Izrada akcijske videoigre pucanja i preživljavanja u programskom alatu Unity

Anić, Thomas

Undergraduate thesis / Završni rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:825764

Rights / Prava: Attribution-NonCommercial-NoDerivs 3.0 Unported/Imenovanje-Nekomercijalno-Bez prerada 3.0

Download date / Datum preuzimanja: **2024-11-06**

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

zir.nsk.hr

dabar
DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Thomas Anić

Izrada akcijske videoigre pucanja i preživljavanja u programskom alatu Unity

ZAVRŠNI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Thomas Anić

Matični broj: 0016152917

Studij: Informacijski i poslovni sustavi – Umreženi sustavi i računalne igre

Izrada akcijske videoigre pucanja i preživljavanja u programskom alatu Unity

ZAVRŠNI RAD

Mentor/Mentorica:

Doc. dr. sc. Mladen Konecki

Varaždin, rujan 2024.

Thomas Anić

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

Kreacija i razvoj videoigre unutar programskog alata Unity predstavlja sofisticiran izazov u programskom razvoju tipično šire primjene. Obuhvaća razvijanje metodologije samog igrača i njegovih podsustava, pozadinsko upravljanje scenom i događajima unutar nje, kao i razvijanje same logike iza neprijatelja i umjetne inteligencije korištenom za njihovo upravljanje. Analizom sličnih igara iz srodnih žanrova nalazi se inspiracija kao i teorijsko-metodološki okvir, dok je tehnički aspekt usmjeren na upotrebu ugrađenih komponenata u samom alatu kao i kreacija novih skripti i ručnih sustava prilagođenih specifično za trenutni slučaj korištenja. Razvoj igre također zahtijeva balansiranje izazova za igrača kroz postepeno povećanje težine neprijatelja. Pažljivom primjenom tehnologija poput prilagođenih skripti, ručno izrađene umjetne inteligencije te manipulacijom događaja i animacija značajno utječemo na kvalitetu i izazovnost igre, pružajući igraču uzbudljivo i dinamično iskustvo.

Ključne riječi: videoigra, Unity, sustavi, umjetna inteligencija, skripte, komponente, igrač, neprijatelji

Sadržaj

1. Uvod

Ovaj završni rad bavi se izradom 2D akcijske videoigre pucanja i preživljavanja u programskom alatu Unity. Cilj igre je omogućiti igraču da preživi kroz valove neprijatelja koji postaju sve izazovniji kako igra napreduje. Projekt se temelji na implementaciji sustava oružja, upravljanju resursima poput zdravlja i kondicije, te razvoju mehanika i logike raznih neprijatelja koji se kreću korištenjem umjetne inteligencije. Rad uključuje analizu i razvoj osnovnih elemenata igre, kao što su sustavi borbe, ponašanje neprijatelja i postupno povećavanje težine izazova. Korištenjem ugrađenih Unity alata za manipuliranje događajima i scenom, poput Cinemachine komponente za detaljno upravljanje kamerom, te brojnih prilagođenih skripti u C# jeziku, osigurava se dinamično i izazovno iskustvo igranja. Teorijski pristup radu temelji se na analizi sličnih igara iz žanra preživljavanja i pucanja, dok metodološki dio obuhvaća praktičnu primjenu stečenih znanja iz programiranja, dizajna i testiranja. Glavni zaključci rada pokazuju kako pravilna primjena umjetne inteligencije i balansiranja težine izazova mogu značajno utjecati na kvalitetu korisničkog iskustva, stvarajući igru koja je istovremeno zabavna i izazovna. Rezultat ovog rada je funkcionalna videoigra kao i prikaz faza razvoja od dizajna i planiranja, do implementacije i testiranja, čime se demonstriraju znanja i vještine, uglavnom stečene na raznim kolegijima tokom semestra, a potrebne za razvoj funkcionalne i dinamične videoigre.

2. Metode i tehnike rada

Provođenje istraživačkih aktivnosti većinski je uključivalo analizu i proučavanje videoigara u srodnim žanrovima te proučavanje onih videoigara koji posjeduju slične mehanike. Alati korišteni za takve aktivnosti uključivale su većinski platforme za digitalnu distribuciju videoigara kao štu su Steam i Xbox aplikacija, te Youtube koja je služila kao usluga za pregledavanje stvarnih iskustava videoigara kao i za pregled programskih koncepata.

Programski alat korišten za razvoj videoigre je Unity [1] u kombinaciji sa Microsoft Visual Studio [2] razvojnim okruženjem za programiranje korištenih skripti. Korišteni resursi kao što su teksture i animacije, preuzete su sa Unity trgovine resursa [3] (eng. *Unity Asset Store*).

2.1. Programski alat Unity

Programski alat Unity [1] je jedan od najčešće korištenih alata za razvoj video igara zbog svoje svestranosti i bogatog seta funkcionalnosti koje omogućuju kreiranje igara različitih žanrova i složenosti. Kao sveobuhvatan razvojni alat, Unity pruža kompletan paket za izgradnju igre, uključujući kompleksno korisničko sučelje, alate za modeliranje, fizičke simulacije, skriptiranje i upravljanje resursima. Za potrebe ovog rada, koji se fokusira na 2D igru, programski alat Unity nudi niz specifičnih alata i mogućnosti.

U kontekstu 2D razvoja, Unity omogućuje jednostavno upravljanje animacijama, fizičkim interakcijama i scenama. Alati za animaciju omogućuju stvaranje dinamičnih pokreta i efekata, dok fizičke simulacije pomažu u realističnom prikazu interakcija unutar igre. Skriptiranje u C# je ključno za implementaciju kompleksne logike igre, uključujući upravljanje neprijateljima, oružjem i napredovanjem kroz valove. Unity trgovina resursa (eng. *Unity Asset Store*) i forumi zajednice pružaju pristup raznim gotovim rješenjima i dodatnim resursima koji mogu unaprijediti igru, od grafičkih elemenata do skripti i alata za optimizaciju.

Dodatno, Unity omogućuje kreiranje vizualnih i zvučnih efekata koji doprinose stvaranju uzbudljivih i intenzivnih trenutaka u igri. Sustav čestica koristi se za generiranje efekata poput eksplozija i pucnjeva, sustav sjena za kreiranja efekata osvijetljena ,a zvučni efekti se integriraju pomoću komponente zvučnih izvora, čime se dodatno poboljšava ukupno igračko iskustvo. Modularni pristup razvoju omogućava lako dodavanje novih funkcionalnosti i prilagodbu postojećih elemenata u hijerarhiji, što je posebno korisno za iterativni razvoj igre.

2.2. Programski alat Microsoft Visual Studio

Microsoft Visual Studio [2] je jedan od najvažnijih i najmoćnijih alata za razvoj programskih rješenja, uključujući igre. Kao integrirano razvojno okruženje (eng. *Integraded Development Environment*), Visual Studio pruža sveobuhvatan skup alata za pisanje, uređivanje, otklanjanje pogrešaka i testiranje koda. U sklopu ovog projekta, Visual Studio se koristi za razvoj i optimizaciju skripti u C#, koji su ključni za implementaciju logike igre u programskom alatu Unity [1].

Microsoft Visual Studio nudi napredne značajke poput IntelliSense komponente, koja pomaže u brzom pisanju koda kroz automatske prijedloge i ispravke, što poboljšava produktivnost i smanjuje mogućnost grešaka. Alat za otklanjanje pogrešaka (eng. *debuging)* omogućuje detaljno praćenje i analizu rada koda u stvarnom vremenu, što olakšava pronalaženje i ispravljanje grešaka. Također, Microsoft Visual Studio podržava razne ekstenzije i dodatke koji mogu unaprijediti funkcionalnosti, kao što su alati za rad sa Git sustavom i dodatni alati za analizu performansi.

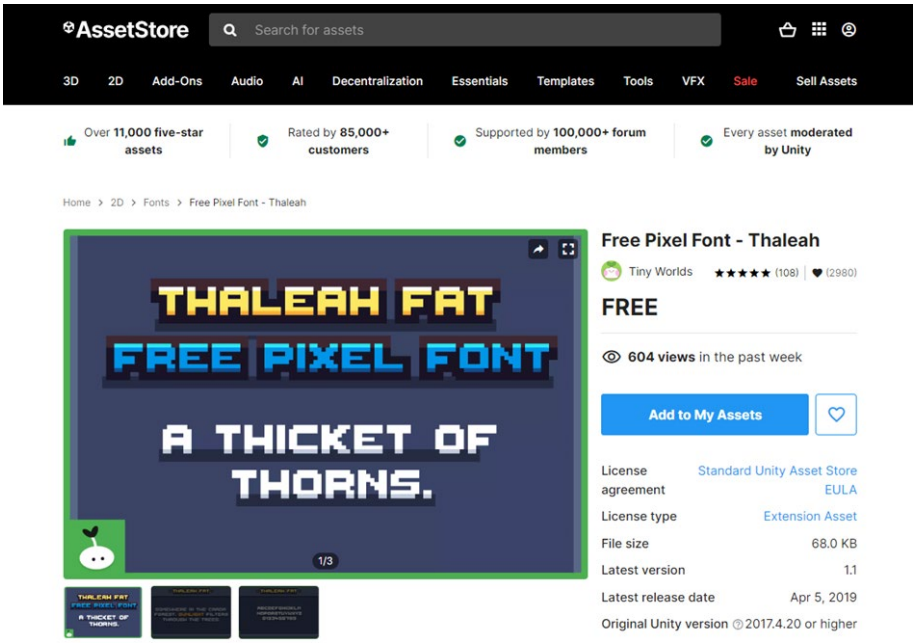
Jedan od ključnih aspekata Visual Studio-a je njegova integracija s programskim alatom Unity. Ova integracija omogućuje lako otvaranje i uređivanje Unity skripti izravno unutar Visual Studio-a, kao i sinkronizaciju promjena između ova dva alata. Također, Visual Studio omogućuje kreiranje i održavanje skripti za razne aspekte igre, uključujući kontrolu neprijatelja, upravljanje aktivnim oružjem i sustavom za napredovanje kroz valove, što je ključno za funkcionalnost igre.

Uz sve ove mogućnosti, Visual Studio doprinosi efikasnom i organiziranom razvoju igre, omogućujući programerima da se usmjere na kreiranje kvalitetnog i funkcionalnog sadržaja za igru. Korištenjem ovog alata, razvojni tim može osigurati visoku razinu točnosti i performansi u svim aspektima igre.

2.3. Unity trgovina resursa (eng. *Unity Asset Store*)

Unity trgovina resursa [3] (eng. *Unity Asset Store*) je ključan resurs za razvoj igara u programskom alatu Unity, nudeći raznoliku ponudu gotovih elemenata, od 2D i 3D modela, animacija, zvučnih efekata, do skripti i alata za optimizaciju. U sklopu ovog projekta, resursi korišteni u igri kao što su materijali i teksture preuzeti su sa Unity trgovine resursa, što je omogućilo brži i efikasniji razvoj igre. Korištenjem trgovine značajno je ubrzan proces razvoja ovog rada jer su već pripremljeni modeli likova, neprijatelja, oružja i efekti integrirani izravno u

projekt, bez potrebe za dodatnim dizajnom ili programiranjem tih elemenata. To je omogućilo usmjerenje na razvoj same mehanike igre, poput kontrola i sustava valova neprijatelja.



Slika 1: Unity trgovina resursa [3]

Unity trgovina resursa [3] pruža pristup visokokvalitetnim resursima koji su optimizirani za upotrebu unutar same okoline programskog alata Unity, čime se osigurava dosljednost i visoka kvaliteta u igri. Također, integracija preuzetih resursa u projekt je jednostavna i omogućava prilagodbu po potrebi, čime se dodatno olakšava razvoj.

Korištenje Unity trgovine resursa u ovom projektu nije samo smanjilo vrijeme razvoja, već je i omogućilo implementaciju naizgled profesionalnih vizualnih i ostalih grafičkih elemenata koji značajno poboljšavaju cjelokupno igračko iskustvo.

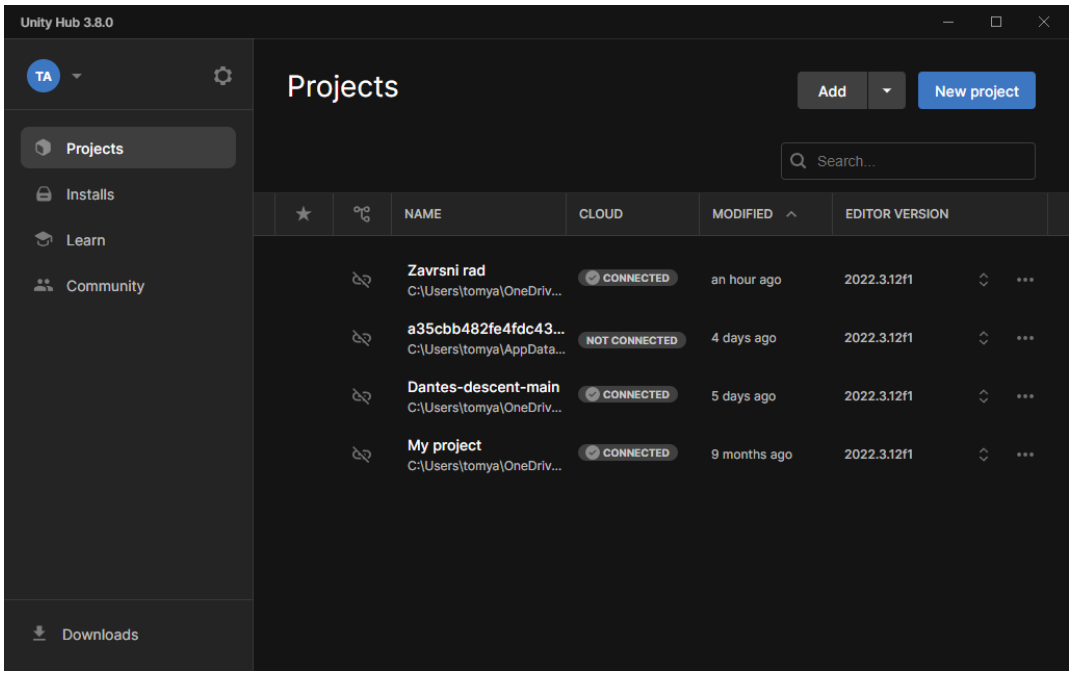
3. Razrada teme

U ovom poglavlju će se opisati i objasniti proces kreacije i razvoja 2D videoigre kojom se rad bavi. Prvo će se prikazati osnove programskog alata Unity kao i njegovo sučelje. Od početnog kreiranja projekta do manipuliranja pojedinim elementima sučelja. Zatim će biti cilj objasniti pojedine programske koncepte u C# programskom jeziku što će služiti kao snažan tehnički temelj za daljnji razvoj. Prikazat će se također i pojedine klase koje su korištene unutar samog projekta. Završno, prolazit će se kroz proces razvoja rada i njegovih elemenata od scena, grafičkih elemenata i animacija do naglašene programske i pozadinske logike korištene u samom projektu.

3.1. Korištenje programskog alata Unity

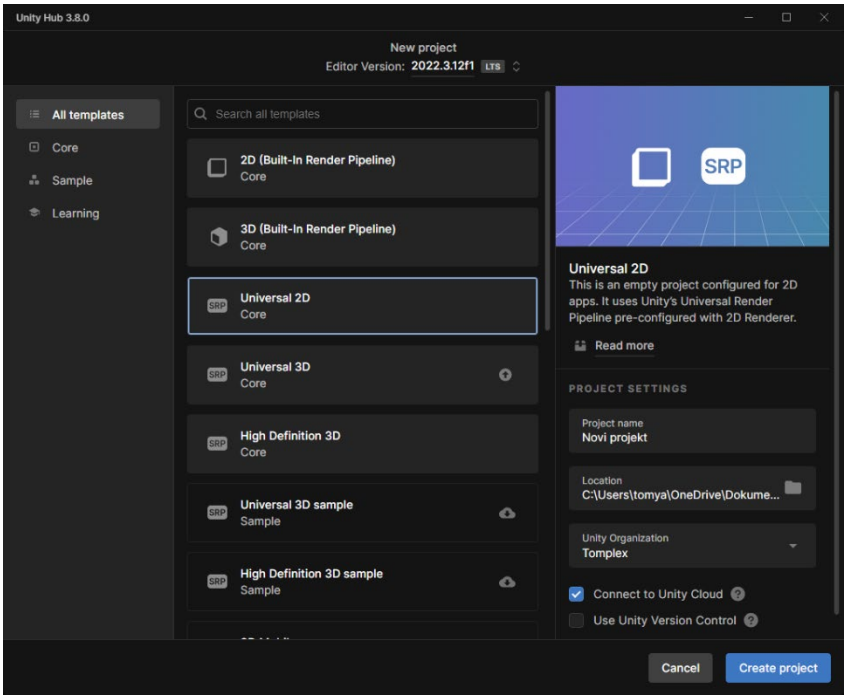
Prolaskom kroz ovo poglavlje je cilj objasniti kako se koristiti alatom i njegovim sučeljem od početka. Pokazat će se proces kreiranja novog projekta, postavljanje njegovih početnih postavki, te prikaz i demonstracija manipulacije nad bitnim elementima sučelja.

3.1.1. Kreacija novog projekta



Slika 2: Kreacija novog projekta – Novi projekt

U ovom slučaju biramo "Universal 2D" budući da se radi o 2D igri. UzimamTakođer, prilikom kreiranja projekta postavljaju se osnovne postavke poput rezolucije igre, ciljne platforme (PC, mobilni uređaji itd.) te kvaliteta grafičkih elemenata.



Slika 3: Kreacija novog projekta – Parametri projekta

3.1.2. Sučelje

Nakon kreiranja novog projekta, korisnik ulazi u radno sučelje Unity-a, koje se sastoji od nekoliko ključnih panela, od kojih svaki ima specifičnu ulogu u razvoju igre:

- Hierarchy: Prikazuje sve objekte unutar trenutne scene. Ovdje se kreiraju, organiziraju i upravljaju svi vidljivi elementi igre, poput igrača, neprijatelja, pozadine i objekata s kojima igrač može imati interakciju. Objekti se mogu grupirati unutar roditeljskih objekata, što olakšava organizaciju i kontrolu nad složenijim scenama.

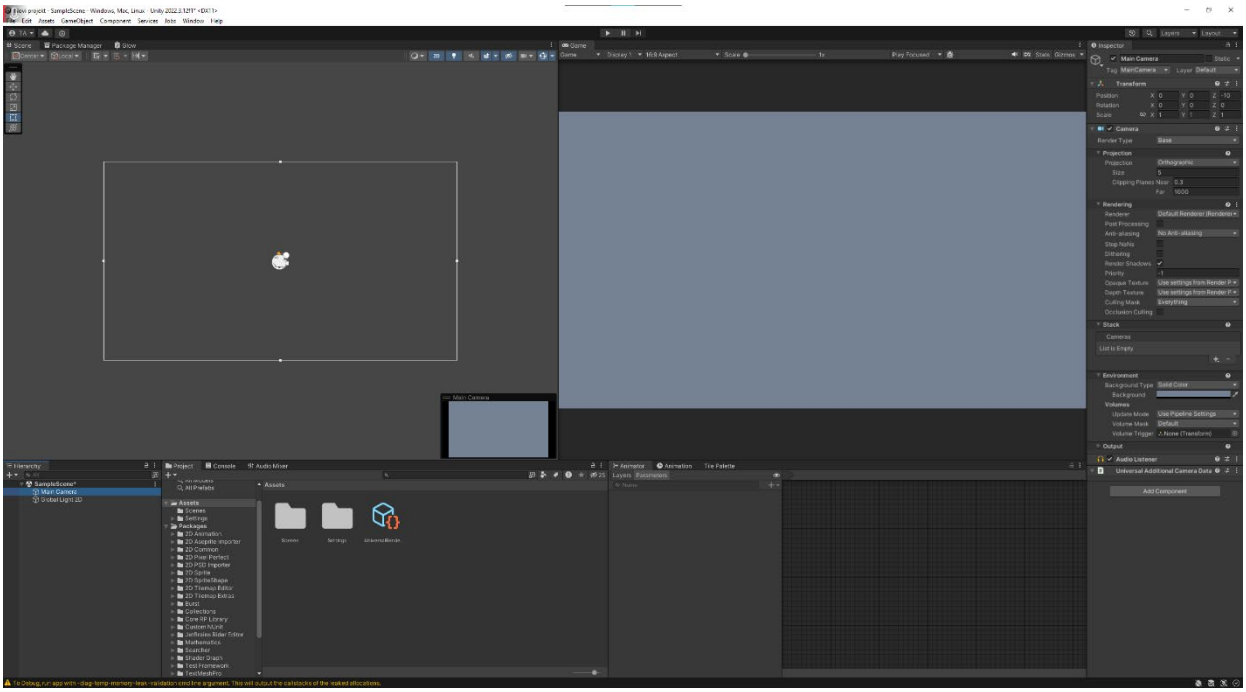
- Scene View: Omogućuje vizualno uređivanje i pregledavanje scene igre. Ovdje se postavljaju objekti u prostor, definiraju njihove pozicije, rotacije i veličine, te se upravlja kamerama i svjetlima.

- Game View: Prikazuje kako igra izgleda kada se pokrene, simulirajući konačni rezultat. Ova perspektiva omogućava korisniku da vidi konačni rezultat svih podešavanja unutar Scene View-a. Korištenjem Game View-a, korisnik može testirati funkcionalnost igre i analizirati kako se ponašaju.

- Inspector: Prikazuje svojstva odabranog objekta u sceni. Ovdje se mogu prilagoditi različiti atributi objekta, kao što su položaj, veličina, boja, te dodati komponente kao što su često korišteni sudarači (eng, *colliders*), fizičke komponente ili skripte. Inspector također omogućava korisniku prilagodbu postavki materijala, animacija i mogućnosti interaktivnosti objekata.

- Project: Ovaj panel sadrži sve resurse projekta, uključujući skripte, slike, zvukove i prefab resurse (gotovi objekti koji se mogu ponovno koristiti). Korisnik može jednostavno povući resurse iz Project panela u Scene View kako bi ih dodao u scenu ili u Inspector kako bi ih povezao s određenim objektima.

- Console Panel: Koristi se za ispis poruka i upozorenja tijekom izvođenja igre. Ovo je važan alat za otklanjanje grešaka, gdje se može pratiti rad skripti, greške i izvršavanje pojedinih funkcija.

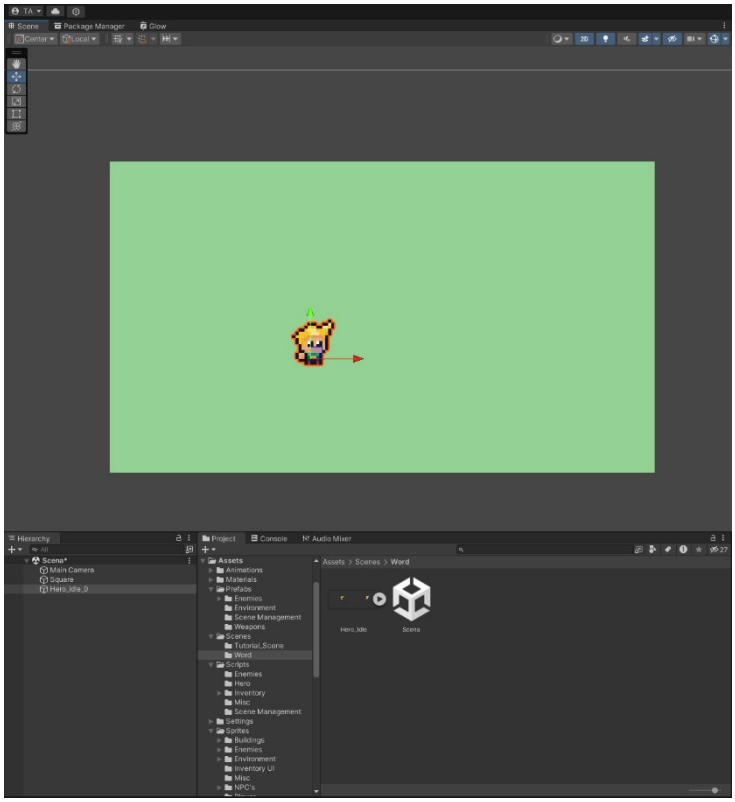


Slika 4: Početno sučelje programskog alata Unity

Svi ovi paneli su prilagodljivi te ih korisnik može premještati i prilagoditi prema vlastitim potrebama, čime se optimizira radni prostor za što brži i efikasniji razvoj igre.

3.1.3. Manipulacija objektima

Manipulacija objektima unutar scene ključni je dio rada u Unity-ju. Objekti poput likova, neprijatelja i pozadine mogu se dodavati iz Project panela jednostavnim povlačenjem u Scene View. Jednom kad su objekti postavljeni u scenu, mogu se dodatno prilagoditi unutar Inspector panela, gdje se mogu podešavati atributi kao što su položaj, rotacija, veličina ili dodavanje fizičkih komponenti (npr. Rigidbody2D za fiziku). Također, unutar Scene View-a korisnik može koristiti alate za pomicanje, rotiranje i skaliranje objekata.

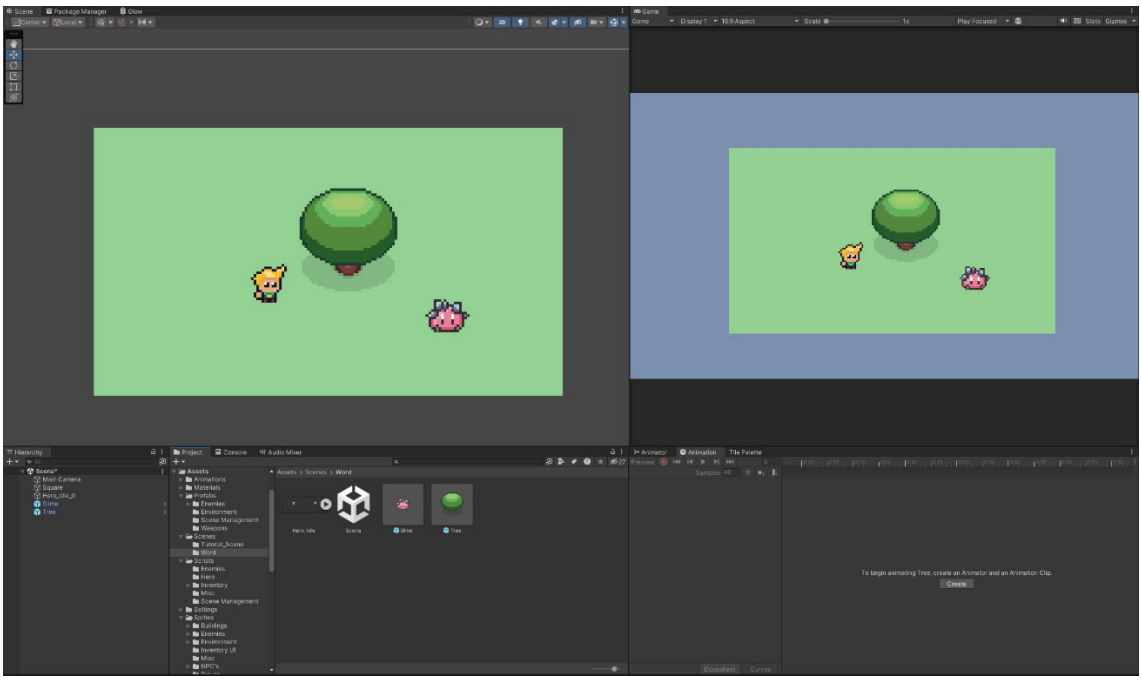


Slika 5: Dodavanje i podešavanje objekata u sceni

Nakon što su objekti dodani u scenu, korisnik može koristiti različite alate za njihovu manipulaciju. Move Tool omogućuje pomicanje objekata po X i Y osi unutar Scene View-a, dok Rotate Tool omogućava rotaciju objekata oko njihove osi, najčešće Z osi u 2D igrama. Scale Tool služi za prilagodbu veličine objekata po X i Y osi, što je korisno za postizanje željene proporcije elemenata unutar scene. Rect Tool omogućuje precizno uređivanje i pozicioniranje pravokutnih objekata, kao što su UI elementi ili pozadine. Ovi alati zajedno omogućuju korisniku potpunu kontrolu nad položajem, veličinom i izgledom objekata u igri, što čini proces izrade scene intuitivnim i fleksibilnim.

3.1.4.Postavljanje osnovne scene

Postavljanje osnovne scene uključuje definiranje pozadine, likova, neprijatelja i drugih ključnih elemenata igre. Prvo se postavlja pozadina koja definira vizualni ton igre. Zatim se dodaju likovi (igrač i neprijatelji), svaki s odgovarajućim skriptama i animacijama koje upravljaju njihovim ponašanjem. Cilj je stvoriti funkcionalnu scenu u kojoj igrač može vršiti interakcije s okolinom i neprijateljima.



Slika 6: Osnovna scena igre

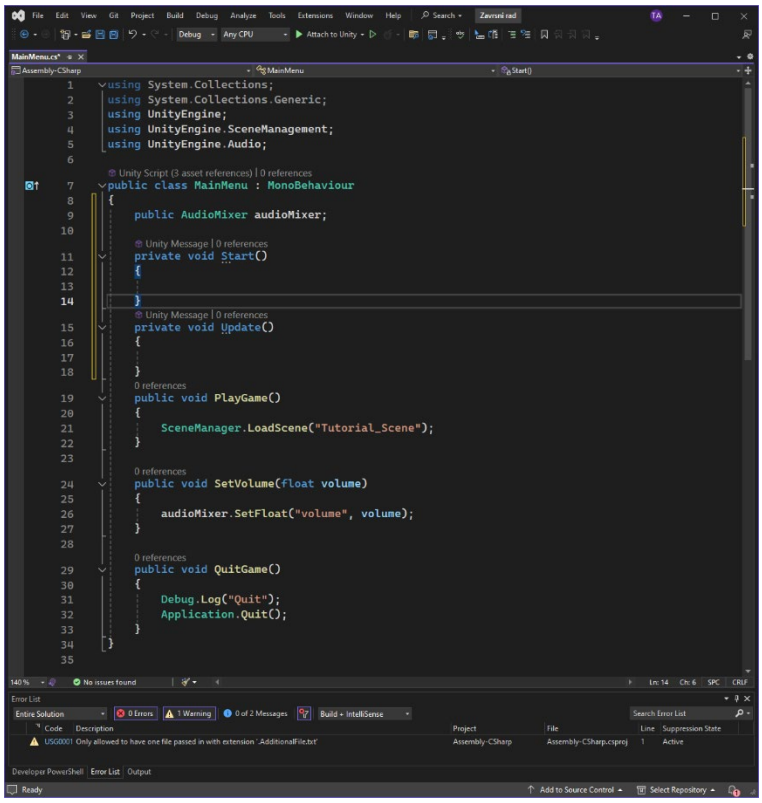
3.2. Programski koncepti

U ovom poglavlju detaljno će se objasniti osnovni programski koncepti korišteni u projektu, koji su ključni za funkcionalnost igre. Naglasak će biti na korištenju C# programskog jezika unutar Unity-a te na specifičnim konceptima i metodama koje omogućuju efikasno upravljanje objektima i komunikaciju između skripti.

3.2.1. Osnove C# skripti u Unity-u

C# je glavni programski jezik korišten u Unity-u za definiranje ponašanja objekata unutar igre. Skripte se povezuju s objektima kao komponente, što omogućuje da svaki objekt ima svoju vlastitu logiku i funkcionalnosti. Jedan objekt može sadržavati više skripti na sebi. Unity koristi `MonoBehaviour` klasu kao osnovu za sve skripte, što omogućuje pristup ključnim

metodama kao što su `Start()`, koja se izvršava prilikom pokretanja objekta odnosno skripte, i `Update()`, koja se poziva svakom iteracijom sličica (eng. *frames*) igre. Ove metode omogućuju stalnu kontrolu nad radom objekata i njihovim ponašanjem unutar igre.



Slika 7: Jednostavna skripta u okruženju Microsoft Visual Studio

3.2.2. Singleton klasa

Jedan od najznačajnijih klasa, a ujedno i sama skripta koja se koristi u projektu je Singleton. Singleton osigurava da od određene klase postoji samo jedna instanca tijekom trajanja igre, što je korisno za upravljanje globalnim stanjima ili resursima kao što su upravljači za igru i igrača, dinamični elementi korisničkog sučelja igre ili postavke [4]. Ključni element Singletona je `protected virtual void Awake()` metoda, koja se koristi kako bi se spriječilo kreiranja više instanci klase. Ova metoda osigurava da će samo prva kreirana instanca klase ostati aktivna, dok će sve ostale biti uništene [11]. Slijedi prikaz Singleton.cs skripte.

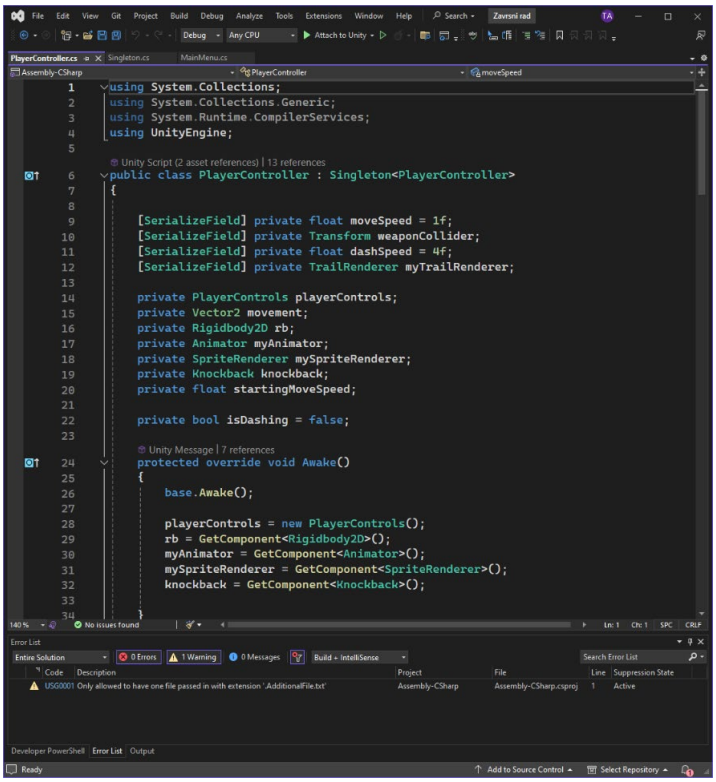
```
public class Singleton<T> : MonoBehaviour where T : Singleton<T>
{
    private static T instance;
    public static T Instance { get { return instance; } }
    protected virtual void Awake()
    {
        if(instance != null && this.gameObject != null)
```

```
        {
            Destroy(this.gameObject);
        }
        else
        {
            instance = (T)this;
        if(!gameObject.transform.parent)
        {
            DontDestroyOnLoad(gameObject);
        }
    }
}
```

Metoda `Awake()` koristi se umjesto `Start()` jer se poziva odmah po učitavanju objekta, osiguravajući da se instanca postavi čim se objekt učita u memoriju [11]. Modifikatori `protected` i `virtual` omogućuju da se metoda `Awake()` može naslijediti i prilagoditi u podklasama, zadržavajući osnovnu funkcionalnost Singletona.

3.2.3. Komunikacija između skripti

U projektima gdje se nalazi poveći broj skripti, važno je omogućiti da skripte međusobno komuniciraju i razmjenjuju podatke. U Unity-u to se postiže dohvaćanjem metoda ili varijabli iz jedne skripte u drugu. To se može postići na više načina, uključujući korištenje `GetComponent<>()` metode za pristup komponentama na objektima, ili direktno referenciranje drugih skripti putem javnih varijabli. Ova komunikacija omogućuje povezivanje različitih elemenata igre, kao što su interakcije između igrača i neprijatelja ili upravljanje globalnim postavkama igre.

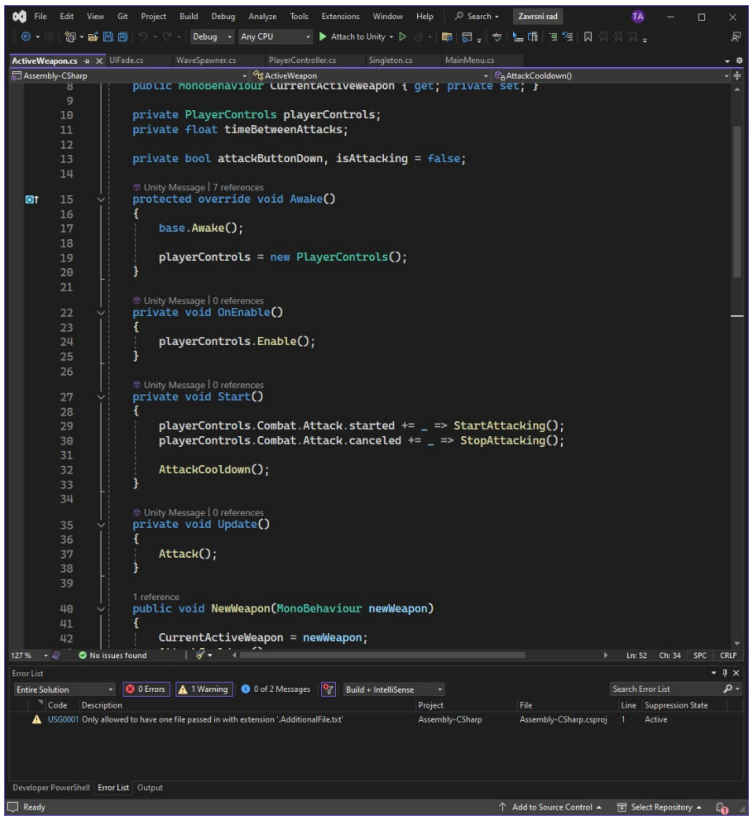


Slika 8: Serijalizacija polja i dohvaćanje komponenti

Serijalizacijom varijabli odnosno naredbom `[SerializeField]`, omogućujemo upravljanje postavljenom varijablom u samom Unity okruženju, dalje olakšajavajući rad unutar alata jer se atributi varijable mogu mijenjati bez ulaženja u kod i mijenjanja ručno [11].

3.2.4. Upravljanje događajima i upravljačima

Jedan od naprednijih aspekata programiranja u Unity-u je korištenje događaja (eng. *events*) i upravljača (eng. *handlers*). Događaji omogućuju da jedan dio koda obavijesti druge dijelove kada se nešto specifično dogodi, bez da direktno komuniciraju. Ovaj koncept je koristan za smanjenje povezanosti među skriptama, što povećava modularnost i održivost koda.

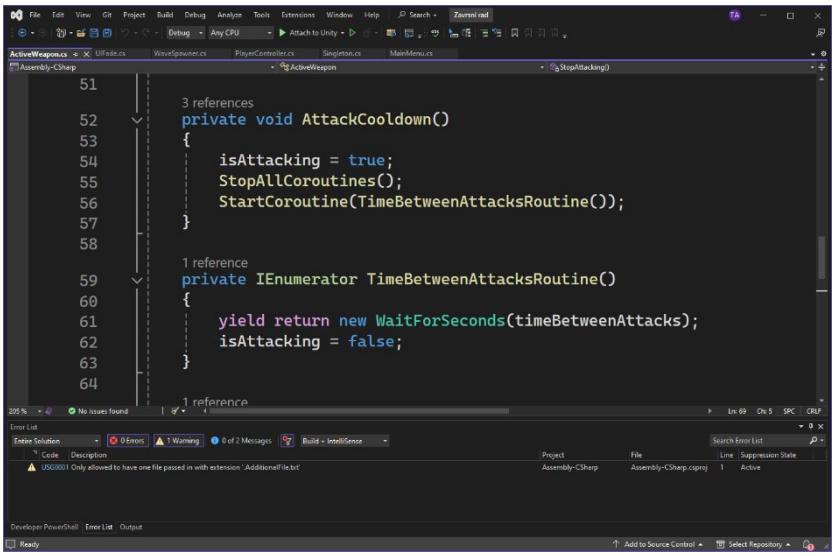


Slika 9: Sustav događaja u kodu

Primjer korištenja događaja je korištenje sustava događaja (eng. *event system*) `playerControls.Combat.Attack` koji omogućuje reagiranje na početak (`started`) i prekid (`canceled`) napada. Time se upravlja sama logika napadanja od strane igrača.

3.2.5. Korutine

Korutine su posebne metode u Unity-u koje omogućuju izvođenje operacija u više faza, što je korisno za zadatke koji se trebaju odvijati kroz više sličica (eng. *frames*). Korutine omogućuju da se određeni dio koda odgodi ili ponovi u toku igre bez blokiranja glavne petlje igre [5]. Ovo je posebno korisno za upravljanje vremenskim intervalima, animacijama i drugim asinkronim operacijama.

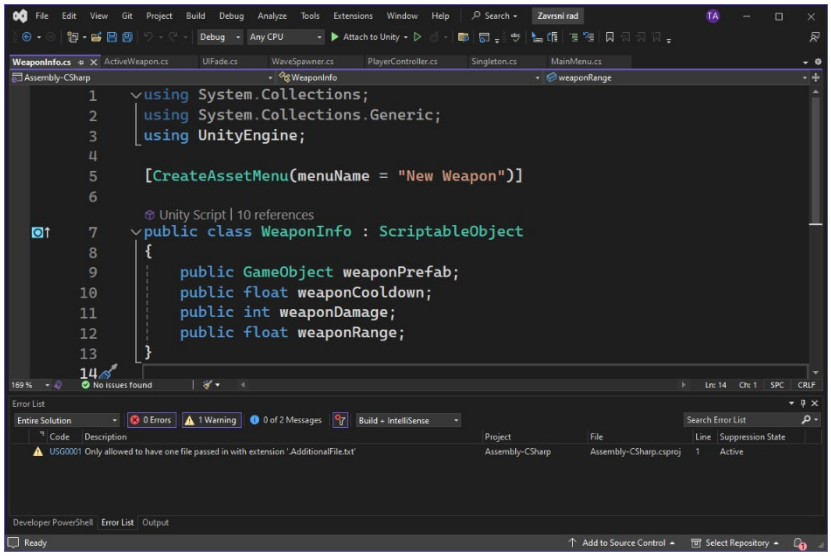


Slika 10: Prikaz korutine u Microsoft Visual Studio

Definira se kao metoda koje vraćaju tip `IEnumerator`. Unutar korutine, `yield return` se koristi za pauzu izvršavanja metode dok se ne zadovolji određeni uvjet ili dok ne prođe određeno vrijeme.

3.2.6. Scriptable objects

Scriptable objects su posebna vrsta objekata u Unity-u koja omogućuje pohranu podataka i konfiguracija bez potrebe za stvaranjem instanci klasa u sceni. Oni su vrlo korisni za pohranu podataka koji se dijele između različitih instanci ili objekata u igri, kao što su informacije o oružju i drugi podaci koji ne ovise o životnom ciklusu igre.



Slika 11: Scriptable object u Microsoft Visual Studio

3.3. Razvoj 2D videoigre pucanja i preživljavanja

Ovo poglavlje predstavlja ključne aspekte razvoja 2D videoigre pucanja i preživljavanja. Fokusirat će se na glavne komponente igre, uključujući dizajn scena, grafičke elemente, animacije, te programske i pozadinske logike. Ovdje će se detaljno obraditi svaki aspekt igre, od koncepta i planiranja do implementacije i testiranja.

3.3.1. Koncept i Planiranje

Osnovna ideja igre temelji se na konceptu preživljavanja kroz valove neprijatelja s ograničenim životnim bodovima. Igrač započinje igru s početnim životnim bodovima i suočava se s valovima neprijatelja koji postaju sve teži kako igra napreduje. Ako igrač izgubi sve životne bodove, vraća se na početnu scenu i mora ponovno pokušati preživjeti sve valove neprijatelja, počevši iznova. Ovaj ciklus pruža izazov i motivaciju igračima da usavrše svoje vještine i strategije za preživljavanje. Glavne mehanike igre uključuju:

- Kontrola Igrača: Igrač kontrolira lik s ograničenim životnim bodovima i regenerirajućom izdržljivošću, koji se kreće kroz različite scene i bori protiv valove neprijatelja.

- Borba: Igra nudi dinamičnu borbu u kojoj igrač mora koristiti različite taktike ne samo za svaki val, već i za svakog neprijatelja te uspješno preživjeti sve valove kao i uništiti sve neprijatelje.

- Preživljavanje: Cilj je preživjeti što duže moguće kroz valove neprijatelja. Kada igrač izgubi sve životne bodove, vraća se na početnu scenu i mora ponovno započeti ciklus igre.

- Napredovanje: Svaki val neprijatelja postaje teži s novim i različitim neprijateljima, zahtijevajući od igrača da prilagodi svoju strategiju i poboljša svoje vještine.

Igra koristi top-down 2D pixel art stil s ortografskom kamerom. Ovaj grafički stil omogućuje jednostavnu, ali bogatu vizualnu prezentaciju koja je lako prepoznatljiva i estetski privlačna. Pixel art stil doprinosi retro osjećaju igre i omogućuje detaljno oblikovanje likova i neprijatelja u jednostavnom, ali efektivnom obliku.

Slika 12: Drvo u stilu pixel art

Na početku razvoja igre izrađen je prototip koji je omogućio osnovno testiranje igrivih mehanika i dizajna. Prototip je korišten za evaluaciju osnovnih koncepata igre i prikupljanje povratnih informacija. Kroz iteracije su rađene prilagodbe i poboljšanja na temelju testiranja, kako bi se osiguralo da igra nudi uzbudljivo i uravnoteženo iskustvo.

3.3.2. Razvoj i dizajn scena

U poglavlju se opisuju tri ključne scene koje čine osnovu igre: scena glavnog izbornika, scena Sela, i scena glavne borbe. Svaka od ovih scena ima specifičnu ulogu u igri i doprinosi ukupnom iskustvu kroz različite mehanike i dizajn.

3.3.2.1. Scena glavnog izbornika

Scena glavnog izbornika predstavlja početnu točku igre i omogućava igraču da započne igru, postavi opcije, kao što je razina zvuka, ili izađe iz igre. Ova scena je statična i dizajnirana je s fokusom na jednostavnost i pristupačnost. Uključuje osnovne UI elemente poput gumba za početak igre, postavke i izlaz.

Slika 13: Scena glavnog izbornika

3.3.2.2. Scena sela

Scena sela služi kao uvod u igru i omogućava igraču da se upozna s osnovnim mehanikama i upravljanjem likom. Ova scena uključuje intuitivnu navigaciju pomoću nacrtanog puta koji vodi do portala. Igrač koristi ovaj portal za prelazak u glavnu borbenu scenu. Dizajn Sela uključuje elemente koji pomažu igraču da se orijentira i razumije osnovne kontrole prije nego što uđe u glavni dio igre.

Slika 14: Scena sela

3.3.2.3. Scena glavne borbe

Scena glavne borbe je ključni dio igre gdje se odvijaju valovi neprijatelja koje igrač mora eliminirati. Ova scena uključuje dinamične borbene mehanike i UI elemente koji informiraju igrača o trenutnom valu i napretku igre. Na kraju svakog vala, igrač prima obavijest o broju sljedećeg vala te kada taj val kreće, a nakon završetka igre dolazi i poruka pobjede i vraćanje na glavni izbornik.

Slika 15: Scena glavne borbe

3.3.2.4. Logika za portal i prelazak između scena

Prelazak između scena, posebno iz sela u glavnu borbenu scenu, upravlja se putem portala. Ovaj mehanizam omogućava igraču da jednostavno napusti uvodnu scenu i započne borbu. Logika za portal uključuje detekciju kolizije i upravljanje prijelazom između scena putem Unity Scene Manager-a. Slijedi prikaz skripte AreaExit.cs za prelazak između scena koja se nalazi na objektu izlaznog portala.

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

public class AreaExit : MonoBehaviour
{
    [SerializeField] private string sceneToLoad;
    [SerializeField] private string sceneTransitionName;

    private float waitToLoadTime = 1f;
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if(collision.gameObject.GetComponent<PlayerController>())
        {
             SceneManagement.Instance.
            SetTransitionName(sceneTransitionName);
            UIFade.Instance.FadeToBlack();
            StartCoroutine(LoadSceneRoutine());
        }
    }

    private IEnumerator LoadSceneRoutine()
    {
        yield return new WaitForSeconds(waitToLoadTime);
        SceneManager.LoadScene(sceneToLoad);
    }
}
```

Unutar samog Unity inspectora, u varijable `sceneToLoad` i `sceneTransitionName` unosimo naziv scene koja se učitava, u ovom slučaju glavna borbena scena, te na koji ulaz se učitava igrač. Slijedi prikaz skripte AreaEnter.cs za prelazak između scena koja se nalazi na objektu ulaznog portala.

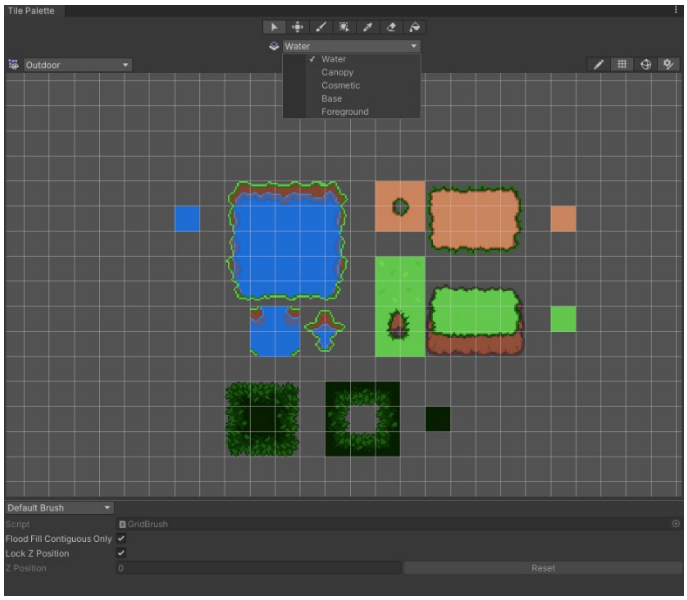
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AreaEntrance : MonoBehaviour
{
    [SerializeField] private string transitionName;

    private void Start()
    {
     if(transitionName==SceneManagement.Instance.sceneTransitionName)
        {
            PlayerController.Instance.transform.
             position              =              this.transform.position;
            CameraController.Instance.SetPlayerCameraFollow();
            UIFade.Instance.FadeToClear();
        }
    }
}
```

Isto tako, u varijabli `transitionName` postavlja se naziv ulaza na prijašnjoj sceni na koju se vraćamo. Ovakva logika s nazivima ulaza u scenama napravljena je kako bi se po potrebi u jednu scenu dodalo više portala te lagano rukovalo prijelazima iz jedne scene u drugu.

3.3.2.5. TileSet mehanizam i Rule Tiles

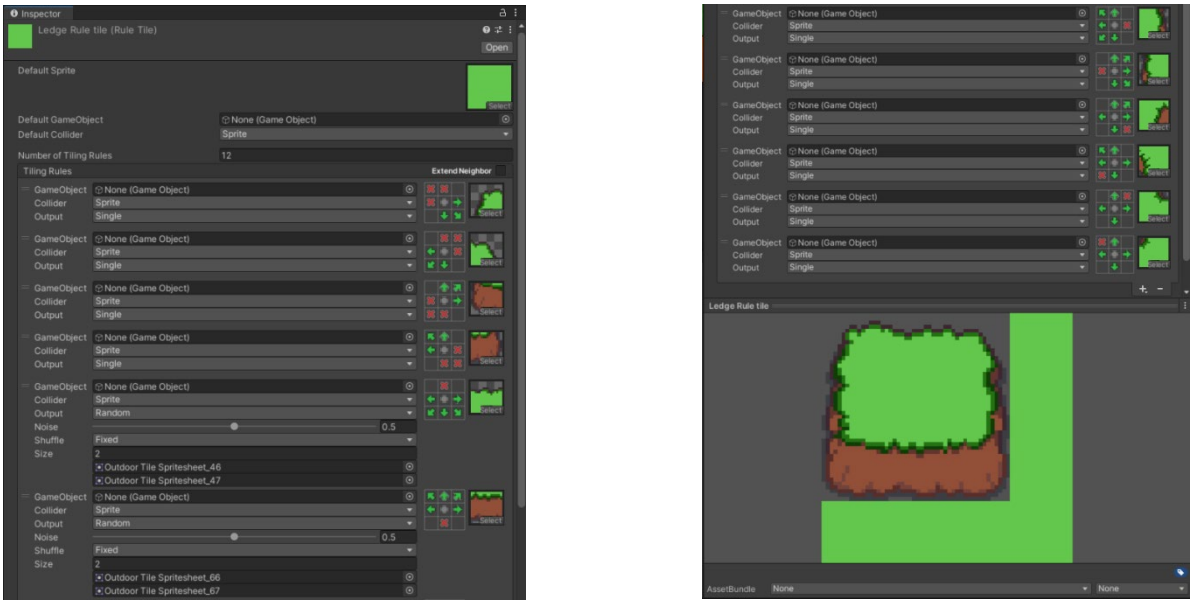
Koristimo set pločica (eng. *TileSet*) mehanizam u Unity-ju kao ključnu ulogu u dizajnu 2D okruženja igre. Ovaj sustav omogućava stvaranje i upravljanje mapama pločica (eng. *Tilemaps*) koristeći zbirke pločica, malih slikovnih blokova koji se ponavljaju i kombiniraju za stvaranje složenih scena [6]. TileSet je koristan jer optimizira grafiku igre koristeći jednu sliku koja sadrži sve pločice, što smanjuje broj rendera i poboljšava performanse igre. Osim toga, TileSet povećava efikasnost kod dizajna jer se vrlo brzo mogu postaviti pločice na Tilemapu, čime se ubrzava proces dizajniranja kao i osigurava dosljednost u grafičkom izgledu igre jer se iste pločice koriste na više mjesta, smanjujući mogućnost nesuglasica u vizualnom prikazu.

TileSet se koristi na sljedeći način: prvo se kreira TileSet dodavanjem slike koja sadrži sve tileove u jedan atlas. Nakon toga, Tilemap komponenta se postavlja kako bi koristila te tileove za izgradnju scene. Tilemap se može uređivati pomoću palete pločica (eng. *Tile Palette*), gdje dizajner može povući i ispustiti tileove na Tilemap kako bi stvorio željeni dizajn.



Slika 16: Tile Pallete u Unity-u

Pločice pravila [7] (eng. *Rule Tiles*) su specijalizirani tipovi pločica koje omogućuju automatsko postavljanje pločica na temelju definiranih pravila. Ovaj sustav je izuzetno koristan za stvaranje dinamičnih i vizualno privlačnih okruženja u igri. Rule Tiles automatski prilagođavaju pločice u odnosu na pločice oko njih, stvarajući besprijekorne prijelaze i dosljedne uzorke bez potrebe za ručnim postavljanjem svake pločice [7]. U projektu, Rule Tile automatski prilagođava postavljene pločice te za primjer planina automatski postavlja odgovarajuće zidove planina gdje bi se one trebale nalaziti i gdje bi se inače ručno postavili.



Slika 17: Pravila Rule Tiles u Unity-u

3.3.3. Grafički elementi

„Kada budete dizajnirali sučelje, suočit ćete se s dvije konfliktne želje; da igraču date što više opcija a da sučelje bude što jednostavnije. Kao i kod mnogih drugih aspekata dizajna igre, ključ je u balansu.“ (Konecki, M., 2023, str. 45)

Svi korišteni spriteovi preuzeti su s Unity Asset Store-a [3], dok su animacije ručno izrađene unutar Unity-a pomoću ručnih animacija. Na taj način, iako se koriste gotovi vizualni elementi, animacije likova i objekata u igri ostaju prilagođene specifičnim potrebama projekta.

Pozadina igre zadržava isti pixel art stil kao i spriteovi, čime se osigurava dosljednost vizualnog identiteta. Jedan od zanimljivijih aspekata dizajna pozadine je sloj krošnje koji koristi parallax skriptu. Ova skripta omogućuje da se krošnja kreće u skladu s kretanjem igrača, stvarajući iluziju dubine i osjećaj da igrač gleda izvan manjih prostora, što dodaje dimenziju svijetu igre. Slijedi prikaz skripte Parallax.cs koja se nalazi na Canopy objektu koji zadnji sloj u tilemap sustavu.

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

public class Parallax : MonoBehaviour
{
    [SerializeField] private float parallaxAmount = -0.15f;

    private Camera cam;
    private Vector2 startPos;
    private Vector2 travel => (Vector2)cam.
    transform.position-startPos;

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

    private void Start()
    {
        startPos = transform.position;
    }

    private void FixedUpdate()
    {
        transform.position = startPos + travel * parallaxAmount;
    }
}
```

U ovom kodu, pozicija pozadine pomiče se proporcionalno udaljenosti koju je kamera prešla od početne pozicije, prilagođeno pomoću varijable `parallaxAmount`. Na taj način,

pozadina se kreće sporije od prednjih elemenata igre, stvarajući osjećaj da se nalazi dalje u prostoru

Korisničko sučelje (skr. *UI*) također prati pixel art stil, uključujući tekstove prikazane igraču. UI elementi informiraju igrača o ključnim aspektima igre, poput trenutnog stanja zdravlja (skr. *HP*) i izdržljivosti (eng. *stamina*), koja se koristi prilikom mehanizma skakanja. Uz to, UI nudi jednostavan izbornik za podešavanje opcija i jednostavan inventar koji omogućuje odabir između tri oružja, uz prikaz trenutno aktivnog oružja. Ovi elementi čine sučelje intuitivnim i preglednim, osiguravajući lakoću korištenja i jasnoću potrebnih informacija tijekom igre. Slijedi prikaz dijela koda u skripti PlayerHealth.cs koja služi za dinamičnu promjenu elementa stanja zdravlja u korisničkom sučelju.

```
public void TakeDamage(int damageAmount, Transform hitTransform)
    {
        if (!canTakeDamage)
        {
            return;
        }
        canTakeDamage = false;
        currentHealth -= damageAmount;
        UpdateHealthSlider();
    }

   private void UpdateHealthSlider()
    {
        if(healthSlider == null)
        {
            healthSlider = GameObject.
            Find("Health Slider").GetComponent<Slider>();
        }
        healthSlider.maxValue = maxHealth;
        healthSlider.value = currentHealth;
    }
}
```

U navedenom kodu, metoda `UpdateHealthSlider()` poziva se u metodi za registraciju nanosa šteta na igrača `TakeDamage()`. Svojim pozivom, kako je klizni element stanja zdravlja postavljen da se manipulira u inkrementima cijelih brojeva od 0 do 5, ažurira stanje klizača sa aktualnom brojkom stanja zdravlja igrača `currentHealth`. Slijedi prikaz djela koda u skripti ActiveInventory.cs za promjenu prikaza odabranog oružja u elementu sučelja za inventar.

```
  private void Start()
    {
        PlayerControls.Inventory.Keyboard.
        performed += ctx => ToggleActiveSlot((int)ctx.
```

```
        ReadValue<float>());
    }

    private void OnEnable()
    {
        PlayerControls.Enable();
    }

    public void EquipStartingWeapon()
    {
        ToggleActiveHighlight(0);
    }

    private void ToggleActiveSlot(int numValue)
    {
        ToggleActiveHighlight(numValue - 1);
        Debug.Log(numValue);
    }

    private void ToggleActiveHighlight(int indexNum)
    {
        activeSlotIndexNum = indexNum;

        foreach (Transform inventorySlot in this.transform)
        {
            inventorySlot.GetChild(0).gameObject.SetActive(false);
        }

        this.transform.GetChild(indexNum).
        GetChild(0).gameObject.SetActive(true);

        ChangeActiveWeapon();
    }

    private void ChangeActiveWeapon()
    {
        if(PlayerHealth.Instance.IsDead) { return; }

        if(ActiveWeapon.Instance.CurrentActiveWeapon != null)
        {
            Destroy(ActiveWeapon.Instance.
```

```
        CurrentActiveWeapon.gameObject);
    }

    Transform childTransform = transform.
    GetChild(activeSlotIndexNum);
    InventorySlot inventorySlot = childTransform.
    GetComponentInChildren<InventorySlot>();
    WeaponInfo weaponInfo = inventorySlot.GetWeaponInfo();
    GameObject weaponToSpawn = weaponInfo.weaponPrefab;


    if (weaponInfo = null)
    {
        ActiveWeapon.Instance.WeaponNull();
        return;
    }
```

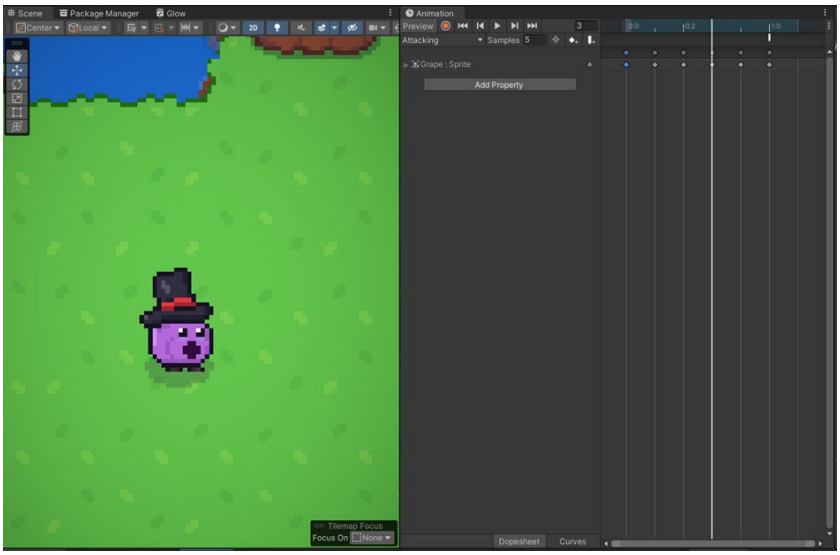
Ukratko, Ovaj kod upravlja promjenom aktivnog oružja u UI elementu tako što bilježi promjenu putem korisničkog unosa na tipkovnici, isključuje prikaz svih oružja u inventaru, aktivira odabrano oružje na temelju pozicije slike u redu te zamjenjuje prethodno aktivno oružje u igri novim.

3.3.4. Animacije

Animacije [8] su ključni element u stvaranju dinamičnog i interaktivnog igračkog iskustva, pa je tako posebna pažnja posvećena njihovoj izradi i integraciji putem Unity Animator Controller-a.

3.3.4.1. Kreacija i uvoz animacijskih klipova

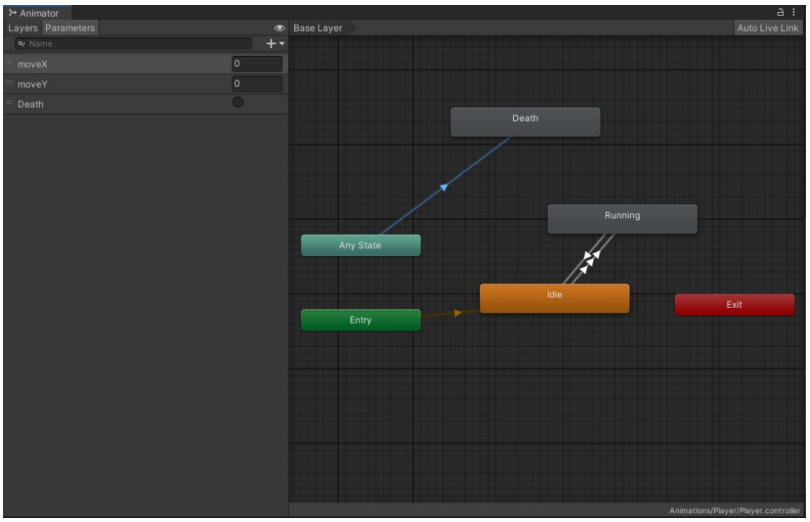
Sve animacije u igri kreirane su ručno unutar Unity-a. Svaki animirani objekt, uključujući likove, koristi zasebne animacijske klipove koji se sastoje od različitog broja sličica u sekundi, ovisno o kompleksnosti animacije. Na primjer, generalno animacija trčanja uključuju više sličica, dok stajanje na mjestu kao jednostavan pokret sastoji se od manje sličica. Svaka animacija se sastoji od nekog broja vrlo sličnih sličica koje imaju minimalne promjene između sebe, u slijedu stvarajući dojam da se objekt kreće. Također, brzina animacija varira, što omogućuje da se određene animacije odvijaju brže ili sporije, ovisno o potrebama igre [8].



Slika 18: Animacija u Unity-u

3.3.4.2. Postavljanje Animator Controller-a

Animator Controller koristi se za upravljanje svim animacijama u igri. Svaki objekt koji zahtijeva animaciju, poput glavnog lika ili neprijatelja, ima svoj zasebni Animator Controller koji upravlja prijelazima između različitih animacijskih stanja. Animator omogućuje definiranje stanja, poput "trčanje", "skakanje" ili "napad", te povezivanje ovih stanja s odgovarajućim animacijama.

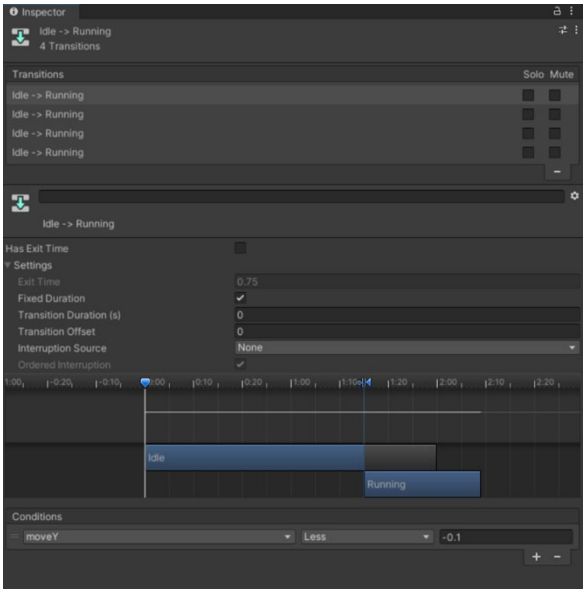


Slika 19: Animator Controller glavnog lika

Koriste se parametri kako bi upravljali prijelazima između stanja, a prijelazi se pokreću ovisno o korisnikovim interakcijama ili događajima unutar igre. Animacija stajanja prelazi u animaciju trčanja kada igrač pritisne određeni tipku, a u trenutku smrti igrača, prebacuje se u animaciju smrti ne bitno u kojoj animaciju bio trenutku prijelaza.

3.3.4.3. Upravljanje prijelazima i animacijskim stanjima

Prijelazi između animacija u igri ostvaruju se pomoću parametara unutar Animator Controller-a. Primjer glavnog lika koristit će se za prikaz složenih prijelaza između stanja, kao što su "Idle", "Running" i "Death". Svaka animacija povezana je s određenim parametrima koji definiraju kada će se stanje promijeniti, a Animator kontrolira glatke prijelaze između animacija kako bi osigurao prirodan i fluidan pokret likova.



Slika 20: Animator Controller glavnog lika

Parametar poput "moveY" aktivira se ovisno o igračevim akcijama, to jest kada igrač pomakne glavnog lika, a u trenutku kada se to dogodi Animator Controller stvara tranziciju iz “Idle“ u “Running“ animaciju. Tu se mogu postaviti ugrađeni parametri za tranziciju kao što je vrijeme same tranzicije između animacija te želimo li da izlazna animacija ima vrijeme izlaska.

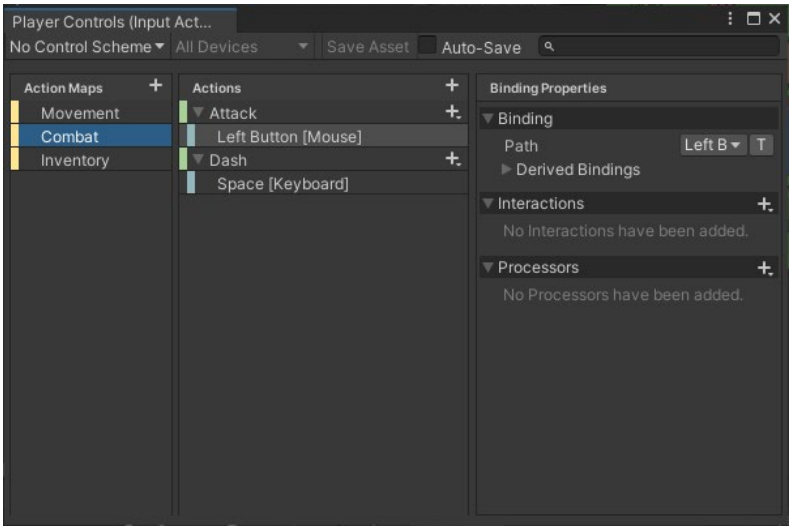
3.3.5. Programska logika

U ovom poglavlju opisat ćemo ključne programske elemente igre, uključujući upravljanje igračevim pokretima, logiku neprijatelja, sustav valova, te međusobne interakcije između neprijatelja i igrača.

3.3.5.1. Upravljanje igrača

Umjesto da su pokreti igrača ručno kodirani, koristi se sučelje Input Action Importer odnosno Action Maps koje su gledane kao normalna skripta, a su podijeljene u nekoliko logičnih cjelina: Movement, Combat i Inventory. Ova podjela omogućuje jasniju organizaciju kontrola, što olakšava upravljanje različitim aspektima igre. Unutar Movement mape kontroliraju se osnovno kretanje igrača po sceni, dok Combat omogućuje izvođenje napada te

izvođenje skoka, a Inventory upravlja inventarom. Ovaj sustav je fleksibilan i omogućuje jednostavno dodavanje novih kontrola ili izmjene postojećih.



Slika 21: Input Action Importer

3.3.5.2. Logika Igrača

Upravljačka skripta za igrača PlayerController.cs implementira osnovnu funkcionalnost kretanja i akcija igrača koristeći objašnjenu komponentu Player Controls (Input Action Importer). Također, koristi se `Awake()` metoda za inicijalizaciju potrebnih komponenti kao što su Rigidbody2D, Animator, i SpriteRenderer.

```
public class PlayerController : Singleton<PlayerController>
{

    [SerializeField] private float moveSpeed = 1f;
    [SerializeField] private Transform weaponCollider;
    [SerializeField] private float dashSpeed = 4f;
    [SerializeField] private TrailRenderer myTrailRenderer;

    private PlayerControls playerControls;
    private Vector2 movement;
    private Rigidbody2D rb;
    private Animator myAnimator;
    private SpriteRenderer mySpriteRenderer;
    private Knockback knockback;
    private float startingMoveSpeed;
    private bool isDashing = false;

    protected override void Awake()
    {
        base.Awake();
        playerControls = new PlayerControls();
        rb = GetComponent<Rigidbody2D>();
        myAnimator = GetComponent<Animator>()   ;
        mySpriteRenderer = GetComponent<SpriteRenderer>();
        knockback = GetComponent<Knockback>();
    }
```

```
private void Start()
{
    playerControls.Combat.Dash.performed += _ => Dash();
    startingMoveSpeed = moveSpeed;
    ActiveInventory.Instance.EquipStartingWeapon();
}

private void OnEnable()
{
    playerControls.Enable();
}

private void OnDisable()
{
    playerControls.Disable();
}

private void Update()
{
    PlayerInput();
}

private void FixedUpdate()
{
    AdjustPlayerFacingDirection();
    Move();
}

public Transform GetWeaponCollider()
{
    return weaponCollider;
}

private void PlayerInput()
{
    movement = playerControls.Movement.Move.ReadValue<Vector2>();
    myAnimator.SetFloat("moveX", movement.x);
    myAnimator.SetFloat("moveY", movement.y);
}

private void Move()
{
    if (knockback.GettingKnockedBack || PlayerHealth.Instance.IsDead)
    {
        return;
    }

    rb.MovePosition(rb.position + movement * (moveSpeed *  Time.
    fixedDeltaTime));
}

private void Dash()
{
    if(!isDashing && Stamina.Instance.CurrentStamina > 0)
    {
        Stamina.Instance.UseStamina();
        isDashing = true;
        moveSpeed *= dashSpeed;
        myTrailRenderer.emitting = true;
        StartCoroutine(EndDashRoutine());
    }
```

```
    }

    private IEnumerator EndDashRoutine()
    {
        float dashTime = 0.2f;
        float dashCD = 0.3f;
        yield return new WaitForSeconds(dashTime);
        moveSpeed = startingMoveSpeed;
        myTrailRenderer.emitting = false;
        yield return new WaitForSeconds(dashCD);
        isDashing = false;
    }

    private void AdjustPlayerFacingDirection()
    {
        Vector3 mousePos = Input.mousePosition;
        Vector3 playerScreenPoint = Camera.main.
        WorldToScreenPoint(transform.position);

        if (mousePos.x < playerScreenPoint.x)
        {
            mySpriteRenderer.flipX = true;
        }
        else
        {
            mySpriteRenderer.flipX = false;
        }
    }
}
```

U Update metodi, `PlayerInput()` bilježi korisnički ulaz i postavlja varijable koje utječu na animaciju kretanja. Metoda `Move()` koristi `FixedUpdate()` za precizno pomicanje igrača, dok metoda `Dash()` omogućava igraču da izvede brzi pomak, s posebnim efektima poput tragova koji se pojavljuju tijekom skoka. Funkcionalnost skoka uključuje promjenu brzine i privremeno omogućavanje vizualnih efekata putem TrailRenderer komponente. Metoda `AdjustPlayerFacingDirection()` osigurava da se igrač pravilno okreće prema smjeru pokazivača miša po svojoj.

Dalje, ActiveWeapon.cs skripta upravlja trenutnim aktivnim oružjem igrača i obavlja napade. U `Start()` metodi se postavljaju događaji za početak i zaustavljanje napada. `Update()` metoda provjerava stanje napada i pokreće napade ako su uvjeti zadovoljeni. Skripta koristi korutine za upravljanje vremenom između napada kako bi spriječila prekomjerno napadanje i osigurala pravilno hlađenje oružja.

```
public class ActiveWeapon : Singleton<ActiveWeapon>
{

    public MonoBehaviour CurrentActiveWeapon { get; private set; }

    private PlayerControls playerControls;
```

```
private float timeBetweenAttacks;
private bool attackButtonDown, isAttacking = false;

protected override void Awake()
{
    base.Awake();
    playerControls = new PlayerControls();
}

private void OnEnable()
{
    playerControls.Enable();
}

private void Start()
{
    playerControls.Combat.Attack.started += _ => StartAttacking();
    playerControls.Combat.Attack.canceled += _ => StopAttacking();
    AttackCooldown();
}

private void Update()
{
    Attack();
}

public void NewWeapon(MonoBehaviour newWeapon)
{
    CurrentActiveWeapon = newWeapon;
    AttackCooldown();
    timeBetweenAttacks = (CurrentActiveWeapon as IWeapon).
    GetWeaponInfo().weaponCooldown;
}

public void WeaponNull()
{
    CurrentActiveWeapon = null;
}

private void AttackCooldown()
{
    isAttacking = true;
    StopAllCoroutines();
    StartCoroutine(TimeBetweenAttacksRoutine());
}

private IEnumerator TimeBetweenAttacksRoutine()
{
    yield return new WaitForSeconds(timeBetweenAttacks);
    isAttacking = false;
}

private void StartAttacking()
{
    attackButtonDown = true;
}

private void StopAttacking()
{
    attackButtonDown = false;
}
```

```
    private void Attack()
    {
        if (attackButtonDown && !isAttacking && CurrentActiveWeapon)
        {
            AttackCooldown();
            (CurrentActiveWeapon as IWeapon).Attack();
        }
    }

}
```

Skripta PlayerHealth.cs prati zdravlje igrača i upravlja posljedicama gubitka zdravlja. Ako igrač primi štetu, metoda `TakeDamage()` smanjuje zdravlje i pokreće vizualne i zvučne efekte poput trešnje ekrana i efekt praksa. Skripta također upravlja metodom `CheckIfPlayerDeath()`, koja provjerava hoće li igrač umrijeti te, ako je to slučaj, pokreće animaciju smrti te učitava početnu scenu nakon kratke odgode.

```
public class PlayerHealth : Singleton<PlayerHealth>
{
    [SerializeField] private int maxHealth = 3;
    [SerializeField] private float knockbackPlayerAmount = 10f;
    [SerializeField] private float damageRecoveryTime = 1f;

    public bool IsDead {  get; private set; }

    private Slider healthSlider;
    private Knockback knockback;
    private Flash Flash;
    private int currentHealth;
    private bool canTakeDamage = true;

    const string START_LEVEL_TEXT = "Tutorial_Scene";
    readonly int DEATH_HASH = Animator.StringToHash("Death");

    protected override void Awake()
    {
        base.Awake();
        knockback = GetComponent<Knockback>();
        Flash = GetComponent<Flash>();
    }

    private void Start()
    {
        IsDead = false;
        currentHealth = maxHealth;
        UpdateHealthSlider();

    }

    private void OnCollisionStay2D(Collision2D collision)
    {
        EnemyAI enemy = collision.gameObject.GetComponent<EnemyAI>()
        if (enemy)
        {
            TakeDamage(1, collision.transform);
        }
```

```
}

public void HealPlayer()
{
    if(currentHealth < maxHealth)
    {
        currentHealth += 1;
        UpdateHealthSlider();
    }
}

public void TakeDamage(int damageAmount, Transform hitTransform)
{
    if (!canTakeDamage)
    {
        return;
    }
    ScreenShakeManager.Instance.ShakeScreen();
    knockback.GetKnockedBack(hitTransform,
    knockbackPlayerAmount);
    StartCoroutine(Flash.FlashRoutine());
    canTakeDamage = false;
    currentHealth -= damageAmount;
    Debug.Log(currentHealth);
    StartCoroutine(DamageRecoveryRoutine());
    UpdateHealthSlider();
    CheckIfPlayerDeath();
}

private void CheckIfPlayerDeath()
{
    if(currentHealth <= 0 && !IsDead)
    {
        IsDead=true;
        Destroy(ActiveWeapon.Instance.gameObject);
        currentHealth = 0;
        GetComponent<Animator>().SetTrigger(DEATH_HASH);
        StartCoroutine(DeathLoadSceneRoutine());
    }
}

private IEnumerator DeathLoadSceneRoutine()
{
    yield return new WaitForSeconds(2);
    Destroy(gameObject);
    Stamina.Instance.ReplenishStaminaOnDeath();
    SceneManager.LoadScene(START_LEVEL_TEXT);
}

private IEnumerator DamageRecoveryRoutine()
{
    yield return new WaitForSeconds(damageRecoveryTime);
    canTakeDamage = true;
}

private void UpdateHealthSlider()
{
    if(healthSlider == null)
    {
        healthSlider = GameObject.
        Find("Health Slider").GetComponent<Slider>();
```

```
        }
        healthSlider.maxValue = maxHealth;
        healthSlider.value = currentHealth;
    }
}
```

Naposlijetku, Skripta DamageSource.cs koristi se za primjenu štete neprijateljima. Kada se oružje igrača sudari s neprijateljem, ugrađena metoda `OnTriggerEnter2D()` uzima iznos štete iz trenutnog aktivnog oružja i primjenjuje ga na neprijatelja.

```
public class DamageSource : MonoBehaviour
{
    private int damageAmount;
    private void Start()
    {
        MonoBehaviour currentActiveWeapon = ActiveWeapon.
        Instance.CurrentActiveWeapon;
        damageAmount = (currentActiveWeapon as IWeapon).
        GetWeaponInfo().weaponDamage;
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.gameObject.GetComponent<EnemyHealth>())
        {
            EnemyHealth enemyHealth = collision.
            gameObject.GetComponent<EnemyHealth>();
            enemyHealth.TakeDamage(damageAmount);
        }
    }
}
```

3.3.5.3. Logika neprijatelja

Logika neprijatelja podijeljena je na nekoliko skripti koje definiraju njihovo ponašanje, napade i interakcije s igračem.

Prva skripta, EnemyAI.cs, upravlja osnovnim ponašanjem neprijatelja. Neprijatelj nasumično luta po mapi (eng. *roaming*) sve dok igrač ne uđe u njegov domet, nakon čega prelazi u stanje napada. Ova skripta također kontrolira povremene promjene smjera i vremenske intervale napada. Ako neprijatelj dosegne domet za napad, metoda `Attack()` inicira napad, a neprijatelj prestaje napadati dok ne istekne trajanje varijable `attackCooldown`.

```
public class EnemyAI : MonoBehaviour
{
    [SerializeField] private float roamChangeDirTimer = 2f;
    [SerializeField] private float attackRange = 0f;
    [SerializeField] private MonoBehaviour enemyType;
    [SerializeField] private float attackCooldown = 2f;
    [SerializeField] private bool stopMovingWhileAttacking = false;

    private bool canAttack = true;
```

```
private enum State
{
    Roaming,
    Attacking
}
private Vector2 roamPosition;
private float timeRoaming = 0f;
private State state;
private EnemyPathfinding enemyPathfinding;
private void Awake()
{
    enemyPathfinding = GetComponent<EnemyPathfinding>();
    state = State.Roaming;
}
private void Start()
{
   roamPosition = GetRoamingPosition();
}
private void Update()
{
    MovementStateControl();
}
private void MovementStateControl()
{
    switch(state)
    {
        default:
            case State.Roaming:
                Roaming();
            break;

            case State.Attacking:
                Attacking();
            break;
    }
}
private void Roaming()
{
    timeRoaming += Time.deltaTime;
    enemyPathfinding.MoveTo(roamPosition);

    if(Vector2.Distance(transform.position,
  PlayerController.Instance.transform.position) < attackRange)
    {
        state = State.Attacking;
    }

    if(timeRoaming > roamChangeDirTimer)
    {
        roamPosition = GetRoamingPosition();
    }
}
private void Attacking()
{
    if (Vector2.Distance(transform.position,
    PlayerController.Instance.transform.position) > attackRange)
    {
        state = State.Roaming;
    }

    if (attackRange != 0 && canAttack)
```

```
        {
            canAttack = false;
           (enemyType as IEnemy).Attack();

            if (stopMovingWhileAttacking)
            {
                enemyPathfinding.StopMoving();
            }
            else
            {
                enemyPathfinding.MoveTo(roamPosition);
            }

            StartCoroutine(AttackCooldownRoutine());
        }
    }
    private IEnumerator AttackCooldownRoutine()
    {
        yield return new WaitForSeconds(attackCooldown);
        canAttack = true;
    }
    private Vector2 GetRoamingPosition()
    {
        timeRoaming = 0f;
        return new Vector2(Random.Range(-1f, 1f),
        Random.Range(-1f, 1f)).normalized;
    }
}
```

Druga skripta, EnemyHealth.cs, upravlja neprijateljskim zdravljem. Kada neprijatelj primi štetu, njegovo zdravlje se smanjuje. Skripta koristi mehaniku povratnog udarca (knockback), odnosno metodu iz skripte Knockback.cs i vizualne efekte kako bi signalizirala štetu. Ako neprijatelj izgubi svo postavljeno zdravlje, skripta uništava neprijatelja.

```
public class EnemyHealth : MonoBehaviour
{
    [SerializeField] private int startingHealth = 3;

    private int currentHealth;
    private Knockback knockback;
    private Flash flash;
    private void Awake()
    {
        flash = GetComponent<Flash>();
        knockback = GetComponent<Knockback>();
    }
    private void Start()
    {
        currentHealth = startingHealth;
    }
    public void TakeDamage(int damage)
    {
        currentHealth -= damage;
        knockback.GetKnockedBack(PlayerController.
         Instance.transform, 12f);
        StartCoroutine(flash.FlashRoutine() );
        StartCoroutine(CheckDetectDeathRoutine());
```

```
    }
    private IEnumerator CheckDetectDeathRoutine()
    {
        yield return new WaitForSeconds(flash.GetRestoreMatTime());
        DetectDeath();
    }
    private void DetectDeath()
    {
        if(currentHealth <= 0)
        {
            Destroy(gameObject);
        }
    }
}
```

Treća skripta, Knockback.cs bavi se samo mehanikom povratnog udarca, odnosno u trenutku nanose šteta bilo igraču ili neprijatelju, pomiče objekt na kojem je skripta instancirana suprotno od smjera od kojeg je šteta primljena. Koristi se rutina `KnockRoutine()` za određivanje vremena trajanja ove mehanike koja je naravno podesiva.

```
public class Knockback : MonoBehaviour
{
    public bool GettingKnockedBack {  get; private set; }

    [SerializeField] private float knockBackTime = 0.2f;

    private Rigidbody2D rb;

    private void Awake()
    {
        rb = GetComponent<Rigidbody2D>();
    }

    public  void GetKnockedBack(Transform damageSource, float
knockBackAmount)
    {
        GettingKnockedBack = true;
        Vector2 difference = (transform.position -
        damageSource.position).normalized * knockBackAmount * rb.mass;
        rb.AddForce(difference, ForceMode2D.Impulse);
        StartCoroutine(KnockRoutine());
    }

    private IEnumerator KnockRoutine()
    {
        yield return new WaitForSeconds(knockBackTime);
        rb.velocity = Vector2.zero;
        GettingKnockedBack = false;
    }
}
```

Četvrta skripta, EnemyPathfinding.cs, koristi Rigidbody2D komponentu kako bi se neprijatelji mogli kretati prema igraču ili nasumičnoj točki na mapi. Ova skripta također prilagođava smjer neprijateljevog kretanja, omogućujući mu da ispravno vizualno reagira na smjer u kojem ide što znači da se samo objekt unutar scene okreće prema lijevo ili desno

zavisno u kojem smjeru se kreće. Također, omogućuje zaustavljanje kretanja kada neprijatelj napada ili se nalazi u blizini igrača.

```
public class EnemyPathfinding : MonoBehaviour
{
    [SerializeField] private float moveSpeed = 2f;
    private Rigidbody2D rb;
    private Vector2 moveDir;
    private Knockback knockback;
    private SpriteRenderer spriteRenderer;

    private void Awake()
    {
        spriteRenderer = GetComponent<SpriteRenderer>();
        knockback = GetComponent<Knockback>();
        rb = GetComponent<Rigidbody2D>();
    }
    private void FixedUpdate()
    {
        if(knockback.GettingKnockedBack)
        {
            return;
        }

        rb.MovePosition(rb.position + moveDir *
        (moveSpeed * Time.fixedDeltaTime));

        if(moveDir.x < 0)
        {
            spriteRenderer.flipX = true;
        }
        else if(moveDir.x > 0)
        {
            spriteRenderer.flipX = false;
        }
    }
    public void MoveTo(Vector2 targetPosition)
    {
        moveDir = targetPosition;
    }
    public void StopMoving()
    {
        moveDir = Vector3.zero;
    }
}
```

Konačno, Shooter.cs je malo kompleksnija skripta koja se koristi za neprijatelje koji napadaju projektilima. Ova skripta upravlja ispaljivanjem projektila u rafalima, prilagođavanjem kuta i brzine projektila prema igraču. Projektili se mogu ispaljivati u serijama s vremenskim odmakom ili slijedom kako bi se stvorio složeniji obrazac napada, a svi parametri podesivi su untar samog Inspector sučelja.

```
public class Shooter : MonoBehaviour, IEnemy
{
    [SerializeField] private GameObject bulletPrefab;
```

```
[SerializeField] private float bulletMoveSpeed;
[SerializeField] private int burstCount;
[SerializeField] private float projectilesPerBurst;
[SerializeField][Range(0, 359)] private float angleSpread;
[SerializeField] private float startingDistance = 0.1f;
[SerializeField] private float timeBetweenBursts;
[SerializeField] private float restTime = 1f;
[SerializeField] private bool stagger;
[SerializeField] private bool oscillate;
private bool isShooting = false;

public void Attack()
{
    if (!isShooting)
    {
        StartCoroutine(ShootRoutine());
    }
}
private IEnumerator ShootRoutine()
{
    isShooting = true;
    float startAngle, currentAngle, angleStep, endAngle;
    float timeBetweenProjectiles = 0f;

    TargetConeOfInfluence(out startAngle, out currentAngle,
    out angleStep, out endAngle);

    if (stagger)
    {
        timeBetweenProjectiles =
        timeBetweenBursts / projectilesPerBurst;
    }

    for (int i = 0; i < burstCount; i++)
    {
        if (!oscillate)
        {
            TargetConeOfInfluence(out startAngle,
            out currentAngle, out angleStep, out endAngle);
        }

        if(oscillate && i % 2 != 1)
        {
            TargetConeOfInfluence(out startAngle,
            out currentAngle, out angleStep, out endAngle);
        }

        else if (oscillate)
        {
            currentAngle = endAngle;
            endAngle = startAngle;
            startAngle = currentAngle;
            angleStep *= -1;
        }

        for (int j = 0; j < projectilesPerBurst; j++)
        {
            Vector2 pos = FindBulletSpawnPos(currentAngle);

            GameObject newBullet = Instantiate(bulletPrefab,
            pos, Quaternion.identity);
```

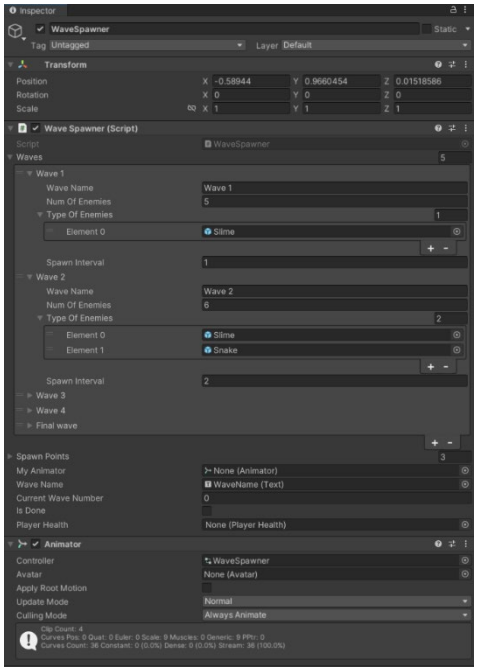
```
                newBullet.transform.right = newBullet.
                transform.position - transform.position;

                if (newBullet.
                    TryGetComponent(out Projectile projectile))
                {
                    projectile.UpdateMoveSpeed(bulletMoveSpeed);
                }

                currentAngle += angleStep;

                if (stagger)
                {
                      yield return new
WaitForSeconds(timeBetweenProjectiles);
                }
            }

            currentAngle = startAngle;

            if (!stagger)
            {
                yield return new WaitForSeconds(timeBetweenBursts);
            }
        }

        yield return new WaitForSeconds(restTime);
        isShooting = false;
    }

    private void TargetConeOfInfluence(out float startAngle,
    out float currentAngle, out float angleStep, out float endAngle)
    {
        Vector2 targetDirection = PlayerController.
        Instance.transform.position - transform.position;
        float targetAngle = Mathf.Atan2(targetDirection.y,
        targetDirection.x) * Mathf.Rad2Deg;
        startAngle = targetAngle;
        endAngle = targetAngle;
        currentAngle = targetAngle;
        float halfAngleSpread = 0f;
        angleStep = 0f;
        if (angleSpread != 0)
        {
            angleStep = angleSpread / (projectilesPerBurst - 1);
            halfAngleSpread = angleSpread / 2f;
            startAngle = targetAngle - halfAngleSpread;
            endAngle = targetAngle + halfAngleSpread;
            currentAngle = startAngle;
        }
    }
    private Vector2 FindBulletSpawnPos(float currentAngle)
    {
        float x = transform.position.x + startingDistance *
                  Mathf.Cos(currentAngle * Mathf.Deg2Rad);
        float y = transform.position.y + startingDistance *
                  Mathf.Sin(currentAngle * Mathf.Deg2Rad);
        Vector2 pos = new Vector2(x, y);

        return pos;
```

```
    }
}
```

Metoda `TargetConeOfInfluence()` unutar koda određuje raspon unutar kojeg neprijatelj ispaljuje projektile. Prvo se izračunava smjer prema igraču koristeći vektorsku razliku pozicija neprijatelja i igrača, a zatim se taj smjer pretvara u kut u stupnjevima pomoću ugrađene metode `Mathf.Atan2()`. Zatim se izračunava raspon kuteva za ispaljivanje, gdje `startAngle` predstavlja početni kut, a `endAngle` završni kut tog raspona, s korakom između projektila određenim parametrom `angleStep`. Ako je `oscillate` uključen, smjer kuteva se obrće pri svakom drugom napadu, stvarajući cik-cak obrazac ispaljivanja projektila.

3.3.5.4. Logika valova

Valovi neprijatelja i logika između njih je jednostavna, a skripta WaveSpawner.cs omogućava i podešavanje svih stavki valova kao što su broj valova i broj neprijatelja u valu unutar samog Inspector sučelja. Ta skripta nalazi se na ne vidljivom objektu WaveSpawner. Također se može podesiti koje neprijatelje želimo u kojem valu jednostavnim postavljanjem prefab asseta biranog neprijatelja u specifični val.



Slika 22: Objekt WaveSpawner

Neprijatelji se generiraju na definiranim točkama koje su u sceni samo prazni i igraču ne vidljivi objekti. Klasa `Wave` definira svaki val neprijatelja, s atributima kao što su ime vala `waveName`, broj neprijatelja u valu `numOfEnemies`, tipovi neprijatelja `typeOfEnemies`, te interval između generiranja neprijatelja `spawnInterval`. Unutar glavne klase `WaveSpawner`, sve ove

vrijednosti postavljaju se putem Unity Inspectora, što omogućava jednostavno prilagođavanje bez potrebe za direktnom izmjenom koda.

```
public class Wave
{
    public string waveName;
    public int numOfEnemies;
    public GameObject[] typeOfEnemies;
    public float spawnInterval;
}

public class WaveSpawner : MonoBehaviour
{
    public Wave[] waves;
    public Transform[] spawnPoints;
    public Animator myAnimator;
    public Text waveName;
    private Wave currentWave;
    public int currentWaveNumber;
    private float nextSpawnTime;

    private bool canSpawn = true;
    private bool canAnimate = false;
    public bool IsDone = false;
    public PlayerHealth playerHealth;

    private void Awake()
    {
        myAnimator = GetComponent<Animator>();
    }

    private void FixedUpdate()
    {
        TellFirstWave();
        currentWave = waves[currentWaveNumber];
        Debug.Log(currentWaveNumber);
        SpawnWave();
        GameObject[] totalEnemies = GameObject.
        FindGameObjectsWithTag("Enemy");
        if (totalEnemies.Length == 0)
        {
            if(currentWaveNumber + 1 != waves.Length)
            {
                if (canAnimate)
                {
                    waveName.text =
  waves[currentWaveNumber + 1].waveName;
                    myAnimator.SetTrigger("WaveComplete");
                    canAnimate = false;
                }
            }
            else
            {
                Debug.Log("GameFinish");
                myAnimator.SetTrigger("GameWon");
                IsDone = true;
                Destroy(playerHealth.gameObject);
            }
        }
```

```
    }

    private void TellFirstWave()
    {
        if(currentWaveNumber + 1 == 1)
        {
            myAnimator.SetBool("IsFirstWave", true);
        }

        else
        {
            myAnimator.SetBool("IsFirstWave", false);
        }
    }

    private void LoadMenu()
    {
        SceneManager.LoadScene("Menu");
    }

    private void SpawnNextWave()
    {
        currentWaveNumber++;
        canSpawn = true;
    }

    private void SpawnWave()
    {
        if(canSpawn && nextSpawnTime <Time.time)
        {
            GameObject randomEnemy = currentWave.
            typeOfEnemies[Random.Range(0, currentWave.
            typeOfEnemies.Length)];
            Transform randomPoint =
            spawnPoints[Random.Range(0, spawnPoints.Length)];
            Instantiate(randomEnemy, randomPoint.position,
            Quaternion.identity);
            currentWave.numOfEnemies--;
            nextSpawnTime = Time.time + currentWave.spawnInterval;
            if(currentWave.numOfEnemies == 0)
            {
                canSpawn = false;
                canAnimate = true;
            }
        }
    }
}
```

U metodi `FixedUpdate()` kontinuirano se prati stanje vala i neprijatelja. Kroz većinu skriptu koristimo `FixedUpdate()` umjesto standardne `Update()` metode kako bi osigurali da kod unutar tih metoda bude precizan i konzistentan na fiksnim vremenskim intervalima, neovisno o varijacijama sličica po sekundi (eng. *framerate*) [8]. Ako su svi neprijatelji u trenutnom valu eliminirani, metoda prelazi na sljedeći val ili pokreće završnu animaciju igre. Metoda `SpawnWave()` generira neprijatelje na nasumičnim točkama unutar određenih intervala, a kada su svi neprijatelji generirani, skripta čeka njihovu eliminaciju prije nego što omogući prelazak na novi val.

4. Zaključak

Razvoj akcijske videoigre pucanja i preživljanja pruža uvid u kompleksnost stvaranja dinamičnog i uzbudljivog igrivog iskustva kroz detaljno planiranje i implementaciju različitih mehanizama igre. Korištenje Unity-a omogućava integraciju različitih funkcionalnosti kao što su upravljanje valovima neprijatelja, stvaranje složenih sustava zdravlja i izdržljivosti, te implementaciju mehanike borbe. Kroz rad na ovom projektu, jasno je kako su različiti elementi igre, poput raznih vrsta neprijatelja, mehanizama za upravljanje igračem i animacija, ključni za stvaranje uravnoteženog i privlačnog iskustva za igrače.

Jedan od značajnih aspekata ovog projekta je i korištenje pixel art stila, koji ne samo da daje igri prepoznatljiv vizualni identitet, već također doprinosi jedinstvenom estetskom doživljaju. Pixel art stil se pokazao kao izvrsan izbor za ovu igru, jer omogućuje stvaranje detaljnih vizualnih prikaza i istovremeno daje retro šarm koji odražava karakter igre. Ovaj stil, u kombinaciji s dinamičnim igranjem i pažljivo dizajniranim mehanikama, doprinosi stvaranju zabavnog i izazovnog iskustva.

Implementacija ključnih komponenti, kao što su sustavi za upravljanje valovima neprijatelja, izgradnja složenih skripti za kontrolu kretanja i borbe, te dizajn korisničkog sučelja, pokazuje koliko je važno imati jasnu viziju i razumijevanje svih aspekata igre. Postizanje visoke kvalitete igre zahtijeva pažljivo planiranje i testiranje, uz stalno usavršavanje svih elemenata kako bi se osigurala besprijekorna funkcionalnost i intuitivno korisničko iskustvo.

Ovaj projekt ističe važnost kombinacije tehničkog znanja, kreativnosti i predanosti u razvoju igara. Kroz rad se pokazuje kako se jednostavan i intuitivan dizajn može uspješno spojiti s tehničkim rješenjima, pružajući igračima nevjerojatno iskustvo. Ovaj projekt bogat je što se tiče same programske logike i koncepata unutar nje bez kojih se ne bi ostvario željeni krajnji rezultat.

Popis literature

[1] Unity Real-Time Development Platform | 3D, 2D, VR & AR Engine. (bez dat.). Pristupano 3. rujna, 2024, sa https://unity.com/

[2] Microsoft Visual Studio: IDE and Code Editor for Software Developers and Teams. (bez dat.). Pristupano 20. svibnja, 2024, sa https://visualstudio.microsoft.com/

[3] Unity Asset Store – The Best Assets for Game Making (bez dat.). Pristupano 3. rujna, 2024, sa https://assetstore.unity.com/

[4] Singleton Pattern – patterns.dev (bez dat.). Pristupano 3. rujna, 2024, sa https://www.patterns.dev/vanilla/singleton-pattern/

[5] Using coroutines in Unity – LogRocket Blog (30. lipanj, 2022).
Pristupano 3. rujna, 2024, sa https://blog.logrocket.com/using-coroutines-unity/

[6] How to create art and gameplay with 2D tilemaps | Unity (bez dat.). Pristupano 3. rujna, 2024, sa https://unity.com/how-to/create-art-and-gameplay-2d-tilemaps-unity

[7] Using Rule Tiles | Unity Learn (bez dat.). Pristupano 3. rujna, 2024, sa https://learn.unity.com/tutorial/using-rule-tiles#

[8] Unity – Scripting API: SceneManager (1. rujan, 2024.). Pristupano 3. rujna, 2024, sa https://docs.unity3d.com/ScriptReference/SceneManagement.SceneManager.html

[9] Definitive guide to animation with Unity 2022 LTS | Unity (13. lipanj, 2024.). Pristupano 3. rujna, 2024, sa https://unity.com/resources/definitive-guide-animation-unity-2022-lts-ebook

[10] Cinemachine Documentation | Package Manager UI Website (18. listopad, 2023.). Pristupano 3. rujna, 2024, sa https://docs.unity3d.com/Packages/com.unity.cinemachine@2.3/manual/index.html

[11] Learn C# For Unity Tutorial – Complete Guide – GameDev Academy. (8. studeni, 2024.). Pristupano 3. rujna, 2024, sa https://gamedevacademy.org/learn-c-for-unity-tutorial-complete-guide/

[12] M. Konecki, „RRI - Sučelje“, nastavni materijal na kolegiju Razvoj računalnih igara [Moodle], Sveučilište u Zagrebu, Fakultet organizacije i informatike, Varaždin, 2023.

Popis slika