Izrada 2D videoigre s pogledom odozgo u programkom alatu Unity

Hajdinjak, Karlo

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Karlo Hajdinjak

IZRADA 2D VIDEOIGRE S POGLEDOM ODOZGO U PROGRAMSKOM ALATU UNITY

ZAVRŠNI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Karlo Hajdinjak

Matični broj: 0016147247

Studij: Informacijski sustavi

IZRADA 2D VIDEOIGRE S POGLEDOM ODOZGO U PROGRAMSKOM ALATU UNITY

ZAVRŠNI RAD

Mentor :

Doc. dr. sc. Mladen Konecki

Varaždin, Travanj 2023.

Karlo Hajdinjak

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

U ovom radu ćemo prikazati razvoj 2D video igre s pogledom odozgo. Koristiti ćemo pretežito Unity i Visual Studio za razvoj same igre i kreaciju koda te ćemo uz pomoć Stable Diffusion-a, Figme i Photoshopa kreirati potreban sadržaj za igru. Kroz poglavlja ćemo proći kroz sve mehanike igre od kretanja igrača, sustava oružja do micanja kamere i spremanja najboljih vremena, objašnjavajući ih, pokazivajući kod i krajnji rezultat implementacije. Kao finalan proizvod ćemo dobiti video igru koja će imati 3 razine, glavni izbornik i izbornik razina. Igrač će koristiti oružja kako bi pobijedio neprijatelje u razini te će se najbolje vrijeme pobjede razine spremiti.

Ključne riječi: Unity, videoigre, softver, razvoj videoigre, C#, 2D, Visual Studio

Sadržaj

1. Uvod

Ovaj završni rad će obraditi temu razvoja video igre u programskom alatu Unity u C# programskom jeziku. Temelj igre će biti pucačina sa pogledom odozgo sa mogućnosti mijenjanja tijela igrača za vrijeme igre. Igra će biti bazirana na razinama te je planirano napraviti tri razine povećavajuće težine u sklopu završnog rada. Glavne mehanike igre će biti mehanike kretanja, mijenjanja tijela, pucanja i skrivanja od neprijatelja. Priča igre je osmišljena ali se ne planira dodati u video igru zbog dodane kompleksnosti na već dovoljno širok projekt. U samoj igri fokus će biti na osiguravanje da su mehanike igre što elegantije i zadovoljavajuće za koristiti. Igra će imati brz tempo kada se igrač bori sa neprijateljima te kada nije u borbi igra će staviti fokus na skrivanje i izbjegavanje neprijatelja kako bi dali igraču trenutak za disanje. Igra će spremati najbolje vrijeme koje je trebalo igraču da završi određenu razinu. Time potičemo igrača da svaku razinu pokuša usavršiti kako bi je mogao što brže završiti. Igra je inspirirana igrama kao što su Hotline Miami, H3VR, Ghostrunner te Cyberpunk 2077.

Same video igre su sve veće i veće područje u IKT industriji te razvoj video igre kombinira mnogo aspekata iz raznih polja kao što su grafički dizajn, razvoj softvera, dizajna zvuka i dr. Sama industrija video igara je u 2020. godini bila vrijedna 165 milijardi dolara te vrijednost samog područja se kroz godine uzdiže [1]. Razvojem video igre se može razviti ogromna količina vještina povezanih za ta područja te se mogu primijeniti principi koji su već poznati i utvrđeni u razvoju softvera.

Zbog gore navedenih razloga, ja sam odabrao ovu temu kako bih razvio meni relevantne i zanimljive vještine. Vještine koje se najviše nadam razviti su uglavnom vezane za razvoj softvera i jednostavnih neprijateljskih agenata te korištenje alata za verzioniranje kao što su Github. Vještine koje također nadam razviti jer smatram da su korisne u industriji IKT-a su vezane za dizajn UI/UX-a koje bi bile razvijene kroz implementaciju samog korisničkog sučelja video igre.

2. Metode i tehnike rada

U ovom poglavlju ćemo opisati na kakav način smo istražili temu, kako smo proveli istraživanje te koje smo programske alate i aplikacije koristili.

Ideja za završni rad se pojavila na početku 5. semestra na kolegiju "Razvoj računalnih igara". Taj kolegij se bavi dizajnom video igara te u sklopu samog kolegija se radi u programskom alatu Unity. Kroz kolegij se stječe osnovno znanje potrebno za dizajn i implementaciju jednostavnih igara. Zbog stečenih znanja na kolegiju te želje za poboljšavanjem znanja o razvoju i implementaciji video igara i svih vještina koje dolaze sa tim smo počeli zapisivati zanimljive ideje za video igre koje bi se mogle razviti u sklopu završnog rada. Jedna od tih ideja je bila 2d pucačina s pogledom odozgo koja je dobila ime "SoulSwap". Nakon zapisivanja ideje te prijedloga mentoru, krenule su istraživačke aktivnosti vezane za razvoj video igre.

Istraživanje se sastojalo od traženja raznih literatura koje su vezane za razvoj video igara i specifičnije razvoj pucačina. Literatura nađena će se bolje pojasniti u kasnijem poglavlju. Nakon pronalaženja literature se provela analiza postojeće ideje igre kako bi se prilagodila sa stečenim znanjem iz kolegija RRI i literature. Sam razvoj aplikacije se sastojao od dodavanja par funkcionalnosti, testiranja funkcionalnosti te analize i popravka problema koji su nastali. Zbog toga možemo reći da je aplikacija nastala iterativno. Nakon što je razvijena verzija igre koja je sadržavala većinu funkcionalnosti potrebnih, igra se izgradila te je bila poslana par ljudi za testiranje. Svi koji su testirali igru su također studenti Fakulteta organizacije i informatike. Korisnici igre su napisali listu problema i bugova koji su nađeni kroz testiranje koji su se maknuli u sljedećoj verziji koja je odlučena da je bila finalna, verzija "SoulSwap v1.05".

2.1. Literatura

U ovom poglavlju ćemo kratko proći kroz literaturu koja je korištena u izradi ovog završnog rada. Literatura se uglavnom sastoji od dokumentacija za alate i pakete Unity alata te od par knjiga vezanih za dizajn video igara.

Najkorisnija literatura kroz izradu projekta je bila dokumentacija za sam Unity platformu. [2] Dokumentacija se sastoji od priručnika na web stranici koja se sastoji od dva glavna dijela, "Unity Editor Manual" i "Scripting Reference", također je i istaknut "Asset Store" koji služi da se funkcionalnost Unity Editora unaprijedi sa uključivanjem korisničkih paketa koji dodavaju ili mijenjaju aspekte Unity Editora, dodaju resure i sl. Unity Editor Manual opisuje kako koristiti sam Unity editor. Podijeljen je na sekcije kako bi bio pregledniji. Scripting Reference je sekcija dokumentacije koja služi kako bi olakšala rad sa skriptama i korištenjem specifičnih funkcija Unity engine-a. Sve sekcije su definirane za posebne verzije Unity Engine-a te su starije verzije također vidljive za pregled u slučaju da se koristi starija verzija editora. Dokumentaciju su pisali sami radnici Unity Technologies kompanije te je najpouzdaniji način da se provjeri neka informacija vezana za Unity editor i korištenje njega.

Uz samu kreaciju video igre, prije kretanja implementacije također se kreirao kratak dokument koji opisuje dizajn video igre. Za pomoć tome se koristila literatura [3]. U njoj se

opisuje mnoštvo elemenata koji su relevantni za sam dizajn video igre bez da se fokusira na određen alat kao što su primjerice Unity, Unreal i sl. Zbog fokusa na samoj implementaciji u ovom završnom radu, ova literatura se koristila kako bi se kreirala ideja same video igre prije implementacije te makar je knjiga iz 2014., stvari koje se spominjanju u njoj još uvijek vrijede u području dizajna video igara.

Zbog korištenja Stable Diffusion-a također je bilo potrebno naći implementaciju koja radi na autorovom osobnom računalu, zbog korištenja AMD grafičke kartice, korištena je implementacija Stable Diffusiona nađena na GitHub-u. [4]

Uz korištenje samih Unity paketa koji dolaze uz osnovnu instalaciju i projekt, bilo je potrebno dodati jedan paket vezan za upravljanje kamere. Cinemachine [5] je bio odabran paket za to te literatura navede je bila korištena kako bi se implementacija mogla izvršiti.

Za kreaciju vidnog polja neprijatelja se pratila kratka Youtube serija gdje se to kreira[6]. Zadnja stvar koja se koristila kod izrade rada je bilo pronalaženja statistike vrijednosti industrije video igara. To se našlo na Statisti [1], stranici koja sadrži veliku količinu statističkih podataka vezanih za ogromnu količinu kategorija.

2.2. Alati i aplikacije

U ovom poglavlju ćemo dati kratak opis koji svi alati i aplikacije su bile korištene u izradi ovog završnog rada.

2.2.1. Unity

Unity je integrirano razvojno okruženje za razvoj video igara u C# programskom jeziku. On je odabran kao platforma za razvoj video igre zbog relativno dobrog poznavanja alata u usporedbi sa drugim platformama za razvoj video igre kao što su npr. Unreal, Godot i sl. U sklopu ovog projekta je korišten kao glavni alat za razvoj video igre. Pod tim se smatra kreacija skripti, objekata, scena, mehanika te funkcionalnosti igre.

2.2.2. Visual Studio

Visual Studio je razvojno okruženje koje se može koristiti za uređivanje, "debugging" i izradu koda. On je odabran jer Visual Studio ima potporu za razvoj sa Unity-om što rezultira lakšim uređivanjem koda i "debugging"-om. U sklopu ovog projekta je korišten kao glavni uređivač koda.

2.2.3. Photoshop i Figma

Photoshop je alat za uređivanje slika kreiran od Adobe korporacija. U sklopu ovog rada Photoshop se koristio za dodatno uređivanje slika nabavljenih za rad kako bi se bolje pasale u stil same video igre. Figma je kolaborativan alat za dizajn. Figma se pretežito koristila za razne

elemente korisničkog sučelja te za inicjalne dizajne ekrana igre. U sklopu projekta su oba alata korištena za svrhe dizajna korisničkog sučelja i drugih vizualnih dijelova igre.

2.2.4. Stable Diffusion

Stable Diffusion je generativan AI otvorenog izvora za generiranje slika. On se koristi za svrhe generiranja slika manje bitnih dijelova igre kao što su slike poda, zida i sl. Također se koristio za pozadinske slike na izbornicima. Implementacija korištena se nalazi na GitHub-u [4].

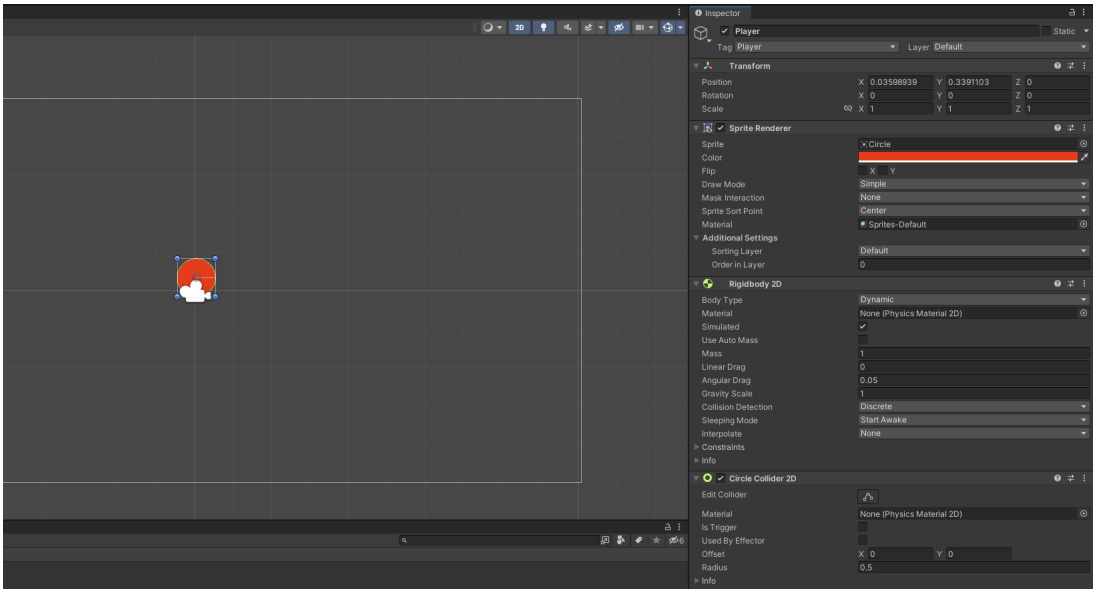
3. Razrada teme

U ovom poglavlju ćemo krenuti sa objašnjavanjem procesa razvoja same igre te pokazivanjem koda. Poglavlje je podijeljen u sekcije u kojem ćemo pokazati kod vezan za određenu mehaniku igre, pokazati što je bilo potrebno kreirati u samom Unity Editoru. Također ćemo pokazati što smo implementirali i razlog zašto je to potrebno.

3.1. Inicijalne postavke

U ovom poglavlju ćemo ukratko spomenuti inicijalne stvari koje se moraju napraviti kako bi mogli krenuti sa kreiranjem naše video igre.

Kako bi započeli sa razvojem video igre, prvo kreiramo projekt u Unity Editoru predloška 2D. To nam automatski podešava Unity Editor, kreira inicijalne objekte za minimalnu scenu u kojoj možemo nastaviti razvoj. U kreiranoj sceni ćemo kreirati našeg igrača tako da dodamo novi objekt tipa "Sprite" te ga preimenujemo u "Player". Dodajemo mu sudarač i komponentu koja će simulirati fiziku na njemu. Također dodajemo par objekta koji će koristiti kao zidovi. Na njih također dodajemo sudarač ali na njih nije potrebno staviti komponentu koja simulira fiziku, "Rigidbody 2D". Sa time imamo inicijalnu razinu i igrača kojeg ćemo u sljedećem poglavlju kroz korištenje programskog koda kontrolirati. Prikazano ispod je izgled inicijalne scene. (slika 1).



Slika 1: Izgled inicijalne scene u Unity Editoru

Također moramo uvesti sve slike, zvukove i ostale elemente koje ćemo koristiti u igri. Kreiramo dvije mape za slike, mapa "Sprites" će sadržavati sve slike koje ćemo koristiti za vrijeme same igre te uključuje slike neprijatelja, objekta igrača, poda, zida i sl. Mapa "UI" će sadržavati sve slike potrebne za korisničko sučelje igre te također sadržava podmapu "Main Menu" koja sadržava sve elemente potrebne za glavni izbornik igre.

3.2. Upravljanje igrača

Kako bi igrač mogao nekako utjecati na video igru trebamo kreirati sustav koji će kreirati vezu između igrača i naše igre. Taj sustav je sustav za unos, koji je Unity Editor automatski uključio u naš projekt. Taj "Input System" [2] omogućuje nama i korisnicima da definiramo koje tipke se koriste za razne akcije. Prvo ćemo definirati koje će sve akcije igrač koristiti za vrijeme igre. Znamo da igrač se mora kretati po razini pa zbog toga definiramo tipke za kretanje po X i Y osi pošto radimo u 2D svijetu. U Unity-u svaka 2D igra je istovremeno i 3D, samo što mi ne koristimo tu treću dimenziju, Z os.

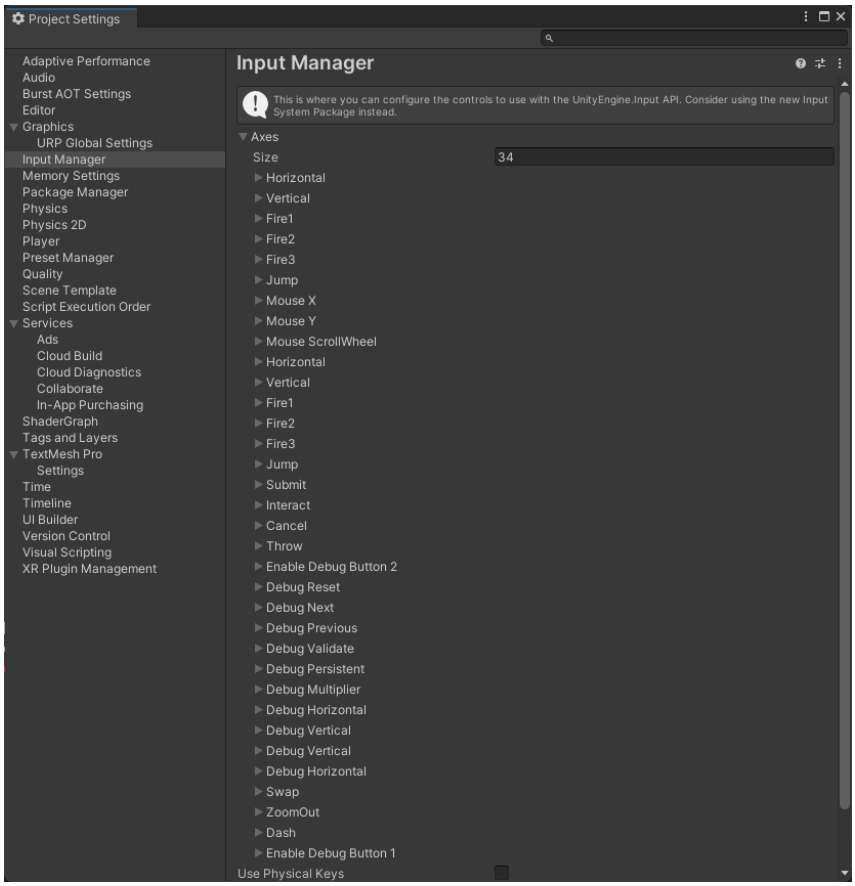
Za kretanje ćemo koristiti tipke "W", "A", "S" i "D". To je najpopularniji odabir za većinu igra u kojima se igrač mora kretati.[3] Znamo također da će igrač koristiti oružja koja moraju nekako pucati, za to će koristiti lijevi klik miša. Za pokupljanje i bacanje oružja ćemo koristiti tipku "F". Za smanjivanje kamere će igrač koristiti lijevi "ctrl" gumb. Za izmicanje ćemo koristiti lijevi "shift". Za zamjenu tijela korisnik će koristiti "space". Zadnja kontrole koju ćemo definirati su vezane za pauziranje igre, za što ćemo koristiti "esc" gumb i za ponavljanje razine ako igrač pogine, što igrač može ako koristi "R" gumb.

Nakon definiranja naše sheme kontrola možemo ih ubaciti u Unity-ov Input Manager. On se koristi za definiranje tipki i njihovih imena kako bi ih kasnije u kodu mogli koristiti za micanje i interakciju sa igrom. Također ćemo odmah u Figmi kreirati grafiku koja će biti vidljiva igraču na glavnom izborniku i kada pauzira igru koja će prikazivati kontrole igre. U Input Manageru potrebno je definirati ime naše kontrole i koji su pozitivni i negativni gumbi. Neke od kontrola samo trebaju pozitivne gumbe zato jer nemamo suprotnu akciju. Primjer akcije koja ima oboje je kretanje po X osi, možemo se kretati lijevo i desno po njoj, dok nešto kao pucanje možemo samo u jednom smjeru. Na sljedeće dvije slike je prikazan naš Input Manager te grafika iz figme (slika 2 i slika 3).

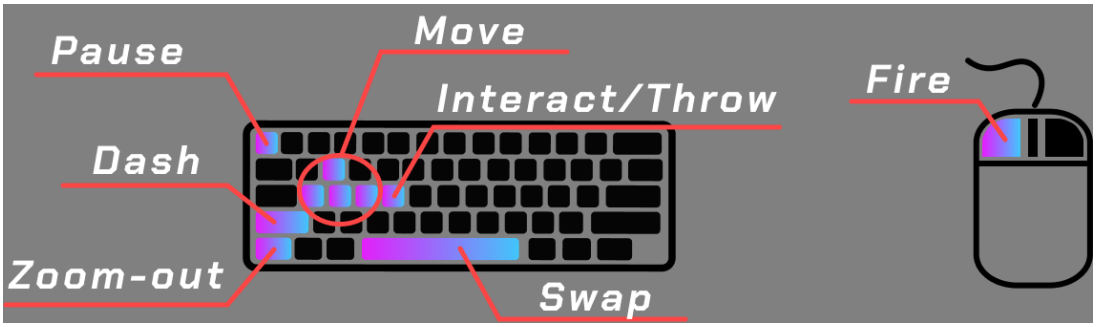
Sada kad smo definirali potrebne stvari za kontrolu igrača, možemo krenuti sa pisanjem koda. Kreirati ćemo novu skriptu sa imenom "PlayerController" i premještamo ju u novu mapu imena "Scripts". Ova skripta će se nalaziti na svakom objektu koji će igrač moći kontrolirati pa trebamo paziti da svaki objekt je ili relativno sličan ili da naša skripta ima provjeri kako bi izbjegli pogreške. Deklariramo varijable za brzinu igrača, količinu života, brzinu bacanja oružja, koliko često i daleko se može izmaknuti te varijablu koja će sadržavati objekt za simulaciju fizike nad objektom. U Start() funkciji dodjeljujemo toj varijabli vrijednost.

Sada ćemo prolaziti kroz funkcije ove klase dio po dio kako bi objasnili za što se koriste. Prvo ćemo prolaziti kroz najvažnije funkcije te zadnje funkcije koje ćemo komentirati će biti vezane za manje interakcije između igrača i okoline.

U Unity-u, svaka skripta koja se kreira automatski ima dvije glavne funkcije, Start() i Update(). Funkcija Start() se izvršava kod prvog pokretanja skripte, prije bilo koje druge funkcije te samo jednom u životu objekta [2]. Update() funkcija se izvršava svaki okvir, "frame". Mi u Start() funkciji dodjeljujemo varijablu potrebnu za rad s simulacijom fizike. U Update() funkciji provjeravamo da li igrač pokušava nešto raditi s objektom igrača, to uključuje ciljanje, korištenje oružja, pauziranje ili izmicanje. Također koristimo uz Update() i FixedUpdate(), to je funkcija



Slika 2: Postavke našeg Input Manager-a



Slika 3: Grafika kontrole igre SoulSwap

uključena u Unity te je jedina razlika između FixedUpdate() i Update() da FixedUpdate() je neovisan o broju sličica u sekundi koje igrač ima. FixedUpdate() je preporučen za bilo kakve akcije koje su vezane za fiziku [2]. U FixedUpdate() pomičemo sam objekt igrača. Za kretanje imamo funkciju PlayerMovement(), prikazana u nastavku (slika 4).

Funkcija PlayerMovement() koristi Input Manager kako bi preko funkcije "GetAxisRaw()" dobili vrijednost koliko i u kojem smjeru da mičemo igrača. Tu vrijednost množimo sa varijablom PlayerSpeed kako bi mogli mijenjati koliko je naš igrač brz te sa Time.deltaTime kako bi izbjegli da se brzina korisnika mijenja sa brojem sličica u sekundi. Nakon toga kreiramo novi trodimenzionalni vektor sa te dvije vrijednosti i pridružujemo ga našem objektu zaduženog za simulaciju fizike nad objektom.

```
void PlayerMovement()
{
    float movementHorizontal = Input.GetAxisRaw("Horizontal") * PlayerSpeed * Time.deltaTime;
    float movementVertical = Input.GetAxisRaw("Vertical") * PlayerSpeed * Time.deltaTime;
    Vector3 playerMovement = new Vector3(movementHorizontal, movementVertical, 0);
    rgd.velocity = playerMovement;
}
void PlayerTargeting()
{
    Vector2 direction = Camera.main.ScreenToWorldPoint(Input.mousePosition) - transform.position;
    float angle = Mathf.Atan2(direction.y, direction.x) * Mathf.Rad2Deg;

    Quaternion rotation = Quaternion.AngleAxis(angle - 180, Vector3.forward);
    transform.rotation = rotation;
}
```

Slika 4: Funkcije PlayerMovement() i PlayerTargeting()

Funkcija PlayerTargeting() koristimo kako bi okrenuli objekt igrača prema mišu. To radimo tako da spremamo u dvodimenzionalni vektor poziciju miša u svijetu te saznamo kut između objekta igrača i miša te rotiramo objekt igrača prema mišu za taj kut.

Sljedeća važna funkcija je funkcija PlayerWeaponHandling(), koja se koristi za interakciju sa raznim oružjima koja se nalaze u igri. Više o oružjima i samim sustavom za oružja ćemo komentirati u kasnijem poglavlju (3.3.1). U funkciji provjeravamo da li igrač već ima oružje opremljeno te ako nema provjerava da li postoji koji oružje na podu koje je dovoljno blizu da ga pokupi, ako ima onda ga uzima i koristi. To oružje se postavlja unutar tijela objekta igrača te se gasi. U slučaju da igrač već ima oružje na sebi, onda ga igrač baca u smjeru kojem je okrenut. Za bacanje se koristi ThrowWeapon() funkcija koja se također nalazi u ovoj klasi. Ona uzima ugašeno oružje, postavlja ga ispred objekta igrača i primjenjuje mu silu u smjeru u kojem smo okrenuti te silu rotacije. Povezano za opremljanje i bacanje oružja također imamo funkcije EnableWeapon() i DisableWeapon(), obje funkcije samo gase i pale određeno oružje koje se oprema ili baca.

Sljedeća funkcija koju ćemo komentirati je vezana za izmicanje, "Dash". Ona je definirana kao "IEnumerator" jer se poziva kao ko-rutina unutar igre. Za izmicanje, nakon što igrač stisne gumb za izmicanje, pokreće se ko-rutina koja pomiće objekt igrača u smjeru u kojem je bio okrenut tako dugo dok ne pređe distancu definiranu varijablom ili dok se ne sudari s nekim objektom.

Zadnje dvije funkcije koje se nalaze u klasi se odnose na primanje štete od strane neprijatelja. Te dvije funkcije su "TakeDamage()" i "Die()". Funkcija TakeDamage() jednostavno smanjuje vrijednost života igrača te poziva funkciju Die() ako igrač izgubi sav život. Kada igrač umre, brišemo ga iz svijeta te pokazujemo igraču ekran da je izgubio. Ekran je vidljiv u nastavku (slika 5).

Ova klasa je pridružena svakom objektu za kojeg je zamišljeno da igrač može kontrolirati. Sada smo kreirali način kako objekt igrača radi ali još nismo implementirali kako će igrač mijenjati tijela. To ćemo sada implementirali u sljedećem poglavlju.

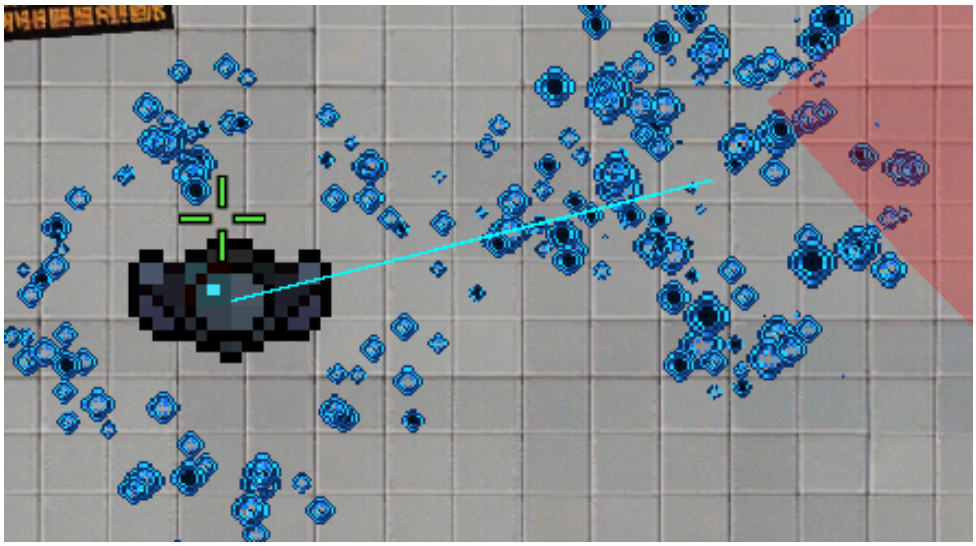
Slika 5: Što igrač vidi nakon što umre

3.2.1. Mijenjanje igrača

Glavna stvar koja dijeli ovu video igru od ostalih sličnih je da igrač može zamijeniti tijelo sa drugim objektima u razini. Kako bi to omogućili moramo kreirati dvije klase, "SwappingController" i "SwapCommunicator". SwappingController klasa će biti pridružena odvojenom objektu u razini koji ćemo nazvati "GameController". Taj objekt će upravljati svim stvarima koje trebaju biti neovisne od objekta igrača. SwapCommunicator skripta će biti pridružena svugdje gdje postoji PlayerController skripta.

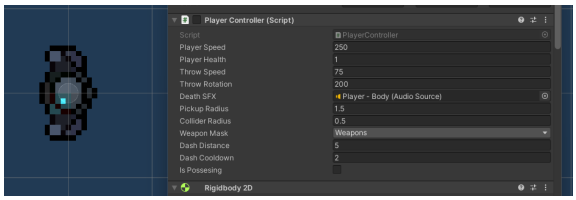
SwappingController u Update() funkciji prvo sprema objekt koji trenutno je označen kao igrač. U našoj igri igrač može biti u dva živa stanja, kao duh kada nije u nijednom tijelu te u stanju kada je u nekom tijelu. Zbog toga trebamo provjeriti u kojem je stanju te preko toga pretražiti scenu za oznake igrač ili duh igrača. Kada igrač se želi zamijeniti s nekim tijelom prvo provjeravamo da li se igrač smije zamijeniti s nekim tijelom te ako može pozivamo funkciju SwapPlayer() i onemogućujemo igraču da se zamjenjuje s nekim drugim.

Funkcija SwapPlayer sada komunicira sa funkcijom SwapCommunicator. Kako bi znali sa kim se igrač želi zamijeniti, mi provjeravamo da li je igraču miš iznad objekta koji ima mogućnost zamjene, ako je, spremamo ga u varijablu "HighlightedEnemy" u klasi SwappingCommunicator te prikazujemo igraču da se može zamijeniti sa tim objektom da aktiviramo sustav čestica koji se nalazi na tom objektu. SwappingController zatim provjerava da li postoji nešto u varijabli "HighlightedEnemy" te ako postoji onda mijenjamo aktivni objekt igrača u taj objekt. Zatim provjeravamo da li je taj objekt vidljiv. To radimo kako bi izbjegli situacije gdje igrač može preuzeti objekt koji je iza zida. To radimo sa "Raycast"-om. To je funkcionalnost Unity-a koja pokušava spojiti dvije pozicije te vraća vrijednost prvog objekta kojeg udari.[2] U našem slučaju, mi ispaljujemo raycast između trenutnog igrača i objekta s kojim se želimo zamijeniti. Ako se ništa ne nalazi između njih onda dopuštamo igraču da se zamijeni s objektom. To se radi tako da gasimo klasu PlayerController na trenutnom objektu i gasimo objekt te istovremeno palimo PlayerController na drugom objektu. U slučaju da igrač nema miš nad objektom s kojim se može mijenjati, provjeravamo da li je već u nekom objektu te ako je, onda ga izbacujemo iz tijela i vraćamo u stanje duha. U nastavku se može vidjeti vizualizacija kada igrač se želi zamijeniti sa nekim objektom (slika 6).

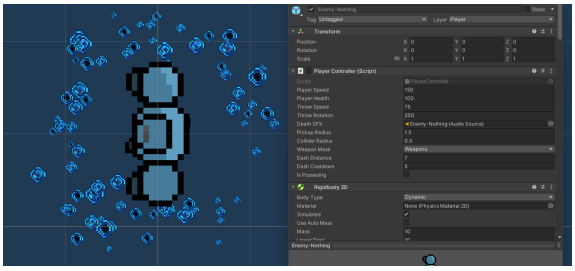


Slika 6: Vizualizacija zamjene igrača(desno) sa tijelom(lijevo), te vizualizacija raycast-a (plava linija)

Kako bi dodali varijaciju igri dodati ćemo više varijanta tijela koje igrač može obuzeti. Kreirati ćemo tri varijante kao predloške, te će svaki imati iste komponente kao i igračevo tijelo, ali sa varijacijama na brzini, izmicivanjem, količini života i s kojim oružjem se pojavljuje. U nastavku možemo vidjeti razlike u varijacijama (slike 7 i 8).

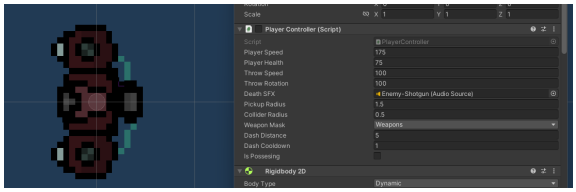


(a) Glavno tijelo

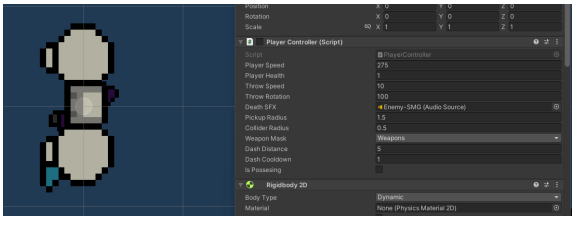


(b) Varijanta tijela 1

Slika 7: Razlike u tijelima, prvi dio.



(a) Varijanta tijela 2



(b) Varijanta tijela 3

Slika 8: Razlike u tijelima, drugi dio.

3.2.2. Praćenje kamerom i nišan

Kroz razinu je očekivano da će se korisnik kretati pa zbog tog razloga moramo kreirati način da ga pratimo kroz razinu. To ćemo napraviti preko jednog Unity paketa pod imenom

"Cinemachine". To je paket koji dodaje posebnu kameru unutar scene koja već dolazi sa mehanizmima za praćenje igrača, kretanje, podrhtavanje zaslona i još mnogo funkcionanosti [5]. Kreiramo novu skriptu "CameraController.cs" te ju pridružujemo objektu u sceni koji je neovisan o ostalima. U našem slučaju to je objekt "GameHandler". U sceni definiramo dvije camere te jednu grupu koju će kamera pratiti. U Start() funkciji pridružujemo glavnu kameru kontroli Cinemachine-a te u Update() funkciji pratimo da li korisnik želi povećati vidljivost kamere (engl. *zoom out*). Zbog te funkcionalnosti su nam potrebne dvije kamere. Jedna kamera ima smanjen pogled dok druga nema. Mijenjamo prioritet kamera ako igrač želi smanjiti te Cinemachine automatski mijenja između njih dvoje. U ovoj skripti također implementiramo i podrhtavanje zaslona kada igrač puca. To radimo da dodajemo šum kameri koji će ju tresti kada igrač puca preko javne funkcije ShakeCamera(). Ta funkcija se poziva u skripti "Weapon.cs" kada igrač puca oružje.

Druga stvar koju moramo kratko spomenuti je nišan koji je vidljiv na ekranu. To smo ostvarili tako da smo kreirali novu skriptu "MouseFollower.cs" koji neprestano čita poziciju miša na ekranu te miše sliku nišana nad njim.

3.3. Arhitektura sistema oružja

U ovom poglavlju ćemo prvo opisati kako naš sustav oružja radi, kako je očekivano da će ga igrač koristiti te ćemo zatim komentirati kod. Za ovu verziju igre je predviđeno da se kreira tri oružja, pištolj, sačmarica te automatska puška. Puške će biti postavljene po razinama te će igrač imati mogućnost pokupljanja i mijenjanja puške kada želi. Igrač neće moći obnoviti broj metaka u pušci i moći će baciti pušku prema neprijateljima kada mu nestane metaka. U slučaju da oružje pogodi neprijatelja, neprijatelj će biti ošamućen na kratko vrijeme.

U pucačinama postoji dva glavna načina kako simulirati pucanje puške. Jedan način je da se simulira fizički objekt koji reprezentira metak. U tom slučaju, igra treba kreirati svaki metak individualno i simulirati njegovu fiziku. Drugi način je poznat kao "hitscan". "Hitscan" oružja ne kreiraju pravi metak nego samo koriste raycast ili sličnu tehnologiju kako bi provjerili da li postoji putanja od igrača koji je ispucao metak do ciljnika igrača. Ako postoji putanja, odradi se što god treba kada metak dostigne cilj. Neke igre kombiniraju te dvije mehanike za različita oružja ali mi u ovoj igri ćemo koristiti prvu varijantu, gdje je metak fizički objekt. Varijanta gdje je metak fizički objekt nam pruža više mogućnosti vezanih za mehaniku igre u usporedbi sa korištenjem hitscan oružja.

Kod kreacije bilo kakvih varijanta mehanike u video igrama, treba biti oprezan kako bi izbjegli situaciju gdje je jedna mehanika ili oružje nadmoćno nad svim drugima [3]. Taj proces se zove balansiranje te ćemo ga mi sada primjeniti nad našim oružjima. U našoj igri imamo tri oružja od kojih sva tri trebaju služiti svojoj svrsi. Pištolj bi trebao biti slabiji od druga dva oružja ali trebamo biti oprezni da nije beskoristan, zbog tog razloga ćemo ga kreirati na takav način da je precizan ali da igrač ne može pucati brzo. On će biti koristan za pucanje neprijatelja od daleko kako bi igrač osigurao put do drugih oružja. Sačmarica treba biti korisna jedino na kratke udaljenosti te zbog toga će biti jako neprecizna ali ako je igrač blizu neprijatelja će imati

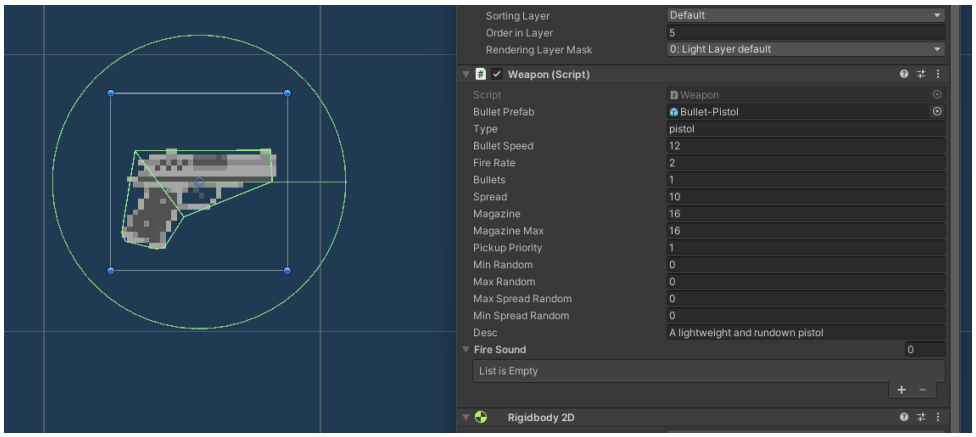
mogućnost uklanjanja neprijatelja sa jednim pucnjom. Automatska puška će se pozicionirati između pištolja i sačmarice, gdje će na srednjim udaljenostima biti najbolja opcija sa velikim količinom metka koje igrač može ispaliti s njom.

Kako bi implementirali oružja i interakcije s njima kreirati ćemo tri klase. Klasa "WeaponHandler" će sadržavati sav kod vezan za interakciju između igrača i samog oružja. Ona će biti pridružena svakom objektu koji će moći koristiti oružja. Klasa "Weapon" će biti opća definicija oružja koja će sadržavati sve funkcije našeg oružja. Klasa "Weapon" će biti pridružena svakom objektu oružja. Zadnja klasa koju trebamo je klasa "BulletHandler", koja će služiti za upravljanje ispaljenim metcima iz puški.

U klasi "WeaponHandler" trebamo znati par stvari, kada smo zadnje ispalili metak, koje oružje koristimo te od kuda pucamo. U Update() funkciji provjeravamo da li igrač pokušava ispaliti metak. Ako je objekt koji pokušava pucati igrač i ima oružje na sebi prvo provjeravamo kada je zadnje ispalio metak te provjeramo da li oružje koje ispaljuje može pucati tako brzo, ako može onda ispaljujemo metak i spremamo vrijeme kada smo ispalili metak. Također provjeravamo da li ispaljujemo sačmaricu ili drugu pušku jer sačmarica ispaljuje na malo drugačiji način od ostalih oružja. Ova klasa također mora obrađivati zahtjeve korisnika kada želi pokupiti pušku. To radi jednostavna funkcija EquipWeapon() koja samo provjerava da li igrač već ima oružje na sebi te ako nema mijenja varijablu trenutnog oružja na novo oružje. Zadnja specifična funkcija koju ova klasa sadržava je FireTurret(), koja omogućava neprijateljima da koriste oružja, više o neprijateljima u poglavlju 3.4.

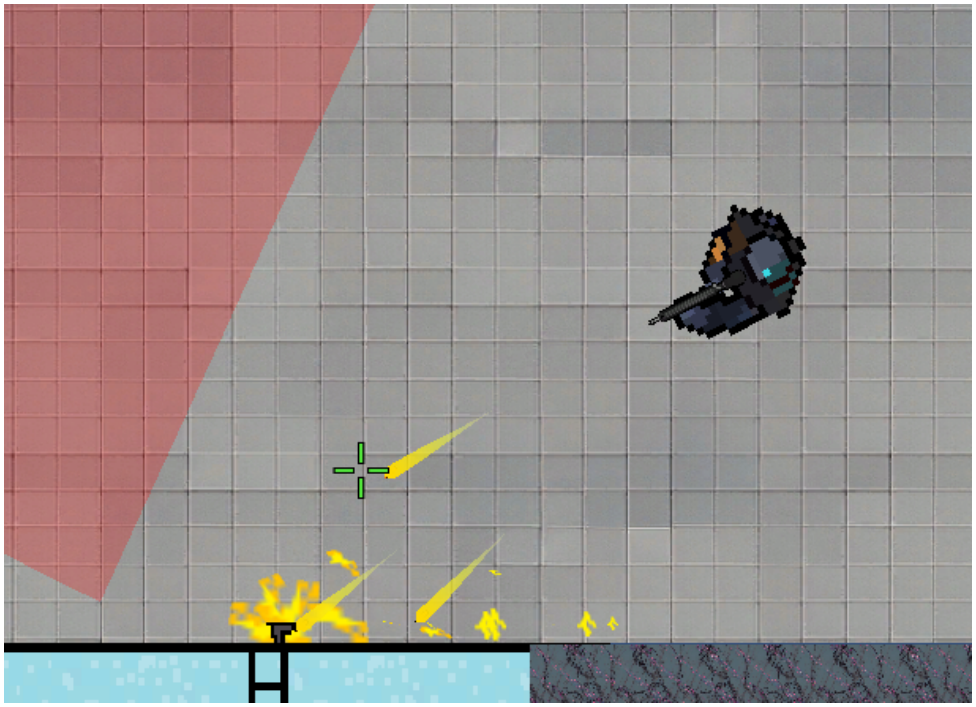
3.3.1. Oružja

Za sama oružja ćemo koristiti predloške koji Unity nudi. Ti predloški, "Prefabs" nam omogućuju da isti objekt se pojavljuje više puta u sceni. Kreiramo novi predložak u novoj mapi "Prefabs" te kreiramo novu skriptu "Weapon" i pridružujemo ju oružju. Klasa "Weapon" će sadržavati sva svojstva našeg oružja te preko varijabla možemo kreirati različite vrste oružja. Kako bi to ostvarili kreirati ćemo varijable koje možemo odmah u Unity Editoru mijenjati. To se radi tako da varijabla koristi ključnu riječ public ili da se koristi [SerializeField] ispred varijable [2]. Varijable koje ćemo mijenjati su bulletSpeed, fireRate, bullets, spread, magazine, magazineMax te varijable specifične za sačmaricu, minRandom, maxRandom, maxSpreadRandom i minSpreadRandom. Sama klasa ima samo tri funkcije koje su bitne za nas. Funkcija FireWeapon() se koristi da se ispali metak iz oružja. Ona radi tako da prvo provjeri da li oružje još ima municije te ako ima nasumično generira kut unutar određenog intervala koji definira varijabla spread. Nakon toga se metak kreira u svijetu i pomičemo ga u smjeru za brzinu koju definira bulletSpeed varijabla. Nakon ispaljivanja metka zatresemo kameru i smanjujemo broj metaka u oružju. Funkcija FireShotgun() radi na skoro identičan način, jedina je razlika da pošto sačmarica ispaljuje više metaka odjedanput ponavljamo taj proces par puta. Zadnja funkcija koja se pojavljuje u klasi je OnCollisionEnter2D koja je uključena u Unity API. Ona se aktivira kada objekt na kojem je klasa uđe u koliziju sa nekim drugim objektom. Ovdje ju koristimo kako bi ošamutili neprijatelja kada ga oružje pogodi nakon bacanja. U nastavku je prikazan predložak za pištolj sa njegovim postavkama varijabli (slika 9).



Slika 9: Predložak pištolja sa vidljivom svojstvima oružja.

Zadnja klasa koju moramo spomenuti jest "BulletHandler". Ona se nalazi na svakom ispaljenom metku. Sadrži definiciju za koliko štete radi neprijateljima i koliko dugo može postojati metak prije nego što se automatski uništi. Metak se automatski uništava nakon par sekundi kako bi se izbjegla situacija da se metci neprestano pojavljuju što bi uzrokovalo probleme sa performansama. Unutar funkcije također provjeravamo kada se sudarimo sa objektima, ako se sudarimo u neprijatelja ili igrača, smanjujemo mu život a ako se sudarimo sa zidom samo nestajemo. U nastavku možemo vidjeti rezultat našeg sustava oružja (slika 10).



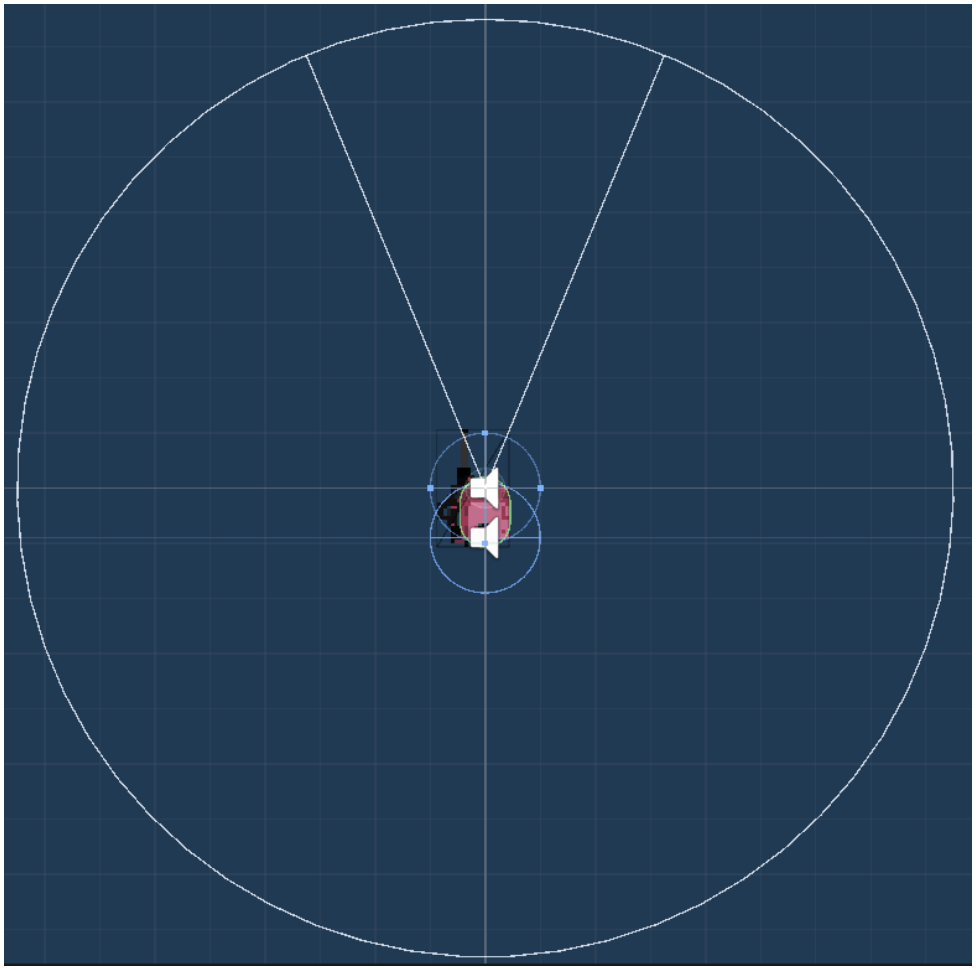
Slika 10: Igrač kako puca prema zidu.

Time smo kreirali naša oružja i sučelja potrebna za interakciju sa njima. Sada trebamo nešto na što ćemo koristiti naša oružja. Kako bi kreirali zanimljivu igru, trebamo dati igraču nekakav izazov koji mora nadvladati.[3] Za tu svrhu ćemo kreirati neprijatelja koji će pokušavati pogoditi neprijatelja kada uđe unutar njegovo polje vida. Mi ciljamo kreirati igru koja je teža za pobijediti pa ćemo morati pružiti veliki izazov igraču kada se susreće s neprijateljem. Ovdje

moramo opet koristiti akciju balansiranja. Moramo osigurati da je neprijatelj težak za pobijediti ali ne toliko težak da frustrira igrača. To ćemo postići tako da neprijatelj ima bolje oružje i više života nego igrač. To će prisiliti igrače da nađu nekakav kreativan način da iskoriste mehanike koje smo im pružili da neprijatelja nadmudre i pobjede.

3.4. Neprijatelji

Prvo kreiramo predložak za neprijatelja jer znamo da će razine imati više neprijatelja u sebi. Uz igrača, neprijatelj je najkompleksniji predložak koji postoji u igri. To je zato jer uz to da mora oponašati igrača što se tiče mehanika osim mogućnosti kretanja, također moramo dodati par elementa kako ne bi frustrirali igrača. U nastavku se može vidjeti predložak neprijatelja u Unity Editoru (slika 11).

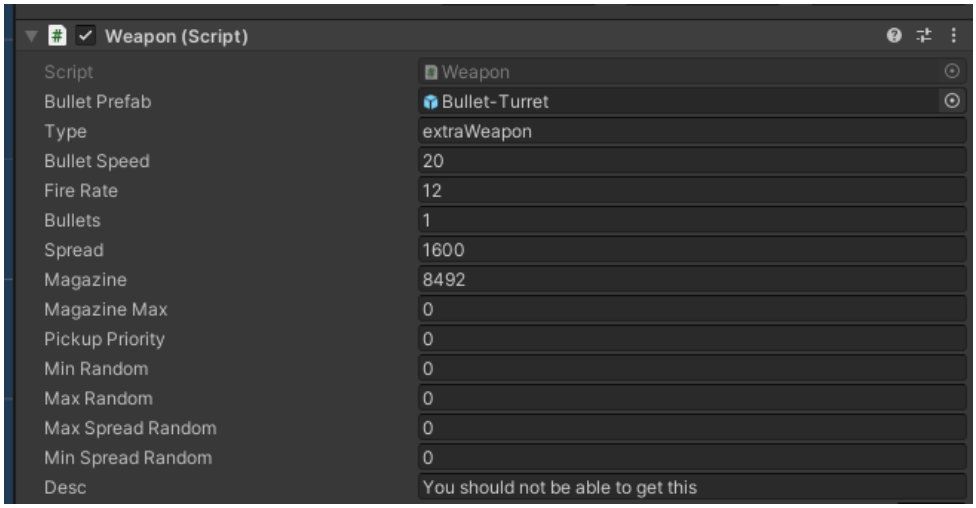


Slika 11: Predložak neprijatelja.

Igrača moramo nekako obavijestiti kakvo je vidno polje te ako je igrač unutar polja. Prvi problem je teži za rješiti te ćemo bolje ga objasniti u kasnijem poglavlju 3.4.1, ali uglavnom ćemo ga riješiti tako da iscrtavamo vidno polje neprijatelja. . Igrača možemo obavijestiti da je u vidnom polju na više načina, možemo dodati element korisničkog sučelja koji će se mijenjati ovisno da li je igrač vidljiv, možemo promjeniti tintu ekrana kada je igrač vidljiv ili još bezbroj drugih mogućih rješenja. Mi ćemo obavijestiti korisnika da je vidljiv tako da neprijatelj koji vidi

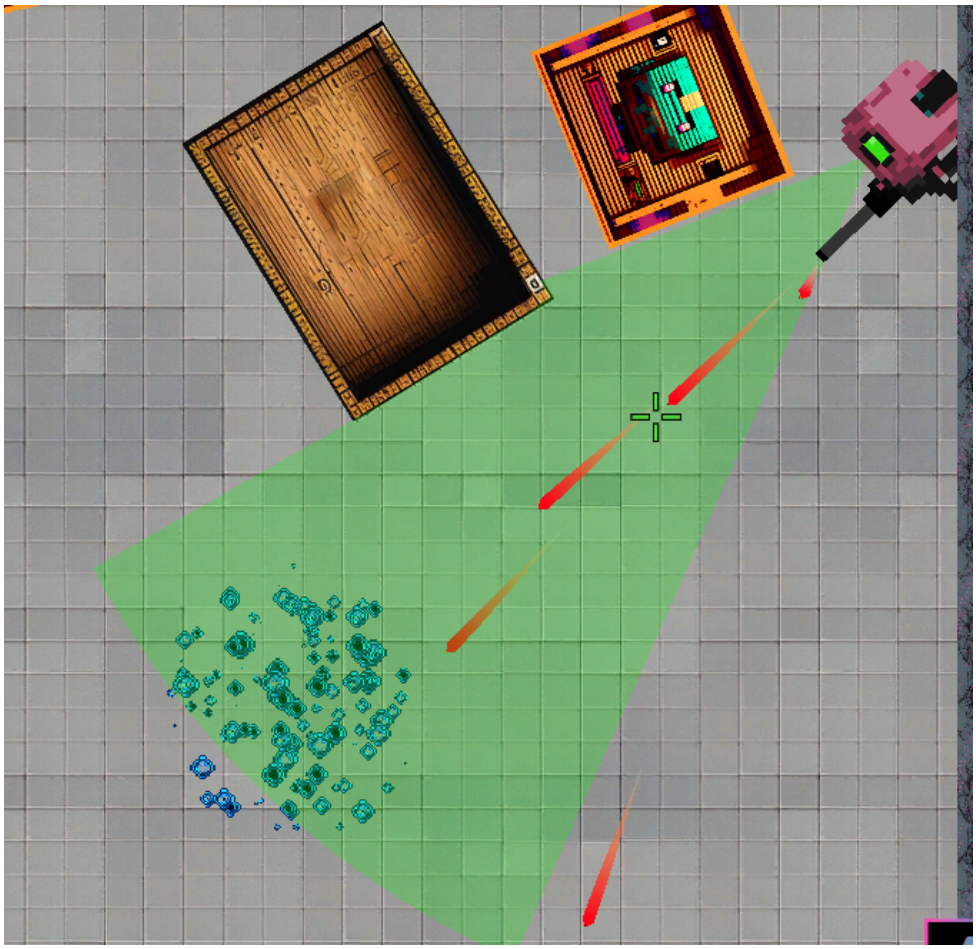
odsvira kratak zvuk. To nam ostavlja i način poboljšanja kasnije ako želimo dodati usmjereni zvuk.

Za svrhe neprijatelja kreiramo novu skriptu "EnemyTurret" i pridružujemo ju našem predlošku. U skripti trebamo kreirati par varijabli kako bi lakše balansirali neprijatelja, slično kako smo i oružja u prethodnom poglavlju 3.3.1. Te varijable su countdownAmount (koja će definirati koliko dugo igrač mora biti u vidnom polju neprijatelja prije nego počnemo pucati), turnSpeed (koja će definirati koliko brzo će neprijatelj se moći okretati) te health (koja će definirati koliko života neprijatelj ima). Neprijatelj koristi istu skriptu kao i igrač za interakciju sa oružjima ("WeaponHandler.cs") ali trebamo kreirati novo oružje za neprijatelja jer trenutno sva oružja koja imamo kreirana nebi pasala svrsi. U nastavku se mogu vidjeti svojstva tog oružja (slika 12).



Slika 12: Oružje neprijatelja.

Skripta ima par specifičnih funkcija, sve su uglavnom vezane za interakciju sa objektom igrača. Funkcija TargetPlayer() pronalazi objekt igrača u sceni te okreće neprijatelja prema njemu. Funkcija CheckPlayerInList() provjerava da li je igrač jedan od objekata u vidnom polju neprijatelja te ako je onda postavlja varijablu "playerSeen" na "true". Ta varijabla se resetira na vrijednost "false" ako igrač izađe iz vidnog polja neprijatelja. Funkcija ReduceHealth() smanjuje neprijatelju život, aktivira ju metak kada se sudari sa neprijateljem. Također moramo implementirati mehaniku da možemo ošamutiti neprijatelja kada ga bačeno oružje pogodi. To radimo preko IEnumeratora, slično kako smo implementirali mehaniku izmicanja za igrača. Kada je neprijatelj ošamućen, vidno polje mu nestaje te počinjemo odbrojavanje uz postavljanje varijable "isStunned" na "true". Nakon što vrijeme istekne, resetiramo varijablu "isStunned" te vraćamo vidljivost vidnog polja. Update() funkcija prvo poziva funkciju CheckPlayerInList() te provjerava da li je igrač vidljiv i nismo ošamućeni, ako sve valja, mijenjamo boju vidnog polja kako bi obavjestili igrača da ga vidimo te sviramo kratak zvučni signal kako bi igrač bio svjestan da ga vidimo. Započinjemo odbrojavanje te kada ističe, pucamo prema igraču. U nastavku vidimo rezultat naše implementacije (slika 13).



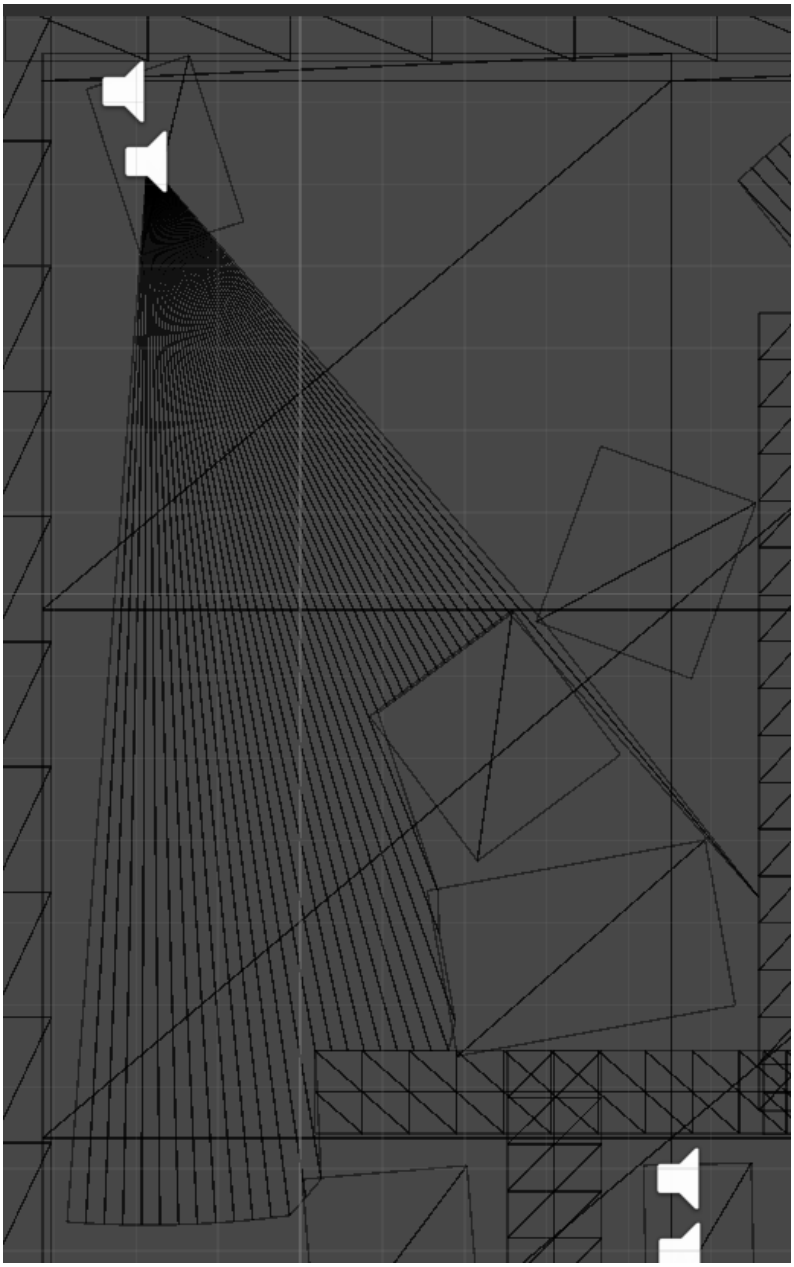
Slika 13: Neprijatelj kako puca prema igraču.

3.4.1. Vizualizacija pogleda

U ovom poglavlju ćemo bolje objasniti kako radi vizualizacija vidnog polja neprijatelja. Za kreaciju vizualizacije koristili smo kratku youtube seriju od Sebastian Lague-a. Uz pojašnjenje i dublje objašnjenje kako se vizualizacija radi preko video, također smo koristili i GitHub repozitorij koji se nudio. [6]. Kod je modificiran kako bi bolje pasao našem slučaju upotrebe.

Kreirana je nova skripta "VisionCone.cs" te je pridružena na predložak neprijatelja. Skripta radi na način da se kreira novi "mesh" koji se dinamički mijenja sa kretanjem neprijatelja. Također skripta nudi funkcionalnost da pronalazi sve objekte koji se nalaze u polju pogleda neprijatelja. Vidno polje se definira pomoću dvije varijable, radijus vidnog polja (koliko daleko doseže) i kut vidnog polja (koliko je vidno polje široko). Kako bi razlikovali između željenih objekta koje trebamo detektirati i same razine, također kreiramo dvije maske, "targetMask" i "obstacleMask". Zadnja važna stvar koju deklariramo je lista koja će sadržavati sve vidljive mete, ona se koristi u skripti "EnemyTurret" kako bi provjerili da li je objekt igrača unutar vidnog polja neprijatelja. Koristimo dvije funkcije tipa Update, Update() kako bi našli sve objekte unutar vidnog polja i LateUpdate() gdje prikazujemo vidno polje. LateUpdate() je varijanta Update() funkcije koja se izvršava nakon svih drugih funkcija tipa Update [2]. Mete nalazimo tako da pozivamo funkciju FindVisibleTargets() unutar LateUpdate()-a. Ta funkcija preklapa krug s

radijusom našeg vidnog polja. Za svaku metu unutar tog radijusa pokušavamo preko raycast-a doseći metku te ako ju možemo doseći, dodajemo ju u listu vidljivih meta. Unutar skripte također kreiramo nove tipove varijabli, ViewCastInfo i EdgeInfo. To radimo kako bi dobili više informacija o našim raycastima. Raycast u Unity-u ne vraća nama relevantne informacije pa moramo kreirati modificiranu verziju raycast-a. Iscrtavanje vidnog polja se radi tako da iz našeg objekta ispaljujemo linije koje se mogu sudariti sa svijetom. Nakon što se linije sudare sa svijetom ili dostignu rub našeg vidnog polja, kreiramo novi mesh od vrhova krajova linija. U nastavku se može vidjet žičani okvir (engl. *wireframe*) kako to izgleda (slika 14).



Slika 14: Žičani okvir vidnog polja neprijatelja.

Sa kreacijom vidnog polja neprijatelja, dovršili smo našeg neprijatelja i sa tim sve bitne mehanike koje su nam potrebne da krenemo sa kreacijom razina.

3.5. Razine

Zamišljeno je kreirati tri razine, svaka s rastućom težinom kako bi dali igraču sve veći izazov u skladu sa njegovim razumijevanjem mehanika. Fokus prve razine će biti na upoznavanje igrača sa osnovnim kontrolama micanja, interakcijama sa oružjima i izmicivanja. Druga razina će imati fokus na duhu igrača i mehaniki mijenjanja tijela. Zadnja razina treba služiti kao kraj igre te će zbog toga kombinirati sve dane mehanike te će služiti kao zadnji izazov za igrača.

U Unity-u, razine se kreiraju preko scena. Svaka scena je neovisna od drugih. Igra se nalazi u skladištu te zbog toga ćemo kreirati razine koje imaju velika otvorena područja sa kutijama koje služe kao mjesto gdje se igrač može sakriti od metaka neprijatelja. Druga razina će također sadržavati uske prolaze kroz koje igrač mora preći kako bi izbjegao neprijatelje. Svi objekti korišteni u razinama nisu kreirani kao predlošci nego su bili kreirani u sceni te kopirani radi brže kreacije razina. Svu umjetnost za objekte je generirana od strane Stable Diffusion-a na lokalnoj instalaciji.

U svakoj razini igrač počinje kao duh a pokraj njega će se nalaziti njegovo tijelo. Razina će sadržavati neki broj neprijatelja koje sve igrač mora pobijediti kako bi završio razinu. Nakon što pobjedi sve neprijatelje, može napustiti razinu. U nastavku se mogu vidjeti sve razine (slika 15, slika 16 i slika 17).

Slika 15: Razina 1.

Slika 16: Razina 2.

Slika 17: Razina 3.

3.5.1. Logika razina

Uz kreaciju rasporeda razina, također moramo kreirati funkcionalnosti kako bi igrač mogao završiti razinu. Svaka razina ima dva koraka koja igrač mora napraviti da je završi, prvo

treba pobjediti sve neprijatelje pa zatim treba doći do izlaza razine. Kako bi to implementirali kreiramo tri skripte, "GameController.cs", "EndLevel.cs" i "SpeedTimer.cs". "GameController" skripta će biti mozak operacije s kojim ostale skripte komuniciraju kako bi znali u kojoj fazi je igrač. Svaka razina ima određen broj neprijatelja pa zbog toga kreiramo varijable koje će spremati broj neprijatelja, trenutan broj neprijatelja i količina neprijatelja prije provjere. Uz te varijable također kreiramo listu koju ćemo napuniti sa svim neprijateljima koji se nalaze u sceni te dvije varijable koje prate da li je igrač izgubio ili završio razinu. U Start() funkciji brojimo koliko ima neprijatelja na razini te postavljamo relevantne varijable za to. U Update() funkciji jedino provjeravamo da li korisnik može ponovno učitati razinu ako je izgubio. U FixedUpdate() funkciji provjeravamo da li je igrač uništio kojeg neprijatelja, te ako je, mičemo ga iz liste te mijenjamo varijable vezane za praćenje toga. Ako nema više neprijatelja u razini, igrač ima mogućnost završavanja razine.

Završavanje razine se implementiralo tako da se dodao okidač kod izlaza na kojeg je pridružena skripta "EndLevel". U njoj provjeravamo da li je igrač uništio sve neprijatelje i ako je unutar okidača. Ako vrijedi oboje, spremamo vrijeme koje je bilo potrebno sa PlayerPrefs-ima [2] te vraćamo igrača na izbornik razina. Vrijeme se prati kroz skriptu "SpeedTimer.cs". U njoj kreiramo novi sat koji se automatski pokreće kad korisnik učita razinu te se zaustavlja kad ju završi. To vrijeme se zatim sprema lokalno na računalo. U nastavku se može vidjeti kod koji se izvršava nakon što igrač završi razinu (slika 18).

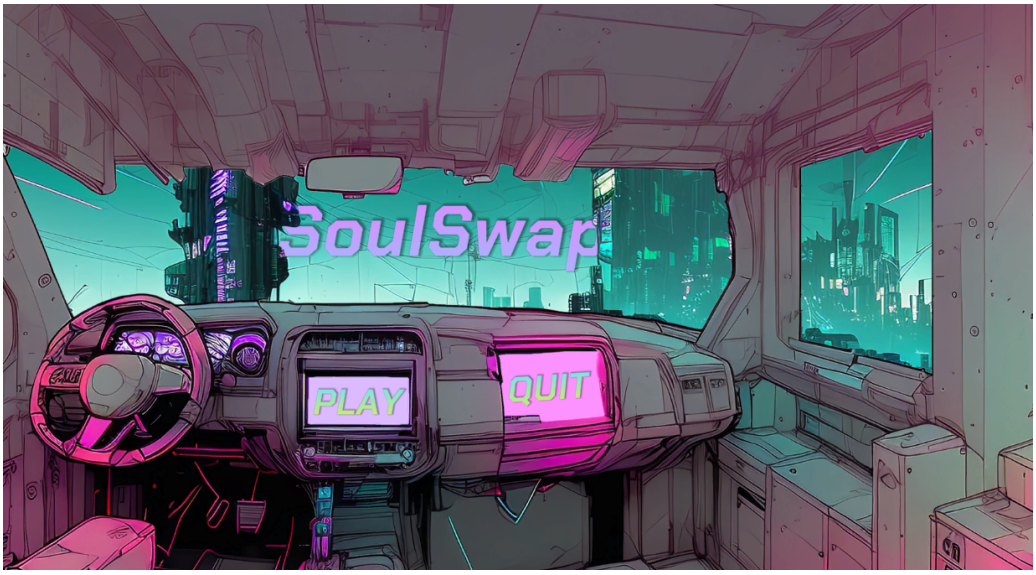
```
private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.tag == "Player" || collision.tag == "PlayerGhost")
        {
            if (isExitActive)
            {
                //end level
                Debug.Log("Level Complete");
                string currentTime = PlayerPrefs.GetString(SceneManager.GetActiveScene().name, "99:99.99");
                Debug.Log(currentTime);
                Debug.Log(String.Compare(currentTime, timerText.text));
                if (String.Compare(currentTime, timerText.text) > 0)
                {
                    PlayerPrefs.SetString(SceneManager.GetActiveScene().name, timerText.text);
                }

                audioHandler.PlayTrackWithIndex(0);
                Cursor.visible = true;
                transitionManager.LoadScene("LevelSelect");
            }
            else { Debug.Log("Not done yet"); }
        }
    }
```

Slika 18: Kod za završavanje razine.

3.6. Glavni i ostali izbornici

Kako bi igrač mogao sa lakoćom nastaviti igru i izaći iz nje, trebamo kreirati izbornike. U našoj igri kreiramo dodatne dvije scene, "MainMenu" i "LevelSelect". "MainMenu" scena će nam koristiti kako bi korisnik mogao izaći i ući u igru. Ta scena će također biti prva stvar koju korisnik vidi kada otvori igru. Za dizajn glavnog izbornika smo generirali sliku unutrašnjosti automobili iz znanstvene fantastike. Kroz njegove prozore se vidi "cyberpunk" grad u kojem lebdi ime igre. Unutar automobila postoje dva gumba, "Play", koji otvara scenu za odabir razine

i "Quit" koji gasi igru. Gumbi su elementi korisničkog sučelja koje Unity uključuje kao standardan element [2]. Također na glavnom izborniku je dodan efekt paralakse, koji daje dubinu sceni. Kako bi se to ostvarilo, kreirala se nova skripta imena "ParallaxEffect.cs". U njoj definiramo slojeve scene koje možemo zatim varirati koliko se miču u usporedbi s drugim slojevima. Ta razlika u kretanju kreira efekt dubine. Micanje se kontrolira sa pokretima miša korisnika. U nastavku se može vidjeti glavni izbornik igre (slika 19).



Slika 19: Glavni izbornik.

Druga scena za odabir razine ima efekt paralakse ali s obzirom da imamo samo jedan sloj, ne može se postići efekt dubine. Druga scena sadrži četiri gumba. Jedan u gornjem lijevom kutu kako bi se korisnik mogao vratiti na glavni izbornik te tri gumba za odabir razine. Ispod gumba se također može vidjeti najbolje vrijeme korisnika na toj razini. To vrijeme se učitava preko skripte "LevelSelectTimes.cs". Ta skripta je pridružena na platno koje sadržava sve gumbe za razine i tekstove koje sadržavaju vremena. Kroz Unity Editor su pridruženi objekti teksta te u skripti se vrijednost teksta mijenja kada se scena učita. Vrijednosti se vade preko PlayerPrefs-a [2] te ako ne postoji vrijednost, ispisuje se "00:00.0". Svi gumbi koji se koriste na korisničkim sučeljima koriste Unity-ov event system [2], ali su malo modificirani kako bi im se proširila funkcionalnost. Ta funkcionalnost se dodaje preko skripte "Button.cs". Ona proširuje gumbe tako da definira par novih funkcija. Funkcija "OnButtonClickWithMusicSwitch()" mijenja koja se muzika svira kada korisnik klikne na gumb. Funkcije "HoverEnterSFX()" i "ClickSFX()" se koriste kako bi gumb imao bolji taktilni osjećaj tako da sviramo poseban zvučni efekt kada korisnik postavi miš nad gumb ili klikne gumb. Zadnja funkcija "QuitGameButton()" je samo potrebna za gumb "Quit" na glavnom izbornikom kako bi ugasili igru kada korisnik klikne na gumb. U nastavku se može vidjeti izbornik razina (slika 20).

3.7. Zvuk

U ovom poglavlju ćemo objasniti kako je implementiran zvuk u igru. Sve audio datoteke su preuzete sa besplatnih internetskih izvora te postavljene u mapu "Audio". Za zvuk kreiramo

Slika 20: Izbornik razina.

samo jednu skriptu, "AudioHandler.cs". Ta skripta je potrebna kako bi kontrolirali zvuk i muziku između scena. Kako bi osigurali da muzika ne stane između scena, trebamo osigurati da se objekt na kojem se svira muzika ne uništi. To možemo lako ako kreiramo novi objekt u glavnom izborniku pod imenom "Music". On će sadržavati ovu skriptu te svu muziku koja može svirati. Na taj objekt pridružujemo našu skriptu te novi AudioSource [2]. To je komponenta koja služi za rukovanje sa zvučnim izvorima. U našoj skripti koristimo Awake() metodu, to je varijanta Start() metode koja se izvršava čak i kada je sama komponenta skripte ugašena [2]. Definiramo instancu ovog objekta, izvor te listu svih datoteka između kojih ćemo birati muziku. U Awake() funkciji provjeravamo da li već postoji instanca ovog objekta te ako ne, postavljamo instancu na ovaj objekt te koristimo "DontDestroyOnLoad()" funkciju kako bi osigurali da se ovaj objekt ne uništi između scena. Ako postoji objekt onda ga uništavamo kako ne bi udvostručili objekte. U Start() funkciji puštamo pjesmu sa indeksom 0. Ostale funkcije koje funkcija definira imaju veze sa sviranjem pjesme s određenim brojem indeksa. Uz te funkcije također imamo funkciju koja preko korutine zagladi mijenjanje muzike tako da se glasnoća glazbe polako smanjuje pa povećava između pjesmi (crossfade). U nastavku se može vidjeti cijela Awake() funkcija (slika 21).

Muzika je gotova međutim još moramo dodati zvuk na ostale objekte unutar igre. To se uglavnom svađa na to da na objekt koji treba imati zvuk dodajemo novu komponentu "AudioSource" te u kodu sviramo zvučni efekt dodan na komponentu. Primjerice, na sve puške je dodan novi dječji objekt imena "AudioHandler" na kojeg su se pridružili potrebni zvučni efekti (pucanje, bacanje, bez metaka i sl.) te u kodu kod pucanja puštamo taj efekt sa "m_fireSound[i].Play()" gdje je "i" broj indeksa koji zvučni efekt želimo pustiti.

```
private void Awake()
{
    if (instance == null)
    {
        instance = this;
        DontDestroyOnLoad(gameObject);
    }
    else
    {
        Debug.Log("Destroying music object");
        Destroy(gameObject);
    }
}
```

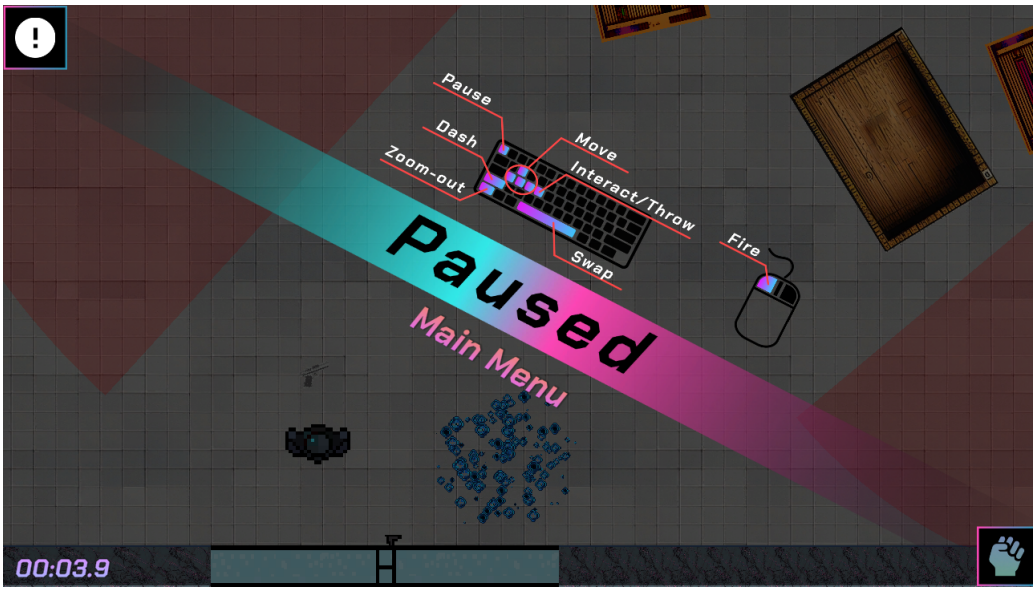
Slika 21: Kod funkcije Awake() u klasi AudioHandler.cs.

3.8. Korisničko sučelje

U ovom poglavlju ćemo proći kroz zadnju relevantnu temu u ovom djelu završnog rada. Korisničko sučelje koristimo kako bi predali informacije o svijetu i igri igraču [3]. U našoj igri imamo relativno mali broj relevantnih informacija za igrača pa će naše korisničko sučelje biti jednostavno, što je dobro jer nećemo korisnika opterećivati sa prevelikom količinom informacija [3].

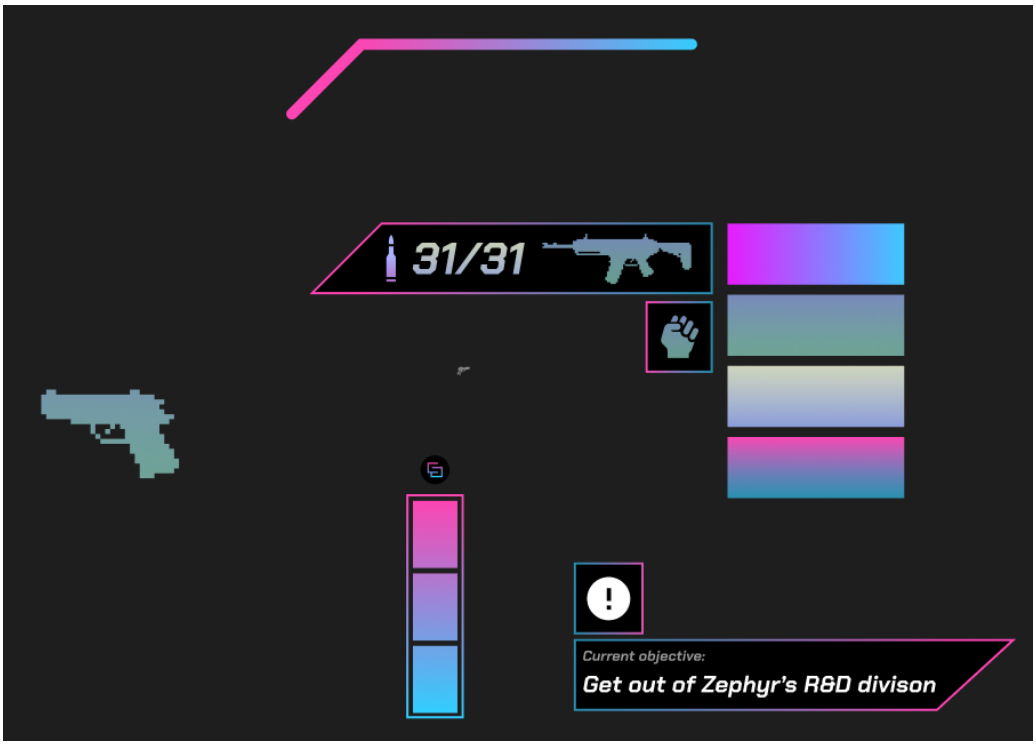
U razinama kreiramo novo platno koje će sadržavati tri stvari na sebi. Mjerač vremena kako bi korisnik znao koliko brzo ide kroz razinu, trenutni cilj i koje oružje ima. Elementi su prvo kreirani u Figmi te smo ih uveli u Unity u mapi "UI". Još dva elementa koje smo kreirali su bili poseban sloj iznad korisničkog sučelja koji bi se prikazao kada korisnik izgubi (slika 5) te još jedan koji se prikazuje kada igrač pauzira igru (slika 22).

Pauziranje se implementiralo tako da se vrijeme odvijanja igre zaustavi u potpunosti kada igrač stisne gumb za pauzu. Kako bi upravljali korisničkim sučeljem te kako bi mogli ga mijenjati kreiramo novu skriptu "UIController.cs". Ona je pridružena na novi objekt koji je neovisan od scene, "EventSystem". U skripti kreiramo varijable u koje ćemo spremati elemente korisničkog sučelja. Za svaki element sučelja ćemo kreirati novu funkciju koja će mijenjati relevantne dijelove korisničkog sučelja. Funkcija UpdateMagazineUI() mijenja se ovisno koliko metaka imamo u oružju i koje oružje imamo opremljeno. Tu funkciju zovemo iz WeaponHander, PlayerController i SwapController klasa kada se dogodi nešto što bi moglo uzrokovati promjenu oružja. UpdateObjectiveUI() mijenja trenutan cilj. Kada korisnik pobjedi neprijatelja taj element sučelja se mijenja kada se pozove funkcija. Također se taj element automatski sakriva nakon dvije sekunde kako bi igrač vidio više ekrana. Kada se prođe mišom preko elementa, on se širi i prikazuje trenutan cilj. Zadnje funkcije su za prikazivanje ona dva sloja koja smo spominjali



Slika 22: Što korisnik vidi kada pauzira igru.

ranije, togglePause() se poziva kada korisnik želi pauzirati igru, a GameOverOverlay() se poziva kada igrač izgubi. U nastavku se mogu vidjeti elementi korisničkog sučelja kada su bili kreirani u Figmi (slika 23).



Slika 23: Elementi korisničkog sučelja.

4. Zaključak

U ovom završnom radu smo prikazali razvoj 2D video igre s pogledom odozgo. Koristili smo Unity i Visual Studio za kreiranje samog koda i logike video igre dok smo koristili Figmu, Photoshop i Stable Diffusion kako bi kreirali potreban sadržaj za igru. U igri smo implementirali mnoštvo mehanika od korištenja i bacanja različitih oružja, posebnog sustava kretanja koji ima mogućnost izmicanja te glavne mehaniku igre gdje igrač može zamijeniti tijelo sa drugim objektom u sceni. Kreirali smo odgovarajuće korisničko sučelje kako bi korisnika obavijestili o statusu igre. Napravili smo glavan izbornik i scenu za odabir razine. Također smo implementirali spremanje najboljeg vremena korisnika za zasebnu razinu. Kreirali smo neprijatelje kojima smo vizualizirali vidno polje te smo implementirali i zvuk.

Kroz ovaj rad smo dokazali koliko opširno je područje samih video igri te koliko raznih disciplina ulazi u razvoj jedne relativno jednostavne igre. Također smo kao rezultat dobili jednu kompletnu video igru koja sadrži najvažnije elemente igara od mehanika, pravila i izazova.

Popis literatue

[1] *Topic: Video game industry*, srpanj 2023. adresa: https://www.statista.com/topics/868/video-games/#topicOverview.

[2] *Unity documentation*. adresa: https://docs.unity.com/.

[3] E. Adams, *Fundamentals of Shooter Game Design*. Pearson Education, 2014., ISBN: 9780133811056. adresa: https://books.google.hr/books?id=eavbBQAAQBAJ.

[4] Automatic, *GitHub - AUTOMATIC1111/stable-diffusion-webui: Stable Diffusion web UI*. adresa: https://github.com/AUTOMATIC1111/stable-diffusion-webui.

[5] U. Technologies, *Cinemachine*. adresa: https://unity.com/unity/features/editor/art-and-design/cinemachine.

[6] SebLague, *GitHub - SebLague/Field-of-View*. adresa: https://github.com/SebLague/Field-of-View.

Popis slika