SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij Računarstvo

MARIN ŠOP

Z A V R Š N I R A D

IZRADA ZOMBIE SURVIAL IGRE U PROGRAMSKOM ALATU UNITY

Split, rujan 2023.

SVEUČILIŠTE U SPLITU
SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij Računarstvo

Predmet: Izrada igre u programskom alatu Unity

Z A V R Š N I R A D

Kandidat: Marin Šop

Naslov rada: Izrada Zombie survival igre u programskom alatu Unity

Mentor: viši predavač dipl. ing. Ljiljana Despalatović

Split, rujan 2023.

SADRŽAJ

Sažetak

Ovaj završni rad bavi se izradom Zombie survival igre u programskom alatu Unity. Cilj projekta je bio stvoriti zabavno i izazovno iskustvo za igrače, gdje će se suočiti s valovima napada zombija gdje moraju preživjeti i doći do kraja nivoa.

U izradi igre korištena je platforma Unity, koja pruža snažne alate za razvoj videoigara. Kroz projektiranje i implementaciju različitih komponenti igre, kao što su likovi, oružja, okruženje i umjetna inteligencija zombija, ostvarena je interaktivna i uvjerljiva igraća atmosfera. Kroz provedeni projekt, dobiveno je dublje razumijevanje procesa izrade videoigara i stjecanje praktičnih vještina u radu s programskim alatom Unity.

Ključne riječi: likovi, oružja, programski alat, Unity, Zombie survival igra.

Summary

Creating a Zombie survival game in the Unity programming tool

This final thesis focuses on the development of a Zombie survival game using the Unity game engine. The aim of the project was to create an entertaining and challenging experience for players, where they would face waves of zombie attacks and strive to survive and get to the end of the level.

The Unity platform was utilized in the game development process, as it provides powerful tools for creating video games. Through the design and implementation of various game components such as characters, weapons, environments, and zombie artificial intelligence, an interactive and immersive gameplay atmosphere was achieved.

Through the completion of this project, a deeper understanding of the game development process was acquired, along with practical skills in working with the Unity game engine. My game provides challenge and enjoyment for players, representing a successful outcome of this final thesis.

Keywords: characters, game engine, Unity, weapons, Zombie survival game.

1. Uvod

Izrada videoigara je postala popularna industrija i sve pristupačnija zahvaljujući napretku tehnologije i razvoju programskih alata. Unity, kao jedan od najpopularnijih programskih alata za razvoj videoigara, omogućuje programerima da stvore visokokvalitetne igre koje privlače široku publiku. Cilj ovog završnog rada je bio upravo istražiti i demonstrirati proces izrade videoigre.

Zombie survival igre su vrlo popularan žanr među igračima diljem svijeta. One kombiniraju elemente akcije, preživljavanja i strategije, te postavljaju igrače u postapokaliptično okruženje pune zombija. Cilj igre je preživjeti što je dulje moguće, koristeći razne resurse i oružja kako bi se oduprli napadima zombija. Ovaj žanr pruža intenzivno iskustvo i izazov za igrače, te ima široku bazu obožavatelja diljem svijeta.

Kroz razvojni proces, koriste se razni alati i tehnike kako bi se stvorila uvjerljiva i zanimljiva igra. Pored toga, istražiti će se popularni elementi zombie survival igara kako bi se osigurala autentičnost i privlačnost igre za ciljnu publiku. Elementi kao što su umjetna inteligencija zombija, okruženje, oružja i likovi biti će implementirani u igru.

Kroz ovaj projekt, fokus je na stjecanju dubljeg razumijevanja procesa razvoja videoigara te na razvoju praktičnih vještina rada s platformom Unity. U nastavku ovog teksta, detaljno će biti opisan proces izrade igre, koraci koji su poduzeti, kao i potencijalni izazovi s kojima se može susresti tijekom razvoja. Također će se pružiti pregled glavnih komponenti igre, njihovih funkcionalnost i načina njihove implementacije.

Rad se sastoji od tri poglavlja: opis igre, izrada igre i zaključka. Prvo poglavlje bavi se detaljnim opisom igre. Obuhvaća prezentaciju alata korištenog za njezinu izradu, kao i analizu žanra kojem pripada. U drugom poglavlju, detaljno se razmatra cijeli proces kreiranja igre. Ovdje se opisuju metode koje su primijenjene kako bi se igra stvorila, te se istražuju izazovi koji su se pojavili tokom tog procesa i načini na koje su riješeni. U zaključnom trećem poglavlju bit će iznesen zaključak koji će obuhvatiti ključne spoznaje koje su se pojavile tijekom čitavog teksta.

2. Opis igre

Igra je izrađena u programskom alatu Unity. Ovaj popularan razvojni okvir pruža sveobuhvatne mogućnosti za razvoj igara na različitim platformama. Kroz Unity, moguće je kreirati bogate 3D svjetove, implementirati kompleksne mehanike igre i optimizirati performanse.

Igra spada u žanr akcije i preživljavanja. Ovaj žanr ističe izazov preživljavanja u neprijateljskom okruženju i obično zahtijeva brze odluke i taktičke vještine. Ova igra također sadrži elemente avanture, istraživanja i horora, što dodatno doprinosi dubini iskustva.

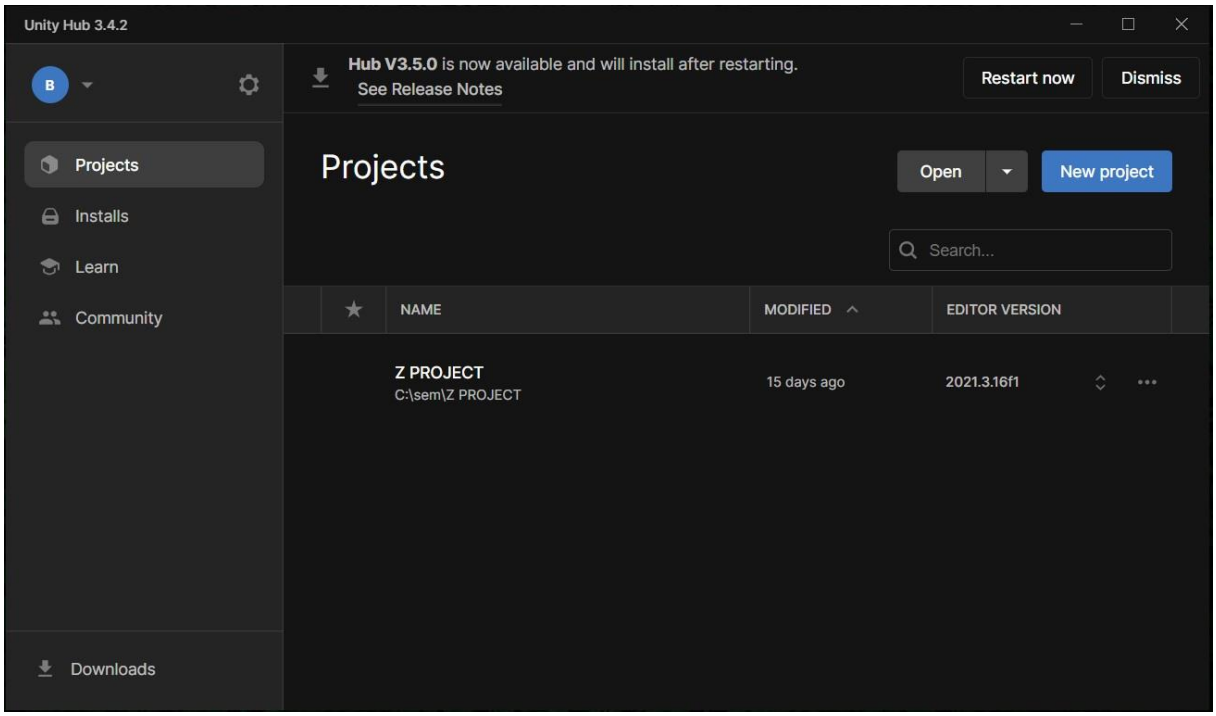
Igra je napeta i izazovna. Ova igra smješta igrača u postapokaliptični svijet zaražen zombijima, gdje igrači moraju preživjeti u okruženju punom opasnosti i nepredvidivih situacija. Kao igrači, suočeni smo s konstantnim napetostima i potrebom za brzim donošenjem odluka kako bismo preživjeli. Cilj igre je steći prolaz putem akumuliranja bodova, ostvarenih eliminiranjem zombija, kako biste naposljetku osigurali prelazak kroz posljednju prepreku koja označava kraj nivoa. Što se nivo dalje razvija, otvara se više pravaca iz kojih zombiji mogu napasti, dodajući dodatni izazov i dinamiku igri. Kako igrač napreduje, suočava se s rastućom zahtjevnošću dok balansira između obrane od zombija i stjecanja dovoljno bodova za prelazak nivoa. Na početku igre, susreće se samo s osnovnim tipovima zombija. No, kako igra napreduje, nakon nekoliko rundi, počinju se pojavljivati zombiji koji su brži i otporniji. Osim toga, svaka sljedeća runda donosi veći broj zombija, postupno povećavajući izazov i intenzitet igre. Nakon uspješnog dovršavanja jednog nivoa, otvara se pristup prelasku na sljedeći nivo igre.

3. Izrada igre

3.1. Kreiranje novog projekta

U ovom poglavlju opisati će se proces kreiranja novog projekta u programskom alatu Unity. Unity je rastavljen na dva dijela, a to su Unity Hub i Unity Editor. Kreiranje projekta prvi je korak u razvoju igre i osigurava pravilan početak rada. Za kreiranje novih projekata te za ažuriranje Unity Editora koristi se aplikacija Unity Hub, a Unity editor je glavni program s kojim se razvija igra. Glavni prozor Unity Hub pokazuje sve trenutne projekte. Sa lijeve strane prozora nalazi se navigacijski dio.

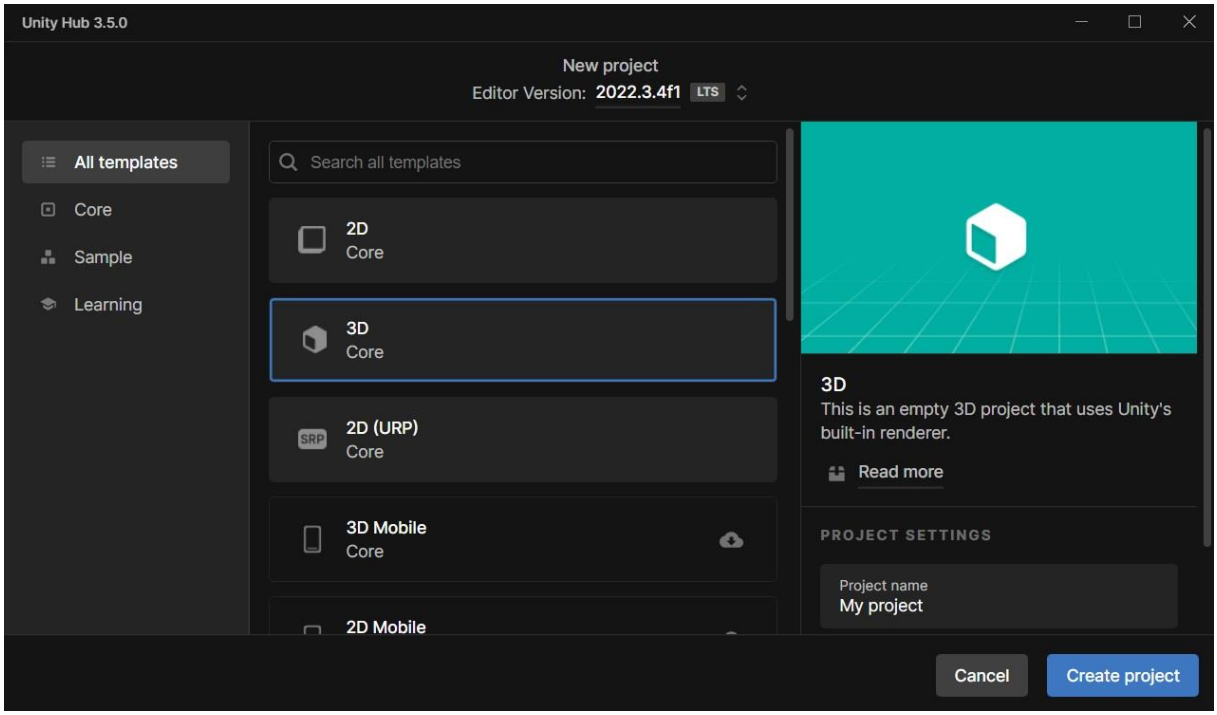
Installs je dio gdje se preuzimaju i ažuriraju verzije Unity Editora. Može biti instalirano više verzija odjednom što omogućuje jednostavnije razvijanje više projekata koji nisu iste verzije Unitya. Learn je dio primarno za početnike. Sadrži razne videozapise koji objašnjavaju kako koristiti program Unity. Community je dio koji vodi do stranica kao Unity trgovina asseta, njihovog službenog foruma itd. Glavni prozor Unity huba se može vidjeti na slici 1.



Slika 1: Unity hub

Kreiranje novog projekta se obavlja tako da korisnik klikne na botun novi projekt (engl. *New project*). Klikom na botun otvoriti će se novi prozor. U novom prozoru ispunjavamo

podatke o projektu kao što je ime projekta, dimenzija projekta i lokacija projekta. Uz podatke se nalazi dio šablona (engl. *Templates*). Šablone mogu znatno ubrzati početak izrade igara jer pola posla se obavi samo jednim klikom. Nikakvi šabloni se neće koristiti u našoj igri. Prozor novi projekt je prikazan na slici 2. Projekt se otvara duplim klikom miša na odgovarajući projekt.



Slika 2: Prozor New project u Unity hub-u

3.2. Osnove Unity Editor-a

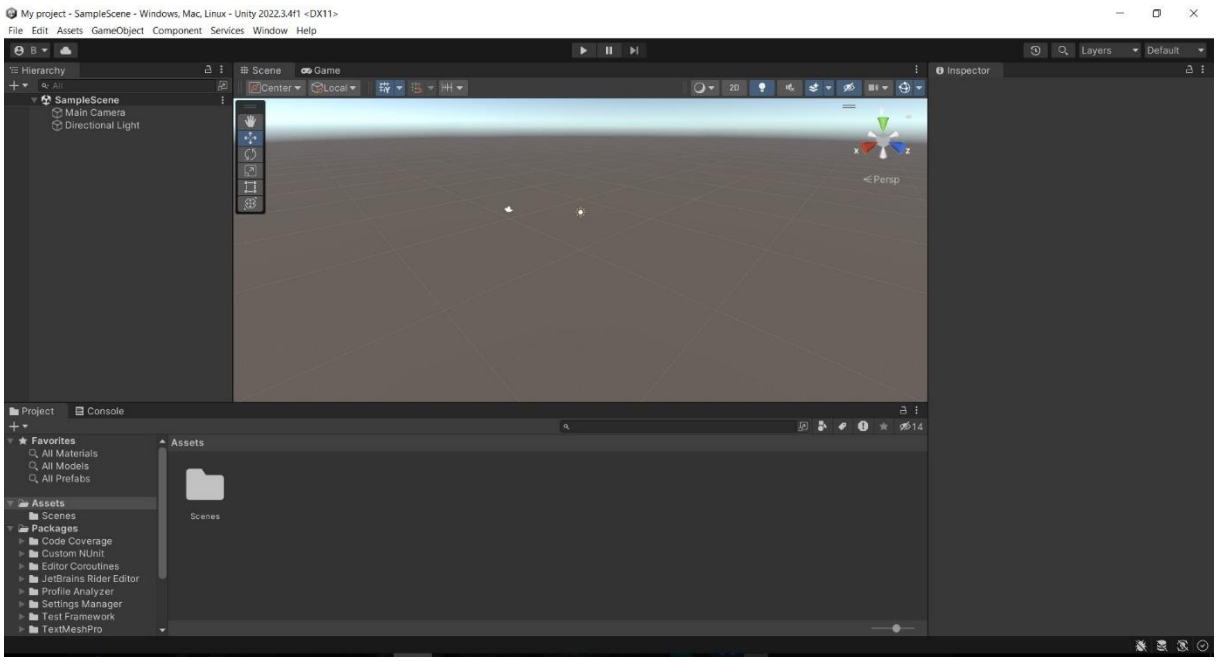
Unity editor se sastoji od više prozora. Ti prozori su hijerarhija (engl. *Hierarchy*), inspektor (engl. *Inspector*), eksplorer (engl. *Explorer*), scena (engl. *Scene*) i igra (engl. *Game*). Hijerarhija je prozor gdje se nalaze svi objekti koji se koriste u igri. Novi objekti se prave pritiskom desnog klika na prozor te odabirom vrste objekta.

Prozor inspektor je interaktivni panel koji vam omogućuje detaljan pregled i manipulaciju svojstvima i komponentama odabranih objekata u projektu. Kada se odabere određeni objekt njegova svojstva se prikazuju u inspektoru. U ovom prozoru se dodaju nove skripte i komponente. Skripta se koristi za definiranje ponašanja objekata, to može uključivati

interakciju s korisnikom, logiku igre, animacije, upravljanje objektima i mnogo više. Komponenta je modularni blok funkcionalnosti koji se može dodijeliti objektu. Svaki objekt može imati različite komponente koje definiraju njegovo ponašanje i svojstva.

Eksplorer je prozor koji ima iste funkcije kao eksplorer u windowsima. Može manipulirati datotekama i mapama projekta. U eksploreru se prave datoteke zvane Prefab. Datoteka prefab je predložak ili šablon objekta koji može biti iskorišten za stvaranje više instanci istog objekta u vašem projektu. Prefab predstavlja unaprijed definirani skup komponenti, svojstava i drugih elemenata objekta koji se mogu koristiti kao temelj za kreiranje identičnih ili sličnih objekata u različitim dijelovima vaše igre.

Prozori scene i game su dva od glavnih prikaza koji vam omogućuju rad na vašem projektu i pregled rezultata u stvarnom vremenu. Prozor scena prikazuje vizualni prikaz vaše igre ili scene u kojoj radite. Ovdje možete kreirati, uređivati i organizirati elemente vaše scene. Možete dodavati objekte, postavljati njihove pozicije, rotacije i skale, mijenjati njihove svojstva, stvarati hijerarhiju objekata i drugo. Prozor igra prikazuje stvarni prikaz vaše igre. Kada pokrenete svoju igru, prozor prikazuje kako će izgledati igračima tijekom igranja. Unity editor se može vidjeti na slici 3.



Slika 3: Unity editor

3.3. Izrada igrača

Nakon što se kreira projekt, prvo se u eksplorer dodaju svi potrebni resursi za izradu igre. To su modeli, zvukovi, slike itd. Igrač se izrađuje tako da se prebaci model igrača u scenu. On tada postaje objekt što nam omogućava da se tim modelom manipulira. Unity automatski dodaje collider na modele što nekad nije potrebno. Collider je komponenta koja definira oblik, veličinu i ponašanje sudara (engl. *collision*) između objekata u sceni. Briše se stari collider te dodajemo kapsulni (engl. *capsule*) collider jer dobro odgovara igraču, a ne treba velika preciznost collidera.

Druga komponenta koja se dodaje na igrača je RigidBody. Rigidbody je komponenta koja se koristi za simuliranje realističnih dinamičkih ponašanja objekata u igri, kao što su gravitacija, kolizije, odbijanje, rotacija i drugi efekti.

3.3.1. Kontroliranje igrača

Igrač se ne bi trebao samostalno rotirati zato se mora zamrznuti rotacija. U komponenti RigidBody se zamrzne rotacija nad svim osima. Skripta se izrađuje tako da se pritisne desni klik na eksplorer te odabere C# Script. Ime skripte može biti proizvoljno pa je važno pametno imenovati sve skripte radi lakše organizacije projekta. Skripta se primjenjuje tako da se prebaci iz eksplorera u inspektor objekta. Pod zadano Unity koristi Visual Studio kao uređivač skripti.

Da bi se neka komponenta koristila u skripti, mora se prvo dohvatiti, to se najčešće radi u funkciji Start koja se pokrene jednom kada se objekt stvori, a funkcija Update se koristi cijelo vrijeme. Funkcija GetComponent omogućava da se dohvati komponenta koja je primijenjena na objektu. Kontrola igrača se ostvaruje tako da se manipulira brzinom (engl. *Velocity*) od komponente RigidBody. Kôd se može vidjeti u ispisu 1.

```
void Start(){
      rb = GetComponent<Rigidbody>();
      playerStats = GetComponent<Player>();
}

private void Update(){
      direction     =     new     Vector3(Input.GetAxis("Horizontal"),     0f,
Input.GetAxis("Vertical")).normalized;

}

void FixedUpdate(){
      Vector3 velocity = direction * playerStats.Speed * Time.deltaTime;
      rb.velocity = velocity;

}
```

Ispis 1: Kôd za kontrolu kretnje

Klasom `Input` se upravlja unosom korisnika, kao što su tipke na tipkovnici, miš, kontroler ili touch ekran. Da bi se `Input` klasa mogla ispravno koristiti, prvo se mora podesiti sav unos u Input Manageru. To je dio u opcijama projekta gdje definiramo što koja tipka radi.

Koristi se `GetAxis` funkcija koja daje vrijednosti u rasponu koji je zadan u opcijama. Obično je od jedan do minus jedan. Tipka w daje vrijednost jedan, a tipka s daje vrijednost minus jedan. Uz pomoć tih vrijednosti instacira se novi vektor koji predstavlja smjer. Taj vektor se pomnoži sa brzinom i deltaTime. `Time.deltaTime` je vrijednost koja predstavlja proteklo vrijeme u sekundama između dvije uzastopne slike (engl. frame). To osigurava u slučaju ako igri padnu sličice po sekundi (engl. frames per second), igrač će dalje hodati istom brzinom.

3.3.2. Rotiranje igrača

Rotaciju igrača se mora dva puta implementirati. Prvi put za Windows, a drugi za platformu Android. Implementacija za Windows se može vidjeti na ispisu 2, a implementacija za Android na ispisu 3.

```
if (Input.GetButton("Fire1")){
      Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
      if (Physics.Raycast(ray, out RaycastHit hit, Mathf.Infinity,floorMask)){
            Vector3 lookDirection = hit.point - transform.position;
      float rotation = Mathf.Atan2(lookDirection.x, lookDirection.z) *
Mathf.Rad2Deg;
            transform.rotation = Quaternion.Euler(0, rotation, 0);
            }
}
```

Ispis 2: Implementacija rotacije na Windows

Unity ima funkciju `ScreenPointToRay` koja puca zraku (engl. *Ray*) koja ide od pozicije kamere prema točki na zaslonu, prolazeći kroz taj piksel na ekranu. Ta zraka se često koristi za detekciju kolizija, interakciju s objektima ili pucanje u igri. Pozicija zrake se oduzima sa pozicijom igrača što daje smjer prema kojem treba okrenuti igrača.

```
private void Update(){
      shootDirection = new Vector3(SimpleInput.GetAxis("LookHorizontal"), 0f,
SimpleInput.GetAxis("LookVertical"));
}
void FixedUpdate(){
      float rotation = Mathf.Atan2(shootDirection.x, shootDirection.z) *
Mathf.Rad2Deg;
      transform.rotation = Quaternion.Euler(0, rotation, 0);
}
```

Ispis 3: Implementacija rotacije na Android

U Android implementaciji rotacija se ostvaruje preko unosa. Vektor koji se dobije od unosa se pretvara u kut. Igrač gleda u smjeru u kojem je gljivica (engl. *Analog*) usmjerena. Koristi se asset `SimpleInput` koji olakšava implementaciju kontrola na dodir.

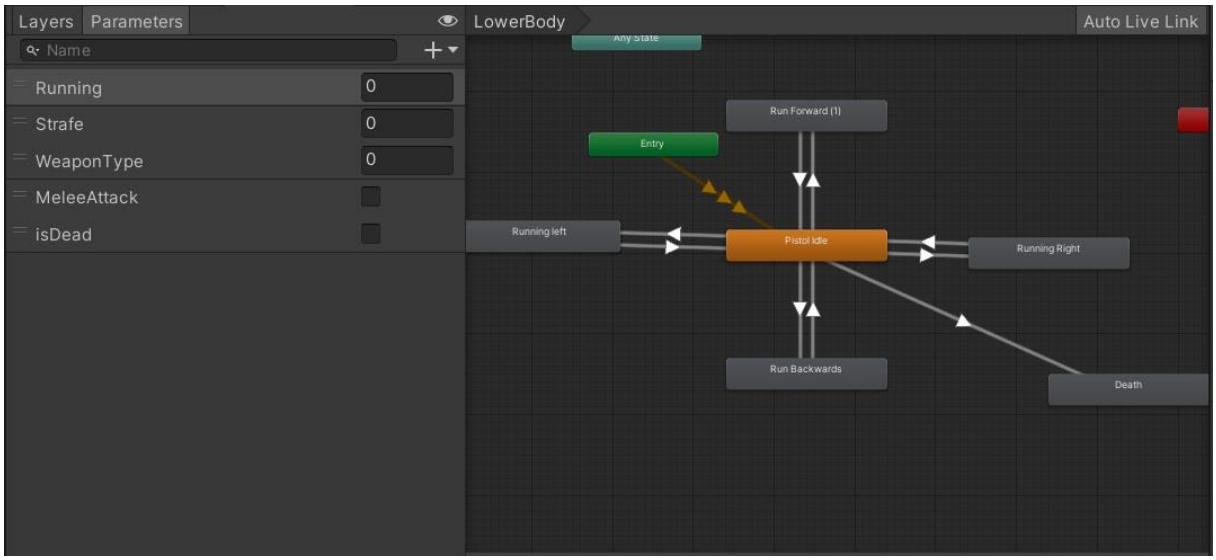
3.3.3. Animacije

Kako bi igrač izgledao bolje dodaju se animacije na igrača. Najlakši način animiranja je korištenje web stranice zvane Mixamo. Mixamo je stranica koja sadrži kolekciju animacija iz koje se mogu birati animacije koje odgovaruju igri. Preuzete animacije se spremaju u projekt.

Na objekt igrača se dodaje komponenta zvana `Animator`, a u eksplorer se stvara `Animator controller`.

`Animator` je komponenta koja se dodaje objektima u sceni kako bi se omogućilo upravljanje animacijama tog objekta. Animator Controller je grafički prikaz stanja i prijelaza između animacija koje se koriste za upravljanje Animatorom.

U animator prozoru se dodaju sve animacije koje objekt koristi te ih se spaja vezama. Da bi animator znao kako pokrenuti neku animaciju koriste se parametri kojima se vrijednost mijenja u skripti. U animator prozoru, vezama se postavljaju uvjeti koji koriste parametre. Povezani animator se može vidjeti na slici 4.



Slika 4: Animator

Igrač u igri će mijenjati animaciju na temelju smjera kretanja i gledanja. To se ostvaruje tako da se izračuna kut između ta dva vektora. Kôd je prikazan na ispisu 4.

```
Quaternion vecRotation = Quaternion.FromToRotation(rb.velocity.normalized,
transform.forward);
float velAngle = vecRotation.eulerAngles.y;
void animationHandler(float angle)
    {
            if (rb.velocity.magnitude > 0.1)
            {
                  if (angle < 45 || angle > 315)
                  {
                        animator.SetFloat("Running", 1);
                        animator.SetFloat("Strafe", 0);
                  }
                  else if (angle > 45 && angle < 135)
                  {
                        animator.SetFloat("Running", 0);
                        animator.SetFloat("Strafe", -1);
                  }
                  else if (angle > 135 && angle < 225)
                  {
                        animator.SetFloat("Running", -1);
                        animator.SetFloat("Strafe", 0);
                  }
                  else if (angle > 225 && angle < 315)
                  {
                        animator.SetFloat("Running", 0);
                        animator.SetFloat("Strafe", 1);
                  }
            }
            else
            {
                  animator.SetFloat("Running", 0);
                  animator.SetFloat("Strafe", 0);
            }
      }
```

Ispis 4: Kontrola animacije

Kut se dobije sa funkcijom `FromRotation` iz klase `Quaternion`. Quaternion je matematički objekt koji se koristi za prikazivanje rotacija u trodimenzionalnom prostoru. U parametre funkcije unose se vektori te se koristi samo kut na osi y.

Parametre animacije mijenjamo s funkcijom `SetFloat` koja je dio klase `Animator`. Mijenjanjem vrijednosti parametara se pokreću određene animacije zbog uvjeta koji se postavljaju u grafičkom prikazu.

3.3.4. Pucanje i mijenjanje oružja

Da bi objekt oružja uzastopno pratio igrača, objekt oružja se postavlja kao dijete objektu igrača. Igračev model je sadrži kostur što omogućuje da objekt oružja postavimo kao dijete ruke

od igrača. Rigged modeli su 3D modeli koji su opremljeni s kostima i kožom kako bi se omogućila animacija i deformacija modela.

Kada igrač kupi novo oružje preko skripte, novi objekt oružja se preko prefaba stvara u igračeve ruke. To oružje se deaktivira ako se trenutno ne koristi. Kako se svi objekti oružja stvaraju u istom mjestu, mora se aktivirati samo oružje koje je aktivno. U skripti se koristi enum `WeaponType` koji će pratiti koji tip oružja igrač trenutno koristi. Preko `WeaponType` enuma se zna koje oružje treba aktivirati, a koje deaktivirati. Objekt se deaktivira s funkcijom `SetActive` u klasi `GameObject`. Tako je ostvareno mijenjanje oružja.

Pucanje je implementirano na način da se zrake pucaju ravno naprijed iz sredine igrača. Kada zraka pogodi zombija tom zombiju će se umanjiti život. Svaki objekt oružja na sebi ima skriptu za pucanje i statistiku. Skripta statistike u sebi sadrži vrijednosti i podatke o oružju kao npr. ime oružja, jačina oružja, brzina oružja, cijena oružja itd. Svakom ispucanom zrakom stvara se novi objekt metka na cijevi oružja. Taj objekt će putovati od cijevi oružja do pozicije gdje zraka pogodi. Objekt metka nema nikakvu drugu funkciju osim izgleda. S tim se dobije efekt da se ispucavaju metci iako je pravi metak zapravo nevidljiv. Kôd za pucanje se može vidjeti na ispisu 5.

```
RaycastHit hit;
if (Physics.Raycast(startShootingPoint, shootingDirection, out
hit,gunStats.Range)){
      if(hit.collider.gameObject.CompareTag("Enemy")){
hit.collider.gameObject.GetComponent<ZombieStats>().takeDamage(gunStats.Damage)
;
      hitPosition = hit.point;
      }
      else{
      hitPosition = startShootingPoint + transform.root.forward *
gunStats.Range + new Vector3(1,0,0);
      }

}
else{
hitPosition = startShootingPoint + transform.root.forward * gunStats.Range +
new Vector3(1, 0, 0);
}
GameObject trail = Instantiate(gunStats.BulletTracer,
gunStats.ShootingPoint.position, Quaternion.identity);
trail.GetComponent<BulletController>().HitPosition = hitPosition;
gunStats.CurrentClipAmmo--;
nextShot = (1.0f / gunStats.FireRate) + Time.time;
```

Ispis 5: Kôd za pucanje iz oružja

`Instantiate` je funkcija s kojom se stvara novi objekt u igri preko skripte. Svaki put kada se ispuca zraka, stvara se novi objekt metka. Ako je zraka pogodila metu metak će putovati

do nje, u slučaju da je zraka promašila, metak će putovati ravno naprijed do udaljenosti koje je specificirano u statistici oružja. Na kraju skripte se određuje vrijeme kada oružje može ponovno ispaliti metak.

3.3.5. Mijenjanje spremnika na oružju

Svako oružje ima više spremnika koje igrač treba mijenjati. Funkciju za mijenjanje spremnika se može vidjeti na ispisu 6.

```
private void reload()
    {
        if (isReloading && reloadTimeElapsed >= gunStats.ReloadTime)
        {
            int shotBullets = Mathf.Abs(gunStats.MaxClip -
gunStats.CurrentClipAmmo);
            gunStats.CurrentClipAmmo = gunStats.CurrentAmmo < shotBullets ?
gunStats.CurrentClipAmmo + gunStats.CurrentAmmo : gunStats.CurrentClipAmmo +
shotBullets;
            gunStats.CurrentAmmo -= shotBullets;
            gunStats.CurrentAmmo = gunStats.CurrentAmmo < 0 ? 0 :
gunStats.CurrentAmmo;
            isReloading = false;
            reloadTimeElapsed = 0;
            player.updateAmmoUI();
            reloadUI.gameObject.SetActive(false);
        }
        if (gunStats.CurrentAmmo > 0 && ((Input.GetKeyDown(KeyCode.R) &&
gunStats.CurrentClipAmmo < gunStats.MaxClip) || gunStats.CurrentClipAmmo <= 0)
&& !isReloading)
        {
            isReloading = true;
            reloadTimeElapsed = 0;
            reloadUI.gameObject.SetActive(true);
            if(gunStats.Type == WeaponType.Secondary)
            {
                audioManager.Play("PistolReload");
            }
            else if(gunStats.Type == WeaponType.Primary)
            {
                audioManager.Play("RifleReload");
            }
        }
        if (isReloading)
        {
            reloadTimeElapsed += Time.deltaTime;
            reloadUI.eulerAngles += new Vector3(0, 0, 50 * Time.deltaTime);
        }
}
```

Ispis 6: Funkcija za mijenjanje spremnika

Kada igrač pokrene akciju mijenjanja spremnika, funkcija postavlja varijablu `isReloading` na istinu. Zatim se pokreće brojač vremena, reprodukcija zvuka za mijenjanje spremnika te animacija koja rotira ikonu koja predstavlja mijenjanje spremnika. U igri,

animacija mijenjanja spremnika se postiže jednostavnim rotiranjem ikone koja vizualno prikazuje tu radnju.

Tijekom odbrojavanja brojača, igraču se onemogućuje pucanje. Nakon što brojač dosegne svoju završnu vrijednost, provjerava se koliko je metaka ispucano i koliko ih je preostalo. Prvo se provjerava koliko metaka je ispucano iz spremnika, a ta vrijednost se oduzima od ukupnog broja metaka. Kada se prikupe sve potrebne informacije, spremnik se puni do kraja ako je preostalo dovoljno metaka. Ako nije, spremnik se puni s preostalim metcima, ili se ne puni uopće ako nisu dostupni metci.

3.4. Izrada zombija

Izrada umjetne inteligencije za neprijatelje zna biti izazovni zadatak, zato postoje mnogi alati koji olakšaju taj proces. Jedan od tih alata je komponenta koja se zove `NavMeshAgent`, ona će nam automatski izračunati najkraći put do tražene lokacije. NavMeshAgent koristi generiranu navigacijsku mrežu (NavMesh) kako bi odredio valjane putove i kretao se od točke A do točke B u igri. Navigacijska mreža se generira na temelju geometrije scene i definira dostupne prolaze i područja za kretanje objekta.

U komponenti se namješta NavMesh tako da njegov collider i ostale vrijednosti pašu s igračem. Kako bi zombiji znali put do igrača trebaju dohvatiti njegovu lokaciju. Unity ima jednostavan način prosljeđivanja objekata drugom objektu. U skripti se privatnoj varijabli doda C# atribut `SerializeField` ili se varijabla pretvori u public. Prva opcija je bolja jer varijabla ostaje privatna.

Za lokaciju igrača koristi se `Transform` varijabla s atributom `SerializeField`. `Transform` je komponenta koja se koristi za predstavljanje i manipulaciju pozicije, rotacije i skale objekta u trodimenzionalnom prostoru. U inspektoru zombie skripte u prazno polje nove varijable priduži se objekt igrača iz hijerahije.

Klasa `NavMeshAgent` ima funkciju `setDestination` koja se koristi da bi zombi pratio igrača. U inspektoru NavMeshAgenta se postavlja udaljenost na kojoj zombije prestaje pratiti metu. Ta udaljenost se koristi kada zombi priđe dovoljno blizu igrača da pokrene fazu udarca. Također se može zaustaviti funkcijom `Stop`.

Naravno da bi zombi znao pronaći put do igrača, mora znati što je put. Označe se svi objekti po kojima bi zombi trebao hodati te kliknemo na karticu navigacija (engl. *Navigation*) koja se nalazi na vrhu do inspektora. U prozoru navigation, u okviru dijela Object treba omogućiti navigation static, a navigation area postaviti na walkable. Isti proces treba ponoviti za zidove samo zamijeniti walkable na not walkable. Kada su objekti označeni u dijelu Bake pritisne se botun Bake koji se nalazi na kraju prozora navigacije. Sada zombiji uspješno mogu pratiti igrača po nivou.

Animacije se postavljaju na isti način kao kod igrača. Napad je implementiran na način kada zombi stane, aktivira se animacija za napad i pokrene se brojač koji traje istom duljinom kao i animacija. Kada brojač dosegne trenutak udarca, provjeravamo poziciju igrača u odnosu na zombija. Ako se igrač nalazi ispred zombija, smanjujemo igračev život. Kada brojač završi, zombi nastavi pratiti igrača. Kôd kontrole zombija se može vidjeti na ispisu 7.

```
if (!isAttacking)
        {
            if (!canAttack)
            {
                attackDelayElapsed += Time.deltaTime;
                if (attackDelayElapsed >= zombieStats.AttackDelayTime)
                {
                    canAttack = true;
                    attackDelayElapsed = 0;
                }
            }
            agent.SetDestination(player.position);
            anim.SetFloat("Walking", agent.velocity.magnitude);
            if (Vector3.Distance(agent.transform.position, player.position) <=
agent.stoppingDistance && canAttack)
            {
                isAttacking = true;
                anim.SetFloat("Walking", 0);
                anim.SetBool("isAttacking", isAttacking);
                attackTimeElapsed = 0;
            }

        }
        else
        {
            attackTimeElapsed += Time.deltaTime;
            if (attackTimeElapsed >= zombieStats.AttackAnimationTime)
            {
                attackDelayElapsed = 0;
                attackTimeElapsed = 0;
                canAttack = false;
                isAttacking = false;
                anim.SetBool("isAttacking", isAttacking);
                damageDealt = false;
            }
            if (attackTimeElapsed >= zombieStats.AttackAnimDamageTime &&
!damageDealt &&
                Physics.CheckSphere(transform.position + transform.forward +
zombieStats.Offset, zombieStats.Range, zombieStats.PlayerLayer))
            {
```

```
                player.GetComponent<Player>().takeDamage(zombieStats.Damage);
                damageDealt = true;
            }
        }
```

Ispis 7: Kôd za kontrolu zombija

Za provjeru pozicije igrača u odnosu na zombija koristi se funkcija `CheckSpehere`. `CheckSphere` je metoda koja se koristi za provjeru kolizije u obliku sfere u 3D prostoru. Parametri funkcije su pozicija, radijus i sloj igrača. Sloj (engl. layer) u Unity-u je kategorizacija koja se koristi za organiziranje objekata u sceni i kontrolu kolizija i vidljivosti između njih. Tag je sličan sloju samo što se tag koristi kao identifikator. Funkcija `CheckSphere` provjerava nalazi li se bar jedan objekt s slojem Player u označenom prostoru, ako se nalazi onda će smanjiti igraču život. Anim je ime varijable za animator.

3.5. Izrada UI-a

UI se radi tako da se napravi platno u kojemu se moraju nalaziti svi UI elementi jer inače neće funkcionirati. Platno se radi tako da se pritisne desnim klikom u hijerarhiji te u sekciji UI pritisne se na Canvas.

Da bi UI igre podržavao sve rezolucije u inspektoru platna treba UI Scale Mode postaviti da se mjeri s veličinom ekrana. Svi elementi platna se pozicioniraju tako da se odabere sidrište elementa. Sidrište elementa može biti bilo koji kut ili sredina na svim stranama kao i sama sredina. Preko sidrišta Unity zna gdje pozicionirati ili kako povećati elemente ovisno o rezoluciji uređaja.

Za UI elemente najčešće se koristi botun. Botun se radi tako da se napravi novi objekt, najčešće je to objekt slika (engl. *Image*), te se na njega stavi komponenta `button`. Komponenta `button` dodjeljuje funkcionalnost botuna objektu, omogućujući izvršavanje određene akcije ili funkcije kada se botun pritisne. Da bi botun odradio neku funkcionalnost, u inspektoru od `button` komponente se dodaje novi OnClick događaj. U novom događaju se dodaje objekt koji na sebi sadrži skriptu s funkcijama koje želimo izvršiti.

S botun komponentama su ostvarene funkcije kupovanja oružja, glavnog izbornika i kontrola. Element botun se automatski može koristiti na android uređaju, odnosno sve što se može koristiti pritiskom miša na Windowsu djeluje i na Androidu. Kôd za kupnju oružja se može vidjeti na ispisu 8.

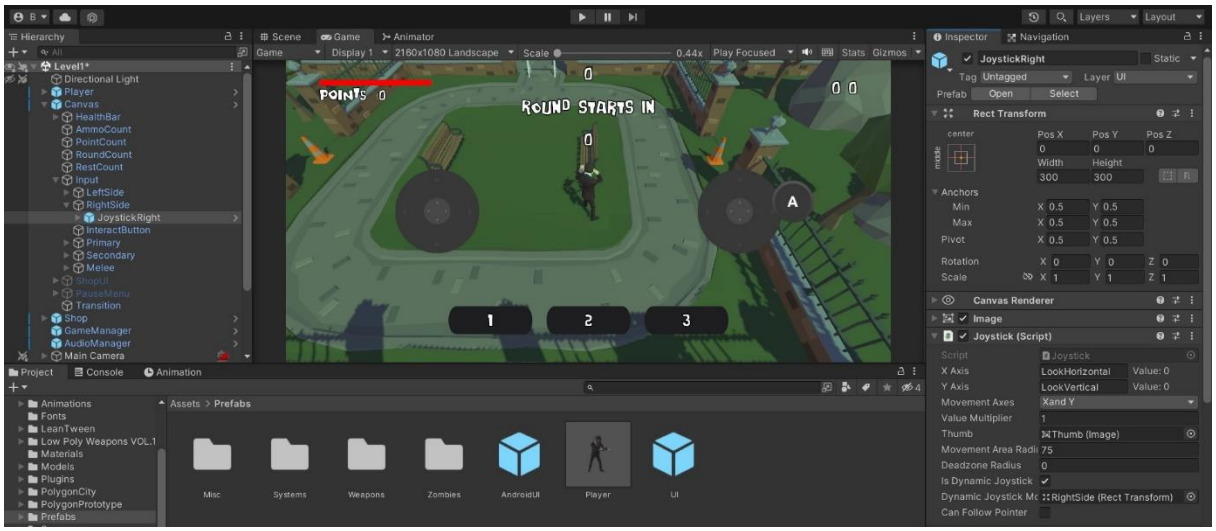
```
    public void getPrimary(int index)
    {
        GunStats stats = primaryWeapons[index].GetComponent<GunStats>();

        if (stats.Cost <= playerStats.Points)
        {
            playerStats.buy(stats.Cost);
            if (playerStats.PrimaryHolder.childCount > 0)
                Destroy(playerStats.PrimaryHolder.GetChild(0).gameObject);
            GameObject gunObj = Instantiate(primaryWeapons[index],
playerStats.PrimaryHolder.position, playerStats.PrimaryHolder.rotation);
            gunObj.transform.SetParent(playerStats.PrimaryHolder);
            gunObj.SetActive(false);
        }
    }
```

Ispis 8: Kôd za kupnju oružja

U skripti postoji lista svih objekata oružja. Svaka kupnja oružja ima svoj indeks preko kojeg funkcija razaznaje koje oružje treba stvoriti u ruke igrača. Funkcija `Destroy` se koristi za uništavanje objekata. U funkciji se uništava stari objekt oružja i na njegovo mjesto se stvara novi.

3.5.1. UI kontrole

Sve kontrole se ostvaruju sa komponentom `Button` osim takozvanih gljivica (engl. *Analog*). Za izradu gljivica koristi se prefab od `SimpleInputa`. `SimpleInput` ima par prefabova koje se mogu koristiti, a jedan od njih je dinamička gljivica. Da bi gljivica funkcionirala prvo će se izraditi dva objekta koji će imati veličinu polovice ekrana, jedan lijeve polovice, a drugi desne polovice. Koristi se sidrište da objekti automatski poprime veličinu polovice ekrana. Unutar tih objekata postavlja se objekt gljivice. U inspektoru gljivice u polja X i Y os unose se imena kontrola za rotaciju. S obzirom da su gljivice dinamičke one će se stvoriti na pozicije dodira prsta s ekranom. Postavljene UI kontrole se mogu vidjeti na slici 5.



Slika 5: UI kontrole

3.6. Zvuk

Kod videoigara zvuk je jedan jako bitan čimbenik. Igrama može dati sasvim drugačiji doživljaj igranja. Kod implementacije zvuka najbolji način je odmah učitati sve zvukove te ih puštati kada je to potrebno. To je važno jer je učitavanje dosta sporo zato ako igra gdje se puca iz oružja i zvuk kasni sa učitavanjem, kvarimo doživljaj igre.

Da se zvuk reproducirao u Unityu koristi se komponenta `AudioSource`. `AudioSource` je komponenta koja se koristi za reprodukciju zvuka. Ona je povezana s audio datotekom ili drugim izvorom zvuka te omogućuje kontrolu nad reprodukcijom, zvučnim efektima i svojstvima zvuka.

Problem nastaje kada igra sadrži više zvukova onda te ih se ne može raspoznati. Ovaj problem se rješava tako da se napravi nova klasa koja će kao atribute imati ime, klip, jačinu zvuka i klasu `AudioSource`. Iznad novostvorene klase mora se staviti C# atribut `System.Serializable`. U skripti za kontrolu zvuka će biti lista ove klase. Ako nema navedeni atribut, u klasi za kontrolu zvuka nije moguće dodati novi objekt klase preko inspektora.

Skripta za kontrolu zvuka će prilikom svog stvaranja automatski učitati sve zvukove i preko nje će se puštati zvukovi. Skripta se može vidjeti na ispisu 9.

```
public Sound[] Sounds;
void Awake()
    {
        foreach(Sound s in Sounds)
        {
            s.audioSource = gameObject.AddComponent<AudioSource>();
            s.audioSource.clip = s.clip;
            s.audioSource.pitch = s.pitch;
            s.audioSource.volume= s.volume;
        }
    }
public void Play(string name)
      {
        Sound s = Array.Find(Sounds, (sound) => sound.name == name);
        s.audioSource.Play();
      }
public void PlayOneShot(string name)
      {
            Sound s = Array.Find(Sounds, (sound) => sound.name == name);
        s.audioSource.PlayOneShot(s.audioSource.clip);
      }

public void Stop(string name)
    {
            Sound s = Array.Find(Sounds, (sound) => sound.name == name);
        s.audioSource.Stop();
    }

public void CheckPlay(string name)
    {
            Sound s = Array.Find(Sounds, (sound) => sound.name == name);
        if(!s.audioSource.isPlaying)
        {
            s.audioSource.Play();
        }
    }
```

Ispis 9: Skripta za kontrolu zvukova

Funkcija `Awake` se pokreće prije Start funkcije. U njoj se učitavaju svi zvukovi. Ostale funkcije služe za pronalaženje specifičnog zvuka te pokretanje tog zvuka.

Funkcija `Play` pokreće zvuk. Problem ove funkcije je kada se zvuk pokrene, a da reproduciranje trenutnog zvuka nije završeno, funkcija će zaustaviti trenutni zvuk te reproducirati novi zvuk. Za pucanje iz oružja koristi se funkcija `PlayOneShot`. Ova funkcija rješava problem funkcije `Play`. Funkcija `Stop` služi da se zvuk zaustavi, a funkcija `CheckPlay` za provjeru dali se zvuk pušta.

Ova skripta se primijeni na prazni objekt koji se ne uništava. Da bi se neki zvuk pustio, u skripti iz koje je potrebno pustiti zvuk, koristimo funkciju `FindObjectOfType`. `FindObjectOfType` je statička metoda u Unityu koja se koristi za pronalaženje jedne instance komponente u sceni. Ova metoda traži i vraća prvu instancu komponente koja se

podudara s traženim tipom. Ova funkcija se pokreće u funkciji `Start` nekog objekta te rezultat spremimo u neku varijablu. Sada imamo pristup našem objektu za kontrolu zvuka. Primjer se može vidjet u ispisu 10.

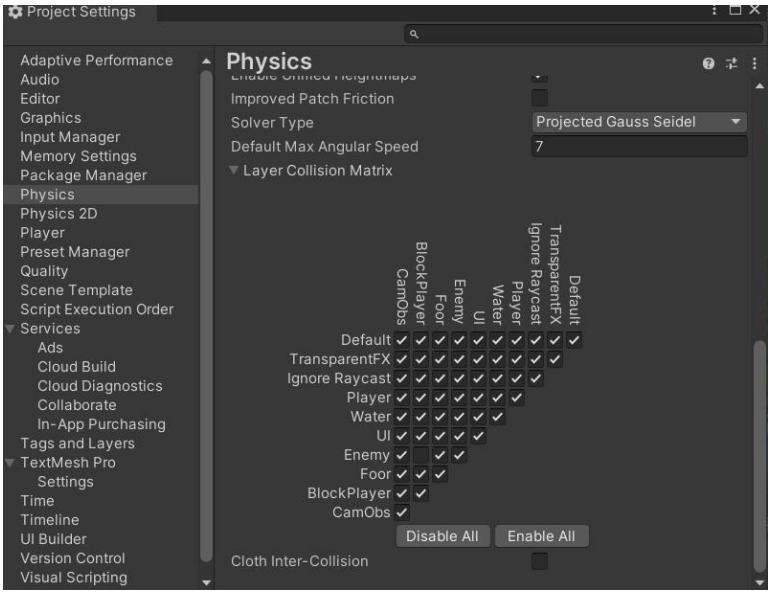
```
void Start()
    {
        audioManager = FindObjectOfType<AudioManager>();
    }

if(gunStats.Type == WeaponType.Secondary)
    {
        audioManager.Play("PistolReload");
    }
```

Ispis 10: Primjer puštanja zvukova

3.7. Prepreke

U igri se koristi više vrsta prepreka. Jedna vrsta prepreke se koristi za napredovanje kroz nivo, dok druga vrsta prepreke služi kao nevidljivi zid koji zaustavlja igrača od pristupa područjima koja nisu namijenjena za njega. Unity omogućuje jednostavno stvaranje nevidljivih zidova pomoću slojeva. Na novi objekt se dodaje posebni sloj i komponenta `collider` koja se proširuje kako bi zatvorila prostor koji treba blokirati.

Nakon stvaranja objekta, u postavkama projekta pod sekcijom fizike, potrebno je onemogućiti koliziju između sloja zombija i sloja prepreke kako se ne bi međusobno blokirali. Zombiji sada mogu proći kroz prepreku, dok igrač ne može. To se radi jer igrač nije namijenjen da hoda lokacijom gdje se zombiji stvaraju.



Slika 6: Postavke gdje se onemogućuje blokiranje

Da bi se ostvarila druga vrsta prepreke koristi se funkcija `OnTriggerEnter` koja prati ulazak objekta u okidač. Prvo, stvara se novi objekt s dva collidera. Jedan collider će spriječiti igrača da prođe kroz prepreku, dok će drugi collider djelovati kao okidač. U inspektoru collidera koji je namijenjen da bude okidač, potrebno je omogućiti opciju Is Trigger.

Obično provjere okidača se provjeravaju nad jednim objektom, a u igri taj objekt će biti igrač jer on ulazi u okidače. Kada igrač uđe u okidač i pritisne botun za kupnju prolaza, provjerava se ima li igrač dovoljno novca. Ako ima dovoljno novca, uklanja se prepreka i dodaju se nove lokacije za stvaranje zombija na novom području.

```
	private void OnTriggerEnter(Collider other)
	{
		if(other.CompareTag("Pickup"))
	  {
		other.GetComponent<Pickup>().pickingUp(this);
	  }
	  if(other.CompareTag("Shop"))
	  {
		other.GetComponent<Shop>().setPlayerClose(true);
	  }
	  if(other.CompareTag("PurchasableBarrier"))
	  {
		other.GetComponent<PurchaseBarrier>().DeOrActiveCostInfo();
	  }
	  else if(other.CompareTag("End"))
		{
			other.GetComponent<PurchaseBarrier>().DeOrActiveCostInfo();
		}
	}
```

Ispis 11: Provjera okidača

Metoda `OnTriggerEnter` se poziva kada igrač uđe u okidač. Provjerava se tag objekta u kojeg je ušao u igrač. Ako igrač ima dovoljno novca, uklanja se prepreka i dodaju se nove lokacije za stvaranje zombija.

3.8. Prikupljanje metaka i života

Igrač treba imati način da vrati svoje potrošene metke ili da vrati svoj izgubljeni život. Kupnjom novog oružja igrač može nastaviti pucati zbog toga što su nova oružja uvijek puna metcima. Taj način vraćanja metaka postaje jako skup jer svoje bodove treba trošiti za napredak kroz nivo.

Zato se dodaje pickup. Pickup u igrama se odnosi na nekakav resurs koji igrač može pokupiti. U igri će se rijetko pojavljivati pickupi, kao što su streljivo i resursi za obnovu zdravlja igrača, s ciljem očuvanja visokog stupnja izazova u igri. Tako se navodi igrača da pametno troši metke i pazi na svoj život.

Pickup se radi tako da se model prebaci u scenu. U inspektoru novostvorenog objekta se omogućuje opcija Is Trigger koja se nalazi u komponenti collider. Igrač ne smije zapinjati za pickup zato ima samo jedan collider koji funkcionira kao okidač. Skripta pickupa sadrži dvije bool varijable, `isAmmo` i `isHealth`, koje služe za definiranje resursa koje će pickup pružiti. U funkciji Start koristi se funkcija LeanTween kako bi se objekt mogao animirati. Funkcija se može vidjeti na ispisu 12.

```
private void Start()
   {
      LeanTween.moveLocalY(gameObject,    transform.localPosition.y    +    0.5f,
2.0f).setEaseInOutSine().setLoopPingPong();
        LeanTween.rotateAround(gameObject, Vector3.up, 360f, 5.0f)
        .setEase(LeanTweenType.linear)
        .setLoopClamp();
    }
```

Ispis 12: Start funkcija u Pickup skripti

Funkcija `moveLocalY` se koristi za pomicanje objekta po y osi. Funkcija `setEaseInOutSine` primjenjuje mekani početak i završetak na animaciji, što rezultira

glatkim prijelazom između početne i krajnje vrijednosti. Kada je metoda `setLoopPingPong` primijenjena, animacija će se automatski izvršavati od početne do krajnje vrijednosti, a zatim natrag od krajnje do početne vrijednosti, stvarajući efekt naprijed-natrag petljanja.

Metoda `rotateAround` omogućava rotaciju objekta u smjeru kazaljke na satu ili suprotno od smjera kazaljke na satu oko zadane točke. Metoda `setEase` se koristi za postavljanje vrste interpolacije za animaciju. U ovom slučaju, `LeanTweenType.linear` označava linearnu interpolaciju, što znači da će se rotacija odvijati ravnomjerno bez ubrzanja ili usporavanja. Kada se primijeni metoda `setLoopClamp`, animacija će se petljati kontinuirano u oba smjera, ali će biti ograničena na određeni raspon.

U skripti se nalazi samo jedna funkcija koja ima zadatak provjeriti i pružiti resurse te odlučiti treba li vratiti resurse ili uništiti objekt. Ako igračev život nije pun, funkcija će popuniti život igrača, a zatim uništiti objekt. Isto se načelo primjenjuje i na metke. Funkcija se može vidjeti na ispisu 13.

```
public void pickingUp(Player player)
    {
            bool isTaken = false;
            if (isAmmo)
            {
                if (player.PrimaryHolder.childCount > 0)
                {
                    if(player.PrimaryHolder.GetChild(0).GetComponent<GunControl
ler>().refillAmmo())
                    {
                        isTaken = true;
                    }
                }
                if (player.SecondaryHolder.childCount > 0)
                {
                    if
(player.SecondaryHolder.GetChild(0).GetComponent<GunController>().refillAmmo())
                    {
                        isTaken = true;
                    }
                }
            }
            if (isHealth)
            {
                if (player.Health < player.MaxHealth)
                {
                    player.restoreHealth();
                    isTaken = true;
                }
            }
            if (isTaken)
            {
                Destroy(gameObject);
            }}
```

Ispis 13: Funkcija za provjeru i pružanje resursa

3.9. Runde i stvaranje zombija

Kako bi kôd bio pregledniji odvajaju se skripte za stvaranje zombija i kontrolu rundi. Obje skripte se dodaju na isti objekt. U skripti za kontrolu rundi se koristi brojač koji će odbrojavati koliko vremena je ostalo prije nego što počne nova runda. U istoj skripti se zadaje u kojoj rundi će se početi stvarati jači ili brži zombiji i ažurira se brojač na UI elementu tako da igrač zna kada počinje nova runda. Skripta za kontrolu rundi se može vidjeti na ispisu 14.

Skripta za stvaranje zombija je malo složenija. U skripti za stvaranje zombija se ne koristi funkcija `Update` niti `Start`. U ovoj skripti su potrebne samo funkcije za provjeru i stvaranje zombija.

```
if(isRestInProgress)
        {
            restTimeElapsed += Time.deltaTime;
                   restUI.text = "Round starts in\n" + (restTime -
Mathf.Round(restTimeElapsed)).ToString();
                   if (restTimeElapsed >= restTime)
            {
                restTimeElapsed = 0;
                isRestInProgress = false;
                isRoundInProgress = true;
                spawner.startSpawning();
                           restUI.text = "Round starts in\n" + (restTime -
Mathf.Round(restTimeElapsed)).ToString();
                LeanTween.scale(restUI.gameObject, Vector3.zero,
0.5f).setEase(LeanTweenType.easeOutExpo)
                                    .setOnComplete(() => {
restUI.gameObject.SetActive(false); });
            }
        }
        else if(isRoundInProgress)
        {
            if(spawner.isRoundOver())
            {
                currentRound++;
                roundUI.text = currentRound.ToString();
                isRoundInProgress = false;
                isRestInProgress = true;
                spawner.resetSpawner();
                spawner.increaseZombies();
                           restUI.gameObject.SetActive(true);
                           LeanTween.scale(restUI.gameObject, Vector3.one, 0.5f);
                           if (currentRound == runnerZombieRoundSpawn)
                {
                    spawner.startSpawningRunnerZombie();
                }
                if(currentRound == tankZombieRoundSpawn)
                {
```

```
                    spawner.startSpawningTankZombie();
                }
            }
        }
```

Ispis 14: Skripta za kontrolu rundi

LeanTween je popularna biblioteka za animaciju i interpolaciju vrijednosti u Unityu. Omogućuje glatko i jednostavno animiranje svojstava objekata, poput pozicije, rotacije, skale i drugih atributa.

Animator je dosta loš po pitanju perfomansi te ako su animacije jako jednostavne, onda je dobro koristiti LeanTween. Skripta se vrti u krug, kada runda počne, brojač čeka da svi zombiji budu eliminirani nakon toga počinje odmor od deset sekundi tijekom kojega se može kupiti novo oružje zatim ponovno ide ispočetka.

```
public void startSpawning()
    {
        if(!isSpawning)
        {
            isSpawning = true;
            StartCoroutine(spawnRegularZombies());
                  if(isSpawningRunnerZombies)
                  {
                        StartCoroutine(spawnFastZombies());
                  }
                  if(isSpawningTankZombies)
                  {
                        StartCoroutine(spawnTankZombies());
                  }
             }
    }
```

Ispis 15: Funkcija za početak stvaranja zombija

Za stvaranje zombija koriste se asinkrone funkcije zvane `coroutine`. Korutina (engl. coroutine) je koncept u Unityu koji omogućuje izvođenje funkcija u asinkronom režimu, omogućujući vremenski određene ili ponavljajuće operacije bez blokiranja glavne niti izvođenja igre. Da bi se takva funkcija napravila, funkcija mora vraćati IEnumerator te prije vraćanja mora koristiti yield. Funkcija za stvaranje zombija se može vidjeti u ispisu 16.

```
    private IEnumerator spawnRegularZombies()
    {
        while(regularZombiesSpawned < regularZombies)
        {
            if(isSpawning)
            {
                int ranIndex = Random.Range(0, spawnPositions.Count);
                Instantiate(regularZombiePrefab,
spawnPositions[ranIndex].position, Quaternion.identity);
                currentlySpawnedZombies++;
                regularZombiesSpawned++;
                        if(currentlySpawnedZombies >= maxSpawnedZombies)
                        {
                              isSpawning = false;
                        }
            }

            yield return new WaitForSeconds(spawnInterval);
        }}
```

Ispis 16: Funkcija za stvaranje zombija

Funkcija prikazana u ispisu 16 se koristi za početne zombije. Za druge vrste zombija koristi se identična funkcija samo se stvara druga vrsta zombija. U funkciji se provjerava dali je stvaranje zombija dozvoljeno ako je onda se nasumično bira jedna lokacija iz liste lokacija i tamo se stvara novi zombi. Kako ne bih došlo do pada FPS-a, igri se treba limitirati koliko zombija može postojati u jednom trenutku. To je ostvarenom s uvjetom u while petlji. Svaka runda treba stvarati više zombija nego prethodna. U igri broj zombija se povećava za 1,5 puta svaku rundu. Funkcija se može vidjeti u ispisu 17.

```
      public void increaseZombies()
      {
            regularZombies = (int)Mathf.Round(regularZombies * spawnMutiplier);
            if(isSpawningRunnerZombies)
                  fastZombies = (int)Mathf.Round(fastZombies *
spawnMutiplier);
            if(isSpawningTankZombies)
                  tankZombies = (int)Mathf.Round(tankZombies *
spawnMutiplier);
      }
```

Ispis 17: Funkcija za povećanje broja zombija koji će se stvoriti za trenutnu rundu

Da bi se znalo kada je runda gotova, koristi se funkcija u kojoj se uspoređuje koliko zombija je eliminirano i koliko zombija bi trebalo biti stvoreno. Funkcija se može vidjeti u ispisu 18.

```
public bool isRoundOver()
	{
		int currZomb = 0;
		currZomb += regularZombiesSpawned;
		int maxZombs = 0;
		maxZombs += regularZombies;
		if(isSpawningRunnerZombies)
		{
			currZomb += fastZombiesSpawned;
			maxZombs += fastZombies;
		}
		if(isSpawningTankZombies)
		{
			currZomb += tankZombiesSpawned;
			maxZombs += tankZombies;
		}
		if(currZomb >= maxZombs && currentlySpawnedZombies <= 0)
		{
			return true;
		}
		return false;
	}
```

Ispis 18: Funkcija za provjeru dali je runda gotova

3.10. Izgradnja nivoa

Kada se dovrše svi glavni elementi i funkcije igre, jedino što preostaje jest da se počne graditi nivo. Izgradnja nivoa izgleda vrlo jednostavno ali to nije ni blizu istine. Izgradnja nivoa je jedan od najbitnijih faza kod izrade igre jer ako nivo nije zanimljiv, igrač će brzo izgubiti pozornost. Nivo treba da izgleda zanimljivo i kreativno kako bi navodio igrača na pravi put i kako bi igraču potakli znatiželju da istraži nivo. Naravno, izgled i osjećaj nivoa zavisi do vrste igre.

U igri se koriste modeli s malo poligona jer su ti modeli jednostavniji za napraviti i dosta zanimljivo izgledaju. Modeli se mogu samostalno izraditi u raznim alatima kao što je Blender. Blender je besplatni softver otvorenog kôda za 3D računalnu grafiku i animaciju. On pruža sveobuhvatne mogućnosti za modeliranje, animaciju, renderiranje, simulaciju i uređivanje digitalnih sadržaja.

Modeli se također mogu koristiti iz tuđih radova koji su dostupni na raznim stranicama. To omogućuje tvrtkama da kupuju već izrađene modele i koriste ih u svojim projektima. Kupovina gotovih modela štedi vrijeme i resurse jer tvrtke ne moraju izrađivati modele iz

početka. Umjesto toga, mogu iskoristiti kvalitetne modele drugih autora kako bi ubrzali svoj proces razvoja.

Prilikom gradnje nivoa, važno je da modeli koji su postavljeni samo radi dekoracije, a kojima se ne može pristupiti, budu jednostavni i bez ikakvih dodatnih komponenti osim svojih poligona. Ovi dekorativni modeli često se koriste kako bi se sakrila praznina iza nivoa, jer nijedan nivo nije beskrajan i moramo koristiti tehnike kako bismo postigli efekt da je prostor veći nego što zapravo jest.

Upotreba ovakvih jednostavnih dekorativnih modela pomaže u optimizaciji performansi igre. Kako se igra povećava i širi, važno je voditi brigu o performansama kako bi se osiguralo da igra radi glatko i bez zastoja. Korištenje jednostavnih modela bez dodatnih komponenti pomaže u smanjenju opterećenja procesora i grafičke kartice, što rezultira boljim performansama igre.

3.10.1. Tehnike izrade nivoa

U razvoju igara, često se koriste različite tehnike gradnje nivoa, među kojima su popularne linearni progres, razgranata struktura, otvoreni svijet, puzzle-based, vertical slice, modularna gradnja i proceduralna generacija. Ove tehnike su samo neke od mnogih pristupa gradnji nivoa u razvoju igara. Odabir pravilne tehnike ovisi o vrsti igre, ciljevima dizajna i željenom iskustvu za igrače.

Linearni progres je tehnika koja uključuje postupno uvođenje igrača kroz niz razina ili zona, pri čemu je svaki nivo teži od prethodnog. Linearni progres omogućuje postupno učenje mehanike igre i povećanje izazova kako igrač napreduje. Ova tehnika se koristi u igri.

Razgranata struktura je tehnika koja uključuje stvaranje različitih putanja ili grana koje igrač može istraživati. Svaka grana može predstavljati različite izazove ili alternative u igri, dajući igraču mogućnost odabira i otkrivanja različitih puteva kroz igru.

Puzzle-based tehnika se koristi u igrama fokusiranim na rješavanje zagonetki ili logičkih izazova. Svaki nivo se gradi oko određenih zagonetki ili problema koje igrač mora riješiti kako bi napredovao.

Vertical Slice je pristup koji uključuje izgradnju manjih cjelina igre koji sadrže sve ključne elemente igre, vizualnu prezentaciju, zvuk i napredak. Ovi vertikalni rezovi pomažu u testiranju i iteriranju ključnih ideja i omogućuju rano dobivanje povratnih informacija.

Modularna gradnja je tehnika koja koristi za stvaranje nivoa putem modularnih dijelova koji se mogu kombinirati na različite načine. Modularni dijelovi su kao građevni blokovi koji omogućuju brže iteracije i izgradnju raznovrsnih nivoa.

Proceduralna generacija je pristup koji koristi algoritme za generiranje nivoa ili terena na temelju određenih pravila ili parametara. To omogućuje stvaranje beskonačnog broja nivoa ili terena bez potrebe za ručnim dizajnom. Na slici 7 se može vidjeti izrađeni nivo za našu igru.

Slika 7: Prvi nivo

4. Zaključak

U ovom završnom radu istražen je proces izrade igre u programskom alatu Unity. Kroz projektiranje i implementaciju različitih komponenti igre, kao što su likovi, oružja, umjetna inteligencija zombija i okruženje, ostvarena je funkcionalna igra.

Upotreba platforme Unity omogućila je efikasno korištenje njenih moćnih razvojnih alata. Ovo je pojednostavilo proces implementacije, zahvaljujući intuitivnom i user-friendly sučelju koje omogućava korisnicima da lako manipuliraju alatima i resursima.

Kroz ovaj projekt, dobiveno je dragocjeno iskustvo u radu s platformom Unity i razumijevanju procesa kreiranja videoigara. Aktivnosti kao što su istraživanje, planiranje, implementacija i testiranje igre omogućile su primjenu stečenog znanja i razvoj vještina u području dizajna i razvoja igara.

Unatoč prisutnim izazovima tijekom razvojnog procesa, igra je uspješno stvorena, a stjecanje novih spoznaja omogućuje njihovu buduću primjenu. Ukupno gledajući, ovaj završni rad pružio je vrijedan uvid u proces izrade videoigara te je omogućio praktično iskustvo u radu s platformom Unity.

5. Literatura

[1] Wikipedia, „Unity game engine“, https://en.wikipedia.org/wiki/Unity_(game_engine) (posjećeno 10.7.2023)

[2] Mixamo, „Mixamo“, https://www.mixamo.com (posjećeno 10.7.2023)