Izrada 3D akcijske videoigre uloga u programskom alatu Unity

Vugrinec, Vili

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Vili Vugrinec

Izrada 3D akcijske videoigre uloga u programskom alatu Unity

ZAVRŠNI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Vili Vugrinec

Matični broj: 0016152964

Studij: Informacijski sustavi - Umreženi sustavi i računalne igre

Izrada 3D akcijske videoigre uloga u programskom alatu Unity

ZAVRŠNI RAD

Mentor/Mentorica:

Doc. dr. sc. Mladen Konecki

Varaždin, rujan 2024.

Vili Vugrinec

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

Razvoj video igara obuhvaća kompleksne aspekte koji zahtijevaju posvećenost i stručnost u različitim disciplinama. Proces uključuje implementaciju kretanja igrača, upravljanje resursima poput zdravlja i energije, te vođenje AI ponašanja, sve s ciljem stvaranja konzistentnog i uzbudljivog iskustva za igrače. Korištenje komponenti kao što su NavMeshAgent za AI navigaciju i ScriptableObject za pohranu podataka o čarolijama omogućava detaljnu kontrolu i prilagodbu. Implementacija ovih elemenata može biti izazovna, no donosi velike nagrade u smislu kreativnog zadovoljstva i uspjeha. Postavljanje NavMeshSurface za navigaciju, vizualizacija resursa kroz UI komponente, i pažljivo planiranje ključni su za stvaranje dinamičnih igara. Današnji alati omogućuju razvoj složenih igara s visokom razinom detalja uz minimalne troškove, naglašavajući važnost predanosti i inovativnosti u stvaranju visokokvalitetnih igara.

Ključne riječi: videoigra, Unity, vatrena kugla, C#, moć, zdravlje, energija, akcijska igra, razvoj videoigara

Sadržaj

1. Uvod 1
2. Metode i tehnike rada 2
2.1. Unity 2
3. Razrada teme 4
3.1. Postavljanje novog projekta u Unity alatu 4
3.2. Izrada 3D akcijske videoigre uloga 6
3.2.1. Ideja i interakcija s videoigrom 6
3.2.2. Animacija igrača 10
3.2.3. Kretanje igrača 13
3.2.4. Upravljanje životnim bodovima 16
3.2.5. Upravljanje moćima 18
3.2.6.1. Scriptable Objects 21
3.2.7. NavMeshAgent 24
3.2.8. Mehanika neprijatelja 26
3.2.9. Kreiranje terena 28
3.2.10. Prikazivanje zdravlja igrača 29
3.2.11. Prikazivanje energije igrača 30
3.2.12. Regeneracija zdravlja igrača 31
3.2.13. Cinemachine 32
4. Zaključak 35
5. Popis literature 36
6. Popis slika 37

1. Uvod

Tema ovog završnog rada usmjerena je na proces kreiranja 3D akcijske videoigre koristeći programski alat Unity, jedan od najpopularnijih alata za razvoj videoigara. Cilj ovog projekta je prikazati sve korake i tehnike korištene pri izradi jedne takve igre. Fokus implementacije bit će na razvoju različitih mehanika korištenja magije, koje će igrač koristiti u borbi uz pomoć mane kao ključnog resursa. Razvit će se i AI sustav za neprijatelje, što će igri dati dinamičnost i interaktivnost, čineći svaku borbu izazovnom i nepredvidljivom. U pisanom dijelu rada, detaljno će se opisati korišteni alati i tehnologije, uključujući Visual Studio i programski jezik C# u kontekstu objektnog programiranja. Osim toga, bit će objašnjeni svi elementi procesa izrade igre, od dizajna do implementacije. Praktični dio rada sastoji se od razvijene igre koja predstavlja konačni produkt ovog projekta. Ovaj projekt kombinira različite aspekte programiranja, dizajna i umjetničkog stvaranja, te istovremeno objedinjuje teorijsko i praktično znanje stečeno tijekom studija. Izrada ove igre omogućit će primjenu i demonstraciju stečenih vještina u stvarnom okruženju razvoja videoigara, što predstavlja važan korak prema daljnjem profesionalnom napretku u ovom području.

2. Metode i tehnike rada

Za potrebe razrade ove teme, istraživačke aktivnosti obuhvatile su analizu videoigara sličnih po žanru i mehanikama igranja. Glavna platforma za istraživanje tih igara bila je YouTube, koja je omogućila detaljno proučavanje različitih koncepata i stilova igranja te pronalazak osnovnih ideja za implementaciju određenih funkcionalnosti unutar igre.

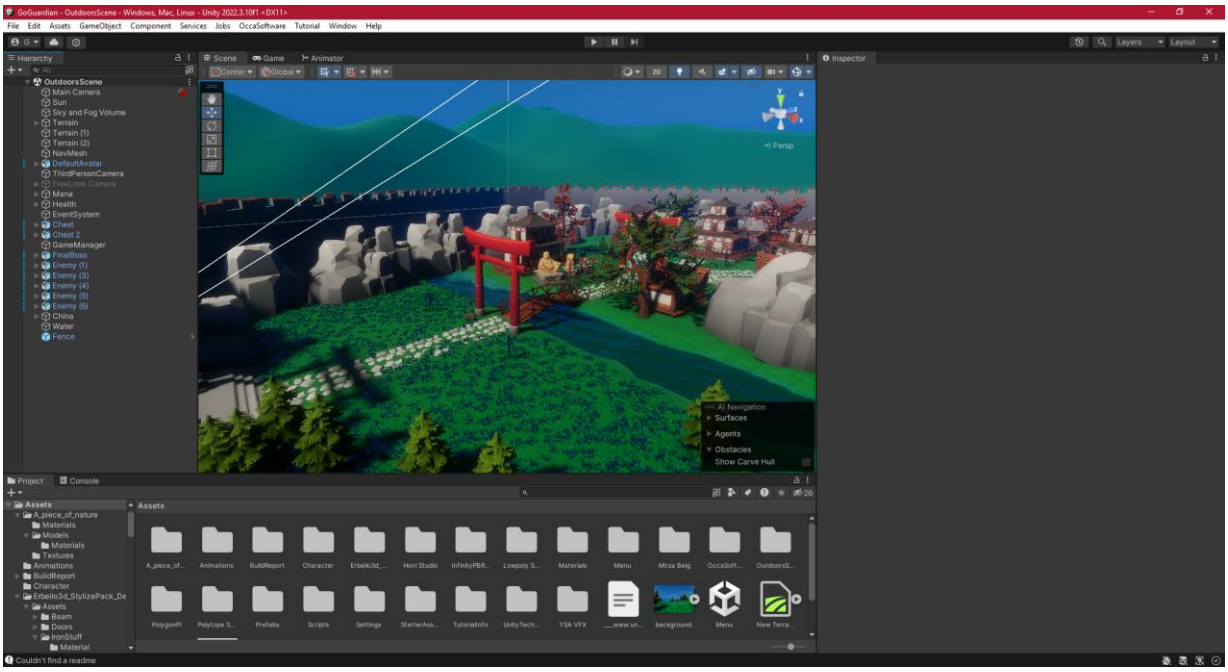
Videoigra je razvijena u Unityju [1], dok je Microsoft Visual Studio korišten za implementaciju programske logike. Za 3D modele i animacije korišteni su resursi iz Unity Asset Store-a i Mixamo platforme. Ovi alati i resursi omogućili su učinkovitu izradu vizualnih elemenata i dodavanje realističnih animacija, što je značajno doprinijelo konačnom izgledu i funkcionalnosti igre.

2.1. Unity

Unity je platforma za razvoj interaktivnih 3D igara i drugih sadržaja u stvarnom vremenu. Pruža programerima sveobuhvatne alate i značajke za izradu igara koje se mogu igrati na različitim platformama, uključujući računala, konzole, mobilne uređaje i virtualnu stvarnost.

Unity nudi intuitivno korisničko sučelje i snažan skriptni jezik C#, koji omogućava jednostavno stvaranje kompleksnih igara. Uključuje brojne ugrađene funkcije i resurse, poput simulacije fizike, upravljanja imovinom, alata za animaciju i robusnog mehanizma za renderiranje unutar Unity Game Enginea.

Platforma je pogodna za sve, od hobista do profesionalnih timova, pružajući svestrane mogućnosti za razvoj igara.



Slika 1. Unity sučelje

Programeri mogu stvarati i manipulirati objektima u 3D ili 2D okruženju, definirajući njihove osobine i ponašanja putem skripti i komponenti. Arhitektura temeljena na komponentama omogućava dodavanje različitih funkcionalnosti objektima igre, pružajući visoku razinu prilagodljivosti. Jedna od ključnih značajki Unityja je njegov vizualni uređivač koji omogućuje dizajn i izgradnju igre kroz jednostavno povlačenje i ispuštanje sredstava poput 3D modela, tekstura i audio datoteka. Programeri mogu lako rasporediti i konfigurirati ove resurse kako bi stvorili željeno iskustvo igranja.

Unity koristi C# kao primarni skriptni jezik, iako podržava i druge jezike poput Rust i JavaScript. C# je najčešće korišteni jezik unutar Unity zajednice, omogućujući programerima da pišu skripte za definiranje ponašanja objekata, upravljanje korisničkim unosom, implementaciju mehanike igre i još mnogo toga.

Platforma također nudi veliku trgovinu imovine, gdje korisnici mogu pronaći unaprijed izrađene resurse, skripte i predloške koji ubrzavaju razvoj i omogućuju fokusiranje na jedinstvene aspekte igre.

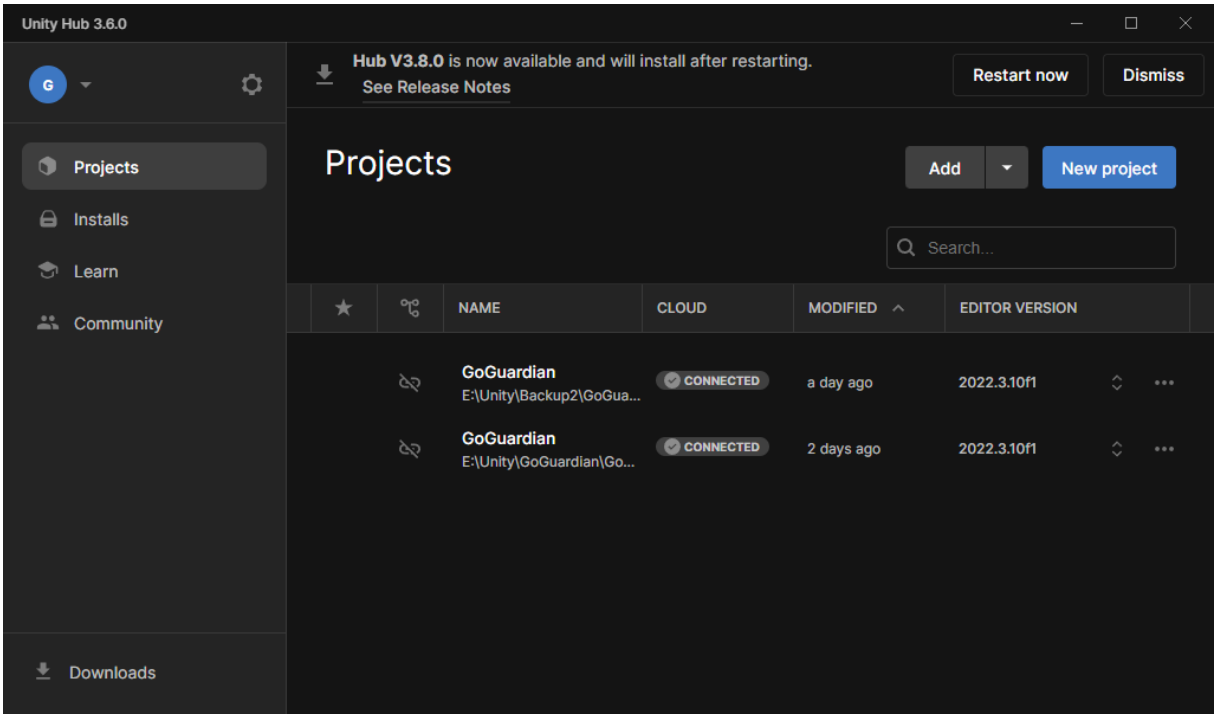
Unity podržava višekorisničke igre i umrežavanje u stvarnom vremenu, s ugrađenim mogućnostima za povezivanje, sinkronizaciju stanja igre i komunikaciju između igrača. Također pruža snažne sigurnosne značajke koje štite od prijetnji poput hakiranja i varanja, dok posvećeni sigurnosni tim nadzire i reagira na sigurnosne propuste.[1]

3. Razrada teme

U ovom poglavlju opisuje se proces razvoja igre. Prvo, objašnjava se kako je započet i organiziran projekt u Unityju, uključujući osnovne korake za postavljanje i izvođenje igre. Zatim se prikazuje proces animacije glavnog lika. Nakon toga, detaljno se razmatraju glavne klase unutar projekta, uz objašnjenje njihovih funkcija i načina na koji doprinose izvedbi i interakcijama unutar igre.

3.1. Postavljanje novog projekta u Unity alatu

Za postavljanje novog projekta u Unity Hubu, potrebno je otvoriti aplikaciju i odabrati opciju "Projects" u izborniku na lijevoj strani. Zatim se klikne na gumb "New project" u gornjem desnom kutu (obojan plavo na slici 2.) kako bi se započeo proces kreiranja novog projekta. Odabire se odgovarajući predložak (template) za vrstu igre ili aplikacije koja se razvija, poput 2D, 3D, HDRP ili URP. Nakon toga, unosi se ime projekta i odabire lokacija na disku gdje će projekt biti pohranjen. Također, iz stupca "EDITOR VERSION" bira se odgovarajuća verzija Unity-a. Na kraju, klikom na gumb "Create," projekt se kreira i automatski otvara u Unity Editoru.



Slika 2. Unity Hub – izrada novog projekta

Nakon klika na gumb "New project" u Unity Hubu, otvara se prozor u kojem se bira predložak (template) za novi projekt. U ovom prozoru može se odabrati vrsta projekta, poput 2D, 3D, HDRP ili URP, ovisno o potrebama razvoja. Detaljnije objašnjenje template-a:

1. Built-in Render Pipeline

Built-in Render Pipeline je zadani i najstariji render pipeline u Unityju. Dizajniran je za jednostavnost i široku kompatibilnost, pružajući solidne performanse na raznim platformama. Radi na principu unaprijed definiranih shadera i postavki koje se mogu mijenjati kroz grafičke postavke u editoru. Iako je lako koristiti i dovoljno fleksibilan za većinu projekata, Built-in Render Pipeline nema napredne mogućnosti prilagodbe kao što ih imaju noviji pipelineovi. Zbog toga se najčešće koristi za manje zahtjevne projekte ili za backward compatibility sa starijim verzijama Unityja.[2]

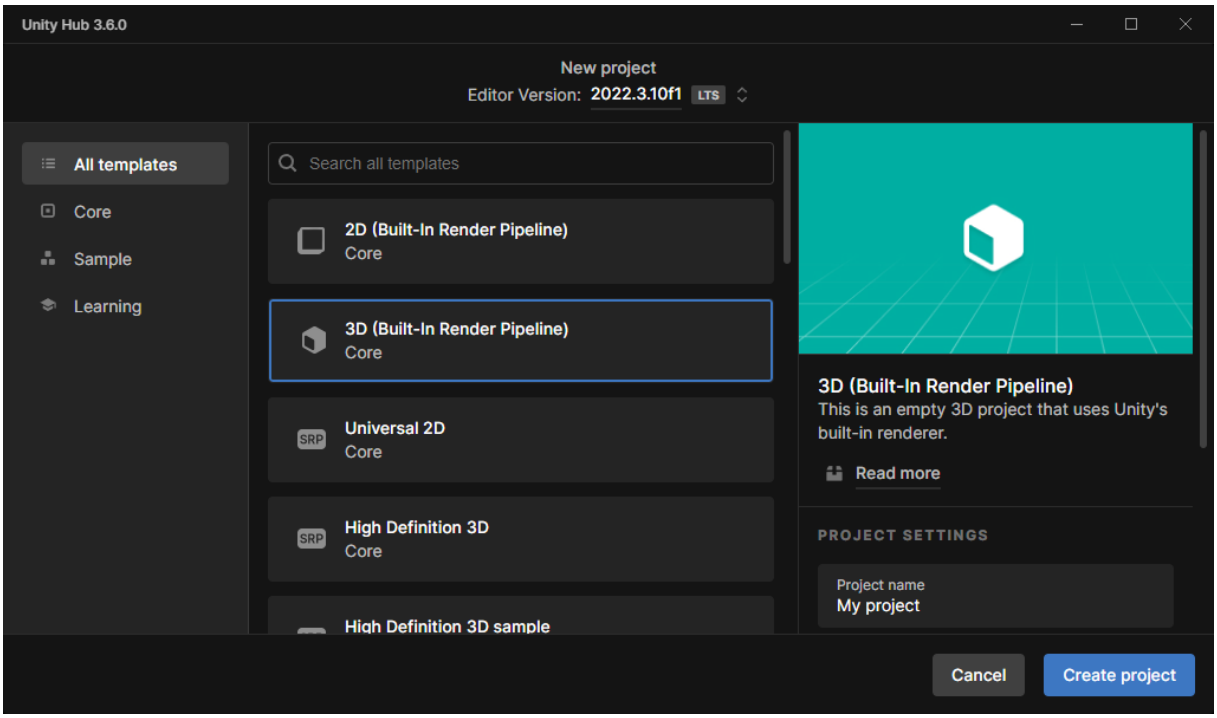
2. Universal Render Pipeline (URP)

Universal Render Pipeline, ranije poznat kao Lightweight Render Pipeline (LWRP), razvijen je kako bi pružio bolji balans između performansi i kvalitete. URP je optimiziran za rad na širokom spektru uređaja, od mobilnih uređaja do konzola i računala. Glavna prednost URP-a je njegova efikasnost u pogledu performansi, dok i dalje omogućuje visoku kvalitetu slike s naprednim efektima kao što su dinamičko osvjetljenje, post-procesiranje, i PBR (Physically Based Rendering). URP je prilagodljiviji od Built-in Render Pipelinea, s podrškom za prilagodbu render procesa putem Shader Graph-a, što omogućuje lakše kreiranje custom shadera bez potrebe za pisanjem koda. [3]

3. High Definition Render Pipeline (HDRP)

High Definition Render Pipeline je namijenjen za vrhunske grafičke performanse i vizualnu kvalitetu, prvenstveno na računalima i konzolama visokih performansi. HDRP koristi napredne tehnike renderiranja kao što su volumetričko osvjetljenje, real-time ray tracing, napredna kontrola refleksija i refrakcija, i podrška za velike količine geometrije i složene materijale. HDRP je idealan za projekte koji teže fotorealističnoj grafici, kao što su AAA videoigre, simulacije, i arhitektonske vizualizacije. Zbog visokih hardverskih zahtjeva, HDRP se obično koristi na projektima gdje se cilja na vrhunske uređaje i grafičke kartice. [4]

Unutar sučelja se također pruža mogućnost unosa imena projekta i odabira lokacije na kojoj će projekt biti pohranjen na disku. Na dnu prozora prikazana je verzija Unity-a koja će se koristiti za izradu projekta, a može se odabrati željena verzija ako je dostupno više opcija. Nakon unosa svih potrebnih informacija, klikom na "Create" započinje kreiranje projekta, koji se automatski otvara u Unity Editoru za daljnji rad.



Slika 3. Unity hub – odabir predloška projekta

3.2. Izrada 3D akcijske videoigre uloga

Igra je napravljena kombinacijom ručno kreiranih elemenata i pažljivo odabranih gotovih resursa kako bi se postigao skladan i smislen svijet igre. Primjerice, vatrena kugla koju igrač može bacati potpuno je ručno modelirana i animirana, čime se osigurava jedinstvenost i originalnost u ključnim aspektima igranja. Također, trava i teren su ručno oblikovani kako bi se prilagodili specifičnim potrebama igre i doprinijeli vizualnoj konzistenciji okoliša.

Osim tih elemenata, značajan dio vremena uložen je u istraživanje, odabir i integraciju gotovih resursa poput 3D modela, tekstura i zvučnih efekata. Ovi elementi su pažljivo odabrani i prilagođeni kako bi se uklopili u estetski stil igre.

3.2.1. Ideja i interakcija s videoigrom

Na pokretanja videoigre, pojavljuje se početni zaslon na kojem se nalaze tri gumba. Nakon što igrač klikne na prvi gumb "Play", igra se automatski pokreće, prelazeći na početnu scenu igre. Drugi gumb, "Manual", otvara priručnik koji igraču pruža osnovne informacije i upute o tome kako igrati igru. Zadnji gumb, "Quit", služi za prekid igre i izlazak iz aplikacije, vraćajući igrača na desktop.

Slika 4. Početni zaslon videoigre

Unutar igre, igrač se pojavljuje u open-world (otvorenom svijetu) okruženju gdje mu je glavni cilj poraziti sve neprijatelje koja ga napadaju. Neprijatelji imaju sposobnost smanjivanja igračevih životnih bodova, a ako se ti bodovi smanje na nulu, igra završava i igrač gubi.

Slika 5. Prikaz kraja igre - poraz

Igraču je omogućeno da se brani i uzvrati napad. Lijevim klikom miša, igrač baca vatrenu kuglu u smjeru u kojem je okrenut, čime smanjuje životne bodove neprijatelja. Međutim, korištenje ove magije troši igračevu manu, koja predstavlja energiju potrebnu za izvođenje magijskih napada i sposobnosti. Mana je ograničen resurs, što znači da igrač ne može beskonačno koristiti vatrene kugle. Mana je prikazana u lijevom gornjem kutu, ljubičasto dok su životni bodovi prikazani crveno.

Slika 6. Prikaz korištenja vaternih kugli

Kako bi regenerirao manu, igrač može držati tipku **E**, pri čemu njegov lik počinje plesati, čime polako obnavlja svoju manu.

Slika 7. Prikaz obnavljanja energije igrača

Životne bodove igrač može regenerirati dolaskom do skrivenih škrinja koje se nalaze na mapi. Kada igrač stoji pokraj škrinje, životni bodovi mu se postupno obnavljaju.

Slika 8. Prikaz obnavljanja životnih bodova

Osim osnovnih napada, igrač ima i posebnu moć koja zahtijeva više mane i ima određeno vrijeme izvođenja. Ova moć se aktivira držanjem tipke **1**. Ako se koristi u pravom trenutku i na pravom mjestu, moć može blokirati napade neprijatelja, pružajući igraču važnu prednost u borbi.

Slika 9. Prikaz korištenja posebne moći

Kada igrač uspješno porazi sve neprijatelje, igra završava i igrač može započeti igru ispočetka.

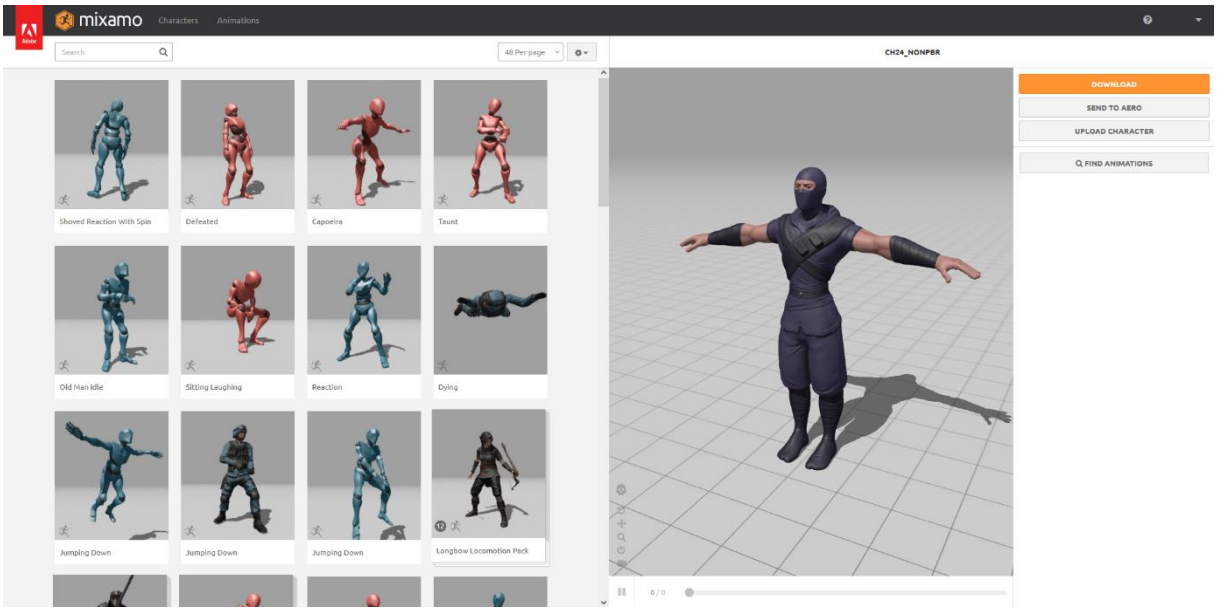
Slika 10. Prikaz kraja igre – pobjeda

3.2.2. Animacija igrača

Za ovu videoigru korišten je Mixamo za animaciju igrača. Mixamo nudi širok spektar kvalitetnih animacija koje su integrirane u igru, omogućujući fluidne i prirodne pokrete likova.

Mixamo je internetska platforma koju je razvio Adobe, a namijenjena je za stvaranje i prilagođavanje 3D modela te njihovu animaciju. Omogućuje korisnicima preuzimanje visokokvalitetnih 3D modela koji su već opremljeni kosturom (riggiranjem), što znači da su spremni za korištenje u raznim projektima, uključujući videoigre i animacije, bez potrebe za dodatnim ručnim radom na riggiranju. Rigging je proces dodavanja kostura 3D modelu kako bi se omogućilo kretanje i animacija, a uobičajeno je vrlo složen i zahtjevan proces. Mixamo eliminira tu složenost, omogućujući korisnicima da odmah započnu s animacijom modela. [5]

Jedna od ključnih prednosti Mixamo platforme je njezina bogata kolekcija 3D modela i animacija. Korisnici mogu birati između različitih modela, prilagoditi ih svojim potrebama i preuzeti ih u formatima koji su kompatibilni s najčešće korištenim alatima za razvoj igara, poput Unityja. Osim toga, Mixamo omogućuje korisnicima pregled i primjenu raznih animacija na odabrane modele, što olakšava integraciju modela u bilo koji projekt. Ove animacije mogu uključivati osnovne pokrete kao što su hodanje, trčanje ili skakanje, ali i složenije radnje, poput borbenih pokreta ili plesova.

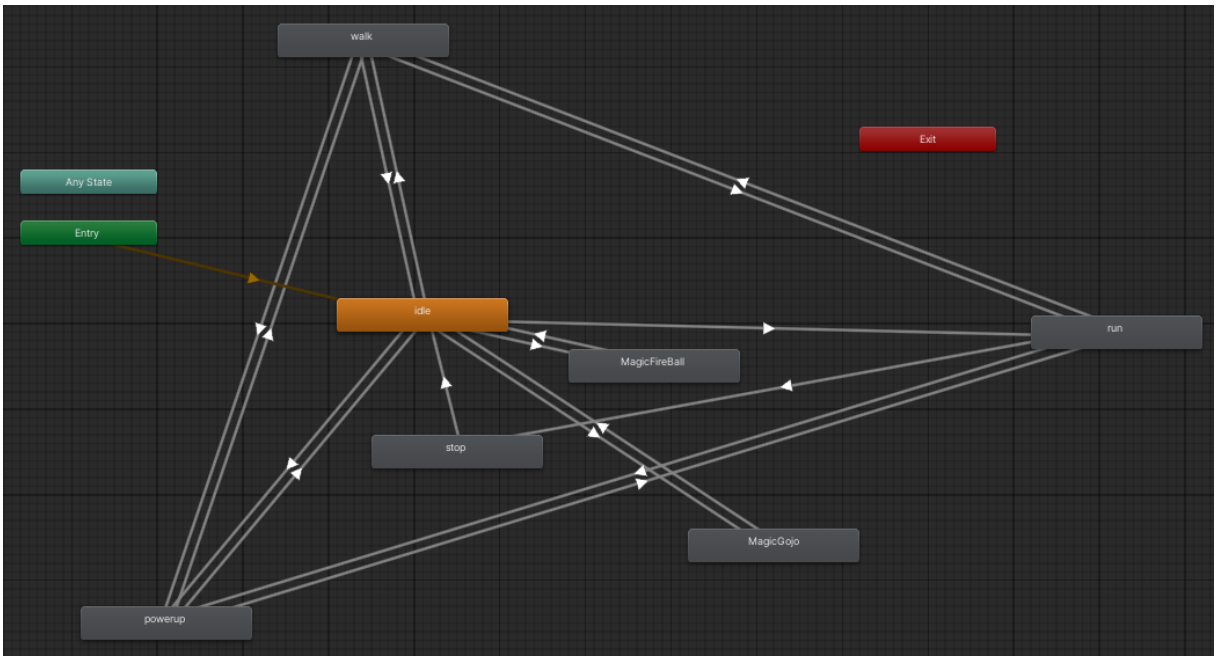


Slika 11. Prikaz mixamo sučelja za odabir animacija

Na glavnoj stranici Mixamo moguće je pregledavati modele prema različitim kategorijama ili koristiti pretraživač kako bi pronašli specifične stilove ili vrste likova. Kada se pronađe odgovarajući model, treba ga preuzeti u formatu koji je kompatibilan s Unityjem, obično kao FBX datoteku.

Nakon preuzimanja 3D modela, sljedeći korak je odabir animacija koje će se koristiti. Mixamo nudi širok raspon animacija, uključujući osnovne pokrete poput hodanja i trčanja, kao i složenije akcije poput borbenih pokreta ili plesova. Na Mixamo stranici za animacije može se filtrirati i pregledavati različite animacije prema vrsti pokreta ili emociji koju treba postići. Kada se pronađe odgovarajuća animacija, potrebno ju je preuzeti i povezati s modelom. [5]

Nakon preuzimanja modela i animacija, sve datoteke treba uvesti u Unity. To se postiže povlačenjem preuzetih FBX datoteka u projekt u Unityju. Unity automatski prepoznaje te datoteke i konvertira ih u materijale koje je moguće koristiti. Za korištenje animacija, treba dodati animaciju u Animator Controller i povezati taj Animator Controller s 3D modelom. Kada se svi potrebni koraci završe, likovi će biti spremni za igranje animacija u Unityju, čime će se postići dinamični i realistični pokreti u igri.



Slika 12. Prikaz animatora

Animator u Unityju ključna je komponenta za upravljanje animacijama 3D modela i likova unutar igre. On omogućuje kreiranje i kontrolu animacija kroz Animator Controller, koji služi kao sredstvo za definiranje kako i kada se animacije odvijaju.

Animator Controller koristi animacijske state-ove (stanja), koji predstavljaju različite faze animacije, poput hodanja, trčanja ili skakanja. Svaki state može biti povezan s jednom ili više animacija. Animator Controller omogućuje definiranje prijelaza između ovih state-ova, koji se aktiviraju na temelju uvjeta i parametara. Na primjer, prijelaz s hodanja na trčanje može se automatski dogoditi kada brzina lika premaši određeni prag. [6]

U Animatoru u Unityju nalaze se različita stanja koja se odnose na različite aktivnosti i sposobnosti igrača:

1. **Stanje "Powerup"**: Ovo stanje označava trenutak kada igrač puni energiju ili manu te pleše. Kada je lik u ovom stanju, animacija prikazuje kako igrač prikuplja energiju i istovremeno izvodi plesne pokrete.
2. **Stanje "Walk"**: Ovo stanje odnosi se na normalno hodanje igrača bez trčanja. Kada je igrač u ovom stanju, animacija prikazuje sporije kretanje koje se koristi kada igrač samo hoda.
3. **Stanje "MagicFireBall"**: Ovo stanje se aktivira kada igrač ispaljuje vatrene lopte. Animacija u ovom stanju prikazuje kako igrač koristi magične sposobnosti za lansiranje vatrenih lopti prema neprijateljima ili ciljevima.

4. **Stanje "Stop"**: Ovo stanje označava situaciju kada igrač prestane s bilo kakvim kretanjem ili akcijama, tj. kada se ništa ne događa i igrač ne koristi tipkovnicu za kontrolu. Animacija u ovom stanju prikazuje lik u mirnom stanju, bez ikakvih pokreta.
5. **Stanje "MagicGojo"**: Ovo stanje odnosi se na posebnu moć koju igrač aktivira držanjem tipke 1. Animacija prikazuje kako igrač koristi ovu posebnu moć ili sposobnost koja može imati različite učinke u igri.
6. **Stanje "Run"**: Ovo stanje označava trčanje igrača kada se drži tipka Shift. Animacija u ovom stanju prikazuje brže kretanje igrača, čime se omogućuje brzo kretanje kroz igru.

Svako od ovih stanja koristi se za različite akcije i sposobnosti igrača, omogućujući dinamično i raznoliko ponašanje unutar igre.

3.2.3. Kretanje igrača

U Unityju, upravljanje kretanjem igrača osigurava prirodno i dinamično iskustvo igre. Videoigra koristi ulaze s tipkovnice za kontrolu smjera i brzine kretanja igrača, te upravlja animacijama za hodanje, trčanje, skakanje i specijalne sposobnosti.

Korištenjem skripte PlayerMovControll za premještanje i Animator za promjenu animacija, kod omogućuje igraču da se kreće, rotira prema smjeru gledanja, i aktivira različite akcije, čime se postiže fluidna i responzivna interakcija unutar igre.

```
[Header("Movement Settings")]
    public float movementSpeed;
    public float dragOnGround;
    public float jumpStrength;
    public float jumpDelay;
    public float airControlMultiplier;
    bool canJump;
    [HideInInspector] public float walkingSpeed;
    private float runningSpeed;
    [Header("Key Bindings")]
    public KeyCode jumpButton = KeyCode.Space;
    [Header("Ground Detection")]
    public float characterHeight;
    public LayerMask groundLayer;
    bool isGrounded;
    public Transform playerOrientation;
    float inputHorizontal;

    float inputVertical;
    Vector3 movementDirection;
    private Animator playerAnimator;
    Rigidbody playerRigidbody;
    bool isSprinting;
```

Na početku, deklariraju se varijable koje postavljaju osnovne parametre za kretanje, skakanje i animacije. Varijable kao što su movementSpeed, jumpStrength, dragOnGround, i

airControlMultiplier definiraju kako se igrač kreće, koliko brzo može trčati, koliko visoko može skočiti, te kako će se ponašati dok je u zraku. Također su prisutni KeyCode varijable poput jumpButton koje omogućuju postavljanje tipki za skok i ostale radnje. Na početku funkcije Start, komponenta Rigidbody se inicijalizira da kontrolira fizičko kretanje lika, dok Animator kontrolira animacije. [7]

```
isGrounded = Physics.Raycast(transform.position, Vector3.down,
characterHeight * 0.5f + 0.3f, groundLayer);

        HandleInput();
        ControlSpeed();

private void HandleInput()
    {
        inputHorizontal = Input.GetAxisRaw("Horizontal");
        inputVertical = Input.GetAxisRaw("Vertical");
        if (Input.GetKey(jumpButton) && canJump && isGrounded)
        {
            canJump = false;
            Jump();
            Invoke(nameof(ResetJump), jumpDelay);
        }
    }
```

U funkciji Update, stalno se provjerava je li igrač na tlu koristeći RayCast, koji šalje zraku prema dolje i provjerava postoji li tlo ispod lika. Zatim se u metodi HandleInput prikuplja ulaz od igrača pomoću GetAxisRawkoji funkcija koji detektiraju smjerove kretanja igrača po osi X i Z. Ako igrač pritisne lijevu tipku Shift, aktivira se trčanje, što povećava brzinu kretanja.

```
private void MovePlayer()
    {
        movementDirection = playerOrientation.forward * inputVertical +
playerOrientation.right * inputHorizontal;

        if (isGrounded)
            playerRigidbody.AddForce(movementDirection.normalized *
movementSpeed * 10f, ForceMode.Force);
        else
            playerRigidbody.AddForce(movementDirection.normalized *
movementSpeed * 10f * airControlMultiplier, ForceMode.Force);
    }
```

Funkcija MovePlayer koristi ulaz od igrača kako bi definirala smjer kretanja koristeći orijentaciju lika (playerOrientation.forward i playerOrientation.right). Zatim se koristi funckaija AddForce koja doda silu na rigidbody komponenti koja se nalazi na igraču odnosno doda se sila na igrača u smjeru u kojem se želi kretati. Ako je igrač na tlu, sila se primjenjuje normalno, dok se u zraku primjenjuje modifikator sile kako bi se kretanje u zraku kontroliralo (air control).

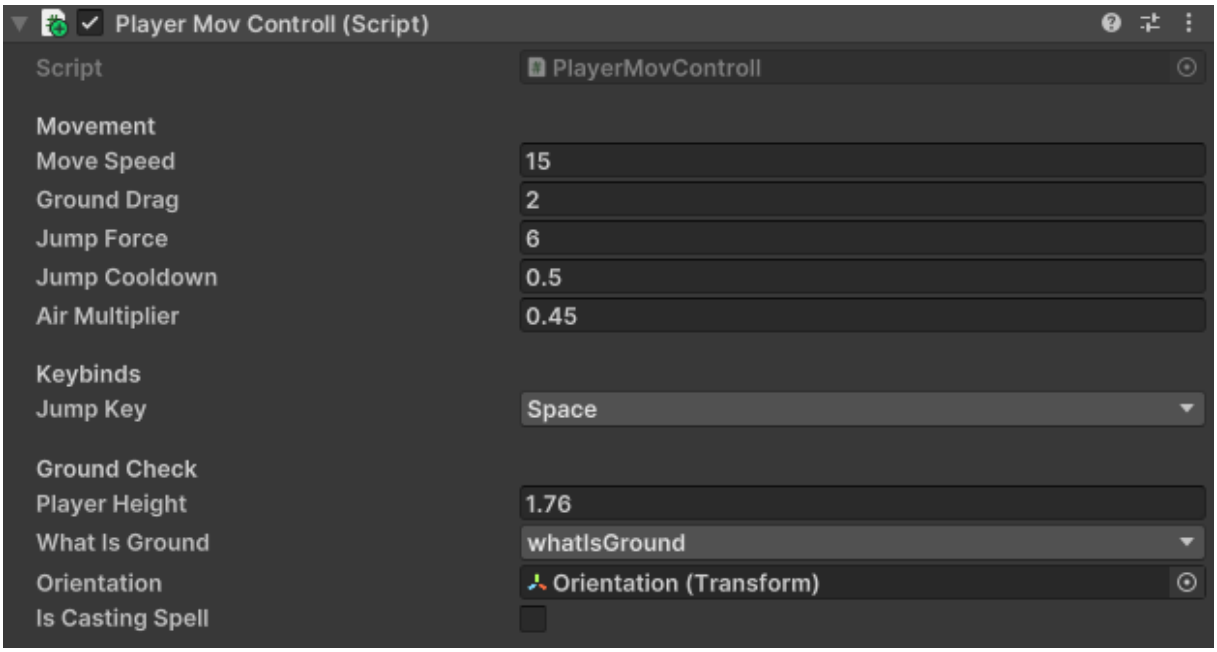
```
private void ControlSpeed()
    {
        if (!isSprinting)
        {
            Vector3 flatVelocity = new Vector3(playerRigidbody.velocity.x,
0f, playerRigidbody.velocity.z);
```

```
            if (flatVelocity.magnitude > movementSpeed)
            {
                Vector3 limitedVelocity = flatVelocity.normalized *
movementSpeed;
                playerRigidbody.velocity = new Vector3(limitedVelocity.x,
playerRigidbody.velocity.y, limitedVelocity.z);
            }
        }
    }
```

Funkcija ControlSpeed osigurava da igrač ne premaši određenu brzinu. Ako je brzina kretanja veća od dozvoljene (movementSpeed), ona se ograničava. Ovo osigurava da igrač ne može dosegnuti prevelike brzine, pogotovo kad nije u modu trčanja.

```
private void Jump()
    {
        playerRigidbody.velocity = new Vector3(playerRigidbody.velocity.x,
0f, playerRigidbody.velocity.z);
        playerRigidbody.AddForce(transform.up * jumpStrength,
ForceMode.Impulse);
    }
```

Skakanje se implementira u funkciji Jump. Kada igrač pritisne jumpButton (Space), funkcija resetira vertikalnu brzinu odnosno postavlja ju na 0 kako bi osigurao dobar skok, zatim dodaje silu prema gore pomoću AddForce funkcije prema gore. Funkcija ResetJump koristi se da omogući ponovno skakanje nakon određenog vremena to je zapravo varijabla jumpDelay, kako bi se spriječilo višestruko skakanje u kratkom vremenu.



Slika 13. Prikaz parametra za kretanje igrača

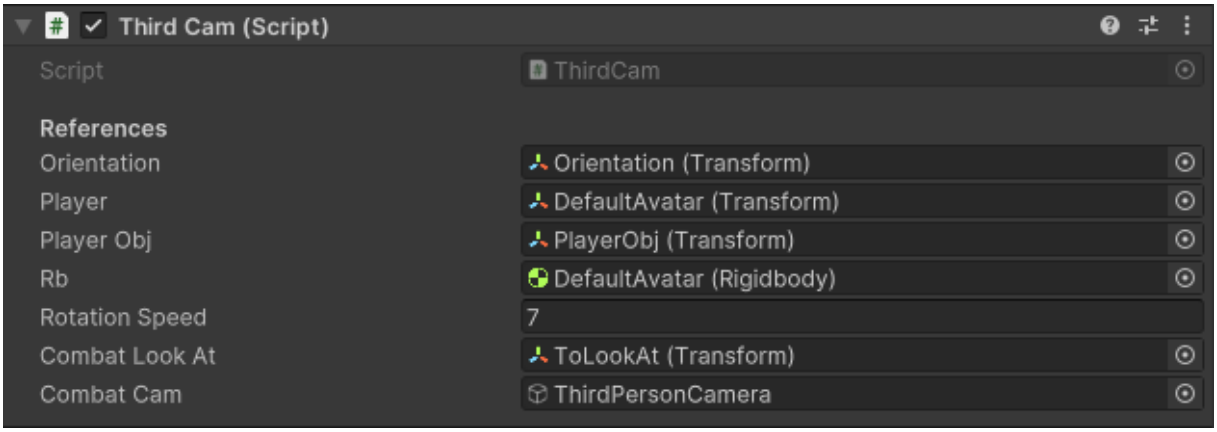
```
[Header("References")]
public Transform playerOrientation;
public Transform playerTransform;
public Transform playerObject;
public Rigidbody playerRigidbody;
```

```
public float playerRotationSpeed;
public Transform combatTarget;
public GameObject thirdPersonCamera;

private void Update()
{
    if (combatTarget != null)
    {
        Vector3 directionToCombatTarget = combatTarget.position - new
Vector3(transform.position.x, combatTarget.position.y,
transform.position.z);
        playerOrientation.forward = directionToCombatTarget.normalized;

playerRigidbody.MoveRotation(Quaternion.Slerp(playerRigidbody.rotation,
Quaternion.LookRotation(directionToCombatTarget.normalized), Time.deltaTime
* playerRotationSpeed));
    }
}
```

Ova skripta ThirdCam kontrolira rotaciju igrača prema točki ciljanja (combatTarget) koristeći fizičku komponentu Rigidbody i orijentaciju (playerOrientation).

U metodi Start, zaključava se pokazivač miša kako bi igrač mogao slobodno pomicati kameru bez ometanja kursora.

U metodi Update, izračunava se smjer od igrača prema točki ciljanja, pri čemu se zanemaruje y os, kako bi igrač ostao uspravan. Orijentacija igrača playerOrentation se postavlja prema smjeru ciljanja, a rotacija se glatko interpolira pomoću Slerp funkcije i primjenjuje na Rigidbody igrača. Ovaj postupak omogućava prirodno i glatko okretanje igrača prema cilju, pri čemu je brzina rotacije kontrolirana varijablom playerRotationSpeed.



Slika 14. Prikaz parametara za rotaciju kamere i igrača

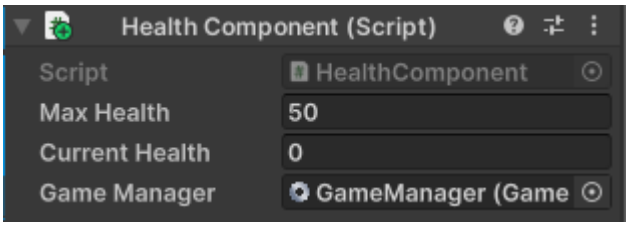
3.2.4. Upravljanje životnim bodovima

Životni bodovi (ili HP - Health Points) su ključni elementi u mnogim video igrama, koji igrača obavještavaju o trenutnom stanju zdravlja njegovog lika. Oni predstavljaju mjernu

jedinicu koja pokazuje koliko štete igrač može primiti prije nego što lik postane nesposoban za daljnju igru ili "umre".

```
[SerializeField]public float maxHealth = 50f;
[SerializeField]public float currentHealth;
[SerializeField]private GameManager gameManager;
```

Ove varijable služe za upravljanje životnim bodovima lika u igri i povezivanje s GameManager komponentom. Varijabla maxHealth postavlja maksimalni broj životnih bodova, currentHealth prati trenutne bodove, a gameManager povezuje igru s općim upravljačem igre za integrirano upravljanje igrom. Atribut [SerializeField] omogućuje da se ove varijable uređuju i prate iz Unityjevog Inspektora, što omogućuje pojednostavljenu konfiguraciju i ispravljanje grešaka.



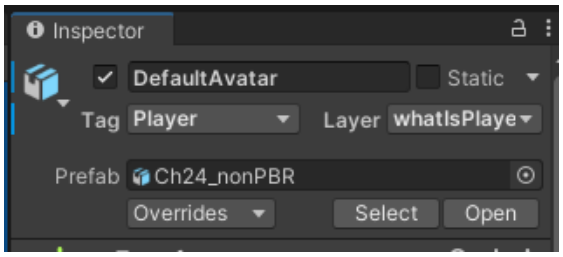
Slika 15. Prikaz Health Componente u Unity Inspektoru

```
private void Awake()
    {
        currentHealth = maxHealth;
    }
    public void TakeDamage(float damageToApply)
    {
        currentHealth -= damageToApply;
        if (currentHealth <= 0)
        {
            if (this.gameObject.CompareTag("Enemy"))
            {
                gameManager.OnEnemyKilled();
            }
            Destroy(this.gameObject);
        }
    }
    public void Heal(float amount)
    {
        currentHealth += amount;
        currentHealth = Mathf.Clamp(currentHealth, 0, maxHealth);
    }
```

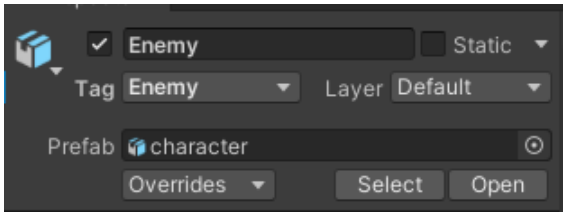
Skripta HealthComponent upravlja zdravljem lika u igri, omogućujući mu da prima štetu i iscjeljuje se. U metodi Awake, koja se poziva prilikom inicijalizacije objekta, trenutni broj životnih bodova currentHealth postavlja se na maksimalnu vrijednost maxHealth. Metoda TakeDamage smanjuje currentHealth za iznos štete damageToApply, provjerava je li zdravlje palo na nulu ili ispod. Ako je objekt označen kao "Enemy" (neprijatelj), poziva se metoda OnEnemyKilled iz gameManager klase, a zatim se objekt uništava. Metoda Heal povećava

currentHealth za zadani iznos i koristi Mathf.Clamp kako bi osigurala da zdravlje ne prelazi maksimalnu vrijednost maxHealth ili ne padne ispod nule. Metoda Heal se poziva kada igrač stoji pokraj škrinje u određenom krugu. Ove metode omogućuju dinamično upravljanje zdravljem lika, uključujući primanje štete, iscjeljivanje i reakciju na smanjenje zdravlja. Dakle ova skripta se koristi i na igraču i na neprijateljima.

Kako bi ovo funkcioniralo potrebno je postaviti zadane oznake na igrača i na neprijatelje.



Slika 16. Prikaz oznake „player“ na igraču



Slika 17. Prikaz oznake „Enemy“ na neprijatelju

3.2.5. Upravljanje moćima

Skripta SpellsControll upravlja moćima u igri omogućuje igraču korištenje specijalnih sposobnosti, poput magičnih napada. Ona upravlja aktiviranjem i deaktiviranjem moći, kao i resursima potrebnim za njihovo korištenje, čime dodaje dinamičnost i složenost igračkom iskustvu.

```
 [SerializeField] private Spell spellToCast;
public float maxMana = 180f;
public float currentMana;
[SerializeField] private float manaRechargeRate = 15f;
[SerializeField] private float timeBetweenCasts = 0.25f;
public RedHollowControl redHollowTransform;
[SerializeField] private Transform redHollowCastPoint;
[SerializeField] private GameObject redHollowObject;
private Animator animator;
private float currentCastTimer;
[SerializeField] private Transform castPoint;
private bool castingMagic = false;
```

Skripta upravlja sustavom mana i moći u igri. Varijable maxMana i currentMana definiraju maksimalni i trenutni broj mana, dok manaRechargeRate postavlja brzinu

obnavljanja mane odnosno energije igrača. Varijabla timeBetweenCasts kontrolira vrijeme između korištenja moći, a spellToCast referencu na moć koju lik može koristiti. redHollowTransform, redHollowCastPoint, i redHollowObject upravljaju pozicijom i vizualizacijom objekta vezanog uz specijalnu moć. animator upravlja animacijama, currentCastTimer prati vrijeme između korištenja moći, dok castPoint označava mjesto aktiviranja moći odnosno vatrene kugle. Varijabla castingMagic prati status korištenja moći, omogućujući učinkovito upravljanje svim aspektima sustava moći i energije igrača.

```
bool hasMana = currentMana - spellToCast.SpellToCast.ManaCost > 0f;
        if (!castingMagic && Input.GetKeyDown(KeyCode.Mouse0) && hasMana)
        {
            currentMana -= spellToCast.SpellToCast.ManaCost;
            castingMagic = true;
            CastSpell();
        }
```

Aktiviranje moći u igri se odrađuje tako što se provjerava ima li lik dovoljno energije i da li se klika lijevi klik miša. Varijabla hasMana provjerava je li preostalo enrgije dovoljno za korištenje moći tako da uspoređuje trenutnu količinu energije s troškom energije za tu moć. Ako je energija dovoljna, trenutna energija se smanjuje za trošak moći, a castingMagic se postavlja na true kako bi označio da je moć aktivirana. Kada su svi uvjeti zadovoljeni — nema aktivne moći, lijevi klik miša je pritisnut i ima dovoljno energije — metoda CastSpell se poziva za izvršenje moći.

```
if (castingMagic)
        {
            currentCastTimer += Time.deltaTime;
            animator.SetBool("SpellCast", true);
      if (currentCastTimer > timeBetweenCasts) castingMagic = false;
        }
else
        {
            animator.SetBool("SpellCast", false);
        }
```

Navedena skripta se odnosi na aktivaciju specijalne moći. Kada je castingMagic postavljen na true, currentCastTimer se povećava svakim frameom(sličicom) prema proteklom vremenu, omogućujući praćenje trajanja aktivacije. U isto vrijeme, animator se obavještava da je moć u aktivaciji postavljanjem SpellCast na true, što može pokrenuti odgovarajuću animaciju. Kada proteklo vrijeme currentCastTimer prelazi unaprijed postavljeni interval između korištenja moći timeBetweenCasts, castingMagic se postavlja na false, označavajući da je proces aktivacije završen. Ovaj kod omogućuje da se aktivacija moći pravilno prikaže i završi nakon što prođe određeno vrijeme.

```
if (currentMana < maxMana && !castingMagic && Input.GetKey(KeyCode.E))
        {
            currentMana += manaRechargeRate * Time.deltaTime;
            if (currentMana > maxMana) currentMana = maxMana;
        }
```

Punjenje energije upravlja se kada igrač drži tipku E. Ako trenutni broj energije je manji od maksimalnog broja energije i ako moć trenutno nije aktivna, tada se energija obnavlja. Varijabla currentMana povećava se za brzinu obnove energije manaRechargeRate pomnoženu s vremenom koje je prošlo od posljednjeg sličice. Ako se nakon obnove energija poveća iznad maksimalnog broja energije, varijabla currentMana se postavlja na maksimalni broj energije maxMana. Ovaj dio omogućuje igraču da obnavlja energiju dok drži tipku E, osiguravajući da energija polako raste i ne prelazi maksimalnu vrijednost.

```
if(redHollowObject==null && Input.GetKeyDown(KeyCode.Alpha1) && currentMana
>= 50)
        {
            CastHollowSpell()
        }
```

Provjerava se je li redHollowObject null, to znači da ne referencira nikakav objekt u igri ili da nije dodijeljena nijedna instanca objekta, nakon toga se provjerava da li je pritisnuta tipka 1 i ima li trenutni broj energije (currentMana) najmanje 50. Ako su svi uvjeti ispunjeni, poziva se metoda CastHollowSpell koja aktivira specijalnu moć.

```
void CastSpell()
    {
        GameObject fireball = Instantiate(spellToCast.gameObject,
castPoint.position, Quaternion.identity);
        Vector3 fireballDirection = Camera.main.transform.forward;
        fireball.transform.forward = fireballDirection;
    }
void CastHollowSpell()
    {
         redHollowObject = Instantiate(redHollowTransform.gameObject,
redHollowCastPoint.position, redHollowCastPoint.rotation, transform);

redHollowObject.gameObject.GetComponent<RedHollowControl>().playerAnimator
= animator;
        currentMana -= redHollowTransform.manaCostHollow;
        }
    }
```

Metoda CastSpell instancira (stvara) novi objekt moći koristeći varijablu spellToCast na poziciji i rotaciji definiranoj ovisno o smjeru glavne kamere. Ovo omogućuje da se moć (čarolija) pojavi u igri na određenom mjestu i s određenim orijentacijom odnosno rotacijom smjera ciljanja igrača.

Metoda CastHollowSpell instancira novi objekt koristeći redHollowTransform.gameObject na poziciji i rotaciji definiranoj varijablom redHollowCastPoint. Novi objekt redHollowObject postavlja se kao dijete „igrača“, što omogućuje kretanje zajedno s njim. Nakon instanciranja, playerAnimator se dodjeljuje komponenti RedHollowControl objekta kako bi se kontrolirale animacije povezane s ovom

moći. Također, trenutni broj energije (currentMana) smanjuje se za trošak moći manaCostHollow, što odražava potrošnju resursa pri aktivaciji specijalne moći.

Skripta RedHollowSpell odnosi se na posebnu moć koju igrač aktivira kada drži tipku 1. Posebna moć ima drugačije varijable od obične moći jer je jača, što se podrazumijeva da troši više energije i radi veću štetu neprijatelju.

```
private Rigidbody myRigidbody;
    public float manaCostHollow = 50f;
    public float specialSpell = 45f;
    void Start()
    {
        myCollider = GetComponent<MeshCollider>();
        myRigidbody = GetComponent<Rigidbody>();
        myRigidbody.isKinematic = true;
    }
    public void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.CompareTag("Enemy"))
        {
            HealthComponent enemyHealth =
collision.gameObject.GetComponent<HealthComponent>();
            enemyHealth.TakeDamage(specialSpell);
        }
```

U metodi Start, dobiva se referenca na MeshCollider i Rigidbody komponente objekta, a rigidbody se postavlja na isKinematic = true, što znači da se fizičke sile ne primjenjuju na taj objekt, ali se i dalje može ručno pomaknuti i upravljati njime. Varijabla manaCostHollow označava trošak energije za specijalnu moć, dok varijabla specialSpell predstavlja količinu štete koju specijalna moć nanosi. U metodi OnCollisionEnter, provjerava se je li objekt u koliziji s objektom koji ima tag "Enemy". Ako jest, dohvaća se komponenta HealthComponent s tog neprijateljskog objekta i poziva se metoda TakeDamage koja nanosi štetu određenom iznosu, definiranim varijablom specialSpell.

3.2.6.1. Scriptable Objects

Scriptable Objects u Unityju su posebna vrsta objekata koji omogućuju spremanje podataka u assetima, a ne samo u instancama u sceni ili u skriptama. Oni su dizajnirani da budu jednostavni, a istovremeno moćni alati za organiziranje i upravljanje podacima koji se često koriste u igrama.

U kontekstu skripte koja koristi Scriptable Objects za upravljanje čarolijama (spells), svaki specifični Spell može biti predstavljen kao Scriptable Object. Ovo omogućuje da se definiraju različite karakteristike i atributi čarolija, kao što su trošak energije, šteta, trajanje i efekti, odvojeno od glavnog koda igre. Na taj način, podaci o čarolijama mogu se lako mijenjati i prilagođavati putem Unity Editor-a bez potrebe za izmjenama u skriptnom kodu. [8]

```
public SpellScriptableObj SpellToCast;
private SphereCollider myCollider;
```

```
private Rigidbody myRigidbody;
[SerializeField] private Collider playerCollider;
public GameObject fireballExplosion;
```

Ovdje su definire varijable koje služe za upravljanje moćima i njihovim interakcijama u igri. Varijabla SpellToCast referencira Scriptable Object koji sadrži podatke o čaroliji koju objekt koristi, uključujući njene karakteristike kao što su trošak energije i efekti. myCollider je privatna varijabla za SphereCollider komponentu, koja će biti korištena za detekciju kolizija u obliku sfere. myRigidbody je privatna varijabla za Rigidbody komponentu, koja omogućava objektu da koristi fizičke sile i interakcije. Varijabla playerCollider je označena s SerializeField i omogućuje ručno postavljanje Collider komponente u Unity Editoru, koja služi za detekciju sudara između igrača i drugih objekata. Na kraju, fireballExplosion je varijabla koja referencira GameObject koji predstavlja eksploziju vatrene lopte, što je vizualni efekt koji se aktivira kada čarolija udari metu.

```
private void Update()
    {
        if(SpellToCast.Speed > 0) transform.Translate(Vector3.forward *
SpellToCast.Speed * Time.deltaTime);
    }
```

Unutar metode Update se provjerava ako je brzina SpellToCast.Speed veća od nule, a zatim pomiče objekt prema naprijed. Ova formula omogućuje da se objekt pomiče ravnomjerno i precizno u prostoru u odnosu na brzinu koju čarolija određuje.

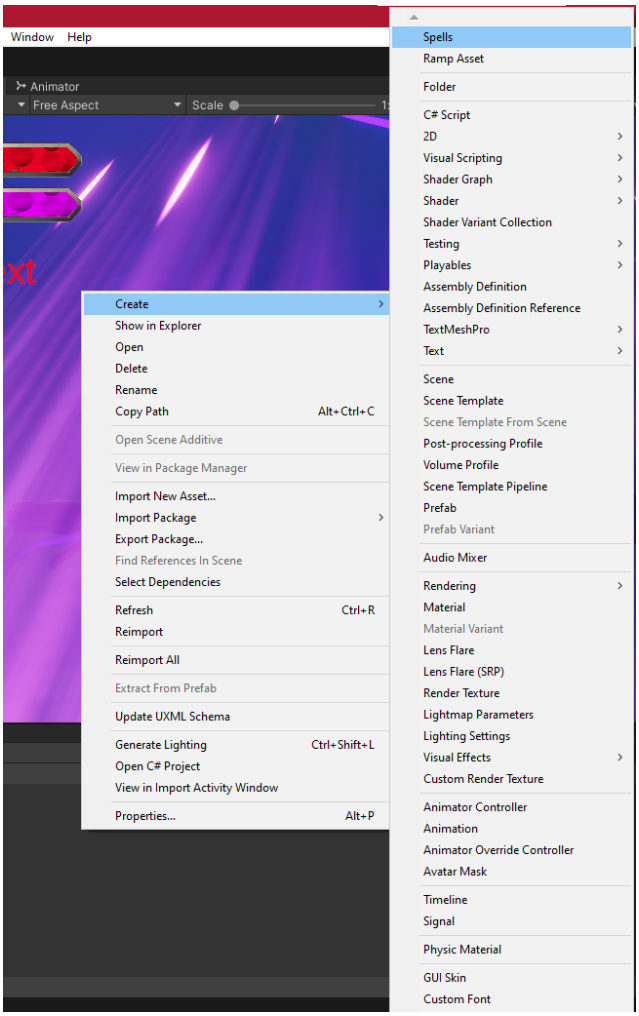
```
private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.CompareTag("Enemy"))
        {
            HealthComponent enemyHealth =
other.gameObject.GetComponent<HealthComponent>();
            enemyHealth.TakeDamage(SpellToCast.DamageAmount);
            GameObject explosion = Instantiate(fireballExplosion,
transform.position, transform.rotation);
            Destroy(explosion, 0.5f);
            Destroy(this.gameObject);
        }
        if (other.gameObject.CompareTag("Player"))
        {
            HealthComponent playerHealth =
other.gameObject.GetComponent<HealthComponent>();
            if (playerHealth != null) {
playerHealth.TakeDamage(SpellToCast.DamageAmount); }
        }
        else if(!other.gameObject.CompareTag("Player") &&
!other.gameObject.GetComponent<CapsuleCollider>())
        {
            Destroy(this.gameObject);
        }
    }
```

U metodi OnTriggerEnter upravlja se interakcijama kada objekt s koliderom udari u drugi kolider. Ako kolider pripada objektu s oznakom "Enemy", dohvaća se komponenta

HealthComponent tog neprijatelja i poziva metodu TakeDamage s količinom štete koju čarolija uzrokuje, definiranu sa varijablom DamageAmount. Također, stvara se eksplozija fireballExplosion na poziciji čarolije i uništava se nakon 0,5 sekundi, dok se sama čarolija zatim uništava.

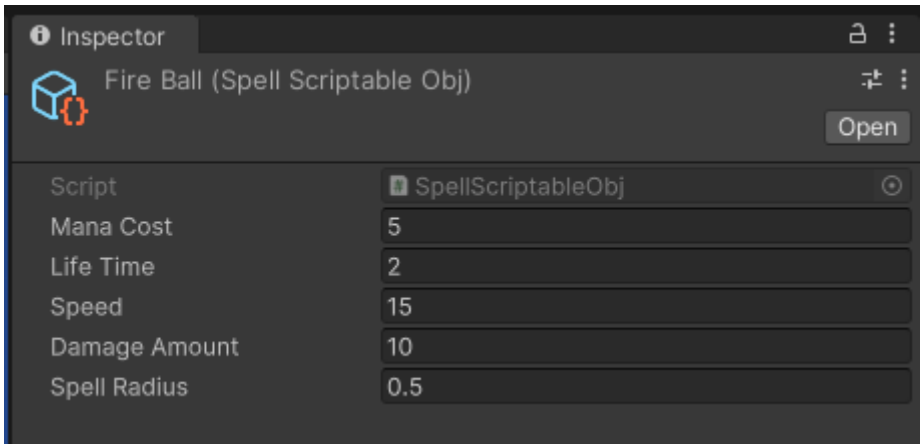
```
[CreateAssetMenu(fileName = "New Spell", menuName = "Spells")]
public class SpellScriptableObj : ScriptableObject
{
    public float ManaCost = 5f;
    public float LifeTime = 2f;
    public float Speed = 15f;
    public float DamageAmount = 10f;
    public float SpellRadius = 0.5f;
}
```

Scriptable Object je u Unityju defeniran pod nazivom SpellScriptableObj. Atribut [CreateAssetMenu] omogućava da se nova instanca ove klase kreira izravno iz Unity Editor-a putem opcije "Create" u izborniku. fileName = "New Spell" postavlja zadano ime za novi asset kao "New Spell", dok menuName = "Spells" dodaje ovu opciju u izbornik za kreiranje asseta, omogućujući korisnicima da lako pronađu i kreiraju nove instance ove klase.



Slika 18. Prikaz kreacije nove moći

Unutar same klase SpellScriptableObj, definirano je nekoliko varijabli koje opisuju karakteristike čarolije. ManaCost označava količinu energije potrebnu za aktiviranje čarolije. LifeTime predstavlja vrijeme tijekom kojeg čarolija ostaje aktivna. Speed definira brzinu kojom čarolija putuje. DamageAmount označava koliko štete čarolija nanosi meti, dok SpellRadius označava radijus u kojem čarolija ima učinak. Ove varijable omogućuju lako podešavanje i konfiguriranje različitih čarolija iz Unity Editor-a bez potrebe za promjenama u kodu.

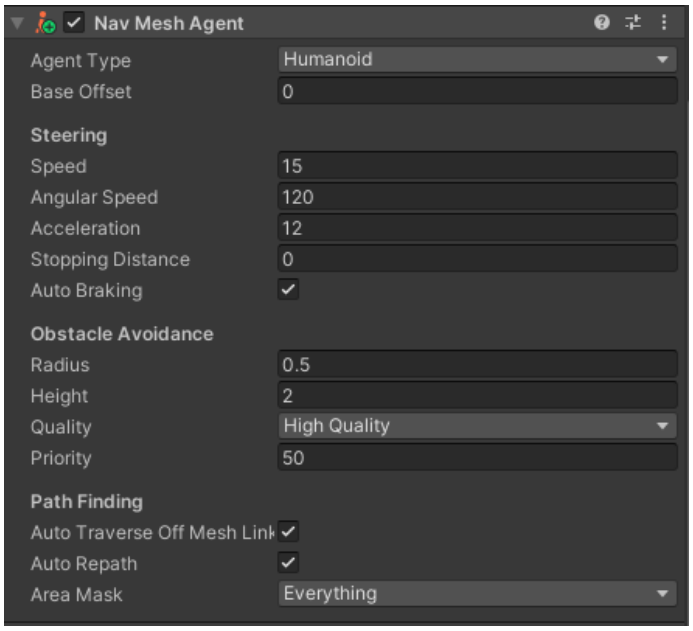


Slika 19. Prikaz vatrene lopte u Unity editoru

3.2.7. NavMeshAgent

Unity-ev NavMeshAgent je komponenta koja omogućuje likovima u igri da se kreću po terenu koristeći navigacijsku mrežu (NavMesh). Ova mreža se unaprijed generira na osnovu terena igre, omogućavajući agentima da se automatski kreću do cilja izbjegavajući prepreke na putu. NavMeshAgent koristi informacije iz NavMesha kako bi pronašao najkraći put između svoje trenutne pozicije i ciljne točke. [9]

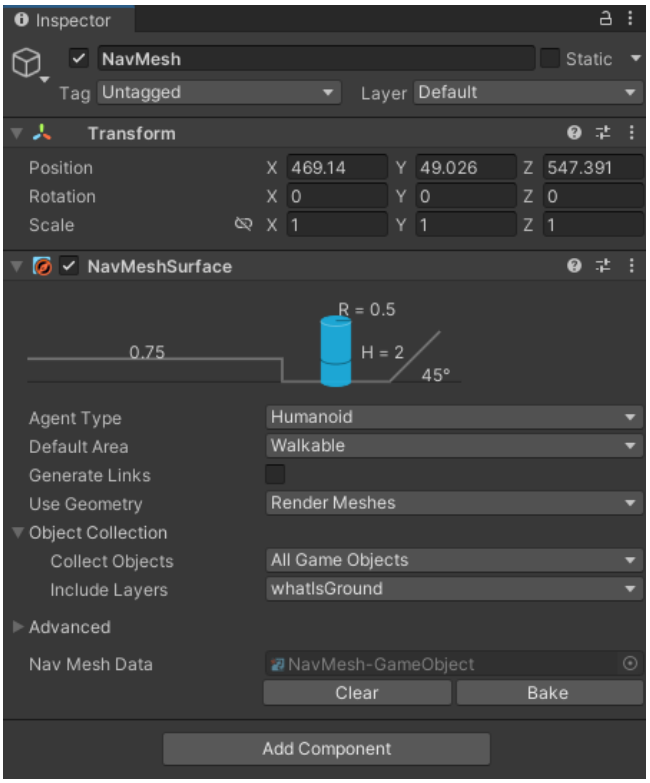
Da bi se koristio NavMeshAgent, prvo se mora pripremiti teren tako da se generira (bake-a) NavMesh. To se postiže označavanjem objekata u sceni kao "Navigation Static", što označava da se ti objekti neće mijenjati tijekom igre. Nakon toga, u Unityju se otvara prozor za navigaciju, gdje se može generirati NavMesh za cijeli teren. Nakon što je navigacijska mreža generirana, NavMeshAgent se može dodati na bilo koji objekt u igri.[10]



Slika 20. Prikaz komponente NavMeshAgent

NavMesh Surface je komponenta u Unityju koja omogućuje stvaranje navigacijske mreže (NavMesh) na specifičnim površinama u sceni. Dok NavMeshAgent koristi već generiranu navigacijsku mrežu za kretanje likova, NavMesh Surface definira koje će površine u sceni biti uključene u tu mrežu. [11]

Ova komponenta omogućuje veću fleksibilnost pri generiranju navigacijske mreže jer se može dodati na bilo koji objekt u sceni, bez obzira na njegov oblik ili položaj. To znači da se može koristiti za stvaranje navigacijske mreže na nagnutim ili zakrivljenim površinama, na različitim visinama, pa čak i na pokretnim platformama.



Slika 21. Prikaz komponente za navigacijsku mrežu

3.2.8. Mehanika neprijatelja

Skripta EnemyAi definira osnovno ponašanje neprijatelja u videoigri koristeći Unity i NavMeshAgent za navigaciju i AI. NavMeshAgent omogućuje neprijatelju da se kreće po unaprijed definiranom terenu. Varijable poput targetPlayer i playerLayer koriste se za prepoznavanje igrača i postavljanje cilja.

Neprijatelj u igri može patrolirati, loviti igrača i napadati ga. Kada neprijatelj primijeti igrača unutar vidnog dometa detectionRange, prelazi na lovljenje igrača koristeći metodu Chase, koja postavlja igračevu poziciju kao cilj. Kada je igrač unutar dometa za napad attackRadius, neprijatelj se zaustavlja i koristi metodu Attack za ispaljivanje projektila prema igraču. Projektil se instancira na poziciji projectileSpawnPoint i pokreće se pomoću sile.

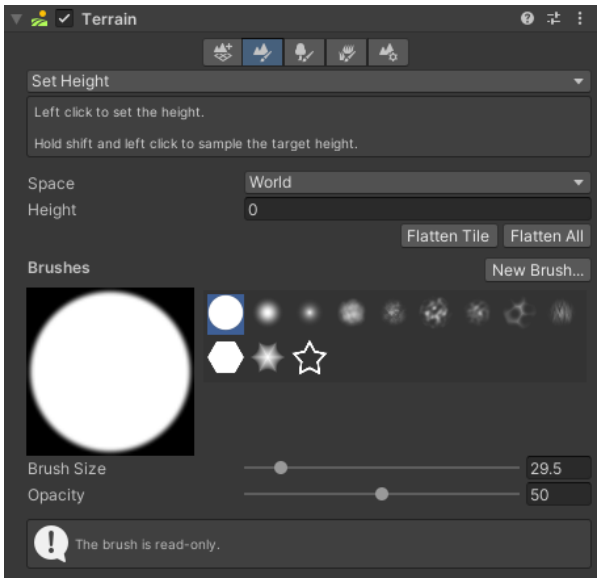
Osigurano je da neprijatelj ne napada kontinuirano postavljanjem varijable hasAttacked, koja se resetira nakon definiranog vremena između napada pomoću metode ResetAttackCooldown. Također, neprijatelj stalno provjerava stanje igrača i odgovarajuće reagira, bilo da patrolira, lovi ili napada, ovisno o situaciji.

```
public NavMeshAgent navMeshAgent;
public Transform targetPlayer;
public LayerMask groundLayer, playerLayer;
public float enemyHealth;
public float attackCooldown;
bool hasAttacked;
```

```
public GameObject attackProjectile;
[SerializeField] private Transform projectileSpawnPoint;
public float detectionRange, attackRadius;
public bool isPlayerInDetectionRange, isPlayerInAttackRadius;
private void Awake()
{
    targetPlayer = GameObject.Find("DefaultAvatar").transform;
    navMeshAgent = GetComponent<NavMeshAgent>();
}
private void Update()
{
    UpdatePlayerDetection();

    else if (isPlayerInDetectionRange && !isPlayerInAttackRadius)
    {
        Chase();
    }
    else if (isPlayerInAttackRadius && isPlayerInDetectionRange)
    {
        Attack();
    }
}
private void UpdatePlayerDetection()
{
    isPlayerInDetectionRange = Physics.CheckSphere(transform.position,
detectionRange, playerLayer);
    isPlayerInAttackRadius = Physics.CheckSphere(transform.position,
attackRadius, playerLayer);
}
private void Chase()
{
    navMeshAgent.SetDestination(targetPlayer.position);
}
private void Attack()
{
    navMeshAgent.SetDestination(transform.position);
    transform.LookAt(targetPlayer);

    if (!hasAttacked)
    {
        LaunchProjectile();
        hasAttacked = true;
        Invoke(nameof(ResetAttackCooldown), attackCooldown);
    }
}
private void LaunchProjectile()
{
    Rigidbody rb = Instantiate(attackProjectile,
projectileSpawnPoint.position,
projectileSpawnPoint.rotation).GetComponent<Rigidbody>();
    rb.AddForce(transform.forward * 32f, ForceMode.Impulse);
    rb.AddForce(transform.up * 8f, ForceMode.Impulse);
}

private void ResetAttackCooldown()
{
    hasAttacked = false;
}
```

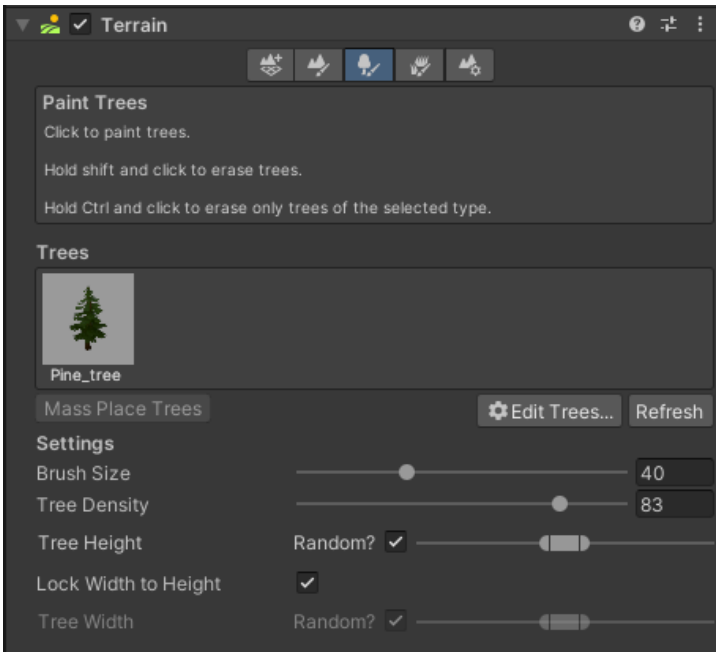
3.2.9. Kreiranje terena

Kreiranje terena u Unityju započelo je korištenjem "Paint Terrain" opcije unutar Terrain Toolseta – alata za oblikovanje terena. Ova opcija omogućila je oblikovanje terena prema željenim potrebama igre. U početku, koristio se alat za podizanje i spuštanje visine terena, čime su se stvorili brežuljci, doline i drugi elementi.



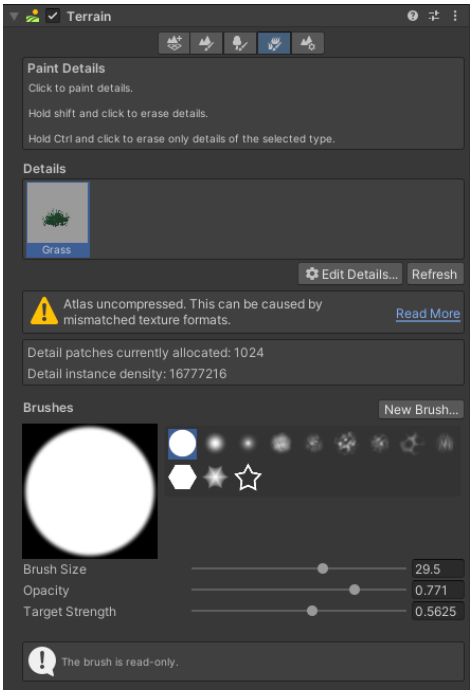
Slika 22. Prikaz „Paint Terrain“ opcije unutar alata za oblikovanje terena

Zatim, koristila se "Paint Trees" – kreiranje drva - opcija, gdje su na terenu postavljena stabla, dodajući prirodni element sceni. Drveće je pažljivo raspoređeno kako bi se postigao izgled šume ili raspršenih stabala, ovisno o potrebi.



Slika 23. Prikaz „Paint Trees“ opcije unutar alata za oblikovanje terena

Za dodatnu vegetaciju, korišten je "Paint Details" – kreiranje detalja - alat koji je omogućio dodavanje trave i drugih manjih detalja na teren. Kombinacijom ovih alata, teren je oblikovan i uređen na način koji odgovara tematici i vizualnom stilu igre.



Slika 24. Prikaz „Paint Details“ opcije unutar alata za oblikovanje terena

3.2.10. Prikazivanje zdravlja igrača

Prikaz zdravlja igrača na korisničkom sučelju ključan je element u većini videoigara jer pruža igraču jasnu povratnu informaciju o stanju njihovog lika.

```
public Image healthBar;
    public TextMeshProUGUI healthText;
    [SerializeField] private GameObject igrac;
    void Start()
    {
        healthText.text = " " +
igrac.GetComponent<HealthComponent>().currentHealth;
    }
    void Update()
    {
        if(igrac == null)
        {
            SceneManager.LoadScene("YouDied");
            return
        }
        if (igrac.GetComponent<HealthComponent>().currentHealth !=
igrac.GetComponent<HealthComponent>().maxHealth)
        {
            healthBar.fillAmount =
igrac.GetComponent<HealthComponent>().currentHealth /
igrac.GetComponent<HealthComponent>().maxHealth;
            healthText.text = "" +
Mathf.RoundToInt(igrac.GetComponent<HealthComponent>().currentHealth);
```

```
        }
    }
}
```

Varijabla healthBar koristi se za vizualni prikaz zdravlja igrača putem UI Image elementa, gdje se postavlja stupac koji se puni ovisno o preostalom zdravlju. healthText je varijabla koja prikazuje trenutnu vrijednost zdravlja igrača kao tekst koristeći TextMeshProUGUI element. U Start metodi, tekstualni prikaz zdravlja odmah se postavlja na početnu vrijednost zdravlja igrača, preuzetu iz komponente HealthComponent.

Unutar Update metode, prvo se provjerava je li igrač (odnosno GameObject na koji se referencira varijabla igrač) uništen ili ne postoji. Ako je, scena se mijenja na scenu "YouDied". Dalje, provjerava se je li trenutačno zdravlje igrača različito od maksimalnog zdravlja, što znači da je igrač primio štetu. U tom slučaju, ažurira se prikaz zdravlja i tekst kako bi prikazivali novo stanje zdravlja.

Slika 25. Prikaz životnih bodova igrača

3.2.11. Prikazivanje energije igrača

Prikaz mane je ključan element u mnogim igrama koje koriste sustav magije ili posebnih sposobnosti, jer omogućuje igračima da u svakom trenutku prate dostupnost svojih resursa za izvođenje tih sposobnosti. Mana predstavlja energiju ili resurs koji se troši pri upotrebi magija, vještina ili specijalnih akcija, pa je jasno i ažurno prikazivanje tog resursa bitno za učinkovito igranje.

```
public Image manaBar;
    public TextMeshProUGUI manaText;
    [SerializeField] private GameObject igrac;
    void Start()
    {
        manaText.text = " " +
igrac.GetComponent<SpellsControll>().currentMana;
    }
    void Update()
    {
        if(igrac == null)
        { return; }
        if (igrac.GetComponent<SpellsControll>().currentMana !=
igrac.GetComponent<SpellsControll>().maxMana)
        {
            manaBar.fillAmount =
igrac.GetComponent<SpellsControll>().currentMana /
igrac.GetComponent<SpellsControll>().maxMana;
```

```
            manaText.text = "" +
Mathf.RoundToInt(igrac.GetComponent<SpellsControll>().currentMana);
        }
    }
```

Ova skripta upravlja prikazom energije (mana) na korisničkom sučelju igre, omogućujući igraču da prati koliko im je energije preostalo. Varijabla manaBar koristi se za prikaz vizualne trake koja pokazuje trenutnu razinu energije igrača, dok manaText prikazuje točnu brojčanu vrijednost preostale energije.

U Start metodi, tekstualni prikaz (manaText) postavlja se na početnu vrijednost energije preuzetu iz komponente SpellsControll koja je pridružena igraču (objektu označenom kao igrač). Ova inicijalizacija omogućuje prikaz trenutnog stanja energije od početka igre.

U Update metodi, provjerava se da li igrač još uvijek postoji (tj. da igrac nije null). Ako je igrač postoji, provjerava se je li trenutačna količina energije različita od maksimalne. Ako jest, ažurira se prikaz trake energije tako da odgovara omjeru trenutačne energije prema maksimalnoj, te se tekst ažurira da prikaže brojčanu vrijednost trenutne energije.

Slika 26. Prikaz energije igrača

3.2.12. Regeneracija zdravlja igrača

Imati funkciju obnove životnih bodova u igri je ključno za poboljšanje igre i igračkog iskustva. Ova funkcija omogućuje igračima da se oporave i produže trajanje igre, što dodaje element strategije i preživljavanja. Obnova zdravlja pruža igračima priliku da se regeneriraju tijekom igre, posebno nakon što su pretrpjeli štetu, te im pomaže da se pripreme za daljnje izazove ili borbe.

```
public float healingAmount = 5f;
    public GameObject healingEffect;
    private GameObject currentHealingEffect;
    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player") && currentHealingEffect == null)
        {
            currentHealingEffect = Instantiate(healingEffect,
transform.position, Quaternion.identity);
        }
```

```
    }
    private void OnTriggerStay(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            HealthComponent playerHealth =
other.GetComponent<HealthComponent>();

            if (playerHealth != null)
            {
                playerHealth.Heal(healingAmount * Time.deltaTime);
            }
        }
```

Varijabla healingAmount definira koliko će se zdravlja obnavljati svaki sekundi, dok healingEffect predstavlja vizualni efekt koji se prikazuje kada igrač dođe u kontakt s područjem obnove zdravlja. Kada igrač (objekt s tagom "Player") uđe u područje obnove zdravlja, u funkciji OnTriggerEnter se provjerava je li trenutno prisutan vizualni efekt obnove zdravlja (currentHealingEffect). Ako nije, kreira se novi efekt koristeći Instantiate metodu, smješten u poziciju trenutnog objekta.

U funkciji OnTriggerStay, koja se aktivira dok igrač ostaje unutar područja obnove, provjerava se je li igrač još uvijek u kontaktu i ima li komponente HealthComponent. Ako sve to odgovara, Heal metoda komponente HealthComponent se poziva, povećavajući igraču zdravlje prema healingAmount množenom s Time.deltaTime za postizanje kontinuiranog obnavljanja zdravlja dok igrač ostaje u području.

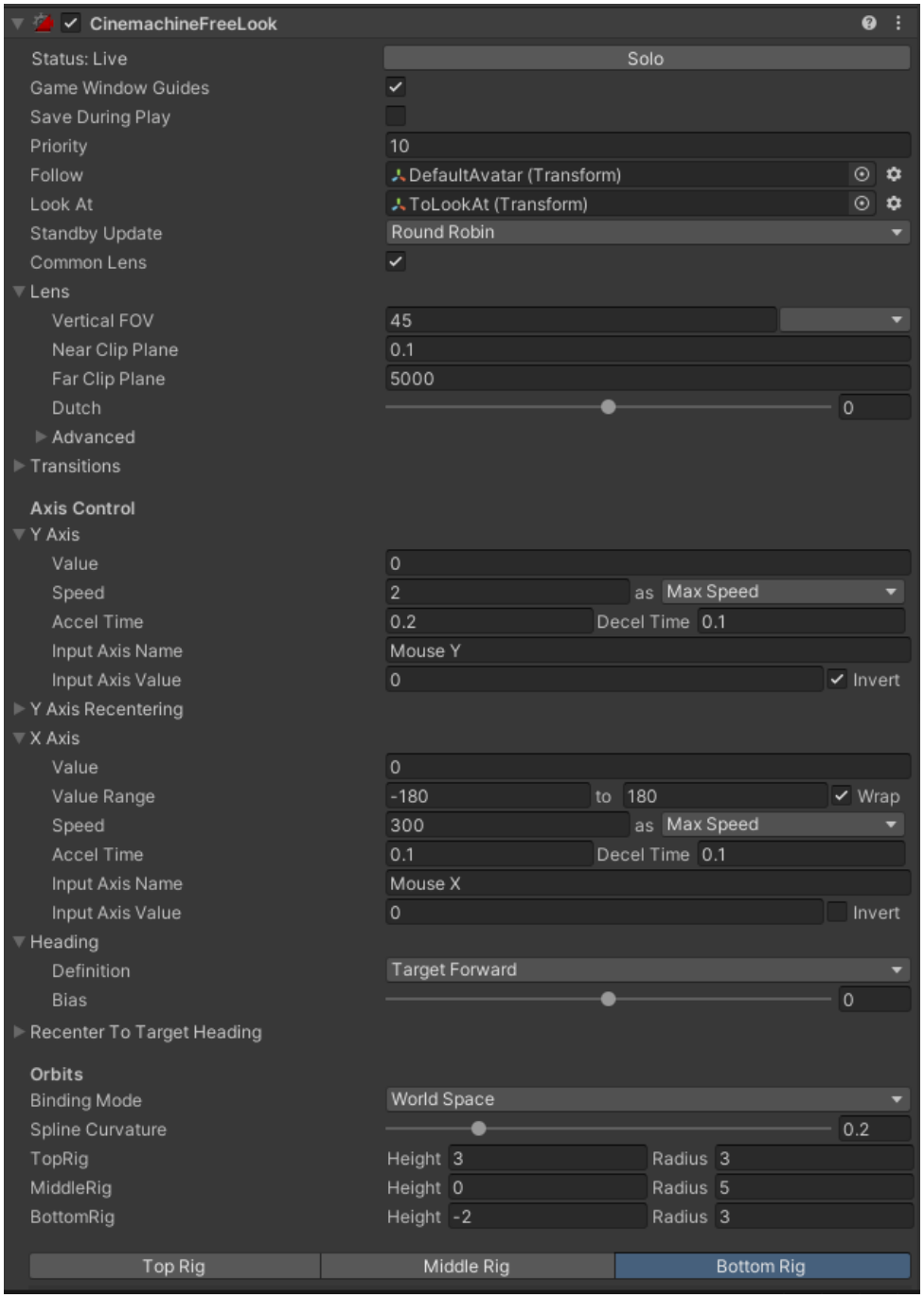
3.2.13. Cinemachine

Asset Cinemachine je moćan alat u Unityju koji omogućuje jednostavno i precizno upravljanje kamerama unutar igre. Razvijen s ciljem da poboljša način na koji se kamera koristi u igrama, Cinemachine pruža širok spektar značajki za postizanje profesionalno usmjerenih kamera i dinamičnih prikaza bez potrebe za kompleksnim kodiranjem. Koristeći Cinemachine, programeri mogu lako kreirati različite stilove kamera, od jednostavnih fiksnih kutova do sofisticiranih pokretnih kamera koje prate igračeve akcije. [12]

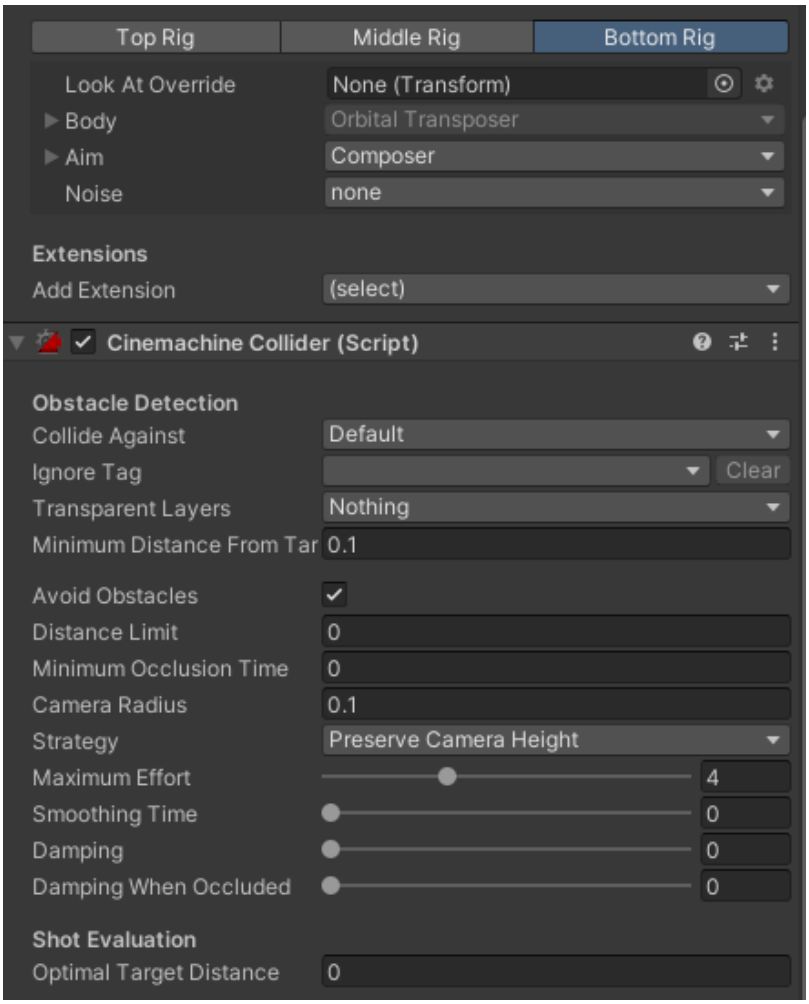
Jedan od ključnih elemenata Cinemachinea je Freelook kamera. Ova vrsta kamere omogućuje stvaranje složenih i fleksibilnih prikaza koje se mogu prilagoditi specifičnim potrebama igre. Freelook kamera nudi tri glavne zone promatranja: Top, Middle i Bottom. Svaka zona može biti konfigurirana s različitim postavkama, što omogućuje kamerama da se kreću oko igrača u različitim smjerovima i kutovima, pružajući dinamičan i vizualno bogat prikaz igre.

Kada se koristi Freelook kamera u Unityju, programer može iskoristiti unaprijed postavljena svojstva i parametre kako bi prilagodio ponašanje kamere prema potrebama igre.

Na primjer, podešavanjem kutova gledanja, udaljenosti i različitih postavki stabilizacije, Freelook kamera može se prilagoditi specifičnom stilu igre ili željenom efektu prikaza. Sve ove postavke mogu se prilagoditi kroz jednostavno sučelje unutar Unity editora, što omogućuje brzu i učinkovitu prilagodbu kamere bez potrebe za ručnim kodiranjem.



Slika 27. Prikaz CinemachineFreeLook postavki kamere



Slika 28. Prikaz Cinemachine postavki collidera

4. Zaključak

Razvoj video igara obuhvaća mnoštvo kompleksnih aspekata koji zahtijevaju posvećenost i stručnost u različitim disciplinama. Proces uključuje razumijevanje kretanja igrača, prikaza resursa poput zdravlja i mane(energije), te upravljanja AI ponašanjem, sve kako bi se stvorio konzistentan i uzbudljiv doživljaj za igrače. Implementacija ovih elemenata u Unityju pokazuje koliko je važno imati jasnu viziju i razumijevanje svih komponenti igre. Korištenje komponenti kao što su NavMeshAgent za kretanje AI neprijatljea, ScriptableObject za pohranu podataka o čarolijama i vizualizacija resursa kao što su zdravlje i energija, omogućava detaljnu kontrolu i prilagodbu.

Razumijevanje i implementacija različitih mehanizama u igri može biti izazovno, no donosi velike nagrade u smislu zadovoljstva i uspjeha. Postavljanje NavMeshSurface za navigaciju AI, korištenje ScriptableObject za konfiguraciju sposobnosti, te prikaz energije i zdravlja kroz UI komponente, ključni su za stvaranje dinamičnih i zanimljivih igračkih iskustava. Svaki od ovih aspekata zahtijeva pažljivo planiranje i testiranje kako bi se osigurala besprijekorna funkcionalnost i intuitivno korisničko iskustvo.

Ovi procesi pokazuju da razvoj igre nije samo tehnički izazov, već i kreativno putovanje koje zahtijeva kombinaciju znanja, vještina i upornosti. Iako razvoj može biti složen i zahtjevan, uspješno implementiranje i optimiziranje svih tih elemenata vodi k stvaranju visokokvalitetne igre koja može pružiti igračima nevjerojatno iskustvo. Istraživanje i učenje kako koristiti dostupne alate i komponente na najbolji način omogućuje programerima da stvaraju inovativne i zabavne igre, čak i s ograničenim resursima.

Na kraju, jasno je da današnji alati i resursi omogućuju razvoj složenih igara s visokom razinom detalja i funkcionalnosti, uz minimalne troškove. Razumijevanje kako primijeniti različite komponente i tehnologije može značajno poboljšati kvalitetu igre i omogućiti stvaranje profesionalnog i zabavnog proizvoda. Ovaj projekt ističe važnost predanosti i kreativnosti u razvoju igara te pruža snažan temelj za buduće projekte i istraživanje novih ideja u svijetu video igara.

5. Popis literature

[1] *What is unity?* (2023) *PubNub*. Dostupno na: https://www.pubnub.com/guides/unity/ (Pristupano: 30.8.2024).

[2] Fakher, H. (2024) *Understanding unity rendering pipelines: Built-in vs. Universal Render Pipeline (URP)*, *Medium*. Dostupno na: https://medium.com/@fakherhusayn/understanding-unity-rendering-pipelines-built-in-vs-universal-render-pipeline-urp-4a9750d892be (Pristupano: 30.8.2024).

[3] *Universal render pipeline overview: Universal rp: 17.0.3* (2024) *Universal RP | 17.0.3*. Dostupno na: https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@17.0/manual/index.html (Pristupano: 01.9.2024).

[4] *High definition render pipeline overview: High definition RP: 17.0.3* (2024) *High Definition RP | 17.0.3*. Dostupno na: https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@17.0/manual/index.html (Pristupano: 30.8.2024.).

[5] (2024) *Mixamo*. Dostupno na: https://www.mixamo.com/#/ (Pristupano: 30.8.2024.).

[6] Technologies, U. (2024.) *Character controller component reference*, *Unity*. Dostupno na: https://docs.unity3d.com/Manual/class-CharacterController.html (Pristupano: 30.8.2024.).

[7] Technologies, U. (2024) *Rigidbody*, *Unity*. Dostupno na: https://docs.unity3d.com/560/Documentation/Manual/class-Rigidbody.html (Pristupano: 30.8.2024.).

[8] Technologies, U. (2024.) *ScriptableObject*, *Unity*. Dostupno na: https://docs.unity3d.com/Manual/class-ScriptableObject.html (Pristupano: 30.8.2024.).

[9] Technologies, U. (2024.) *Navmesh agent*, *Unity*. Dostupno na: https://docs.unity3d.com/560/Documentation/Manual/class-NavMeshAgent.html (Pristupano: 30.8.2024.).

[10] *Working with Navmesh Agents* (2024.) *Unity Learn*. Dostupno na: https://learn.unity.com/tutorial/working-with-navmesh-agents# (Pristupano: 30.8.2024.).

[11] Technologies, U. (2024.) *Navmesh surface*, *Unity*. Dostupno na: https://docs.unity3d.com/560/Documentation/Manual/class-NavMeshSurface.html (Pristupano: 30.8.2024.).

[12] *Cinemachine* (2024.) *Unity*. Dostupno na: https://unity.com/unity/features/editor/art-and-design/cinemachine (Pristupano: 30.8.2024.).

6. Popis slika

Slika 1. Unity sučelje 3
Slika 2. Unity Hub – izrada novog projekta 4
Slika 3. Unity hub – odabir predloška projekta 6
Slika 4. Početni zaslon videoigre 7
Slika 5. Prikaz kraja igre - poraz 7
Slika 6. Prikaz korištenja vaternih kugli 8
Slika 7. Prikaz obnavljanja energije igrača 8
Slika 8. Prikaz obnavljanja životnih bodova 9
Slika 9. Prikaz korištenja posebne moći 9
Slika 10. Prikaz kraja igre – pobjeda 10
Slika 11. Prikaz mixamo sučelja za odabir animacija 11
Slika 12. Prikaz animatora 12
Slika 13. Prikaz parametra za kretanje igrača 15
Slika 14. Prikaz parametara za rotaciju kamere i igrača 16
Slika 15. Prikaz Health Componente u Unity Inspektoru 17
Slika 16. Prikaz oznake „player“ na igraču 18
Slika 17. Prikaz oznake „Enemy“ na neprijatelju 18
Slika 18. Prikaz kreacije nove moći 23
Slika 19. Prikaz vatrene lopte u Unity editoru 24
Slika 20. Prikaz komponente NavMeshAgent 25
Slika 21. Prikaz komponente za navigacijsku mrežu 26
Slika 22. Prikaz „Paint Terrain“ opcije unutar alata za oblikovanje terena 28
Slika 23. Prikaz „Paint Trees“ opcije unutar alata za oblikovanje terena 28
Slika 24. Prikaz „Paint Details“ opcije unutar alata za oblikovanje terena 29
Slika 25. Prikaz životnih bodova igrača 30
Slika 26. Prikaz energije igrača 31
Slika 27. Prikaz CinemachineFreeLook postavki kamere 33
Slika 28. Prikaz Cinemachine postavki collidera 34