Izrada top-down RPG igare

Mijatović, Luka

MEĐIMURSKO VELEUČILIŠTE U ČAKOVCU
STRUČNI PRIJEDIPLOMSKI/DIPLOMSKI STUDIJ RAČUNARSTVO

LUKA MIJATOVIĆ
03113026671

IZRADA TOP-DOWN RPG IGRE

ZAVRŠNI RAD

Čakovec, rujan 2024.

MEĐIMURSKO VELEUČILIŠTE U ČAKOVCU
STRUČNI PRIJEDIPLOMSKI STUDIJ RAČUNARSTVO

LUKA MIJATOVIĆ
03113026671

IZRADA TOP-DOWN RPG IGRE

DEVELOPMENT OF A TOP-DOWN RPG GAME

ZAVRŠNI RAD

Mentor:
Nenad Breslauer, v.pred

Čakovec, rujan 2024.

MEĐIMURSKO VELEUČILIŠTE U ČAKOVCU

PRIJAVA TEME I OBRANE ZAVRŠNOG/DIPLOMSKOG RADA

Stručni prijediplomski studij:

Računarstvo ☒ Održivi razvoj ☐ Menadžment turizma i sporta ☐

Stručni diplomski studij Menadžment turizma i sporta: ☐

Pristupnik: LUKA MIJATOVIĆ, JMBAG: 03113026671
(ime i prezime)

Kolegij: Razvoj računalnih igara
(na kojem se piše rad)

Mentor: Nenad Breslauer, v. pred.
(ime i prezime, zvanje)

Naslov rada: IZRADA TOP-DOWN RPG IGRE

Naslov rada na engleskom jeziku: DEVELOPMENT OF A TOP-DOWN RPG GAME

Članovi povjerenstva:
1. mr.sc. Ivan Hegeduš, v. pred, predsjednik
(ime i prezime, zvanje)
2. Tibor Rodiger, v. pred, član
(ime i prezime, zvanje)
3. Nenad Breslauer, v. pred, mentor
(ime i prezime, zvanje)
4. dr.sc. Sanja Brekalo, zamjenski član
(ime i prezime, zvanje)

Broj zadatka: 2023-RAČ-11

Kratki opis zadatka: Zadatak je napraviti top-down RPG (Role Playing Games) igru.

Izraditi računalnu igru 3D RPG žanra (Role Playing Games) odnosno "igra igranja uloga" jedan je od najpopularnijih te ujedno i najprodavanijih žanrova u industriji računalnih igara.

Korištenjem platforme za izradu računalnih igara. Izraditi scenu te sve potrebne elemente, koristiti mogućnosti koje pruža podsustav za osvjetljenje scene,

animacije te koristiti simulaciju fizikalnih svojstava. Izraditi početnu scenu s izbornikom.

Koristiti platformu za izradu igara, programski jezik te dodatne programske alate. Završni rad mora sadržavati sažetak, sadržaj i uvod, nakon čega slijedi poglavlje u kojem je potrebno navesti i objasniti osnovne ciljeve rada te očekivani rezultat.

U narednom poglavlju potrebno je opisati primijenjene postupke, alate i metode. Poglavlje koje slijedi obrađivati će postignute rezultate nakon čega slijedi poglavlja u kojem se kritički raspravlja o primijenjenim metodama i postupcima te se u narednom

Datum: 18.9.2024 Potpis mentora: Breslauer

SAŽETAK

Ovaj završni rad prikazuje razvoj i implementaciju top-down RPG (eng. *Role-Playing Game*) igre izrađene pomoću Unity Game Engine-a. Cilj rada bio je stvoriti interaktivno iskustvo koje kombinira klasične RPG elemente poput borbe, istraživanja i sakupljanja nagrada, uz korištenje modernih tehnologija i metoda razvoja igara. Kroz ovaj projekt istražene su i primijenjene ključne tehnike i pristupi dizajnu igara, kao i njihova integracija u jedinstveni, kohezivni proizvod.

Igra sadrži nekoliko osnovnih elemenata koji su ključni za top-down RPG žanr. Igrač preuzima ulogu glavnog lika koji se kreće unutar dvodimenzionalnog svijeta, sudjeluje u borbama protiv raznih neprijatelja, istražuje okruženje, prikuplja nagrade i pokušava preživjeti. Za ostvarenje tih elemenata korištene su različite funkcije, algoritmi i skripte. Jedna od glavnih funkcionalnosti je sustav pokreta i animacije likova, koji je implementiran korištenjem Unity-evog Animator-a i C# skripti za kontrolu fizike i kolizija. Borbeni mehanizmi, uključujući sustave napada igrača i neprijatelja, oslanjaju se na detekciju sudara, manipulaciju stanjima likova, dok su neprijatelji opremljeni osnovnom umjetnom inteligencijom (eng. *artificial intelligence*) za micanje.

Skoro svi elementi u igri su modularni, što je postignuto korištenjem "serialized fieldova" u Unity-u. Ovaj pristup omogućava laku prilagodbu i izmjenu karakteristika neprijatelja, predmeta i raznih drugih elemenata igre, bez potrebe za dubokim izmjenama u kodu. Takva modularnost omogućava fleksibilniji razvoj i testiranje novih ideja te olakšava buduće nadogradnje igre. U razvoju igre korištene su tehnologije kao što su Unity za integraciju i dizajn svijeta, Visual Studio kao okruženje za programiranje, GitHub za kontrolu verzija i suradnju, te mnogo različitih online izvora za razne besplatne "sprite-ove" (eng. *sprites*) te njihovu izradu. Unity omogućava korištenje alata poput Tilemap-a za izradu svijeta igre i Sprite Editor-a za upravljanje vizualnim komponentama, dok je C# korišten za skriptiranje ponašanja i logike igre.

Kroz ovaj projekt, dobiveno je duboko razumijevanje procesa razvoja videoigara, od planiranja i dizajna do implementacije i testiranja. Igra predstavlja rezultat uspješne primjene stečenih znanja u području programiranja i dizajna te koristi moderne alate i tehnologije kako bi se postigla željena interaktivnost i angažman igrača.

Ključne riječi: *Unity, RPG, razvoj igara, C# programiranje, animacija, AI, modularnost*

ABSTRACT

This thesis presents the development and implementation of a top-down RPG (Role-Playing Game) created using the Unity Game Engine. The goal of the project was to create an interactive experience that combines classic RPG elements such as combat, exploration, and reward collection while using modern game development technologies and methods. Throughout this project, key techniques and approaches to game design were explored and applied, as well as their integration into a unique, cohesive product.

The game contains several core elements that are crucial for the top-down RPG genre. The player assumes the role of the main character, who moves within a two-dimensional world, engages in battles against various enemies, explores the environment, collects rewards, and tries to survive. To achieve these elements, different functions, algorithms, and scripts were utilized. One of the main functionalities is the movement and character animation system, implemented using Unity's Animator and C# scripts to control physics and collisions. Combat mechanisms, including player and enemy attack systems, rely on collision detection and character state manipulation, while enemies are equipped with basic artificial intelligence for movement.

Almost all elements in the game are modular, achieved through the use of serialized fields in Unity. This approach allows for easy customization and modification of enemy characteristics, items, and various other game elements without requiring deep code changes. Such modularity enables more flexible development and testing of new ideas and facilitates future game upgrades. Technologies such as Unity for world integration and design, Visual Studio as a programming environment, GitHub for version control and collaboration, and various online sources for different free sprites and their creation were used in game development. Unity provides tools like the Tilemap for world creation and the Sprite Editor for managing visual components, while C# was used for scripting game behavior and logic.

Through this project, a deep understanding of the video game development process was gained, from planning and design to implementation and testing. The game represents the successful application of acquired knowledge in programming and design and utilizes modern tools and technologies to achieve the desired interactivity and player engagement.

Keywords: *Unity, RPG, game development, C# programming, animation, AI, modularity*

POPIS KORIŠTENIH KRATICA

2D – (eng. two-dimensional)

3D – (eng. three-dimensional)

.cs – (eng. C# script)

AAA – (eng. triple-A games)

AI – (eng. Artificial Intelligence)

AR – (eng. Augmented Reality)

D&D – (eng. Dungeons & Dragons)

GH – (eng. GitHub)

HP – (eng. Health Points)

I/O – (eng. Input/Output)

IDE – (eng. Integrated Development Environment)

LVL – (eng. Level)

MMORPG – (eng. Massively Multiplayer Online Role-Playing Game)

PIP – (eng. Pixel Import Preset)

RPG – (eng. Role-Playing Game)

SFX – (eng. Sound Effects)

SP – (eng. Stamina Points)

UAS – (eng. Unity Asset Store)

UI – (eng. User Interface)

VFX – (eng. Visual Effects)

VR – (eng. Virtual Reality)

SADRŽAJ

1. UVOD

Razvoj računalnih igara, osobito igara u RPG žanru, značajno je evoluirao od svojih skromnih početaka. RPG igre su tijekom posljednjih desetljeća doživjele izniman rast popularnosti, evoluirajući od jednostavnih tekstualnih avantura u složene, vizualno impresivne interaktivne svjetove koji kombiniraju različite ključne elemente, uključujući borbu, istraživanje i razvoj likova. S razvojem tehnologije, a osobito alata kao što su Unity game engine, Visual Studio i platformi poput GitHub-a, postalo je moguće stvoriti igre visoke kvalitete čak i u malim timovima ili individualnim projektima.

Ovaj završni rad bavi se razvojem i implementacijom (eng. *top-down*) RPG igre koristeći Unity, jedan od najpopularnijih i najmoćnijih alata za razvoj igara. Svrha rada je prikazati cjelokupan proces razvoja igre, od početne ideje i dizajna do konačne implementacije i testiranja. Glavni cilj projekta bio je stvoriti zabavno iskustvo koje kombinira klasične RPG elemente, poput borbe, istraživanja i prikupljanja nagrada, uz korištenje modernih tehnologija i metoda razvoja igara.

Rad se fokusira na nekoliko ključnih aspekata razvoja igre. Prvo, istražuje povijesni razvoj i značaj RPG igara te pruža kontekst i pregled trenutnog stanja i trendova u industriji igara. Zatim, detaljno opisuje korištene tehnologije i alate, poput Unity razvojnog okruženja, Visual Studio razvojnog okruženja i GitHub-a za *verzioniranje* koda. Nadalje, rad se bavi konceptualnim dizajnom igre, uključujući strukturu igre, organizaciju scena, kreiranje grafičkih elemenata i animacija te implementaciju ključnih mehanika igre, kao što su kontrola igrača, borbeni sustav i umjetna inteligencija neprijatelja.

U zaključnom dijelu rada raspravlja se o izazovima s kojima se susretalo tijekom razvoja igre, o naučenim lekcijama te se daje osvrt na potencijalne smjerove budućeg razvoja i nadogradnje igre. Rad također uključuje pregled korištenih resursa, kao što su *sprite-ovi*, i dodatke koji pružaju dublji uvid u tehničke aspekte i kreativni proces iza razvoja igre.

Kroz ovaj završni rad čitatelj će steći razumijevanje cjelokupnog procesa razvoja ove računalne igre, kao i dublji uvid u specifičnosti dizajna, programiranja i implementacije RPG igara u top-down formatu.

2. POVIJEST RPG IGARA

2.1. Počeci i razvoj RPG igara

Povijest **RPG** igara započinje u (eng. *tabletop*) igrama iz 20. stoljeća, poput **D&D** (eng. *Dungeons & Dragons*) objavljene 1974. godine od strane TSR-a. Dungeons & Dragons, kreirana od strane Garyja Gygax-a i Davea Arneson-a, postavila je temelje RPG žanra: igrači preuzimaju uloge fiktivnih likova, istražuju izmišljene svjetove, rješavaju zadatke i bore se protiv neprijatelja koristeći svoju maštu. Ova je igra postavila temelje za osnovne mehanike kao što su napredovanje kroz nivoe, razvoj priče i interakcija s okolinom. Inspiracija za rane RPG igre često je dolazila iz djela *fantasy* književnosti poput **"Gospodara prstenova"** J.R.R. Tolkiena i **"Conana Barbarina"** Roberta E. Howarda, čije su priče pružale bogate svjetove, kompleksne zaplete i epske avanture koje su postale osnovni elementi RPG iskustava.

S razvojem računalne tehnologije krajem 1970-ih i početkom 1980-ih, RPG koncepti su počeli ulaziti u digitalni svijet. Prve digitalne RPG igre pojavile su se kao tekstualne avanture na velikim računalima, poput Colossal Cave Adventure (1976.) i Zork (1977.), koje su se oslanjale na unos tekstualnih komandi za istraživanje svjetova i rješavanje zagonetki. Pojava igara s jednostavnom grafikom, poput Rogue (1980.), stvorila je temelje za pod žanr "roguelike", uvodeći proceduralno generiranje razina i dinamične igre. Igre poput Wizardry (1981.) i Ultima (1981.) dodatno su razvile te ideje, donoseći napredniju grafiku, kompleksnije sustave borbe i bogate priče, omogućujući igračima dublje uranjanje u virtualne svjetove.

Tijekom 1980-ih godina, RPG igre su postajale sve popularnije na osobnim računalima i konzolama. Naslovi poput **The Bard's Tale** (1985.), **Might and Magic** (1986.) i **Final Fantasy** (1987.) pridonijeli su širenju žanra diljem svijeta. Final Fantasy je oblikovao japanski stil RPG igara (JRPG), donoseći likove s detaljnim pričama, strateške borbe i kompleksne narative. U isto vrijeme, igre poput Ultima IV: Quest of the Avatar (1985.) uvele su moralne odluke koje su mogle promijeniti tok igre, dajući iskustvu dodatnu dubinu.

Devedesete godine donijele su tehnološke inovacije koje su omogućile stvaranje kompleksnijih igara s otvorenim svjetovima i naprednim mehanikama. Igre kao što su **The Elder Scrolls: Arena** (1994.) i **Fallout** (1997.) uvele su otvorene svjetove i slobodu istraživanja, dok su **Diablo** (1996.) i **Baldur's Gate** (1998.) redefinirale akcijski RPG žanr kombinirajući brzu akciju s dubokim narativima. Ovo je razdoblje također obilježeno procvatom RPG igara na konzolama, gdje su naslovi poput **Final Fantasy VII** (1997.) i **Chrono**

Trigger (1995.) postali globalni hitovi, uvođenjem inovativnih sustava borbe, vizualnih efekata i naracija koje su postavile standarde za buduće igre.

Početkom 2000-ih, RPG igre su se još više individualizirale, kombinirajući elemente iz različitih žanrova. Naslovi kao što su **Mass Effect** (2007.) **i The Witcher** (2007.) spojili su intenzivnu akciju s dubokim pričama i moralnim dilemama, dok su igre poput **Dark Souls** (2011.) stvorile novi pod žanr „**souls-like**“, usredotočen na izazove i mehaniku učenja kroz pokušaje i pogreške. MMORPG naslovi, poput **World of Warcraft** (2004.), postali su globalni fenomeni, okupljajući milijune igrača u zajedničkim online svjetovima. Danas su RPG igre prisutne na svim platformama, od velikih naslova s visokim budžetima do nezavisnih projekata, a nove tehnologije poput virtualne i proširene stvarnosti nude dodatne mogućnosti za još imerzivnija iskustva.

2.2. Značaj i budućnost Top-Down RPG igara

Top-down RPG igre koriste perspektivu odozgo prema dolje, omogućujući igračima strateški pregled cijelog područja igre. Ova perspektiva nudi jednostavnost navigacije i borbe, ističući taktički element igranja. Jedan od prvih primjera ovog stila je **The Legend of Zelda** iz 1986. godine, koja je uvela akcijski borbeni sustav, zagonetke i otvorene svjetove. Taj naslov postao je uzor za mnoge kasnije igre, poput Secret of Mana (1993.) i Terranigma (1995.), koje su proširile žanr dodavanjem više igrivih likova, složenijih narativa i opcija za više igrača.

Razvoj mobilnih i indie igara donio je novu popularnost top-down RPG-ovima. Naslovi poput "**Hyper Light Drifter**" (2016.) i "**Moonlighter**" (2018.) spajaju klasičan izgled i mehanike ovog podžanra s modernim grafikom i inovativnim pristupima, čineći ga relevantnim i u suvremenoj industriji igara.

S obzirom na tehnološki napredak i sve češće miješanje žanrova, budućnost RPG igara izgleda iznimno perspektivno. Razvoj umjetne inteligencije, proceduralnog generiranja sadržaja i tehnologija poput virtualne (**VR**) i proširene stvarnosti (**AR**) otvaraju nove mogućnosti za kreiranje još složenijih i bogatijih svjetova. RPG igre će nastaviti biti ključni dio gaming industrije, nudeći igračima duboka i emotivna iskustva te beskrajne avanture u maštovitim svjetovima.

3. TEHNOLOGIJE I ALATI

Razvoj 2D Top-Down RPG igre zahtijevao je korištenje više različitih tehnologija i alata kako bi se postigao željeni nivo funkcionalnosti, performansi i izgleda same igre. Ključni alati korišteni u ovom projektu su **Unity Game Engine**, **Visual Studio** kao integrirano razvojno okruženje (eng. *IDE – Integrated development environment*) za pisanje i *debugiranje* koda te GitHub kao platforma za *verzioniranje* i suradnju.

3.1. Unity Game Engine

Unity je jedan od najpopularnijih i najrasprostranjenijih *game engine-a* u industriji videoigara. Razvijen je od strane *Unity Technologies*, Unity omogućava razvoj igara na više platformi, uključujući PC, konzole, mobilne uređaje, web preglednike, kao i AR/VR uređaje. Unity podržava razvoj igara u 2D i 3D formatima te pruža niz alata za dizajn, kodiranje, testiranje i optimizaciju.

Neke od glavnih značajka Unity-a uključuju:

- **Vizualno sučelje i alat za uređivanje:** Unity nudi intuitivno korisničko sučelje s mogućnostima *drag-and-drop* za brzo postavljanje i manipulaciju objekata unutar scene. Alati poput ***Tilemap-a*** omogućavaju dizajnerima izradu složenih svjetova igre korištenjem tile-ova, dok ***Sprite Editor*** omogućava obradu i uređivanje 2D vizualnih elemenata.

- **Kodiranje u C#:** Unity koristi C# kao glavni programski jezik za pisanje skripti i koda koji kontroliraju ponašanje objekata, fiziku, umjetnu inteligenciju i druge aspekte igre. Skripte se jednostavno integriraju s Unity-jevim sučeljem (npr. ***Serialized Field***), omogućavajući izravnu manipulaciju svim komponentama igre.

- **Podrška na više platformi:** Unity podržava razvoj igara za brojne platforme i uređaje. Jednom razvijena igra može se izvesti i prilagoditi za više platformi, što značajno štedi vrijeme i resurse.

- **Mogućnost proširenja:** Unity Asset Store i velika zajednica korisnika omogućuju dodavanje novih alata, resursa, *pluginova* i dodataka koji proširuju osnovnu funkcionalnost Unity-a, prilagođavajući ga specifičnim potrebama svakog projekta.

Unity je odabran kao glavni alat za razvoj ove igre zbog svoje funkcionalnosti, intuitivnog sučelja i mogućnosti brze izrade prototipa i testiranja, što je ključno za razvoj indie i studentskih projekata.

3.2. Visual Studio

Visual Studio je integrirano razvojno okruženje (IDE) koje je razvila tvrtka Microsoft. Namijenjen je za razvoj aplikacija na različitim platformama i podržava više programskih jezika, uključujući C#, koji je glavni jezik za kodiranje unutar Unity-a.

Neke od glavnih značajka Unity-a uključuju:

- **Jedan od najboljih okruženja za pisanje koda:** Visual Studio nudi napredne alate za uređivanje koda, uključujući sintaksno bojanje, automatsko dovršavanje koda (eng. *IntelliSense*), napredne funkcije pretraživanja i zamjene, te *refaktoringa* koda. Ovi alati pomažu programerima da pišu čitljiviji, efikasniji i kod manje podložan greškama.
- **Debugging i testiranje:** Visual Studio dolazi s moćnim alatima za *debugiranje* koji omogućavaju programerima praćenje izvršavanja koda, postavljanje točaka prekida (eng. *breakpoints*), provjeru varijabli i praćenje svih aspekata izvedbe igre. Ovo je ključno za brzo pronalaženje i ispravljanje grešaka tijekom razvoja.
- **Integracija s Unity-jem:** Visual Studio ima ugrađenu podršku za Unity, uključujući integrirane alate za *debugging*, pregledavanje Unity-jeve dokumentacije i jednostavno upravljanje projektima. Ova integracija omogućava programerima brz i učinkovit rad unutar jednog okruženja, smanjujući vrijeme potrebno za prebacivanje između alata.

Visual Studio je izabran kao glavni alat za razvoj i kodiranje zbog svoje fleksibilnosti, naprednih mogućnosti *debugiranja*, i jednostavne integracije s Unity-jem, što značajno poboljšava produktivnost programera.

3.3. GitHub

GitHub je platforma za *verzioniranje* koda i suradnju koja koristi **Git** sustav za kontrolu verzija. GitHub omogućava programerima i timovima da upravljaju promjenama u kodu, surađuju na projektima, prate promjene i povijest koda, te integriraju različite grane razvoja na jednom mjestu.

Neke od glavnih značajka Unity-a uključuju:

- **Kontrola verzija:** GitHub omogućuje praćenje svih promjena u kodu tijekom razvoja igre, čime se osigurava da se sve verzije koda pravilno bilježe i pohranjuju. Ovo omogućava jednostavno vraćanje na prethodne verzije ako dođe do greške ili ako se želi isprobati drugačiji pristup.
- **Suradnja i timski rad:** GitHub pruža platformu za jednostavnu suradnju među članovima tima, omogućujući svakom članu da doprinosi projektu iz vlastitog okruženja. Alati za pregled promjena (eng. *pull requests*) omogućuju transparentnu reviziju koda i osiguravaju da se samo kvalitetne izmjene integriraju u glavni kod.
- **Automatizacija i CI/CD:** GitHub Actions omogućuju automatizaciju različitih zadataka, kao što su testiranje koda, izrada verzija i implementacija, što može značajno ubrzati razvojni proces i smanjiti broj grešaka.
- **Sigurnost i pohrana:** GitHub osigurava sigurnu pohranu koda, sa zaštitom od neovlaštenog pristupa i mogućnostima zaštićenih grana i privatnih repozitorija, što je korisno za zaštitu intelektualnog vlasništva.

GitHub je odabran kao platforma za *verzioniranje* i suradnju zbog svoje popularnosti, pouzdanosti i jednostavnog sučelja, koje omogućuje sigurno upravljanje kodom tijekom svih faza razvoja igre.

3.4. Dodatni alati

Uz sam Unity i druge glavne alate prethodno spomenute, tijekom izrade igre korišteni su i dodatni resursi i programi koji su značajno doprinijeli vizualnom i funkcionalnom aspektu igre. Ovi alati su pružili raznovrsne grafičke elemente, zvukove, animacije i druge resurse koji su ubrzali proces izrade i unaprijedili kvalitetu završene igre.

3.4.1. Unity, Itch.io & Craftpix Asset Store

Unity, Itch.io i Craftpix Asset Store korišteni su kao glavni izvor za preuzimanje različitih resursa potrebnih za razvoj igre. Unity Asset Store nudi širok spektar gotovih resursa poput modela, tekstura, skripti i alata koji su olakšali razvoj specifičnih funkcionalnosti i poboljšali performanse igre. Itch.io Asset Store pruža jedinstvene *assete* od indie umjetnika, uključujući *sprite-ove*, zvukove i glazbu, čime je igri dodana raznolikost i originalnost. Craftpix.net

specijalizirana je web stranica za 2D *assetove*, kao što su likovi, predmeti i pozadine, koji su omogućili brzu izradu i prilagodbu vizualnih komponenti igre uz očuvanje estetike.

3.4.2. Aseprite

Aseprite je alat za stvaranje i uređivanje 2D pikselske grafike i animacija. Koristio se za izradu originalnih *spriteova* potrebnih za igru. Njegove mogućnosti za precizno uređivanje piksela omogućile su kreiranje detaljnih i fluidnih animacija likova i objekata, što je pridonijelo vizualnom identitetu igre i usklađenosti stila.

3.4.3. Pixabay & Soundsnap

Pixabay i Soundsnap su online platforme korištene za preuzimanje zvukova i glazbe koji su dodali dodatnu dimenziju atmosferi igre. Pixabay je pružio besplatne zvučne efekte i pozadinsku glazbu, koji su korišteni za stvaranje imerzivnog zvučnog okruženja. Soundsnap je omogućio pristup širokoj paleti profesionalnih zvukova i audio efekata koji su dodatno unaprijedili zvučni dizajn igre, pružajući kvalitetne audio resurse za razne akcije, efekte i ambijentalne zvukove.

4. IZRADA TOP-DOWN RPG IGRE

4.1. Konceptualni dizajn igre

Ovo je klasična **2D Top-Down RPG igra** koja koristi perspektivu odozgo prema dolje (eng. ***Top-Down***) unutar 2D prostora, sa snažnim naglaskom na elemente RPG igra. Cilj igre je omogućiti igraču uzbudljivo iskustvo borbe i preživljavanja kroz nekoliko složenih i uzbudljivih nivoa, u kojima igrač koristi alate i oružja za suočavanje s raznim izazovima i neprijateljima, te pri tome skuplja novčiće koji se kasnije uračunavaju u konačni rezultat. Da bi smo to postigli, igra koristi razne sustave, skripte i kodove.

Jedan od tih sustava je (eng. ***combat system***) gdje se igrač kroz svaki nivo suočava s različitim vrstama neprijatelja. Kako bi porazio neprijatelje i osiguraj svoje preživljavanje, igrač koristi razne alate i oružja koja su mu dostupna u igri, čineći borbu dinamičnom i izazovnom.

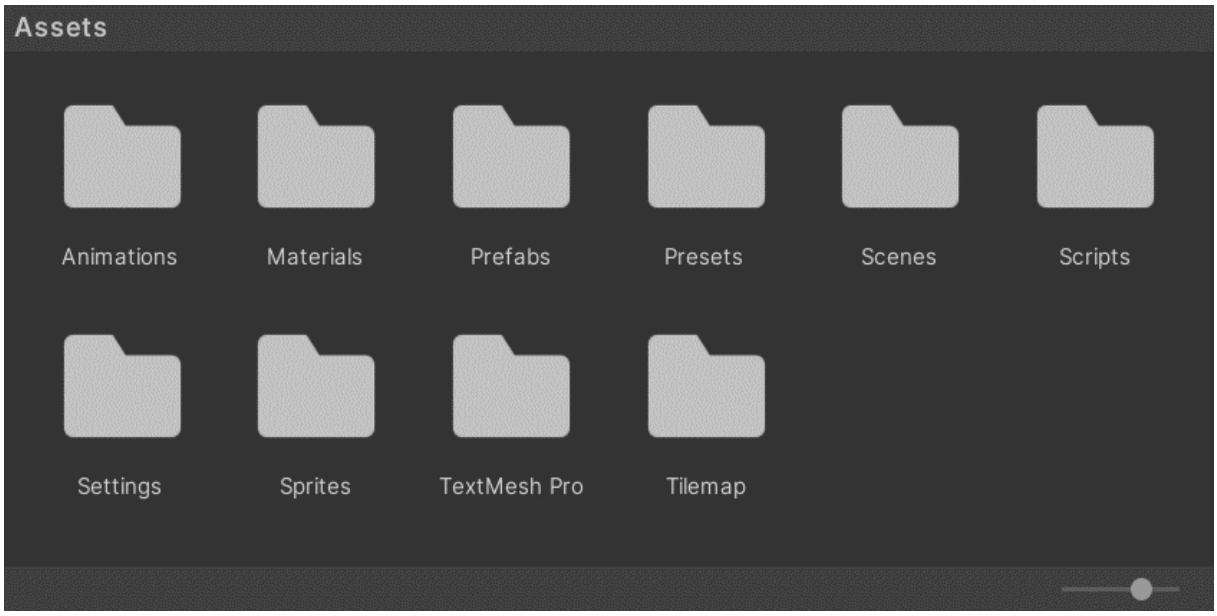
Kroz borbeni sustav i ubijanje neprijatelja, igrač dobiva i sakuplja novčiće, koji se kasnije zbrajaju na konačni rezultat na kraju zadnje razine (na *Game Over* sceni). Novčići predstavljaju nagradu za uspješno savladavanje prepreka i neprijatelja te motiviraju igrača na daljnje istraživanje nivoa, te ubijanje svakog neprijatelja na nivou.

Glavna ideja igre je pružiti igračima zabavno iskustvo u Top-Down svijetu s fokusom na borbu i korištenje oružja i drugih sposobnosti. Kroz razvoj igre, implementirane su razne komponente koje čine osnovu svakog dobrog *RPG-a* – elementi borbe, istraživanja, prikupljanja nagrada, te napredovanja kroz razine. Ove značajke, uz puno drugih funkcionalnosti i sustava koji će biti detaljno opisani u sljedećim odjeljcima, čine igru uzbudljivim iskustvom za igrača.

4.2. Struktura igre i organizacija scena

4.2.1. Struktura igre

Organizacija strukture projekta u Unity-u je ključna za efikasan razvoj igre, održavanje čistoće koda, te za omogućavanje lakšeg pronalaženja i korištenja potrebnih elemenata. U ovom projektu, *Assets* mapa sadrži sve potrebne datoteke, resurse i skripte koji čine igru, a organizirana je u nekoliko ključnih direktorija (**Slika 1.**).



Slika 1. Organizacija Assets foldera
Izvor: Autor

Assets mapa je organizirana u sljedeće poddirektorije:

- **Animations** – Sadrži sve animacijske kontrolere i animacije za različite elemente igre, uključujući likove, oružja i objekte u igri.
- **Materials** – Sadrži materijale i *shadere* korištene za definiranje izgleda površina i specijalnih efekata (eng. *VFX – visual effects*) u igri.
- **Prefabs** – Ova mapa uključuje sve *prefab* objekte, koji su unaprijed definirani predlošci objekata spremi za višekratnu upotrebu u sceni, kao što su sam igrač, neprijatelji, predmeti, oružja, vizualni efekti itd.
- **Presets** – Sadrži predloške, poput postavki za uvoz *Sprite-ova* (eng. *Pixel Import preset*) koje osiguravaju konzistentan izgled i veličinu *sprite-ova*.
- **Scenes** – Direktorij sadrži sve scene igre, uključujući glavni izbornik (*Main Menu*), glavnu scenu (*Town*), razine igre, te *Game Over* scenu.
- **Scripts** – Najbitnija mapa, sadrži sve C# skripte koje definiraju funkciju svega u igri, kao npr. ponašanje likova, neprijatelja, upravljanje scenama, korisničko sučelje (eng. *User Interface*) i razne druge funkcionalnosti igre.

- **Settings** – Sadrži različite postavke, poput *renderiranja*, globalnih postavki igre, i predložaka scena.
- **Sprites** – Ovdje se nalaze svi vizualni elementi igre, uključujući *sprite-ove* likova, objekata, neprijatelja, okruženja i korisničkog sučelja.
- **Tilemap** – Direktorij za sve *tile* resurse korištene u izradi svijeta igre, poput *tilemap sprite-ova* i pravila za automatsko postavljanje *tile-ova*.

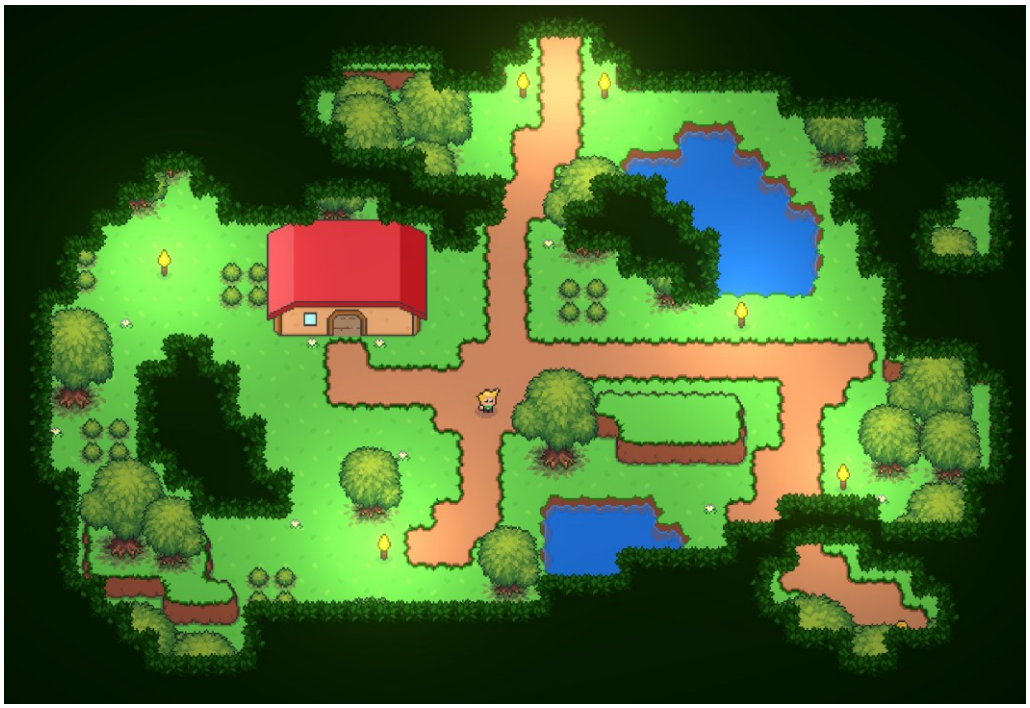
4.2.2. Organizacija scena

Svaka scena unutar igre predstavlja jedinstveno okruženje u kojem se igrač nalazi. U ovom projektu, scene su podijeljene prema svojim funkcijama:

Glavni izbornik (Main Menu) – Prikazuje se prilikom pokretanja igre, omogućava igraču da započne igru ili izađe iz nje.

Slika 2. Main Menu scena
Izvor: Autor

Glavna scena (Town) – ovo je početna lokacija (kuća na početku šume) u kojoj se igrač nalazi na početku igre ili nakon smrti. Ovdje igrač može obnoviti zdravlje (eng. *health*), skupiti neke početne novčiće i uzeti predah prije nastavka dalje.



Slika 3. Town scena
Izvor: Autor

Razine igre – Svaka razina predstavlja različito okruženje sa drugačijim neprijateljima i rasporedom, koje igrač mora savladati kroz igru. Detaljne slike scena razina igre neće biti prikazane u ovom radu, kako bi se izbjegli *spoiler-i* i očuvalo „svježe“ iskustvo igrača.

Slika 4. Primjer razine igre
Izvor: Autor

Game Over scena – Prikazuje se kada igrač završi igru, prikazujući konačni rezultat (od skupljanja novčića), te daje mogućnosti izlaska iz igre.

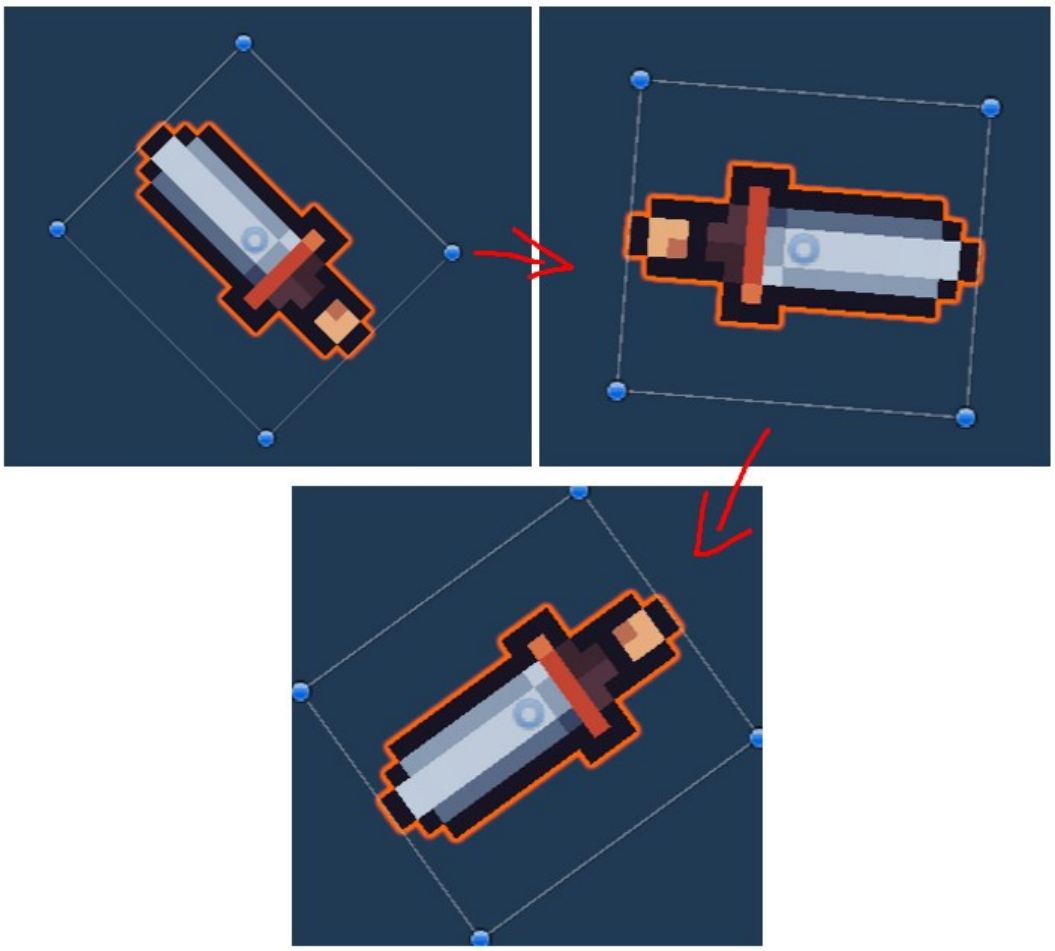
Slika 5. Game Over scena
Izvor: Autor

4.3. Grafički elementi i animacije

U igri su korišteni razni grafički elementi kako bi se stvorilo privlačno i dinamično vizualno iskustvo. To uključuje kreiranje *sprite-ova* za likove, oružja i objekte, te izradu animacija koje oživljavaju te elemente i unose dinamiku u svijet igre.

Kreiranje *sprite-ova*

Sprite-ovi, koji predstavljaju osnovne grafičke elemente igre, korišteni su za prikazivanje likova, neprijatelja, oružja, predmeta i drugih objekata unutar svijeta igre. Izrada *sprite-ova* je obavljena pomoću alata kao što je **Aseprite**, koji omogućuje precizno uređivanje piksel grafike i detaljnu kontrolu nad bojama i oblicima. Na primjer, *sprite-ovi* oružja poput mača prikazani su u različitim položajima i rotacijama kako bi omogućili glatke prijelaze tijekom animacija napada (**Slika 6.**).

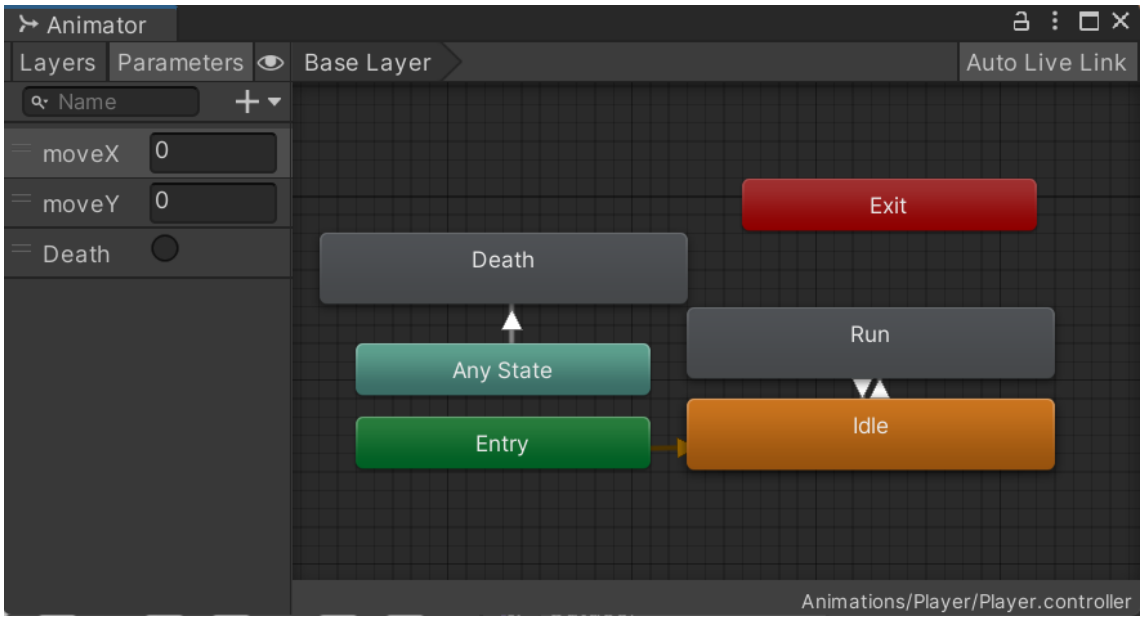


Slika 6. Animacija napada mačem
Izvor: Autor

Većina sprite-ova je preuzeta s **Unity Asset Store-a** i drugih gore spomenutih izvora, poput **Itch.io Asset Store-a** i **Craftpix.net**. Nakon preuzimanja, *sprite-ovi* su prilagođeni specifičnim potrebama igre kroz dodatne izmjene i dorade. Time se osigurava jedinstven izgled koji se uklapa u cjelokupni vizualni stil igre.

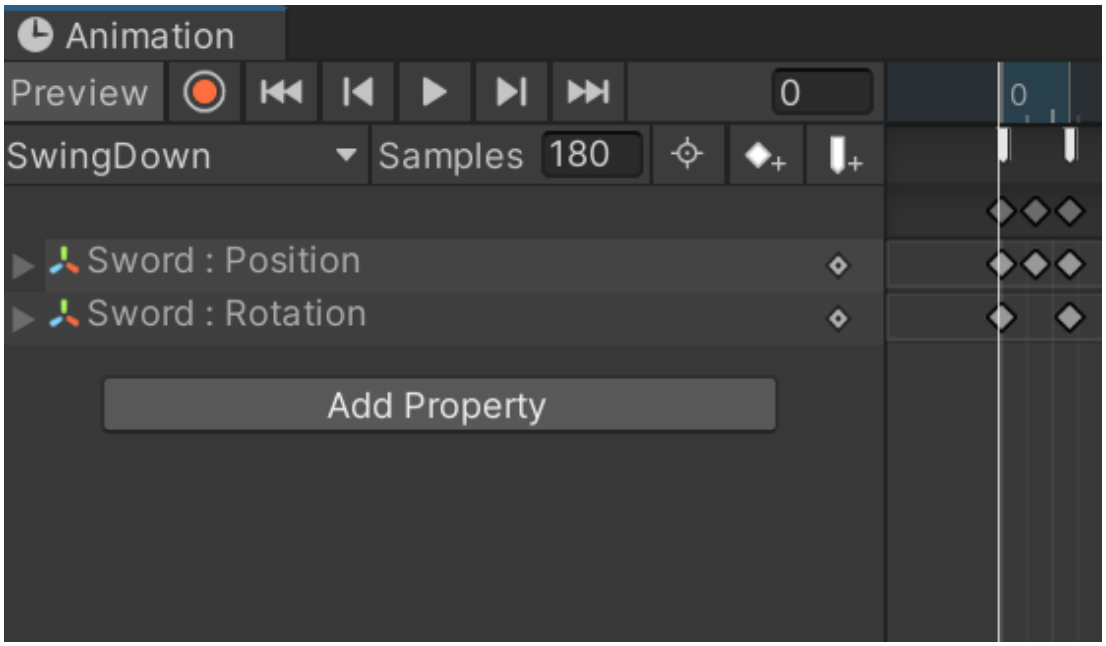
Izrada animacija

U igri je korišten Unity-jev **Animator** za izradu i upravljanje animacijama likova i objekata. Animator omogućuje stvaranje složenih animacijskih grafova koji definiraju prijelaze između različitih stanja, poput *"Idle"*, *"Run"*, *"Attack"*, ili *"Death"*. Na **Slici 7.** ispod, prikazan je primjer Animator prozora za glavnog lika, gdje su definirani prijelazi između stanja mirovanja (eng. *Idle*), trčanja (eng. *Run*) i smrti (eng. *Death*). Animator koristi parametre, kao što su "*moveX*", "*moveY*" i "*Death*", za određivanje prijelaza između tih stanja, što omogućuje fluidno kretanje i reakcije lika na igračeve unose.



Slika 7. Unity Animator prozor
Izvor: Autor

Uz Animator, korišten je i Unity-jev **Animation** alat za kreiranje (eng. ***ANIM keyframes***) za specifične akcije, poput animacije mača tijekom zamaha. Na **Slici 8.** ispod, prikazana je animacija zamaha mačem (eng. *SwingDown*), gdje su pozicija i rotacija mača kontrolirane putem ključnih sličica. Ovo omogućava glatku i realističnu animaciju napada, pridonoseći osjećaju udarca i dinamike tijekom borbe.



Slika 8. Unity Animation prozor
Izvor: Autor

4.4. Mehanike igre i sustavi

U ovom odjeljku detaljno ćemo objasniti ključne mehanike igre i sustave koji određuju način na koji se igra odvija, kao i samo iskustvo igrača. Fokusirat ćemo se na osnovne elemente igre, što uključuje kontrolu kretanja i interakciju igrača s svijetom, borbeni sustav, sustav *Inventory-a* i prikupljanje predmeta, *AI* neprijatelja, te sustave upravljanja scenama i korisničkim sučeljem. Ovi sustavi zajedno čine temelj igre i utječu na dinamiku i izazovnost igre, pružajući igračima zabavno i zadovoljavajuće iskustvo.

4.4.1. Kontrola igrača i kretanje

Kontrola igrača u igri je ključni element koji omogućava fluidno kretanje, borbu i interakciju sa svijetom igre. Sustav kontrole igrača sastoji se od nekoliko komponenti, uključujući skripte za upravljanje kretanjem, zdravljem, izdržljivošću i uništenjem objekata nakon njihove upotrebe. Ovi sustavi osiguravaju responzivnost i realizam.

Kontrola kretanja i akcija igrača

Sustav kontrole igrača implementiran je kroz nekoliko skripti (**Slika 9.**), među kojim su ključne **Player Controls** (Player Controls.inputactions i Player Controls.cs), **PlayerController, PlayerHealth, Stamina** i **SelfDestroy**.

Slika 9. Skripte upravljanja Player-a
Izvor: Autor

PlayerController.cs

Skripta PlayerController upravlja svim načinima kretanja igrača i interakcije s okolinom. Korištenjem Unity-jevog (eng. *Input System-a*), skripta obrađuje (eng. *input-e*) igrača (kao što su kretanje i napad) te ih pretvara u odgovarajuće akcije unutar igre. Skripta koristi komponente kao što su **Rigidbody2D** za fiziku kretanja, **Animator** za upravljanje animacijama lika, i **SpriteRenderer** za promjenu smjera u kojem lik gleda, ovisno o poziciji miša.

Kretanje igrača se kontrolira pomoću **moveSpeed** varijable koja definira brzinu kretanja, dok funkcija **Move()** upravlja stvarnim pomicanjem pomoću fizikalnog sustava Unity-ja. Ova funkcija uzima u obzir smjer kretanja koji igrač definira putem tipkovnice te ažurira poziciju lika u stvarnom vremenu.

```
1 reference
private void PlayerInput()
{
    movement = playerControls.Movement.Move.ReadValue<Vector2>();

    myAnimator.SetFloat("moveX", movement.x);
    myAnimator.SetFloat("moveY", movement.y);
}

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

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

Slika 10. Metode Move() i PlayerInput()
Izvor: Autor

Sustav za (eng. ***Dash***) (brzi pokret) implementiran je metodom **Dash()**, koja koristi dodatnu brzinu kretanja (eng. *dashSpeed*) za brza izbjegavanja neprijatelja i njihovih napada. Skripta također koristi **TrailRenderer** komponentu kako bi prikazivala vizualni efekt *dash-a* i dodatno ga naglasila.

```
private void Dash()
{
    // Provjera da li je igrač već u dash-u i ima li dosta dash-a za potrošiti
    if(!isDashing && Stamina.Instance.CurrentStamina > 0)
    {
        Stamina.Instance.UseStamina(); // Potroši jedan dash
        isDashing = true; // Stavlja igrača u stanje dash-a
        moveSpeed *= dashSpeed; // Naglo povećava brzinu kretanja
        myTrailRenderer.emitting = true; // Aktivira vizualni efekt dash-a
        // Reproducira zvuk dash-a
        SFXManager.instance.PlaySFXClip(dashSoundClip, transform, SoundVolume);
        StartCoroutine(EndDashRoutine()); // Pokreće korutinu završetka dash-a
    }
}
```

Slika 11. Metoda Dash()
Izvor: Autor

Skripta **AdjustPlayerFacingDirection()** koristi poziciju miša kako bi odredila smjer u kojem lik treba gledati.

```
private void AdjustPlayerFacingDirection()
{
    Vector3 mousePos = Input.mousePosition;
    // Dobiva položaj igrača na ekranu
    Vector3 playerScreenPoint = Camera.main.WorldToScreenPoint(transform.position);

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

Slika 12. Metoda AdjustPlayerFacingDirection()
Izvor: Autor

PlayerHealth.cs

Skripta PlayerHealth prati stanje (eng. *health*-a) igrača, upravlja procesima (eng. *damage*-a) i smrti, te omogućava sustav (eng. *heal*-anja). Igrač može izgubiti health kada stupi u kontakt s neprijateljima, što se detektira pomoću (eng. *collider*-a). Skripta također kontrolira vizualne i zvučne efekte povezane sa stanjem zdravlja igrača, kao što su efekti tresanja ekrana, te zvukovi

udarca i smrti. Funkcija **TakeDamage()** smanjuje trenutni *health* igrača kada primi *damage*, pokreće animaciju „treptanja“ likova i efekte zvuka, te ispituje da li je igrač došao do nula *health-a*.

```
public void TakeDamage(int damageAmount, Transform hitTransform)
{
    if (!canTakeDamage) { return; }

    // Efekti prilikom primanja damage-a
    ScreenShakeManager.Instance.ShakeScreen();
    knockback.GetKnockedBack(hitTransform, knockBackThrustAmount);
    StartCoroutine(flash.FlashRoutine());
    SFXManager.instance.PlaySFXClip(HitSoundClip, transform, HitSoundVolume);

    canTakeDamage = false;
    currentHealth -= damageAmount;
    StartCoroutine(DamageRecoveryRoutine());
    UpdateHealthSlider();
    CheckIfPlayerDeath();
}
```

Slika 13. Metoda TakeDamage()
Izvor: Autor

U slučaju smrti, funkcija **CheckIfPlayerDeath()** pokreće animaciju smrti i proceduru za *resetiranje* igre, uključujući ponovno učitavanje Town scene.

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

Slika 14. Metoda CheckIfPlayerDeath()
Izvor: Autor

Stamina.cs:

Skripta Stamina upravlja igračevom (eng. *Staminom*), koja se troši prilikom korištenje posebne akcije zvana (eng. *Dash*). Ako igrač ima dovoljno „charge-ova“ *dash-a*, može ga izvesti koristeći tipke *Space* na tipkovnici i funkciju **UseStamina()**, koja smanjuje trenutni iznos *stamine* i pokreće funkciju obnove *stamine* nakon određenog vremena. Skripta također upravlja vizualnim prikazom *stamine*, koristeći UI elemente slika i (eng. *slider-a*).

```
public void UseStamina()
{
    CurrentStamina--;
    UpdateStaminaImages();
    StopAllCoroutines();
    StartCoroutine(RefreshStaminaRoutine());
}

2 references
public void RefreshStamina()
{
    if(CurrentStamina < maxStamina && !PlayerHealth.Instance.IsDead)
    {
        CurrentStamina++;
    }
    UpdateStaminaImages();
}
```

Slika 15. Metode UseStamina i RefreshStamina
Izvor: Autor

SelfDestroy.cs:

Skripta SelfDestroy koristi se za uništavanje objekata koji više nisu potrebni, kao što su vizualni efekti ili (eng. *particles*) nakon završetka neke animacije. Ova skripta periodično provjerava stanje *particle sistema* i uništava objekt kada animacija završi, čime se optimizira performanse igre.

```
private void Update()
{
    if (ps && !ps.IsAlive())
    {
        DestroySelfAnimEvent();
    }
}

1 reference
public void DestroySelfAnimEvent()
{
    Destroy(gameObject);
}
```

Slika 16. SelfDestroy skripta
Izvor: Autor

Detekcija sudara

Detekcija sudara (eng. *collision detection*) implementirana je pomoću Unity-jevih komponenti kao što su **Rigidbody2D** i različite vrste **Collider2D-a** (kao npr. BoxCollider2D, CircleCollider2D, itd.) ovisno o objektu. Na primjer, igrači i neprijatelji koriste **Rigidbody2D** za simuliranje fizike kretanja, dok različiti *collider-i* definiraju granice sudara između likova i objekata. (Slika 17. – zeleni *collider*)

Slika 17. Rigidbody2D i CapsuleCollider2D igrača
Izvor: Autor

OnCollisionStay2D() metoda iz skripte **PlayerHealth.cs** koristi se za kontinuiranu provjeru sudara između igrača i neprijatelja. Kada se otkrije sudar s neprijateljem, aktivira se funkcija **TakeDamage()**, koja smanjuje *health* igrača i aktivira odgovarajuće efekte.

```
private void OnCollisionStay2D(Collision2D other)
{
    EnemyAI enemy = other.gameObject.GetComponent<EnemyAI>();

    if(enemy)
    {
        TakeDamage(1, other.transform);
    }
}
public void TakeDamage(int damageAmount, Transform hitTransform)
{
    if (!canTakeDamage) { return; }

    // Efekti prilikom primanja damage-a
    ScreenShakeManager.Instance.ShakeScreen();
    knockback.GetKnockedBack(hitTransform, knockBackThrustAmount);
    StartCoroutine(flash.FlashRoutine());
    SFXManager.instance.PlaySFXClip(HitSoundClip, transform, HitSoundVolume);

    canTakeDamage = false;
    currentHealth -= damageAmount;
    StartCoroutine(DamageRecoveryRoutine());
    UpdateHealthSlider();
    CheckIfPlayerDeath();
}
```

Slika 18. Metode OnCollisionStay2D() i TakeDamage()
Izvor: Autor

Rigidbody2D komponenta upravlja fizikalnim interakcijama i kretanjem, dok *collider-i* osiguravaju da objekti ispravno detektiraju sudare i pravilno reagiraju na njih, čime se stvara realistična fizikalna simulacija u igri.

Ovaj sustav kontrole kretanja i akcija igrača sa sustavima za *health* i *staminu*, omogućavaju visoku razinu interakcije i realizma u igri, pružajući igraču osjećaj fluidnosti i zabave tijekom igranja.

4.4.2. Dizajn svijeta

Dizajn svijeta igre sastoji se od različitih objekata. Ovi objekti uključuju različite vrste (eng. *pickup*) predmeta, sustav ekonomije s novčićima, *generatore* predmeta, kao i razne (eng. *destructible* i *indestructible*) objekte. Svaki od ovih elemenata pridonosi raznolikosti igre i motivira igrač na istraživanje i interakciju sa svijetom igre.

Pickup.cs

Skripta Pickup upravlja različitim vrstama predmeta koje igrač može pokupiti u igri, kao što su novčići, kugle *stamine* i kugle *health-a*. Skripta koristi **OnTriggerEnter2D()** metodu za detekciju kolizije s igračem i poziva **DetectPickupType()** kako bi utvrdila vrstu predmeta koji je igrač pokupio.

```
Unity Message | 0 references
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.GetComponent<PlayerController>())
    {
        DetectPickupType();
        SFXManager.instance.PlaySFXClip(SoundClip, transform, SoundVolume);
        Destroy(gameObject);
    }
}
1 reference
private void DetectPickupType()
{
    switch(pickUpType)
    {
        case PickUpType.GoldCoin:
            EconomyManager.Instance.UpdateCurrentGold();
            break;
        case PickUpType.HealthGlobe:
            PlayerHealth.Instance.HealPlayer();
            break;
        case PickUpType.StaminaGlobe:
            Stamina.Instance.RefreshStamina();
            break;
        default:
            break;
    }
}
```

Slika 19. Metode OnTriggerEnter2D() i DetectPickupType()
Izvor: Autor

EconomyManager.cs:

Skripta EconomyManager prati količinu novčića koje igrač prikupi tijekom igre. Metoda **UpdateCurrentGold()** ažurira količinu novčića i prikazuje je na korisničkom sučelju. Ovaj sustav daje igraču osjećaj napretka i nagrade, motivirajući ga da skuplja predmete kroz razine igre.

```
1 reference
public void UpdateCurrentGold()
{
    currentGold += 1;

    if(goldText == null)
    {
        goldText = GameObject.Find(COIN_AMOUNT_TEXT).GetComponent<TMP_Text>();
    }

    goldText.text = currentGold.ToString("D3");
}
```

Slika 20. Metoda UpdateCurrentGold()
Izvor: Autor

PickUpSpawner.cs:

Skripta PickUpSpawner upravlja generiranjem predmeta koje igrač može pokupiti nakon što određeni neprijatelji ili objekti budu uništeni. Metoda **DropItems()** koristi logiku šanse za odlučivanje o vrsti i količini predmeta koji će se generirati. Na primjer, šanse da igrač dobije novčić su 50%, unutar kojih postoji 70% šanse da dobije jedan novčić, 25% šanse da dobije dva novčića i 5% šanse da dobije tri novčića. Osim novčića, postoji 15% šanse da igrač dobije kuglicu zdravlja (eng. *health globe*), 15% šanse za kuglicu izdržljivosti (eng. *stamina globe*), te 20% šanse da ne dobije ništa.

```
public void DropItems()
{
    // Generira nasumični broj od 1 do 100
    int randomNum = Random.Range(1, 101);

    // 50% šanse za novčiće
    if (randomNum <= 50)
    {
        // Zagarantiran barem 1 novčić
        int coinsToDrop = 1;
        int additionalCoins = Random.Range(1, 101); // Generaraj random broj za dodatne novčiće

        if (additionalCoins <= 25) // 25% za 2
        {
            coinsToDrop = 2;
        }
        else if (additionalCoins <= 30) // 5% za 3
        {
            coinsToDrop = 3;
        }

        for (int i = 0; i < coinsToDrop; i++)
        {
            Instantiate(goldCoin, transform.position, Quaternion.identity);
        }
    }
    // 15% za health
    else if (randomNum > 50 && randomNum <= 65)
    {
        Instantiate(healthGlobe, transform.position, Quaternion.identity);
    }
    // 15% za staminu
    else if (randomNum > 65 && randomNum <= 80)
    {
        Instantiate(staminaGlobe, transform.position, Quaternion.identity);
    }
    // 20% za ništa
}
```

Slika 21. Metoda DropItems()
Izvor: Autor

Destructible.cs i Indestructible.cs:

Skripta Destructible omogućava objektima u igri da budu uništeni, dok skripta Indestructible koristi objekte koji su trajni i ne mogu se uništiti. Kada igrač uništi *destructible* objekt, skripta OnTriggerEnter2D() pokreće efekte poput vizualnih efekata i zvukova te koristi **PickUpSpawner** za generiranje predmeta.

```
Unity Message | 0 references
private void OnTriggerEnter2D(Collider2D other)
{
    if(other.gameObject.GetComponent<DamageSource>() || other.gameObject.GetComponent<Projectile>())
    {
        PickUpSpawner pickUpSpawner = GetComponent<PickUpSpawner>();
        pickUpSpawner?.DropItems();
        Instantiate(destroyVFX, transform.position, Quaternion.identity);
        SFXManager.instance.PlaySFXClip(SoundClip, transform, SoundVolume);
        Destroy(gameObject);
    }
}
```

Slika 22. Metoda za Destructible.cs
Izvor: Autor

Vizualni efekti i animacije

RandomIdleAnimation.cs:

Skripta RandomIdleAnimation koristi se za animacije objekata ili neprijatelja u stanju mirovanja. Skripta nasumično reproducira animaciju unutar animatora kako bi se spriječila monotonija u igri.

```
private void Start()
{
    if (!myAnimator) { return; }

    AnimatorStateInfo state = myAnimator.GetCurrentAnimatorStateInfo(0);
    myAnimator.Play(state.fullPathHash, -1, Random.Range(0f, 1f));
}
```

Slika 23. Metoda za RandomIdleAnimation.cs
Izvor: Autor

SpriteFade.cs

Skripta SpriteFade koristi se za postepeno smanjivanje prozirnosti objekata ili *sprite-ova*, stvarajući efekt nestajanja. Ovaj efekt se koristi za stvaranje dramatičnih trenutaka ili prijelaza unutar igre.

```
public IEnumerator SlowFadeRoutine()
{
    float elapsedTime = 0;
    float startValue = spriteRenderer.color.a;

    while (elapsedTime < fadeTime)
    {
        elapsedTime += Time.deltaTime;
        float newAlpha = Mathf.Lerp(startValue, 0f, elapsedTime / fadeTime);
        spriteRenderer.color = new Color(spriteRenderer.color.r, spriteRenderer.color.g, spriteRenderer.color.b, newAlpha);
        yield return null;
    }
    Destroy(gameObject);
}
```

Slika 24. Metoda SlowFadeRoutine()
Izvor: Autor

TransparencyDetection.cs:

Skripta TransparencyDetection koristi se za detekciju prolaska igrača kroz objekte i prilagodbu njihove prozirnosti, što omogućava bolju preglednost prilikom kretanja kroz složena okruženja.

```
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.GetComponent<PlayerController>())
    {
        if (spriteRenderer)
        {
            StartCoroutine(FadeRoutine(spriteRenderer, fadeTime, spriteRenderer.color.a, transparencyAmount));
        }
        else if (tilemap)
        {
            StartCoroutine(FadeRoutine(tilemap, fadeTime, tilemap.color.a, transparencyAmount));
        }
    }
}
```

Slika 25. Metoda za TransparencyDetection.cs
Izvor: Autor

Parallax.cs:

Skripta Parallax implementira *parallax* efekt koji stvara iluziju dubine u 2D svijetu igre, pomicanjem pozadinskih slojeva brzinom koja je različita od brzine prednjih slojeva.

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

Unity Message | 0 references
private void Start()
{
    startPos = transform.position;
}

Unity Message | 0 references
private void FixedUpdate()
{
    transform.position = startPos + travel * parallaxOffset;
}
```

Slika 26. Metode za Parallax.cs
Izvor: Autor

4.4.3. Upravljanje scenama i UI

Upravljanje scenama i **korisničkim sučeljem** (UI) ključni su elementi u dizajnu igre koji osiguravaju glatke prijelaze između različitih dijelova igre, učinkovito upravljanje kamerom, te konzistentno iskustvo za igrača. Ovaj odjeljak opisuje funkcionalnosti vezane za upravljanje scenama, elementima korisničkog sučelja i izbornicima, te korištenje *Singleton* obrasca za globalno upravljanje podacima i objektima unutar igre.

Upravljanje scenama

Upravljanje scenama u igri uključuje prijelaze između različitih lokacija, kontrolu nad kamerom i prikazivanje poruka za igrače. Ključne skripte koje omogućuju ove funkcionalnosti su **AreaEntrance**, **AreaExit**, **CameraController** i **SceneManagement**.

AreaEntrance.cs:

Skripta **AreaEntrance.cs** određuje ulaznu točku igrača prilikom učitavanja nove scene. Ako ime prijelaza odgovara prethodno postavljenom imenu prijelaza u **SceneManagement**, igrač se *teleportira* na definiranu ulaznu poziciju, a kamera se automatski postavlja na praćenje igrača.

```
[SerializeField] private string transitionName;

Unity Message | 0 references
private void Start()
{
    if(transitionName == SceneManagement.Instance.SceneTransitionName)
    {
        PlayerController.Instance.transform.position = this.transform.position;
        CameraController.Instance.SetPlayerCameraFollow();

        UIFade.Instance.FadeToClear();
    }
}
```

Slika 27. Metoda za AreaEntrance.cs
Izvor: Autor

AreaExit.cs:

Skripta AreaExit definira izlazne točke između scena. Kada igrač stigne do izlazne točke, provjerava se jesu li svi neprijatelji poraženi prije nego što se omogući prijelaz na sljedeću scenu. Ako su svi neprijatelji poraženi, scena se mijenja uz efekt zatamnjenja ekrana.

```
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.GetComponent<PlayerController>())
    {
        if (AreAllEnemiesDefeated())
        {
            SceneManagement.Instance.SetTransitionName(sceneTransitionName);
            UIFade.Instance.FadeToBlack();
            StartCoroutine(LoadSceneRoutine());
        }
        else
        {
            // Pozovi metodu da pokaže da nisu ubijeni svi neprijatelji
            SceneManagement.Instance.DisplayDefeatEnemiesMessage();
        }
    }
}
```

Slika 28. Metoda za AreaExit.cs
Izvor: Autor

CameraController.cs:

Skripta CameraController koristi **CinemachineVirtualCamera** za praćenje igrača tijekom igre. Metoda **SetPlayerCameraFollow()** osigurava da kamera uvijek slijedi igrača, stvarajući osjećaj kontinuiteta tijekom igranja.

```
private CinemachineVirtualCamera cinemachineVirtualCamera;

Unity Message | 0 references
private void Start()
{
    SetPlayerCameraFollow();
}

3 references
public void SetPlayerCameraFollow()
{
    cinemachineVirtualCamera = FindObjectOfType<CinemachineVirtualCamera>();
    cinemachineVirtualCamera.Follow = PlayerController.Instance.transform;
}
```

Slika 29. Metoda SetPlayerCameraFollow()
Izvor: Autor

SceneManagement.cs:

Skripta SceneManagement upravlja prijelazima između scena, postavlja nazive prijelaza i prikazuje poruke igračima, kao što je zahtjev za porazom svih neprijatelja prije napuštanja scene.

```
1 reference
public void DisplayDefeatEnemiesMessage()
{
    StartCoroutine(DisplayDefeatEnemiesMessageCoroutine());
}

1 reference
private IEnumerator DisplayDefeatEnemiesMessageCoroutine()
{
    if (defeatEnemiesText != null)
    {
        defeatEnemiesText.SetActive(true);

        yield return new WaitForSeconds(messageDisplayDuration);

        if (defeatEnemiesText != null)
        {
            defeatEnemiesText.SetActive(false);
        }
    }
}
```

Slika 30. Metode za SceneManagement.cs
Izvor: Autor

Izbornici i korisničko sučelje (UI)

MainMenu.cs:

Skripta MainMenu kontrolira glavni izbornik igre, omogućujući igraču da započne igru, učita scenu ili napusti igru.

```
0 references
public void PlayGame()
{
    SceneManager.LoadSceneAsync(1);

    SceneManager.sceneLoaded += OnSceneLoaded;
}

0 references
public void QuitGame()
{
    Application.Quit();
}
```

Slika 31. Metode za MainMenu.cs
Izvor: Autor

MusicManager.cs:

Skripta MusicManager dinamički upravlja glazbom igre ovisno o sceni. Na primjer, kada igrač uđe u određenu scenu (poput borbe s glavnim *bossom* ili Game Over scene), glazba se automatski mijenja.

```
private void OnSceneLoaded(Scene scene, LoadSceneMode mode)
{
    if (scene.name == "Scene_5")
    {
        ChangeMusic(bossMusic);
    }
    else if (scene.name == "Game Over")
    {
        ChangeMusic(gameOverMusic);
    }
    else
    {
        ChangeMusic(bgmMusic);
    }
}
```

Slika 32. Metoda za MusicManager.cs
Izvor: Autor

PauseMenu.cs:

Skripta PauseMenu omogućuje pauziranje igre i pristup izborniku pauze. Igrač može nastaviti igru, pristupiti glavnom izborniku ili zatvoriti igru.

```
public void Update()
{
    if (Input.GetKeyDown(KeyCode.Escape))
    {
        if (isPaused)
        {
            ContinueGame();
        }
        else
        {
            PauseGame();
        }
    }
}
```

Slika 33. Metoda za PauseMenu.cs
Izvor: Autor

SFXManager.cs

Skripta SFXManager upravlja svim zvučnim efektima unutar igre. Koristeći metodu **PlaySFXClip()**, može reproducirati različite zvučne efekte u određenim trenucima igre.

```
public void PlaySFXClip(AudioClip audioClip, Transform spawnTransform, float volume)
{
    AudioSource audioSource = Instantiate(sfxObject, spawnTransform.position, Quaternion.identity);

    audioSource.clip = audioClip;
    audioSource.volume = volume;
    audioSource.Play();
    float clipLength = audioSource.clip.length;

    Destroy(audioSource.gameObject, clipLength);
}
```

Slika 34. Metoda PlaySFXClip()
Izvor: Autor

UIFade.cs:

Skripta UIFade upravlja prijelazima između scena koristeći efekte zatamnjenja ili posvjetljenja ekrana.

```
1 reference
public void FadeToBlack()
{
    if(fadeRoutine != null)
    {
        StopCoroutine(fadeRoutine);
    }

    fadeRoutine = FadeRoutine(1);
    StartCoroutine(fadeRoutine);
}

1 reference
public void FadeToClear()
{
    if (fadeRoutine != null)
    {
        StopCoroutine(fadeRoutine);
    }

    fadeRoutine = FadeRoutine(0);
    StartCoroutine(fadeRoutine);
}
```

Slika 35. Metode za UIFade.cs
Izvor: Autor

Singleton obrasci

Korištenje **BaseSingleton** i **Singleton** skripti omogućava globalno upravljanje objektima i podacima unutar igre. *Singleton* obrazac osigurava da postoji samo jedna instanca objekta kroz cijelu igru, što omogućava jednostavan pristup i dijeljenje podataka između scena. Na primjer, **SceneManagement.cs** koristi *Singleton* obrazac kako bi upravljao scenama i prijelazima globalno.

```
public class Singleton<T> : MonoBehaviour where T: Singleton<T>
{
    private static T instance;
    56 references
    public static T Instance { get { return instance; } }

    Unity Message | 12 references
    protected virtual void Awake()
    {
        if(instance != null & this.gameObject != null)
        {
            Destroy(this.gameObject);
        } else {
            instance = (T)this;
        }

        if(!gameObject.transform.parent)
        {
            DontDestroyOnLoad(gameObject);
        }
    }
}
```

Slika 36. Metode za Singleton.cs
Izvor: Autor

Ovi elementi upravljanja scenama, korisničkog sučelja i globalnog upravljanja podacima kroz *Singleton* obrasce osiguravaju glatko iskustvo igranja, omogućavajući igraču da se fokusira na uživanje u igri bez ometanja tehničkim detaljima.

4.5. Borba i načini napadanja

Borba je ključni element (eng. *gameplay-a*) u ovoj igri, a sustav borbe uključuje različita oružja, napade i vizualne efekte koji poboljšavaju iskustvo igrača. Implementacija borbe uključuje upravljanje oružjem, efektima napada te vizualnim i zvučnim efektima kako bi se stvorio dinamičan i izazovan osjećaj borbe. Ovaj odjeljak objašnjava dizajn sustava oružja, načina napada, te implementaciju efekata borbe koji unapređuju vizualni i taktički doživljaj.

4.5.1. Sustav oružja i napada

Sustav oružja i napada implementiran je kroz nekoliko skripti koje omogućuju različite vrste oružja i specifične napade. Svako oružje ima jedinstvene karakteristike i koristi različite napade, omogućavajući igraču da prilagodi svoj stil igre prema preferencijama i situacijama.

ActiveWeapon.cs:

Skripta ActiveWeapon služi kao centralni menadžer za aktivno oružje koje igrač trenutno koristi. Korištenjem ove skripte, igra omogućuje jednostavno upravljanje raznim oružjima (poput mača, luka i strijele ili magičnog štapa) te omogućava igraču da brzo mijenja oružja i napada.

Funkcionalnosti ove skripte uključuju:

- Upravljanje trenutnim aktivnim oružjem putem **CurrentActiveWeapon** varijable, koja može biti bilo koje oružje koje implementira sučelje **IWeapon**.
- Upravljanje napadima kroz **Attack()** metodu koja detektira kada je pritisnut gumb za napad, te izvršava napad trenutno aktivnog oružja:

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

Slika 37. Metoda Attack()
Izvor: Autor

- Postavljanje vremena između napada putem metode AttackCooldown(), što omogućava različitim oružjima da imaju različite brzine napada, dodajući taktičku raznolikost borbi:

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

Slika 38. Metoda TimeBetweenAttacksRoutine()
Izvor: Autor

Sword.cs:

Skripta Sword implementira ponašanje mača kao oružja za bliske borbe. Mač omogućuje igraču da izvede brze napade u blizini neprijatelja, uz korištenje različitih animacija napada.

- **Attack()** metoda pokreće animaciju napada i aktivira kolizijski okvir mača, čime se omogućava detekcija sudara s neprijateljima:

```
2 references
public void Attack()
{
    SFXManager.instance.PlaySFXClip(damageSoundClip, transform, SoundVolume);
    myAnimator.SetTrigger("Attack");
    weaponCollider.gameObject.SetActive(true);

    slashAnim = Instantiate(slashAnimPrefab, slashAnimSpawnPoint.position, Quaternion.identity);
    slashAnim.transform.parent = this.transform.parent;
}
```

Slika 39. Metoda Attack() za Sword.cs
Izvor: Autor

- Skripta također koristi metode **SwingUpFlipAnimEvent()** i **SwingDownFlipAnimEvent()** za prilagodbu rotacije i smjera animacija ovisno o poziciji miša, čime se dodaje vizualna preciznost prilikom ciljanju napada.

```
public void SwingUpFlipAnimEvent()
{
    slashAnim.gameObject.transform.rotation = Quaternion.Euler(-180, 0, 0);

    if(PlayerController.Instance.FacingLeft)
    {
        slashAnim.GetComponent<SpriteRenderer>().flipX = true;
    }
}
```

Slika 40. Metoda SwingUpFlipAnimEvent()
Izvor: Autor

Bow.cs:

Skripta Bow upravlja ponašanjem luka, oružja za napade na daljinu. Luk omogućava igraču da puca strelice prema neprijateljima iz daljine, čime se igrač može držati na sigurnoj udaljenosti tijekom borbe.

- **Attack()** metoda pokreće animaciju pucanja strelice i *instancira* strelicu kao projektil, definirajući njezin raspon na temelju atributa luka:

```
public void Attack()
{
    myAnimator.SetTrigger(FIRE_HASH);
    SFXManager.instance.PlaySFXClip(damageSoundClip, transform, SoundVolume);
    GameObject newArrow = Instantiate(arrowPrefab, arrowSpawnPoint.position,
        ActiveWeapon.Instance.transform.rotation);
    newArrow.GetComponent<Projectile>().UpdateProjectileRange(weaponInfo.weaponRange);
}
```

Slika 41. Metoda Attack() za Bow.cs
Izvor: Autor

- Strelice su definirane kao projektili koji mogu putovati određenom udaljenosti prije nego što nestanu ili pogode metu, čime se dodaje dinamičnost i realizam u borbi.

Staff.cs:

Skripta Staff implementira magični štap, oružje koje omogućava dalekometne napade u obliku magičnih zraka. Magični štap je dizajniran za igrače koji preferiraju napade iz daljine, ali žele snažnije i efektivnije oružje.

- **Attack()** metoda pokreće animaciju napada, dok metoda **SpawnStaffProjectileAnimEvent()** *instancira* magičnu zraku, koja raste i napada neprijatelje na putu:

```
public void Attack()
{
    myAnimator.SetTrigger(ATTACK_HASH);
}

0 references
public void SpawnStaffProjectileAnimEvent()
{
    GameObject newLaser = Instantiate(magicLaser,
        magicLaserSpawnPoint.position, Quaternion.identity);
    newLaser.GetComponent<MagicLaser>().UpdateLaserRange(weaponInfo.weaponRange);
}
```

Slika 42. Metoda Attack() za Staff.cs
Izvor: Autor

- Magična zraka može pogoditi više neprijatelja istovremeno i ima prilagodljiv raspon, ovisno o snazi i sposobnostima štapa.

MagicLaser.cs i DamageSource.cs:

Skripta MagicLaser upravlja ponašanjem magične zrake, uključujući rast dužine zrake i kolizije s neprijateljima ili objektima, dok skripta DamageSource upravlja mehanizmom nanošenja štete neprijateljima kada dođe do kolizije s oružjem ili projektilom igrača.

Efekti borbe

Efekti borbe su ključni za stvaranje dojmljivog vizualnog i taktilnog iskustva. Oni poboljšavaju osjećaj udarca, pokreta i interakcije s neprijateljima tijekom borbe.

Flash.cs:

Skripta Flash koristi se za stvaranje efekta bljeskanja prilikom primanja štete, signalizirajući igraču da je neprijatelj pogođen.

```
public IEnumerator FlashRoutine()
{
    spriteRenderer.material = whiteFlashMat;
    yield return new WaitForSeconds(restoreDefaultMatTime);
    spriteRenderer.material = defaultMat;
}
```

Slika 43. Metoda FlashRoutine()
Izvor: Autor

Knockback.cs:

Skripta Knockback upravlja efektom odbacivanja, gdje neprijatelji ili igrač bivaju odbijeni unatrag nakon udarca. Ovo dodaje realizam i dinamičnost u borbene interakcije.

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

Slika 44. Metoda GetKnockedBack()
Izvor: Autor

ScreenShakeManager.cs i MouseFollow.cs:

Skripta ScreenShakeManager koristi se za stvaranje efekta podrhtavanja ekrana kada se dogodi značajan događaj poput jakog udarca ili eksplozije. Ovaj efekt doprinosi intenzitetu borbe, dok skripta MouseFollow omogućava da oružje ili napadi prate miša igrača, dajući osjećaj kontrole i preciznosti prilikom ciljanja.

Ovi sustavi i efekti zajedno stvaraju dinamičan i vizualno uzbudljiv borbeni sustav, koji omogućava igraču raznolike stilove igre, taktičke pristupe i zadovoljavajuće osjećaje tijekom borbe.

```
public class ScreenShakeManager : Singleton<ScreenShakeManager>
{
    private CinemachineImpulseSource source;


    protected override void Awake()
    {
        base.Awake();
        source = GetComponent<CinemachineImpulseSource>();
    }


    public void ShakeScreen()
    {
        source.GenerateImpulse();
    }
}
```

Slika 45. ScreenShakeManager.cs skripta
Izvor: Autor

```
public class MouseFollow : MonoBehaviour
{
    Unity Message | 0 references
    private void Update()
    {
        FaceMouse();
    }

    1 reference
    private void FaceMouse()
    {
        Vector3 mousePosition = Input.mousePosition;
        mousePosition = Camera.main.ScreenToWorldPoint(mousePosition);

        Vector2 direction = transform.position - mousePosition;

        transform.right = -direction;
    }
}
```

Slika 46. MouseFollow.cs skripta
Izvor: Autor

4.5.2. Neprijatelji

U igri je implementiran raznolik sustav neprijatelja s različitim ponašanjima i napadima kako bi se igraču pružilo izazovno iskustvo borbe. Neprijatelji se ponašaju prema unaprijed definiranim pravilima i koriste različite strategije napada kako bi igrača držali na oprezu. Ovaj odjeljak opisuje skripte koje upravljaju umjetnom inteligencijom (AI) neprijatelja, njihovim ponašanjem te specifičnim napadima.

AI i ponašanje neprijatelja

EnemyAI.cs:

Skripta EnemyAI upravlja ponašanjem neprijatelja, definirajući različite faze i stanja kretanja. Neprijatelji se mogu nalaziti u stanju lutanja (eng. ***roaming***) ili napada (eng. ***attacking***). Stanje lutanja omogućava neprijatelju da se nasumično kreće unutar određene zone, dok stanje napada aktivira napad na igrača kada je unutar dometa.

- U stanju **Roaming**, neprijatelj se nasumično kreće unutar definirane zone dok se ne nađe unutar dometa napada, nakon čega prelazi u stanje napada:

```
private void Roaming()
{
    timeRoaming += Time.deltaTime;

    if (PlayerController.Instance != null)
    {
        if (PlayerFollow && Vector2.Distance(transform.position,
            PlayerController.Instance.transform.position) < PlayerDistance)
        {
            enemyPathfinding.FollowPlayer(PlayerController.Instance.transform.position);
        }
        else
        {
            enemyPathfinding.MoveTo(roamPosition);

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

            if (timeRoaming > roamChangeDirFloat)
            {
                roamPosition = GetRoamingPosition();
            }
        }
    }
    else
    {
        enemyPathfinding.MoveTo(roamPosition);
    }
}
```

Slika 47. Metoda Roaming()
Izvor: Autor

- U stanju **Attacking**, neprijatelj prestaje lutati i napada igrača. Ako je igrač izvan dometa, neprijatelj se vraća u stanje lutanja:

```
private void Attacking()
{
    if (PlayerController.Instance != null)
    {
        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());
        }
    }
    else
    {
        state = State.Roaming;
    }
}
```

Slika 48. Metoda Attacking()
Izvor: Autor

EnemyPathfinding.cs:

Skripta EnemyPathfinding upravlja kretanjem neprijatelja, omogućavajući im da prate igrača ili se kreću prema ciljanoj točki. Koristeći **Rigidbody2D** komponentu, skripta pokreće neprijatelje u smjeru definiranog vektora. Ova metoda omogućava dinamično praćenje igrača, što doprinosi izazovnosti i dinamičnosti borbe:

```
3 references
public void MoveTo(Vector2 targetPosition)
{
    moveDir = targetPosition;
}

1 reference
public void FollowPlayer(Vector2 playerPosition)
{
    moveDir = (playerPosition - rb.position).normalized;
}
```

Slika 49. Metode za EnemyPathfinding.cs
Izvor: Autor

EnemyHealth.cs:

Skripta EnemyHealth upravlja zdravljem neprijatelja. Kada neprijatelj primi štetu, provjerava se njegova trenutna razina zdravlja, primjenjuje efekt bljeskanja pomoću skripte **Flash**, te se izvršava efekt odbacivanja koristeći **Knockback** komponentu:

```
1 reference
public void TakeDamage(int damage)
{
    currentHealth -= damage;
    SFXManager.instance.PlaySFXClip(SoundClip, transform, SoundVolume);
    knockback.GetKnockedBack(PlayerController.Instance.transform, knockBackThrust);
    StartCoroutine(flash.FlashRoutine());
    StartCoroutine(CheckDetectDeathRoutine());
}
```

Slika 50. Metoda TakeDamage()
Izvor: Autor

Kada zdravlje padne na nulu, neprijatelj umire, aktiviraju se vizualni efekti, a predmeti koje neprijatelj može ispustiti generiraju se koristeći skriptu **PickUpSpawner**.

Shooter.cs:

Skripta Shooter je jedna od najbitnijih skripta jer definira ponašanje neprijatelja koji napadaju iz daljine ispaljujući projektile. Neprijatelj puca u serijama (eng. *bursts*), s različitim parametrima poput brzine metaka, broja ispaljenih projektila i kutova širenja. Skripta koristi **IEnumerator** *korutine* kako bi upravljala ispaljivanjem projektila:

```
2 references
public void Attack()
{
    if (!isShooting)
    {
        StartCoroutine(ShootRoutine());
    }
}
```

Slika 51. Metoda Attack() za Shooter.cs
Izvor: Autor

Ključna metoda u ovoj skripti je **ShootRoutine**, koja upravlja cijelim procesom ispaljivanja projektila kroz *korutinu*. Ova metoda omogućava neprijatelju da puca u serijama, s definiranim brojem projektila, kutevima ispaljivanja, te vremenskim razmakom između serija i projektila. Evo detaljnijeg pregleda kako **ShootRoutine** metoda funkcionira:

1. Inicijalizacija parametara: Metoda započinje s postavljanjem početnih parametara koji definiraju način ispaljivanja projektila:

- **startAngle**, **currentAngle**, **angleStep**, i **endAngle** definiraju kutove ispaljivanja projektila u odnosu na položaj igrača.
- **timeBetweenProjectiles** definira vrijeme između ispaljivanja svakog projektila unutar jedne serije. Ako je omogućena opcija (eng. *stagger*), tada se između svakog ispaljivanja dodaje mala pauza.

```
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; }
```

Slika 52. Inicijalizacija parametara metode ShootRoutine()
Izvor: Autor

2. Ispaljivanje projektila u serijama (bursts): Glavni dio metode sastoji se od dva for petlje. Prva petlja upravlja brojem serija projektila koje neprijatelj ispaljuje, dok druga petlja upravlja ispaljivanjem pojedinačnih projektila unutar svake serije.

```
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))...

        currentAngle += angleStep;

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

Slika 53. Dvije for petlje za ispaljivanje
Izvor: Autor

- Unutar svake serije, metoda određuje smjer ispaljivanja projektila prema poziciji igrača. Ako je opcija (eng. *oscillate*) omogućena, smjer ispaljivanja mijenja se između serija, stvarajući "zig-zag" efekt.

- Za svaki projektil unutar serije, metoda *instancira* metak na temelju trenutnog kuta (**currentAngle**). Ovaj kut se povećava nakon svakog ispaljivanja kako bi se projektili raspršili.

- Ako je opcija (eng. *stagger*) omogućena, postoji pauza između ispaljivanja svakog projektila unutar serije, a ako je omogućena opcija *oscillate*, smjer ispaljivanja projektila mijenja se između serija, stvarajući nepravilni uzorak koji je teže predvidjeti i izbjeći.

3. Pauza između i završetak burst-ova: Nakon svake serije ispaljivanja, metoda pauzira ispaljivanje na određeno vrijeme definirano parametrom **timeBetweenBursts**. Ovo omogućava igraču da reagira na napad i pruži priliku za napad. Nakon što neprijatelj završi sve serije ispaljivanja, metoda čeka određeno vrijeme (**restTime**) prije nego što započne novi ciklus napada. To omogućava balansiranu dinamiku borbe, gdje neprijatelji napadaju u intervalima, dajući igraču priliku za strategiju.

```
            currentAngle = startAngle;

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

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

Slika 54. Pauza i završetak burst-ova
Izvor: Autor

4. Korištenje metode TargetConeOfInfluence: Metoda **TargetConeOfInfluence** izračunava kutove između kojih će projektile biti ispaljeni. Uzima u obzir položaj igrača i širenje kutova (eng. *angle spread*) kako bi se definirala preciznost i širina (eng. *cone*-a) ispaljivanja:

```
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 = 0;
    if (angleSpread != 0)
    {
        angleStep = angleSpread / (projectilesPerBurst - 1);
        halfAngleSpread = angleSpread / 2f;
        startAngle = targetAngle - halfAngleSpread;
        endAngle = targetAngle + halfAngleSpread;
        currentAngle = startAngle;
    }
}
```

Slika 55. Metoda TargetConeOfInfluence()
Izvor: Autor

Metoda **ShootRoutine** u skripti **Shooter.cs** omogućava neprijatelju da napada igrača složenim načinima ispaljivanja projektila. Korištenjem opcija kao što su "burst", "stagger", i "oscillate", metoda omogućava različite strategije napada, koje pridonose raznolikosti i dinamičnosti borbenih situacija u igri. Time se igraču pruža izazovno iskustvo, koje zahtijeva brze reflekse i strateško razmišljanje kako bi izbjegao napade i pobijedio neprijatelje.

Specifični neprijateljski napadi

Neki neprijatelji koriste specifične metode napada koje im omogućavaju jedinstveno ponašanje i strategije u borbi.

Grape.cs:

Skripta Grape definira ponašanje neprijatelja koji napadaju igrača bacanjem projektila. Napad se aktivira kada je igrač unutar dometa, a projektili se ispaljuju prema igraču:

```
2 references
public void Attack()
{
    myAnimator.SetTrigger(ATTACK_HASH);

    if(transform.position.x - PlayerController.Instance.transform.position.x < 0)
    {
        spriteRenderer.flipX = false;
    }
    else
    {
        spriteRenderer.flipX = true;
    }
}

0 references
public void SpawnProjectileAnimEvent()
{
    Instantiate(grapeProjectilePrefab, transform.position, Quaternion.identity);
}
```

Slika 56. Metode za Grape.cs
Izvor: Autor

GrapeProjectile.cs:

Skripta GrapeProjectile upravlja ponašanjem projektila koji neprijatelj ispaljuje. Projektil se kreće prema igraču koristeći animiranu krivulju kako bi dodao realizam letu projektila:

```
private IEnumerator ProjectileCurveRoutine(Vector3 startPosition, Vector3 endPosition)
{
    float timePassed = 0f;

    while (timePassed < duration)
    {
        timePassed += Time.deltaTime;
        float linearT = timePassed / duration;
        float heightT = animCurve.Evaluate(linearT);
        float height = Mathf.Lerp(0f, heightY, heightT);

        transform.position = Vector2.Lerp(startPosition, endPosition, linearT) + new Vector2(0f, height);

        yield return null;
    }

    Instantiate(splatterPrefab, transform.position, Quaternion.identity);
    Destroy(gameObject);
}
```

Slika 57. Metoda ProjectileCurveRoutine()
Izvor: Autor

Ova metoda dodaje element nepredvidivosti i izazovnosti igraču, jer se projektil kreće po paraboličnoj putanji.

GrapeLandSplatter.cs:

Skripta GrapeLandSplatter upravlja efektom udara projektila kada projektil pogodi tlo. Efekt stvara vizualne tragove i može nanijeti štetu igraču ako se nalazi unutar područja udara:

```
Unity Message | 0 references
private void Start()
{
    StartCoroutine(spriteFade.SlowFadeRoutine());

    Invoke("DisableCollider", 0.2f);
}

Unity Message | 0 references
private void OnTriggerEnter2D(Collider2D other)
{
    PlayerHealth playerHealth = other.gameObject.GetComponent<PlayerHealth>();
    playerHealth?.TakeDamage(1, transform);
}
```

Slika 58. Metoda za GrapeLandSplatter.cs
Izvor: Autor

Ove skripte zajedno omogućuju različite neprijateljske taktike i ponašanja, čineći igru dinamičnom i izazovnom za igrača. Neprijatelji koriste različite napade i sposobnosti kako bi prilagodili svoje ponašanje situacijama na terenu, što zahtijeva od igrača da prilagodi svoje strategije i tehnike borbe.

4.6. Inventory

Upravljanje (eng. *inventory*-em) igrača ključni je element *gameplay-a* u ovoj igri, omogućavajući igračima da prikupljaju, mijenjaju i koriste različita oružja kako bi se prilagodili različitim borbenim situacijama. Ovaj sustav upravlja raznim predmetima u *inventory-u*, od kojih svaki može utjecati na *gameplay*, pružajući igraču mogućnost izbora i strategije. U ovom odjeljku opisane su glavne skripte koje omogućavaju funkcioniranje sustava *inventory-a*: **ActiveInventory**, **InventorySlot**, i **IWeapon**.

ActiveInventory.cs:

Skripta ActiveInventory glavni je upravljač *inventory-a* igrača. Ova skripta omogućava igraču da mijenja trenutno aktivni (eng. *slot*) *inventory-a* i oružje koje je opremljeno. Implementacija inventara je dizajnirana tako da igrač može brzo mijenjati oružje ili predmete putem tipkovnice, čineći borbu i interakciju s okolinom fleksibilnijom i dinamičnijom.

- Na početku igre, skripta povezuje unos s tipkovnice putem klase **PlayerControls**, što omogućava igraču da mijenja aktivni *slot* u *inventory-u* pritiskom na tipke.
- Kada igrač pritisne tipku koja odgovara određenom *slotu*, metoda **ToggleActiveSlot** poziva funkciju **ToggleActiveHighlight** koja aktivira vizualni pokazatelj za taj *slot* i mijenja trenutno aktivno oružje:

```
private void ToggleActiveSlot(string keyName)
{
    if (int.TryParse(keyName, out int numValue))
    {
        ToggleActiveHighlight(numValue - 1);
    }
    else
    {
        Debug.LogWarning("Invalid key pressed: " + keyName);
    }
}
```

Slika 59. Metoda ToggleActiveSlot()
Izvor: Autor

- Metoda **ChangeActiveWeapon()** prvo provjerava je li igrač mrtav, a zatim uništava trenutno aktivno oružje, ako postoji. Zatim *instancira* novo oružje iz odabranog *slota* u inventaru. Ova funkcionalnost omogućava igraču da brzo mijenja oružja tijekom borbe:

```
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();

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

    GameObject weaponToSpawn = weaponInfo.weaponPrefab;

    GameObject newWeapon = Instantiate(weaponToSpawn,
        ActiveWeapon.Instance.transform.position, Quaternion.identity);
    ActiveWeapon.Instance.transform.rotation = Quaternion.Euler(0, 0, 0);
    newWeapon.transform.parent = ActiveWeapon.Instance.transform;

    ActiveWeapon.Instance.NewWeapon(newWeapon.GetComponent<MonoBehaviour>());
}
```

Slika 60. Metoda ChangeActiveWeapon()
Izvor: Autor

InventorySlot.cs:

Skripta InventorySlot predstavlja svaki pojedinačni *slot* u inventaru igrača. Svaki *slot* može sadržavati jedan predmet ili oružje, definirano pomoću WeaponInfo objekta. Ovaj pristup omogućava modularnost i fleksibilnost u rukovanju inventarom. Skripta pruža metodu za dohvaćanje informacija o oružju koje se nalazi u određenom *slotu*. Ta metoda se koristi u **ActiveInventory.cs** skripti kako bi se dobile informacije o oružju koje igrač trenutno želi koristiti.

IWeapon.cs:

Sučelje IWeapon osigurava da sva oružja u igri implementiraju osnovne funkcionalnosti potrebne za interakciju s *inventory-em* i sustavom borbe. Definira dvije metode:

- **Attack()**: Metoda koja se poziva za izvršavanje napada specifičnog za svako oružje.
- **GetWeaponInfo()**: Metoda koja vraća informacije o oružju, poput štete, raspona, brzine napada itd.

```
interface IWeapon
{
    4 references
    public void Attack();
    5 references
    public WeaponInfo GetWeaponInfo();
}
```

Slika 61. IWeapon interface
Izvor: Autor

Ovo sučelje omogućava lako dodavanje novih vrsta oružja u igru, budući da svako novo oružje treba implementirati ove metode kako bi pravilno funkcioniralo unutar sustava inventara i borbe.

Sustav inventara omogućava igračima da upravljaju oružjima u igri, što dodaje sloj dubine i strategije u *gameplay*. Kombinacijom **ActiveInventory**, **InventorySlot**, i **IWeapon** skripti, igra omogućuje igračima da se brzo prilagode različitim borbenim situacijama, koristeći različita oružja na temelju vlastitih preferencija i trenutnih potreba. Sustav je dizajniran tako da bude fleksibilan i proširiv, čime se omogućava lakša nadogradnja i proširenje igre u budućnosti.

4.7. Tilemap-e i dizajn nivoa

Kreiranje nivoa u ovoj igri oslanja se na upotrebu **Tilemap** alata u Unity-u, koji omogućava dizajnerima da brzo i jednostavno kreiraju 2D svjetove pomoću (eng. *tile*-ova). Ovaj pristup omogućava stvaranje detaljnih i vizualno privlačnih okruženja koja su modularna i lako prilagodljiva.

Tilemap sustav u Unity-u pruža alate za jednostavno kreiranje i uređivanje *tile-ova* koji čine svjetove igre. *Tile-ovi* se mogu rasporediti na ***grid*** kako bi se stvorile različite vrste terena, poput staza, voda, planina ili vegetacije. Upotreba **Tilemap** alata omogućava brze izmjene i prilagodbe dizajna nivoa, što je ključno za razvoj igara s mnogo nivoa ili dinamičkih okruženja.

Kako bi se olakšao proces kreiranja mapa i *level-a*, koristili smo **Rule Tile** alata, što su posebni *tile-ovi* koje se automatski mijenjaju ovisno o kontekstu u kojem se nalaze. Na primjer, *Rule Tile* može automatski odabrati odgovarajući *tile* za rubove ili unutrašnje dijelove terena, čime se značajno smanjuje vrijeme potrebno za kreiranje mapa. Ovaj pristup omogućava dizajnerima da postignu složene i vizualno kohezivne rezultate uz minimalan napor:

Slika 62. Rule Tile-ovi
Izvor: Autor

Prednosti korištenja Tilemap-a:

- **Fleksibilnost dizajna:** Dizajneri mogu lako mijenjati dizajn nivoa pomoću vizualnog alata bez potrebe za promjenom skripti ili ručnog uređivanja svakog dijela karte.
- **Optimizacija performansi:** *Tilemap* koristi optimizirane metode za *renderiranje*, što omogućava igre s velikim i složenim svjetovima bez gubitka performansi.
- **Modularnost i ponovno korištenje:** *Tile-ovi* se mogu ponovno koristiti u različitim nivoima, smanjujući potrebu za stvaranjem jedinstvenih resursa za svaki dio igre.

Ovaj pristup omogućava brz i učinkovit proces razvoja, olakšavajući iteraciju dizajna i prilagodbu mapa prema potrebama igre.

4.8. Modularnost i fleksibilnost igre

Modularni dizajn je ključni princip u razvoju ove igre, omogućavajući jednostavne prilagodbe i proširenja bez potrebe za opsežnim promjenama koda. Korištenje ***SerializedField*** polja i modularnih skripti omogućava dizajnerima i programerima da lako izmijene elemente igre ili dodaju nove funkcionalnosti.

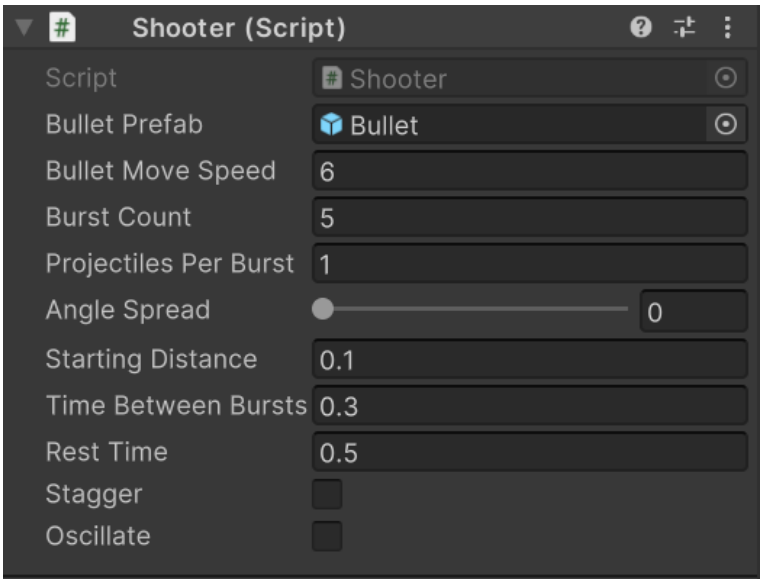
Jedan od ključnih načina na koji se postigla modularnost u igri je korištenje *SerializedField* polja u Unity-u. Ova polja omogućavaju programerima da izlažu privatne varijable u Unity Editoru, gdje ih dizajneri mogu lako prilagoditi bez mijenjanja koda. Na primjer, skripta **Shooter.cs** koristi *SerializedField* polja za postavljanje parametara kao što su brzina metaka, broj metaka u seriji, i kut širenja:

```
[SerializeField] private GameObject bulletPrefab;
[SerializeField] private float bulletMoveSpeed;
[SerializeField] private int burstCount;
[SerializeField] private float projectilesPerBurst;
[SerializeField][Range(0, 359)] private float angleSpread;
[Tooltip("Distance from the shooter where the bullets will spawn.")]
[SerializeField] private float startingDistance = 0.1f;
[Tooltip("Time in seconds between consecutive bursts.")]
[SerializeField] private float timeBetweenBursts;
```

Slika 63. Primjer SerializeField polja u Shooter.cs
Izvor: Autor

Ovaj pristup omogućava dizajnerima da brzo testiraju različite postavke i balanse u igri, čineći igru fleksibilnijom i lakšom za iteraciju.

Skripte su napisane na način da se mogu lako prilagoditi i ponovno koristiti u različitim kontekstima. Na primjer, skripta **Shooter.cs** može se koristiti za različite vrste neprijatelja koji imaju slične napadačke mehanike, ali različite parametre poput brzine napada ili kuta širenja. Također, skripta **EnemyAI.cs** koristi modularne metode za različita stanja neprijatelja (lutanje, napadanje), što omogućava stvaranje neprijatelja s različitim obrascima ponašanja bez potrebe za pisanjem potpuno novih skripti.



Slika 64. Shooter.cs SerializeField-ovi prilagodljivost
Izvor: Autor

Modularni pristup omogućava lako proširivanje igre dodavanjem novih elemenata bez opsežnog mijenjanja postojeće strukture. Na primjer, novi tip oružja može se dodati jednostavnim stvaranjem nove skripte koja implementira sučelje **IWeapon**.

Svako novo oružje samo treba implementirati metode **Attack** i **GetWeaponInfo** kako bi se integriralo u sustav igre, čineći proces dodavanja novih elemenata jednostavnim i brzom.

Prednosti modularnosti:

- **Jednostavne promjene:** Svi elementi igre mogu se brzo prilagoditi kroz editor bez potrebe za pisanjem novog koda.
- **Ponovna upotrebljivost:** Komponente i skripte mogu se koristiti na različitim mjestima u igri, smanjujući potrebu za pisanjem redundantnog koda.
- **Brza iteracija:** Dizajneri mogu lako testirati i implementirati promjene, čime se smanjuje vrijeme potrebno za razvoj i povećava fleksibilnost igre.

5. ZAKLJUČAK

Ovaj završni rad prikazuje razvoj i implementaciju *top-down RPG* igre koristeći Unity *Game Engine-a*, jednog od najpopularnijih i najboljih alata za razvoj igara. Fokus ovog rada bio je na kreiranju zabavne igre koja kombinira klasične *RPG* elemente poput borbe, istraživanja i prikupljanja nagrada, uz korištenje modernih tehnologija i metoda razvoja igara. Kroz osvrt na povijest i značaj *RPG* igara, dobiven je jasan pregled napretka ovog žanra, od njegovih početaka poput „Dungeons & Dragons-a“ do današnjih digitalnih *RPG* naslova. Posebna pažnja posvećena je razvoju tehnologija i alata koji su omogućili izradu ove igre, uključujući Unity, Visual Studio, GitHub, te brojne druge platforme i resurse.

U razradi su detaljno opisani ključni aspekti razvoja igre, kao što su kontrola igrača, borbeni sustav, umjetna inteligencija neprijatelja te njihovi napadi, upravljanje *inventory-om*, dizajn nivoa i sustavi upravljanja scenama i korisničkim sučeljem. Rad je također istražio kako se koriste razne tehnologije i alati za kreiranje grafičkih elemenata, zvukova i efekata, te optimizaciju igre. Implementirani sustavi i skripte omogućili su fluidnu i dinamičnu igru, s raznolikim izazovima za igrače.

Jedan od glavnih zaključaka ovog rada je važnost modularnosti i fleksibilnosti u razvoju igara. Korištenjem modularnog dizajna i *SerializeField* polja, omogućena je brza i jednostavna prilagodba elemenata igre, što je olakšalo *kodiranje* i testiranje tijekom izrade igre. Ovaj pristup omogućava i jednostavnu nadogradnju igre, dodavanje novih elemenata i prilagodbu postojećih funkcionalnosti, bez potrebe za velikim promjenama u samom kodu i skriptama. Primjena ovog pristupa rezultirala je igrom koja je dinamična, prilagodljiva i spremna za daljnje proširenje.

Tijekom razvoja igre, suočio sam se s brojnim izazovima, poput optimizacije *performansi*, dizajna interaktivnih i vizualno privlačnih nivoa, te balansiranja težine igre i protivnika. Međutim, ti su izazovi također pružili priliku za stjecanje vrijednog iskustva i novih znanja o procesima razvoja igara. Radom na ovom projektu, postignut je cilj stvaranja kvalitetne *top-down RPG* igre koja pruža igračima zabavno i izazovno iskustvo, ali također nudi fleksibilnost za daljnje nadogradnje i prilagodbe.

Zaključno, ovaj rad demonstrira kako se korištenjem modernih alata i metoda razvoja igara, čak i mali timovi ili pojedinci mogu upustiti u stvaranje kvalitetnih i zabavnih igara. Projekt je također ukazao na važnost modularnosti i fleksibilnosti u razvoju igara, pružajući korisne smjernice za potencijalne buduće projekte.

6. Izjava o autorstvu

MEĐIMURSKO VELEUČILIŠTE U ČAKOVCU

Bana Josipa Jelačića 22/a, Čakovec

IZJAVA O AUTORSTVU

Završni/diplomski rad isključivo je autorsko djelo studenta te student odgovara za istinitost, izvornost i ispravnost teksta rada. U radu se ne smiju koristiti dijelovi tuđih radova (knjiga, članaka, doktorskih disertacija, magistarskih radova, internetskih i drugih izvora) bez pravilnog citiranja. Dijelovi tuđih radova koji nisu pravilno citirani, smatraju se plagijatom i nezakonitim *prisvajanjem tuđeg znanstvenog ili stručnoga rada. Sukladno navedenom studenti su dužni* potpisati izjavu o autorstvu rada.

Ja, LUKA MIJATOVIĆ (ime i prezime studenta) pod punom moralnom, materijalnom i kaznenom odgovornošću, izjavljujem da sam isključivi autor/ica završnog/diplomskog rada pod naslovom

IZRADA TOP-DOWN RPG IGRE

te da u navedenom radu nisu na nedozvoljeni način (bez pravilnog citiranja) korišteni dijelovi tuđih radova.

Student/ica:

Luka M

(vlastoručni potpis)

7. Literatura

[1] https://docs.unity.com (Datum pristupa: 01.07.2024.)

[2] https://www.aseprite.org/docs/ (Datum pristupa: 01.07.2024.)

[3] https://itch.io/game-assets/free (Datum pristupa: 10.07.2024.)

[4] https://craftpix.net/freebies/ (Datum pristupa: 10.07.2024.)

[5] https://pixabay.com/sound-effects/ (Datum pristupa: 10.07.2024.)

[6] https://gamedevacademy.org/how-to-create-an-rpg-game-in-unity-comprehensive-guide/ (Datum pristupa: 10.07.2024.)

[7] https://jdookeran.medium.com/creating-an-engaging-rpg-in-unity-a-step-by-step-guide-727c3889363c (Datum pristupa: 10.07.2024.)

[8] https://www.youtube.com/@ChrisTutorialsYT (Datum pristupa: 15.07.2024.)

[9] https://www.youtube.com/@BMoDev (Datum pristupa: 15.07.2024.)

[10] https://www.linkedin.com/company/unity/about/ (Datum pristupa: 10.08.2024.)

[11] https://discussions.unity.com (Datum pristupa: 01.07. – 20.08.2024.)

[12] https://www.pcgamer.com/the-complete-history-of-rpgs/ (Datum pristupa: 23.08.2024.)

[13] https://www.britannica.com/topic/role-playing-video-game (Datum pristupa: 23.08.2024.)

[14] https://ogres.fandom.com/wiki/History_of_Role-playing_Games (Datum pristupa: 23.08.2024.)

8. Popis ilustracija