koda. Petlja se sastoji od tri dijela: inicijalizacija, uvjet i izražaj za inkrementiranje (for(inicijalizacija, uvjet, inkrement)).

Primjer „*for*“ petlje:

```
for (int i = 1; i <= 5; i++)
{
    Console.WriteLine("Broj: " + i);
}
```

U navedenom primjeru će se na konzolu ispisati pet puta „Broj: “ popraćen sa jednim od brojeva od 1 do 5.

Petlje sa izrazom „*while*“ i „*do-while*“ se koriste kada želimo ponoviti određen blok koda dok se uvjet ne ispuni. Petlja „*while*“ provjera uvjet prije svakog izvršavanja bloka koda, a „do-while“ provjerava uvjet nakon izvršavanja bloka koda što znači da će se kod izvršiti barem jednom, čak i ako uvjet nije ispunjen.

Primjer „*while*“ petlje:

```
int broj = 1;
while (broj <= 5)
{
    Console.WriteLine("Broj: " + broj);
    broj++;
}
```

Primjer „*do-while*“ petlje:

```
int broj = 1;
while (broj <= 5)
{
    Console.WriteLine("Broj: " + broj);
    broj++;
}
```

Navedeni primjeri će, isto kao i „*for*“ petlja, ispisati pet puta na konzolu „Broj: “ popraćen sa jednim od brojeva od 1 do 5.

Petlja sa izrazom „*foreach*“ se koristi za iteriranje kroz kolekcije i nizove. Ona omogućava jednostavan pristup svakom elementu kolekcije bez potrebe za praćenjem indeksa ili borja elemenata.

Primjer „*foreach*“ petlje:

```
string[] tjedan = {"Ponedjeljak", "Utorak", "Srijeda", "Četvrtak", "Petak", "Subota", "Nedjelja" };


foreach (string dan in tjedan)
{
    Console.WriteLine("Dan: " + dan);
}
```

U navedenom primjeru „*foreach*“ petlja prolazi kroz niz „tjedan“, te za svaki element niza će varijabla „dan“ sadržavati trenutni element i ispisati na konzolu „Dan: “ i trenutni element.

2.1.5.Funkcije

Funkcije predstavljaju blok koda koji se izvršava kada se funkcija pozove. Ona omogućuje bolju organizaciju i ponovnu iskoristivost koda. Funkcije mogu imati jedna od tri modifikatora vidljivosti, to su „*public*“, „*private*“ i „*protected*“. Funkcije također mogu vraćati vrijednost (ključna riječ „*return*“ unutar bloka koda funkcije) ili ne moraju (funkcije sa ključnom riječi „*void*“), te mogu primati parametre ili ne moraju. Parametri su ulazni podatci koje funkcija prima pri svom pozivu.

Primjer deklaracije funkcije koja prima parametre i vraća vrijednost, te poziva navedene funkcije:

```
public int Zbroji(int broj1, int broj2)
{
    int rezultat = broj1 + broj2;
    return rezultat;
}
Console.WriteLine(„Zbroj: „ + Zbroji(1,2));
```

U navedenom primjeru funkcija prima parametre 1 i 2, te vraća njihov zbroj i ispisuje ga na konzolu.

Primjer deklaracije funkcije bez parametara i povratne vrijednosti:

```
public void Pozdrav()
{
    Console.WriteLine("Dobro došli!");
}
```

U navedenom primjeru funkcija nema povratnu vrijednost, te samo ispisuje na konzolu poruku „Dobro došli!“.

Preopterećenje funkcija je mogućnost funkcija da imaju isto ime, ali različiti broj ili tip parametara.

Primjer Preopterećenja funkcija:

```
public int Zbroji (int broj1, int broj2)
{
    int rezultat = broj1 + broj2;
    return rezultat;
}

public float Zbroji(float broj1, float broj2)
{
    float rezultat = broj1 + broj2;
```

```
        return rezultat;
    }
```

U navedenom primjeru možemo vidjeti preopterećenje funkcije „*Zbroji*“. Jedna funkcija prima cjelobrojne vrijednosti, dok druga prima decimalne vrijednosti.

2.1.6.Klase i objekti

Klase i objekti su fundamentalni koncepti u objektno orijentiranom programiranju. Klasa je temeljna konstrukcija u OOP-u unutar koje definiramo šablonu strukture i ponašanja objekta koji se temelji na toj klasi. Klase se sastoje od atributa (varijabli) i metoda (funkcija), te svaki atribut i metoda mogu poprimiti jedan od tri modifikatora vidljivosti (ako se modifikator ne definira program postavlja kao zadano modifikator „*private*“). Atributi i metode sa modifikatorom „*private*“ su vidljive samo u klasi u kojoj se deklarirani, sa modifikatorom „*public*“ su vidljive i mogu se pozvati iz bilo koje klase i sa modifikatorom „*protected*“ se atributi i metode pretvaraju u „*private*“ kod nasljeđivanja. Klase također imaju metodu zvanu konstruktor koja se automatski poziva kada se stvara nova instanca objekta te klase. Konstruktor se koristi za inicijalizaciju atributa objekata, te on ime isti naziv kao i klasa u kojoj se nalazi.

Objekti su instance klase i koriste se za izvođenje metoda definiranih u klasi i pristup atributima tog objekta, te svaki objekt ima svoj vlastiti skup vrijednosti za atribute definirane klase. Oni se stvaraju s pomoću konstruktora klase i svaki put kada se stvori novi objekt se rezervira memorija za atribute tog objekta.

Primjer klase:

```
    public class Osoba
    {
        public string Ime;
        public int Godine;

        public Osoba(string ime, int godine)
        {
            Ime = ime;
            Godine = godine;
        }

        public void PredstaviSe()
        {
            Console.WriteLine("Moje ime je " + Ime + " i imam " + Godine +
" godina.");
        }
    }
```

U navedenom primjeru možemo vidjeti klasu sa 2 „*public*“ atributa „*Ime*“ i „*Godine*“, te konstruktorom klase i funkcijom „*PredstaviSe*“ koja pri pozivu ispisuje ime i godine spremljene u atribute objekta klase.

Primjer nasljeđivanja klase „*Osoba*“ iz gornjeg primjera i stvaranje objekata tih klasa:

```
public class Student : Osoba
{
    private string StudijskiProgram;

    public Student(string ime, int godine, string program)
        : base(ime, godine)
    {
        StudijskiProgram = program;
    }

    public void Studiraj()
    {
        Console.WriteLine(Ime + " studira na programu " + StudijskiProgram + ".");
    }
}

Osoba osoba = new Osoba("Ana", 20);
osoba.PredstaviSe();

Student student = new Student(osoba.Ime, osoba.Godine, „Informacijski i poslovni sustavi“);
student.Studiraj();
```

U primjeru možemo vidjeti nasljeđivanje klase „*Osoba*“ u klasu „*Student*“, sa „*base(ime, godine)*“ se poziva konstruktor nadklase. Klasa Student ima privatni atribut „*StudijskiProgram*“ i javnu metodu „*Studiraj*“ koja na konzolu ispisuje ime i studij spremljeni u objekt klase. Na kraju možemo vidjeti primjere stvaranja objekata klasa, te korištenje metoda klasa.

2.2. Unity grafičko sučelje

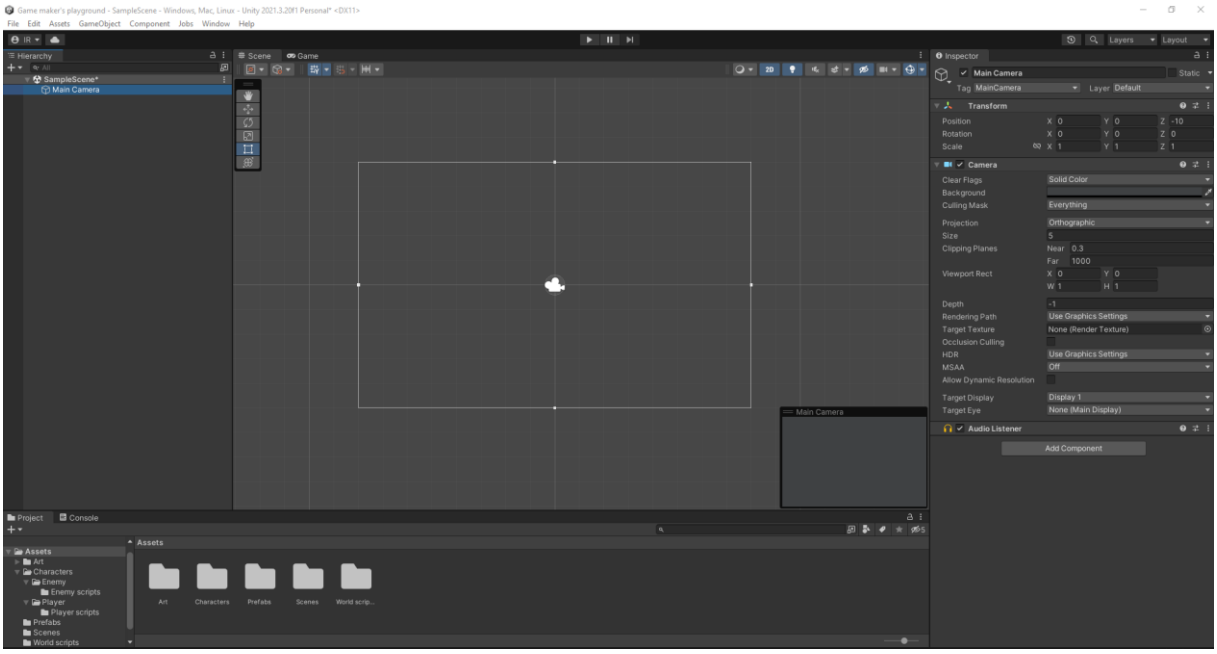
Unity grafičko sučelje se sastoji od scene i igračke hijerarhije (desna strana na slici 1), inspektora (lijeva strana na slici 1), projektnog prozora (donji dio slike 1) i prozora za prikaz (engl. „*viewport*“, srednji dio slike 1). Unity se sastoji od još elemenata grafičkog sučelja, ali ova 4 navedena elementa su prikaza kod generiranja projekta. Ostali elementi koji su korišteni tijekom izrade ovog projekta će biti spomenuti i objašnjeni u svojim zasebnim cjelinama.

Scene u Unity-u predstavljaju različite dijelove igre, tj. nivoe igre, te svaki objekt koji se postavi na scenu se može vidjeti unutar igračke hijerarhije. Objekti se na scenu učitavaju sa vrha igračke hijerarhije prema dnu.

Unutar inspektora možemo vidjeti komponente koje su povezani na odabrani objekt. Unutar njega možemo i dodavati ili brisati komponente, te modificirati postavljene komponente. Većina objekata koje se dodaju na scenu poprimaju komponentu „Transform“ pomoću koje možemo pozicionirati i skalirati odabrani objekt.

Projektni prozor služi za organizaciju resursa (tekstura, objekata, fontova, zvukova, animacija) koji nam se nalaze u projektu. Unutar njega možemo generirati direktorije za bolju organizaciju resursa.

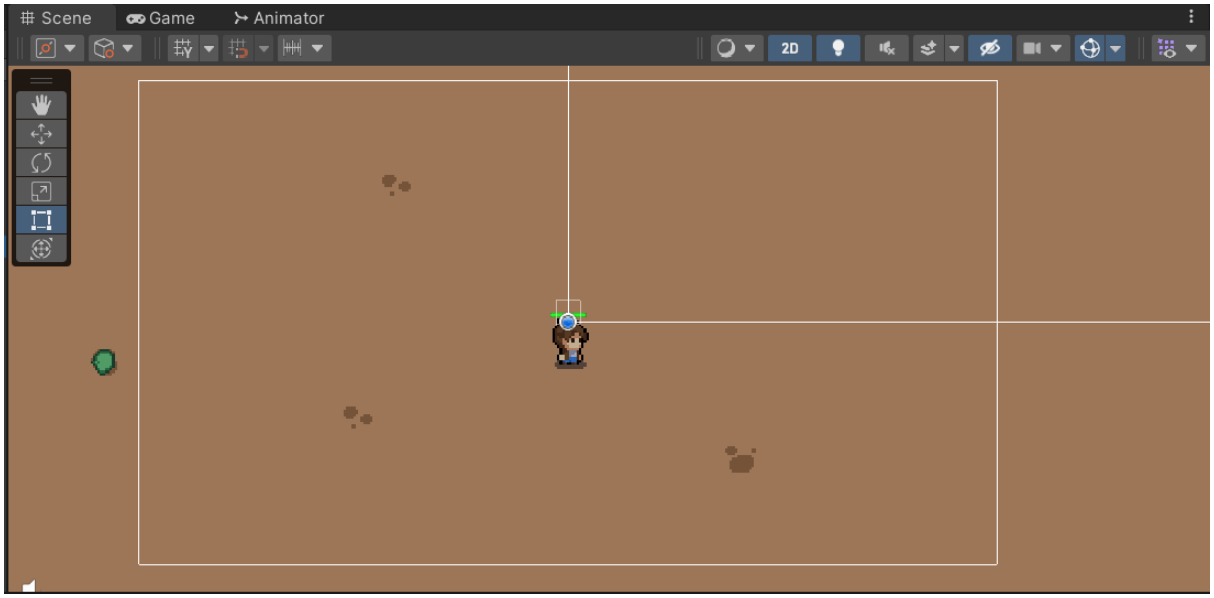
Prozor za prikaz je prozor unutar kojeg možemo vidjeti kako igra izgleda tijekom njenog razvijana, što omogućuje vizualnu provjeru video igre.



Slika 1. Prikaz grafičkog sučelja Unity-a (vlastita izrada)

2.3. Scene

Scene služe kao jedan nivo unutar video igre. Svaka novo kreirana scena se generira sa kamerom koja služi za prikaz resursa koji se nalaze na njoj. U scenu možemo postaviti objekte, teren, svjetla, kamere i ostale elemente od kojih je sastavljena video igra. Unutar nje možemo i vizualno vidjeti raspored elemenata na njoj te isto tako ih možemo i raspoređivati što olakšava rad. [2] Na slici 2 možemo vidjeti jednostavnu scenu na kojoj je prikazan teren i objekt igrača.

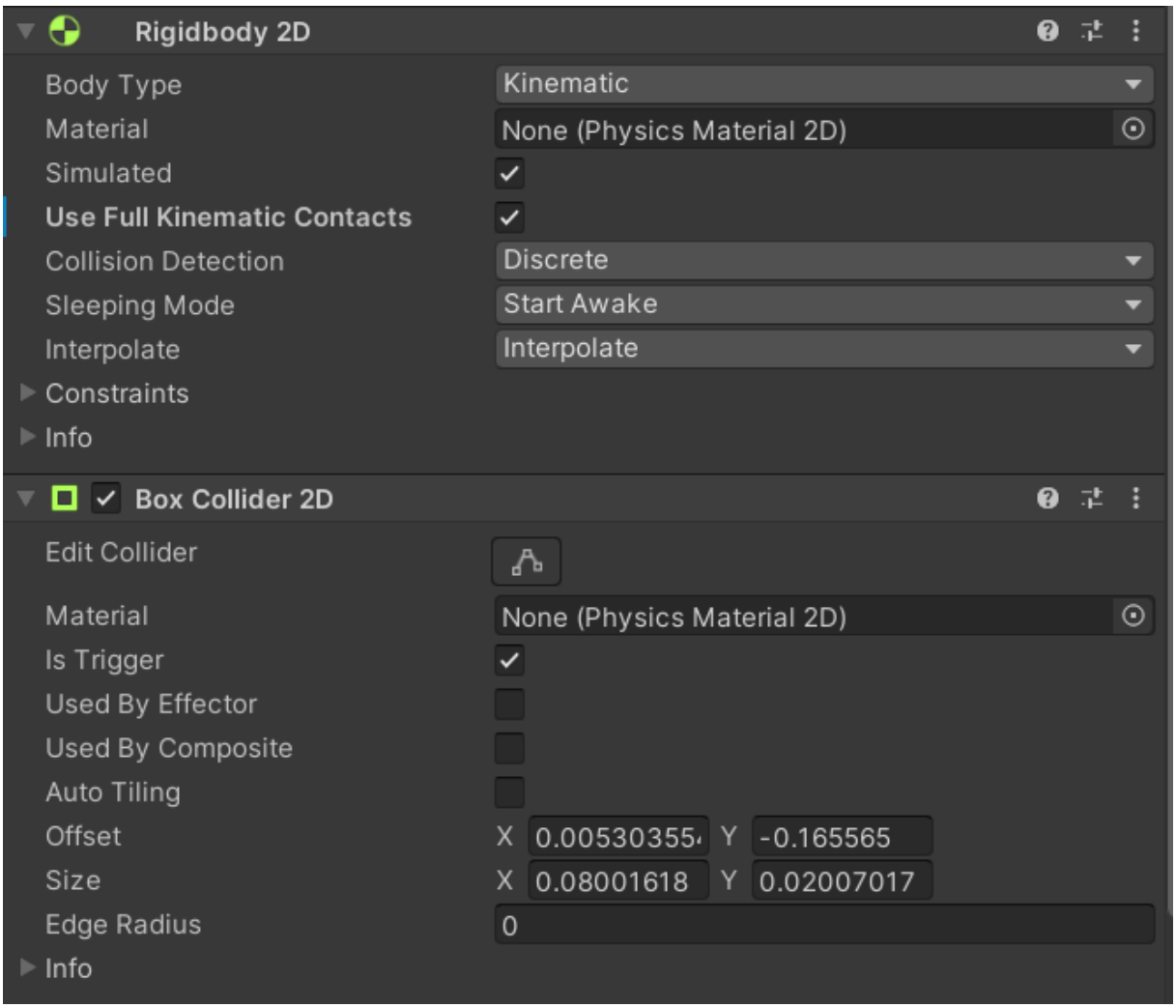


Slika 2. Izgled scene u Unity-u (vlastita izrada)

2.4. Collider i RigidBody2D

Sudarne komponente (engl. „*Collider*“) su komponente koje se koriste za definiranje oblika i površine objekta u igri, kako bi omogućili detekciju sudara i fizičku interakciju između objekata. Glavna svrha sudarnih komponenti je da simulira stvarni prostor objekta unutar virtualnog svijeta, npr. da objekt koji predstavlja led se ponaša kao led, tj. da bude sklizak. Unity ima različite vrste sudarnih komponenti, uključujući kvadratni oblik (engl. „*Box Collider*“), oblik sfere (engl. „*Sphere Collider*“), oblik mreže (engl. „*Mesh Collider*“) i drugi. Sudarne komponente se mogu postaviti da se ponašaju kao okidači (engl. „*Trigger*“), oni ne uzrokuju fizičke sudare, već se koriste za detekciju prolaska jednog objekta kroz drugi. [6]

RigidBody2D služi za simulaciju fizike nad objektom, kao što su gravitacija i brzina, te omogućuje koliziju između dve objekta sa sudarnim komponentama. Unutar komponente RigidBody2D možemo izabrati 1 od 3 vrste tipa tijela, statičko (engl. „*static*“), kinematičko (engl. „*kinematic*“) i dinamičko (engl. „*dynamic*“). Statički tip služi kod objekata koji su nepomični i ne reagiraju na sile/sudare. Kinematički tip služi za objekte koji se kontroliraju kroz skripte ali ne reagiraju na vanjske sile. Dinamički tip služi za objekte koji reagiraju na sile, kao što je gravitacija i sudari. [6]

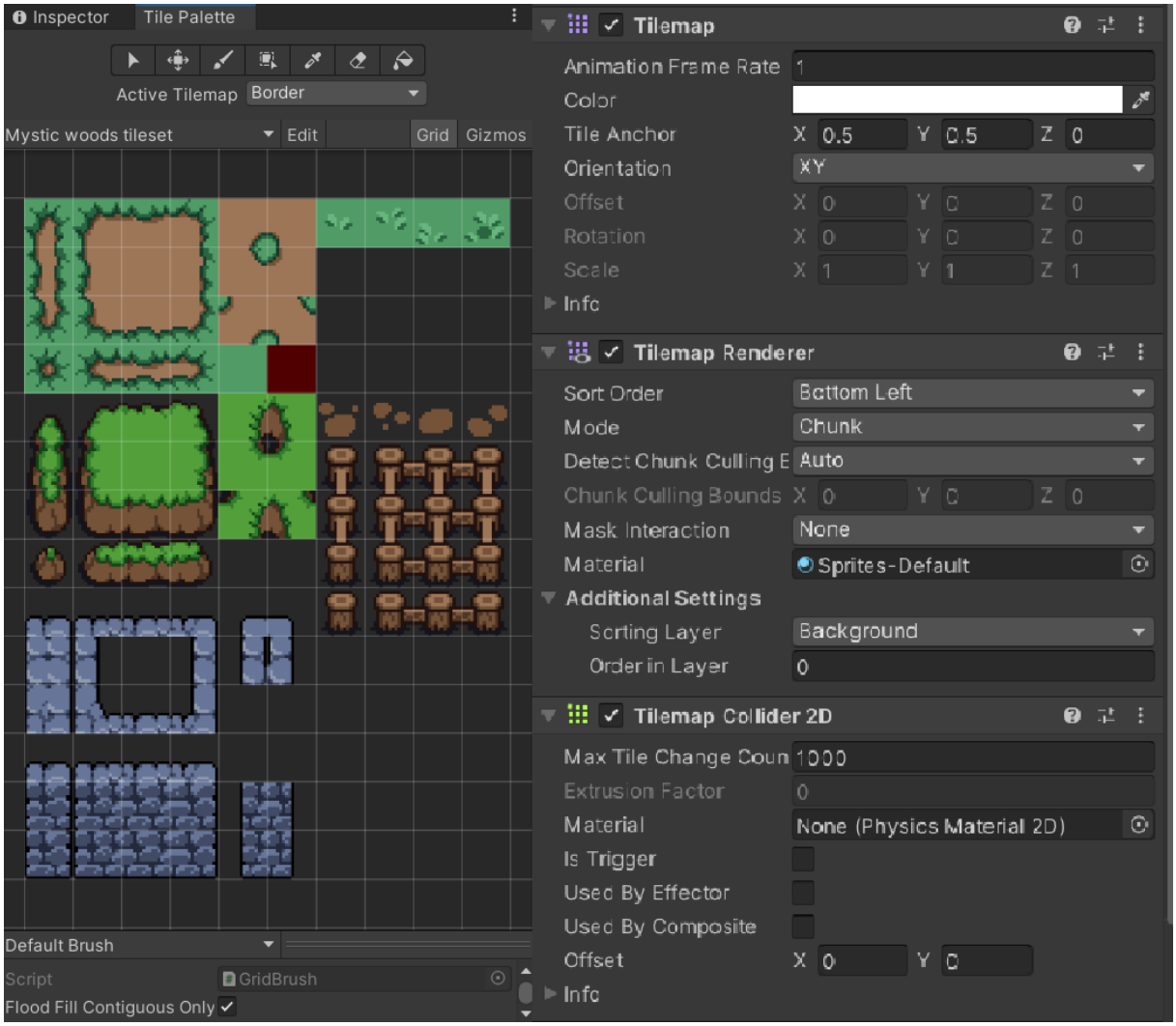


Slika 3. Prikaz Collider i RigidBody2D komponenti (vlastita izrada)

2.5. Grid, Tilemap i Tile palette

Grid komponenta služi kao vodič za lakše poravnavanje objekata, kao što su pločice (engl. „*Tiles*“), po nekom rasporedu na scenu. [4] „*Tilemap*“ je komponenta pomoću koje

možemo pohraniti 2D resurse za kreiranje terena. Ona u kombinaciji sa Grid komponentom olakšava crtanje i generiranje terena. Uz „*tilemap*“ dolazi i komponente „*Tile Renderer*“ unutar koje specificiramo kako će se koji sloj prikazivati, te „*Tile palette*“ koja služi kao plata koja sadrži sve pločice za bojanje po grid-u. Također postoji i komponenta „*Tilemap Collider 2D*“ koja može služiti kao zid unutar video igre.[5]



Slika 4. Prikaz Tilemap komponenti i Tile Palette-a (vlastita izrada)

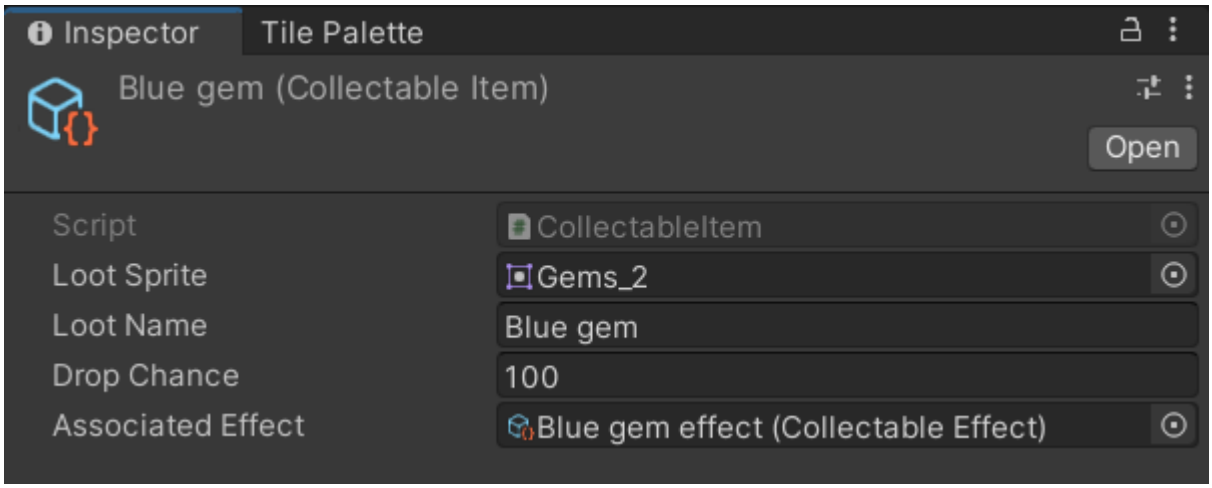
2.6. ScriptableObject

ScriptableObject je kontejner za pohranu podataka koji smanjuje upotrebu memorije u projektima tako da se izbjegava kopiranje podataka. On se može koristiti za spremanje podataka kao resurse koji se dijele između različitih instanci istog objekta. Podatke unutar ScriptableObject-a možemo uređivati i pomoću Unity uređivača (slika 5) [7].

Primjer ScriptableObject-a „*collectable item*“ iz projekta:

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

[CreateAssetMenu(fileName = "New item", menuName = "Collectable item")]

public class CollectableItem : ScriptableObject
{
    public Sprite lootSprite;
    public string lootName;
    public int dropChance;
    public CollectableEffect associatedEffect;

    public CollectableItem(string lootName, int dropChance)
    {
        this.lootName = lootName;
        this.dropChance = dropChance;
    }

    public void ApplyEffectToPlayer()
    {
        if (associatedEffect != null)
        {
            associatedEffect.ApplyEffect();
        }
        else
        {
            Debug.LogWarning("No associated effect for this item: " +
lootName);
        }
    }
}
```

U primjeru možemo vidjeti ScriptableObject koji sprema sliku objekta (engl. „*Sprite*“), znakovni niz, cijeli broj i također može spremiti jedan ScriptableObject.



Slika 5. Prikaz izgleda ScriptableObject-a unutar Unity uređivača (vlastita izrada)

2.7. PlayerPrefs

PlayerPrefs je Unity-eva komponenta koja omogućuje jednostavno spremanje i čitanje podataka između različitih sesija igre. PlayerPrefs omogućuje spremanje nizova znakova, cijele brojeve ili booleanske vrijednosti, te ih on sprema lokalno bez enkripcije podataka. PlayerPrefs su dobri za spremanje personaliziranih postavki igre ili jednostavne pohrane video igre. PlayerPrefs koristi metode „*SetInt*“, „*SetString*“ i „*SetBool*“ za postavljanje podataka i „*Save*“ za spremanje podataka, te „*GetInt*“, „*GetString*“ i „*GetBool*“ za dohvaćanje podataka.[8]

Primjer spremanja i dohvaćanja niza znakova u PlayerPrefs:

```
PlayerPrefs.SetString("Pozdrav", poruka);
PlayerPrefs.Save();
Debug.Log(PlayerPrefs.GetString("Pozdrav"));
```

2.8. Sprite animations

Animacije slika (engl. „*Sprite animations*“) su animacijski isječci koji se stvaraju za 2D resurse. Animacijski isječci se mogu raditi na različite načine i jedan od njih je sa listom slika (engl. „*Sprite Sheet*“). [9] Lista slika je skupina sličica vezanih uz jedan objekt, te sličice uglavnom prate istu temu slike, naprimjer lista slika čovjeka će sadržavati sličice koje sadrže istu boju kose, odjeće, te jedine promjene su radnja koju taj čovjek izvršava (hoda, trči umire, napada i slično) (slika 6).

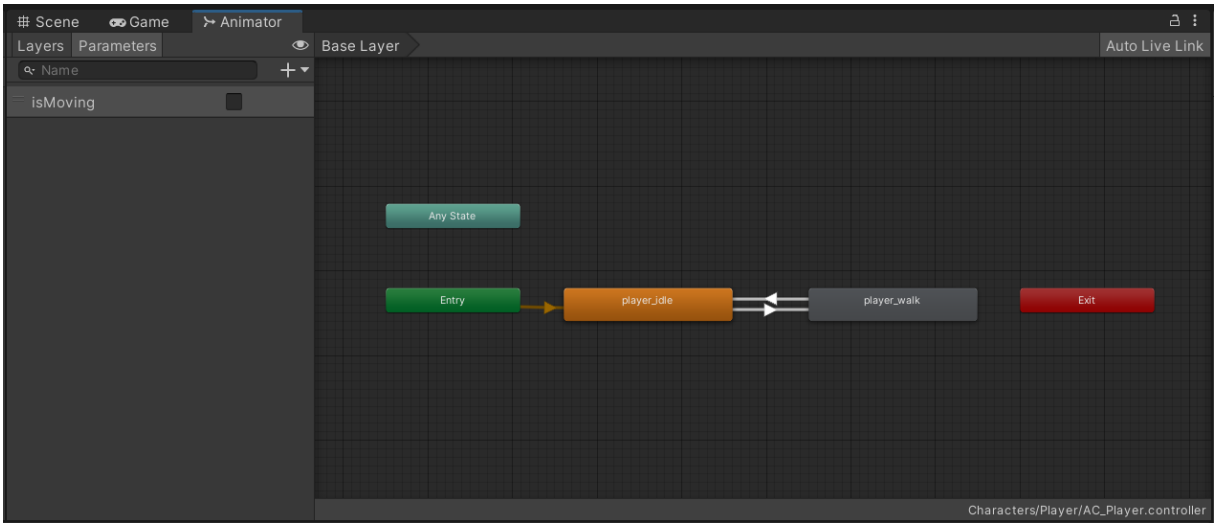
Slika 6. Primjer liste slika (vlastita izrada)

Za animiranje objekta pomoću sličica možemo koristiti grafičko sučelje od Unity-a, prozor za animacije (slika 7). Za animiranje trebamo napraviti animacijski isječak, te onda

možemo sličice iz liste slika povući i ispustit (engl. „*drag and drop*“) u desni dio prozora za animacije na slici 7.

Slika 7. Prikaz prozora za animacije (vlastita izrada)

Kada napravimo animacijske isječke možemo unutar Unity-ovog grafičkog sučelja Unity Animator (slika 8) upravljati redoslijedom izvođenja animacijskih isječaka. Svaki animacijski isječak se pri kreiranju sam doda u animator i svi animacijski isječci i animator su povezani uz svoj objekt. Unutar animatora sa desnim klikom na element možemo napraviti prijelaz (engl. „*Transition*“) sa jedne animacije na drugu što olakšava upravljanjem ponašanjem i izgledom objekta tijekom igre. Animator također nudi opciju za kreiranje parametara pomoću kojih možemo koristiti za kontrolu animacija. Na slici 8 možemo vidjeti parametar „*isMoving*“ koji služi za provjeru da li se igrač kreće te kod ispunjenog uvjeta pokreće animaciju „*player_walk*“.



Slika 8. Prikaz Unity Animator-a (vlastita izrada)

3. Razvoj video igre

3.1. Opis video igre

Video igra izrađena za ovaj završni rad je roguelike shoot-em up igra nazvana Gladiator arena. Ona se sastoji od 2 scene, scene glavnog izbornika i scene nivoa. Igra se sastoji od 6 neprijatelja, 4 obična neprijatelja i 2 šefa neprijatelja. Cilj igre je preživjeti što duže uz konstantno generirajuće valove neprijatelja. Igra sadrži mehaniku povećavanje razine igrača (engl. „*leveling*“) koja pri svakom povećanju razine nudi igraču da poboljša svoje statistike. Pri gubitku igre igraču je ponuđeno spremanje svog postignuća, te se na tablicu rezultata (engl. „*Scoreboard*“) prikazuje 5 najboljih postignuća.

3.2. Scene igre

3.2.1. Glavni izbornik

Glavni izbornik (slika 9) igre sa sastoji od platna (engl. „*Canvas*“) koji u svojoj hijerarhiji ima 4 djeteta. Svako dijete, osim „*BackGround*“ koji služi za postavljanje pozadine, mijenja izgled sučelja.

3.2.1.1. MainMenu panel

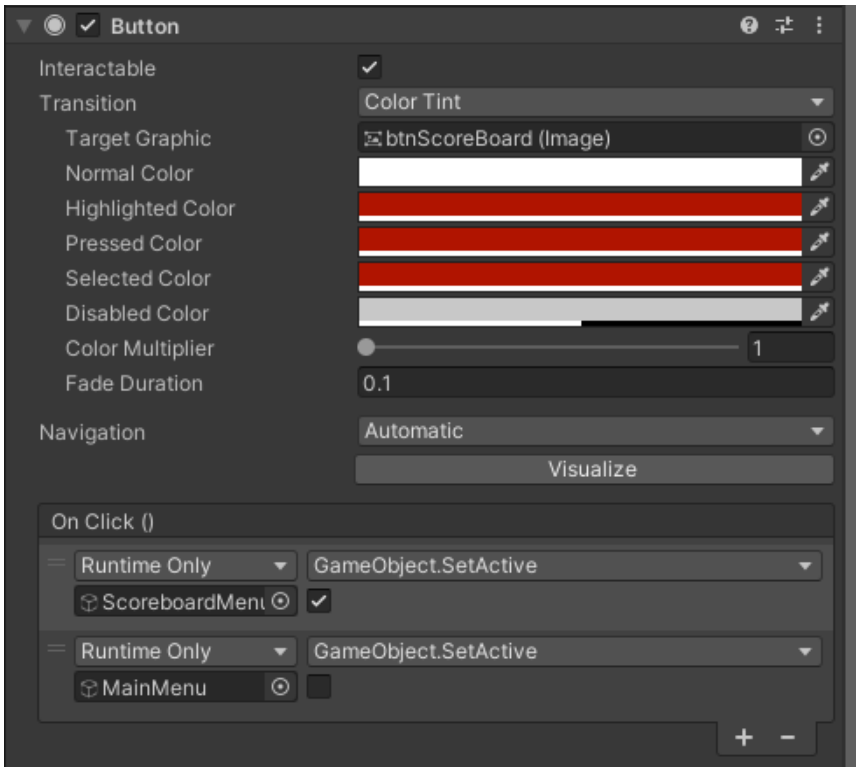
Slika 9. Prikaz glavnog izbornika i njegove hijerarhije (vlastita izrada)

Na slici 9 vidimo dijete „*MainMenu*“ koje se sastoji od naslova Gladiator arena i 4 gumba. Klikom na gumb „*Play*“ se pokreće nivo igre uz pomoć metode „*PlayGame*“ koja se nalazi u kodu ispod odlomka. Klikom na gumb „*Scoreboard*“ se stanje „*isActive*“ djeteta

„*MainMenu*“ postavlja na „*false*“, a stanje djeteta „*ScoreboardMenu*“ na „*true*“. Isto tako klikom na gumb „*Info*“ se stanje djeteta „*MainMenu*“ postavlja na „*false*“, a stanje djeteta „*InfoMenu*“ na „*true*“. Klikom na gumb „*Quit*“ se igra gasi uz pomoć metode „*QuitGame*“ koja se nalazi u kodu ispod odlomka.

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

public class MainMenuController : MonoBehaviour
{
    public void PlayGame()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex+1);
    }

    public void QuitGame()
    {
        Debug.Log("Quit");
        Application.Quit();
    }
}
```

Za postavljanje funkcionalnosti gumba trebamo ući u inspektor objekta gumba i pod komponentom „*Button*“ u „*On Click ()*“ dijelu dodijeliti ponašanje gumbu (slika 10).



Slika 10. Prikaz komponente "Button" u inspektoru (vlastita izrada)

3.2.1.2. InfoMenu panel

Slika 11 prikazuje dijete „*InfoMenu*“ koje se sastoji od naslova „*Info*“, tekstnog okvira koji navodi kontrole video igre, te gumb „*Back*“ koji postavlja „*InfoMenu*“ panel na „*false*“, a „*MainMenu*“ panel na „*true*“.

Slika 11. Prikaz "Info menu" panela (vlastita izrada)

3.2.1.3. ScoreboardMenu panel

Slika 12 prikazuje „*ScoreboardMenu*“ panel koji se sastoji od naslova „*Scoreboard*“, gumba „*Back*“ koji radi na isti način kao i na „*InfoMenu*“ panelu, te panela koji služi za prikaz 5 najboljih postignuća (na slici 12 se trenutno vidi samo template postignuća).

Slika 12. Prikaz "Scoreboard menu" panela (vlastita izrada)

Za prikaz postignuća koristim skriptu koja je razbijena u nekoliko metoda radi bolje organizacije i vidljivosti. Ta skripta se poziva kod ulaska na panel „*ScoreboardMenu*“. Skripta sva postignuća sprema unutar PlayerPrefs u Json obliku.

Kako bi osigurao da skripta može spremiti podatke unutar PlayerPrefs-a sam postavio jednog igrača sa najgorim mogućim postignućem. Taj igrač se neće nikada prikazati jer se on samo postavi u Json formatu, te kada se napravi prvi pravi upis u tablicu rezultata on se prebriše i upiše se novi rezultat. Postavljanje inicijalnog igrača sam napravio sa metodom „ *InitialScore*“:

```
private void InitialScore()
    {
        string hardSetPlayer = "{\"score\":0,\"name\":\"BestGamer\"}";
        PlayerPrefs.SetString("ScoreboardTable", hardSetPlayer);
    }
```

Dodavanje novog postignuća u tablicu se postiže metodom „*AddScoreboardEntry*“ koja prima dva parametra. Prvi parametar predstavlja postignuto vrijeme, a drugi kao naziv igrača. Ta dva parametra se spremaju u objekt klase „*ScoreboardEntry*“ koja služi kao template za podatke. Nakon što se spreme podatci pozivaju se spremljeni podatci iz PlayerPrefs te ih se pretvara iz Json formata i sprema u instancu objekta klase „*Scoreboard*“ koja sadrži listu formata „*ScoreboardEntry*“, te se u taj objekt dodaje novo postignuće. Nakon što se doda novo postignuće u listu, ona se sortira metodom „*SortList*“ koja koristi algoritam mjehuričastog sortiranja (engl. „*Bubble sort*“), te onda se uklanjaju sva postignuća koja su lošija od petog upisanog postignuća metodom „*RemoveExcessScores*“. Kada se sve to izvrši lista se postavlja u Json format, te ga sprema u PlayerPrefs.

Metoda „*AddScoreboardEntry*“:

```
public void AddScoreboardEntry(float score, string name)
    {
        ScoreboardEntry scoreboardEntry = new ScoreboardEntry { score
= score, name = name };
        string jsonString = PlayerPrefs.GetString("ScoreboardTable");
        Scoreboard scoreboard =
JsonUtility.FromJson<Scoreboard>(jsonString);

        scoreboard.scoreboardEntryList.Add(scoreboardEntry);
        SortList(scoreboard.scoreboardEntryList);
        RemoveExcessScores(scoreboard.scoreboardEntryList);

        string json = JsonUtility.ToJson(scoreboard);
        PlayerPrefs.SetString("ScoreboardTable", json);
```

```
        PlayerPrefs.Save();
    }
```

Metoda „*SortList*“:

```
private void SortList(List<ScoreboardEntry> scoreList)
    {
        for (int i = 0; i < scoreList.Count; i++)
        {
            for (int j = i + 1; j < scoreList.Count; j++)
            {
                if (scoreList[j].score > scoreList[i].score)
                {
                    ScoreboardEntry tmp = scoreList[i];
                    scoreList[i] = scoreList[j];
                    scoreList[j] = tmp;
                }
            }
        }
    }
```

Metoda „*RemoveExcessScores*“:

```
private void RemoveExcessScores(List<ScoreboardEntry> scoreList)
    {
        if (scoreList.Count > 5)
        {
            for (int h = scoreList.Count; h > 5; h--)
            {
                scoreList.RemoveAt(5);
            }
        }
    }
```

Klase „*Scoreboard*“ i „*ScoreboardEntry*“:

```
private class Scoreboard
    {
        public List<ScoreboardEntry> scoreboardEntryList;
    }

    [System.Serializable]
    private class ScoreboardEntry
    {
        public float score;
```

```
        public string name;
    }
```

Prikazivanje postignuća u tablici postižem metodom „*CreateScoreboardEntryTransform*“. Ta metoda prima 3 parametra, jedan parametar predstavlja 1 postignuće, drugi kontejner u kojem će se postignuća prikazivati, te treći predstavlja listu pozicija na kojim će se postignuća generirati. U metodi prvo dohvaćamo poziciju kontejnera, te onda postavljam poziciju na kojoj će se generirat postignuće. Nakon što se postave pozicije prikažem objekt postignuća i upisujem u njega postignuti rank, postignuto vrijeme i naziv igrača. Kad se sve to izvrši dodajem postignuće u listu pozicija kako bi sljedeće postignuće mogao pozicionirat na novoj poziciji.

```
    private void CreateScoreboardEntryTransform(ScoreboardEntry scoreboardEntry, Transform container, List<Transform> transformList)
    {
        float templateHeight = 100f;

        Transform entryTransform = Instantiate(entryTemplate, container);
        RectTransform entryRectTransform = entryTransform.GetComponent<RectTransform>();
        entryRectTransform.anchoredPosition = new Vector2(0, -templateHeight * transformList.Count);
        entryTransform.gameObject.SetActive(true);

        int rank = transformList.Count + 1;
        string rankString;
        switch (rank)
        {
            case 1: rankString = "1ST"; break;
            case 2: rankString = "2ND"; break;
            case 3: rankString = "3RD"; break;
            default: rankString = rank + "TH"; break;
        }

entryTransform.Find("txtPosition").GetComponent<TextMeshProUGUI>().text = rankString;

        float score = scoreboardEntry.score;

entryTransform.Find("txtScore").GetComponent<TextMeshProUGUI>().text = score.ToString();

        string name = scoreboardEntry.name;
```

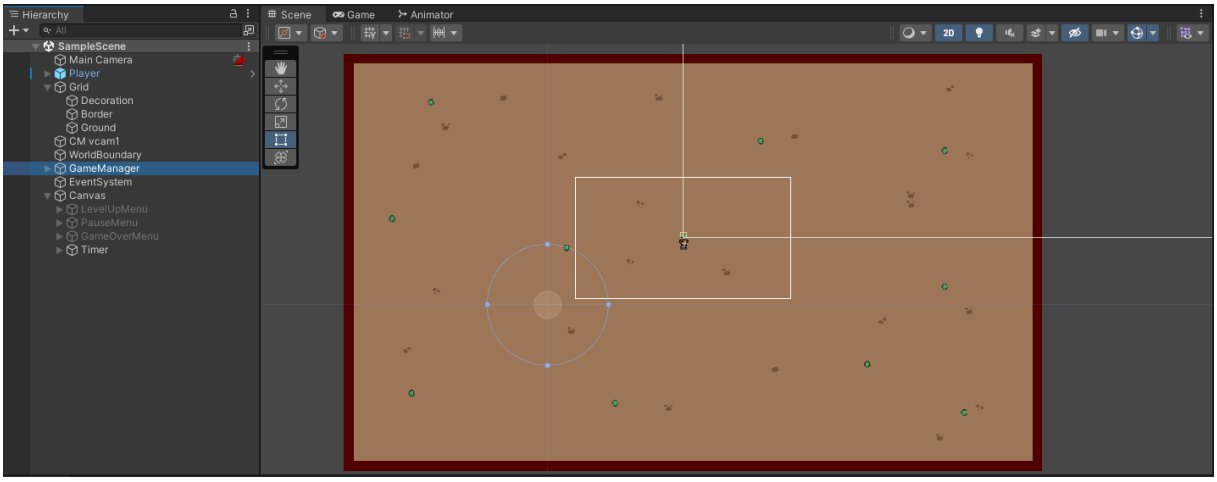
```
entryTransform.Find("txtName").GetComponent<TextMeshProUGUI>().text = name;

            transformList.Add(entryTransform);
        }
```

Metoda „*Awake*“ je metoda koja se poziva samo jednom kada se objekt učita (metoda „*Start*“ se također poziva samo jednom, ali metoda „*Awake*“ se poziva prije metode „*Start*“). Unutar ove metode sam postavio inicijalizaciju objekata koji sadrže kontejner i šablonu postignuća. Nakon što se dohvate ta 2 objekta se dohvaćaju podatci iz PlayerPrefs-a, te se radi provjera da li je PlayerPrefs prazan, ako je poziva se metoda „*InitialScore*“. Nakon provjere se podatci pretvaraju iz Json formata u ScoreboardEntry format te se onda izvršava „*foreach*“ petlja koja prolazi kroz sva postignuća i ispisuje ih sa metodom „*CreateScoreboardEntryTransform*“ na ekran.

```
        public Transform entryContainer;
        public Transform entryTemplate;
        private List<Transform> scoreboardEntryTransformList;
        private void Awake()
        {
            entryContainer = transform.Find("ScoreboardEntryContainer");
            entryTemplate                                                    =
entryContainer.Find("ScoreboardEntryTemplate");

            entryTemplate.gameObject.SetActive(false);
            string checkPrefs = PlayerPrefs.GetString("ScoreboardTable");
            if(checkPrefs.Length==0)
            {
                InitialScore();

            }
            Debug.Log(PlayerPrefs.GetString("ScoreboardTable"));

            string jsonString = PlayerPrefs.GetString("ScoreboardTable");
            Scoreboard                    scoreboard                    =
JsonUtility.FromJson<Scoreboard>(jsonString);
            scoreboardEntryTransformList = new List<Transform>();
            foreach (var entry in scoreboard.scoreboardEntryList)
            {
                CreateScoreboardEntryTransform(entry,      entryContainer,
scoreboardEntryTransformList);
            }
        }
```

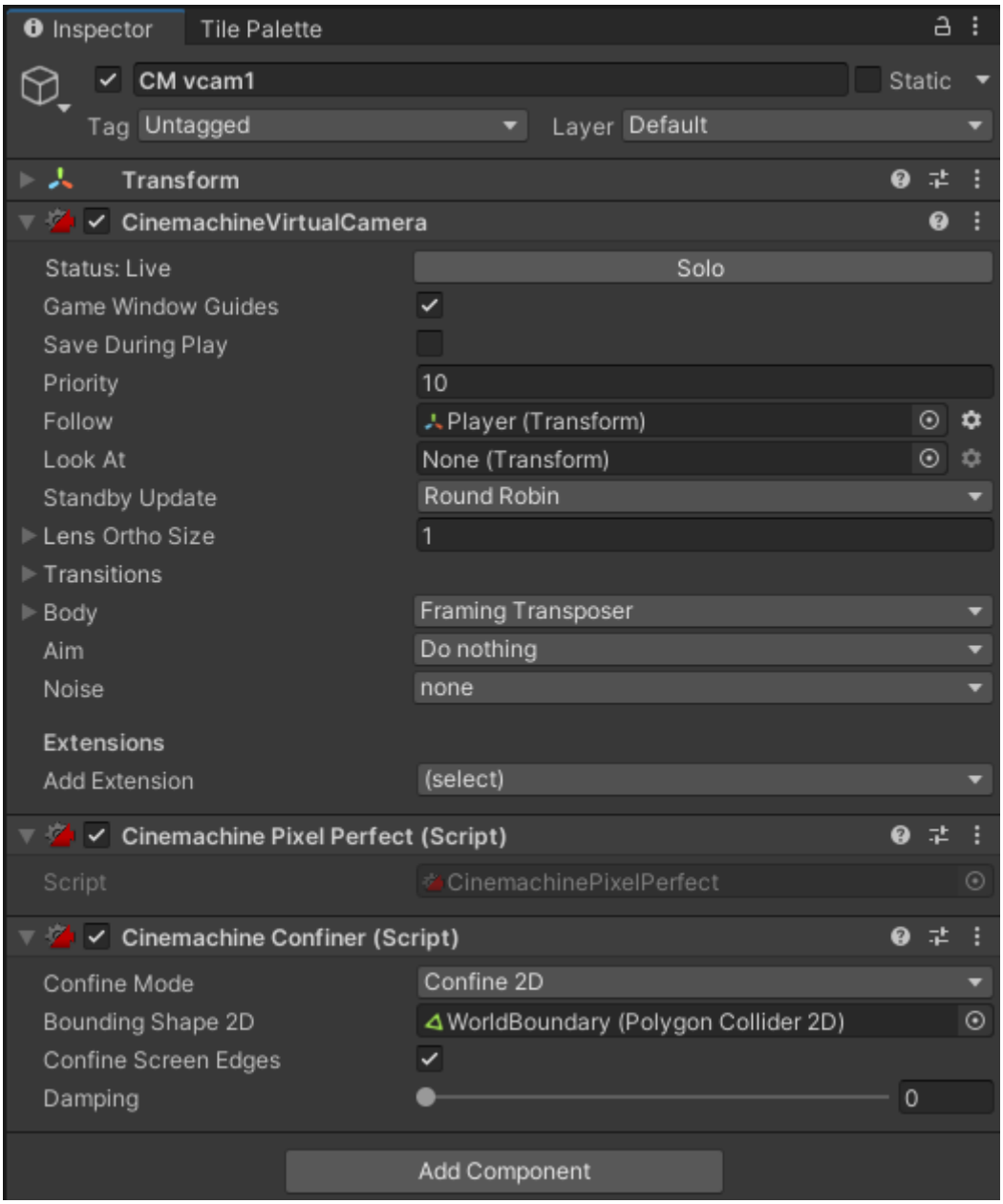
3.2.2.Scena nivoa

Scena nivoa služi za igranje video igre, te se sve mehanike i logika video igre nalaze u njoj. Scena nivoa naspram velikog broja funkcionalnosti koje sadrži ima jednostavnu hijerarhiju objekata (slika 13), te se svi ostali objekti potrebni za igranje video igre generiraju pomoću skripta. Najvažniji objekt unutar ove scene bi bio „*GameManager*“, jer on sadrži sve skripte koje se odnose na događaje vezane za nivo, kao što je skripta vezana za stvaranje valova neprijatelja, skripta za poboljšanje razine, skripte za otvaranja panela pauzirane igre i panela gubitka igre, te skripte praćenja postignuća igrača.



Slika 13. Prikaz scene nivoa i njegove hijerarhije (vlastita izrada)

Na slici 13 se mogu vidjeti i ostali objekti, ostali važni objekti bi bili Grid, „*WorldBoundary*“ i „*Canvas*“. Grid sadrži „*Tilemap*“ komponentu i služi kao pozadina nivoa, a „Canvas“ sadrži četiri panela, „*LevelUpMenu*“ panel, „*PauseMenu*“ panel, „*GameOverMenu*“ panel i „*Timer*“ panel. „*WorldBoundary*“ objekt služi kao granica koju igrač ne može proći, te isto tako služi kao granica i za kameru. Za postavljanje granice na kameru koristio sam „*Cinemachine*“ paket kojeg sam postavio na objekt „*CM vcam1*“. To je paket koji ima dodaje mogućnosti kameri unutar Unity-a i on sadrži komponentu „Cinemachine Confiner“ unutar kojeg sam postavio da je objekt „WorldBoundary“ granica kameri (slika 14).



Slika 14. Prikaz "Cinemachine" komponenti (vlastita izrada)

3.2.2.1. ScriptableObject „CharacterStats“

Ovaj ScriptableObject služi za postavljanje statistika igrača i svih neprijatelja. On se sastoji od nekoliko ključnih metoda i varijabli. On ima 2 skupine varijabli, jedna služi kao skupina koja sadrži prvobitne statistike, dok druga služi za statistike koje se mijenjaju i koriste tijekom igre.

```
//Osnovne statistike
[SerializeField]
private float baseMaxHealth;
[SerializeField]
private float baseDamage;
[SerializeField]
private float baseMovementSpeed;
[SerializeField]
private float baseCollisionOffset;
[SerializeField]
public float baseInvincibilityFrames;
[SerializeField]
public float baseCollectionRadius;
[SerializeField]
public GameObject projectile;
[SerializeField]
public float baseProjectileSpeed;
[SerializeField]
public float baseFireRate;
[SerializeField]
public float baseAttackRange;

//Promjenjive statistike
private float maxHealth;
private float damage;
private float movementSpeed;
private float collisionOffset;
private float invincibilityFrames;
private float collectionRadius;
private float projectileSpeed;
private float fireRate;
private float attackRange;

public float MaxHealth => maxHealth;
```

```
    public float Damage => damage;
    public float MovementSpeed => movementSpeed;
    public float CollisionOffset => collisionOffset;
    public float InvincibilityFrames => invincibilityFrames;
    public float CollectionRadius => collectionRadius;
    public float ProjectileSpeed => projectileSpeed;
    public float FireRate => fireRate;
    public float AttackRange => attackRange;
```

Najvažnija metoda u ovoj skripti je „*InitializeStats*“, jer pomoću nje postavljamo vrijednosti iz osnovnih statistika u promjenjive statistike. Kada bi promjenjive statistike ostale prazne igra nebi bila igriva jer sve skripte rade sa tim varijablama.

```
public void InitializeStats()
    {
        maxHealth = baseMaxHealth;
        damage = baseDamage;
        movementSpeed = baseMovementSpeed;
        collisionOffset = baseCollisionOffset;
        invincibilityFrames = baseInvincibilityFrames;
        collectionRadius = baseCollectionRadius;
        projectileSpeed = baseProjectileSpeed;
        fireRate = baseFireRate;
        attackRange = baseAttackRange;
    }
```

Metode „*IncreaseStats*“ i „*StatChecker*“ služe za mijenjanje statistika. Metoda „*IncreaseStats*“ mijenja statistike, dok „*StatChecker*“ osigurava da statistike neće prijeći određenu količinu. Metoda „*StatChecker*“ je povezana samo sa igraćom kako bi se osigurala da igrač ne postane prejak. Neprijatelji se mogu neograničeno ojačavati.

```
public void IncreaseStats(float healthIncrease, float damageIncrease,
        float collectionRadiusIncrease, float projectileSpeedIncrease,
        float fireRateIncrease, float attackRangeIncrease)
    {
        maxHealth += healthIncrease;
        damage += damageIncrease;
        collectionRadius += collectionRadiusIncrease;
        projectileSpeed += projectileSpeedIncrease;
        fireRate += fireRateIncrease;
        attackRange += attackRangeIncrease;
    }
```

```
public void StatChecker()
{
    if (maxHealth > 50) maxHealth = 50;
    if (damage > 10) damage = 10;
    if (collectionRadius > 5) collectionRadius = 5;
    if (projectileSpeed > 5) projectileSpeed = 5;
    if (fireRate < 0.2f) fireRate = 0.2f;
    if(attackRange > 3) attackRange = 3;
}
```

3.2.2.2. Skripta za stvaranje valova neprijatelja

Ova skripta ima funkcionalnost stvaranja valova neprijatelja i to postiže sa koordiniranim metodama (engl. „*Coroutine*“). Koordinirane metode omogućavaju izvršavanje bloka koda kroz određeno vrijeme. Unutar metode „Awake“ skripta inicijalizira statistike neprijatelja na njihovu prvo definiranu vrijednost koja je postavljena unutar ScriptableObject-a.

```
public float spawnRate = 1f;
    public float timeBetweenWaves = 5f;

    public int enemyCount;
    public int waveCount = 1;
    private float health=2f;
    private float damage=0.2f;

    public GameObject[] enemies;
    public GameObject[] bossEnemies;
    public Transform[] spawnPoints;
    public CharacterStats[] enemyStats;

    bool waveIsDone = true;

    private void Awake()
    {
        foreach(var enemy in enemyStats)
        {
            enemy.InitializeStats();
        }
    }
```

„spawnWave“ metoda je koordiniranim metoda koja služi za stvaranje neprijatelja i ona prima 2 parametra, jedan za broj neprijatelja koji će se stvorit i drugi tip neprijatelja koji će se

stvoriti. Prvo se izvrši stvaranje neprijatelja u „*for*“ petlji koja se izvršava sve dok ne stvori potreban broj neprijatelja. Unutar te petlje prvo se izabire nasumičan neprijatelj iz liste tipa neprijatelja, te se onda nasumično bira pozicija iz liste pozicija gdje će se neprijatelj stvoriti i na kraju se instancira objekt neprijatelja. Kada završi „*for*“ petlja smanjuje vrijeme potrebno za stvaranje neprijatelja, te se povećava broj neprijatelja koji se stvaraju u valu i broj vala. Ako je val djeljiv sa brojem 3 poziva se metoda „*increaseEnemyStats*“ s kojom se povećavaju statistike neprijatelja.

```
IEnumerator spawnWave(int count, GameObject[] enemy)
    {
        for (int i = 0; i < count; i++)
        {
            GameObject randomEnemy = enemy[Random.Range(0, enemy.Length)];
            Transform randomSpawner = spawnPoints[Random.Range(0, spawnPoints.Length)];
            Instantiate(randomEnemy, randomSpawner.position, Quaternion.identity);

            yield return new WaitForSeconds(spawnRate);
        }

        if(spawnRate > 0.5f) spawnRate -= 0.01f;
        enemyCount += 1;
        waveCount++;
        if (waveCount % 3 == 0)
        {
            increaseEnemyStats();
        }

        yield return new WaitForSeconds(timeBetweenWaves);
        waveIsDone = true;

    }

    private void increaseEnemyStats()
    {
        foreach(CharacterStats enemy in enemyStats)
        {
            enemy.IncreaseStats(health, damage, 0, 0, 0, 0);
        }
```

```
    }
```

Metoda „*FixedUpdate*“ se poziva svakih određenih nekoliko sličica u sekundi, te se u njemu nalazi uvjet koji provjerava da li je varijabla „*waveIsDone*“ postavljena na „*true*“, te ako je omogućuje stvaranje novog vala. Metoda „*waveSpawner*“ postavlja varijablu „*waveIsDone*“ na „*false*“ i provjerava da li je val djeljiv s brojem 5, ako je stvara val šefa neprijatelja inače stvara obične neprijatelje.

```
private void FixedUpdate()
    {
        if (waveIsDone) waveSpawner();
    }

    private void waveSpawner()
    {
        Debug.Log("Wave " + waveCount + " has started!");
        waveIsDone = false;
        if (waveCount % 5 != 0)
        {
            StartCoroutine(spawnWave(enemyCount, enemies));
        }
        else
        {
            StartCoroutine(spawnWave(1, bossEnemies));

        }
    }
```

3.2.2.3. Skripta za izbornik unapređenja statistika igrača

Ova skripta služi za otvaranje izbornika za unapređenje statistika igrača (slika 15). Ova skripta se pokreće kada igrač unaprijedi svoju razinu.

Slika 15. Prikaz izbornika unaprijeđena statistike (vlastita izrada)

Ona u „Awake“ metodu postavlja svoju vidljivost na „false“ i inicijalizira statistike igrača i svoje tekstualne objekte.

```
public CharacterStats playerStats;
    public GameObject LevelUpMenu;
    public static bool isLevelUp;

    //UI elements
    public GameObject maxHealthUI;
    public GameObject damageUI;
    public GameObject fireRateUI;
    public GameObject collectionRadiusUI;
    public GameObject projectileSpeedUI;
    public GameObject attackRangeUI;

    //Text element
    private TextMeshProUGUI textMaxHealth;
    private TextMeshProUGUI textDamage;
    private TextMeshProUGUI textFireRate;
```

```
        private TextMeshProUGUI textCollectionRadius;
        private TextMeshProUGUI textProjectileSpeed;
        private TextMeshProUGUI textAttackRange;

        //Button element
        public Button btnMaxHealth;
        public Button btnDamage;
        public Button btnFireRate;
        public Button btnCollectionRadius;
        public Button btnProjectileSpeed;
        public Button btnAttackRange;

        private void Awake()
        {
            LevelUpMenu.SetActive(false);
            playerStats.InitializeStats();
            InitializeTextObjects();
        }
    private void InitializeTextObjects()
        {
            textMaxHealth = maxHealthUI.GetComponent<TextMeshProUGUI>();
            textDamage = damageUI.GetComponent<TextMeshProUGUI>();
            textFireRate = fireRateUI.GetComponent<TextMeshProUGUI>();
            textCollectionRadius                                    =
collectionRadiusUI.GetComponent<TextMeshProUGUI>();
            textProjectileSpeed                                     =
projectileSpeedUI.GetComponent<TextMeshProUGUI>();
            textAttackRange                                         =
attackRangeUI.GetComponent<TextMeshProUGUI>();
        }
```

Metoda „*InitializeLevelUp*“ otvara izbornik za unapređenje statistika i pokreće „*StatChecker*“, te onda izvršava metodu „*SetPlayerStats*“ i „*UpdateButtonInteractivity*“, te zaustavlja vrijeme igre, što učini igru pauziranom. Metoda „*SetPlayerStats*“ postavlja statistike igrača u tekstualne okvire, a „*UpdateButtonInteractivity*“ provjeri da li su statistike igrača došle do granice i postavlja gumbe da budu bez interakcije.

```
    public void InitializeLevelUp()
        {
            isLevelUp = true;
            LevelUpMenu.SetActive(true);
            playerStats.StatChecker();
```

```
            SetPlayerStats();
            UpdateButtonInteractivity();
            Time.timeScale = 0;
        }

        private void UpdateButtonInteractivity()
        {
            if (playerStats.MaxHealth >= 50) btnMaxHealth.interactable =
false;
            if (playerStats.Damage >= 10) btnDamage.interactable = false;
            if (playerStats.FireRate <= 0.2f) btnFireRate.interactable =
false;
            if (playerStats.CollectionRadius >= 5)
btnCollectionRadius.interactable = false;
            if (playerStats.ProjectileSpeed >= 5)
btnProjectileSpeed.interactable = false;
            if (playerStats.AttackRange >= 3) btnAttackRange.interactable
= false;
        }
    public void SetPlayerStats()
        {
            textMaxHealth.text = playerStats.MaxHealth.ToString();
            textDamage.text = playerStats.Damage.ToString();
            textFireRate.text = playerStats.FireRate.ToString();
            textCollectionRadius.text =
playerStats.CollectionRadius.ToString("F2");
            textProjectileSpeed.text =
playerStats.ProjectileSpeed.ToString();
            textAttackRange.text = playerStats.AttackRange.ToString();
        }
```

Ostale metode unutar skripte su povezane sa svojim gumbima, te su relativno slične jer pozivaju istu metodu iz „*CharacterStats*“ i povećavaju svoju statistiku. Jedina različita metoda je „*HealPlayer*“ koja poziva metodu „*Heal*“ iz klase „*HealthController*“ za liječenje igrača.

```
    public void MaxHealthIncrement()
        {
            float healthIncrement = playerStats.MaxHealth / 10;
            string formattedHealthIncrement =
healthIncrement.ToString("F2");
            if(float.TryParse(formattedHealthIncrement, out
healthIncrement))
            {
                playerStats.IncreaseStats(healthIncrement, 0, 0, 0, 0, 0);
```

```
                HealthController playerCurrentHealth = GameObject.FindWithTag("Player").GetComponent<HealthController>();
                if (playerCurrentHealth != null) playerCurrentHealth.currentHealth += healthIncrement;
            }
            else
            {
                playerStats.IncreaseStats(healthIncrement, 0, 0, 0, 0, 0);
                HealthController playerCurrentHealth = GameObject.FindWithTag("Player").GetComponent<HealthController>();
                if (playerCurrentHealth != null) playerCurrentHealth.currentHealth += healthIncrement;
            }

            LevelUpMenu.SetActive(false);
            Time.timeScale = 1;
            isLevelUp = false;
        }
        public void DamageIncrement()
        {
            playerStats.IncreaseStats(0, 0.5f, 0, 0, 0, 0);
            LevelUpMenu.SetActive(false);
            Time.timeScale = 1;
            isLevelUp = false;
        }
        public void FireRateIncrement()
        {
            playerStats.IncreaseStats(0, 0, 0, 0, -0.05f, 0);
            LevelUpMenu.SetActive(false);
            Time.timeScale = 1;
            isLevelUp = false;
        }
        public void CollectionRadiusIncrement()
        {
            playerStats.IncreaseStats(0, 0, 0.1f, 0, 0, 0);
            LevelUpMenu.SetActive(false);
            Time.timeScale = 1;
            isLevelUp = false;
        }
        public void ProjectileSpeedIncrement()
        {
            playerStats.IncreaseStats(0, 0, 0, 0.5f, 0, 0);
```

```
            LevelUpMenu.SetActive(false);
            Time.timeScale = 1;
            isLevelUp = false;
        }
        public void AttackRangeIncrement()
        {
            playerStats.IncreaseStats(0, 0, 0, 0, 0, 0.2f);
            LevelUpMenu.SetActive(false);
            Time.timeScale = 1;
            isLevelUp = false;
        }

        public void HealPlayer()
        {
            HealthController playerHeal = GameObject.FindWithTag("Player").GetComponent<HealthController>();
            if (playerHeal != null) playerHeal.Heal(0.3f);
            LevelUpMenu.SetActive(false);
            Time.timeScale = 1;
            isLevelUp = false;
        }
```

3.2.2.4. Skripta za izbornik pauzirane igre

Ova skripta služi za otvaranje izbornika za pauziranu igru(slika 15). Ova skripta se pokreće kada igrač pritisne tipku „ESC“. Gumb „*Main menu*“ vraća na glavni izbornik igre, a gumb „*Resume game*“ vraća igrača u igru.

Slika 16. Prikaz izbornika pauzirane igre (vlastita izrada)

Skripta unutar „*Awake*“ metode se postavlja svoju vidljivost na „*false*“ i pokreće „InitializeTextObjects“ metodu unutar koje se inicijaliziraju tekstualni okviri.

```
public CharacterStats playerStats;
    public GameObject PauseMenu;
    public bool isPaused;

    //UI elements
    public GameObject maxHealthUI;
    public GameObject damageUI;
    public GameObject fireRateUI;
    public GameObject collectionRadiusUI;
    public GameObject projectileSpeedUI;
    public GameObject attackRangeUI;

    //Text element
    private TextMeshProUGUI textMaxHealth;
    private TextMeshProUGUI textDamage;
    private TextMeshProUGUI textFireRate;
    private TextMeshProUGUI textCollectionRadius;
    private TextMeshProUGUI textProjectileSpeed;
    private TextMeshProUGUI textAttackRange;

    private void Awake()
    {
        PauseMenu.SetActive(false);
        InitializeTextObjects();
    }
private void InitializeTextObjects()
    {
        textMaxHealth = maxHealthUI.GetComponent<TextMeshProUGUI>();
        textDamage = damageUI.GetComponent<TextMeshProUGUI>();
        textFireRate = fireRateUI.GetComponent<TextMeshProUGUI>();
        textCollectionRadius                                    =
collectionRadiusUI.GetComponent<TextMeshProUGUI>();
        textProjectileSpeed                                     =
projectileSpeedUI.GetComponent<TextMeshProUGUI>();
        textAttackRange                                         =
attackRangeUI.GetComponent<TextMeshProUGUI>();
    }
```

Ova skripta koristi „*Update*“ metodu za praćenje da li je igrač pritisnuo tipku „ESC“, te ako je tipka pritisnuta, a igra je pauzirana otvara izbornik pauzirane igre, suprotno gasi izbornik

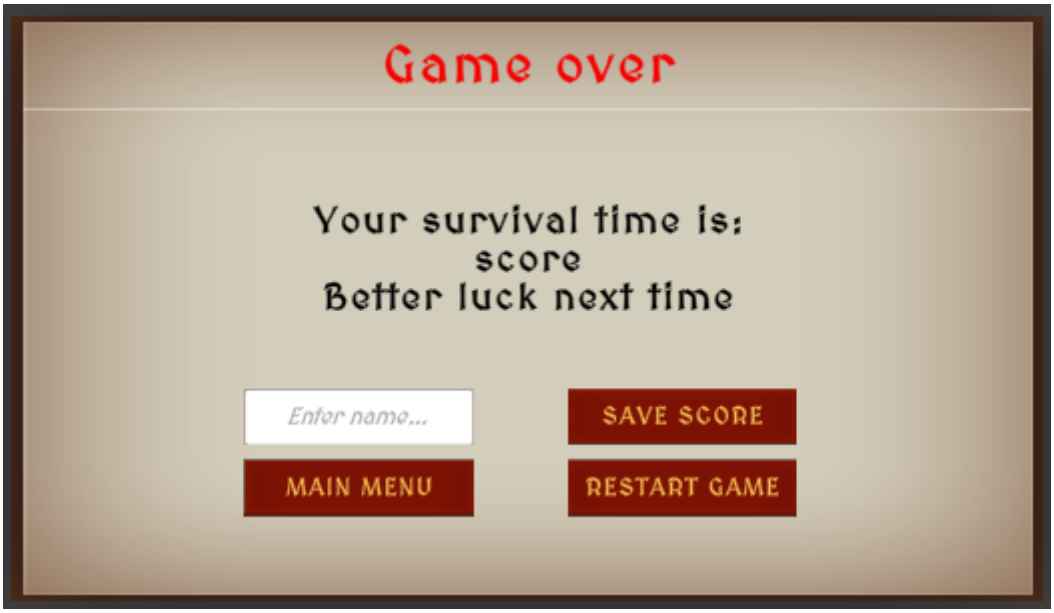
pauzirane igre. Metoda „*PauseGame*“ postavlja vidljivost izbornika na „true“, pauzira vrijeme igre i postavlja statistike igrača sa metodom „*SetPlayerStats*“ u tekstualne okvire. Metoda „*ResumeGame*“ gasi izbornik i ponovno pali vrijeme igre. Metoda „*OpenMainMenu*“ otvara glavni izbornik igre.

```
    private void Update()
        {
            if
(Input.GetKeyDown(KeyCode.Escape)&&LevelUpMenuController.isLevelUp==false){
                if (isPaused)
                {
                    ResumeGame();
                }
                else
                {
                    PauseGame();
                }
            }
        }
    public void SetPlayerStats()
        {
            textMaxHealth.text = playerStats.MaxHealth.ToString();
            textDamage.text = playerStats.Damage.ToString();
            textFireRate.text = playerStats.FireRate.ToString();
            textCollectionRadius.text                                    =
playerStats.CollectionRadius.ToString();
            textProjectileSpeed.text                                     =
playerStats.ProjectileSpeed.ToString();
            textAttackRange.text = playerStats.AttackRange.ToString();
        }
        private void PauseGame()
        {
            SetPlayerStats();
            PauseMenu.SetActive(true);
            Time.timeScale = 0;
            isPaused = true;
        }

        public void ResumeGame()
        {
            PauseMenu.SetActive(false);
            Time.timeScale = 1;
```

```
            isPaused = false;
        }
        public void OpenMainMenu()
        {
            Time.timeScale = 1;
SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex - 1);
        }
```

3.2.2.5. Skripta za izbornik gubitka igre

Ova skripta služi za otvaranje izbornika gubitka igre(slika 16). Ova skripta se pokreće kada igrač pogine u igri. Igrač može napisati željeno ime u polje za unos, te kliknuti na gumb „*Save score*“ koji sprema trenutno postignuće igrača. Gumb „*Main menu*“ vraća igrača na glavni izbornik igre sa metodom „*OpenMainMenu*“ iz skripte za izbornik pauzirane igre, a gumb „*Restart game*“ ponovno pokreće nivo igre.



Slika 17. Prikaz izbornika gubitka igre (vlastita izrada)

Ova skripta u svojoj „Awake“ metodi postavlja svoju vidljivost na „*false*“. Metoda „*GameOver*“ pauzira zvuk igre, postavlja vidljivost na „*true*“, te dohvaća mjerač vremena i ispisuje ga u tekstualni okvir i pauzira vrijeme igre. Metoda „*RestartGame*“ ponovno pali vrijeme igre i ponovno učitava nivo igre. Metoda „*SaveScore*“ sprema trenutno postignuće igre, tako što poziva metodu „*AddScoreboardEntry*“ iz „ScoreboardTable“ klase.

```
    [Header("Components")]
    public GameObject GameOverMenu;
    public TextMeshProUGUI showcaseSore;
    public TMP_InputField inputField;
    public AudioSource gameAudio;
    public static bool isOver;

    private void Awake()
    {
        isOver = false;
        GameOverMenu.SetActive(false);
    }

    public void GameOver()
    {
        gameAudio.Pause();
        ScoreTimer scoreTimer = GameObject.Find("GameManager").GetComponent<ScoreTimer>();
        if (scoreTimer != null)
        {
            isOver = true;
            GameOverMenu.SetActive(true);
            Debug.Log("Timer: "+scoreTimer.ShowScore());
            showcaseSore.text = scoreTimer.ShowScore();
            Time.timeScale = 0;
        }

    }

    public void RestartGame()
    {
        isOver = false;
        Time.timeScale = 1;
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }

    public void SaveScore()
    {
        ScoreboardTable scoreboardTable = new ScoreboardTable();
        ScoreTimer scoreTimer = GameObject.Find("GameManager").GetComponent<ScoreTimer>();
```

```
            float timer;
            if(float.TryParse(scoreTimer.ShowScore(), out timer))
            {
                scoreboardTable.AddScoreboardEntry(timer,
inputField.text);
            }
        }
```

3.2.2.6. Skripta za prikaz vremena i trake iskustva

Skripta za prikaz vremena služi za pokretanje mjerača vremena i prikazuje ga na ekran u gornjem lijevom kutu, dok skripta trake iskustva prikazuje trenutnu količinu iskustva koju igrač ima, te koja je igračeva trenutna razina i koliko iskustvenih jedinica treba skupiti do sljedeće razine (slika 18).

Slika 18. Prikaz mjerača vremena i trake iskustva (vlastita izrada)

Skripta mjerača vremena u metodi „*Start*“ postavlja vrijeme na 0, u metodi „*Update*“ provjerava da li je igra završila i ako nije mjerač vremena normalno mjeri vrijeme, inače zaustavi mjerenje vremena. Metoda „*ShowScore*“ vraća trenutno vrijeme do kojeg je mjerač vremena došao.

```
    [Header("Component")]
        public TextMeshProUGUI timerText;

        private float currentTime;
        void Start()
        {
            currentTime = 0;
        }
```

```
    void Update()
    {
        if (!GameOverMenuController.isOver)
        {
            currentTime += Time.deltaTime;
            timerText.text = currentTime.ToString("F2");
        }
    }
    public string ShowScore()
    {
        return currentTime.ToString("F2");
    }
```

Skripta trake iskustva ima samo jednu metodu koja prihvaća 3 parametra, prvi parametar predstavlja trenutno iskustvo igrača, drugi predstavlja potreban broj iskustva za sljedeću razinu i treći predstavlja trenutnu razinu igrača.

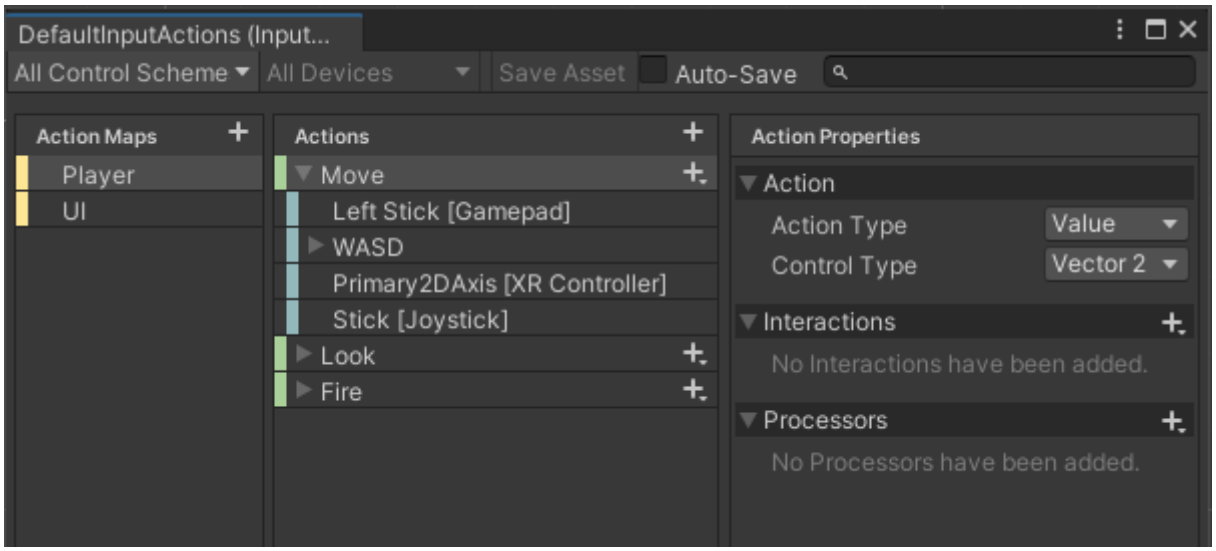
```
[SerializeField]
    private Slider experienceBar;

    [SerializeField]
    private TextMeshProUGUI textLevel;

    public void UpdateExperienceBar(float currentExp, float maxExp, int
level)
    {
        experienceBar.value = currentExp / maxExp;
        int currentLevel = level + 1;
        textLevel.text = "Level " + currentLevel + ": " + currentExp +
" / " + maxExp;
    }
```

3.3. Igrač

Igrač (engl. „*Player*“) je objekt kojim korisnik može upravljat. Osoba koja upravlja igračem ima samo mogućnosti kretanja, pucanje se izvršava automatski prema najbližem objektu neprijatelja. Objekt igrača sadrži 4 objekta djeteta. Prvi objekt služi kao točka iz koje se instanciraju projektili koje ispucava igrač. Drugi objekt služi za prikazivanje slike igrača i njegovih animacija. Treći objekt služi kao platno za prikazivanje zdravstvene trake i četvrti objekt služi kao platno za prikazivanje trake iskustva (traka iskustva je prikazana na slici 18).

3.3.1. Skripta za upravljanje igrača

Unity ima ugrađeni paket („*Input Actions*“) koji služi za unaprijed postavljanje akcija tipki unutar video igre. Unutar tog paketa dođe se može izabrati „*DefaultInputActions*“ resurs koji dolazi sa nekoliko predefiniranih akcija (slika 19). Unutar ovog projekta su samo važne „*move*“ akcije, te metoda „*OnMove*“ koja prima jedan parametar koji predstavlja smjer kretanja igrača.



Slika 19. Prikaz predefiniranih akcija unutar paketa „Input Actions" (vlastita izrada)

Unutar metode „*Start*“ dohvaćaju se sve potrebne komponente za kretanje igrača, to su „*RigidBody2D*“ komponentu, animator komponentu i „*SpriteRenderer*“ komponentu.

```
public CharacterStats characterStats;

    public ContactFilter2D movementFilter;

    Vector2 movementInput;

    SpriteRenderer spriteRenderer;
```

```
        Rigidbody2D rb;
        Animator animator;
        List<RaycastHit2D> castCollisions = new List<RaycastHit2D> ();

        void Start()
        {
            rb = GetComponent<Rigidbody2D>();
            animator = GetComponentInChildren<Animator> ();
            spriteRenderer = GetComponentInChildren<SpriteRenderer> ();
        }
```

S pomoću metode „*TryMove*“ se pomiče objekt igrača. Unutar metode prvo provjerava da li se igrač kreće, te ako se ne kreće izlazimo iz metode. Nakon toga pozivamo metodu od „*RigidBody2D*“ komponente „*Cast*“ koja prima 4 parametra, prvi predstavlja putanju u kojoj se igrač kreće, druga predstavlja postavke ugrađenog filtera u Unity-u (njega se može uređivati unutar Unity grafičkog sučelja), treći predstavlja listu objekata koji se otkriju na putanji sa „*RaycastHit2D*“ i četvrti predstavlja daljinu na kojoj se traže sudarni objekti. Ta funkcija vraća broj objekata koji su se našli na putanji objekta igrača. Nakon vraćenog rezultata „*Cast*“ metode provjerava se da li ima objekta na putanji, ako nema onda pomoću metode „*MovePosition*“ se pomiče objekt igrača u željenom smjeru, a ako ima objekata na putanji igrač stoji na mjestu.

```
    private bool TryMove(Vector2 direction)
    {
          if(direction == Vector2.zero) return false;
            int count = rb.Cast(
                    direction,
                    movementFilter,
                    castCollisions,
                    characterStats.MovementSpeed * Time.fixedDeltaTime + characterStats.CollisionOffset);
            if (count == 0)
            {
                rb.MovePosition(rb.position + direction * characterStats.MovementSpeed * Time.fixedDeltaTime);
                return true;
            } else
            {
                return false;
```

```
        }
```

Unutar „*FixedUpdate*“ metode prvo se provjerava da li se igrač kreće, ako se kreće provjeravamo u kojem smjeru se igrač kreće i postavljamo varijablu animatora na „*true*“, a ako se igrač ne kreće postavljamo varijablu animatora na „*false*“. Nakon prolaska uvjeta kretanja se izvršava još jedna provjera koja gleda da li se igrač kreće u lijevo ili desno, te prema dobivenoj informaciji okreće sliku igrača u lijevo ili desno.

```
private void FixedUpdate(){
        if(movementInput != Vector2.zero){
            bool success = TryMove(movementInput);

            if (!success ) {
                success = TryMove(new Vector2(movementInput.x,0));
            }

            if (!success )
            {
                success = TryMove(new Vector2(0, movementInput.y));
            }
            animator.SetBool("isMoving", success);
        }
        else
        {
            animator.SetBool("isMoving", false);

        }

        if (movementInput.x < 0)
        {
            spriteRenderer.flipX = true;
        }else if (movementInput.x > 0)
        {
            spriteRenderer.flipX = false;

        }
    }
void OnMove(InputValue movementValue){
        movementInput = movementValue.Get<Vector2>();
    }
```

3.3.2. Skripta za upravljanje zdravljem i zdravstvena traka

Ova skripta i objekt zdravstvene trake se odnosi na objekte igrača i neprijatelja. Skripta i objekt su napravljeni sa namjerom da se mogu koristiti na svim potrebnim objektima bez potrebe za mijenjanjem koda.

3.3.2.1. Skripta za upravljanje zdravljem

Ova skripta upravlja zdravljem objekta. Postavlja trenutno količinu zdravlja objekta, te upravlja primanjem, liječenjem i uništavanjem tog objekta. Unutar metode „Awake“ skripta dohvaća zdravstvenu traku objekta i postavlja trenutno zdravlje da bude jednako maksimalnom zdravlju.

```
public CharacterStats characterStats;

    FloatingHealthBar healthBar;

    public float currentHealth;

    bool damageCooldown = false;

    private void Awake()
    {
        healthBar = GetComponentInChildren<FloatingHealthBar>();
        currentHealth = characterStats.MaxHealth;
    }
```

Metoda „*Heal*“ služi za liječenje objekta i ona prima jedan parametar koji predstavlja količinu liječenja zdravlja objekta. Metoda primljeni parametar množi sa maksimalnim zdravljem objekta i dodaje ga na trenutno zdravlje, ako trenutno zdravlje pređe preko maksimalnog ono se postavi na količinu maksimalnog zdravlja. Na kraju poziva „*UpdateHealthBar*“ metodu koja upravlja prikazom trenutnog zdravlja objekta.

```
public void Heal(float amount)
    {
        float healAmount = characterStats.MaxHealth * amount;
        currentHealth += healAmount;
        if (currentHealth >= characterStats.MaxHealth)
        {
            currentHealth = characterStats.MaxHealth;
        }
        healthBar.UpdateHealthBar(currentHealth,
characterStats.MaxHealth);
```

```
    }
```

Metoda „*TakeDamage*“ upravlja primanjem štete objekata. Ona prima jedan parametar koji predstavlja količinu štete koju će objekt primiti. Unutar metode se prvo provjerava da li je aktivan razmak između štete, ako nije metoda se nastavlja izvršavati, a ako je se izlazi iz metode. Nakon toga se nanosi šteta objektu, te ako je zdravlje objekta ispod 0 se poziva metoda „*Die*“. Ako objekt još ima zdravlja se ažurira zdravstvena traka sa metodom „*UpdateHealthBar*“, te se poziva koordinirana metoda „*DelayDamage*“ koja čeka da prođe vrijeme razmaka između štete.

```
public void TakeDamage(float amount)
    {
        if (damageCooldown) return;

        damageCooldown = true;
        currentHealth -= amount;

        if(currentHealth <= 0)
        {
            currentHealth = 0;
            Die(gameObject);
        }
        healthBar.UpdateHealthBar(currentHealth,
characterStats.MaxHealth);
        StartCoroutine(DelayDamage());
    }
private IEnumerator DelayDamage()
    {
        yield                    return                    new
WaitForSeconds(characterStats.InvincibilityFrames);
        damageCooldown = false;
    }
```

Metoda „*Die*“ upravlja uništavanjem objekata. Unutar ove metode se prvo provjerava da li je objekt sa etiketom (engl. „*Tag*“) „*Enemy*“, te ako je instancira objekt spremljen u klasi „LootBag“ (Funkcionalnosti ove klase su objašnjene u poglavlju Neprijatelji) i uništava trenutni objekt. Ako etiketa nije „*Enemy*“ se postavlja mogućnost napretka razine na „*false*“ i dohvaća se „*GameManger*“ objekt. Uz pomoć „*GameManger*“ objekta se poziva izbornik gubitka igre, te se uništava objekt igrača.

```
private void Die(GameObject gameObject)
    {
        if (gameObject.CompareTag("Enemy")){
```

```
gameObject.GetComponent<LootBag>().InstantiateLoot(transform.position);
                Destroy(gameObject);
            }
            else
            {
                LevelUpMenuController.isLevelUp = false;
                GameObject worldObjects = GameObject.Find("GameManager");
                if (worldObjects != null)
                {
                    GameOverMenuController gameOverMenuController =
worldObjects.GetComponent<GameOverMenuController>();
                    Destroy(gameObject);
                    gameOverMenuController.GameOver();
                }
                else
                {
                    Debug.Log("Couldn't find GameManager");
                    Destroy(gameObject);
                }
            }
        }
```

3.3.2.2. Zdravstvena traka

Zdravstvena traka je objekt povezan na sve objekte koji koriste skriptu za upravljanjem zdravlja (slika 18 iznad igrača). Zdravstvena traka se sastoji od objekta klizača (engl. „Slider“) koji unutar sebe sadrži skriptu „*FloatingHealthBar*“ koja služi za ažuriranje zdravstvene trake prema trenutnom zdravlju objekta. „*FloatingHealthBar*“ skripta sadrži jednu metodu „*UpdateHealthBar*“ koja služi za prikaz trenutne količine zdravlja.

```
    [SerializeField]
        private Slider healthBar;

        public void UpdateHealthBar(float currentHealth, float maxHealth)
        {
            healthBar.value = currentHealth / maxHealth;
        }
```

3.3.3. Skripta za upravljanje napredovanja igrača

Ova skripta upravlja funkcionalnostima napredovanja igrača. Ona u svojoj „Start“ metodi inicijalizira izbornik za napredovanje i traku iskustva, te u svom konstruktoru postavlja razinu i jedinice iskustva na 0, a potrebne jedinice iskustva za napredovanje razine na 2.

```
private int level;
    private float experience;
    private float experienceToNextLevel;
    private FloatingExperienceBar experienceBar;

    public LevelUpMenuController levelUpMenuController;

    private void Start()
    {
        GameObject worldObjects = GameObject.Find("GameManager");

        if (worldObjects != null)
        {
            levelUpMenuController = worldObjects.GetComponent<LevelUpMenuController>();
            experienceBar = GameObject.FindWithTag("Player").GetComponentInChildren<FloatingExperienceBar>();
            experienceBar.UpdateExperienceBar(experience, experienceToNextLevel, level);

        }
        else
        {
            Debug.LogError("Could not find 'GameManager' in the scene.");
        }
    }
    public LevelController()
    {
        level = 0;
        experience = 0;
        experienceToNextLevel = 2;
    }
```

Metoda „*AddExperience*“ prima jedan parametar koji predstavlja količinu jedinica iskustva, te ih dodaje na trenutne jedinice iskustva igrača. Nakon što promijeni količinu iskustva

provjerava da li ona prelazi potrebnu količinu za napredovanje razine, te ako je poziva metodu „*LevelUp*“. Na kraju se ažurira traka iskustva sa metodom „*UpdateExperienceBar*“. Metoda „*LevelUp*“ otvara izbornik za napredovanje, te povećava razinu igrača za 1, resetira jedinice iskustva i povećava potrebne jedinice iskustva za napredovanje razine za 2.

```
    public void AddExperience(float amount)
        {
            experience += amount;
            if(experience >= experienceToNextLevel)
            {
                LevelUp();
            }
            experienceBar.UpdateExperienceBar(experience,
experienceToNextLevel, level);
        }

        private void LevelUp()
        {
            levelUpMenuController.InitializeLevelUp();
            level++;
            experience -= experienceToNextLevel;
            if (experienceToNextLevel < 25)
            {
                experienceToNextLevel += 2;
            }
            else if (experienceToNextLevel > 25)
            {
                experienceToNextLevel = 25;
            }
        }
```

Metoda „*BossKillLevelUp*“ je slična metodi „*LevelUp*“, jedina razlika je što ona samo povećava razinu igrača bez utjecaja na jedinice iskustva. Ova metoda je povezana samo sa šefovima neprijateljima.

```
    public void BossKillLevelUp()
        {
            levelUpMenuController.InitializeLevelUp();
            level++;
            if (experienceToNextLevel < 25)
            {
                experienceToNextLevel += 2;
```

```
            } else if(experienceToNextLevel > 25)
            {
                experienceToNextLevel = 25;
            }
        }
```

3.3.4.Mehanike borbe igrača

Osoba koja upravlja objektom igrača nema direktni utjecaj na pucanje, jer se pucanje izvršava automatski prema najbližem objektu neprijatelja. Jedini utjecaj igrača na borbu je njegovo pozicioniranje naspram neprijatelja i povećavanje statistike.

3.3.4.1. Skripta za pucanje

Skripta za pucanje je pridružena objektu „*ShootingPoint*“ koji je dijete objekta igrača. Skripta se sastoji od metoda „*ShootProjectile*“ i „*FixedUpdate*“. Metoda „*ShootProjectile*“ prima jedan argument koji predstavlja poziciju neprijatelja. Taj parametar koristi kako bi se objekt projektila mogao rotirati u smjeru neprijatelja, te se onda instancira objekt projektila i postavlja mu se brzina koja se uzima iz „CharacterStats“ povezanog sa skriptom i smjer kretanja.

```
    private void ShootProjectile(Transform enemyPosition)
        {
            Vector3 norTar = (enemyPosition.position - transform.position).normalized;
            float angle = Mathf.Atan2(norTar.y, norTar.x) * Mathf.Rad2Deg;
            Quaternion rotation = new Quaternion();
            rotation.eulerAngles = new Vector3(0, 0, angle - 90);
            transform.rotation = rotation;

            GameObject projectal = Instantiate(characterStats.projectile, transform.position, transform.rotation);
            Rigidbody2D rb = projectal.GetComponent<Rigidbody2D>();

            rb.velocity = characterStats.ProjectileSpeed * transform.up;
        }
```

Metoda „*FixedUpdate*“ koristi metodu „*OverlapCircleAll*“ koja pozicije svih objekata sa komponentama sudara stavlja u listu. Ona prima 3 parametra, prvi predstavlja centar krug (u ovom slučaju poziciju igrača), drugi predstavlja radijus u kojem traži objekte i treća služi kao maska sloja unutar kojeg djeluje (u slučaju igrača djeluje na sloj „*Enemy*“). Nakon što dohvatimo objekte sortiramo ih sa prilagođenim načinom sortiranjem. Taj način sortiranja je napisan u klasi koja nasljeđuje „*IComparer*“, te uspoređuje pozicije objekata kako bi pronašao

najbližu. Nakon sortiranja „*FixedUpdate*“ metoda provjerava kada je zadnji put ispalila projektil, te ako je prošlo dovoljno vremena ispaljuje novi projektil sa metodom „*ShootProjectile*“.

Metoda za sortiranje:

```
public class DistanceComparer : IComparer
{
    private Transform compareTransform;
    public DistanceComparer(Transform compTransform)
    {
        compareTransform = compTransform;
    }
    public int Compare(object x, object y)
    {
        Collider2D xCollider = x as Collider2D;
        Collider2D yCollider = y as Collider2D;

        Vector2 offset = xCollider.transform.position - compareTransform.position;
        float xDistance = offset.sqrMagnitude;

        offset = yCollider.transform.position - compareTransform.position;
        float yDistance = offset.sqrMagnitude;

        return xDistance.CompareTo(yDistance);
    }
}
```

Metoda „*FixedUpdate*“:

```
public CharacterStats characterStats;
    [SerializeField]
    private LayerMask layerMask;

    private float lastprojectileTime;

    private void FixedUpdate()
    {
        Collider2D[] colliderArray = Physics2D.OverlapCircleAll(transform.position, characterStats.AttackRange,layerMask);
        Array.Sort(colliderArray,new DistanceComparer(transform));
```

```
            foreach (Collider2D collider in colliderArray)
            {
                float timeSinceLastProjectile = Time.time -
lastprojectileTime;
                if (timeSinceLastProjectile >= characterStats.FireRate)
                {
                    ShootProjectile(collider.transform);
                    lastprojectileTime = Time.time;
                }
            }
        }
```

3.3.4.2. Skripta za projektile

Komponente sudara u objektu projektila se postavljene kao okidač, jer projektili ne trebaju imati kolizije već trebaju samo ispuniti događaj kada priđu određenom objektu. Projektil unutar „*Awake*“ metode inicijalizira kameru, te u „*FixedUpdate*“ poziva metodu „*DestroyWhenOffScreen*“ koja uništava objekt projektila kada ode sa ekrana.

```
    public CharacterStats characterStats;

        private new Camera camera;

        private void Awake()
        {
            camera = Camera.main;
        }

        private void FixedUpdate()
        {
            DestroyWhenOffScreen();
        }
    private void DestroyWhenOffScreen()
        {
            Vector2 screenPosition =
camera.WorldToScreenPoint(transform.position);

            if(screenPosition.x < 0 ||
                screenPosition.x > camera.pixelWidth ||
                screenPosition.y < 0 ||
                screenPosition.y > camera.pixelHeight)
            {
                Destroy(gameObject);
```

```
            }
        }
```

Pomoću „*OnTriggerEnter2D*“ metode se provjerava da li je objekt projektila došao u kontakt s drugim objektom. Ako je projektil došao u kontakt sa drugim objektom koji sadrži etiketu „*EnemyBody*“ uzima njegov upravljač zdravljem i nanosi mu štetu, te onda projektil uništi samog sebe.

```
    private void OnTriggerEnter2D(Collider2D collision)
        {
            if(collision.gameObject.CompareTag("EnemyBody"))
            {
                GameObject                  enemyMainBody                  =
collision.transform.parent.gameObject;
                var                   healthController                   =
enemyMainBody.GetComponent<HealthController>();

                healthController.TakeDamage(characterStats.Damage);
                Destroy(gameObject);
            }
        }
```

Slična skripta se koristi i kod neprijatelja, jedina razlika se nalazi unutar metode „*OnTriggerEnter2D*“ koja gleda etiketu „*Player*“ umjesto etikete „*EnemyBody*“.

```
    private void OnTriggerEnter2D(Collider2D collision)
        {
            if (collision.gameObject.CompareTag("Player"))
            {
                var                   healthController                   =
collision.GetComponent<HealthController>();

                healthController.TakeDamage(characterStats.Damage);
                Destroy(gameObject);
            }
        }
```

3.3.5. Skripta za radijus skupljanja predmeta

Ova skripta upravlja skupljanjem predmeta koje ispuštaju neprijatelji pri umiranju. Skripta unutar metode „*FixedUpdate*“ koristi metodu „*OverlapCircleAll*“ koja traži objekte sa etiketom „*CollectableItem*“. Kada dođe u kontakt sa objektom izvršava metodu „*ApplyEffectToPlayer*“ i uništava taj objekt.

```
    private void FixedUpdate()
        {
            Collider2D[] colliderArray = Physics2D.OverlapCircleAll(transform.position, characterStats.CollectionRadius);
            foreach (Collider2D collider in colliderArray)
            {
                if (collider.CompareTag("CollectableItem"))
                {
                    CollectableController collectableItem = collider.GetComponent<CollectableController>();

collectableItem.collectableItem.ApplyEffectToPlayer();
                    Destroy(collider.gameObject);
                    break;
                }
            }
        }
```

3.4. Neprijatelj

Neprijatelji su objekti kojima je cilj uništiti igrača. U igri postoje 4 vrste običnih neprijatelja, od kojih jedan koristi napade na daljinu i 2 vrste šefa neprijatelja. Šef neprijatelji su veće verzije normalnih neprijatelja i sa većim statistikama. Svaki neprijatelj baca predmet pri smrti, ti predmeti mogu biti jedna od 2 vrste jedinica iskustva (jedna dodaje samo 1 jedinicu iskustva, a druga dodaje 3 jedinice iskustva), te jednog predmeta za liječenje igrača. Šefovi neprijatelji mogu baciti samo jednu vrstu predmeta koja daje jednu više razinu igraču.

3.4.1. Skripta za kretanje neprijatelja

Skripta za kretanje neprijatelja je slična kao i kod igrača, samo što ona implementira kretanje bez mogućnosti upravljanja od strane korisnika. Ta skripta u svojoj „Awake“ metodi inicijalizira poziciju igrača i komponentu „SpriteRenderer“.

```
public CharacterStats characterStats;
    public ContactFilter2D movementFilter;
    public Vector2 DirectionToPlayer { get; private set; }



    private SpriteRenderer spriteRenderer;
    private Rigidbody2D rb;
    private Transform playerPosition;

    List<RaycastHit2D> castCollisions = new List<RaycastHit2D>();
    private void Awake()
    {
        if(GameObject.FindGameObjectWithTag("Player")     !=     null)
playerPosition = GameObject.FindGameObjectWithTag("Player").transform;
        rb = GetComponent<Rigidbody2D>();
        spriteRenderer = GetComponentInChildren<SpriteRenderer>();
    }
```

Metoda „*MoveToPlayer*“ je ista kao i metoda „*TryMove*“ kod igrača, te metoda „*RotateTowardsTarget*“ rotira sliku neprijatelja u smjeru kretanja. Metoda „FixedUpdate“ osigurava da se neprijatelji kreću prema igraču.

```
private void FixedUpdate()
    {
        if(playerPosition != null)
        {
```

```
                Vector2 enemyToPlayerVector = playerPosition.position -
transform.position;
                DirectionToPlayer = enemyToPlayerVector.normalized;

                RotateTowardsTarget(DirectionToPlayer);
                MoveToPlayer(DirectionToPlayer);
            }
        }

        private void RotateTowardsTarget(Vector2 playerPosition)
        {
            if (playerPosition.x < 0)
            {
                spriteRenderer.flipX = true;
            }
            else if (playerPosition.x > 0)
            {
                spriteRenderer.flipX = false;

            }
        }

        private bool MoveToPlayer(Vector2 direction)
        {
            if (direction == Vector2.zero) return false;
            int count = rb.Cast(
                direction,
                movementFilter,
                castCollisions,
                characterStats.MovementSpeed * Time.fixedDeltaTime +
characterStats.CollisionOffset
                );
            if (count == 0)
            {
                rb.MovePosition(rb.position + direction *
characterStats.MovementSpeed * Time.fixedDeltaTime);
                return true;
            }
            else
            {
                return false;
```

```
        }
    }
```

3.4.2. Mehanike borbe neprijatelja

Mehanike borbe neprijatelja se svode na hvatanje igrača i sudaranja s njim kako bi mu nanijeli štetu. Neprijatelj koji koriste i napade na daljinu također koriste istu logiku borbe kao i ostali neprijatelji, ali oni u svom arsenalu imaju i napade iz daljine te tako predstavljaju konstantnu prijetnju dok su na ekranu. Ti neprijatelji koriste istu skriptu za pucanje kao i igrač, jedina razlika je što ovaj neprijatelj ima postavljenu masku sloja na sloj „*Player*“. Skripta koja služi za udarce na blizinu koristi „*OnTriggerStay2D*“ metodu koja gleda da li je neprijatelj došao u kontakt sa igračem, te onda nanosi štetu igraču. Razlika između „*OnTriggerStay2D*“ metode i „*OnTriggerEnter2D*“ metode je što se „*OnTriggerEnter2D*“ izvrši samo jednom kada objekti dođu u kontakt, dok se metoda „*OnTriggerStay2D*“ izvršava kod dokle god su objekti u kontaktu. „*OnTriggerStay2D*“ metoda u kombinaciji sa „*DelayDamage*“ metodom iz „*HealthController*“ komponente omogućuju povremenu štetu pri kontaktu.

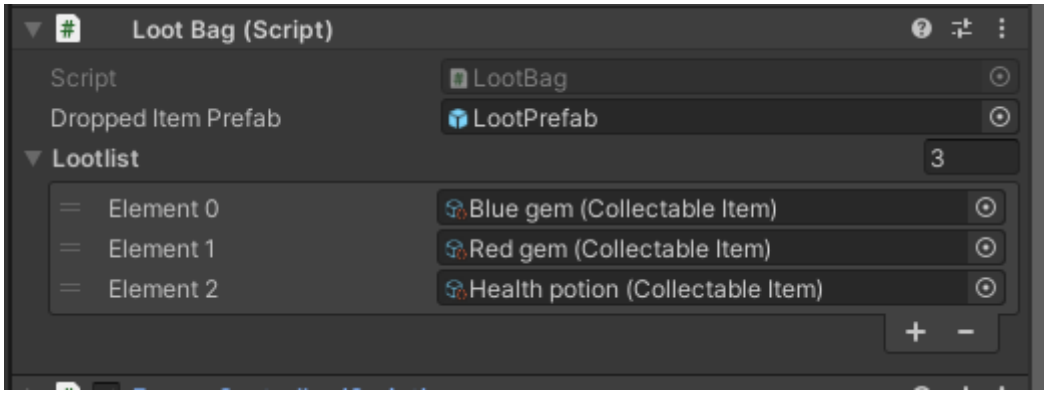
```
    public CharacterStats characterStats;

        private void OnTriggerStay2D(Collider2D collision)
        {
            if (collision.gameObject.CompareTag("Player"))
            {
                var                    healthController                    =
collision.gameObject.GetComponent<HealthController>();

                healthController.TakeDamage(characterStats.Damage);

            }

        }
```

3.5. Mehanika ispuštanja predmeta i interakcija sa predmetima

3.5.1. Mehanika ispuštanja predmeta

Ova mehanika je ključna koncept ove video igre, jer su jedinice iskustva predmeti koji se trebaju pokupiti nakon što se ubije neprijatelja. Omogućavanje ispuštanja predmeta postižemo sa skriptom „*LootBag*“. Kako bi postavio koje predmete želim da neprijatelj ispusti

moram preko grafičkog sučelja povući i ispustiti „*CollectablItem*“ (to je „*ScriptableObject*“ vezan uz predmete) u „*LootBag*“ komponentu (slika 20).



Slika 20. Prikaz komponente "*LootBag*" (vlastita izrada)

Komponenta „LootBag“ se sastoji od metode „*GetDroppedItem*“ i „*InstantiateLoot*“. Metoda „*GetDroppedItem*“ prvo generira jedan nasumičan broj, te onda prolazi kroz listu predmeta i provjerava da li su šanse tog predmeta veće od nasumičnog broja, ako jesu dodaju se u listu mogućih predmeta za ispuštanje pri uništenju objekta neprijatelja. Nakon što dohvati sve moguće predmete, ona ponovno generira nasumičan broj i vraća jedan predmet koji ima isti indeks kao taj broj. Unutar metode „*InstantiateLoot*“ prvo se iskoristi metoda „*GetDroppedItem*“ kako bi se dobio jedan predmet. Nakon što se dobije predmet se instancira jedan prazan objekt koji unutar „SpriteRenderer“ komponente poprimi sliku predmeta i unutar komponente „*CollectableController*“ spremi statistike tog predmeta. Komponenta „*CollectableController*“ samo sadrži jednu varijablu tipa „*CollectableItem*“.

Komponenta „*CollectableController*“:

```
public class CollectableController : MonoBehaviour
{
    public CollectableItem collectableItem;
}
```

Metode „*GetDroppedItem*“ i „*InstantiateLoot*“:

```
public GameObject droppedItemPrefab;
    public List<CollectableItem> lootlist = new List<CollectableItem>
();


    CollectableItem GetDroppedItem()
    {
        int randomNumber = Random.Range(1, 101);
```

```
            List<CollectableItem>       possibleItems       =       new
List<CollectableItem>();
            foreach (CollectableItem item in lootlist)
            {
                if (randomNumber <= item.dropChance)
                {
                    possibleItems.Add(item);
                }
            }
            if(possibleItems.Count > 0)
            {
                return
possibleItems[Random.Range(0,possibleItems.Count)];
            }
            Debug.Log("No loot dropped");
            return null;
        }

        public void InstantiateLoot(Vector3 spawnPosition)
        {
            CollectableItem droppedItem = GetDroppedItem();
            if(droppedItem != null)
            {
                GameObject lootGameObject = Instantiate(droppedItemPrefab,
spawnPosition, Quaternion.identity);
                lootGameObject.GetComponent<SpriteRenderer>().sprite    =
droppedItem.lootSprite;

                CollectableController          itemController          =
lootGameObject.AddComponent<CollectableController>();
                itemController.collectableItem = droppedItem;
            }
        }
```

3.5.2.Interakcija sa predmetima

Svaki „*CollectablItem*“ u sebi sadrži još jedan „*ScriptableObject*“ koji sadrži ponašanje tog predmeta. On se zove „*CollectableEffect*“ i može poprimiti jedan od tri efekta, „*Health*“, „*Experience*“ i „*Level*“. Efekt „*Health*“ služi kao naznaka da taj predmet utječe na liječenje igrača, efekt „*Experience*“ služi kao naznaka da utječe na iskustvo igrača, a efekt „*Level*“ služi kao naznaka da utječe na razinu igrača. „*CollectableEffect*“ također sadrži jednu metodu koja efektima definira funkcionalnost.

```
    [CreateAssetMenu(fileName = "New effect", menuName = "Collectable effect")]
    public class CollectableEffect : ScriptableObject
    {
        public enum EffectType
        {
            Health,
            Experience,
            Level
        }
        public EffectType effectType;
        public float effectValue;

        public void ApplyEffect()
        {
            GameObject player = GameObject.FindGameObjectWithTag("Player");

            if (player != null)
            {
                if(effectType == EffectType.Experience)
                {
                    LevelController levelController = player.GetComponent<LevelController>();
                    levelController.AddExperience(effectValue);
                }else if(effectType == EffectType.Health)
                {
                    HealthController healthController = player.GetComponent<HealthController>();
                    healthController.Heal(effectValue);
                }else if (effectType == EffectType.Level)
                {
                    LevelController levelController = player.GetComponent<LevelController>();
                    levelController.BossKillLevelUp();
                }
            }
        }
```

4. Zaključak

Ovaj rad istražuje i demonstrira proces izrade 2D video igre preživljavanja na vrijeme u programskom alatu Unity, koristeći razvojni okvir ovog alata i programski jezik C#. Rad pruža uvid u osnove Unity-ja i njegove mogućnosti, istražuje proces stvaranja video igre sa svim elementima koji su potrebni za izazovan i zabavan igrački doživljaj.

Igra "Gladiator Arena" nudi dinamičnu igrivost, sa scenom glavnog izbornika, jednim nivoom igre i izazovima koji dolaze s njim. Igrač se suočava s različitim vrstama neprijatelja, uključujući i šefove neprijatelja, dok pokušava preživjeti što je duže moguće. Kroz skupljanje jedinica iskustva, igrač ima priliku unaprijediti statistike svog lika, što dodaje dubinu igri i potiče ponovno igranje kako bi se postigao što bolji rezultat.

Jedna od ključnih značajki igre je mjerenje vremena preživljavanja, što dodaje natjecateljski aspekt i potiče igrače na poboljšanje vlastitih rezultata. Također, opcija spremanja postignuća i usporedba s lokalnim rekordima pruža dodatni motivacijski faktor.

Kroz ovaj rad stekao sam dubok uvid u upotrebu „ScriptableObject-a“ za organizaciju skripti i komponenata unutar Unity okoline. Ovaj pristup omogućio mi je strukturiranu organizaciju koda i resursa, te povećao produktivnost daljnje razvoja igre.

Popis literature

[1] Joseph Sibony, „The Best Gaming Engines You Should Consider for 2023“, [Blog post] 2.2.2023. [Na internetu], Dostupno: https://www.incredibuild.com/blog/top-gaming-engines-you-should-consider [Pristupano: 6.9.2023.]

[2] Microsoft „Unity : Developing Your First Game with Unity and C#“, 7.1.2015. [Na internetu], https://learn.microsoft.com/en-us/archive/msdn-magazine/2014/august/unity-developing-your-first-game-with-unity-and-csharp [Pristupano: 6.9.2023.]

[3] Microsoft, „A tour of the C# language“, 4.4.2023. [Na internetu], https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/ [Pristupano: 6.9.2023.]

[4] Unity Technologies, „Grid component reference“, 1.9.2023. [Na internetu], https://docs.unity3d.com/Manual/class-Grid.html [Pristupano: 7.9.2023.]

[5] Unity Technologies, „Introduction to Tilemaps“, 31.1.2023. [Na internetu], https://learn.unity.com/tutorial/introduction-to-tilemaps# [Pristupano: 7.9.2023.]

[6] Unity Technologies, „Introduction to collision“, 1.9.2023. [Na internetu], https://docs.unity3d.com/Manual/CollidersOverview.html [Pristupano: 7.9.2023.]

[7] Unity Technologies, „ScriptableObject“, 1.9.2023. [Na internetu], https://docs.unity3d.com/Manual/class-ScriptableObject.html [Pristupano: 8.9.2023.]

[8] Unity Technologies, „PlayerPrefs“, 1.9.2023. [Na internetu], https://docs.unity3d.com/ScriptReference/PlayerPrefs.html [Pristupano: 8.9.2023.]

[9] Unity Technologies, „Introduction to Sprite Animations“, 10.11.2022. [Na internetu], https://learn.unity.com/tutorial/introduction-to-sprite-animations# [Pristupano: 8.9.2023.]

Popis slika