Izrada igre pucanja iz prvog lica u programskom alatu Unity

Budak, Josip

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Josip Budak

IZRADA IGRE PUCANJA IZ PRVOG LICA U PROGRAMSKOM ALATU UNITY

ZAVRŠNI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Josip Budak

JMBAG: 0016147588

Studij: Informacijski i poslovni sustavi

IZRADA IGRE PUCANJA IZ PRVOG LICA U PROGRAMSKOM ALATU UNITY

ZAVRŠNI RAD

Mentor:

Doc. dr. sc. Mladen Konecki

Varaždin, rujan 2023.

Josip Budak

Izjava o izvornosti

Izjavljujem da je moj završni 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 potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Ovaj završni rad bavi se izradom videoigre pucanja iz prvog lica koristeći programski alat Unity. U prvim poglavljima predstavlja se alat Unity, opisuje se njegovo sučelje i tumači se njegov način rada. Zatim se definira pojam videoigre i govori se o kratkoj povijesti razvoja videoigara. Posebno se definira pojam videoigre pucanja iz prvog lica. Praktični dio rada se sastoji od kreiranja i dizajniranja razine na kojoj se igrač nalazi, implementacije programske logike potrebnih mehanika (poput pucanja i skakanja) i izrade korisničkog sučelja. U ovom dijelu se opisuje programski kôd napisan unutar skripti, kako se dodaju zvučni i svjetlosni efekti, ubacivanje modela, interakcija igrača s okolinom i integracija svih komponenata u zajedničku cjelinu. Cilj rada je izraditi jednostavnu videoigru pucanja iz prvog lica koja obuhvaća sve glavne koncepte takvog žanra.

Ključne riječi: videoigra pucanja iz prvog lica, Unity, game engine, programiranje, C#, singleplayer, razvoj videoigre;

Sadržaj

1. Uvod

Razvoj videoigara je postigao značajan napredak u posljednjim desetljećima transformirajući se iz jednostavnih pikseliziranih igara u tehnološki napredne i detaljne svjetove. Videoigre više nisu samo forma zabave, već i ogroman industrijski sektor.

Tržište videoigara konstantno raste i privlači veliki broj igrača diljem svijeta. Prema statistikama iz 2023. godine, industrija videoigara vrijedi gotovo 200 milijardi dolara. Broj aktivnih igrača je prešao 3 milijarde što je porast od 32% u posljednjih 7 godina [1].

Sa sigurnošću se može reći da je razvoj videoigara siguran posao u današnjem digitalnom dobu. Videoigre su dostupne na različitim platformama, tzv. diversifikacija platforma, a tehnološki napredak i inovacije ne pokazuju znakove usporavanja.

Za izradu složenih videoigara potrebni su alati koji ih pokreću (eng. *Game engine*). Oni pružaju resurse za stvaranje grafike, implementaciju logike, simulaciju fizike i optimizaciju performansi. Prema prikupljenim podacima, najkorišteniji alati za izradu i pokretanje 3D videoigara su Unity, Godot i Unreal [2]. Navedeni alati, ali i mnogi drugi, podržavaju razvoj videoigara za više platforma (eng. *cross-platform*) što omogućuje širenje videoigara na mnoge uređaje kao što su konzole i mobilni uređaji.

Unity je programski alat koji se koristi za razvoj videoigre u ovom radu. Više o njegovim značajkama i korištenju govori se u nastavku.

2. Metode i tehnike rada

Primarni alat korišten za izradu praktičnog dijela rada je Unity. Za pisanje C# programskog kôda korišteno je razvoj okruženje *Visual Studio 2022*. Unutar stvorene videoigre nalaze se zvučni efekt, a neki od njih su uređeni u alatu *Audacity*. Modeli za izradu razine preuzeti su s Unity trgovine (*Unity Asset Store*). U nastavku će biti opisan alat Unity.

2.1. Unity

Unity je više-platformski (eng. *cross-platform*) alat za razvoj igara koju je razvila tvrtka Unity Technologies, a koji je prvi put najavljen i pušten u uporabu u lipnju 2005. godine. Od tada je postupno proširen kako bi podržao razne platforme kao što su stolna računala (eng. *desktop computers*), mobilni uređaji, konzole i virtualna stvarnost. Posebno je popularan za razvoj mobilnih igara za iOS i Android, smatra se jednostavnim za uporabu za početne razvojne programere i cijenjen je za razvoj neovisnih igara.

Ovaj alat se može koristiti za izradu trodimenzionalnih i dvodimenzionalnih igara, kao i interaktivnih simulacija. Također je usvojen u industrijama izvan videoigara kao što su film, automobilska industrija, arhitektura, inženjering, građevinarstvo i oružane snage SAD-a [3].

Slika 1: Unity logotip (Izvor: https://logos-world.net/unity-logo/)

Unity se ističe kao pristupačan, fleksibilan i kreativan alat. Njegova sposobnost stvaranja igara bez potrebe dubokih tehničkih znanja učinili su ga konkurentnim alatom u industriji. Neke od značajka su sljedeće:

- Više-platformski alat – omogućuje razvoj igara za razne platforme
- Prilagodljivost – sposoban za izradu malih projekata i složenih AAA naslova
- Bogatstvo resursa – korisnički forumi su aktivni i bogati, postoji puno dostupnih materijala za lakše učenje i rješavanje problema

- Unity Asset Store – službena trgovina na kojoj korisnici kupuju, prodaju i dijele gotove resurse poput tekstura, 3D modela, animacija i sl.
- Pretplatnički paketi – uz plaćene planove Unity nudi i besplatan plan koji je dovoljan za većinu neovisnih korisnika

2.1.1. Korisničko sučelje

Nakon pokretanja projekta prikazuje se sljedeće sučelje (slika 2). Na skroz lijevoj strani nalazi se kartica *Hierarchy* unutar koje se nalaze Unity scene. Scene se sastoje od objekata. *Hierarchy* kartica služi za pregled i organizaciju objekata u sceni. Primjerice, kreirali smo prazan objekt *Floor* i u njega smo stavili sve objekte korištene za postavljanje tla na razini.

Na desnoj strani smješten je *Inspector*. On služi za mijenjanje svojstava odabranog objekta, poput njegove veličine i širine. Također, inspektor se koristi za dodavanje komponenti na objekte poput skripti ili, primjerice, sudarača (eng. *collider*). U praksi se razni objekti i skripte povlače iz neke druge kartice i ispuštaju (eng. *drag and drop*) u inspektor.

Na dnu sučelja se nalazi kartica *Project*. Unutar nje su vidljive sve skripte, datoteke, modeli, teksture, audio zapisi i sve preuzete datoteke s Unity trgovine. Iz ove kartice se najčešće modeli povlače u scenu.

U centru sučelja nalazi se prostor za uređivanje i vizualiziranje igre. Prikaz scene (eng. *Scene View*) omogućuje manipulaciju objektima u sceni. U ovom prikazu se objekti dodavaju, premještaju, rotiraju, skaliraju i organizira se prostor. Prikaz scene je zapravo radna površina. Prikaz igre (eng. *Game View*) omogućuje korisniku uvid u izgled i ponašanje igre tijekom stvarnog izvođenja.

Slika 2: Unity korisničko sučelje (Izvor: samostalna izrada)

2.1.2. Način rada

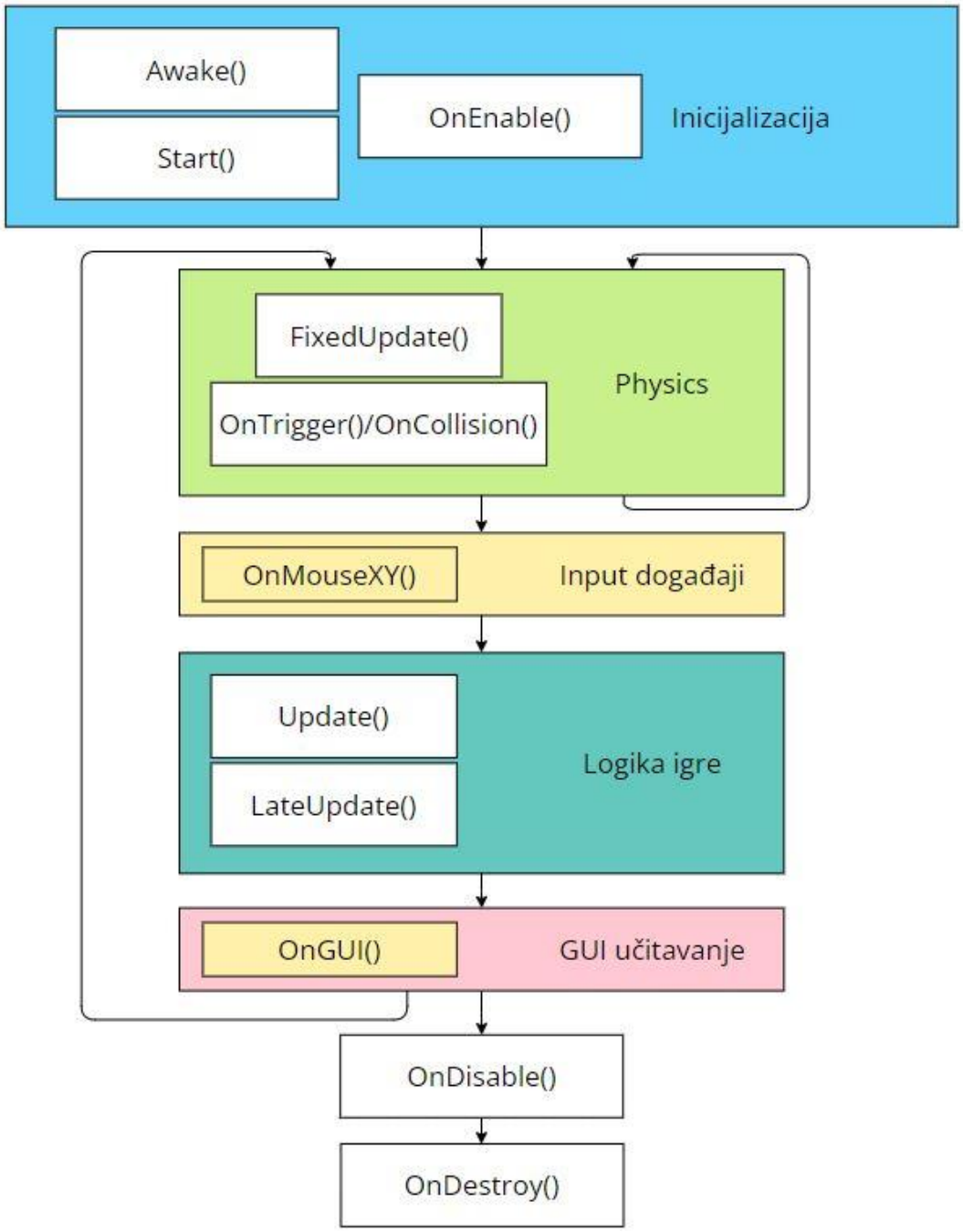
Kao i većina alata za izradu igara (eng. *game engine*), Unity se sastoji od nekoliko komponenata. Prva i glavna komponenta je program koji sadrži logiku igre. Pokretač za prikaz (eng. *rendering engine*) generira 3D animirane grafike. Komponenta zvučnog pokretača (eng. *audio engine*) pruža algoritme za stvaranje i manipulaciju zvukovima. Pokretač fizike (eng. *physics engine*) rukuje kretanjem i sudarima. Za radnje i odluke koje ne kontrolira igrač postoji komponenta umjetne inteligencije (eng. *artificial intelligence*) [4].

Unity-jev način rada se bazira na objektima (eng. *game object*) i komponentama koje im se dodjeljuju. Komponente su elementi koji definiraju ponašanje i funkcionalnosti objekata unutar scene. One se mogu zamisliti kao skripte ili moduli koji se pridodaju objektima kako bi im se upravljalo ponašanjem i svojstvima.

Unity nudi programsko sučelje za pisanje kôda u C# jeziku koristeći Mono, programski okvir kompatibilan sa .NET okvirom. *MonoBehaviour* je osnovna klasa koju nasljeđuju većina klasa koje kreira korisnik. MonoBehaviour klasa sadržava funkcije životnog ciklusa koje olakšavaju rad razvoja igre [5]:

- *Awake* – poziva se samo jednom prilikom učitavanja instance skripte
- *Start* – izvršava se prije prvog ažuriranja kadra (eng. *frame*) i koristi se za postavljanje početnih varijabli i stanja
- *Update* – poziva se svakog framea, ovdje se smješta kôd koji je potrebno stalno ažurirati poput kôda za dohvat unosa s tipkovnice
- *FixedUpdate* – poziva se u fiksnim koracima vremena

- *LateUpdate* – poziva se nakon što su sve *Update* metode pozvane
- *OnEnable* – poziva se kada se skripta omogući
- *OnDisable* – poziva se kada se skripta onemogući
- *OnDestroy* – poziva se kada se objekt uništi i korisna je za oslobađanje resursa
- *OnGUI* – poziva se više puta u frameu kada se grafičko sučelje učita
- *Ostalo* – funkciju koje se pozivaju kada igrač uđe u *collider* i sl.



Slika 3: Životni ciklus MonoBehaviour klase (Izvor: samostalna izrada)

3. Videoigra (eng. *videogame*)

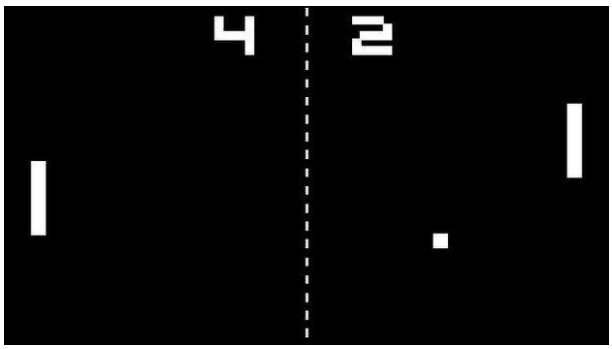
Videoigra (*eng. Video game*) je igra u kojoj igrač kontrolira pokretne slike na zaslonu pritiskom tipki [6]. Generirane slike su rezultat korištenja uređaja za unos podataka kao što je upravljač (eng. *joystick*) te miš i tipkovnica ili zaslon na dodir. Većina modernih videoigara je audiovizualna, pri čemu se zvuk prenosi putem zvučnika ili slušalica, a ponekad se koristi i druga vrsta osjetilnih povratnih informacija poput haptičke tehnologije koja pruža taktilne senzacije. Također, neke videoigre omogućuje upotrebu mikrofona i web kamera za razgovor tijekom igre i prijenos uživo (eng. *livestream*) [7].

Videoigra je oblik zabave koji omogućuje igraču da sudjeluje u virtualnom svijetu, upravljajući likovima ili objektima. Osim toga, videoigre se koriste i u obrazovne svrhe, u treninzima, simulacijama i istraživanjima, ali i kao oblik natjecanja (eng. *esports*).

3.1. Kratka povijest

Videoigre su počele nastajati u 1950-ima i 1960-ima kada su računalni znanstvenici počeli razvijati jednostavne igre i simulacije na malim računalima (eng*. minicomputers*) i računalima velikih kapaciteta (eng*. mainframe computers*). Jedan od prvih primjera je igra *Spacewar!* koju su 1962. godine stvorili MIT studenti [8].

Jedna od prvih videoigara koja je postigla ogromnu popularnost je igra Pong. Igra je 2D simulacija tenisa za dva igrača. Postala je fenomen i smatra se prvom uspješnom videoigrom. Njezina popularnost je otvorila vrata za razvoj i širenje videoigara kao popularnog oblika zabave.



Slika 4: Pong (Izvor: https://www.bbc.com/news/technology-33005297)

Klasične igre poput *Pac-Man*, *Donkey Kong* i *Super Mario Bros* su obilježile 1980-e godine, a desetljeće se često naziva zlatnim dobom arkadnih videoigara. Tijekom ovog razdoblja izdan je velik broj omiljenih i najprodavanijih arkadnih igara.

1990-ih godina pojavljuje se 16-bitna grafika koja dominira igraćom industrijom, a počinju se pojavljivati i prve 3D igre kao što je *Super Mario 64*. Arkadne igre gube na popularnosti jer se igraće konzole pojavljuju u gotovo svim domovima. Žanrovi poput pucačina iz prvog lica (*eng. first-person shooter*) i strategije se počinju sve više razvijati i privlače veliku pozornost.

Posljednjih 20-ak godina je obilježila era 3D grafika i napredak u tehnologiji. Sony PlayStation 2 postaje najprodavanija konzola svih vremena. Mobilne igre postaju izuzetno popularne dolaskom pametnih telefona i tableta. Distribucija igara se okreće digitalizaciji, pojavljuju se platforme kao što su Steam, Origin i Blizzard za kupovinu i kolekciju igrica. Pandemija COVID-19 je dovela do porasta popularnosti videoigara jer su ljudi tražili način zabave od kuće.

Slika 5: Sony Playstation 2 (Izvor: https://commons.wikimedia.org/wiki/File:Sony-PlayStation-2-30001-wController-L.jpg)

Još jedan značajan razvoj su online igre i *multiplayer* iskustva. Mogućnost igranja s drugim igračima širom svijeta je omogućila stvaranje zajednica, natjecanja i suradnje među igračima. Natjecanja privlače velike gledateljske baze i pružaju novi oblik sportske zabave.

3.2. Pucačina iz prvog lica (eng. *first-person shooter*)

Kako bi bolje razumjeli što je pucačina iz prvog lica prvo ćemo definirati pucačine.

Igrica pucačina (*eng. shooter game*) je vrsta videoigre u kojoj je glavni aspekt borba koristeći različite vrste oružja. Igrač je postavljen u situacije gdje mora ciljati i pucati na neprijatelje kako bi preživio ili napredovao prema sljedećoj razini ili zadatku. Pucačine se dodatno dijele na sljedeće podžanrove [9]:

- Klasična pucačina (eng. *Shoot 'em up*) – igrač puca ispred sebe, a kreće se gore-dolje i lijevo-desno
- Igre gađanja (eng. *Shooting gallery*) – igrač je prikazan pomoću avatara na zaslonu (obično na dnu) koji se može kretati i izbjegavati napade neprijatelja dok uzvraća vatru
- Pucačina sa svjetlosnim pištoljem (eng. *Light gun shooter*) – igre koje koriste svjetlosne pištolje koji emitiraju zrake svjetla, a mete sadrže svjetlosno-osjetljiv uređaj
- Pucačina iz prvog lica (eng. *First-person shooter*) – simuliraju pogled lika unutar igre
- Pucačina iz trećeg lica (eng. *Third-person shooter*) – potpuno prikazuje lik igrača u njegovom okruženju
- Pucačina s herojima (eng. *Hero shooter*) – igrači su podijeljeni u ekipe i biraju prethodno dizajnirane „herojske“ likove koji posjeduju jedinstvene sposobnosti
- Taktička pucačina (eng. *Tactical shooter*) – simuliraju realistične sukobe, sadrže kompleksniji izbor oružja i složenije sustave liječenja (eng. *healing*)

Dakle, pucačina iz prvog lica (*eng. First-person shooter, FPS*) je vrsta videoigre gdje igrač doživljava igru kroz perspektivu lika kojeg upravlja. Igrač gleda kroz oči tog lika i koristi oružje kako bi se borio protiv neprijatelja u trodimenzionalnom okruženju. Ovaj žanr videoigri su obilježili serijali poput *Counter-Strike, Call of Duty, Rainbow Six Siege* i drugi, a dnevno broje igrače u milijunima.

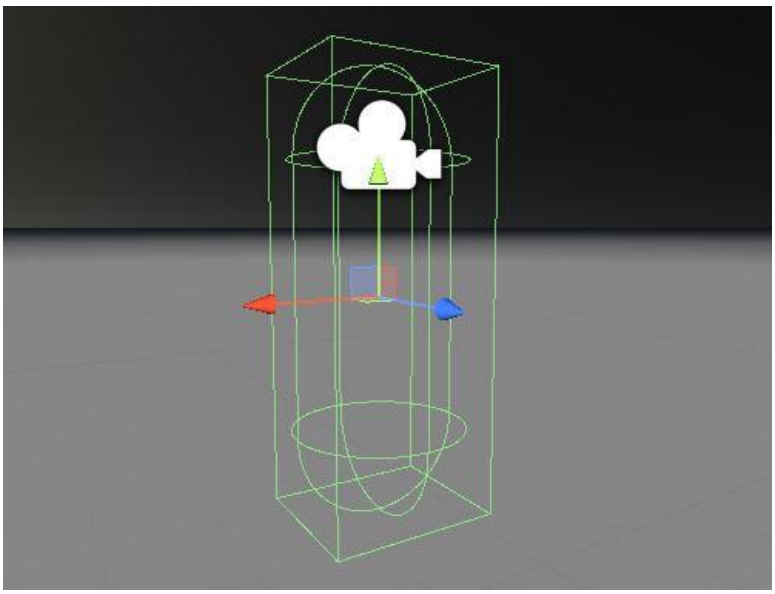
4. Izrada videoigre

Radnja igre smještena je u svemiru u znanstveno-fantastičnoj neprijateljskoj bazi. Glavni cilj igre je eliminirati sve neprijatelje i uspješno pobjeći iz baze. Ova igra pružit će uvid u ključne mehanike svake pucačke igre iz prvog lica: gledanje, kretanje, pucanje i regeneracija zdravlja. Razvijeni su i interaktivni izbornici pomoću kojih igrač može prilagoditi postavke igre prema vlastitim željama. Pomoću minimape igrač će se lakše navigirati, dok će mu grafičko sučelje pružati sve nužne informacije.

4.1. Kreiranje igrača

Unutar Unity projekta kreiramo novi prazan objekt i preimenujemo ga u *Player*. Na taj objekt dodajemo komponentu *Character Controller*, a kasnije ćemo nadodati još i *Box Collider* komponentu i nekoliko skripti. *Character Controller* komponenta služi za upravlje kretanjem i interakcijama lika kojeg igrač kontrolira, u ovom slučaju upravlja objektom *Player*.

Svaka scena sadrži i glavnu kameru (eng. *main camera*) koju također postavljamo unutar objekta *Player*. Uobičajeno je da u sceni postoji samo jedna glavna kamera. Glavna kamera je prikaz igre, odnosno, ono što igrač vidi pa je postavljamo na poziciju glave. Unutar *Player* objekta kasnije ćemo dodati i model puške uz sve potrebne efekte za pucanje.



Slika 6: Kostur objekta *Player* (Izvor: samostalna izrada)

4.1.1. Mogućnost okretanja i gledanja

Kako bi kontrolirali smjer i kut okretanja kreirali smo novu skriptu i preko inspektora je pridodali objektu *Main Camera*. Prije toga smo pozicionirali kameru na sam vrh *Player* objekta gdje bi bila glava.

Kako bi saznali u kojem smjeru okrenuti igrača potrebno je saznati gdje je korisnik pomakao svoj miš. Također nam je potrebna i osjetljivost miša pa smo napravili novu varijablu *Sensitivity* koju će kasnije igrač moći podesiti u postavkama. S obzirom da se ova skripta nalazi na *Main Camera* objektu potrebna nam je referenca na cijeli Player objekt kako bi se mogao cijeli objekt zarotirati, a ne samo kamera.

U Unityu glavna klasa koja obavlja rotacije je *Quaternion*. Quaternioni omogućuju glatke i kontinuirane prijelaze između rotacija. To su vektori s četiri dimenzije gdje četvrta dimenzija predstavlja količinu rotacije.

U *Update* metodi spremamo pomak miša u varijable i množimo ga sa *Sensitivity* varijablom. Koristeći *Clamp* funkciju ograničavamo igrača da se previše okrene, odnosno da ne može pogledati više od 90 stupnjeva u svim smjerovima. Zatim izvršavamo rotaciju kamere koristeći *Quaternion.Euler* metodu i rotiramo cijelog igrača koristeći *player.Rotate*.

```
void Update()
    {
        float mouseX = Input.GetAxis("Mouse X") * Sensitivity;
        float mouseY = Input.GetAxis("Mouse Y") * Sensitivity;

        xRotation -= mouseY;
        xRotation = Mathf.Clamp(xRotation, -90f, 90f);

        transform.localRotation = Quaternion.Euler(xRotation, 0f, 0f);
        player.Rotate(Vector3.up * mouseX);
    }
```

Unutar *Start* metode provjeravamo postoji li zapis osjetljivosti miša u *PlayerPrefs* klasi. Ukoliko igrač igra prvi put, zapis neće postojati pa se *Sensitivity* varijabla postavlja na 2.5f. U *PlayerPrefs* klasu mogu se zapisivati i dohvaćati igračeve preference, a često se koristi i za dohvaćanje podataka koji su potrebni između sesija igranja.

```
void Start()
    {
        Cursor.lockState = CursorLockMode.Locked;

        if (PlayerPrefs.HasKey("Sensitivity"))
        {
            Sensitivity = PlayerPrefs.GetFloat("Sensitivity");
        }else
        {
            Sensitivity = 2.5f;
        }
    }
```

4.1.2. Kretanje

Kretanje igrača uključuje standardno kretanje naprijed-nazad i lijevo-desno, skakanje, čučanj i ustajanje. Međutim, potrebne su nam dodatne varijable koje opisuju stanja u kojima se igrač nalazi i varijable poput brzine kretanja i iznosa gravitacije kako bi dobili efekt što bliži stvarnom svijetu. Prvo definiramo normalnu visinu igrača i visinu igrača dok je u čučnju te dozvoljenu visinu skoka. Bool varijable *IsOnGround, IsCrouched i ObjectAboveHead* će ograničiti igrača u kretanju.

U *Update* metodi provjeravamo nalazi li se igrač na tlu. To smo implementirali pomoću *CheckSphere* metode koja projicira nevidljivu sferu ispod igrača i provjerava nalazi li se površina sa slojem (eng. *layer*) *Ground* unutar *GroundCheckDistance* udaljenosti. Ukoliko se tlo nalazi unutar te udaljenosti, funkcija vraća *true*. Prije toga smo kreirali novi sloj imena *Ground* i postavili ga na sve objekte koji predstavljaju tlo na našoj mapi te smo kreirali prazan objekt koji će projicirati sferu unutar *Player* objekta i pozicionirali ga na dno igrača.

Potrebno je i provjeriti nalazi li se objekt iznad igrača jer u slučaju da igrač čuči i postoji objekt iznad njegove glave mora mu se onemogućiti ustajanje. To smo implementirali koristeći *raycast* tehnologiju koja projicira nevidljivu zraku od igračeve pozicije do određene udaljenosti iznad igrača i vraća *true* ukoliko je zraka pogodila objekt. O samoj *raycast* tehnologiji govorit ćemo kod implementacije pucanja.

```
public CharacterController PlayerCharacterController;
    public Transform PlayerPosition;
    public float MovementSpeed = 6f;
    public float Gravity = -19.62f;
    public float JumpHeight = 2f;
```

```
    public float PlayerHeight = 3f;
    public float CrouchedPlayerHeight = 0.5f;

    public Transform GroundCheck;
    public float GroundCheckDistance = 0.4f;
    public LayerMask GroundLayer;

    Vector3 Velocity;
    bool IsOnGround = false;
    bool IsCrouched = false;
    bool ObjectAboveHead = false;

    void Update()
    {

        IsOnGround = Physics.CheckSphere(GroundCheck.position,
GroundCheckDistance, GroundLayer);
        ObjectAboveHead = Physics.Raycast(PlayerPosition.position,
Vector3.up, 2.5f);

        if (IsOnGround && Velocity.y < 0) Velocity.y = -10f;

        Move();

        Jump();

        CrouchOrStandUp();

        Velocity.y += Gravity * Time.deltaTime;
        PlayerCharacterController.Move(Velocity * Time.deltaTime);

    }
```

4.1.2.1. Osnovno kretanje

Kretanje igrača naprijed-nazad i lijevo-desno je implementirano u metodi *Move*. U metodi se dohvaćaju podaci uneseni preko tipkovnice, konkretnije slova *W, A, S i D*. Horizontalni ulaz predstavljaju *A i D*, a vertikalni *W i S*. Kreira se novi vektor koji se množi sa brzinom kretanja i *Time.deltaTime* vremenom i igrač se pomiče. *Time.deltaTime* je interval u

sekundama između frameova i koristi se za postizanje glatkih stopa okvira (eng. *frame rate*) što pridonosi boljem korisničkom iskustvu.

```
void Move()
    {
        float x = Input.GetAxis("Horizontal");
        float z = Input.GetAxis("Vertical");

        Vector3 DirectionToMove = transform.right * x + transform.forward *
z;

        PlayerCharacterController.Move(DirectionToMove * MovementSpeed *
Time.deltaTime);
    }
```

4.1.2.2. Skakanje

Ukoliko je igrač pritisnuo tipku za skok, potrebno je provjeriti nalazi li se na tlu, je li u čučnju i nalazi li se iznad njega prepreka. Ako igrač stoji na tlu i ne nalazi se prepreka iznad njega, usmjeravamo njegovo ubrzanje prema gore, odnosno, igrača postavljamo u skok. Ako je igrač u čučnju neće moći skočiti.

```
void Jump()
    {
        if (Input.GetButtonDown("Jump"))
        {
            if(IsOnGround && !ObjectAboveHead && !IsCrouched)
            {
                Velocity.y = Mathf.Sqrt(JumpHeight * -1.5f * Gravity);
            }
        }
    }
```

4.1.2.3. Čučanj i ustajanje

Kod čučnja i ustajanja jednostavno mijenjamo visinu objekta *Player*. Ukoliko je igrač pritisnu tipku za čučanj, smanjujemo njegovu visinu i postavljamo varijablu *IsCrouched* u *true*. U slučaju da se igrač već nalazi u čučnju i iznad njega ne postoji prepreka, resetiramo njegovu visinu i postavljamo *IsCrouched* u *false*.

```
void CrouchOrStandUp()
    {
        if (Input.GetKey(KeyCode.LeftControl))
        {
            PlayerCharacterController.height = CrouchedPlayerHeight;
            IsCrouched = true;
        }
        else
        {
            if (IsCrouched && !ObjectAboveHead)
            {
                PlayerCharacterController.height = PlayerHeight;
                IsCrouched = false;
            }
        }
    }
```

4.1.2.4. Penjanje ljestvama

Penjanje ljestvama je implementirano u zasebnoj skripti jer je penjanje jedna potpuno odvojena mehanika. Kada se igrač približi ljestvama i uđe u njihov collider, aktivira se *OnTriggerEnter* funkcija koja onemogućuje standardnu skriptu kretanja i postavlja globalnu varijablu *isClimbing* na *true*. Nakon što igrač izađe iz collidera ljestvi, aktivira se funkcija *OnTriggerExit* gdje se standardna skripta za kretanje ponovo aktivira i vrijednost varijable *isClimbing* se postavlja na *false*.

```
private void OnTriggerEnter(Collider col)
    {
        Player.Instance.GetComponent<Movement>().enabled = false;
        isClimbing = true;
    }

private void OnTriggerExit(Collider col)
    {
        Player.Instance.GetComponent<Movement>().enabled = true;
        isClimbing = false;
    }
```

Glavna mehanika penjanja se odvija u *Update* metodi. Ukoliko je vrijednost *isClimbing* varijable *true*, dohvaća se unos igrača i njegova pozicija na ljestvama se pomiče. Dodatno su deklarirane globalne varijable *climbSpeed* i *climbSpeedLeftRight* za upravljanje brzinom penjanja.

```
private void Update()
    {
        if (isClimbing)
        {
            if (Input.GetKey(KeyCode.W))
            {
                player.transform.position += Vector3.up * climbSpeed *
Time.deltaTime;
            }

            if (Input.GetKey(KeyCode.S))
            {
                player.transform.position += Vector3.down * climbSpeed *
Time.deltaTime;
            }

            if(Input.GetKey(KeyCode.A))
            {
                player.transform.position += Vector3.back * climbSpeed/2 *
Time.deltaTime;
            }
            if (Input.GetKey(KeyCode.D))
            {
                player.transform.position += Vector3.forward * climbSpeed /
2 * Time.deltaTime;
            }
        }
    }
```

4.1.3. Upravljanje puškom

Unutar objekta glavne kamere smjestili smo model naše puške. Model se zove „*Sci-Fi Gun Light*“ i preuzet je s Unity Asset Store-a [10]. Također, kreirali smo još jednu kameru i nazvali je *WeaponCamera* koja se isto nalazi unutar *Main Camera* objekta. Kamera oružja prikazuje isključivo model puške dok glavna kamera ignorira pušku i pokazuje sve ostalo. Naime, bez kreiranja dodatne kamere model puške bi prolazio kroz zidove što bi stvorilo

neugodno korisničko iskustvo (eng. *clipping*). Bilo je potrebno kreirati novi sloj imena *Weapon* i postaviti ga na objekt puške, a zatim unutar *Culling Mask* svojstva glavne kamere taj sloj ukloniti. Unutar kamere oružja odabrali smo da isključivo prikazuje taj sloj.

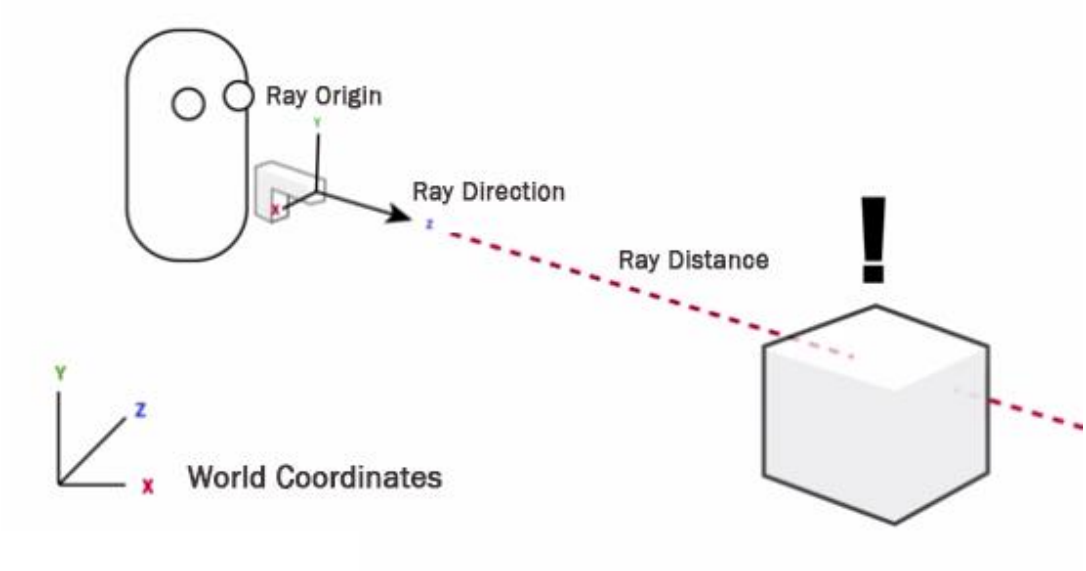
Slika 7: Kamera za oružje (Izvor: samostalna izrada)

4.1.3.1. Pucanje

Pucanje smo implementirali koristeći *raycast* tehnologiju koju smo prethodno već spomenuli, a sada ćemo je još detaljnije objasniti.

Raycasting je tehnika kojom se emitira virtualna zraka (eng. *ray*) u 3D prostoru kako bi se provjerilo što se na toj zraci nalazi ili gdje se ta zraka sudarila s objektom. Ova tehnologija je jako korisna za detekciju sudara, interakciju s objektima i određivanje položaja objekata u sceni. U kontekstu pucanja nas zanima interakcija s objektima, odnosno, zanima nas je li objekt koji je zraka dotaknula neprijatelj ili nešto drugo. *Raycasting* na primjeru pucanja funkcionira na sljedeći način:

- Definiranje polazne točke – pozicija s koje se emitira zraka, kod pucanja to je pozicija igrača ili glavne kamere
- Definiranje smjera – smjer u kojem je zraka usmjerena, kod pucanja trebamo znati gdje igrač gleda
- Projiciranje zrake
- Dohvaćanje informacija o sudaru – podaci o točki sudara se spremaju u varijablu
- Obrada sudara – ako je neprijatelj pogođen nanosi mu se šteta, a ako nije pogođen instancira se slika rupe



Slika 8: Ilustracija *raycast* tehnike (Izvor: https://medium.com/@miguel.araujo/raycast-o-que-diabos-%C3%A9-isso-e7368b6b07be)

Preostaje nam još dodati efekt i zvuk pucanja. Efekt pucanja je kreiran koristeći *Particle System*. On je zadužen za stvaranje velikog broja malih čestica koje mogu predstavljati efekte poput dima, plamena, kiše, eksplozije i sl. Unutar *Gun* objekta dodali smo novi *Particle System* objekt i podesili ga da rasprskava (eng. *burst*) žutu boju. Postavili smo ga na vrh cijevi puške i dodali mu svjetlo.

Za prikaz zvučnog efekta kreirali smo novi *Audio Source* objekt i dodali mu audio zapis efekta pucanja kao izvor.

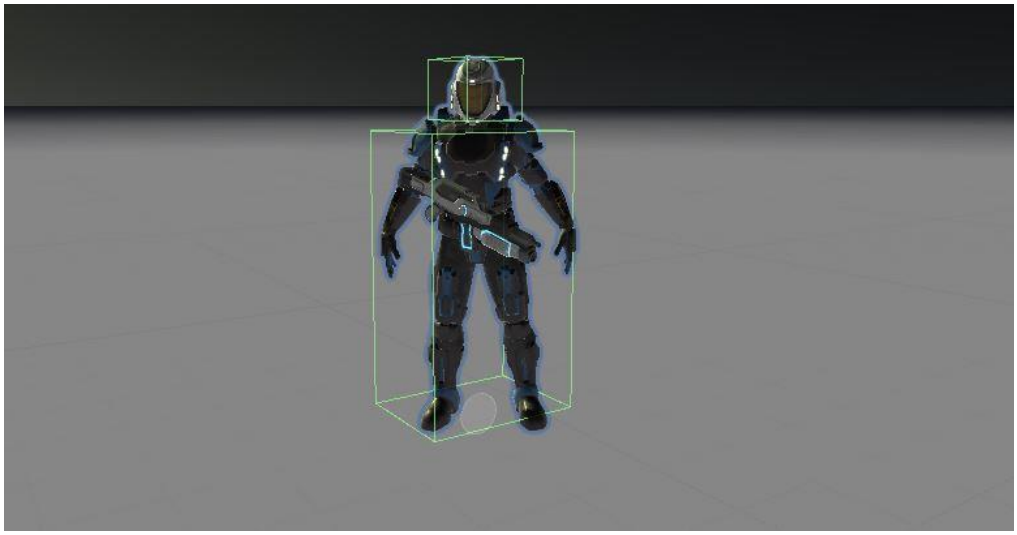
Ukoliko igrač ne pogodi neprijatelja, kreira se nova instanca slike rupe i postavlja se na pogođeni objekt, najčešće zid. Efekt pucanja i zvuk prikazujemo svaki put kada igrač pukne pušku.

Slika 9: Prikaz instance rupe od metka (Izvor: samostalna izrada)

Unutar funkcije *Shoot* smjestili smo svu logiku pucanja. Funkcija se poziva u *Update* metodi kada korisnik pritisne lijevi klik na mišu. Kada igrač pogodi neprijatelja potrebno je dohvatiti njegov objekt. Svaki objekt neprijatelja ima na sebi skriptu *Enemy* o kojoj ćemo kasnije detaljnije pričati. Za sada je dovoljno znati da se u skripti nalazi metoda *TakeDamage* koja skida zdravlje objektu neprijatelja (eng. *health points*). Također nas zanima i gdje je igrač pogodio neprijatelja kako bismo odredili koliku količinu štete mu nanijeti. Svakom neprijatelju smo dodali dva collidera (komponenta *BoxCollider*), *head collider* i *body collider*. Pogodak u glavu odmah uništava neprijatelja dok pogodak u tijelo mu nanosi štetu definiranu u *BodyDamage* varijabli.

```
void Shoot()
    {
        GunfireSound.Play();
        MuzzleFlash.Play();
        CurrentAmmo--;
        GUIManager.Instance.UpdateAmmo(CurrentAmmo, MaximumAmmo);

        RaycastHit hit;
        if(Physics.Raycast(MainCamera.transform.position,
MainCamera.transform.forward, out hit))
        {
            if(hit.transform.name == "Head Collider" || hit.transform.name
== "Body Collider")
            {

                Enemy enemy = hit.transform.parent.GetComponent<Enemy>();

                if (enemy != null)
                {
                    if (hit.transform.name == "Head Collider")
                    {
                        enemy.TakeDamage(100);
                    }
                    else
                    {
                        enemy.TakeDamage(BodyDamage);
                    }
                }
            }
```

```
            else
            {
                GameObject bullethole = Instantiate(BulletHole, hit.point,
Quaternion.LookRotation(hit.normal));
                Destroy(bullethole, 1f);
            }
        }
    }
```



Slika 10: Prikaz collidera na neprijatelju (Izvor: samostalna izrada)

4.1.3.2. Punjenje šaržera (eng. *reload*)

Kako bi znali kada i s koliko metaka napuniti pušku, kreirali smo sljedeće varijable:

- *IsReloading* – govori nam nalazi li se igrač već u ponovnom punjenju puške
- *MaximumAmmo* – definira veličinu šaržera (eng. *mag*)
- *CurrentAmmo* – prati trenutnu količinu metaka u šaržeru
- *RateOfFire* – dozvoljena brzina pucanja

Reload započinje ako je igrač pritisnuo tipku ili ako je šaržer prazan. Kreirali smo metodu *Reload* koja vraća tip *IEnumerator*. Kako bi bolje razumjeli asinkrone operacije, pojasnit ćemo koncept korutina (eng. *coroutine*).

U Unity-u, korutine zajedno s *IEnumerator* sučeljem kontroliraju asinkrone operacije. Povratni tip metode *Reload* je *IEnumerator* što nam govori da se metoda može pauzirati i nakon nekog vremena nastaviti izvoditi. Korutina nam koristi kako bi pušku mogli ponovno napuniti bez da blokiramo glavnu dretvu (eng. *thread*) igre jer punjenje traje nekoliko sekundi. Unutar *Reload* funckije koristimo *WaitForSeconds* metodu koja pauzira korutinu na određeno

vrijeme što je dovoljno da izvršimo animaciju punjenja. Unutar *Update* metode korutina se pokreće funkcijom *StartCoroutine* i prosljeđuje joj se funkcija *Reload* kao parametar.

Kako bi prikazali korisniku da se puška ponovno puni, kreirali smo posebnu animaciju. Animacija se sastoji od dva stanja, kada puška miruje i kada se puška ponovno puni. Promjenu između tih stanja obavljamo pomoću varijable *Reloading* koju postavljamo na *true* kada je punjenje započelo i na *false* kada je punjenje završilo.

Vrijeme punjenja smo definirali u posebnoj varijabli *ReloadingTime*, a nakon što se puška napuni igrač mora pričekati još četvrtinu sekunde prije nego što može pucati. Razlog je bolje korisničko iskustvo.

Slika 11: Pozicija puške dok se puni (Izvor: samostalna izrada)

```
IEnumerator Reload()
    {
        ReloadingSound.Play();

        IsReloading = true;
        ReloadingAnimator.SetBool("Reloading", true);

        yield return new WaitForSeconds(ReloadingTime - .25f);
        ReloadingAnimator.SetBool("Reloading", false);
        yield return new WaitForSeconds(.25f);

        CurrentAmmo = MaximumAmmo;
        GUIManager.Instance.UpdateAmmo(CurrentAmmo, MaximumAmmo);

        IsReloading = false;
    }
```

```
void Start ()
    {
        CurrentAmmo = MaximumAmmo;
        IsReloading = false;
        GUIManager.Instance.UpdateAmmo(CurrentAmmo, MaximumAmmo);
    }

    void Update()
    {

        if (IsReloading)
            return;

        if(((Input.GetKeyDown(KeyCode.R) && CurrentAmmo < MaximumAmmo) ||
CurrentAmmo <= 0))
        {
            StartCoroutine(Reload());
            return;
        }

        if(Input.GetButton("Fire1") && Time.time >= ShootingTime &&
GameManager.GamePaused == false)
        {
            ShootingTime = Time.time + 1f / RateOfFire;
            Shoot();
        }

    }
```

4.1.4. Upravljanje zdravljem

Našem objektu *Player* dodali smo novu skriptu imena *Player* u kojoj ćemo kontrolirati zdravlje igrača. U našoj igrici uvijek postoji samo jedan igrač, a njegovo zdravlje je resurs (varijabla) koji koristimo i u drugim skriptama. Zbog toga je klasa *Player* singleton.

U programskom inženjerstvu, *singleton* je uzorak dizajna koji ograničava instanciranje klase na jedinstvenu instancu. Konkretno, singleton uzorak dizajna omogućuje objektima da:

- Osiguraju da imaju samo jednu instancu
- Pruže jednostavan pristup toj instanci
- Kontroliraju svoje instanciranje (na primjer, skrivanje konstruktora)

Implementacije singletona osiguravaju da postoji samo jedna instanca klase i obično pružaju globalni pristup toj instanci. Uobičajeno se to postiže deklariranjem svih konstruktora klase privatnim ili pružanjem statičke metode koja vraća referencu na instancu. Instanca se obično pohranjuje kao privatna statička varijabla [11].

U Unity-u smo koristili metodu životnog ciklusa *Awake* kako bi postavili statičku varijablu *Instance* da referencira trenutnu instancu klase. U mnogim implementacijama singletona u Unity-u koristi se metoda *Awake* jer osigurava da je singleton instanca dostupna i ispravno dodijeljena prilikom inicijalizacije skripte.

```
public static Player Instance;

    void Awake()
    {
        Instance = this;
    }
```

4.1.4.1. Regeneracija

Na bitnim dijelovima razine postoje mjesta za regeneraciju zdravlja. Na zidovima se nalaze objekti kutije prve pomoći koji nestanu kada ih igrač iskoristi. Model kutije prve pomoći je preuzet s Unity Asset Store-a [12]. Iznad kutije se nalazi zeleno svjetlo.

Slika 12: Model kutije prve pomoći (Izvor: https://assetstore.unity.com/packages/3d/props/first-aid-set-160073)

Na objekte je postavljena komponenta *BoxCollider* i kreirana je posebna skripta u kojoj se pomoću funkcije *OnTriggerEnter* povećava zdravlje igrača. Unutar funkcije pozivaju se metode singleton klase *Player*. Također je dodan zvučni efekt regeneracije.

```
private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.tag == "Player" &&
(Player.Instance.CurrentHealth != Player.Instance.MaxHealth))
        {
            healingSound.Play();
            Player.Instance.RegenerateHealth(healingAmount);
            GUIManager.Instance.UpdateHealth();
            Destroy(gameObject);
        }
    }

// Metoda klase Player

public void RegenerateHealth(int healingAmount)
    {
        if (CurrentHealth + healingAmount >= MaxHealth)
        {
            CurrentHealth = MaxHealth;
        }
        else
        {
            CurrentHealth += healingAmount;
        }
    }
```

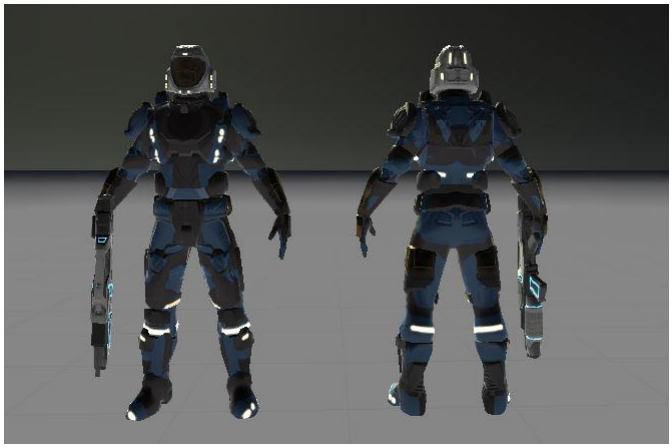
4.1.4.2. Primanje štete

Detaljnije o primanju štete ćemo objasniti u nastavku rada. Za sada je dovoljno znati da postoji metoda *TakeDamage* koja smanjuje zdravlje igrača. U slučaju da je vrijednost igračevog zdravlja nula, poziva se funkcija iz *GameManager* klase koja će završiti igru. Više o toj klasi će biti objašnjeno u zasebnom poglavlju.

```
public void TakeDamage(int damage)
    {
        if(damage >= CurrentHealth)
        {
            CurrentHealth = 0f;
            GameManager.Instance.EndGame(true);
        }
        else
        {
            CurrentHealth -= damage;
        }
}
```

4.2. Kreiranje neprijatelja

U našoj igrici neprijatelji su stacionarni objekti koji se okreću za igračem. Preuzeli smo gotov dizajn neprijatelja s Unity trgovine i ubacili u naš projekt [13]. Mehanika pucanja je implementirana na isti način kao kod igrača, a to je *raycasting* tehnologija pa o samom pucanju nećemo puno govoriti. Na početku smo kreirali novu skriptu imena *Enemy* i dodijelili svakom objektu neprijatelja. U nastavku ćemo pričati o tome kako neprijatelj nanosi štetu.



Slika 13: Model neprijatelja (Izvor: samostalna izrada)

4.2.1. Praćenje igrača

Kao što smo prije spomenuli kod igrača, *raycasting* tehnologija emitira zraku u smjeru u kojem igrač gleda, u ovom slučaju je to smjer gdje neprijatelj gleda. Zbog toga je nužno da neprijatelj konstantno gleda u igrača kako bi zraka pogodila igrača.

Unutar *Enemy* skripte kreirali smo novu metodu *LockOnPlayer* koja se poziva u *Update* metodi što omogućuje da neprijatelj prati igrača svaki *frame*. Metoda kalkulira smjer između neprijatelja i igrača i rotira objekt prema X-osi. Dodali smo i globalnu varijablu *RotationSpeed* za kontroliranje brzine okretanja i postavili je na veliku vrijednost.

```
void LockOnPlayer()
    {
        Vector3 direction = PlayerPos.position - transform.position;
        direction.y = 0;

        if (direction != Vector3.zero)
        {
            Quaternion targetRotation = Quaternion.LookRotation(direction);
```

```
            transform.rotation =
Quaternion.RotateTowards(transform.rotation, targetRotation, Time.deltaTime
* RotationSpeed);
        }
    }
```

4.2.2. Nanošenje štete

Na objekt *Player* smo postavili collider kako bi neprijatelj znao koji je objekt njegova zraka pogodila. Kreirali smo metodu *DamagePlayer* koja se poziva svaki put kada neprijatelj pogodi igrača. Unutar metode smo odabrali nasumični broj između 0 i 12 koji predstavlja količinu štete. Kako neprijatelj ne bi stalno pucao i brzo uništio igrača, implementirali smo korutinu koja pauzira pucanje na nekoliko sekundi. Vrijeme pauze je nasumičan broj sekundi između 2 i 4.

```
void DamagePlayer()
    {
        int damage = (int)(Random.Range(0.0f, 12.0f));
        Player.Instance.TakeDamage(damage);
        GUIManager.Instance.UpdateHealth();
    }

IEnumerator WaitToShoot()
    {
        CanShoot = false;
        yield return new WaitForSeconds(Random.Range(2.0f, 4.0f));
        CanShoot = true;
    }
```

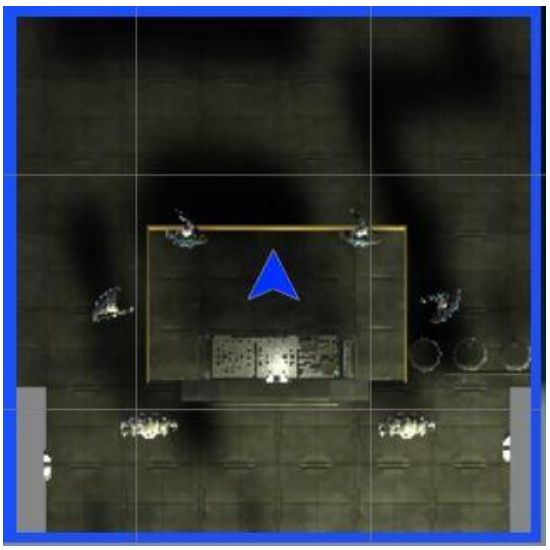
4.3. Grafičko korisničko sučelje

Grafičko sučelje služi igraču kako bi se znao navigirati se po mapi, koliko neprijatelja je još preostalo, koliko ima bodova zdravlja (eng. *health points*), koliko mu je metaka preostalo u šaržeru i pomoću točke (eng. *crosshair*) vidi gdje puca.

Slika 14: Grafičko sučelje (Izvor: samostalna izrada)

4.3.1. Mapa za navigaciju (eng. *minimap*)

Minimapa u igri predstavlja malu preglednu verziju igraćeg svijeta. Ona pruža brz i kompaktan prikaz ključnih elemenata okoline, uključujući igrača, objekte, teren i neprijatelje. Minimapa omogućava igračima da se lakše orijentiraju u svijetu igre, planiraju putanje, lociraju važne točke interesa i prate dinamiku igre iz ptičje perspektive.



Slika 15: Minimap (Izvor: samostalna izrada)

4.3.1.1. Dodavanje sekundarne kamere

Kako bi stvorili minimapu potrebna nam je sekundarna kamera koja će gledati na mapu od gore prema dolje (eng. *top-down view*). Unutar hijerarhije kreiramo novi *Camera* objekt, postavljamo ga na istu poziciju gdje se nalazi igrač, pomičemo ju prema „gore“ i rotiramo je za 90 stupnjeva kako bi gledala prema „dolje“.

4.3.1.2. Kreiranje teksture za prikaz pogleda kamere

Tekstura za prikaz (eng. *render texture*) je tekstura koja se može ažurirati. U našem slučaju, sekundarna kamera će teksturu konstantno ažurirati onim što vidi. Zatim ćemo tu teksturu mi moći prikazati na sučelju igre preko objekta *RawImage*.

Unutar hijerarhije kreirali smo novi objekt tipa *RawImage* i unutar prikaza scene smo ga pozicionirali u gornji lijevi kut zaslona te mu podesili visinu i širinu. Unutar *Project* kartice kreirali smo novi objekt tipa *RenderTexture* i imenovali ga *MinimapRenderTexture*. Visinu i širinu smo postavili istu kao i kod *RawImage* objekta.

Kako bi spojili kameru s teksturom, odabrali smo objekt kamere i kod svojstva *Target Texture* odabrali *MinimapRenderTexture*. Još je preostalo objektu *RawImage* postaviti izvor teksturu *MinimapRenderTexture* što smo napravili preko svojstva *Texture*.

4.3.1.3. Praćenje igrača

Trenutno naša sekundarna kamera ne prati igrača. Kako bi to popravili napravili smo novu skriptu *Minimap*. Koristimo metodu *LateUpdate* koja se poziva nakon *Update* i *FixedUpdate* metoda tako da ažuriramo minimapu tek nakon što se igrač pomaknuo. U metodi zapisujemo poziciju igrača u novi vektor, ali zanemarujemo njegovu Y-poziciju kako se kamera ne bi spustila dolje. Za vrijednost pozicije kamere postavljamo kreirani vektor.

Sada naša kamera prati igrača, ali se ne rotira za igračem. Pomoću Quaterniona rotiramo kameru za 90 stupnjeva prema X-osi, koristimo igračevu rotaciju na Y-osi i za 0 stupnjeva rotiramo na Z-osi.

```
void LateUpdate()
    {
        Vector3 newPosition = player.position;
        newPosition.y = transform.position.y;
        transform.position = newPosition;
        transform.rotation = Quaternion.Euler(90f,player.eulerAngles.y,0f);
}
```

4.3.2. Status zdravlja (eng. *health bar*)

Za prikaz statusa zdravlja (eng. *health bar*) koristili smo sliku okvira [14] u *Sprite* formatu i sliku srca [15] kao dekoraciju.

Unutar *Canvas* objekta u hijerarhiji kreirali smo prazan objekt i nazvali ga *HealthBar*. Unutra smo kreirali *Image* objekt i postavili sliku okvira kao izvor slike. Isto smo napravili i sa slikom srca i pozicionirali smo je lijevo od okvira. Kako bi okvir ispunili bojom koja predstavlja status zdravlja potrebno je kreirati još jedan *Image* objekt i promijeniti mu veličinu i širinu tako da popuni prostor okvira. Taj objekt smo nazvali *Fill*.

Komponentu *Slider* (klizač) dodajemo na *HealthBar* objekt. Klizač je element korisničkog sučelja koji omogućuje korisnicima odabir vrijednosti unutar određenog raspona pomicanjem vizualnog pokazivača duž trake. Zatim smo za svojstvo *Fill Rect* odabrali prethodno kreirani objekt *Fill* što nam omogućuje kontroliranje popunjenosti okvira.

Za bolje razumijevanje status zdravlja dodali smo postotak zdravlja unutar okvira. Kreirali smo objekt *TextMeshPro – Text* i nazvali ga *HealthPercentage*.

4.3.2.1. GUIManager klasa

GUIManager klasu koristimo za upravljanje svim elementima korisničkog sučelja. Ova klasa je zadužena za ažuriranje status zdravlja, broja neprijatelja i broja metaka u šaržeru. Prilikom učitanja ove skripte postavlja se početna vrijednost statusa zdravlja. Kasnije ćemo u skriptu dodati ostale metode za ažuriranje drugih elemenata korisničkog sučelja. Ova klasa je također singleton.

Metoda *SetHealthbarValue* dohvaća vrijednost zdravlja iz singleton klase *Player* i postavlja vrijednost klizača i postotka zdravlja. Za ljepši prikaz statusa kreirali smo *Gradient*.

```
public void SetHealthbarValue()
    {
        slider.maxValue = Player.Instance.MaxHealth;
        slider.value = Player.Instance.CurrentHealth;
        healthText.text = slider.value.ToString();
        fill.color = healthbarGradient.Evaluate(1f);
    }
```

Metoda *UpdateHealth* je vrlo slična prethodnoj metodi i ona samo ažurira status. Ovu metodu pozivamo u skripti *Enemy* svaki put kada neprijatelj pogodi igrača.

```
public void UpdateHealth()
    {
        slider.value = Player.Instance.CurrentHealth;
        healthText.text = slider.value.ToString();
        fill.color = healthbarGradient.Evaluate(slider.normalizedValue);
    }
```

Slika 16: Status zdravlja (Izvor: samostalna izrada)

4.3.3. Broj neprijatelja i metaka

Kreirali smo metode *UpdateAmmo* i *UpdateEnemyCount*. *UpdateEnemyCount* pozivamo svaki put kada uništimo objekt neprijatelja, a *UpdateAmmo* svaki put kada igrač ispali metak. Metode jednostavno ažuriraju tekstove na sučelju.

```
public void UpdateAmmo(int ammo, int totalAmmo)
    {
        ammoText.text = ammo.ToString() + " / " + totalAmmo.ToString();
    }


public void UpdateEnemyCount()
    {
        int numOfEnemies = GameManager.Instance.NumberOfEnemies -
GameManager.Instance.NumberOfKills;
        if (numOfEnemies > 0) enemiesText.text = numOfEnemies.ToString() +
" / " + GameManager.Instance.NumberOfEnemies;
        else
            enemiesText.text = "Enemies down";
    }
```

4.4. Izbornici

U sljedećim poglavljima pojasnit ćemo načine izrade izbornika i njihove funkcionalnosti. U našoj igri nalazi se nekoliko različitih izbornika:

- Glavni izbornik (eng. *Main menu*)
- Izbornici s postavkama (eng. *Options menu*)

- Izbornik za pauzu (eng. *Pause menu*)
- Izbornici za završetak igre (eng. *End menu*)

Za početak ćemo objasniti klasu *GameManager* koja upravlja tokom igrice, a čije se metode pozivaju u izbornicima.

4.4.1. GameManager klasa

Kao što je prethodno navedeno, *GameManager* skripta upravlja tokom igrice. Ona nam služi za pokretanje igre, pauziranje, otvaranje raznih izbornika, završetak igre i prati broj živih/mrtvih neprijatelja. Ova klasa ima samo jednu instancu (singleton).

Programski kôd ove klase prikazat ćemo u sljedećim poglavljima o izbornicima.

4.4.2. Glavni izbornik (eng. *Main menu*)

Glavni izbornik je prvi izbornik koji korisnik vidi kada upali igricu. Također, ovom izborniku se može pristupiti kada igrač odustane od igranja preko izbornika za pauzu. Za izradu glavnog izbornika kreirali smo novu scenu i postavili ju kao prvu scenu koja se učita prilikom otvaranja igrice. U glavnom izborniku se također nalazi jedan od dva izbornika s postavkama.

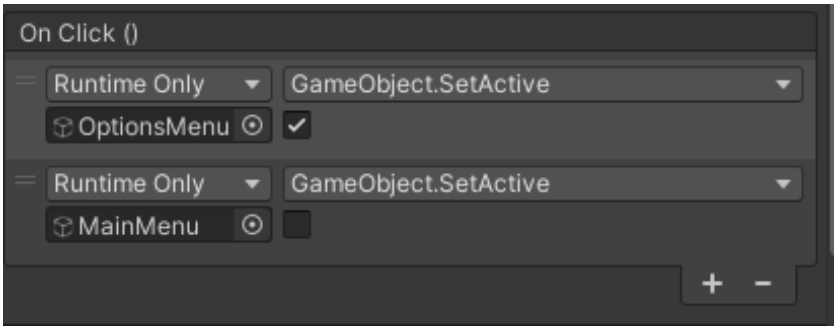
Slika 17: Glavni izbornik (Izvor: samostalna izrada)

Unutar nove scene kreirali smo prazan objekt tipa *Image* i kao izvor slike dodali sliku gradijenta koji služi kao pozadina. Zatim smo kreirali novi prazan objekt *MainMenu* i unutar njega dodali tri objekta *Button – TextMeshPro* i redom ih preimenovali. Svako dugme ima *OnClick* slušač (eng. *listener*) koji pozove odabranu funkciju kada igrač na njega klikne.

Tako smo za *Start* dugme odabrali funkciju *StartGame* iz *GameManager* klase. Ova funkcija dohvaća indeks scene koja je trenutno aktivna, a indeksiranje počinje od nule. Ovu funkciju zovemo na još nekim mjestima unutar druge scene (scena s razinom) pa nam je stoga važno znati u kojoj sceni se igrač nalazi. Pomoću *LoadScene* funkcije učitava se potrebna scena i započinje se razina.

```
public void StartGame()
    {
        int buildIndex = SceneManager.GetActiveScene().buildIndex;
        if (buildIndex == 0)
            SceneManager.LoadScene(buildIndex + 1);
        else
        {
            SceneManager.LoadScene(1);
            Time.timeScale = 1f;
            GameEnded = false;
            GamePaused = false;
            InOptionsMenu = false;
            NumberOfKills = 0;
        }
    }
```

Dugme *Options* otvara izbornik s postavkama. Kreirali smo novi prazan objekt i nazvali ga *OptionsMenu* o kojemu ćemo pričati više u nastavku. Klikom na dugme jednostavno sakrivamo glavni izbornik i aktiviramo izbornik sa postavkama.

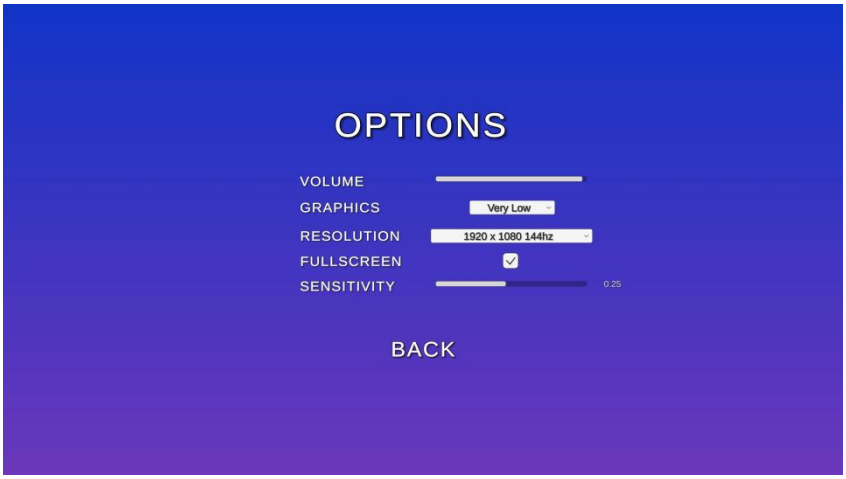


Slika 18: OnClick listener (Izvor: samostalna izrada)

Za zatvaranje igrice pozivamo funkciju *QuitGame* koja ima jednu liniju `Application.Quit();`

4.4.3. Izbornik s postavkama (eng. *Options menu*)

U ovom izborniku igrač može podesiti volumen, odabrati kvalitetu grafike, odabrati rezoluciju, odabrati prikaz preko cijelog zaslona (eng. *fullscreen*) i podesiti osjetljivost miša. Nakon što se primjene, sve ove postavke se spremaju u *PlayerPrefs* klasu kako bi korisnik imao iste postavke kad god pokrene igricu.



Slika 19: Izbornik s postavkama unutar glavnog izbornika (Izvor: samostalna izrada)

4.4.3.1. SettingsMenu klasa

SettingsMenu klasa sadrži metode za upravljanje volumenom, rezolucijama, kvalitetom grafika, prikaza igre na cijelom zaslonu i osjetljivosti miša. Za svaku od postavki postoji *Load* i *Set* metoda za učitavanje i postavljanje odabranih vrijednosti. U *Start* funkciji pozivaju se sve *Load* metode koje učitavaju vrijednosti iz *PlayerPrefs* klase (ako te vrijednosti postoje) i ažuriraju grafičko sučelje.

4.4.3.2. Podešavanje zvuka

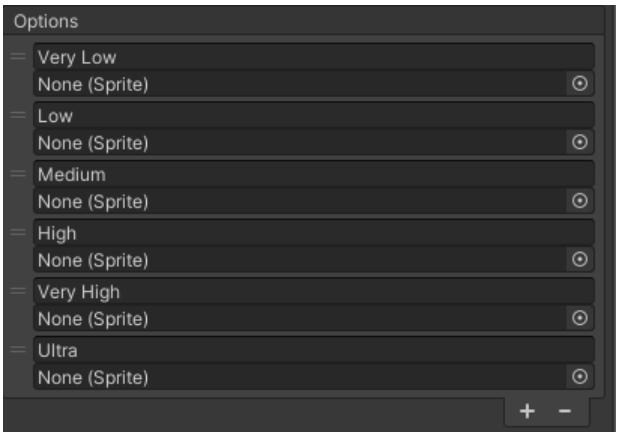
U projekt je potrebno dodati *Audio Mixer* koji upravlja zvukovima u igrici. Zatim je svim zvučnim datotekama potrebno staviti kao *Output* novi mixer. Sada će podešavanje volumena u mixeru podesiti sve zvukove u igri. Unutar mixera potrebno je parametar *Volume* izložiti (eng. *exposed parameter*) kako bi ga mogli mijenjati u našoj *SettingsMenu* skripti.

Na sučelju smo dodali klizač za volumen koji sadrži *On Value Changed* listener kojemu prosljeđujemo funkciju *SetVolume*. Funkcija kao parametar prima vrijednost klizača.

```
public void SetVolume(float volume)
    {
        volume = (float)Math.Round(volume, 2);
        audioMixer.SetFloat("volume", volume);
        PlayerPrefs.SetFloat("Volume", volume);
    }
```

4.4.3.3. Odabir grafičke kvalitete

Na sučelje dodajemo novi objekt tipa *Dropdown – TextMeshPro*. U inspektoru ručno upisujemo vrijednosti koje će se nalaziti u padajućem izborniku.



Slika 20: Popis grafičkih kvaliteta (Izvor: samostalna izrada)

On Value Changed listener-u prosljeđujemo funkciju SetVideoQuality. Funkcija kao parametar prima indeks odabrane kvalitete iz padajućeg izbornika.

```
public void SetVideoQuality(int selectedQualityIndex)
{
    PlayerPrefs.SetInt("VideoQuality", selectedQualityIndex);
    QualitySettings.SetQualityLevel(selectedQualityIndex);
}
```

4.4.3.4. Odabir rezolucije

Dostupne rezolucije prvo je potrebno učitati jer one ovise o hardveru igrača. U metodi *LoadResolutions* dohvaćaju se sve dostupne rezolucije koje se spremaju u globalno polje i

dodaju se u padajući izbornik. Zatim se u metodi *SetResolution* dohvaća indeks odabrane rezolucije iz padajućeg izbornika i postavlja se kao aktivna rezolucija.

```
public void LoadResolutions()
    {
        resolutions = Screen.resolutions;
        resolutionDropdown.ClearOptions();

        List<string> options = new List<string>();

        int currentResIndex = 0;

        for(int i = 0; i < resolutions.Length; i++)
        {
            string option = resolutions[i].width + " x " +
resolutions[i].height + " " + resolutions[i].refreshRate + "hz";
            options.Add(option);

            if (resolutions[i].width == Screen.width &&
resolutions[i].height == Screen.height) currentResIndex = i;
        }

        resolutionDropdown.AddOptions(options);
        resolutionDropdown.value = currentResIndex;
        resolutionDropdown.RefreshShownValue();
    }
public void SetResolution(int resIndex)
    {
        Resolution res = resolutions[resIndex];
        Screen.SetResolution(res.width, res.height, Screen.fullScreen);
    }
```

4.4.3.5. Postavljanje prikaza preko cijelog zaslona (eng. *fullscreen*)

Na sučelju dodajemo novi *Toggle* objekt koji poziva funkciju *SetFullscreen* kada korisnik na njega klikne. Funkcija prima varijablu tipa boolean.

```
public void SetFullscreen(bool fullscreen)
    {
        Screen.fullScreen = fullscreen;
    }
```

4.4.3.6. Postavljanje osjetljivosti miša

Za postavljanje osjetljivosti miša dodajemo još jedan klizač na korisničko sučelje. Kada god se promjeni vrijednost klizača poziva se funkcija *SetSensitivity*. S obzirom da ovu funkciju pozivamo i kada igrač podesi osjetljivost miša dok je u izborniku za pauzu, potrebno je odmah postaviti osjetljivost miša u skripti *Look* jer bi inače trebalo ponovo učitati scenu da bi se vrijednost osjetljivosti miša ažurirala.

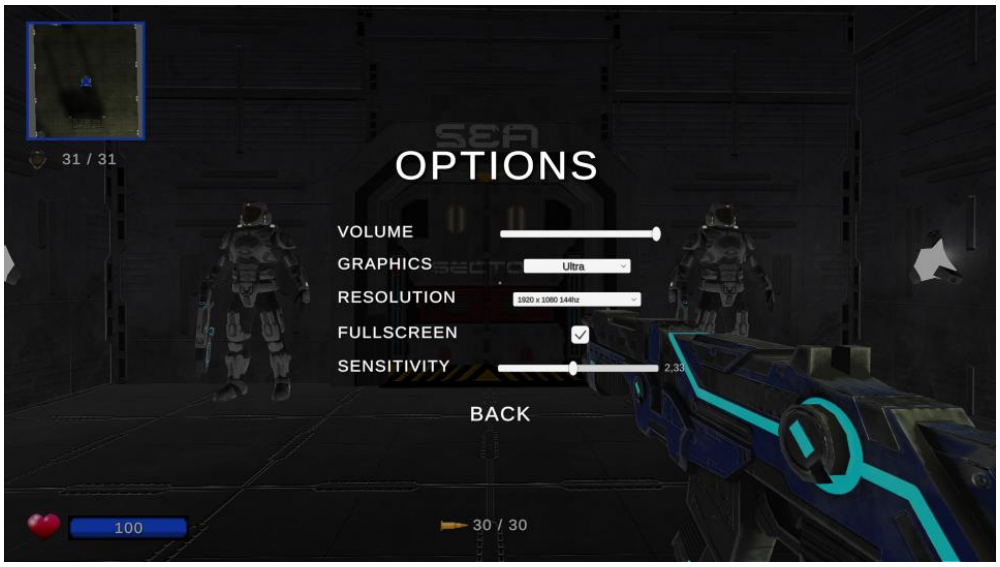
```
public void SetSensitivity(float sens)
    {
        sens = (float)Math.Round(sens, 2);
        PlayerPrefs.SetFloat("Sensitivity", sens);
        Look.Instance.Sensitivity = sens;
        sensitivityText.text = sens.ToString();
    }
```

4.4.4. Izbornik za pauzu (eng. *Pause menu*)

Igraču je potrebno omogućiti pauzirati igru i promijeniti postavke igre bez da izlazi u glavni izbornik. Pritiskom na tipku *Escape* igra se zaustavlja i otvara se izbornik za pauzu. U tom izborniku igrač može pristupiti postavkama, a može i napustiti igru.

Izbornik s postavkama ima jednake funkcionalnosti i poziva iste metode kao i u postavkama u glavnom izborniku.

Slika 21: Izbornik za pauzu (Izvor: samostalna izrada)



Slika 22: Izbornik s postavkama unutar izbornika za pauzu (Izvor: samostalna izrada)

Kako bi se igrica uistinu pauzirala, potrebno je vremensku skalu igrice (eng. *time scale*) zaustaviti. To obavljamo pomoću `Time.timeScale = 0f`. U *GameManager* skripti smo dodali i varijable tipa boolean kako bi kontrolirali u kojem izborniku se igrač nalazi (primjerice `public static bool InOptionsMenu = false;`).

4.4.5. Izbornici za završetak igre (eng. *End menu*)

Igrica može završiti na dva načina: igrač je uspješno prešao razinu ili su igrača uništili neprijatelji. Za oba slučaja kreirali smo poseban izbornik.

4.4.5.1. Izbornik za uspješan prijelaz razine

Na kraju razine nalaze se pomična vrata koja se otvaraju jedino kada igrač pobijedi sve neprijatelje. Na vratima se nalazi *collider* koji provjerava jesu li svi objekti neprijatelja uništeni. Ukoliko jesu, vrata se otvaraju i ispred njih se nalazi još jedan collider sa skriptom *Finish* koja poziva funkciju *EndGame* iz *GameManager* skripte.

EndGame funkcija prima parametar tipa boolean koji joj govori je li igrač uspješno prešao razinu. Ukoliko je parametar *false*, aktivira se završni izbornik.

Slika 23: Izbornik za uspješan prijelaz razine (Izvor: samostalna izrada)

4.4.5.2. Izbornik za neuspješan završetak razine

Na isti način je implementiran neuspješan kraj igre. Funkciji *EndGame* se prosljeđuje *true* vrijednost kao parametar (što znači da je igrač umro) i prikazuje se izbornik.

Slika 24: Izbornik za neuspješan prijelaz razine (Izvor: samostalna izrada)

```
public void EndGame(bool playerDied)
    {
        Cursor.lockState = CursorLockMode.None;
        Cursor.visible = true;
```

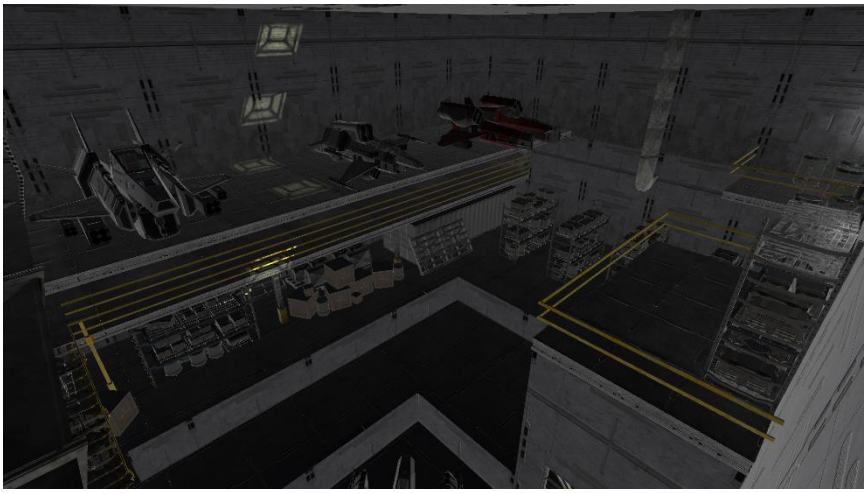
```
        Time.timeScale = 0f;
        Camera.main.GetComponent<Look>().enabled = false;
        GameEnded = true;

        if (playerDied)
        {
            YouDiedMenu.SetActive(true);
        }
        else
        {
            EndMenu.SetActive(true);
        }
    }
```

4.5. Dizajn razine

Za izradu razine koristili smo materijale preuzete sa Unity Asset Store-a [16] [17] [18] [19]. Razina se sastoji od nekoliko dijelova / soba:

- Početna soba – mjesto sa kojeg igrač započinje razinu
- Prva soba sa neprijateljima
- Streljana – u njoj se nalazi otvor za prolazak na sljedeći dio mape
- Soba sa letjelicama

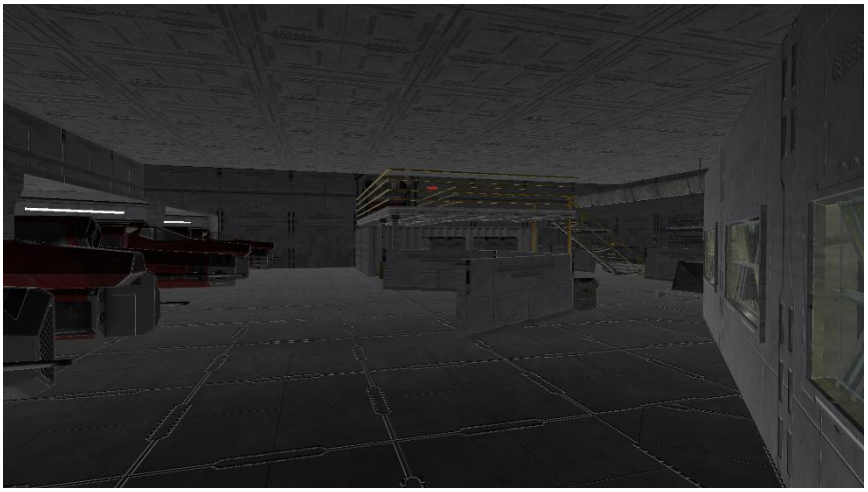
Slika 25: Razina – početna soba (Izvor: samostalna izrada)



Slika 26: Razina – prva soba s neprijateljima (1) (Izvor: samostalna izrada)

Slika 27: Razina – prva soba s neprijateljima (2) (Izvor: samostalna izrada)

Slika 28: Razina – streljana (Izvor: samostalna izrada)



Slika 29: Razina – soba s letjelicama (Izvor: samostalna izrada)

4.5.1. Prijelaz na drugi dio razine

Igrač ne može napredovati na drugi dio razine ukoliko nije pobijedio sve neprijatelje u prvom dijelu. Stoga smo dodali novu funkciju *OpenVents* unutar *GameManager* klase. Kao što je vidljivo na slici 28, na streljani se nalazi prolaz na tlu koji se otvara jedino kada je igrač pobijedio sve neprijatelje do te sobe.

Kako bi prolaz mogli otvoriti, kreirali smo novi prazan objekt i nazvali ga *Vents* (ventilacija). U objekt smo stavili model koji predstavlja otvor ventilacije, a zatim taj objekt dohvaćamo i uništavamo u programskom kôdu.

```
public void OpenVents()
    {
        ventOpeningSound.Play();
        GameObject vents = GameObject.Find("Vents");
        Destroy(vents);
    }
```

5. Zaključak

Kroz ovaj završni rad istraženi su i realizirani ključni aspekti izrade videoigre pucačine iz prvog lica u Unity engine-u. U početnom dijelu rada predstavljen je alat Unity, a istaknuto je njegovo korisničko sučelje i način rada. Opisana je kratka povijest razvoja videoigara, a usmjerena je posebna pozornost na razumijevanje žanra pucačine iz prvog lica.

U središtu pažnje rada je bio postupak izgradnje videoigre ovog žanra. Prije izrade opisana je ideja videoigre. Razvijene su mehanike gledanja, kretanja, skakanja, čučnja, pucanja i regeneracije zdravlja. Priložen je i analiziran napisani programski kôd čija je analiza omogućila dublje razumijevanje osnovnih komponenti FPS videoigre. Ovaj dio rada pruža uvid u tehničku složenost procesa razvoja videoigre. Nadalje, istraženi su poznati programski koncepti poput singleton uzorka dizajna i asinkronih operacija i opisane su njihove svrhe i primjene u radu.

Poseban segment rada je posvećen izradi raznih korisničkih izbornika unutar same igre, omogućavajući igračima prilagodbu različitih postavki poput zvuka, rezolucije i kvalitete grafike. Ovaj dio naglašava važnost adaptacije igre prema korisničkim preferencama i tehničkim mogućnostima. Na samom kraju prezentirane su slike kreirane razine.

Unity programski alat se pokazao kao jako moćan i lagan za uporabu. Zahvaljujući bogatom online sadržaju i mnoštvu gotovih komponenti i modela, programeri mogu brže stvoriti funkcionalne dijelove igre čime se ubrzava proces razvoja. Kreiranje FPS igre u Unityu predstavlja izazov koji je istovremeno i izuzetno zanimljiv. FPS igre su postale jako popularne i nastavljaju rasti, a kompetitivni aspekti potiču formiranje globalni esport zajednica i natjecanja.

Popis literature

[1] Exploding Topics, „How Many Gamers Are There? (New 2023 Statistics)“ (10.08.2023.). [Na internetu]. Dostupno: https://explodingtopics.com/blog/number-of-gamers [pristupljeno 22.08.2023.]

[2] Itch, „Most used Engines“ (bez dat.). [Na internetu]. Dostupno: https://itch.io/game-development/engines/most-projects [pristupljeno 22.08.2023.]

[3] Wikipedia, „Unity (game engine)“ (bez dat.). [Na internetu]. Dostupno: https://en.wikipedia.org/wiki/Unity_(game_engine) [pristupljeno 22.08.2023.]

[4] Udemy, „What is the Unity Game Engine?“ (bez dat.). [Na internetu]. Dostupno: https://blog.udemy.com/unity-game-engine/ [pristupljeno 23.08.2023.]

[5] Unity, „MonoBehaviour“ (bez dat.). [Na internetu]. Dostupno: https://docs.unity3d.com/ScriptReference/MonoBehaviour.html [pristupljeno 23.08.2023.]

[6] Cambridge Dictionary, „video game“ (bez dat.). [Na internetu]. Dostupno: https://dictionary.cambridge.org/dictionary/english/video-game [pristupljeno 24.08.2023.]

[7] Wikipedia, „Video game“ (bez dat.). [Na internetu]. Dostupno: https://en.wikipedia.org/wiki/Video_game [pristupljeno 24.08.2023.]

[8] Wikipedia, „History of video games“ (bez dat.). [Na internetu]. Dostupno: https://en.wikipedia.org/wiki/History_of_video_games [pristupljeno 24.08.2023.]

[9] Ultimate Pop Culture Wiki, „Shooter game“ (bez dat.). [Na internetu]. Dostupno: https://ultimatepopculture.fandom.com/wiki/Shooter_game [pristupljeno 24.08.2023.]

[10] Unity Asset Store, „Sci-Fi Gun Light“ (bez dat.). [Na internetu]. Dostupno: https://assetstore.unity.com/packages/3d/props/guns/sci-fi-gun-light-87916 [pristupljeno 11.07.2023.]

[11] Wikipedia, „Singleton pattern“ (bez dat.). [Na internetu]. Dostupno: https://en.wikipedia.org/wiki/Singleton_pattern [pristupljeno 24.08.2023.]

[12] Unity Asset Store, „First-Aid Set“ (bez dat.). [Na internetu]. Dostupno: https://assetstore.unity.com/packages/3d/props/first-aid-set-160073 [pristupljeno 18.07.2023.]

[13] Unity Asset Store, „Sci Fi Space Solider PolygonR“ (bez dat.). [Na internetu]. Dostupno: https://assetstore.unity.com/packages/3d/characters/humanoids/sci-fi/sci-fi-space-soldier-polygonr-66384 [pristupljeno 10.07.2023.]

[14] GitHub repozitorij, „Health-Bar“ (bez dat.). [Na internetu]. Dostupno: https://github.com/Brackeys/Health-Bar/blob/master/Health%20Bar/Assets/Sprites/Bar.png [pristupljeno 15.07.2023.]

[15] FreeImages, „Heart icon free photo by flaivoloka“ (bez dat.). [Na internetu]. Dostupno: https://www.freeimages.com/photo/heart-icon-1155968 [pristupljeno 15.07.2023.]

[16] Unity Asset Store, „Sci-Fi Construction Kit (Modular)“ (bez dat.). [Na internetu]. Dostupno: https://assetstore.unity.com/packages/3d/environments/sci-fi/sci-fi-construction-kit-modular-159280 [pristupljeno 09.07.2023.]

[17] Unity Asset Store, „Sci-Fi Door“ (bez dat.). [Na internetu]. Dostupno: https://assetstore.unity.com/packages/3d/environments/sci-fi/sci-fi-door-21813 [pristupljeno 09.07.2023.]

[18] Unity Asset Store, „Hi-Rez Spaceships Creator Free Sample“ (bez dat.). [Na internetu]. Dostupno: https://assetstore.unity.com/packages/3d/vehicles/space/hi-rez-spaceships-creator-free-sample-153363 [pristupljeno 09.07.2023.]

[19] Unity Asset Store, „Free Steel Ladder Pack“ (bez dat.). [Na internetu]. Dostupno: https://assetstore.unity.com/packages/3d/props/exterior/free-steel-ladder-pack-24892 [pristupljeno 09.07.2023.]

Popis slika