

UNIVERZITET U NIŠU
PRIRODNO-MATEMATIČKI FAKULTET
DEPARTMAN ZA RAČUNARSKE NAUKE



Unity : Osnovni koncepti i razvoj 3D igre

MASTER RAD

Mentor:

dr Marko D. Petković

Student:

Miljan Mijić

Niš, 2016.

Sadržaj

Sadržaj	1
1 Uvod	3
1.1 Šta je Unity i čemu služi	3
1.2 Lokalni i globalni prostor	3
1.3 Vektori	4
1.4 Kamere	4
1.5 Temena, Ivice, Poligoni, Mesh-evi	4
1.6 Materijali, shader-i, teksture	5
1.7 Rigidbody fizika	6
1.8 Detekcija kolizije	6
1.9 Osnovni koncepti Unity tehnologije	6
2 Unity UI	8
2.1 Kreiranje novog projekta	8
2.2 Layout	10
2.3 Scene View	10
2.4 Game View	11
2.5 Hierarchy View	11
2.6 Project View	12
2.7 Inspector	12
2.8 Toolbar	13
2.9 Meniji	14
2.10 Zašto Unity?	16
3 Primer	17
4 Razvoj 3D igre	18
4.1 Ideje	18
4.2 Uvoz asset-a	18
4.3 Environment	20
4.4 Muzika	21
4.5 Glavni karakter	22
4.5.1 Kontrola animacija	22
4.5.2 Fizika i kontrola kretanja	23
4.6 Glavna kamera	25
4.7 Neprijatelj	26
4.7.1 Fizika	27
4.7.2 Audio	28
4.7.3 Kretanje	28

4.7.4	Animacija	28
4.7.5	Skripta.....	29
4.8	Glavni lik protiv neprijatelja	30
4.8.1	Energija glavnog lika.....	30
4.8.2	Napad neprijatelja.....	32
4.8.3	Energija neprijatelja	35
4.8.4	Napad glavnog lika	37
4.9	Score	42
4.10	Novi neprijatelji	44
4.10.1	Dodavanje neprijatelja iz koda	46
4.11	Završni radovi na okruženju	49
4.12	UI Meniji	50
4.12.1	Glavni meni.....	51
4.12.2	Highscore meni.....	59
5	Windows platforma.....	64
6	Android platforma.....	65
6.1	Kontrola kretanja.....	66
6.2	Facebook integracija.....	70
6.3	Optimizacija i korisničko iskustvo	77
7	Zaključak.....	79
8	Literatura.....	81

1 Uvod

Ovaj rad ima za cilj upoznavanje jedne, danas široko prihvaćene, tehnologije za razvoj igara i interaktivnih 2D i 3D aplikacija - Unity. Paralelno sa uvođenjem osnovnih koncepata Unity tehnologije, u radu je dat opis razvoja 3D igre u kojoj su akteri složeni objekti koji se kreću na sceni i međusobno interaguju. U prvom delu rada opisani su osnovni pojmovi, među kojima su: 3D prostor, koordinate i vektori, prosti i složeni objekti, materijali i teksture, detekcija sudara. Obrađeni su osnovni koncepti tehnologije koju koristimo, ideje, način rada i razvoja, i funkcionisanje same tehnologije. Dat je opis interfejsa i razvojnog okruženja, kao i osnovni elementi razvoja jedne 3D igre.

1.1 Šta je Unity i čemu služi

Unity je višeplatformski game engine. Predstavlja platformu i integrisano razvojno okruženje za razvoj 2D i 3D interaktivnih multimedijalnih aplikacija i igara za računare, konzole, mobilne uređaje i web sajtove, sa mogućnošću izvršavanja na preko 20 podržanih platformi. Jezgro Unity-a napisano je u C/C++ programskom jeziku, dok je Unity UI Editor napisan u jeziku C#. Za pristup najnižem sloju, odnosno jezgru i funkcijama Unity-a, dostupan je API za korišćenje u .NET Framework-u, i jezicima C#, Boo, ali i JavaScript. Za pisanje koda se standardno koristi Monodevelop, ali je moguće korišćenje bilo kog drugog okruženja, npr. Microsoft Visual Studio.

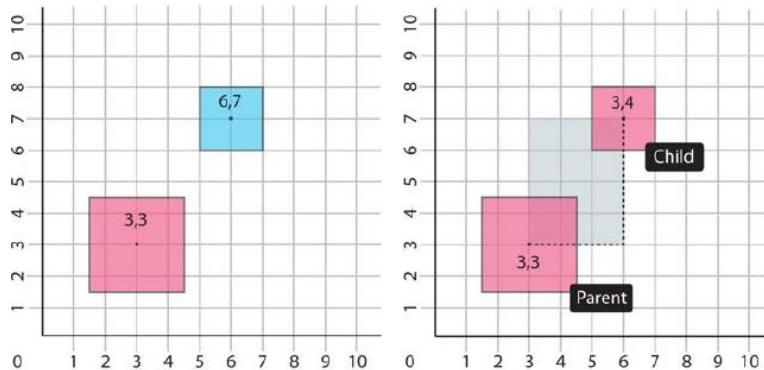
Unity razvija kompanija Unity Technologies od 2004. godine, sa vizijom da olakša i približi razvoj interaktivnih aplikacija što većem broju ljudi. Danas se može reći da je vizija ostvarena, s obzirom na podatak iz 2015. godine, da Unity ima preko milion aktivnih korisnika mesečno, i oko 4.5 miliona registrovanih. Prva verzija Unity-a objavljena je 2005. godine od strane trojice osnivača kompanije - David Helgason, Joachim Ante i Nicholas Francis.

1.2 Lokalni i globalni prostor

Svaka tačka 3D prostora ima tri koordinate, X, Y, i Z, koje nazivamo i Dekartovim koordinatama. X predstavlja horizontalnu, Y vertikalnu, a Z komponentu dubine. Kada kažemo da se tačka nalazi na poziciji (3, 5, 1), to zapravo znači da je njena X komponenta jednaka 3, Y komponenta jednaka 5, a Z komponenta jednaka 1.

U 3D prostoru uvek postoji tačka koja označava koordinatni početak (0, 0, 0), i pozicija svih objekata globalnog prostora se određuje u odnosu na ovu tačku. Međutim, da bi dodatno pojasnili i olakšali stvari, uvodimo pojam lokalnog prostora ili prostora objekta, kako bismo mogli da definišemo poziciju jednog objekta u odnosu na neki drugi objekat. Ova veza među objektima je poznata i kao roditelj-dete veza. Ovakva veza u Unity-u se lako uspostavlja povlačenjem jednog objekta na drugi. Ukoliko je objekat dete na istoj poziciji kao i roditelj, njegova pozicija je (0, 0, 0), iako globalna pozicija roditelja ne mora biti nula pozicija. Na osnovu ovakve postavke, razdaljinu između objekata računamo u lokalnom prostoru, gde za svaki dete objekat, roditelj objekat uvek ima nula poziciju.

Razliku globalnog i lokalnog prostora prikazujemo kroz primer u 2D prostoru (slika 1.1). Roditeljski objekat (crvene boje) je u oba slučaja prikazan sa globalnim koordinatama, a objekat dete je na prvoj slici prikazan sa globalnim, a na drugoj sa lokalnim koordinatama, u odnosu na roditeljski objekat.



Slika 1.1: Globalne (levo) i lokalne koordinate (desno)

1.3 Vektori

U razvoju igara često se koriste **vektori**. 3D vektore opisujemo preko Dekartovih koordinata. Oni predstavljaju veličine koje imaju pravac, smer i intenzitet. Mogu da se pomeraju u 3D prostoru, ali se time ne menjaju. Korisni su pri računanju razdaljina, relativnih uglova između objekata, itd. Unity koristi strukturu Vector3 za reprezentaciju vektora, ali i tačaka.

1.4 Kamere

Kamere su od suštinske važnosti u 3D prostoru i igrama, jer se ponašaju kao okviri za prikaz, i direktno određuju šta igrač vidi na svom ekranu. Kamere mogu da se postave na bilo kojoj poziciji u prostoru, mogu da se animiraju, ili prikače za neki pokretni objekat u igri. Na sceni može biti postavljen veći broj kamera, ali u svakom trenutku je samo jedna glavna - *Main Camera*, koja renderuje ono što igrač vidi. Ovo je razlog zašto Unity automatski dodaje *Main Camera* objekat pri kreiranju novog projekta ili scene. Postojanje više kamera na istoj sceni igraču pruža mogućnost prikaza nekog drugog dela okruženja, ili prikaz okruženja iz nekog drugog ugla, kada za tako nešto postoji potreba.

Projekcija kamere govori o tome da li će se renderovanje vršiti u 3D ili 2D režimu, tj. Perspective ili Orthographic režimu. Perspektivni režim je standardni, i kao takav, ima FOV (Field Of View) u obliku piramide. Ortografski režim se koristi kod razvoja 2D igara, ili za postavljanje HUD (Heads Up Display) 2D elemenata u 3D igramu, koji služe za prikaz podataka o trenutnom rezultatu ili ostvarenom broju poena, preostale energije, mapa, itd..

1.5 Temena, Ivice, Poligoni, Mesh-evi

Složeni 3D objekti ne postoje kao takvi, već se dobijaju na osnovu jednostavnih 2D poligona koji su međusobno povezani i zajedno čine tzv. **Mesh**. Složeni modeli konstruišu se

uz pomoć softvera za 3D modelovanje, i kao takvi se mogu jednostavno uvući u Unity projekat, nakon čega je rad sa njima isti kao i sa bilo kojim drugim objektom u Unity-u. Pri uvozu modela iz nekog softvera za modelovanje, Unity konvertuje sve poligone u trouglove. Ovako dobijeni trouglovi predstavljaju strane, i sastoje se od tri povezane ivice. Mesta spajanja ivica su temena. Uz pomoć ovih podataka, Game Engine može da vrši razna izračunavanja vezana za objekat, poput onih koja se odnose na određivanje mesta udarca, ili kolizije dva objekta, o čemu ćemo kasnije govoriti. Na osnovu Mesh podataka, može se odrediti vizuelno približno isti oblik objekta, ali jednostavniji i sa manje detalja, koji je pogodniji za izračunavanja. Ovaj pristup doprinosi boljim performansama. Što je više poligona koji čine objekat, to objekat ima više detalja, i bolji, grafički realističniji izgled, ali to vuče intenzivnija izračunavanja, i zahteva jače mašine koje bi to podržale.

1.6 Materijali, shader-i, teksture

Materijali su zajednički koncept svih 3D aplikacija, i grafike uopšte, jer obezbeđuju vizuelni izgled modela. Materijal predstavlja definiciju kako površina treba biti renderovana, uključujući reference tekstura koje su korišćene, boja, i drugo. Podešavanjem materijala mogu se dobiti razni efekti, od obične boje do reflektujuće površine, i ove efekte dodelujemo objektima postavljanjem odgovarajućeg materijala. Materijal određuje koji će shader koristiti, a shader definiše koje će opcije biti dostupne za detaljno podešavanje tog materijala.

Shader je mali program ili skripta zadužena za stil renderovanja. Shader sadrži matematičke kalkulacije i algoritme za određivanje boje svakog renderovanog piksela, na osnovu osvetljenja i konfiguracije materijala objekta. Npr., kod reflektujućeg shadera, materijal će renderovati refleksije objekata okruženja, a pritom će zadržati i prethodno dodatu boju ili teksturu.

Teksture su bitmap slike koje možemo nalepiti na objekat u cilju promene njegove vizuelne pojave. Korišćenjem tekstura se imitira izgled objekata iz stvarnog sveta. Materijal može imati referencu teksturom, i tada je shader materijala uzima u obzir pri kalkulaciji boje površine objekta. Sem boje, tekstura može predstavljati i druge aspekte površine materijala, kao što su na primer refleksija i hrapavost.

U slučaju nekog normalnog renderovanja, pod čime se podrazumeva renderovanje okruženja, karaktera, jednobojnih i transparentnih objekata i sl., standardni shader je obično najbolji izbor. Ovo je izuzetno fleksibilan shader koji je u stanju da realistično renderuje razne vrste površina. Postoje i situacije kada standardni shader jednostavno nije dovoljan - kod renderovanja tečnosti, refraktivnog stakla, vegetacije, specijalnih efekata termovizije, noćnog vida i sl. Tada postaju korisni drugi ugrađeni, uvezeni ili samostalno napisani shaderi.

Korišćenje materijala u Unity-u je lako. Materijali koji su razvijeni van Unity-a mogu da se u par klikova uvezu u Unity i koriste u bilo kojem projektu. Takođe, Unity omogućava i kreiranje materijala od početka.

Kreiranje tekstura moguće je u svakom programu za obradu slika, poput Photoshop-a ili GIMP-a. Ono o čemu treba voditi računa prilikom kreiranja teksturom je rezolucija slika. Veća

rezolucija znači više detalja, ali i sporije renderovanje. Teksture u Unity-u se uvek skaliraju na stepen broja 2, npr. 64x64, 128x128, itd.

1.7 Rigidbody fizika

Kod razvoja igara, fizika je vrlo bitna, jer obezbeđuje simulaciju stvarnog sveta, i stvarnih reakcija i ponašanje objekata. Kao engine fizike, Unity koristi Nvidia PhysX engine. Ovaj engine je danas vrlo popularan, a sem toga nudi i visok nivo preciznosti.

U game engine-ima važi pretpostavka da fizika nema uticaja na sve vidljive objekte na sceni. Za tako nešto nema potrebe. To bi samo zahtevalo veću obradu, što se direktno odražava i na performanse. Ukoliko imamo igru čija je tema trka automobila, logično je da automobili budu kontrolisani nekom fizikom, ali za prepreke, zidove, put, i druge statične objekte ovo bi bilo bez efekata i suvišno. Iz ovog razloga, fizika na novododata objekte standardno ne utiče. Fiziku nad objektima uključujemo dodavanjem **Rigidbody** komponente. Na ovaj način objektu možemo dodeliti osobine mase, gravitacije, ubrzanja, i trenja, i ta svojstva možemo podešavati po svojoj volji.

1.8 Detekcija kolizije

Kako je u igrama često potrebno ispitati da li je došlo do kolizije dva objekta, Unity ima svoj pristup u rešavanju ovog problema. Oko posmatranih objekata formira se nevidljiva mreža, uz pomoć **Collider** komponente, koja imitira formu i oblik objekata, i na osnovu nje se vrše izračunavanja koja vode ka određivanju kolizije. U Unity-u su na raspolaganju dva tipa Collider-a - Primitive Collider i Mesh Collider.

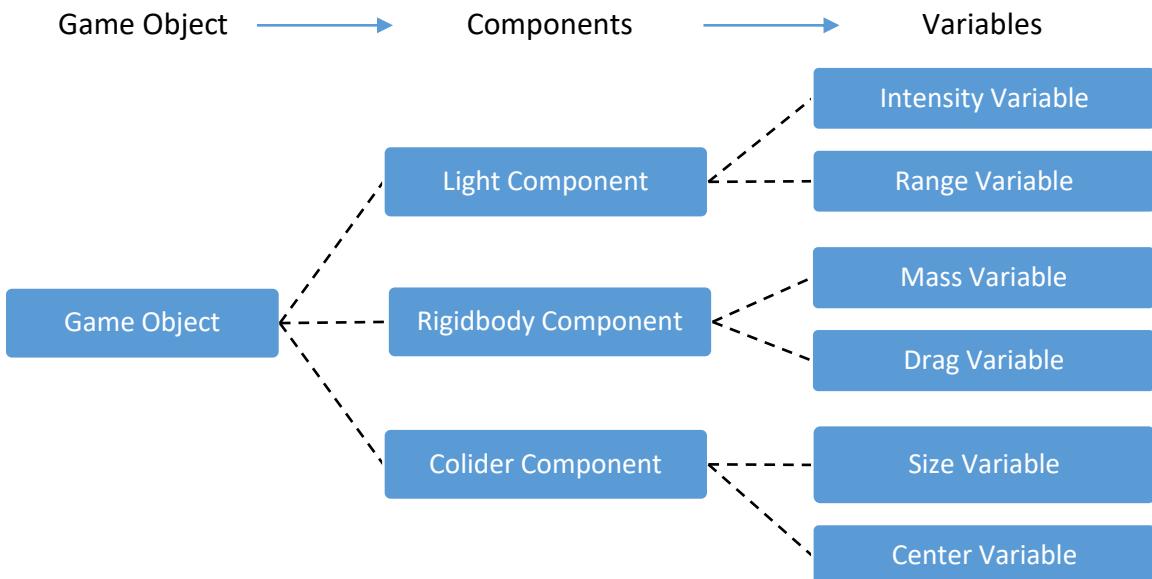
Primitivni oblici su jednostavnii 3D oblici poput kvadra, sfere i kapsule. **Primitive Collider** koji predstavlja kvadar, zadržava ovu formu, uprkos stvarnom obliku samog objekta kojem pridružujemo ovu komponentu. Primitive Collider-e koristimo kada nam preciznost nije na prvom mestu, jer su jednostavnii i efikasni u smislu izračunavanja.

Mesh Collider se zaniva na 3D Mesh-u koji čini objekat. Što je ovaj Mesh kompleksniji, to je Collider precizniji, sa više detalja, ali uz ovakva podešavanja treba očekivati slabije performanse. Zlatna sredina je korišćenje Mesh Collider-a koji prilično verno oslikava glavni objekat, ali ipak sa smanjenim nivoom detalja. Ovakav Collider ne pruža potpunu preciznost, ali je po ovom pitanju uvek bolji od Primitive Collider-a, dok zadržava visoke performanse.

1.9 Osnovni koncepti Unity tehnologije

Prvi osnovni koncept Unity-a je **GameObject**. Korišćenjem Game objekata, igra se može podeliti na delove kojima se lako upravlja, i koji se jednostavno povezuju i podešavaju. Game objekti se sastoje od komponenti. **Komponente** su zapravo funkcionalnosti koje objekat može da ima. Svaki objekat može imati skoro beskonačno komponenti, i dakle, isto toliko funkcionalnosti i osobina. Komponente imaju **promenjive**. To su osobine ili podešavanja koja datu komponentu kontrolišu. Podešavanjem ovih promenljivih, dobijamo potpunu kontrolu

nad efektima koje data komponenta ima nad objektom. Sledeći dijagram ove koncepte ilustruje kroz primer.



Slika 1.2: Osnovni koncepti Unity-a

Sledeći Unity koncept su **Asset**-i. To su svi oni delovi ili blokovi koji grade projekat i objekte - zvukovi, slike, teksture, materijali, animacije, modeli, ali i nadogradnje Unity-a u vidu ekstenzija, dodataka i skripti. Sve to predstavlja asset-e. Upravo iz tog razloga su svi fajlovi koji se koriste u projektu zapamćeni, standardno, u folderu *Assets*.

Skripte su sledeća bitna stavka. Nakon dodavanja objekata na scenu, i komponenti koje određuju njihov izgled, poziciju i druge osobine, potrebno je ugraditi logiku, čime se određuje i njihova uloga, tj. funkcija i ponašanje, što je i poenta celog razvoja igara. Ovo se postiže pisanjem skripti i njihovim pridruživanjem objektima u vidu komponenti. Unity podržava pisanje koda u jezicima C#, JavaScript, i Boo. Pored ovoga, Unity pruža i mogućnost korišćenja svojih biblioteka i ugrađenih funkcija, što dodatno olakšava posao programerima. Standardni editor za pisanje koda je Monodevelop, koji dolazi uz instalaciju Unity-a. Naravno, moguće je korišćenje i drugih editora, poput Microsoft Visual Studio okruženja, mada treba imati u vidu da često korišćena funkcionalnost auto-completion koda tokom kucanja nije implementirana u svim editorima. Ova funkcionalnost pozitivno utiče na brzinu pisanja koda.

Unity omogućava pamćenje kreiranog objekta u vidu prefabrikovanog modela, sa svim komponentama, osobinama i logikom, koje ga čine onim što jeste. Ovo je koncept Unity-a, koji se naziva **Prefabs**, a ovakvi objekti su u potpunosti spremni za kasnije korišćenje u istom projektu (npr. kod dinamičkog kreiranja više istih objekata programabilno), ili u drugim projektima jednostavnim uvozom.

Tags, ili tagovi, predstavljaju način identifikovanja objekata u Unity-u. Jedan od načina identifikovanja objekta je naravno korišćenje imena tog objekta. U pojedinim situacijama korisno je imati načina za zajedničko identifikovanje sličnih ili istih objekata. Na primer, igra može sadržati objekte poput tenkova, aviona, vojnika, a svi oni mogu biti pod istim tagom -

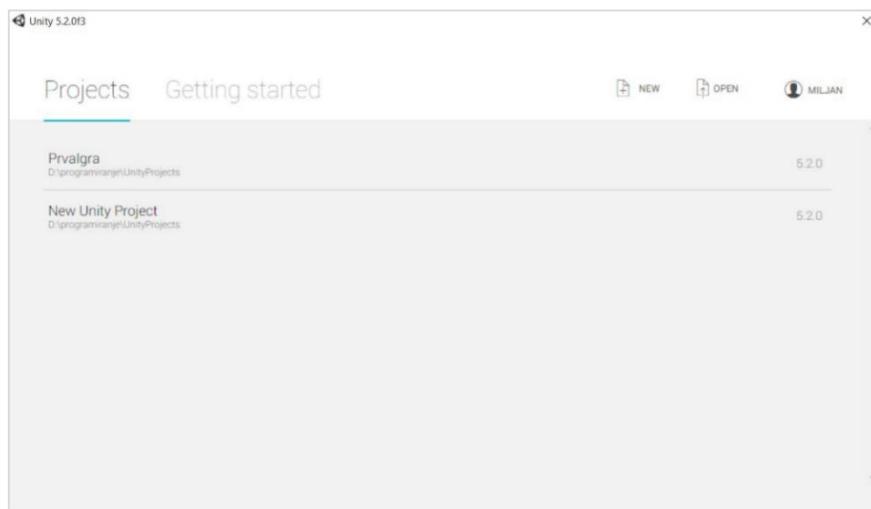
Enemy. Korišćenjem ovog taga, programskim kodom možemo lako pristupiti i proveriti sve neprijateljske objekte.

Layers, ili slojevi, čine sledeći koncept. Layer-i ukazuju na neke funkcionalnosti koje su zajedničke različitim, pa i nesrodnim, objektima. Na primer, slojevi mogu da ukazuju na to koji će objekti biti iscrtani, ili koji će biti sakriveni, ili na koje će moći da se puca, ili jednostavno koji će objekti imati neku specifičnu osobinu. Grupisanje objekata u slojeve je jednostavno, i vrši se iz padajuće liste Inspector-a. **Inspector** je jedan od najbitnijih delova radnog okruženja, i upravo ćemo kroz proučavanje radnog okruženja i korisničkog interfejsa, govoriti o ovom, ali i i drugim važnim konceptima Unity-a.

2 Unity UI

2.1 Kreiranje novog projekta

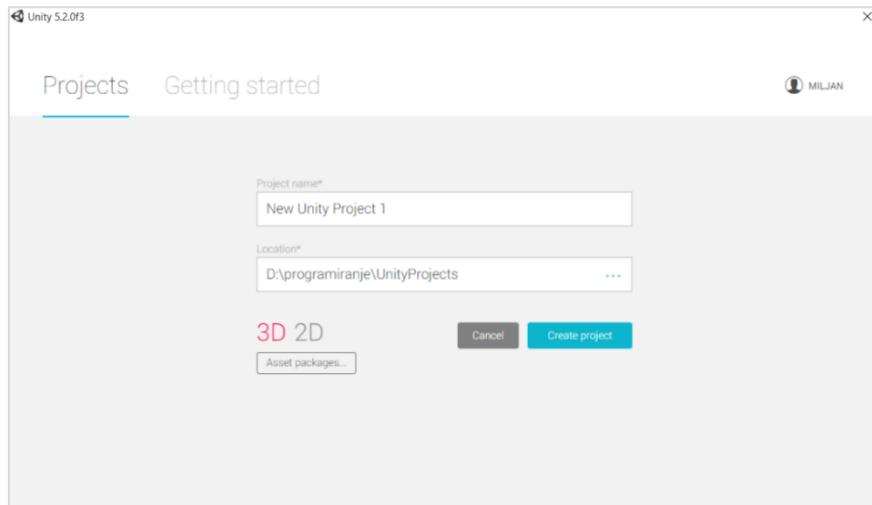
Instalacija Unity-a je prilično jednostavna. Potrebno je najpre preuzeti instalacione fajlove besplatne verzije Unity-a sa zvanične web strane - <https://store.unity.com/download>. Nakon toga treba pokrenuti instalaciju i pratiti korake. Tokom instalacije vrši se besplatna registracija proizvoda, kreiranjem Unity naloga. Kreiranje naloga je lako, i brzo se završava popunjavanjem traženih podataka, poput imena, email adrese, i sl. Pri svakom pokretanju programa, Unity od korisnika zahteva kreiranje novog, ili učitavanje postojećeg projekta, što se vrši kroz Unity Project Wizard, koji je prikazan na slici 2.1.



Slika 2.1: Unity Project Wizard

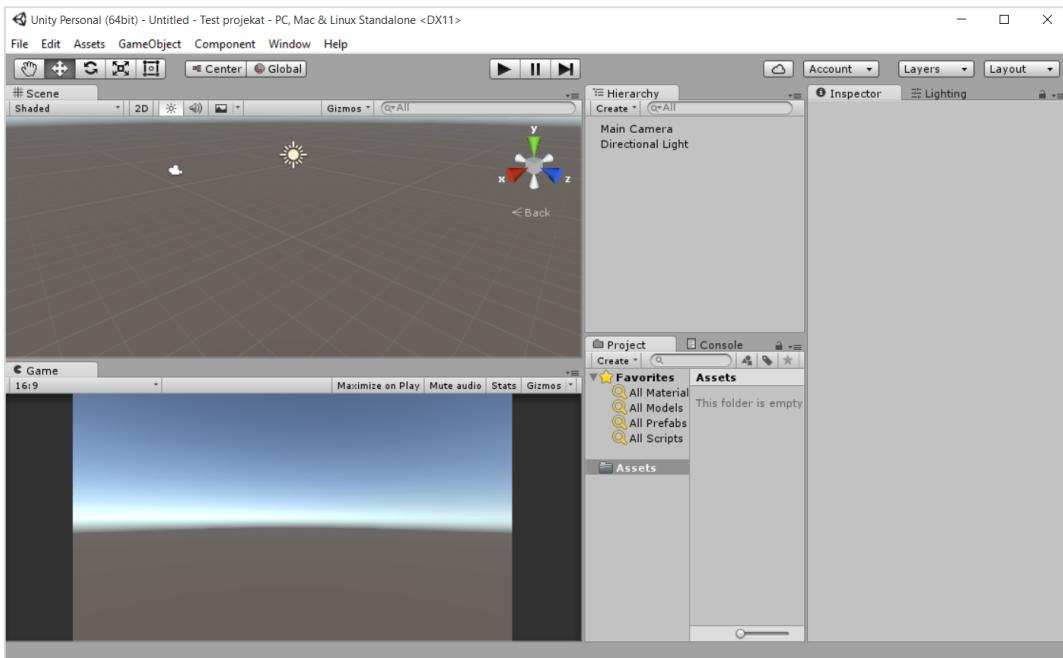
Klikom na neki od postojećih projekata iz Recent liste, ili klikom na dugme OPEN, Unity učitava potrebne fajlove, i pokreće odabrani projekat. Klikom na dugme NEW, kreira se novi projekat. U prozoru koji se otvara upisuje se željeni naziv projekta, bira se putanja za smeštanje projekta i svih pratećih fajlova, zatim tip projekta - 2D ili 3D, i opcionalno, označe se Asset paketi koje želimo da koristimo u projektu. Asset pakete možemo uvesti i kasnije iz okruženja, i o njima će kasnije biti više reči. Nakon popunjavanja ovih polja, klikom na dugme Create Project, proces kreiranja projekta je završen, i Unity okruženje se pokreće.

Svaki Unity projekat karakteriše poseban folder na hard disku, u okviru kojeg se pamte i smeštaju svi fajlovi koje kreiramo ili uvozimo tokom razvoja. Ovom folderu se može pristupiti direktno iz Unity-a, što je ispravno i svakako dobra praksa, ili iz Windows Explorer-a. Brisanjem fajlova, promenom njihovih putanja, ili naziva, i to klasičnim pristupom iz Windows Explorer-a, ruše se veze koje je Unity uspostavio sa ovim fajlovima, i u datom projektu će se gotovo sigurno javiti greške.



Slika 2.2: Unity New Project Window

Unity okruženje ima dve boje, ili teme, Dark i Light, međutim korisnici besplatne verzije programa mogu da uživaju samo u Light temi, koju vidimo na slici 2.3. Korisnici plaćene Pro verzije programa mogu iz menija Edit / Preferences promeniti temu iz padajuće liste.



Slika 2.3: Unity okruženje (Light tema)

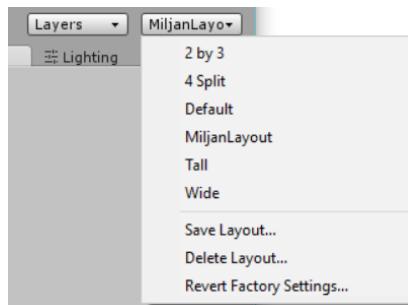
Dobra je navika, već u ovom trenutku zapamtiti scenu. **Scene** možemo posmatrati kao nivoe koji sadrže elemente igre - okruženje, karaktere, osvetljenje, kamere, i sl. Pokretanjem

igre zapravo se pokreće neka scena. Mada scene zaista možemo koristiti kao nivoe u igri, ne smemo ih posmatrati samo kao takve. Koristeći scene možemo kreirati menije, splash (uvodne) ekrane itd. Kako scena čuva sve podatke vezane za igru i ono što igra prikazuje, pametno je stalno pamtiti promene, kako ne bismo došli u situaciju da usled neke greške ili zamrzavanja ekrana, izgubimo prethodni rad.

Iz File menija, biramo opciju Save Scene, i upisujemo željeni naziv scene pri prvom pamćenju. Za veće projekte koji sadrže više nivoa ili scena, poželjno je kreirati novi folder u okviru projekta, i ovde pamtiti i skladištiti sve scene, radi bolje preglednosti i kasnije lakšeg pronalaženja.

2.2 Layout

Izgled Unity-a i raspored komponenti razvojnog okruženja može da se podešava na puno načina, tako da svako može da pronađe svoj najbolji **Layout**. Osnovna, predefinisana Layout podešavanja, nalaze se u gornjem desnom uglu. Klikom na ovu opciju dobijamo padajući meni gde možemo odabratи neki od postojećih, kreirati novi, ili izvršiti brisanje prethodno kreiranog Layout-a.



Slika 2.4: Odabir Layout-a iz Layout menija

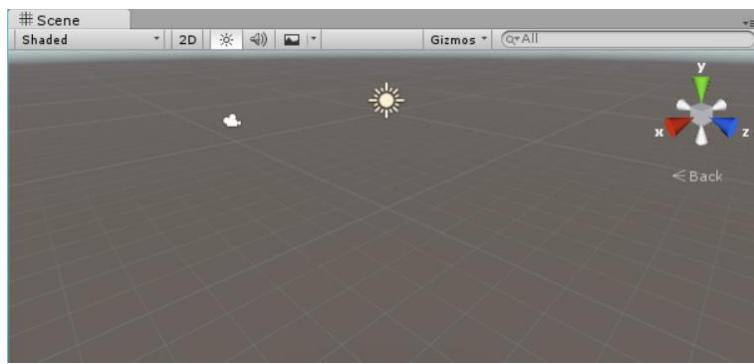
Programeri podešavaju svoje radno okruženje po svojoj volji i ukusu, a u cilju boljeg snalaženja i bržeg razvoja. Često korišćeno Layout podešavanje je 2 by 3. Delimično izmenjen 2 by 3 Layout se može videti na slici 2.3.

2.3 Scene View

Scene View je mesto gde se razvija vizuelni aspekt scene. Korišćenjem Scene View-a, dobijamo pogled u 3D svet naše igre. Na scenu prevlačimo elemente igre, nakon čega im možemo menjati poziciju, veličinu i rotaciju, i to pomoću strelica oko obeleženog objekta. Kroz Scene View, na jednostavan način organizujemo izgled čitavog nivoa igre. Precizna podešavanja i manipulaciju sa objektima scene možemo vršiti uz pomoć Inspector-a, koji se takođe pojavljuje obeležavanjem objekata, bilo iz Scene ili Hierarchy View panela.

U gornjem desnom uglu Scene View-a je tzv. **Gizmo**, pomoću kojeg menjamo pogled na scenu, po X, Y, ili Z osi. Klik po sredini Gizma vodi u perspektivni pogled.

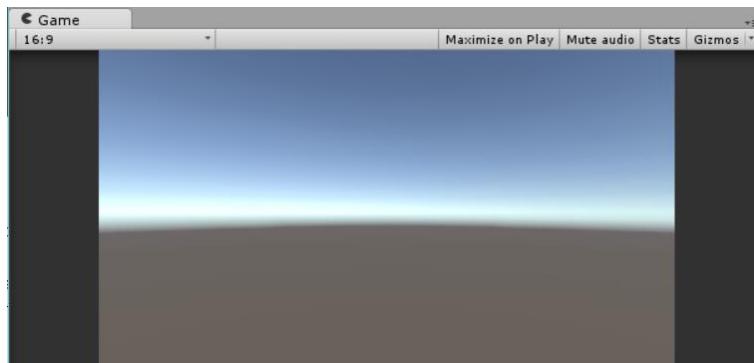
Scene View sadrži i polje za pretragu koja može da se vrši po tipu ili nazivu objekta, i u slučaju kada imamo puno postavljenih objekata na sceni, ovo može biti od velike koristi.



Slika 2.5: Scene View

2.4 Game View

Game View je komponenta gde se vrši testiranje aplikacije, pre krajnjeg build-ovanja za neku platformu. Za razliku od Scene View-a, gde postoji mogućnost proizvoljnog kretanja po sceni, zumiranja, manipulacije objekima, ovde tih mogućnosti nema, i svet igre posmatramo samo kroz prethodno definisani objekat kamere. Dugme koje se ovde često koristi je Maximize on Play, i ono omogućava pokretanje aplikacije preko celog ekrana. Tu je i opcija isključivanja zvuka (Mute Audio), Aspect Ratio podešavanje u gornjem levom uglu, ali i dugme Stats koje prikazuje statistiku igre koja se odnosi na zvuk, grafiku (broj temena, trouglova, rezolucija, senke, animacije, fps, i sl.) i zauzeće resursa.

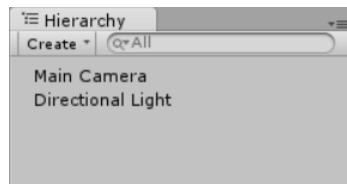


Slika 2.6: Game View

2.5 Hierarchy View

Hierarchy View prikazuje sve objekte koji se trenutno nalaze na aktivnoj sceni, bilo da su oni uključeni ili isključeni, vidljivi ili sakriveni. To mogu biti objekti okoline, karaktera, izvori svetlosti, kamere, i drugi. Game objekti mogu da se dodaju i dinamički, iz koda, i tom slučaju su prikazani kroz Hierarchy View jedino u aktivnom stanju, u toku igre. Uklanjanje objekata sa scene, opet, moguće je iz programskog koda. Dupli klik na objekat iz Hierarchy View-a fokusira i zumira Scene View na dati objekat. Isto postižemo označavanjem objekta pomoću tastature ili miša, i pritiskom tastera F na tastaturi.

Dakle, Hierarchy View omogućava detaljniji i lakši prikaz i pristup objektima scene, njihovu hijerarhijsku organizaciju, kao i kreiranje novih objekata iz kontekstnog menija koji se dobija pomoću desnog tastera miša.

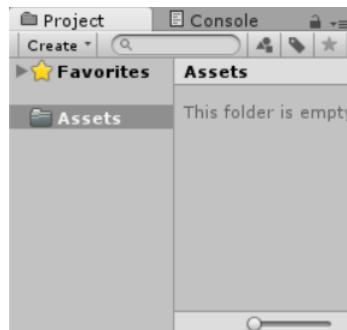


Slika 2.7: Hierarchy View

2.6 Project View

Project View je deo Unity okruženja i služi za prikaz i rad sa folderima i fajlovima projekta. To su fajlovi koje smo sami kreirali ili uvezli u projekat - scene, skripte, modeli, tekture, zvuci, itd. Project View zapravo predstavlja reprezentaciju stvarnog foldera projekta koji se nalazi na određenoj lokaciji na hard disku. Modifikacijom ili brisanjem fajlova iz Project View panela, Unity ažurira veze sa istim, pa ukoliko su prethodne promene dovele do neke greške, to se relativno lako i brzo može korigovati. U suprotnom, ukoliko se vrše promene foldera i fajlova iz Windows Explorer-a ili drugog File Manager-a, kidaju se prethodno uspostavljene veze projekta sa fajlovima, što dovodi do pojave većih grešaka, pucanja scene, ili neučitavanja projekta.

Project View ima dve layout opcije, tj. dve mogućnosti prikaza fajlova, kroz jednu ili dve kolone. Na korisniku je da izabere željeni izgled, mada se prikaz kroz dve kolone uglavnom pokazao kao pregledniji i efektivniji. Prva kolona u tom slučaju služi za odabir foldera, a druga za prikaz fajlova u odabranom folderu, i njihovu modifikaciju.

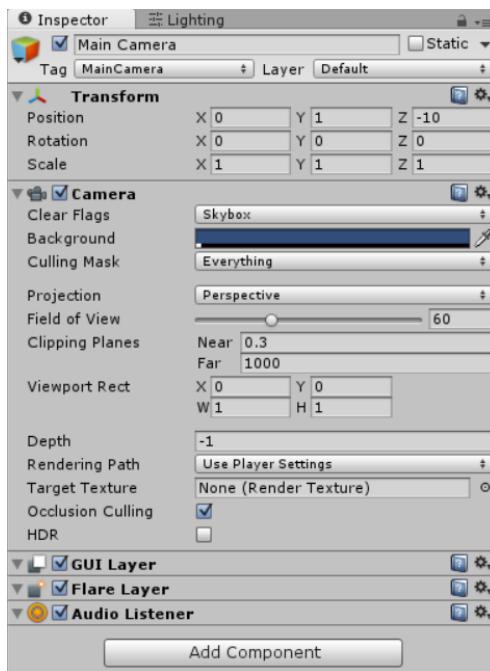


Slika 2.8: Project View sa two-columns layout-om

2.7 Inspector

Inspector je bitna, vrlo korisna, i verovatno najkorišćenija komponenta Unity okruženja. Koristi se za pristup svim podešavanjima Unity-a, kako onima koja se odnose na samo razvojno okruženje, ili učitani projekat, tako i podešavanjima karakteristika konkretnih objekata scene, i njihovih komponenti. Primer izgleda Inspector-a možemo videti na slici 2.9, a radi se o podešavanjima *Main Camera* objekta.

Različiti objekti na sceni mogu imati različite karakteristike i podešavanja. Osobine koje možemo videti u Inspector-u objekta glavne kamere, nećemo naći kod, npr., objekta koji predstavlja izvor svetlosti.



Slika 2.9: Inspector objekta glavne kamere

Pomenimo prvu grupu podešavanja Inspector-a objekata - komponentu **Transform**. Ova komponenta je zajednička za sve objekte scene, a njena podešavanja se odnose na poziciju, rotaciju, i skaliranje označenih objekata. S obzirom da radimo u 3D prostoru, sve navedene karakteristike imaju X, Y, i Z komponentu. Ove promjenjive podešavamo pomoću klizača, ili upisom vrednosti u odgovarajuća polja.

Svaka komponenta ili grupa podešavanja objekta u Inspector-u može da se isključi ili uključi, i to, uz pomoć check-box-a koji se nalazi pored naziva te komponente. Dobra osobina Unity-a je što Scene View automatski registruje svako ažuriranje podešavanja u Inspector-u, i svaku promenu vrednosti atributa komponenti objekata, tako da je rezultat promena odmah vidljiv.

2.8 Toolbar

Ispod linije menija, nalazi se linija sa alatima, tzv. **Toolbar**, sačinjen iz nekoliko različitih grupa kontrola koje se odnose na Scene View i manipulaciju sa scenom, ali pored toga sadrži i kontrole za pokretanje igre u editoru.



Slika 2.10: Toolbar

Prva grupa kontrola sastoji se iz alata za navigaciju scenom, i transformaciju objekata. Prvi alat s leve strane, hand tool, koristi se za kretanje po sceni. Korišćenjem ovog alata uz pritisnut taster Alt na tastaturi, vrši se rotiranje, tj. orbitiranje, dok se uz pritisak desnog tastera miša zajedno sa tasterom Alt, vrši zumiranje scene. Isto se postiže i okretanjem točkića miša. Skoro sve u Unity-u može da se uradi na više načina, pa tako su dostupne i prečice na tastaturi za pristup ovoj grupi alata, i to su tasteri Q, W, E i R.

Drugi alat (W) ove grupe služi za pomeranje i pozicioniranje objekata po sceni. Može se vršiti pomeranje po X, Y i Z osi, povlačenjem odgovarajuće strelice objekta u fokusu. Slobodno pomeranje objekta po svim osama vrši se prevlačenjem kvadrata iz njegovog centra. Ukoliko istovremeno držimo pritisnutim Ctrl taster na tastaturi, omogućava se pomeranje objekta za precizne inkremente, čija je standardna vrednost jednaka 1.

Treći alat (E) služi za rotaciju objekta. Crvena, zelena i plava linija koje se pojavljuju odabirom ovog alata, odnose se na X, Y, i Z osu rotacije.

Četvrti alat (R) funkcioniše i koristi se na isti način kao i prethodni, ali je njegova funkcija skaliranje objekata. Povlačenjem strelice miša po nekoj osi objekta, uvećava se njegova dimenzija po toj osi, dok se povlačenjem kvadra iz centra, dimenzije uvećavaju po svim osama.

Nakon prve grupe alata slede kontrole Center/Pivot i Global/Local, koje služe za podešavanje koordinatog sistema i centra u odnosu na koji će se vršiti promene pomoću prethodno pomenutih alata.

U centralnom delu se nalaze kontrole za pokretanje i pauziranje igre u toku testiranja. Poslednje dugme u ovom nizu koristi se za bolju kontrolu toga šta se dešava u igri, jer pruža mogućnost frame-by-frame prikaza.

Slede opcije za podešavanje naloga, Layer-a, i Layout-a. Layer-i se između ostalog koriste radi kontrole kojim kamerama će objekti biti renderovani ili kojim će svetlima biti osvetljeni. Različiti Layer-i mogu imati različite funkcije i o tome smo već govorili.

2.9 Meniji

Kako Unity pruža zaista veliki broj opcija i funkcionalnosti, nisu sve mogле biti prikazane u glavnom prozoru i najvažnijim panelima, pa su tako sve ostale raspoređene u padajućim listama glavne linije menija. Često korišćene opcije ovih menija uglavnom imaju i svoju prečicu na tastaturi, što ima za cilj da pojednostavi i ubrza rad.



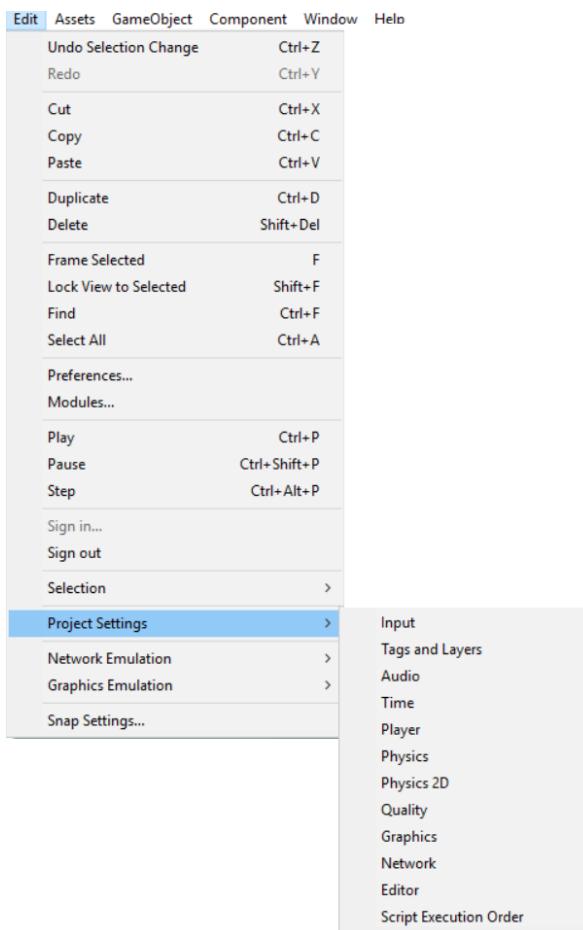
Slika 2.11: Linija menija

File

Iz File menija možemo učitati, kreirati i pamtititi scene i projekte. Bitna podešavanja ovog menija su podešavanja vezana za odabir platforme i Build aplikacije sa dodatnim opcijama u Inspector-u.

Edit

Edit meni sadrži očekivane Undo, Redo, Cut, Copy, Paste, Duplicate i Delete komande, kao i nekoliko runtime komandi. Tu su prethodno pomenute Play, Pause i Step opcije, opšta podešavanja - Preferences, i podešavanja projekta - Project Settings, koja možemo videti na slici 2.12. Odabirom bilo koje od ovih opcija, odgovarajuća podešavanja prikazuju se u Inspector-u. U okviru Project Settings menija možemo pronaći razna podešavanja vezana za projekat - podešavanje ulaza, tagova, slojeva, zvuka, grafike, kvaliteta, itd.



Slika 2.12: Edit / Project Settings meni

Assets

Asset-i predstavljaju blokove koji grade svaki Unity projekat. Teksture, slike, 3D modeli, zvuci, sve su to asset-i. Podmeni koji se najviše koristi je Create, i on služi za kreiranje asset-a, a sem navedenih, tu su i skripte, materijali, animacije, fontovi, itd. Ovaj meni služi i za kreiranje novih foldera. Grupisanje fajlova u odgovarajuće foldere puno olakšava organizaciju projekta. Create meni je dostupan i iz kontekstnog menija Project View panela. Veoma korisna mogućnost Unity-a je jednostavan uvoz asset-a iz drugih softvera, poput 3D Studio Max-a, ali i preuzimanje gotovih asset-a iz Unity Asset prodavnice, sa kojom se upoznajemo kasnije.

GameObject

Osnovna jedinica građe i funkcije Unity aplikacija jeste Game Object. Game objekat može biti prazan objekat, kontejner za druge objekte, skripte, a može biti i kompletni složeni model koji imitira stvarni objekat iz realnog sveta. Game objekat se sastoji iz komponenti. Osnovne komponente svih Game objekata se odnose na lokaciju, rotaciju i veličinu, i to su zajedničke osobine svih vizuelnih objekata scene. Dodavanjem i podešavanjem komponenti, Game objekat može da postane baš ono što mi želimo. Između Game objekata može da postoji veza roditelj-dete. Osnovni Unity Game objekti su ravan, kvadar, sfera, cilindar, ali se ovih, pod Game objektima ubrajamo i sisteme čestica, kamere, svetla, i dr. Iz menija GameObject kreiramo nove objekte i postavljamo ih na scenu.

Component

Component meni daje pristup komponentama koje određuju objekat - Collider-i, Mesh-evi, skripte, efekti, itd. Komponente mogu određivati izgled, ponašanje, kao i druge funkcionalnosti i svojstva koje objekat može da ima. Komponente možemo kreirati, a ukoliko ih već imamo gotove, možemo ih samo prevući na željeni objekat, čime će automatski biti dodate. Sve komponente koje su pridružene objektu, možemo menjati i prilagođavati, promenom njihovih atributa u Inspector-u. Sem ovih standardnih komponenti koje vezujemo za objekte, tu su i UI komponente koje se često koriste u kreiranju menija, ili bilo kakvog prikaza dugmića, slika i teksta u 2D ili 3D aplikacijama.

Window

Window meni prikazuje komande sa odgovarajućim prečicama na tastaturi, za uređenje samih prozora i editora Unity okruženja. Takođe, ovaj meni pruža pristup oficijalnoj Unity Asset prodavnici, gde se mogu pronaći razni gotovi asset-i, Unity dodaci, ekstenzije. Mnogi od njih su besplatni, ali se neki, pogotovo oni kvalitetniji, naplaćuju.

Help

Ovo je meni za pomoć. Tu su Unity uputstva, zatim Unity Answers sa odgovorima na česta pitanja, kao i Unity Forum sa velikom podrškom Unity programera. U ovom meniju pronalazimo i uputstva za prijavu bagova, i beleške o trenutno instaliranoj verziji Unity-a.

2.10 Zašto Unity?

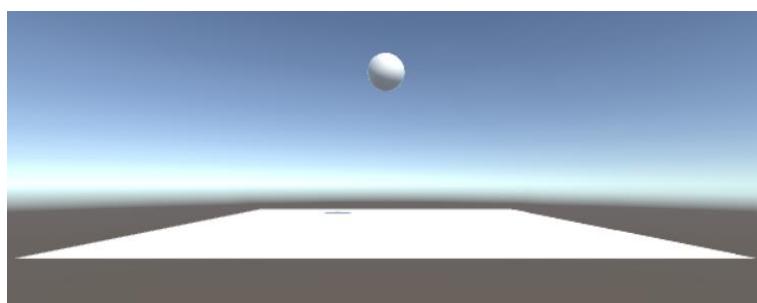
Danas je dostupan veliki broj game engine-a, framework-a, i alata za razvoj igara. Nijedan nije savršen, i za različite svrhe jedan može biti bolji od drugog na određenom polju. Zašto je Unity jedna od boljih opcija?

- **Velika zajednica:** Unity ima veliku bazu korisnika, Internet zajednicu, forume, i sve to puno olakšava i ubrzava pronađenje odgovora na razna pitanja. Pored toga, puno je knjiga, tutorijala, i video klipova za učenje, koji objašnjavaju nove i postojeće koncepte. Velika zajednica podrazumeva nastavak razvoja softvera.
- **Podrška za više platformi:** Unity omogućava razvoj aplikacija za veliki broj platformi. PC, Mac, Linux, iOS, Android, BlackBerry, Playstation, Xbox, itd. Na ovako raznolokom tržištu kakvo imamo danas, pametna je odluka koristiti tehnologiju koja cilja više platformi.
- **Prilagodljivost:** Developeri mogu prilagoditi Unity okruženje svojim potrebama, a i sami mogu kreirati svoje prozore, dugmiće, i panele, koje mogu i da dele preko Unity Asset prodavnice, sa drugim developerima. Time se iskustvo u razvoju igara još više poboljšava.
- **Dosadašnji uspeh:** Unity koristi veliki broj developera, a svoju moć Unity pokazuje i time što se na listama najpopularnijih igara raznih platformi nalazi veliki broj igara koje su razvijane upravo pomoću Unity tehnologije. Drugi dokaz uspeha je i partnerstvo sa velikim kompanijama, poput Electronic Arts.

3 Primer

Kako smo proučili osnovne koncepte Unity-a, želimo na jednostavnom primeru pokazati kako oni zajedno funkcionišu u praksi. Kreiramo 3D aplikaciju u kojoj objekat u obliku sfere pada sa određene visine, odskakuje od podloge, i imitira ponašanje prave lopte.

Najpre postavljamo objekat ravni na scenu. Ovaj objekat će predstavljati podlogu za odskakivanje, a dodajemo ga odabirom opcije **Plane** iz GameObject / 3D object menija. Poziciju novokreirane ravni postavljamo iz Inspector-a na (0, 0, 0), dok dimenziju podešavamo proizvoljno. Bez podloge, lopta ne bi imala od čega da odskače, i samo bi nastavila da pada. Loptu realizujemo pomoću **Sphere** objekta, i na scenu je dodajemo iz istog menija u kojem smo pronašli i Plane objekat. Klikom na objekat sfere iz hijerarhijskog panela, otvara se Inspector sa podešavanjima priključenih komponenti. Unity automatski dodaje **Mesh Renderer** komponentu koja sferu čini vidljivom. Manuelno dodajemo komponentu **Rigidbody**, klikom na dugme Add Component u Inspector-u, i pronalaženjem ove komponente u Physics grupi Unity komponenti, ili jednostavnije, upisom njenog naziva u polje pretrage. Rigidbody komponenta daje Unity-u naredbu da primenjuje fiziku nad objektima koji je sadrže. Ovom komponentom objektu pružamo nove osobine, kao što su masa, gravitacija, otpor, itd. Opcija Use Gravity komponente Rigidbody je standardno čekirana, i tako treba i da ostane i u našem primeru, kako bi gravitacija delovala na loptu i privukla je do podloge. Objekat koji predstavlja loptu postavljamo iznad podloge, tj. Y komponenta pozicije u okviru Transform podešavanja ovog objekta treba biti pozitivna. Ukoliko u ovom trenutku pokrenemo aplikaciju, rezultat je da lopta pada na podlogu, i tamo ostaje, ali bez odskakivanja. Šta je potrebno uraditi kako bi lopti dali mogućnost da odskače? Collider komponenta objekta sfere ima osobinu **Physic Material**, koja definiše kako objekat reaguje u odnosu na površinu drugih objekata. Klikom na kružić pored ove osobine, biramo opciju Bouncy. Ova opcija predstavlja fizički materijal koji Unity nudi za korišćenje, a odnosi se na osobinu skakutavosti i elastičnosti, pa će odlično poslužiti za naše potrebe. Ukoliko ne koristimo predefinisane asset-e koje Unity pruža, možemo kreirati svoj materijal sa istim osobinama. Iz Create menija, biramo opciju Physic Material, i unosimo naziv materijala (npr. *BouncyMaterial*). U Inspector-u materijala postavljamo osobinu Bounciness na 1. Ovaj materijal dodajemo objektu lopte na već pomenuti način, ili jednostavnije, prevlačenjem materijala iz Project View panela na ciljani objekat u hijerarhijskom panelu. Klikom na Play iz linije alata, pokrećemo igru, i vidimo traženi rezultat. Objekat lopte pada sa visine koju smo odredili, i usled delovanja fizike i osobina dodeljenog materijala, odskače nekoliko puta, pre nego što se konačno zaustavi, poput prave lopte.



Slika 3.1: Primer 3D aplikacije - odskakanje lopte

4 Razvoj 3D igre

4.1 Ideje

Proučili smo osnovne koncepte i ideje Unity tehnologije, 3D prostora, i to primenili u razvoju jednostavne 3D aplikacije sa odskakivanjem lopte. Sada je vreme da ono što smo naučili proširimo i primenimo u razvoju jedne složenije 3D igre za Windows platformu. Ideja je sledeća.

Igra će se odvijati na prostoru koji je ograničen zidovima, i po izgledu trebalo bi da podseća na sobu. Da bismo postigli ovaj efekat, obogatićemo prostor elementima koje inače možemo naći u svakoj sobi. Zatim, odredićemo više lokacija sa kojih se pojavljuju neprijatelji. Funkcija neprijatelja je napad na glavnog lika igre kojeg kontroliše korisnik. Neprijatelji imaju isti cilj, ali različite početne pozicije, brzinu kretanja, i druge osobine. Napad izvode direktnim kontaktom, pomoću sekire. Glavni karakter igru počinje iz centra prostorije, a kako ga neprijatelji napadaju sa svih strana, mora sve vreme da beži kako bi izbegao udarce i duže ostao živ. Na napade, glavni lik odgovara puškom. Na prvi pogled, možda ne izgleda kao fer borba, međutim kako vreme odmiče, broj neprijatelja se povećava, kao i njihova brzina, pa će igra postajati sve teža. Cilj igrača je da ubije što više neprijatelja i time ostvari što veći broj poena. Igra će imati samo jedan nivo, i uvek se završava smrću glavnog karaktera. Neprijatelji, kao i glavni lik, imaće svoju energiju koja se usled primljenih udaraca sve više smanjuje, dok se konačno ne izgubi život.

Kontrola glavnog lika igre na Windows platformi vršiće se preko tastature i miša. Tastatura će biti zadužena za kretanje po sceni, a miš za ciljanje i pucanje iz oružja. Svakim ubistvom, ostvaruju se novi poeni, što na završetku igre formira krajnji rezultat. Najbolji rezultati biće evidentirani, i njihov pregled omogućen kroz tabelu.

Igra će imati više stanja i play / pause režime rada, koje prikazujemo kroz UI menije, baš kao i listu rezultata, podešavanja i druge opcije. Igrač će u svakom trenutku moći da pauzira igru, krene igru ispočetka, pogleda najbolje rezultate, ili promeni neko podešavanje.

Za razvoj koristimo Unity, verzije 5.2.0f3 (Personal).

4.2 Uvoz asset-a

Na već prikazan način, kreiramo novi Unity projekat, biramo destinaciju za smeštanje fajlova, i naziv aplikacije. Kao destinaciju projekta obično je najbolje odabrati particiju diska na kojoj nije instaliran operativni sistem, i koja služi za čuvanje podataka. Naziv aplikacije zavisi od kreativnosti autora. Mi upisujemo *Li'l Killer (Mali Ubica)*.

Idea ovog rada nije dizajn, 3D modelovanje, kreiranje tekstura, animacija modela i sl., pa se za naše potrebe služimo gotovim elementima koje nudi prodavnica - **Unity Asset Store**. Za objekat glavnog karaktera igre biramo 3D model čovečuljka iz *Survival Shooter* paketa, dok za objekte neprijatelja koristimo modele paketa *3 Free Characters*. Do ovih paketa lako se dolazi pretragom prodavnice.

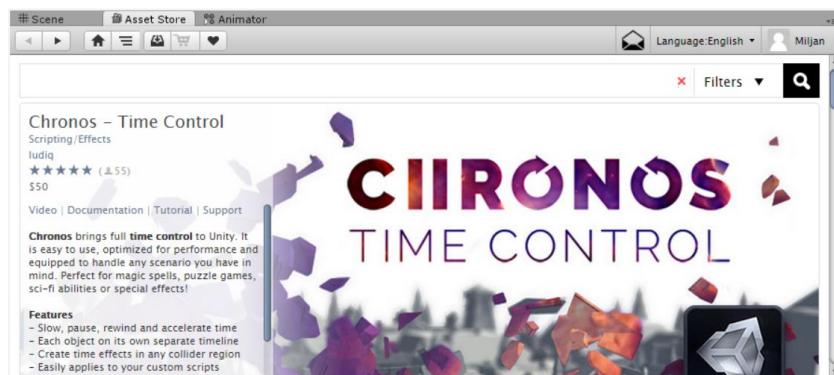


Slika 4.1: *Survival Shooter* model (levo) i 3 *Free Characters* modeli (desno)

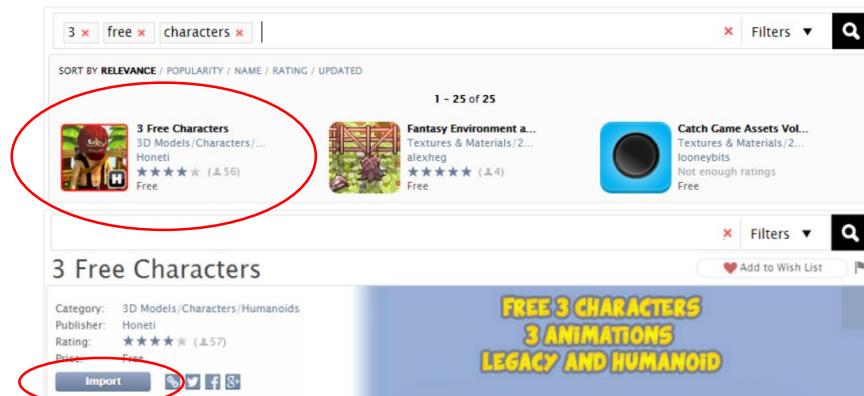
Prodavnici se pristupa na dva načina, preko Internet pretraživača, ili direktno iz Unity okruženja - iz Windows padajućeg menija, ili prečicom Ctrl+9 na tastaturi. Pregled prodavnice iz Unity okruženja možemo videti na slici 4.2.

Asset prodavnica nudi veliku bazu besplatnih asseta, ali i onih koji se plaćaju, i to sve kroz prikaz po kategorijama, što olakšava pregled i pretragu. Pretraga je omogućena i unosom ključnih reči, a dodatni filteri rezultata se nalaze sa desne strane.

Kako znamo koji su nam paketi potrebni, kucamo njihove nazive, nakon čega ih među ponuđenim rezultatima otvaramo, i preuzimamo klikom na dugme Download. Nakon toga vršimo uvoz u Unity projekat klikom na dugme Import. Pri uvozu, moguće je čekirati ili dečekirati određene elemente preuzetog paketa. Čekirani asset-i odabrani za uvoz se zatim pojavljuju u odgovarajućim folderima našeg Project View panela, spremni za modifikaciju i korišćenje.



Slika 4.2: Asset Store



Slika 4.3: Uvoz paketa iz Asset Store-a

4.3 Environment

Pre pisanja bilo kakve logike, i postavljanja karaktera na scenu, poželjno bi bilo formirati bar početnu verziju okruženja u kojem će se karakteri kretati, i gde će se voditi borba između glavnog lika i neprijatelja.

U hijerarhijskom panelu iz menija Create dodajemo novi Empty Object objekat, sa nazivom *Environment*. Ovo je glavni objekat u okviru kojeg dodajemo sve ostale objekte koji zajedno čine čitavo okruženje. Objekat pozicioniramo u tački (0, 0, 0) podešavanjem Transform komponente. Najpre postavljamo podlogu po kojoj će se karakteri kretati. Kreiramo novi 3D objekat Cube, menjamo naziv na *Bottom* i postavljamo dimenzije na (40, 0.2, 40), a poziciju na (0, 0, 0). Preuzimamo asset *Wood Texture Floor* iz Asset prodavnice, uvozimo, i iz novonastalog foldera *Wood Textures* prevlačimo teksturu na *Bottom* objekat, a rezultat je odmah vidljiv. Kreiramo još dva Cube objekta, *Wall1* i *Wall2*, sa dimenzijama (40, 15, 0.2), i pozicijama (0, 7.5, 20) za prvi zid, i (-20, 7.5, 0) za drugi, uz rotaciju od 90 stepeni po Y osi. Preuzimamo i uvozimo paket *10 High Resolution Wall Textures* iz Asset prodavnice, i iz foldera *ADG_Textures* Project View panela biramo tekturu po želji, i prevlačimo direktno u hijerarhijski panel, na objekte *Wall1* i *Wall2*.

Pozicija, dimenzija, izgled i druge karakteristike objekata su potpuno konfigurabilne, i podešavaju se po želji kreatora igre. Rezultat našeg dosadašnjeg rada vidimo na slici 4.4.



Slika 4.4: *Environment*, nakon dodavanja podloge i zidova

U ovom trenutku dobra je ideja grupisati foldere uvezenih tekstura u jedinstveni folder *Textures*, u kojem ćemo skladištiti i sve buduće teksture.

Kako bismo onemogućili glavnom karakteru da izađe van predviđenog polja za igru, ograđujemo i preostale dve strane prostorije kreiranjem nevidljivih zidova, *Wall3* i *Wall4*, sa pozicijama (0, 2.5, -20) i (20, 2.5, 0) i dimenzijom (40, 5, 0.2), uz rotaciju četvrtog zida od 90 stepeni po Y osi. Efekat nevidljivih zidova dobijamo brisanjem Mesh Renderer komponente ovih objekata.

Praćenje pogleda glavnog karaktera, tj. podešavanje njegove rotacije i smera za pucanje, vršiće se na osnovu pozicije miša. U Unity-u ovo funkcioniše na način da se pušta nevidljiva linija, odnosno zrak, od kamere do pozicije miša, što nazivamo Raycasting, na osnovu čega dobijamo poziciju na podlozi ka kojoj usmeravamo pogled glavnog lika i određujemo smer za ispaljivanje metaka. Imajući ovo u vidu, kao i to da podloga može biti nekonzistentna

s obzirom na postojanje objekata koji će predstavljati prepreke ovom zraku, trebaće nam poseban objekat koji će imitirati podlogu i imati istu poziciju i orientaciju, ali neće biti u okviru *Environment-a*. Cilj je da zraci mogu stizati do njega bez prepreka. Zbog toga, kreiramo novi 3D objekat Quad, van *Environment-a*, sa dimenzijama (100, 100, 1) i rotacijom 90 stepeni po X. Objekat nazivamo *Floor*, postavljamo ga na istoimeni Layer, i brišemo Mesh Renderer komponentu jer ne želimo da ovaj objekat bude vidljiv.

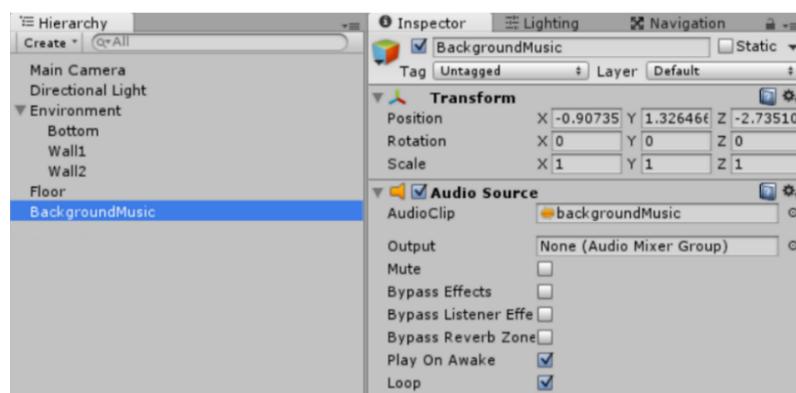
U centru naše scene je sada postavljeno okruženje u vidu ograđene prostorije, ali van tog okruženja je prazan prostor. Ovo popunjavamo kreiranjem nove ravni, tj. Plane objekta. Veličinu novog objekta postavljamo na (20, 1, 20), i ove dimenzijsu su veće u odnosu na celokupno okruženje. Da bismo postigli efekat zamračenog prostora, kreiramo novi materijal u folderu *Materials*, pod nazivom *BackgroundMaterial* i postavljamo Albedo boju na crnu. Pamtimo i prevlačimo materijal na kreirani Plane objekat.

Sada je pametno zapamtiti dosadašnji rad. Kreiramo novi folder *Scenes* u Project View panelu i u njemu preko opcije File / Save scene, ili Ctrl+S na tastaturi, pamtimo scenu.

4.4 Muzika

U cilju poboljšanja iskustva igranja, projektu dodajemo melodiju koja će svirati u pozadini. Za ovo može da posluži bilo koji audio fajl sa Interneta ili iz lične kolekcije. Mi se odlučujemo za besplatnu *hypnotic loop* mp3 melodiju koju preuzimamo sa web stranice <http://www.freesound.org>. U Project View panelu najpre kreiramo novi folder - *Audio*, i ovde uvozimo preuzeti mp3 fajl, prevlačenjem, ili korišćenjem opcije Import New Asset iz kontekstnog menija istog panel-a.

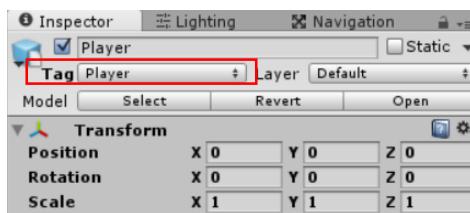
U hijerarhijskom panelu kreiramo novi Empty Object objekat, koji neće biti vizuelno predstavljen na sceni. Menjamo njegov naziv u *BackgroundMusic*. Korišćenjem Inspector-a, novom objektu dodajemo komponentu AudioSource. Ovoj komponenti treba postaviti atribut Audio clip na preuzeti mp3 fajl iz *Audio* foldera. Od ostalih opcija AudioSource komponente, čekiramo opciju Loop, kako bi se pesma ponavljala iznova i iznova, i PlayOnAwake, kako bi zvuk počeo sa reprodukcijom odmah pri pokretanju igre. Naravno, kasnije možemo dodati opciju gašenja i paljenja zvuka, tzv. Sound On / Off opciju.



Slika 4.5: Postavljanje pozadinske muzike

4.5 Glavni karakter

Sledeći koraci razvoja naše igre odnose se na dodavanje glavnog karaktera na scenu i implementaciju kontrole kretanja i animacije. Kao objekat glavnog karaktera koristimo složeni 3D model uvezen iz Asset prodavnice. Pronalazimo ga u *Models / Characters* folderu, pod nazivom *Player*. Možemo ga prevući direktno na scenu ili u hijerarhijski panel, nakon čega postaje vidljiv. Da bismo dali Unity-u do znanja da je ovo glavni karakter, postavljamo vrednost polja Tag, u Inspector-u ovog objekta, na Player. Podatak o Tag-u objekta možemo koristiti za pristup istom iz koda.



Slika 4.6: Postavljanje Tag-a objektu glavnog karaktera

Skaliranje objekta glavnog lika podešavamo po želji. Mi ostavljamo standardne vrednosti (1, 1, 1), dok za njegovu poziciju biramo centar sobe. Y komponentu pozicije postavljamo na 0.1, jer će na taj način objekat biti malo iznad podloge.

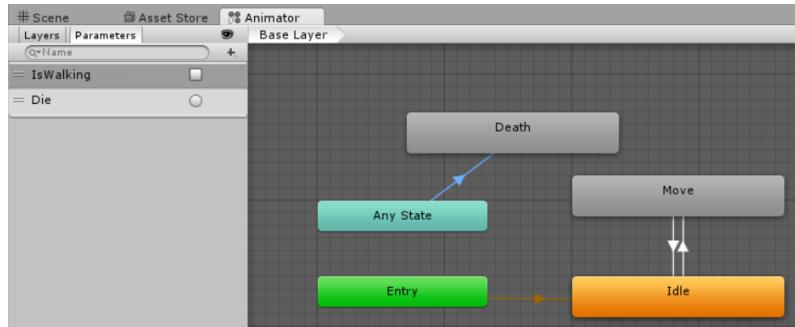
Ukoliko u ovom trenutku pokrenemo igru, vidimo glavnog lika u centru kreirane prostorije, ali on je statičan, baš kao i ostatak scene. Da bismo ovo promenili, moramo uraditi dve stvari. Prvo, treba napisati programsku skriptu koja omogućava pomeranje objekta očitavanjem komandi koje zadaje igrač na tastaturi. Druga je omogućavanje prikaza animacija, koje ovaj objekat ima kao predefinisane. Za to je potrebno kreirati tzv. mašinu stanja, na osnovu koje će Unity znati u kom se stanju model trenutno nalazi i koju animaciju treba pokrenuti. Na primer, ukoliko igrač zadaje komande za kretanje, model objekta glavnog karaktera prelazi u stanje kretanja, pa se pokreće odgovarajuća animacija kretanja. Ukoliko nema komandi, glavni karakter je u takozvanom *Idle* stanju, kome odgovara animacija mirovanja. Na ovaj način dobijamo efekat prilično realnog kretanja humanoidnog modela.

4.5.1 Kontrola animacija

Za kontrolu animacija i stanja objekata u Unity-u koristi se **Animator Controller**. Iz Project panela, projektu dodajemo novi folder - *Animation*, i u okviru njega kreiramo novi Animator Controller sa nazivom *PlayerAC*. Ovu komponentu prevlačimo preko *Player* objekta da bismo mu je pridružili. Pokretanjem *PlayerAC* kontrolera otvara se Animator prozor. Ovde najpre treba definisati stanja u kojima objekat može da bude, i njima opcionalno pridružiti odgovarajuće animacije. U okviru modela objekta glavnog lika, pronalazimo animacije *Idle*, *Move*, i *Death*. U isto vreme ovim su predstavljena sva njegova moguća stanja koja, prevlačenjem animacija u Animator, kreiramo. Stanje *Idle* je standardno, i to potvrđujemo odabirom opcije *Set As Layer Default State* iz njegovog kontekstnog menija u Animator-u.

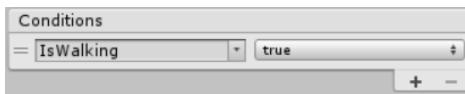
Sledeći korak je implementacija logike, i prelaska iz jednog stanja u drugo. U ovu svrhu, koriste se parametri. Animator prikazuje listu parametara u Parameters tabu. Parametri u

Unity-u mogu biti tipa Bool, Int, Float, i Trigger. Mi kreiramo dva parametra, *IsWalking* tipa Bool, i *Die* tipa Trigger. Trigger može imati vrednost True i False, a razlika u odnosu na Bool je u tome što se kod trigera vrednost odmah nakon postavljanja na True, vraća na False. Ovo je korisno za nešto što se desi jedanput, a onda se resetuje.



Slika 4.7: Player Animator Controller sa stanjima i parametrima

Kreirane parametre treba na neki način iskoristiti. Treba postaviti uslove prelaska iz jednog stanja u drugo. Klikom desnog tastera miša na stanje *Idle*, biramo opciju Make Transition, a onda levim tasterom miša biramo u koje stanje objekat treba da pređe. To je stanje *Move*. Klikom na strelicu koja se pojavila između *Idle* i *Move* stanja otvaraju se podešavanja vezana za tu tranziciju. Nama bitno podešavanje je pod grupom Conditions, a tiče se konfiguracije uslova. Ovde biramo parametar *IsWalking* i postavljamo vrednost na True. Ovo znači da će objekat preći iz stanja *Idle* u stanje *Move*, onda kada je zadovoljen uslov da parametar *IsWalking* ima vrednost True. U tom trenutku se menja i animacija.



Slika 4.8: Podešavanja tranzicije i uslov prelaska iz stanja *Idle* u stanje *Move*

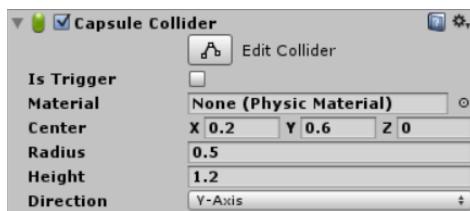
Na isti način postavljamo tranziciju iz stanja *Move* u stanje *Idle*, a uslov je False vrednost parametra *IsWalking*. Kako glavni karakter može izgubiti energiju i život u bilo kom stanju, postavljamo tranziciju iz bilo kog stanja, tj. *Any State*, u stanje *Death*, a uslov je vrednost trigera *Die*.

U nastavku, kreiramo pozive za promenu parametara animacije iz koda, i tako iniciramo prelazak iz jednog stanja u drugo, iz jedne animaciju u drugu.

4.5.2 Fizika i kontrola kretanja

Kako je *Player* objekat koji interaguje sa okolinom, potrebno je dodati mu komponentu Rigidbody. U okviru ove komponente podešavamo Constraints, i to: isključujemo promenu pozicije po Y, jer se glavni lik kreće po ravnoj površini i ne želimo da skakuće ili propadne, i isključujemo rotaciju po X i Z, jer se on okreće samo levo i desno, tj. rotira oko svoje ose - Y ose.

Sledeća komponenta koju dodajemo *Player* objektu, je **Capsule Collider**, na osnovu koje se omogućava interakcija sa drugim objektima. Podešavamo centar, radius, visinu i smer. Collider daje objektu fizičku prisutnost na sceni.



Slika 4.9: Capsule Collider komponenta

Uz model glavnog karaktera, u paketu asset-a kojeg smo preuzeli iz prodavnice, dolaze i zvučni efekti. U Audio Source komponenti *Player* objekta, postavljamo *Player Hurt* audio fajl, koji će se pokretati usled primanja udarca od strane neprijatelja. U okviru iste komponente, dečekiramo Play On Awake opciju, kako se ovaj zvuk ne bi reprodukovao pri samom kreiranju *Player* objekta, na početku igre.

Pišemo prvu skriptu, i pravimo logiku za kontrolu kretanja glavnog lika. Kreiramo novi folder *Scripts* u Project panelu, i u okviru njega dodajemo novu C# skriptu, sa nazivom *PlayerMovement*. Ovu skriptu prevlačimo u hijerarhijski panel, i spuštamo na objekat glavnog lika, čime mu je pridružujemo. Dvaklikom otvaramo skriptu, i krećemo sa kodiranjem.

PlayerMovement skripta:

```
public class PlayerMovement : MonoBehaviour {
    public float speed = 3f;           // brzina kretanja (menjamo iz Inspector-a)
    float currSpeed;                 // trenutna brzina
    Vector3 movement;                // vektor smera kretanja
    Animator anim;                   // referenca Animator komponente
    Rigidbody playerRigidbody;      // referenca Rigidbody komponente
    float camRayLength = 100f;        // dužina zraka koji se pušta od kamere
    int floorMask;                  // layer maska koju gađa zrak
    bool fastMove = false;           // kretanje većom brzinom (Shift taster)

    void Awake() {
        currSpeed = speed;           // trenutna brzina je standardna
        floorMask = LayerMask.GetMask("Floor"); // kreira layer masku za Floor
        anim = GetComponent<Animator>(); // referenca Animator komponente
        playerRigidbody = GetComponent<Rigidbody>(); // referenca Rigidbody komponente
    }
    void FixedUpdate() {
        float h = Input.GetAxisRaw("Horizontal");
        float v = Input.GetAxisRaw("Vertical");

        Move(h, v);
        Turning();
        Animating(h, v);
    }
    void Move(float h, float v) {
        movement.Set(h, 0f, v);
        // Shift taster - brzina kretanja veća 1.5 puta
        if (Input.GetKey(KeyCode.LeftShift) || Input.GetKey(KeyCode.RightShift))
            fastMove = true;
        else
            fastMove = false;
        currSpeed = fastMove ? speed * 1.5f : speed;
        movement = movement.normalized * currSpeed * Time.deltaTime;
        playerRigidbody.MovePosition(transform.position + movement);
    }
}
```

```

void Turning() {
    Ray camRay = Camera.main.ScreenPointToRay(Input.mousePosition);
    RaycastHit floorHit;

    if(Physics.Raycast(camRay, out floorHit, camRayLength, floorMask)) {
        Vector3 playerToMouse = floorHit.point - transform.position;
        playerToMouse.y = 0f; // rotacija po Y je 0
        Quaternion newRotation = Quaternion.LookRotation(playerToMouse);
        playerRigidbody.rotation = newRotation;
    }
}

void Animating(float h, float v) {
    bool walking = h != 0f || v != 0f;
    anim.SetBool("IsWalking", walking);
}
}

```

Awake() funkcija se poziva jednom pri učitavanju skripte, bez obzira da li je ona omogućena (enabled) ili ne, a koristi se za inicijalizaciju promenljivih, kao i za referenciranje komponenti i drugih game objekata. **Start()** funkcija se koristi za razmenu informacija između istih. Poziva se nakon Awake(), ali pre Update(). **Update()** funkcija se poziva svakog frejma, i služi za implementaciju ponašanja objekata, očitavanje Input-a, pomeranje objekata na koje ne utiče fizika i drugo. Interval između dva poziva ove funkcije nije konstantan, i sklon je promenama, iz razloga što određeni frejmovi zahtevaju veće procesiranje podataka, a drugi manje. **FixedUpdate()** služi za rad sa fizikom, izračunavanjima, i Rigidbody komponentama, a ponaša se kao Update(). Razlikuje se po tome što su intervali poziva ove funkcije jednaki i konstantni.

Svakog frejma se proverava da li korisnik zadaje komandu za kretanje glavnog karaktera. Na osnovu ovih komandi, izabranog smera kretanja, i prethodne pozicije, računa se nova koju postavljamo kao trenutnu. Kontrola kretanja se vrši preko strelica na tastaturi ili A, W, S, D tastera, a brzina kretanja se povećava držanjem Shift tastera.

Želimo da *Player* bude uvek okrenut ka poziciji miša, kako bi kasnije omogućili pucanje u istom smeru. Ovo postižemo tako što projektujemo nevidljivu liniju, ili zrak, od kamere, ka poziciji miša i dobijamo tačku na podlozi, ka kojoj rotiramo *Player* objekat. U zavisnosti od toga da li korisnik pritiska komande za kretanje ili ne, postavlja se vrednost *IsWalking* parametra u Animatoru, što omogućava tranziciju iz stanja *Idle* u stanje *Move*, i obrnuto, a to dodatno vuče i aktivaciju odgovarajuće animacije.

4.6 Glavna kamera

Igra će imati jednu glavnu kameru, koja gleda odozgo, i prati kretanje glavnog lika. Pozicija kamere i rotacija se podešava proizvoljno, po želji i oku developer-a. Mi biramo (0, 15, -22) poziciju, i rotaciju po X od 30 stepeni. Projekcija će biti perspektivna, a ugao vidnog polja (Field Of View) 30 stepeni. Kamera standardno ne prati niti jedan objekat, već stoji na poziciji koju smo mi odredili. Međutim, kao i bilo koji drugi objekat, i kameru možemo kontrolisati korišćenjem skripte. Pa tako, kreiramo novu C# skriptu, *CameraScript*, i smeštamo je u *Scripts* folderu. Ovu skriptu prevlačimo na objekat glavne kamere u hijerarhijskom prikazu, i na taj način je ovom objektu i pridružujemo.

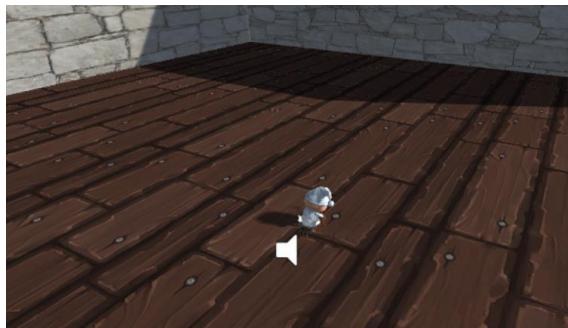
CameraScript skripta:

```
public class CameraScript : MonoBehaviour
{
    public Transform target;           // koji objekat prati kamera
    public float smoothing = 5f;      // brzina prelaza kamere na novu poziciju
    Vector3 offset;                  // inicijalni offset od target-a

    void Start() {
        offset = transform.position - target.position;
    }
    void FixedUpdate() {
        Vector3 newCamPos = target.position + offset;
        transform.position = Vector3.Lerp(transform.position, newCamPos, smoothing * Time.deltaTime);
    }
}
```

Promenljive koje su javne (public) u skriptama mogu da se koriguju i podešavaju direktno u Inspector-u objekta koji tu skriptu sadrži, pa se iz ovog razloga često i koriste. U ovom slučaju promenljivu *smoothing*, koja predstavlja brzinu prelaza kamere na novu poziciju, postavljamo na 5, ali u toku testiranja igre iz Game View panela možemo podešavati ovu vrednost u Inspector-u, i videti koja vrednost daje najbolji efekat. Takođe ostavljamo mogućnost da ciljani objekat može da se promeni u Inspector-u, tako da kamera prati bilo koji, a ne nužno objekat glavnog karaktera. Kako je nama cilj da upravo *Player* objekat bude praćen kamerom, ovaj objekat prevlačimo na poziciju *target* u komponenti skripte Inspector-a glavne kamere.

Na početku, postavljamo vektor razlike pozicija kamere i glavnog karaktera, kao *offset*, a u svakom sledećem frejmu se nova pozicija kamere izračunava kao nova pozicija Playera, plus *offset*. Tu poziciju ne postavljamo direktno, jer bi prelazak bio preoštar i neprijatan za oko, pa koristimo *Lerp* funkciju za glatko prelaženje iz jedne pozicije u drugu.



Slika 4.10: Scene View igre sa glavnim karakterom u centru

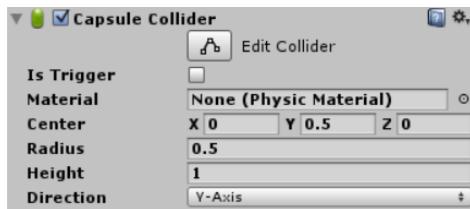
4.7 Neprijatelj

Kako smo već završili uvoz paketa *3 Free Characters* iz Asset prodavnice u projekat, modele neprijatelja možemo pronaći u folderu *Free Characters / Prefabs*. Postoje 3 različita modela, i svi oni zapravo predstavljaju modele drvoseče, ali kako u rukama drže sekire, i deluju neprijateljski nastrojeno, nama će poslužiti kao objekti koji napadaju glavnog lika, odnosno kao neprijatelji.

Prevlačimo model *lumberjack1* na scenu, i krećemo sa sređivanjem i dogradnjom ovog objekta, dodavanjem fizike, animacije i skripte za kretanje.

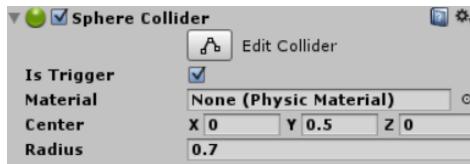
4.7.1 Fizika

Najpre dodajemo komponentu Rigidbody koju podešavamo na isti način kao i kod glavnog karaktera. Ograničavamo rotaciju samo na Y, a kretanje na X i Z osu. Opcija Use Gravity ostaje uključena. Da bi neprijatelj imao osobinu fizičke prisutnosti na sceni, dodajemo Capsule Collider komponentu, sa podešavanjima kao na slici 4.11.



Slika 4.11: Capsule Collider komponenta neprijatelja

Sledeću komponentu dodajemo radi detekcije kolizije neprijatelja i glavnog lika. Tačnije, neprijatelj će biti u mogućnosti da detektuje igrača u blizini, i odmah kreće u napad. Koristimo **Sphere Collider** komponentu. Biramo opciju Add Component / Sphere Collider, uključujemo Is Trigger opciju i podešavamo dimenziju tako da bude veća od prave dimenzije objekta. Na ovaj način ćemo imati informaciju kada glavni lik bude dovoljno blizu, i pre prave kolizije ovih objekata. Komponentu Sphere Collider podešavamo kao na slici 4.12, a Scene View prikaz iste vidimo na slici 4.13.



Slika 4.12: Komponenta Sphere Collider



Slika 4.13 Scene View pregled Sphere Collider komponente

Trigger Collider objektu koji ga sadrži ne omogućava fizičku prisutnost, što znači da drugi objekti ne mogu naleteti i imati fizički uticaj na njega, ali se svaki put kada dođe do presecanja ovog Collider-a sa Collider-om drugog objekta, okida triger koji poziva neku funkciju. Funkciju koja se usled kolizije poziva, implementira se kroz programski kod skripte. Funkcionalnost Trigger Collider-a iskoristićemo u cilju iniciranja napada neprijatelja na glavnog lika, kada se on nađe u blizini.

4.7.2 Audio

Da bismo što uverljivije predstavili napade i reakcije na primljene udarce, koristimo zvučne efekte. Za reprodukciju zvučnih efekata u Unity-u, koristi se komponenta Audio Source. Nama su potrebni različiti zvučni efekti za svaki tip neprijatelja, i to kada oni budu pogodeni, ili izgube život. Mi preuzimamo odgovarajuće fajlove sa adrese <http://www.freesound.org>, i prebacujemo u *Audio* folder projekta. Neprijatelju postavljenom na sceni dodajemo Audio Source komponentu i prevlačimo željeni mp3 fajl u Audio Clip polje. Ovaj zvučni efekat se reprodukuje samo u trenucima primanja udaraca, pa opciju Play on Awake isključujemo. Odgovarajuće zvučne efekte usled izgubljenog života puštamo iz skripte.

4.7.3 Kretanje

Sledeća komponenta je vrlo bitna. Unity Game AI nudi sistem koji se naziva **Nav Mesh**. Ovaj sistem podrazumeva najpre određivanje prohodnih delova scene, a zatim postavljanje agenta koji će automatski moći da se kreće po sceni, pametno, izbegavajući objekte preko kojih ne može da pređe. Objekat koji se prati određuje se iz skripte. Dakle, da bismo postigli efekat praćenja glavnog lika od strane neprijatelja, treba postaviti **Nav Mesh Agent** komponentu neprijateljskim objektima. Postavljamo height atribut na 0.8, radius na 0.3 i speed na 3. Vrednosti ovih atributa biramo na osnovu veličine samog objekta neprijatelja, a kako su nama svi neprijatelji istih dimenzija, i ove vrednosti će biti iste za sve. U Navigation Unity panelu, pod opcijom Bake, postavljamo height i radius na iste vrednosti. Iz ovog panela radimo takozvano pečenje scene, klikom na dugme Bake. Radi se o tome da Unity uzima u obzir sve prepreke u prostoru, i generiše prohodne delove scene koje potom koristi Nav Mesh Agent u navođenju objekata. Svaka promena okruženja koja utiče na prohodnost, zahteva ponovno pečenje scene.

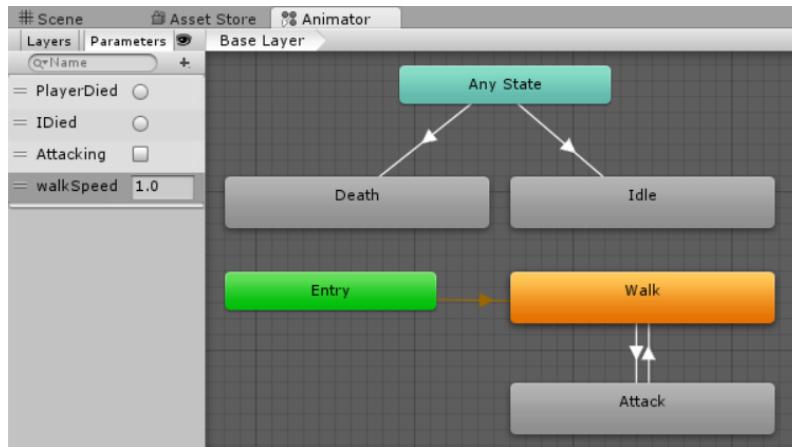
Objekte neprijatelja treba postaviti na Shootable Layer, čime obaveštavamo Unity da na njih može da se puca. Ova informacija će i nama biti od koristi pri pisanju koda skripti.

4.7.4 Animacija

Poput glavnog lika, i neprijatelji će imati različite animacije u zavisnosti od stanja u kom se nalaze. Za kontrolu animacija, kao i ranije, koristimo Animator Controller. U folderu *Animation* kreiramo novi Animator Controller pod nazivom *EnemyAC*. Ovde prevlačimo animacije iz paketa modela neprijatelja, jer one ujedno predstavljaju i stanja u kojima neprijatelj može da bude. Imamo animaciju za *Idle*, *Walk*, i *Lumbering* stanja. *Lumbering* stanju menjamo naziv u *Attack*, i dodajemo još jedno prazno stanje – *Death* (desni klik / Create State / Empty) koje se odnosi na smrt neprijatelja. Ovo stanje neće imati animaciju.

Kao parametre ubacujemo dva trigera *PlayerDied*, i *IDied*, koji se odnose na smrt glavnog karaktera i smrt samog neprijatelja, i jedan Bool *Attacking*, koji služi za označavanje stanja napada. Dodajemo i parametar *walkSpeed*, tipa Float. Ovaj parametar predstavlja brzinu animacije. Naime, brzina animacije je standardno 1, međutim, u cilju otežavanja igre, možemo povećavati brzinu kretanja neprijatelja, pa da bi animacija ispratila ubrzano kretanje, i sama mora da se ubrza. Pošto se ovo odnosi samo na animaciju kretanja, klikom na stanje

Walk, u Inspector-u kao Multiplier brzine biramo kreirani *walkSpeed* parametar. Njegovu vrednost kasnije kontrolišemo i modifikujemo iz programskog koda skripte.



Slika 4.14: Animator Controller neprijatelja

Standardno stanje neprijatelja je stanje *Walk*, a iz bilo kog stanja u stanje *Idle* on prelazi ukoliko je glavni karakter pobeđen, pa kao uslov tranzicije *Any State – Idle*, postavljamo triger *PlayerDied*. Iz stanja *Walk* u stanje *Attack* neprijatelj prelazi kada je parametar *Attacking* jednak true, i obrnuto kada je jednak false. U stanje *Death* se prelazi pod uslovom trigera *IDied*.

Veza između maštine stanja, kontrole kretanja i animacija je neka logika, koju realizujemo kroz programski kod. Za početak, pišemo skriptu kontrole kretanja neprijatelja.

4.7.5 Skripta

Skripta za kretanje neprijatelja treba da omogući praćenje glavnog lika na sceni. Na početku, referencira se samo pozicija glavnog lika i NavMeshAgent komponenta. U *Update()* funkciji postavlja se trenutna pozicija glavnog lika kao tačka koju agent neprijatelja treba da prati, i ovo se izvršava svakog frejma. Kako kreiramo tek prvog neprijatelja, i ne vodimo podatak o tome koliko je neprijatelja na sceni, još uvek ne možemo koristiti različite brzine kretanja neprijatelja i pripremljeni *walkSpeed* parametar animacije. Za sada, neprijatelji imaju jedinstvenu brzinu kretanja, pa tako i standardnu brzinu animacije.

EnemyMovement skripta:

```

public class EnemyMovement : MonoBehaviour {
    Transform player;           // referenca Transform komponente Player objekta
    NavMeshAgent nav;          // NavMeshAgent komponenta neprijatelja
    void Awake() {
        player = GameObject.FindGameObjectWithTag("Player").transform;
        nav = GetComponent<NavMeshAgent>();
    }
    void Update() {
        // NavMeshAgent neprijatelja prati novu poziciju igrača
        nav.SetDestination(player.position);
    }
}
  
```

Zaključno sa ovom skriptom, rezultat dosadašnjeg rada je okruženje predstavljeno ograđenom prostorijom, u čijem centru se nalazi glavni karakter, čije kretanje kontrolišemo pomoću tastature i miša, a koga sve vreme prati jedan objekat neprijatelja. To je to. Fali nam

logika koja bi nam omogućila implementaciju pucanja na neprijatelje iz puške, kao i napade na glavnog lika od strane neprijatelja, sekirama. Da bismo ovo uspešno realizovali, a pored toga implementirali i uslove smrti neprijatelja i glavnog lika, potrebno je uvesti praćenje energije ovih objekata.



Slika 4.15: Igra nakon dodavanja neprijatelja

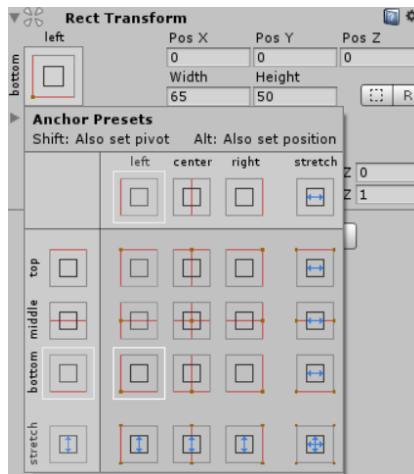
4.8 Glavni lik protiv neprijatelja

4.8.1 Energija glavnog lika

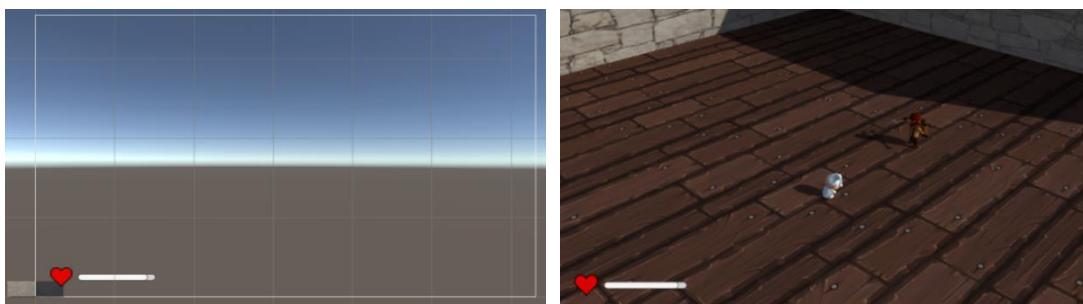
Kao što smo napomenuli, naš glavni lik mora imati svoju energiju ili snagu. Na početku ona će biti na maksimumu. Primanjem udaraca, energija će se smanjivati. Informaciju o preostaloj energiji korisnik treba sve vreme imati pred sobom. Ovo možemo implementirati na dva načina. Prvi podrazumeva korišćenje 3D objekata, koje bismo pridružili objektu glavnog lika, i na taj način postigli stalnu vidljivost na ekranu, s obzirom da glavna kamera sve vreme prati njegovo kretanje. Drugi način je korišćenje ovoj svrsi namenjenih 2D Unity elemenata, kako za 2D, tako i za 3D igre.

Najpre, kreiramo novi objekat, tipa **Canvas**, koji pronalazimo u podmeniju UI, i dodeljujemo mu naziv **2DCanvas**. Obeležavanjem novog objekta i pritiskom na taster F, on dolazi u fokus. Vidimo kreirani 2D objekat, ali, čini se van scene. Radi se samo o načinu na koji Unity vrši prikaz Canvas-a kroz Scene View. Pri pokretanju igre, ovaj Canvas biva nalepljen preko 3D prikaza scene, pa dodatna podešavanja nisu potrebna. **2DCanvas** objektu dodajemo komponentu **Canvas Group**, koja omogućava providnost njegovih elemenata, kao i interakciju. Kako nam interaktivnost Canvas-a u ovom trenutku nije potrebna, jer on ne sadrži dugmiće, ovu osobinu isključujemo. Dečekiramo i **Blocks Raycasts** opciju, jer Raycasting koristimo za usmeravanje glavnog lika. Alpha opciju ostavljamo na 1.

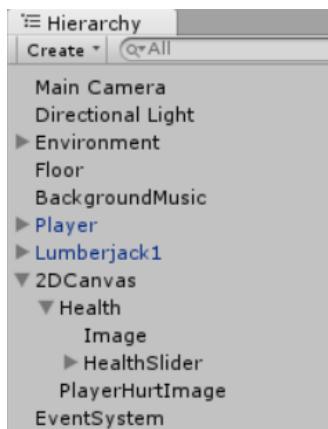
U okviru **2DCanvas**-a dodajemo **Empty Object** koji preimenujemo u **Health**, a u okviru njega kreiramo **Image** objekat iz UI podmenija, koji preimenujemo u **Heart**. Proizvoljnu sliku srca sa Interneta uvozimo u Project View panel i prevlačimo je u komponentu **Image / Source Image**, objekta **Heart**. Slika ne treba da zauzima puno mesta na ekranu, pa tako njenu dimenziju postavljamo na razumnih 30x30 piksela. **Health** objektu postavljamo dimenziju na 65x50, a poziciju na donji levi ugao. To radimo birajući podešavanje sa slike 4.16, uz držanje Alt i Shift tastera, pa na taj način podešavamo i poziciju i pivot.

Slika 4.16: Rect Transform komponenta *Health* objekta

Da bismo vizuelno prikazali procenat preostale energije koristimo **Slider**, još jedan element iz UI menija. Njega kreiramo i dodajemo u okviru *Health* objekta, pod nazivom *HealthSlider*. U okviru *HealthSlider*-a postoji podobjekat *Handle Slide Area*. Ovaj objekat služi za interakciju korisnika sa Slider-om, tj. podešavanje njegove vrednosti. Kako korisnik u toku igre ne može sam podešavati energiju lika koga kontroliše, ovaj podobjekat nam nije potreban, pa ga uklanjamo. Dalje, podešavamo poziciju Slider-a i njegovu dimenziju. Ovo radimo proizvoljno, menjajući vrednosti, i prateći Game View, dok ne dobijemo rezultat koji nam najviše odgovara. Dimenzija koju mi biramo je 100x20, a pozicija (70, 0, 0). Početna energija treba da bude jednaka maksimalnoj, pa vrednosti Value i Max Value postavljamo na 100. Primanjem udaraca, smanjuje se energija, pa treba ažurirati i Value vrednost Slider-a za prikaz. U odnosu na količinu preostale energije i vrednosti Slider-a, može se podešavati i njegova boja, pa tako na primer smanjivanjem energije ona može ići od zelene ka crvenoj, što takođe želimo da implementiramo u cilju postizanja boljeg vizuelnog efekta.

Slika 4.17: Scene View i Game View nakon dodavanja *2DCanvas*-a i *Health* objekata

Kako bismo pojačali vizuelni efekat borbe sa neprijateljima, u trenutku primanja udarca celu scenu će na kratko prekriti nijansa crvene boje. U okviru *2DCanvas*-a kreiramo novi Image objekat, i podešavamo njegovu dimenziju tako da prekriva celu površinu ekrana. Boju ovog objekta postavljamo na crvenu, a vrednost njene Alpha komponente na nulu. Time postižemo da slika na početku igre bude nevidljiva. Načinićemo je vidljivom iz programskog koda, i to tek u trenutku primanja udarca. Kreiranom objektu dodelujemo naziv *PlayerHurtImage*. Prikaz svih objekata hijerarhijskog panela vidimo na slici 4.18.



Slika 4.18: Hierarchy View

Naši sledeći koraci odnose se na pisanje skripte, i omogućavanje funkcionalnosti međusobnih napada između glavnog lika i neprijatelja, usled kojih se gubi energija, i na kraju život. Koristimo postavljene komponente i realizujemo prethodne ideje.

4.8.2 Napad neprijatelja

Kreiramo novu skriptu, *PlayerHealth*, koja će omogućiti praćenje preostale energije glavnog karaktera, i pružiti mu mogućnost da registruje primljene udarce. Usled udaraca se njegova energija smanjuje, i podatak o tome odmah prikazuje korisniku.

PlayerHealth skripta:

```
public class PlayerHealth : MonoBehaviour {
    public int startingHealth = 100;           // početna energija
    public int currentHealth;                  // trenutna energija
    public Slider healthSlider;               // referenca Health Slider-a
    public Image playerHurtImage;             // referenca playerHurt slike
    public AudioClip deathSound;              // referenca zvuka smrti Player-a

    public Color flashColour = new Color(1f, 0f, 0f, 0.1f); // boja playerHurtImage
    public float flashSpeed = 5f;              // trajanje prikaza playerHurtImage slike

    Animator anim;                          // referenca Animatora
    AudioSource playerAudio;                // referenca AudioSource komponente
    PlayerMovement playerMovement;          // referenca playerMovement skripte
    bool isDead;                           // da li je Player mrtav
    bool hurt;                             // da li je Player primio udarac

    Color maxHealthColor = Color.green;     // boja health Slider-a na maksimumu snage
    Color minHealthColor = Color.red;       // boja health Slider-a na minimumu snage
    public Image sliderFillImg;              // slika klizača čiju boju podešavamo

    void Awake() {
        anim = GetComponent<Animator>();
        playerAudio = GetComponent<

```

```

        }
    else {
        playerHurtImage.color = Color.Lerp (playerHurtImage.color, Color.clear,
            flashSpeed * Time.deltaTime);
    }
    hurt = false;
}

public void TakeDamage(int amount) {
    hurt = true;
    currentHealth -= amount;
    healthSlider.value = currentHealth;
    playerAudio.Play();
    if(currentHealth <= 0 && !isDead) {
        Death();
    }
}

void Death() {
    isDead = true;
    anim.SetTrigger("Die");
    playerAudio.clip = deathSound;
    playerAudio.Play();
    playerMovement.enabled = false;
}

public void updateHealthBarColor() {
    sliderFillImg.color = Color.Lerp(minHealthColor, maxHealthColor,
        (float)currentHealth / startingHealth);
}
}
}

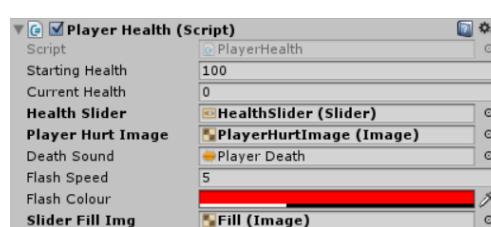
```

U **Awake()** funkciji inicijalizujemo reference *PlayerMovement* skripte, Animator-a, audio izvora, i postavljamo trenutnu energiju na početnu.

U **Update()** funkciji ispitujemo da li je *Player* povređen, i ako jeste postavljamo crvenu boju slike *playerHurtimage*, u suprotnom postepeno skidamo ovu boju kroz vreme određeno *flashSpeed* promenljivom.

TakeDamage(int) funkcija postavlja boolean vrednost *hurt* promenljive na true, smanjuje energiju i ažurira njen prikaz na ekranu, reproducuje audio zvuk prikačen *Player* objektu (*PlayerHurt.mp3*), i proverava da li je on izgubio svu energiju. Ukoliko jeste, poziva se funkcija **Death()** koja postavlja promenljivu *isDead* na true, onemogućuje skriptu kretanja, postavlja i pokreće zvuk koji označava njegovu smrt, i trigeruje Animator komponentu promenom parametra *Die*, što dovodi do promene animacije.

Pamtimo skriptu, i dodajemo je *Player* objektu. Zatim podešavamo javne atribute klase u Inspector-u, prevlačenjem odgovarajućih objekata u data polja, kao na slici 4.19.



Slika 4.19: *PlayerHealth* skripta kao komponenta *Player* objekta

U polje Player Hurt Image, postavljamo prethodno kreiranu sliku iz 2D Canvas-a. Polje Death Sound popunjavamo sa *Player Death* mp3 fajlom iz *Audio* foldera. U polje Health Slider prevlačimo *HealthSlider* UI objekat. Pod ovim objektom pronalazimo Image objekat, naziva *Fill*, koji prevlačimo u Slider Fill Img polje. Njemu se u zavisnosti od količine preostale energije podešava boja, a da bi sve funkcionalnost funkcije **updateHealthBarColor()**. U Inspector-u *HealthSlider* objekta, pod komponentom Slider, postavljamo poziv ove funkcije na On Value Changed događaj. Na ovaj način, svaka promena vrednosti Slider-a automatski poziva funkciju za ažuriranje njegove boje na osnovu preostale energije.

Ovom skriptom smo implementirali funkcionalnost objekta glavnog karaktera da može primiti udarac, što dodatno vuče pozive funkcija za ažuriranje energije, i vizuelni prikaz iste korisniku. Neprijatelj još uvek nema mogućnost da napada, pa upravo na tome radimo u nastavku.

EnemyAttack skripta:

```
public class EnemyAttack : MonoBehaviour {
    public float timeBetweenAttacks = 1f; // vreme između napada
    public int attackDamage = 10;           // koliko energije oduzima jedan napad

    Animator anim;                      // referenca Animator-a
    GameObject player;                  // referenca Player objekta
    PlayerHealth playerHealth;          // referenca skripte PlayerHealth
    bool playerInRange;                // flag - Player u blizini
    float timer;                       // tajmer (meri vreme za sledeći napad)

    void Awake() {
        player = GameObject.FindGameObjectWithTag("Player");
        playerHealth = player.GetComponent<PlayerHealth>();
        anim = GetComponent<Animator>();
    }

    void OnTriggerEnter(Collider other) {
        if(other.gameObject == player) {
            playerInRange = true;
            anim.SetBool("Attacking", true);
            timer = 0f; // reset tajmera
        }
    }

    void OnTriggerExit(Collider other) {
        if(other.gameObject == player) {
            playerInRange = false;
            anim.SetBool("Attacking", false);
        }
    }

    void Update() {
        if(timer >= timeBetweenAttacks && playerInRange) {
            Attack();
        }
        timer += Time.deltaTime;
        if (playerHealth.currentHealth <= 0) {
            anim.SetTrigger("PlayerDied");
        }
    }
}
```

```

void Attack() {
    timer = 0f;
    if(playerHealth.currentHealth > 0) {
        playerHealth.TakeDamage(attackDamage);
    }
}
}

```

EnemyAttack skriptu kreiramo i smeštamo u folder sa ostalim skriptama, i dodajemo je objektu neprijatelja kao komponentu.

U **Awake()** funkciji postavljamo referencu glavnog lika, njegove energije, i Animator-a.

OnTriggerEnter i **OnTriggerExit** su Unity funkcije koje se okidaju kada dođe do preklapanja sa trigerom drugog objekta. Mi ove funkcije koristimo kako bismo znali kada je *Player* objekat blizu ili u kontaktu sa objektom neprijatelja, i isto tako, kada on prestaje da bude u blizini ili u kontaktu sa istim. Najpre proveravamo da li je objekat sa kojim se došlo u kontakt baš *Player* objekat. U **OnTriggerEnter** funkciji, ukoliko je ispunjen ovaj uslov, promenljivu *playerInRange* postavljamo na true. Pored toga, Animator trigeruje vrednost parametra *Attacking*, a vrednost ovog parametra predstavlja uslov da objekat neprijatelja iz stanja i animacije *Walk* pređe u stanje i animaciju *Attack*. Na kraju, postavljamo tajmer na 0. Na osnovu tajmera, određuje se kada je vreme na naredni napad, a kada za pauzu između dva napada. Postavljanjem tajmera na nulu, odbrojavanje do narednog napada kreće ispočetka. U **OnTriggerExit** funkciji radimo sve suprotno.

Update() funkcija ispituje da li je prošlo vreme između dva udarca, i da li je *Player* u okolini. Ako je ovaj uslov ispunjen, poziva se funkcija *Attack()*. Tajmer zatim povećavamo i ispitujemo da li je glavni karakter i dalje živ. Ako nije, Animator postavlja triger *PlayerDied*, što menja animaciju neprijatelja u *Idle*.

U **Attack()** funkciji se resetuje tajmer, i zadaje udarac, ukoliko je *Player*-u preostalo energije.

Ova skripta ima dve javne promenljive koje modifikujemo direktno iz Inspector-a. Testiranjem, došli smo do toga da je čekanje između dva napada najbolje postaviti na 1. Tako dolazi do usklađivanja animacije udarca sekirom sa napadom, kao i zvukom glavnog karaktera.

Pokretanjem igre, vidimo traženi rezultat. Ukoliko se neprijatelj dovoljno približi glavnom karakteru, počinje da ga napada i oduzima mu energiju. Kako glavni karakter još uvek nema mogućnost da koristi svoju pušku i uzvrati udarac, trenutna situacija nije fer, pa to rešavamo u narednim koracima.

4.8.3 Energija neprijatelja

Glavni lik još uvek ne može da koristi svoje oružje i da napada, međutim, isto tako neprijatelj još uvek nema svoju energiju, odnosno snagu. Ova osobina neprijateljskog objekta je ključna, i bez nje nije moguće uspešno realizovati logiku napada. Iz tog razloga najpre vršimo njenu implementaciju.

Pišemo skriptu *EnemyHealth*. Ova skripta treba da drži podatak o preostaloj energiji, što se koristi pri ispitivanju da li je neprijatelj mrtav ili ne. Takođe, treba napisati funkciju za

primanje udarca koji zadaje glavni lik, gde se trenutna energija neprijatelja smanjuje, što prati i odgovarajući zvučni efekat. Da bismo što realnije dočarali napad i postigli bolji vizuelni efekat, pogodak neprijatelja pratiće i animacija raspršavanja krvi iz njegovog tela. Konačno, skripta sadrži i funkciju koja označava smrt neprijatelja, što podiže odgovarajuću animaciju, i reproducuje odabrani zvuk, nakon čega neprijatelj propada kroz tlo i nestaje.

EnemyHealth skripta:

```
public class EnemyHealth : MonoBehaviour {
    public int startingHealth = 100;           // početna energija
    public int currentHealth;                  // trenutna energija
    public float sinkSpeed = 2.5f;            // brzina propadanja kroz tlo nakon smrti
    public AudioClip deathClip;              // audio zvuk nakon smrti
    public GameObject bloodSplat;            // blood particle efekat

    Animator anim;                          // referenca Animator komponente
    AudioSource enemyAudio;                // referenca AudioSource komponente
    CapsuleCollider capsuleCollider;        // referenca Capsule Collider komponente
    bool isDead;                           // da li je neprijatelj mrtav
    bool isSinking;                        // da li neprijatelj propada kroz tlo

    void Awake() {
        anim = GetComponent<Animator>();
        enemyAudio = GetComponent<AudioSource>();
        capsuleCollider = GetComponent<CapsuleCollider>();
        currentHealth = startingHealth;
    }

    void Update() {
        if(isSinking) {
            transform.Translate(-Vector3.up * sinkSpeed * Time.deltaTime);
        }
    }

    public void TakeDamage(int amount) {
        Vector3 bloodPosition = this.transform.position;
        bloodPosition.y = 0.5f;
        bloodPosition.z += 0.2f;
        Instantiate(bloodSplat, bloodPosition, this.transform.rotation);

        if(isDead)
            return;
        enemyAudio.Play();
        currentHealth -= amount;
        if(currentHealth <= 0) {
            Death();
        }
    }

    void Death() {
        isDead = true;
        capsuleCollider.isTrigger = true;
        anim.SetTrigger("IDied");
        enemyAudio.clip = deathClip;
        enemyAudio.Play();
        GetComponent<NavMeshAgent>().enabled = false;
        GetComponent<Rigidbody>().isKinematic = true;
        isSinking = true;
        Destroy(gameObject, 2f);
    }
}
```

Awake() funkcija postavlja reference, i vrednost trenutne energiju na početnu.

Update() funkcija proverava da li je neprijatelj mrtav, tj. u stanju propadanja kroz tlo, pa ako jeste, pomera ga u smeru na dole, i to brzinom `sinkSpeed` u sekundi. Da bi ova brzina bila po sekundi, a ne po frejmu, imamo množenje sa `Time.deltaTime`.

Funkcija `TakeDamage(int)` se ne poziva u ovoj skripti, ali je javna, pa će se koristiti u drugoj, i to u trenutku kada glavni lik upuća neprijatelja. Funkcija najpre inicira kreiranje objekta koji predstavlja animaciju rasprštavanja krvi, na dатој poziciji i sa datom rotacijom. Objekat koji koristimo u ovu svrhu preuzeli smo iz besplatnog *Blood Splatter* paketa Unity Asset prodavnice. Pored kreiranja ovog vizuelnog efekta, funkcija `TakeDamage(int)` smanjuje energiju neprijatelja, reproducuje pridruženi zvuk koji označava da je neprijatelj pogoden, i ukoliko je trenutna energija manja ili jednaka nuli, poziva funkciju `Death()`. Ukoliko je neprijatelj već mrtav, tj. `isDead` promenljiva je true, onda se prekida sa izvršenjem `TakeDamage(int)` funkcije. Parametar koji ova funkcija uzima je celobrojnog tipa, i označava koliko se energije oduzima neprijatelju.

Death() funkcija proglašava neprijatelja mrtvim. Promenljiva `IsDead` se postavlja na true, i vrši se promena Audio Source / Audio clip propertija na zvučni efekat smrti. Animator trigeruje `IDied` parametar čime se pokreće tranzicija objekta u stanje *Death*. Ovo stanje nema određenu animaciju, pa bi u ovom trenutku neprijatelj postao potpuno statičan. Međutim, usled promene `isSinking` promenljive na true, i regularnog poziva `Update()` funkcije, ovaj objekat dobija programski dodeljenu animaciju propadanja kroz tlo. U `Death()` funkciji takođe postavljamo `isTrigger` osobinu Capsule Collider komponente neprijatelja na true, kako bi nakon smrti ovaj objekat izgubio fizičku prisutnost, pa se tako prekidaju i dalja izračunavanja vezana za koliziju. NavMeshAgent se isključuje, pa se prekida i praćanje glavnog karaktera. Osobina `isKinematic` komponente Rigidbody se postavlja na true, pa se ni fizika ovog objekta više ne uzima u obzir. Nakon dve sekunde objekat se potpuno uništava, pozivom `Destroy()` funkcije, i konačno nestaje sa scene.

Ovu skriptu pridružujemo objektu neprijatelja, prevlačenjem, ili uz pomoć opcije Add Component iz Inspector-a. Nakon toga, odgovarajući audio snimak prevlačimo iz *Audio* foldera na polje Death Clip, a objekat animacije krvi – *BloodSplat*, u istoimenno polje.

Ovim smo završili implementaciju energije neprijatelja. Kako sada imamo vrlo korisnu informaciju o tome da li je on živ ili nije, iskoristićemo priliku da ažuriramo *EnemyAttack* skriptu. Naime, trenutni uslov napada neprijatelja je da je glavni lik u blizini i da je istekao interval između dva napada. Međutim, može se desiti da je neprijatelj u međuvremenu izgubio život, pa treba i to uzeti u obzir. Naredbom `GetComponent<EnemyHealth>()` pristupamo skripti *EnemyHealth*, a dodatni uslov za napad je `enemyHealth.currentHealth > 0`.

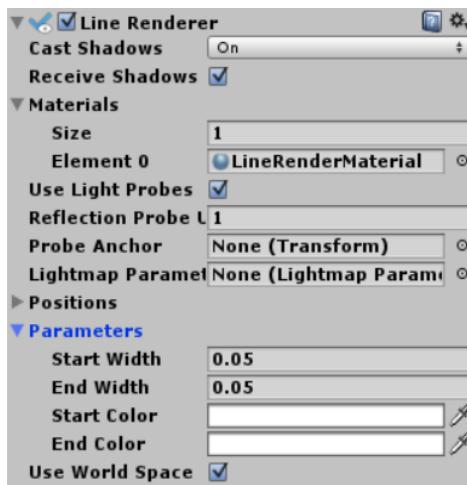
4.8.4 Napad glavnog lika

Konačno smo u situaciji da možemo da implementiramo i funkcionalnost glavnog lika da puca i ubija neprijatelje. Ovo je bitna stavka, i pre kucanja skripte koja će ispaljivati metak i ranjavati neprijatelje, moramo poraditi još malo na modelu glavnog lika. Glavni lik već ima u okviru svog 3D modela ugrađenu pušku, ali da bi se iz nje ispaljivao metak, i da bi sve to

izgledalo realnije, treba dodati još nekoliko elemenata. Prvi u nizu je **Particle System**, koji će služiti za uverljiviji vizuelni prikaz, a zatim **LineRenderer** koji će iscrtavati liniju metka od puške do objekta koji je pogoden. Dodaćemo i audio efekat, kao i svetlo, koji će se zajedno aktivirati usled pucnja, i pojačati ukupan efekat ispaljivanja metka.

Najpre postavljamo Particle System. Pomoću ove komponente, inače, mogu se postići najrazličitiji efekti, od raspršavanja čestica, poput vatrometa, do efekata vatre ili dima. Uz model glavnog lika, u Folderu *Prefabs*, imamo i *GunParticleSystem* objekat. Iz ovog objekta kopiramo Particle System komponentu, klikom na opciju Copy Component iz Settings menija koji se pojavljuje klikom na Cog dugme u gornjem desnom uglu komponente. U okviru *Player* objekta nalaze se podobjekti *Player*, *Gun*, i *GunBarrelEnd*. Nazivi podobjekata govore o tome šta oni zapravo predstavljaju. Nas trenutno interesuje *GunBarrelEnd*, jer je to objekat koji predstavlja vrh puške, i iz kojeg se ispaljuje metak. Cilj je na ovoj poziciji prikazati kopiranu Particle System komponentu, pa je upravo zato i priključujemo ovom objektu. Klikom na Cog dugme bilo koje komponente *GunBarrelEnd* objekta, biramo opciju Paste Component as New, čime završavamo kopiranje Particle System-a. Označavanjem ove komponente, i prelaskom na Scene View, dobijamo opciju Simulate, koje reprodukuje njen efekat.

Dalje, *GunBarrelEnd* objektu dodajemo i LineRenderer komponentu. U okviru atributa Materials ove komponente, pod Element 0 podatributom dodajemo *LineRenderMaterial*, materijal koji je došao u paketu sa modelom glavnog lika. Ovaj materijal određuje boju i izgled linije metka. Otvaramo atribut Parameters i postavljamo Start Width i End Width na željenu dimenziju. Ova dva atributa predstavljaju širinu linije na svom početku i kraju.



Slika 4.20: LineRenderer komponenta *GunBarrelEnd* objekta sa podešavanjima

Kako metke ispaljujemo iz skripte, potrebno je još samo onemogućiti LineRenderer komponentu, dečekiranjem polja u gornjem levom uglu. U suprotnom, jedan metak bi bio ispaljen i pri samom pokretanju igre.

U trenutku ispaljivanja metka, želimo da deo scene bude na kratko osvetljen, i to baš pri vrhu cevi puške, odakle izlazi metak. Objektu *GunBarrelEnd* dodajemo **Light** komponentu. Ovu komponentu pronađemo pod opcijom Rendering u meniju Add Component, ili jednostavno pretragom po ključnoj reči. Podešavamo boju svetla na nijansu žute, i konačno, isključujemo komponentu, pošto njenu kontrolu vršimo iz skripte.

Pucanje iz pištolja ili puške obično karakteriše i zvuk, pa *GunBarrelEnd* objektu dodajemo i Audio Source komponentu, gde postavljamo željeni audio fajl. Ovaj zvuk ne treba da se čuje pri samom pokretanju igre, pa isključujemo opciju Play on Awake. Takođe, zvuk treba da se čuje samo jednom, bez ponavljanja, pa stoga dečekiramo opciju Loop.

Vreme je za poslednji korak. Imamo glavnog lika, imamo neprijatelja. I jedan i drugi imaju svoju energiju, i funkcionalnost da je usled primljenog udarca gube. Postavili smo svetlo, zvuk i LineRenderer, što bi trebalo da dovoljno dobro imitira ispaljivanje metaka iz puške. Ostalo je samo da sve to povežemo i konačno dobijemo funkcionalnost igrača da puca i ubija neprijatelje. Kreiramo novu skriptu, *PlayerShooting*, koju smeštamo u folder *Scripts*, i prevlačimo na *GunBarrelEnd* objekat u okviru roditeljskog *Player* objekta.

PlayerShooting skripta:

```
public class PlayerShooting : MonoBehaviour {
    public int damagePerShot = 20;           // količina štete po metku
    public float timeBetweenBullets = 0.15f; // vreme između dva pucanja
    public float range = 100f;                // domet metka

    float timer;                           // tajmer intervala između dva pucanja
    Ray shootRay;                         // linija pružanja metka
    RaycastHit shootHit;                  // informacija o pogotku
    int shootableMask;                   // Shootable Layer maska
    ParticleSystem gunParticles;         // referenca ParticleSystem-a
    LineRenderer gunLine;                // referenca LineRenderer komponente
    AudioSource gunAudio;               // referenca AudioSource komponente
    Light gunLight;                     // referenca Light komponente
    float effectsDisplayTime = 0.2f;     // koliko dugo će efekti biti prikazani

    private int shotsFired = 0;           // broj ispaljenih metaka
    private int shotsToReload = 20;        // broj metaka u šaržeru
    private bool reloading = false;       // promena šaržera

    void Awake() {
        shootableMask = LayerMask.GetMask("Shootable");
        gunParticles = GetComponent<ParticleSystem>();
        gunLine = GetComponent<LineRenderer>();
        gunAudio = GetComponent<

```

```
void Shoot() {
    if (shotsFired == shotsToReload) {
        reloading = true;
        shotsFired = 0;
        gunAudio.PlayOneShot(reloadAudio);
    }
    else if (reloading == true && gunAudio.isPlaying) {
        return;
    }
    else if(reloading == true) {
        reloading = false;
    }
    else {
        timer = 0f;
        gunAudio.Play();
        gunLight.enabled = true;
        gunParticles.Stop();
        gunParticles.Play();
        gunLine.enabled = true;
        gunLine.SetPosition(0, transform.position);
        shootRay.origin = transform.position;
        shootRay.direction = transform.forward;
        if(Physics.Raycast(shootRay, out shootHit, range, shootableMask)) {
            EnemyHealth enemyHealth = shootHit.collider.GetComponent<EnemyHealth>();
            if(enemyHealth != null) {
                enemyHealth.TakeDamage(damagePerShot);
            }
            gunLine.SetPosition(1, shootHit.point);
        }
        else {
            gunLine.SetPosition(1, shootRay.origin + shootRay.direction * range);
        }
        shotsFired++;
    }
}
```

Awake() funkcija, kao i uvek, služi za postavljanje referenci. Objekti neprijatelja, zajedno sa svim objektima okruženja postavljeni su na *Shootable* Layer. Ovo znači da na njih može da se puca, i do njih stižu metkovi. *ShootableMask* predstavlja redni broj Layer-a koji nosi naziv *Shootable*.

Update() funkcija treba da pruži glavnu funkcionalnost, da omogući pucanje, i to korisniku prikaže na odgovarajući način. Najpre, inkrementiramo tajmer za vreme izvršavanja prethodnog frejma. Zatim, ukoliko je vreme za naredni pucanj, i igrač je pritisnuo levi taster miša, poziva se funkcija **Shoot()**. Vizuelni efekti svetla i linije metka se isključuju nakon vremena predviđenog za njihov prikaz, pozivom funkcije **DisableEffects()**.

DisableEffects() isključuje LineRenderer i Light komponentu *GunBarrelEnd* objekta, tj. postavlja vrednost enabled atributa na false.

Shoot() funkcija završava sav posao. Zamisao je da puška ima ograničen broj metaka u šaržeru, tako da je neophodno menjati ih. Broj dostupnih šaržera s druge strane nije ograničen. U toku promene šaržera, igrač nije u mogućnosti da puca. Da bismo ispratili ovu ideju, uveli smo promenljive koje prate broj ispaljenih metaka (`shotsFired`), broj metaka u šaržeru (`shotsToReload`), i promenljivu tipa bool (`reloading`), koja govori o tome da li je u

toku promena šaržera. Ukoliko je repetiranje u toku, resetuje se brojač ispaljenih metaka i reprodukuje se odgovarajući zvuk. Čeka se na završetak repetiranja i reprodukcije zvuka, a onda je igrač spremam za napad sa punim šaržerom. Tajmer se tada resetuje na nulu. Particle System se, ukoliko je i dalje uključen, zaustavlja, a onda opet pokreće. Time se izbegava situacija da se usled brzog pucanja, animacija uopšte ne pokrene ukoliko prethodna još uvek nije završena. Zatim se reprodukuje audio zvuk ispaljivanja metka, i prikazuje njegova putanja, uključivanjem LineRenderer komponente. Samo uključivanje ove komponente ne završava posao. Linija ne može sama odrediti svoj početak i kraj. Dakle, krajeve određujemo mi. Trenutna pozicija vrha puške (`transform.position`) biće jedan kraj linije. Drugi kraj određuje se RayCasting-om, na osnovu podataka o poziciji izvora zraka (pozicija vrha puške), i njegovog usmerenja. Puštanjem zraka uz pomoć Raycast funkcije, dobijamo poziciju na *Shootable* Layer-u koju ova linija pogađa. Ovo je istovremeno i drugi kraj naše LineRenderer komponente. Naravno, može se desiti da igrač ne pogodi baš ništa na ovom Layer-u, pa se u tom slučaju drugi kraj linije određuje na osnovu smera i maksimalne dužine prostiranja metka, određene promenljivom `range`. Na osnovu pozicije pogotka, ukoliko se radi o *Shootable* Layer-u, proveravamo da li je pogodjeni objekat neprijatelj, i ukoliko jeste, oduzimamo mu energiju, pozivom funkcije `TakeDamage(int)` nad referencom `enemyHealth` skripte. Na kraju, inkrementiramo broj ispaljenih metaka.

Ovim smo završili logiku koja glavnom karakteru omogućava korišćenje puške. U ovom trenutku, bitno je zapamtiti scenu, i sve otvorene fajlove. Konačno možemo da pokrenemo igru i vidimo rezultat. Glavni lik je u mogućnosti da se kreće, nišani, i puca. Neprijatelj, s druge strane, prati glavnog lika u stopu, i kada se dovoljno približi upućuje udarac. Energija oba objekta se usled napada na njih smanjuje, sve dok na kraju neko ne izgubi život.

Na trenutak se vraćamo na skriptu za kretanje neprijatelja, jer smo uočili potencijalni bug. Pomoću `Update()` funkcije, svakog frejma postavlja se nova pozicija koju neprijateljski objekat treba da prati, a to je trenutna pozicija glavnog karaktera. Pored ovoga ne vrši se nikakva dodatna provera. Međutim, potrebno je ispitati da li su ovi objekti uopšte i dalje na sceni, i u mogućnosti da prate ili budu praćeni. Dakle, dodajemo uslov da su glavni karakter i neprijatelj živi, a kako u *PlayerHealth* i *EnemyHealth* skriptama vodimo podatak o njihovoj preostaloj energiji, referenciramo ove dve skripte i proveravamo date vrednosti. Ukoliko je uslov ispunjen, funkcija nastavlja sa radom kao i do sada, u suprotnom, onemogućuje se agent za praćenje.

Ažurirana EnemyMovement skripta:

```
public class EnemyMovement : MonoBehaviour {
    Transform player;
    PlayerHealth playerHealth;
    EnemyHealth enemyHealth;
    NavMeshAgent nav;

    void Awake() {
        player = GameObject.FindGameObjectWithTag("Player").transform;
        playerHealth = player.GetComponent<PlayerHealth>();
        enemyHealth = GetComponent<EnemyHealth>();
        nav = GetComponent<NavMeshAgent>();
    }
}
```

```

void Update() {
    if(enemyHealth.currentHealth > 0 && playerHealth.currentHealth > 0) {
        nav.SetDestination(player.position);
    }
    else {
        nav.enabled = false;
    }
}
}

```

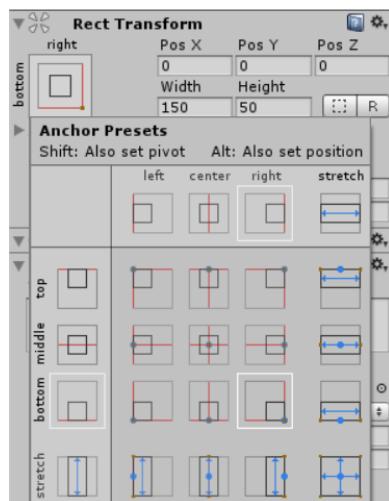
Ažuriraćemo i *PlayerHealth* skriptu. Glavni karakter nakon smrti više ne sme da bude u mogućnosti da puca, pa tada treba isključiti sve efekte vezane za pušku. Najpre kreiramo referencu *PlayerShooting* skripte uz pomoć GetComponentInChildren<PlayerShooting>() naredbe u Awake() funkciji - setimo se da je *PlayerShooting* skripta prikačena za objekat *GunBarrelEnd* koji je dete *Player* objekta. U okviru Death() funkcije onemogućujemo efekte pucanja i isključujemo *PlayerShooting* skriptu, preko playerShooting.DisableEffects() i playerShooting.enabled = false.

Player objekat je u ovom trenutku već kompletiran i spreman za korišćenje, na ovoj sceni, ili nekoj drugoj, u okviru istog, ili sasvim novog projekta uz minimalne izmene. Prevlačimo ga u *Prefabs* folder Project View panela, odakle ga kasnije možemo izvući na bilo koju scenu, ili jednostavno ga instancirati direktno iz koda.

4.9 Score

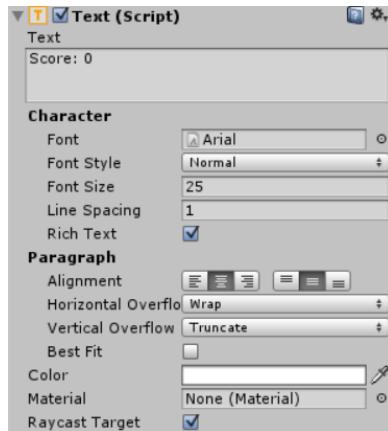
Cilj igara skoro uvek je ostvarivanje što boljeg rezultata. U ovoj igri rezultat se meri brojem poena koje igrač ostvaruje pucanjem i ubijanjem neprijatelja. Tokom igre, igrač treba imati uvid u to koliko je bodova osvojio, a naš zadatak će još biti i kreiranje logike za čuvanje liste najboljih rezultata, kao i njen prikaz.

Trenutni broj poena prikazujemo kroz tekstualni element *2DCanvas*-a. Dakle, kreiramo novi Text UI objekat, dodeljujemo mu naziv *Score*, postavljamo dimenziju na 150x50 piksela, i pozicioniramo ga u donjem desnom uglu *2DCanvas*-a. Za poziciju i poravnanje biramo podešavanja kao na slici 4.21, uz držanje Alt i Shift tastera na tastaturi, čime se postavlja i pozicija i pivot.



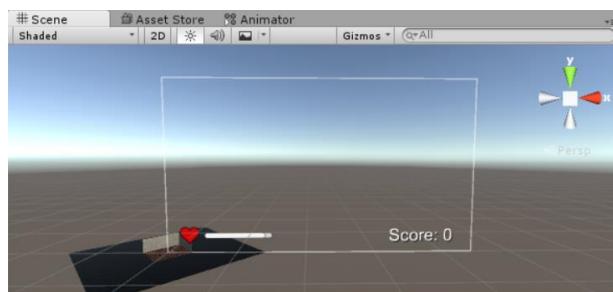
Slika 4.21: Podešavanje poravnjanja Score Text UI objekta

Tekst Score objekta je bele boje, poravnat po sredini, i horizontalno i vertikalno, što podešavamo uz pomoć atributa Color i Alignment. Za početak koristimo standardni, *Arial*, Font. Unity za sada ne podržava automatsko skeniranje Windows Font-ova, već zahteva eksplisitni Import Font-a u projekat, nakon čega je on spreman za korišćenje. Veličinu teksta postavljamo na 25, međutim, kao i ostala podešavanja koja se tiču izgleda i dizajna, najbolje je vrednost odrediti testiranjem i isprobavanjem.



Slika 4.22: Text komponenta Score objekta sa podešavanjima

Tekstu opcionalno dodajemo i efekat senke, pomoću Shadow komponente. Ovu komponentu pronalazimo u Add Component / UI / Effects meniju. Boju senke podešavamo na crnu, a udaljenost na 2 po X, i -2 po Y osi.



Slika 4.23: Scene View nakon dodavanje Score objekta



Slika 4.24: Game View nakon dodavanje Score objekta

Pišemo skriptu koja je kratka i jasna, i ima za cilj ispisivanje uvek najsvežijeg rezultata.

ScoreManager skripta:

```
public class ScoreManager : MonoBehaviour {
    public static int score;          // rezultat
    Text text;                      // referenca Text komponente
```

```

void Awake() {
    text = GetComponent<Text>();
    score = 0;
}
void Update() {
    text.text = "Score: " + score;
}
}

```

Od atributa imamo statičku celobrojnu promenljivu i Text objekat. Statičkoj promenljivoj, koja je pritom i javna, možemo pristupati iz drugih klasa ili skripti. Ona postoji na nivou klase, i zajednička je sviminstancama. Pri detektovanju pogotka ili ubistva, inkrementiramo njenu vrednost.

U **Awake()** funkciji postavljamo referencu teksta, i početni rezultat na nulu.

Update() funkcija svakog frejma upisuje novi rezultat u odgovarajući Text UI objekat.

Rezultat treba da se ažurira svaki put kada neprijatelj pogine ili bude pogoden, što znači da treba da ažuriramo *EnemyHealth* skriptu. Dodajemo novi javni atribut, *scoreValue*, koji predstavlja broj bodova koje igrač dobije za ubistvo neprijatelja. Kasnije ćemo dodavati nove neprijatelje, pa će različiti neprijatelji imati različite *scoreValue* vrednosti. Kako je atribut javni, lako ga modifikujemo iz Inspector-a. U funkciji *Death()*, inkrementiramo trenutni rezultat upravo za vrednost novododatog atributa klase, i to kao: *ScoreManager.score += scoreValue*. Još ažuriramo i *TakeDamage(int)* funkciju, pošto se ona poziva pri svakom primljenom metku, i dodajemo inkrementaciju rezultata za vrednost nove javne promenljive *pointsPerShot*, čiju podrazumevanu vrednost postavljamo na 2. Ovo sve znači da za svaki pogodak neprijatelja iz puške, igrač ostvaruje *pointsPerShot* poena, a za svako ubistvo dodatnih *scoreValue* poena.

Pamtimo scenu i vršimo testiranje urađenog. Rezultat na početku je nula. Pogađanjem neprijatelja, rezultat polako raste, a konačno ubistvom, dobija se još veći broj poena, čiju ukupnu vrednost prati tekstualni prikaz na ekranu.



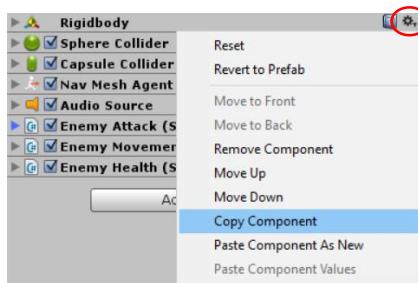
Slika 4.25: Score na početku i nakon ubistva neprijatelja

4.10 Novi neprijatelji

Objekat koji predstavlja neprijatelja je sada kompletan. Dodali smo mu potrebne komponente tako da bude vidljiv i fizički prisutan na sceni. Kreirali smo mašinu stanja koja kontroliše i animacije. Implementirali smo funkcionalnost praćenja glavnog lika, zatim zadavanje i primanje udarca. Sve to prate zvučni i vizuelni efekti.

U paketu *3 Free Characters* iz Asset prodavnice, dobili smo tri 3D modela čovekolikih karaktera, a iskoristili smo samo jedan. Ideja od početka je postojanje više različitih neprijatelja. Različitih, ne samo po pitanju izgleda, već i po snazi, brzini, i drugim karakteristikama. Međutim, logika koja ih prati i kontroliše njihovo ponašanje je ista. Zbog toga, kreiranje novih neprijatelja je vrlo jednostavno, i radi se u par koraka.

Najpre, iz *Free Characters / Prefabs* foldera, na scenu prevlačimo objekte *Lumberjack2* i *Lumberjack3*. Ukoliko otvorimo Inspector, vidimo da je ovim objektima standardno pridružena samo Transform komponenta. Podešavamo poziciju i veličinu po želji, a zatim kopiramo gotove komponente sa prvog neprijateljskog objekta na nove. Klikom na Cog ikonu komponente dobijemo i biramo opciju Copy Component, kao na slici:



Slika 4.26: Kopiranje komponenti sa jednog objekta na drugi

U istom meniju dobijamo i opciju Paste Component As New. U Inspector-u novododatih objekata neprijatelja koristimo upravo ovu opciju, čime završavamo kopiranje. Na prethodnoj slici možemo videti i spisak svih komponenti koje neprijateljima na ovaj način dodajemo, a to su: Rigidbody, Sphere i Capsule Collider, Nav Mesh Agent, Audio Source i skripte.

Novi neprijatelji po nekim karakteristikama, kao što smo već rekli, treba i da se razlikuju. Ne želimo da proizvode iste zvuke, da donose isti broj poena, niti da budu iste snage.

Najpre, vršimo promenu u Audio Source komponentama neprijatelja, i postavljamo nove audio efekte koje smo preuzeli sa Interneta. Podsećanja radi, ove zvučne efekte reprodukujemo pri primanju udarca.

U okviru *EnemyHealth* skripte postavljamo nove *Death* zvuke, koji se reprodukuju nakon smrti, tj. nakon uništenja objekata. Na istoj skripti menjamo i promenljivu *ScoreValue*. Ova promenljiva se odnosi na broj poena koje igrač dobije za ubistvo datog neprijatelja. Postavljamo vrednosti 15 i 20, za drugog, odnosno trećeg neprijatelja.

U skripti *EnemyAttack*, iz Inspector-a podešavamo promenljivu *AttackDamage*, opet na vrednosti 15 i 20, a one se odnose na količinu energije koja se oduzima igraču za primljene udarce.

Da bismo omogućili pucanje na nove neprijatelje, potrebno je postaviti ih na *Shootable* Layer. Nakon ovoga, možemo pokrenuti igru i testirati urađeno.

Možemo zaključiti da je željeni efekat postignut, a promene vidljive. Neprijatelji funkcionišu onako kako smo i zamislili. Logika koja ih pokreće i kontroliše je ista, a opet, pojedinačne karakteristike se razlikuju.

Objekte neprijatelja možemo prevući u *Prefabs* folder, gde se već nalazi *Player* objekat. Kao što smo već naglasili, korišćenje prefabrikovanih objekata je dobra prakse, i omogućava lako ponovno korišćenje istih objekata, sa malo, ili bez ikakvih promena i adaptacije. Pomenimo još i to, da svaka naredna promena ovih objekata ne zahteva ponovno prevlačenje u *Prefabs* folder. Dovoljno je samo upamtiti promene, klikom na dugme *Apply* u gornjem delu Inspector-a.



Slika 4.27: Scene View (iznad) i Game View (ispod)

4.10.1 Dodavanje neprijatelja iz koda

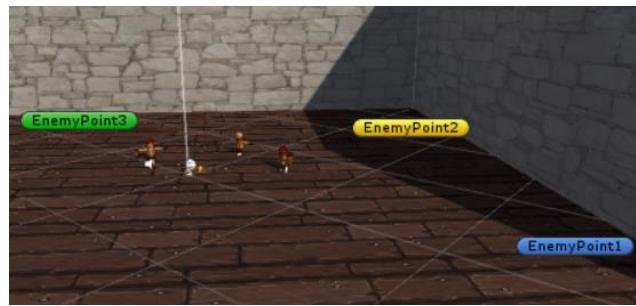
Na sceni se nalaze tri objekta koja predstavljaju neprijatelje. Ukoliko bismo želeli da njihov broj bude veći, morali bismo da kreiramo nove instance kopiranjem ili prevlačenjem iz *Prefabs* foldera, ali ovo nije dobro i održivo rešenje. Mi želimo veći broj neprijatelja, i želimo da se oni stalno iznova kreiraju i pojavljuju na sceni po potrebi. Do rešenje dolazimo kroz kod skripte.

Najpre, potrebno je odrediti tačke ili pozicije gde će se neprijatelji pojavljivati. Kreiramo tri nova Empty GameObject-a u hijerarhijskom panelu, za tri tipa neprijatelja. Preimenujemo ih u *EnemyPoint1*, *EnemyPoint2*, i *EnemyPoint3*. Prazni objekti standardno imaju samo Transform komponentu, što nam je za potrebe određivanja lokacije jedino i bitno. Kasnije nas čeka rad na izgradnji i upotpunjavanju okruženja scene, pa tada treba voditi računa da ove lokacije ostanu slobodne, kako ne bi došlo do neželjenih preklapanja objekata. Tačke pojavljivanja neprijatelja smo odredili kao: (21, 0, 5) sa rotacijom 240 po Y, (3, 0, 21) sa rotacijom 200 po Y, i (-20, 0, 0) sa rotacijom 110 po Y. Rotaciju ovih objekata ćemo koristiti kao odgovarajuću rotaciju neprijatelja, tj. njihovo usmerenje, i iz tog razloga je i postavljamo. Naše vrednosti su izabrane tako da svi neprijatelji budu okrenuti ka početnoj poziciji glavnog karaktera.

Radi lakšeg uočavanja objekata na sceni, Unity je omogućio dodeljivanje oznaka u vidu sličica ili ikona određene boje. Ovo se postiže klikom na kocku obojenih stranica u Inspector-u objekta, nakon čega se bira ikona iz padajuće liste, ili se učitava nova, klikom na dugme Other. Na ovaj način možemo obeležiti početne pozicije neprijatelja.



Slika 4.28: Ikone za označavanje objekata na sceni



Slika 4.29: Označavanje pozicija neprijatelja

Maksimalni broj neprijatelja ograničavamo, i vodimo evidenciju o tome koliko je njih trenutno na sceni. Ove podatke koristimo za ažuriranje *EnemyMovement* skripte, gde na osnovu broja neprijatelja određujemo brzinu sledećeg. Ubistvom neprijatelja, njihov broj se dekrementira, pa treba ažurirati i *EnemyHealth* skriptu.

Najpre, kreiramo novu *EnemyManager* skriptu, za kreiranje neprijatelja i realizaciju prethodno navedenih ideja. Nakon toga vršimo potrebna ažiriranja ostalih skripti.

EnemyManager skripta:

```
public class EnemyManager : MonoBehaviour {
    public PlayerHealth playerHealth; // referenca PlayerHealth skripte
    public GameObject enemy; // referenca objekta neprijatelja
    public float firstEnemyWaitTime = 2f; // vreme do prve pojave neprijatelja
    public float nextEnemyWaitTime = 3f; // vreme do svake naredne pojave neprijatelja
    public Transform enemyPoint; // referenca pozicije pojave neprijatelja

    public static int numOfEnemies = 0; // trenutni broj neprijatelja
    private int allowedNumOfEnemies = 20; // dozvoljeni broj neprijatelja

    void Start() {
        InvokeRepeating ("NewEnemy", firstEnemyWaitTime, nextEnemyWaitTime);
    }

    void NewEnemy() {
        if(playerHealth.currentHealth <= 0f) {
            return;
        }
        else if(numOfEnemies <= allowedNumOfEnemies)
        {
            Instantiate(enemy, enemyPoint.position, enemyPoint.rotation);
            numOfEnemies++;
        }
    }
}
```

Skripta je vrlo jasna. Imamo referencu objekta neprijatelja, njegove početne pozicije, i *PlayerHealth* skripte. Dve public promenljive vode računa o tome kada se vrši kreiranje novog neprijatelja po prvi put, a kada svaki naredni put. Koristimo još dve promenljive, koje govore o trenutnom i maksimalnom dozvoljenom broju neprijatelja na sceni.

Start() funkcija inicira poziv funkcije *NewEnemy()*, prvi put nakon vremena *firstEnemyWaitTime*, a zatim na svakih *nextEnemyWaitTime* sekundi.

NewEnemy() funkcija proverava da li je glavni karakter mrtav, i ukoliko nije, kreira novog neprijatelja na poziciji i rotaciji koje određuje *enemyPoint* objekat, a sve to ukoliko nije premašen dozvoljeni broj neprijatelja.

Dodajemo novi Empty Object objekat u hijerarhijskom panelu - *EnemyManager*, i kako imamo tri različite pozicije za tri različita neprijatelja, dodajemo mu istoimenu skriptu tri puta. U *PlayerHealth* atribut svih instanci skripte prevlačimo *Player* objekat. U *Enemy* atribut prevlačimo odgovarajući objekat neprijatelja iz *Prefabs* foldera, a u *Enemy Point* odgovarajući objekat njegove pozicije. Podešavamo različite intervale prvog i svakog narednog kreiranja neprijatelja, za svaku instance skripte. Inspector *EnemyManager* objekta sa svim podešavanjima vezanim za istoimenu skriptu, možemo videti na slici 4.30.



Slika 4.30: Enemy Manager

EnemyHealth skriptu ažuriramo samo jednim novim redom u *Death()* funkciji, a to je: *EnemyManager.numOfEnemies--*. Dakle, pri ubistvu neprijatelja, dekrementiramo vrednost trenutnog broja neprijatelja na sceni.

EnemyMovement skripta zahteva malo veće izmene. Dodajemo referencu Animator-a pomoću *GetComponent<Animator>()*; naredbe u *Awake()* funkciji. U istoj funkciji dodajemo i sledeće redove:

```
float navSpeed;
if (EnemyManager.numOfEnemies % 20 == 0) {
    navSpeed = 4.5f;
    nav.speed = navSpeed;
}
else if (EnemyManager.numOfEnemies % 10 == 0) {
    navSpeed = 4;
    nav.speed = navSpeed;
}
```

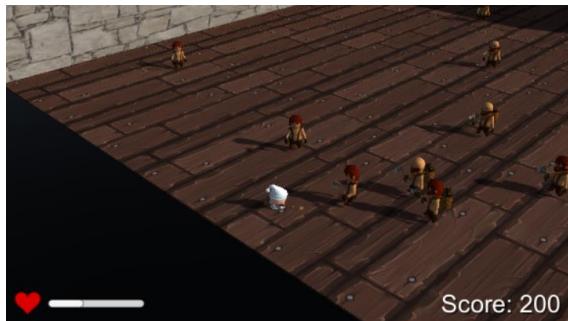
```

else {
    navSpeed = 3;
    nav.speed = navSpeed;
}
enemyAnimator.SetFloat("walkSpeed", navSpeed / normalNavMeshSpeed);

```

Ovde koristimo podatak o broju neprijatelja na sceni i činimo igru težom, time što svakog dvadesetog ili desetog neprijatelja učinimo bržim. Sem povećanja brzine kretanja neprijatelja, povećavamo i brzinu odgovarajuće animacije. Podsećanja radi, promenljiva `walkSpeed`, koja je tipa float, predstavlja parametar kojim se multiplicira brzina izvršavanja animacije stanja *Walk* neprijateljskog 3D modela. Promenljiva `normalNavMeshSpeed` čuva standardnu vrednost brzine praćenja igrača, tj. brzinu kretanja neprijatelja.

Određivanjem početnih pozicija neprijatelja i pisanjem skripte koja kontroliše njihovo kreiranje, više nije potrebno manuelno prevlačiti objekte neprijatelja na scenu, a postojeće možemo i obrisati. Obavezno pamtimo sve promene, čitavu scenu, i testiramo urađeno.



Slika 4.31: Gameplay

4.11 Završni radovi na okruženju

Idea je od samog početka bila da se radnja igre odigrava u nekoj sobi. Mi smo postavili samo pod i zidove, pa okruženje deluje nekako prazno i neispunjeno. Pretražili smo Asset prodavnici, i pronašli besplatni *Free Furniture Props* paket, koji sadrži sasvim solidne modele kreveta, fotelja, vaza, i lampi. Preuzimamo ovaj paket i uvozimo u naš projekat, nakon čega se u Project View panelu pojavljuje novi folder, *BigFurniturePack*. U okviru paketa možemo pronaći i dobre materijale i teksture drveta, tekstila, plastike itd. Sve ovo koristimo da napravimo bolje okruženje, koje će što realnije da oslikava jednu sobu.

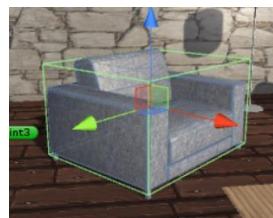
Objekte koje budemo dodavali na scenu postavljamo pod objektom *Environment* u hijerarhijskom panelu, i na *Shootable Layer*-u, kako bi i na njih igrač mogao da puca.

Korišćenje objekata iz preuzetog paketa i njihovo podešavanje se vrši po želji, i isprobavanjem različitih vrednosti atributa sve dok se ne dođe do nekog zadovoljavajućeg rezultata. Mi na scenu postavljamo nekoliko kreveta i fotelja, stočić, par lampi i vaza, i naš krajnji rezultat vidi se na slici 4.32. Ono što treba pomenuti je da smo u okviru *Torchere* modela koji predstavlja lampu, dodali Light komponentu, i to pri samom vrhu lampe, tako da imitira upaljenu sijalicu. Tip svetla je Point, a intenzitet ima vrednost 3. Za tepihe smo iskoristili Plane 3D objekat sa *roof* materijalom, mada se može iskoristiti bilo koji drugi, a nije loša ideja pretražiti Asset prodavnici u potrazi za odgovarajućim.



Slika 4.32: Scene View (nakon sređivanja *Environment-a*)

Nakon postavljanja novih objekata okruženja na scenu, potrebno je dati im i fizičku prisutnost, što se postiže dodavanjem Box Collider komponente. Veličinu ove komponente podešavamo tako da grubo predstavlja dati objekat, što vidimo na primeru fotelje na slici 4.33.



Slika 4.33: Komponenta Box Collider

Zatim, čekiramo i opciju Static, koju nalazimo u gornjem desnom uglu Inspector-a čitavog *Environment* objekta, a kada nas Unity pita o tome, pamtimo ovu promenu i za svu decu objekte. Na kraju, da bi Nav Mesh agent neprijatelja bio svestan novih prepreka, i mogao uspešno da prati glavnog lika, moramo ponovo “ispeći” celo okruženje. Iz Navigation prozora, biramo opciju Bake.



Slika 4.34: GamePlay

Pri kreiranju ovog projekta, ali i svakog drugog, na scenu se automatski postavlja svetlo i glavna kamera. Pozicija svetla nije toliko bitna, pošto se radi o Directional tipu. Bitna je samo rotacija, a to je ugao pod kojim padaju zraci, i na osnovu toga vidimo odgovarajuće senke na sceni. Mi podešavamo glavno svetlo tako da ima rotaciju 30 po X, i 290 po Y.

4.12 UI Meniji

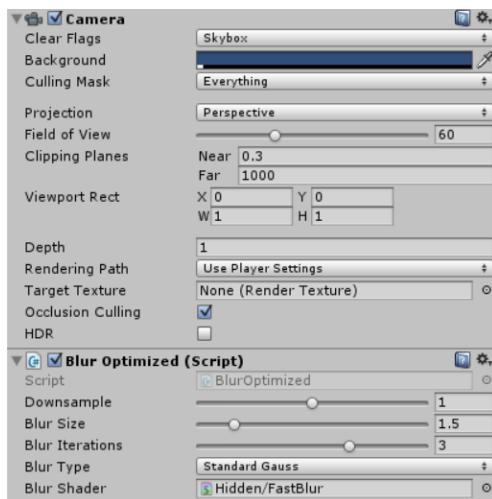
Igra polako dobija svoj konačni oblik. Logiku kretanja, napada, animacije i snage smo već implementirali, a sada imamo i sasvim sređeno okruženje u kojem se radnja odvija.

Međutim, igra pri pokretanju počinje odmah, a pri gubljenju života prestaje bez ikakvih poruka i opcija. Igrač nema mogućnosti da pokrene igru ispočetka, niti da tokom igre napravi pauzu, što su opcije koje moraju biti implementirane. Takođe, još uvek nedostajes lista najboljih rezultata. Sve navedeno možemo realizovati koristeći UI menije. Konkretno, koristićemo 2D Canvas-e, kao kod prikaza preostale energije i rezultata. Kreiraćemo i novi Animator, koji će kontrolisati stanja igre, i na osnovu njih dati odgovarajući prikaz.

4.12.1 Glavni meni

Glavni meni naše igre treba da ponudi korisniku sve bitne funkcionalnosti. Za različita stanja igre, ovaj meni treba da ponudi i različite opcije. Zajedničke opcije svih stanja su: Exit, Highscores i Sound On/Off, tj. zatvaranje aplikacije, prikaz liste rezultata, i paljenje/gašenje zvuka. Ovaj meni pri pokretanju aplikacije treba da ponudi dodatnu opciju za pokretanje igre - Play. Kada igra uđe u gameover stanje, dodatna opcija je Restart, dok se u stanju prekinute igre, sem opcije Restart, dobija i opcija Resume, za nastavak već započete partije.

Kao pozadinu svih UI menija postavljamo nejasni obris scene, koji realizujemo kroz novi Canvas objekat - *BlurCanvas*. Render Mode, Canvas komponente ovog objekta, postavljamo na Screen Space - Camera, a Render Camera atribut na objekat glavne kamere. Plane Distance promenljiva predstavlja udaljenost na kojoj kamera renderuje Canvas, i njenu vrednost postavljamo na 13. Ovom Canvas-u dodajemo još dve komponente da bismo ostvarili ciljni efekat, a to su: Camera, i Blur Image Effect kao deo Standard Assets paketa, sa sledećim podešavanjima:



Slika 4.35: Komponente *BlurCanvas* objekta

Glavni meni realizujemo kroz niz Text i Image UI elemenata, na prozirnoj pozadini, kako bi do izražaja došao prethodno kreirani *BlurCanvas*. Postavljamo ukupno 7 Text UI elemenata na *MainMenuCanvas*-u: *GameOverText*, gde upisujemo poruku o kraju igre, pauzi, ili pozdravnu poruku pri prvom pokretanju, *ScoreText* i *HighscoreText*, gde upisujemo ostvareni i najbolji rezultat pri kraju igre, i još 4 Text UI elemenata koji će služiti kao dugmići, a to su: *PlayText*, *ResumeText*, *RestartText* i *ExitText*. Napomenimo i to da za sve Text UI elemente koristimo Neuropol Font, preuzet sa Interneta. Glavnom Canvas-u dodajemo i dva Image elementa, koja služe za otvaranje liste najboljih rezultata i Sound On/Off funkcionalnost.

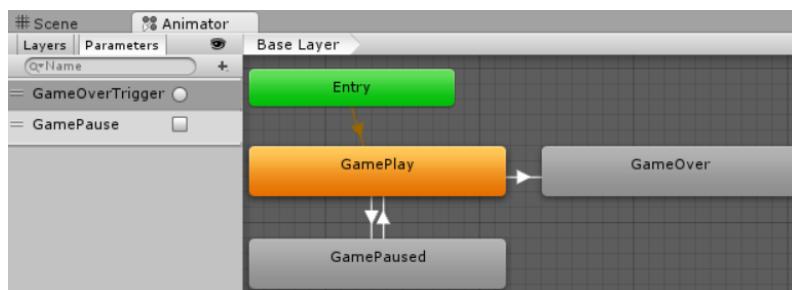
Konačni izgled *MainMenuCanvas*-a sa *BlurCanvas*-om u pozadini vidimo na slici 4.36. Primetimo da se neki elementi ovde preklapaju, no, kako svako stanje igre ima svoju grupu elemenata, ne prikazuju se svi istovremeno, pa ovo neće predstavljati problem.



Slika 4.36: *MainMenuCanvas*

Tekstualnim elementima Canvas-a postavljamo veličinu Font-a, boju, poravnjanje i poziciju po želji. Opciono dodajemo Shadow komponentu crne boje radi postizanja efekta senke teksta. Elementima tipa Image, ali i Text, koji treba da predstavljaju dugmiće i da klikom pozivaju neku funkciju, treba dodati komponentu Button. U okviru ove komponente su podešavanja koja se odnose na izgled dugmeta kada se preko njega pređe mišem, ili se na njega klikne. Dobra je praksa iskoristiti ova podešavanja, jer na ovaj način igrač dobija povratnu informaciju da se nešto dešava i da igra nije ukočena. U On Click () događaju Button komponente kasnije ćemo postaviti akcije koje treba da se izvrše, a kako ćemo dugmiće aktivirati samo iz skripte, obavezno isključujemo polje Interactable. Dugmićima dodeljujemo Tag koji se podudara sa nazivom objekta na koji se odnose, zarad lakšeg pristupanja iz skripte.

Vizuelni prikaz glavnog menija sa svim potrebnim elementima je spreman. Potrebno je organizovati prikaz ovih elemenata u grupe, u zavisnosti od stanja u kojem se igra nalazi. Da bismo ovo realizovali, najpre kreiramo novu mašinu stanja, odnosno Animator. Nazovimo ga *GameStateAnimator*.

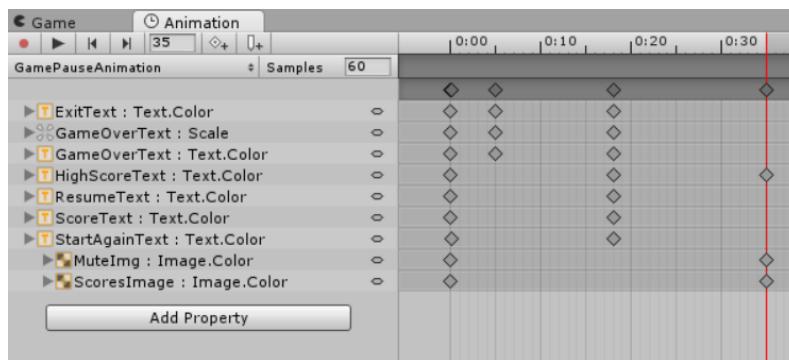


Slika 4.37: *GameStateAnimator*

Igra može biti u toku, pauzirana (po želji), i završena (usled smrti glavnog lika), i upravo su ovo stanja koja *GameStateAnimator* oslikava na slici 4.37. Dodajemo moguće tranzicije između ovih stanja, kao što smo to radili i do sada, a zatim kreiramo i dva parametra kao uslove tranzicija, i to: *GameOverTrigger* parametar, koga postavljamo za uslov tranzicije iz *GamePlay* u *GameOver* stanje, i *GamePause* parametar boolean tipa, koga dvostruko koristimo kao uslov tranzicije između *GamePlay* i *GamePaused* stanja, u oba smera, u zavisnosti od njegove vrednosti. Da bismo upotpunili ovaj Animator, kreiraćemo jednu jednostavnu animaciju i dodaćemo je stanjima *GameOver* i *GamePaused*, i to podešavanjem Motion atributa iz

Inspector-a ovih stanja. Ideja je da elementi glavnog menija u toku trajanja igre budu sakriveni, što postižemo smanjivanjem Alpha vrednosti boja odgovarajućih komponenti na 0. Prelaskom u drugo stanje, ova vrednost će se postepeno kroz animaciju povećavati, tako da se dobije jednostavan fade-in efekat prikaza odgovarajuće grupe elemenata.

Klikom na Animation iz Create menija Project View panela, kreiramo novu animaciju. Naziv animacije postavljamo na *GamePauseAnimation*. Ukoliko se već nije automatski pokrenuo Animation Prozor, pokrećemo ga ručno preko Ctrl+6 prečice na tastaturi, ili iz Window padajućeg menija. U ovom prozoru formiramo animaciju. S leve strane postavljaju se osobine objekata koje se kroz određeni interval frejmova menjaju, i kreiraju animirani prikaz. Klikom na Add Property dugme ovog prozora, dodajemo Text.Color i Image.Color osobine svih Text i Image elemenata *MainMenu* objekta. U početnom frejmu, Alpha vrednost boja svih elemenata je 0. Zatim je potrebno odrediti frejm završetka, i tu postaviti željene vrednosti osobina na kraju animacije. Unity, vrednosti osobina u frejmovima između početnog i krajnjeg, proračunava automatski. Nakon određivanja frejma završetka animacije, biramo opciju Add Key kontekstnog menija Timeline-a, koji se nalazi u desnom delu Animation prozora. Zatim, klikom na dobijene kvadratiće obeležavamo i podešavamo vrednosti osobina iz Inspector-a, što animacija snima i pamti. Ukoliko obeležimo bilo koji frejm između, možemo potvrditi da je Unity proračunao i dodelio odgovarajuće vrednosti osobina. Naša podešavanja su sledeća: Na 18. frejmu Alpha vrednosti boja Text elemenata dostižu maksimum, dok se za Image elemente dešava na 35 frejmu, koji predstavlja ujedno i poslednji frejm animacije. Sem toga, dodali smo skaliranje *GameOverText-a*, gde upisujemo poruku o pauzi ili kraju igre, i to: Na 5. frejmu skaliranje po svim osama postavljeno je na 1.2, dok se u 18. vraća na 1. Na ovaj način kreiramo pop-up efekat. Poruka o trenutnom stanju igre iskače, a zatim se vraća u zadati položaj, što daje zanimljiv vizuelni efekat.



Slika 4.38: Animation Window

Ovim smo *GameStateAnimator* kompletirali. Ne zaboravimo da kreirani Animator prikačimo *MainMenu* objektu, ukoliko to već nismo učinili. Još jedna bitna napomena je da su sve kreirane animacije standardno postavljene na stalno ponavljanje, ili tzv. loop, što nama ne odgovara. Cilj je izvršiti animaciju jedanput, bez ponavljanja. Zbog ovoga, i iz Inspector-a animacije isključujemo ovu opciju.

U hijerarhijskom panelu kreiramo još jedan objekat - *StartScreenInfoHolder*. Objekti jedne scene postoje dok je scena aktivna, a zatim se uništavaju. Međutim u nekim slučajevima potrebno je određene objekte održati u životu jer nose informacije koje se tiču čitave igre, a

ne samo trenutne scene ili nivoa. Ovo se postiže pozivom `DontDestroyOnLoad(GameObject)` Unity funkcije iz skripte prikačene datom objektu. `StartScreenInfoHolder` će biti upravo ovakav objekat, a u nastavku sledi skripta koja ga čini jedinstvenim i održava ga u životu.

DontDestroyScript skripta:

```
public class DontDestroyScript : MonoBehaviour {
    public static bool showStartScreenOnGameStart;
    private static DontDestroyScript instance;

    void Awake() {
        if (instance != null && instance != this) {
            Destroy(this.gameObject);
            showStartScreenOnGameStart = false;
            return;
        }
        else {
            instance = this;
            showStartScreenOnGameStart = true;
            DontDestroyOnLoad(this.gameObject);
        }
    }
}
```

Skripta je vrlo kratka i prilično jasna. Ona nosi statičku referencu same sebe, i u `Awake()` funkciji ovu referencu ispituje. Ukoliko se radi o prvoj instanci skripte, poziva se pomenuta `DontDestroyOnLoad(GameObject)` funkcija. Svaka naredna instanca skripte, koja se kreira pri ponovnom pokretanju tj. restartovanju igre, se odmah uništava. Promenljiva `showStartScreenOnGameStart` nosi informaciju o tome da li treba prikazivati početni ekran, odnosno da li se radi o prvom pokretanju igre, ili ne.

Ovaj objekat savršeno može da se iskoristi i za puštanje pozadinske muzike. Naime, kako je on sve vreme aktivan, pozadinska muzika će se neprestano reproducovati, bez obzira na stanje igre i učitavanje nivoa, što svakako želimo da iskoristimo.

U nastavku, pišemo kod skripte glavnog menija i implementiramo funkcije koje će se pozivati klikom na neke od njegovih ponuđenih opcija.

GameOverManager skripta:

```
public class GameOverManager : MonoBehaviour {
    public static bool gamepaused;           // pauzirana igra
    public bool gameover = false;             // kraj igre

    public PlayerHealth playerHealth;

    public GameObject canvas2D, canvasBlur, canvasPause;
    public Text scoreText, highScoreText, gameOverText;

    Animator anim;
    GameObject restartText, exitText, resumeText, playText;
    GameObject imgScores, imgMute;

    void Awake() {
        anim = GetComponent<Animator>();
        this.GetComponent<Animator>().enabled = true;

        restartText = GameObject.FindGameObjectWithTag("restartText");
    }
}
```

```

exitText = GameObject.FindGameObjectWithTag("exitText");
resumeText = GameObject.FindGameObjectWithTag("resumeText");
playText = GameObject.FindGameObjectWithTag("playText");

imgScores = GameObject.Find("ScoresImage");
imgMute = GameObject.Find("MuteImg");

if (DontDestroyScript.showStartScreenOnGameStart) {
    this.GetComponent<Animator>().enabled = false;
    canvasBlur.SetActive(true);
    canvas2D.SetActive(false);
    canvasPause.SetActive(false);

    playText.GetComponent<Button>().interactable = true;
    exitText.GetComponent<Button>().interactable = true;
    imgScores.GetComponent<Button>().interactable = true;
    imgMute.GetComponent<Button>().interactable = true;

    Color cColor = imgScores.GetComponent<Image>().color;
    cColor.a = 1;
    imgScores.GetComponent<Image>().color = cColor;
    imgMute.GetComponent<Image>().color = cColor;

    cColor = exitText.GetComponent<Text>().color;
    cColor.a = 1;
    exitText.GetComponent<Text>().color = cColor;

    var playText_text = playText.GetComponent<Text>();
    cColor = playText_text.color;
    cColor.a = 1;
    playText_text.color = cColor;

    gameOverText.text = "WELCOME";
    gameOverText.color = new Color(gameOverText.color.r, gameOverText.color.g,
                                    gameOverText.color.b, 1);
    gamepaused = true;
}
else {
    playText.GetComponent<Button>().interactable = false;
    var playText_text = playText.GetComponent<Text>();
    Color cColor = playText_text.color;
    cColor.a = 0;
    playText_text.color = cColor;

    playText.SetActive(false);
    gamepaused = false;
}
}

void Start() {
    gameover = false;
}

void Update() {
    if (!gamepaused && playerHealth.currentHealth <= 0 && !gameover) {
        EnemyManager.numOfEnemies = 0;

        gameOverText.text = "GAME OVER";
        resumeText.SetActive(false);
        anim.SetTrigger("GameOverTrigger");

        int score = ScoreManager.score;
        int highscore = PlayerPrefs.GetInt("HighScore", 1);
    }
}

```

```
    if (score >= highscore) {
        PlayerPrefs.SetInt("HighScore", score);
        scoreText.text = "NEW HIGHSCORE - " + score + " !";
        highScoreText.text = "";
    }
    else {
        scoreText.text = "You scored: " + score;
        highScoreText.text = "Highscore: " + highscore;
    }

    gameover = true;

    exitText.GetComponent<Button>().interactable = true;
    restartText.GetComponent<Button>().interactable = true;
    imgScores.GetComponent<Button>().interactable = true;
    imgMute.GetComponent<Button>().interactable = true;

    canvas2D.SetActive(false);
    canvasPause.SetActive(false);
    canvasBlur.SetActive(true);
}
}

public void setPaused() {
    anim.SetBool("GamePause", true);

    canvasBlur.SetActive(true);
    canvas2D.SetActive(false);
    canvasPause.SetActive(false);

    gamepaused = true;

    resumeText.SetActive(true);
    exitText.GetComponent<Button>().interactable = true;
    restartText.GetComponent<Button>().interactable = true;
    resumeText.GetComponent<Button>().interactable = true;
    imgScores.GetComponent<Button>().interactable = true;
    imgMute.GetComponent<Button>().interactable = true;

    scoreText.text = "";
    highScoreText.text = "";
    gameOverText.text = "PAUSED";
}
}

public void setNotPaused() {
    anim.SetBool("GamePause", false);

    exitText.GetComponent<Button>().interactable = false;
    restartText.GetComponent<Button>().interactable = false;
    resumeText.GetComponent<Button>().interactable = false;
    imgScores.GetComponent<Button>().interactable = false;
    imgMute.GetComponent<Button>().interactable = false;

    canvas2D.SetActive(true);
    canvasPause.SetActive(true);
    canvasBlur.SetActive(false);

    gamepaused = false;
}
}

public void deactivateGameOverCanvas() {
    for (int i = 0; i < this.transform.childCount; i++) {
        var child = this.transform.GetChild(i).gameObject;
        if (child != null)
```

```

        child.SetActive(false);
    }
}
public void activateGameOverCanvas() {
    for (int i = 0; i < this.transform.childCount; i++) {
        var child = this.transform.GetChild(i).gameObject;
        if (child != null)
            child.SetActive(true);
    }
    if (gameover)
        resumeText.SetActive(false);
    if (playText.activeSelf && !DontDestroyScript.showStartScreenOnGameStart)
        playText.SetActive(false);
}
public void RestartGame() {
    UnityEngine.SceneManagement.SceneManager.LoadSceneAsync(0);
}
public void ExitGame() {
    Application.Quit();
}
}

```

Awake() funkcija kreira reference objekata potrebnih za rad. Konkretno radi se o elementima glavnog menija, ali i o Canvas elementima - *Canvas2D* i *CanvasBlur*, koje u zavisnosti od potrebe prikazujemo ili sakrivamo, zatim, Animator-u i *PlayerHealth* objektu. Napomenimo samo da pristup objektima preko Tag-a nije baš efikasna funkcija, pa se upravo zbog toga ovakav pristup najčešće koristi samo u **Awake()** funkciji koja se izvršava samo jednom. Primetimo da smo u okviru ove klase definisali dva bool atributa, gameover i staticki gamepaused, o čijoj nameni govore njihovi nazivi. Dalje, u okviru **Awake()** funkcije proveravamo da li se radi o prvom pokretanju igre. Ukoliko je to slučaj, privremeno isključujemo Animator komponentu, i manuelno aktiviramo grupu elemenata za prikaz na glavnom meniju, i to, podešavanjem Alpha vrednosti boje na 1, i Interactable atributa na true. Ova grupa podrazumeva opcije Play, Exit, Highscores i Sound On/Off. Zatim, uključujemo *CanvasBlur* u pozadini, i pauziramo igru postavljanjem gamepaused propertija na true. Ukoliko prethodni uslov nije ispunjen, odnosno radi se o nekom narednom pokretanju nivoa, a ne prvom, sakrivamo opciju Play, i postavljamo gamepaused na false, čime označavamo da je igra u toku.

Start() funkcija samo postavlja gameover na false.

Update() funkcija stalno, svakog frejma, proverava da li je došlo do smrti glavnog karaktera, a da to već nije procesirano. Ukoliko je ovaj uslov ispunjen resetuje se brojač neprijatelja, trigeruje se parametar *GameOverTrigger* Animator-a čime se pokreće kreirana animacija i prikazuje game over meni. Zatim, deaktivira se Resume opcija, dok se Exit, Restart, Sound On/Off i Highscore opcijama vraća interaktivnost, i konačno, postavlja se vrednost gameover promenljive na true. Ovde pratimo i najbolji rezultat, i upisujemo u odgovarajući Text element. PlayerPrefs je klasa koja se koristi za čuvanje podataka igrača na nivou cele igre, i funkcioniše kao key-value struktura.

Funkcije **setPaused()** i **setNotPaused()** rade upravo ono što njihovi nazivi nagoveštavaju, tj. vrše tranziciju između *GamePlay* i *GamePaused* stanja. *GamePause*

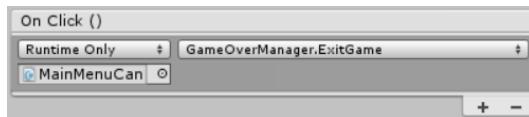
parametar Animator-a postavljaju na true, odnosno false, a isto rade i sa gamepaused promenljivom, i omogućavaju interaktivnost odgovarajuće grupe elemenata glavnog menija.

Funkcija **deactivateMainMenuCanvas()** isključuje sve elemente glavnog menija, dok ih funkcija **activateMainMenuCanvas()** aktivira, sem Resume i Play opcije ukoliko to nije potrebno.

Funkcija **RestartGame()** pokreće ponovo scenu, dok funkcija **ExitGame()** gasi aplikaciju. Napomenimo samo da isključivanje aplikacije ne funkcioniše iz okruženja za testiranje, tzv. Game View prozora, i izvršava se samo iz konačnog build-a aplikacije, na ciljanoj platformi.

Skriptu priključujemo *MainMenuCanvas* objektu ukoliko to već nismo uradili i iz Inspector-a podešavamo public promenljive. Canvas *canvasPause* nismo još uvek kreirali i ovu promenljivu skripte trenutno ostavljamo nepotpunjeno.

U okviru Button komponente Text i Image objekata glavnog menija, koji predstavljaju dugme, podešavamo On Click () događaje. Klikom na znak plus dodaje se nova akcija. U object delu kreirane akcije prevlačimo *MainMenuCanvas* objekat, jer upravo on sadrži *GameOverManager* skriptu. Resume opciji dodajemo funkciju *setNotPaused()*, opcijama Play i Restart funkciju *RestartGame()*, dok će Exit dugme izvršavati *ExitGame()* funkciju iste *GameOverManager* skripte.



Slika 4.39: OnClick događaj Button komponente *ExitText* objekta

Iz Create menija u hijerarhijskom panelu kreiramo novi Canvas objekat na UI Layer-u - *canvasPause*, kojim popunjavamo odgovarajuće polje *GameOverManager* skripte. Pri vrhu Canvas-a dodajemo centrirano poravnat Image objekat - *PauseImg*, proizvoljnih dimenzija. Priključujemo mu Button komponentu koja izvršava *setPaused()* funkciju skripte *GameOverManager*. Krerani Image objekat, iz svega navedenog, predstavlja dugme za pauziranje igre, pa shodno tome, poželjno je priključiti mu sličicu koja će izgledom ukazivati na njegovu funkcionalnost.

Od dugmića glavne forme, sem Highscores opcije koju ćemo kasnije implementirati, samo još Sound On/Off opcija nema priključenu funkciju. Da bismo ovo promenili, kreiramo novu skriptu *ButtonsManager*, koja će kasnije imati nešto širu funkcionalnost, a sada u okviru nje implementiramo funkciju isključivanja i uključivanja zvuka. Priključujemo je nekom objektu koji je uvek aktivan na sceni, za šta može dobro da posluži objekat glavne kamere, tj. *Main Camera* objekat.

ButtonsManager skripta:

```
public class ButtonsManager : MonoBehaviour
{
    public Image imgMute;
    public Sprite MuteOnSprite;
    public Sprite MuteOffSprite;
```

```

void Start() {
    if (!PlayerPrefs.HasKey("Mute"))
        PlayerPrefs.SetInt("Mute", 0);
    MuteOnOff_onStart();
}

private void MuteOnOff_onStart() {
    int muteOnOff = PlayerPrefs.GetInt("Mute");
    if (muteOnOff == 0) { // sound on
        AudioListener.volume = 1;
        imgMute.overrideSprite = MuteOffSprite;
    }
    else { // sound off
        AudioListener.volume = 0;
        imgMute.overrideSprite = MuteOnSprite;
    }
}

public void MuteOnOff() {
    int muteOnOff = PlayerPrefs.GetInt("Mute");
    if (muteOnOff == 0) { // zvuk je uključen, treba da se isključi
        AudioListener.volume = 0;
        imgMute.overrideSprite = MuteOnSprite;
        PlayerPrefs.SetInt("Mute", 1);
    }
    else { // obrnuto
        AudioListener.volume = 1;
        imgMute.overrideSprite = MuteOffSprite;
        PlayerPrefs.SetInt("Mute", 0);
    }
}
}

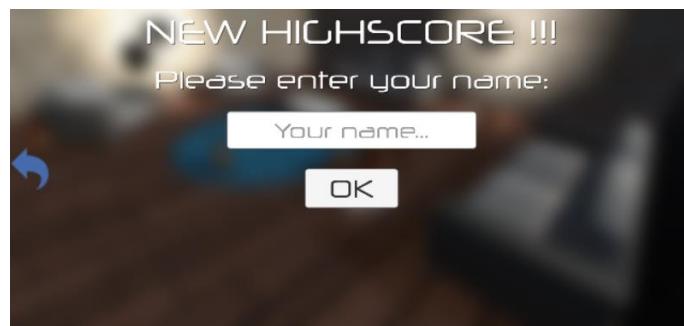
```

Funkcija `MuteOnOff()` invertuje jačinu zvuka, i čuva ovaj podatak kroz `PlayerPrefs` klasu. Sa njim, u mogućnosti smo da pri pokretanju igre učitamo prethodno zapamćeno stanje. Navedenu funkciju dodelujemo kao akciju klika na dugme Sound On/Off glavnog menija. Njeno izvršenje prati i vizuelni efekat, u vidu promene sličice dugmeta, u zavisnosti od toga da li je zvuk uključen ili isključen.

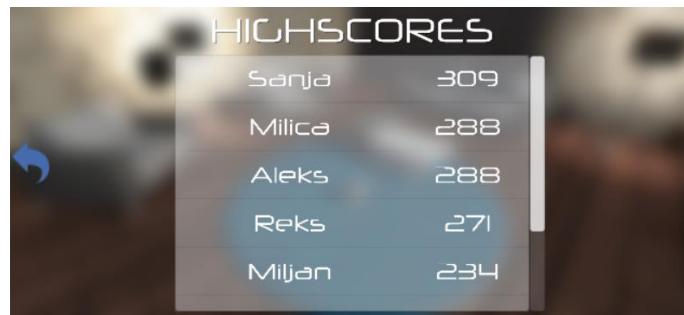
Na kraju, potrebno je ažurirati većinu postojećih skripti, jer u pauziranom stanju igre, o kome sada imamo podatak, nije potrebno izvršavati `Update()` funkciju, i to važi za gotovo sve skripte objekata, a kod nekih bi to čak bilo i pogrešno. Iz ovog razloga, kao uslov izvršavanja `Update()` funkcija dodajemo `GameOverManager.gamepaused == false`, i to u sledećim skriptama: `PlayerMovement`, `EnemyAttack`, `PlayerShooting`, `EnemyManager`, i na kraju `EnemyMovement`. U poslednjoj, potrebno je dodati i `else` granu, gde postavljamo destinaciju Nav Mesh agenta na poziciju samog objekta. Na ovaj način, neprijatelji prestaju da prate glavnog karaktera.

4.12.2 Highscore meni

Naredni zadatak je implementacija logike pamćenja, čuvanja, i prikaza najboljih rezultata. Za ovo su nam potrebna dva nova Canvas objekta, koja kreiramo u hijerarhijskom panelu. Jedan od njih služiće za upis imena igrača i čuvanje ovog podatka zajedno sa ostvarenim rezultatom, dok će drugi služiti za prikaz skrolabilne liste svih najboljih rezultata. Krajnji izgled ovih Canvas-a vidimo na slikama 4.40 i 4.41.

Slika 4.40: *NewHighscoreCanvas*

NewHighscoreCanvas objekat sadrži dva Text elementa, i po jedan Input Field, Button i Image objekat, a sve njih pronalazimo u okviru Component / UI menija. Kao Font svih UI elemenata i dalje koristimo Neuropol Font.

Slika 4.41: *HighscoreCanvas* sa listom najboljih rezultata

HighscoreCanvas je nešto složeniji za kreiranje. Sem jednog Text i jednog Image objekta, ovaj Canvas sadrži i Scrollbar, kao i složeniji *ListPanel* Canvas podobjekat. Boja *ListPanel*-a ostaje standardna bela, ali sa smanjenom Alpha komponentom, radi dobijanja transparentne pozadine. U okviru ovog objekta kreiramo još jedan Canvas objekat koga nazivamo *Grid* i u okviru njega dodajemo objekte koji predstavljaju elemente liste rezultata. Kako *Grid* Canvas može sadržati veći broj elemenata, koji znatno prevaziđa visinu ekrana, to *ListPanel* Canvas-u moramo priključiti Mask komponentu, koja ograničava prikaz njegovog sadržaja. Da bismo igraču ipak omogućili pristup čitavom sadržaju liste, dodajemo ScrollRect komponentu, koja pruža tzv. scroll funkcionalnost, odnosno prelistavanje. U Content polju ove komponente postavljamo *Grid* Canvas, a kao Vertical Scrollbar kreirani Scrollbar objekat. Na ovaj način je omogućeno prelistavanje liste najboljih rezultata. Kako je jedino bitno pomeranje liste po vertikali, čekiramo atribut Vertical, komponente ScrollRect. U okviru iste komponente možemo podestiti i način pomeranja sadržaja, postavljanjem Movement Type atributa, kao i vidljivost, podešavanjem Visibility atributa. Želimo da *Grid* Canvas bude potpuno proziran, a ovo postižemo smanjivanjem Alpha komponente boje na 0. Ovom Canvas-u dodajemo još jednu komponentu, koju do sada nismo koristili, a to je Vertical Layout Group. Korišćenjem ove komponente, ne moramo vršiti nikakva izračunavanja u cilju određivanja pozicije novih elemenata liste, već će oni poziciju, i dimenziju dobijati automatski. Može se odrediti i Padding, Spacing (razmak između dva elementa) i Alignment. Kao poslednju komponentu *Grid* Canvas-a, dodajemo Content Size Fitter. Ova komponenta služi za kontrolu dimenzije objekta koji je sadrži, i to na osnovu dimenzija Layout Element komponenti njegove dece. Content Size Fitter u okviru svojih Vertical i Horizontal Fit atributa nudi sledeće vrednosti: Preferred, Minimum i

None. Preferred prati poželjnu dimenziju, Minimum prati minimalnu, dok None uopšte ne prati dimenziju Layout Element komponenti dece. Mi biramo Preferred opciju kako za Vertical, tako i za Horizontal Fit.

Element *Grid*-a koji odgovara jednom rezultatu, nazivamo *ScoreItem* i kreiramo ga kao Panel UI objekat čiju providnost postavljamo na 25%. U Inspector-u ovog objekta dodajemo Layout Element komponentu, gde podešavamo Preferred Width i Preferred Height atribut, na vrednosti 414 i 55. Druga komponenta koju dodajemo je Horizontal Layout Group. Naime, svaki *ScoreItem* će biti sastavljen iz dva dela - s leve strane će biti podatak o imenu igrača, a s desne broj ostvarenih poena. Mi želimo da njihov prikaz bude horizontalno uređen. Za poravnanje dece *ScoreItem*-a biramo opciju Upper Left, a čekiramo i opcije Width i Height propertija Child Force Expand, čime se deci upućuje naredba da prošire svoje dimenzije kako bi popunili preostali dostupni prostor. Deca *ScoreItem* objekta su Text UI objekti sa centralnim poravnanjem i bojom po želji - *ScoreItemName* i *ScoreItemScore*. Da bi Horizontal Layout Group komponenta roditelja funkcionala, objektima dece dodajemo komponentu Layout, sa popunjениm Preferred Width atributom. Naše vrednosti ovog atributa su 270 i 144 piksela, respektivno. Celokupni *ScoreItem* objekat pamtimo kao Prefab, i uklanjamo ga sa scene.

Sledeći korak je kreiranje nove skripte i pisanje koda koji će pokriti čitavu logiku iza liste najboljih rezultata. Ovu skriptu nazivamo *HighScoreManager*, i dodajemo je objektu *MainMenuCanvas*.

HighScoreManager skripta:

```
public class HighScoreManager : MonoBehaviour {
    public GameObject scoreItemInAList, listObject;
    public GameObject enterNewHighscore, InputTextNewHighscore;
    public GameOverManager gameoverManager;

    List<KeyValuePair<string, int>> highscores;
    int numOfHighScores, maxNumOfHighScores = 10;

    void Start() {
        if (!PlayerPrefs.HasKey("hsNum"))
            PlayerPrefs.SetInt("hsNum", 0);
        fillTheDictionary();
    }

    private void fillTheDictionary() {
        highscores = new List<KeyValuePair<string, int>>();
        numOfHighScores = PlayerPrefs.GetInt("hsNum");
        for (int i = 0; i < numOfHighScores; i++) {
            string playerName = PlayerPrefs.GetString("hsName" + i);
            int playerScore = PlayerPrefs.GetInt("hsScore" + i);
            highscores.Add(new KeyValuePair<string, int>(playerName, playerScore));
        }
        fillAndShowList();
    }

    public void fillAndShowList() {
        foreach (Transform child in listObject.transform)
            Destroy(child.gameObject);

        highscores = highscores.OrderByDescending(v => v.Value).ToList();
        foreach (var score in highscores) {
```

```

GameObject go = (GameObject)Instantiate(scoreItemInAList);
go.transform.SetParent(listObject.transform, false);

var pname = go.transform.Find("ScoreItemName").GetComponent<Text>();
var pscore = go.transform.Find("ScoreItemScore").GetComponent<Text>();
pname.text = score.Key;
pscore.text = score.Value.ToString();
}

}

public bool newHighscoreYesNo(int score) {
    if (highscores == null)
        return true;
    KeyValuePair<string, int> minKVP;
    minKVP = highscores.OrderBy(k => k.Value).FirstOrDefault();
    if (numOfHighScores < maxNumOfHighScores || score >= minKVP.Value)
        return true;
    else
        return false;
}

public void addHighScore() {
    int score = ScoreManager.score;
    string name = InputTextNewHighscore.GetComponent<Text>().text;
    if (name.Length > 0) {
        addNewHighScore(name, score);
        hideEnterNewHighscore();
        gameoverManager.activateMainMenuCanvas();
    }
}

public void addNewHighScore(string name, int score) {
    if (numOfHighScores < maxNumOfHighScores) {
        PlayerPrefs.SetInt("hsScore" + numOfHighScores, score);
        PlayerPrefs.SetString("hsName" + numOfHighScores, name);
        increaseNumOfHihgscores();
    }
    else {
        KeyValuePair<string, int> minKVP;
        minKVP = highscores.OrderBy(k => k.Value).FirstOrDefault();
        if (score >= minKVP.Value) {
            for (int i = 0; i < numOfHighScores; i++) {
                string playerName = PlayerPrefs.GetString("hsName" + i);
                int playerScore = PlayerPrefs.GetInt("hsScore" + i);

                if (minKVP.Key == playerName && minKVP.Value == playerScore) {
                    PlayerPrefs.SetString("hsName" + i, name);
                    PlayerPrefs.SetInt("hsScore" + i, score);
                    break;
                }
            }
        }
    }
    fillTheDictionary();
}

private bool increaseNumOfHihgscores() {
    if (numOfHighScores < maxNumOfHighScores) {
        numOfHighScores += 1;
        PlayerPrefs.SetInt("hsNum", numOfHighScores);
        return true;
    }
}

```

```

        }
    else
        return false;
}

public void showEnterNewHighscore() {
    if (enterNewHighscore.activeSelf == false)
        enterNewHighscore.SetActive(true);
}

public void hideEnterNewHighscore() {
    if (enterNewHighscore.activeSelf == true)
        enterNewHighscore.SetActive(false);
}
}

```

Glavnu ulogu i ove skripte igrat će PlayerPrefs klasa. Uz pomoć nje vodimo evidenciju o ukupnom broju rezultata u listi, ali i o svakom rezultatu (ime igrača i broj poena) pojedinačno. Funkcije `fillTheDictionary()` i `fillAndShowList()` učitavaju podatke iz PlayerPrefs klase u Key-Value strukturu, gde je ključ Ime igrača, a vrednost broj ostvarenih poena. Ovu strukturu sortiramo po broju ostvarenih poena, a zatim element po element, kroz prethodno kreirani `ScoreItem` objekat, ubacujemo u `Grid` Canvas. Funkcija `newHighscoreYesNo(int score)` ispituje da li je ostvareni rezultat dovoljno veliki da bi ušao na listu najboljih. Broj najboljih rezultata ograničili smo na 10. Funkcije `addNewHighScore(string name, int score)` i `addHighScore()` vrše dodavanje novoostvarenog rezultata u PlayerPrefs klasu, i ažuriraju vrednost Key-Value strukture, nakon čega je ona spremna za prikaz igraču. Funkcije `showEnterNewHighscore()` i `hideEnterNewHighscore()` služe za aktiviranje i deaktiviranje Canvas-a za unos novog rezultata.

Nakon svega ovoga, potrebno je ažurirati `GameOverManager` skriptu. Najpre joj dodajemo reference skripte `HighScoreManager`, kao i reference Canvas objekata za unos i prikaz najboljih rezultata. Potom dodajemo funkcije:

```

public void showScores() {
    if (!highScoresCanvas.activeSelf)
        highScoresCanvas.SetActive(true);
    deactivateGameOverCanvas();
}

public void hideScores() {
    activateGameOverCanvas();
    if (highScoresCanvas.activeSelf)
        highScoresCanvas.SetActive(false);
}

public void showEnterNewHighscore() {
    if (enterNewHighscore.activeSelf == false) {
        deactivateGameOverCanvas();
        enterNewHighscore.SetActive(true);
    }
}

public void hideEnterNewHighscore() {
    if (enterNewHighscore.activeSelf == true) {
        enterNewHighscore.SetActive(false);
        activateGameOverCanvas();
    }
}

```

Funkciju `Update()` ažuriramo dodavanjem redova:

```
if (highScoreManager.newHighscoreYesNo(score))
    showEnterNewHighscore();
```

pomoću kojih zapravo proveravamo da li je ostvaren rezultat dovoljno veliki da bi ušao na listu najboljih, i samo u tom slučaju prikazujemo Canvas za upis novog rezultata.

Na kraju, potrebno je iskoristiti implementirane funkcije, i dodati funkcionalnost preostalim dugmićima *MainMenuCanvas*, *HighScoreCanvas* i *NewHighScoreCanvas* UI menija.

Klik na Highscores opciju glavnog menija treba da izvrši *ShowScores()* funkciju skripte *GameOverManager*, a zatim funkciju *fillAndShowList()* skripte *HighScoreManager*.

Back opcija *NewHighScoreCanvas* menija treba da izvršava funkciju *hideScores()* *GameOverManager* skripte, dok ista opcija *HighScoreCanvas*-a treba da izvršava funkcije *hideEnterNewHighScore()* i *activateMainMenuCanvas()*, skripti *HighScoreManager* i *GameOverManager*.

Dugmetu OK, Canvas-a *NewHighScoreCanvas*, koji služi za čuvanje novog najboljeg rezultata, priključujemo *addHighScore()* funkciju skripte *HighScoreManager*.

Sada je dobra ideja sačuvati sve promene. Zatvaramo i čuvamo promene pokrenutih skripti, pamtimo scenu, i pokrećemo igru iz Unity okruženja radi testiranja dodatih funkcionalnosti.

Rad na UI menijima je završen, a sa njima igra izgleda mnogo kompletnije. Korisnik u svakom trenutku ima mogućnost da prekine igru, i da nakon određenog vremena nastavi partiju tamo gde je stao. Ukoliko izgubi život, o tome dobija poruku kroz UI meni koji mu omogućava da kreće igru ispočetka. Dobija povratnu informaciju i o tome koliko je poena osvojio u poslednjoj partiji, kao i najbolji rezultat koji je ikad ostvario. Kada broj osvojenih bodova bude dovoljno veliki, igrač ima mogućnost da upiše svoje ime na listu najboljih rezultata. Istoj može u svakom trenutku da pristupi iz glavnog menija, pri pokretanju igre, usled pauze, ili kada završi partiju. Ukoliko mu pozadinska melodija ne odgovara, ili iz nekog razloga želi da je isključi, pritiskom na jasno označeno dugme iz glavnog menija, to i postiže. Igra se startuje klikom na dugme Play, a ne kao do sada odmah pri pokretanju aplikacije, pa se igraču daje vremena da se pripremi.

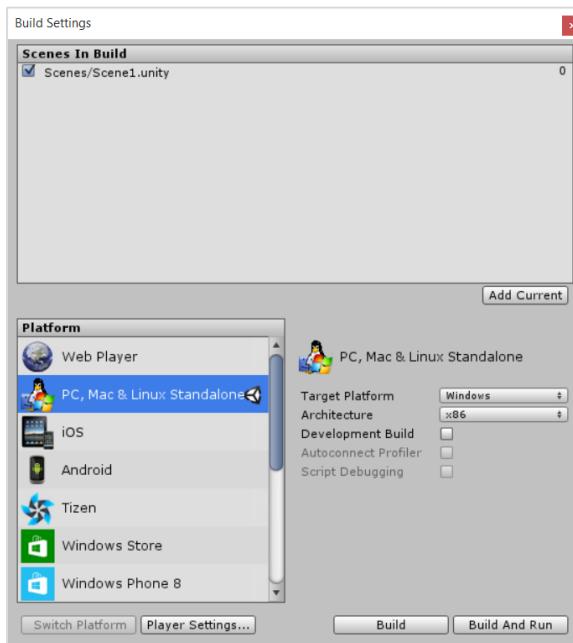
Možemo zaključiti da organizacija igre kroz UI menije čini igru kompletnej, omogućava implementaciju različitih funkcionalnosti, i značajno poboljšava korisničko iskustvo.

5 Windows platforma

Kada smatramo da je igra završena, ili jednostavno želimo da je testiramo na ciljanoj platformi, potrebno je podesiti još par stvari, i izvršiti build-ovanje aplikacije. Iz File menija biramo opciju Build Settings. Otvara se prozor kao na slici 5.1.

Klikom na dugme Player Settings dobijamo listu podešavanja u Inspector-u. Ovde možemo odabrati naziv igre, kompanije, podesiti ikonicu, itd. Dobijamo i više podešavanja vezana za rezoluciju, renderovanje, i izgled početnog ekrana (Splash Screen). Ovde se nalaze i

specifična podešavanja vezana za svaku platformu pojedinačno. Treba napomenuti da nisu sve opcije dostupne u Personal (besplatnoj) verziji Unity-a, što je i logično.



Slika 5.1: Build Settings

Kada zavšimo sa Player Settings podešavanjima vraćamo se na Build Settings prozor i dodajemo scenu, pritiskom na dugme Add Current, ili prevlačenjem odgovarajućeg fajla iz *Scenes* foldera. Zatim, u donjem levom uglu biramo platformu iz poprilično dugačke liste ponuđenih. Naša opcija je **PC, Mac & Linux Standalone**, a nakon odabira bilo koje opcije, dobijaju se njena specifična podešavanja u desnom delu prozora. Ovde biramo tačnu ciljanu platformu i arhitekturu kojoj je aplikacija namenjena. Kako koristimo Windows operativni sistem, naša podešavanja ciljaju Windows platformu sa x86_x64 arhitekturom. Klikom na dugme **Build**, pojavljuje se novi prozor, gde se određuje destinacija za čuvanje potrebnih fajlova, kao i naziv pod kojim će aplikacija biti upamćena. U zavisnosti od složenosti i veličine projekta, proces build-ovanja traje od nekoliko sekundi do nekoliko minuta, a po završetku, otvara se folder aplikacije. U njemu se nalaze svi potrebni fajlovi za pokretanje i rad igre. Igru pokrećemo kroz egzekucioni (.exe) fajl kreiranog foldera. Kopiranjem celokupnog foldera, igru možemo prebaciti na eksternu memoriju, drugi računar ili poslati preko Interneta, a ona će biti funkcionalna na svim računarima koji rade pod istom platformom. Pokretanjem igre i testiranjem, zaključujemo da sve funkcioniše kako treba i kako smo zamislili, a konačno možemo testirati i funkcionalnost zatvaranja aplikacije - opcija Exit, koju nismo mogli da koristimo u Game View prozoru Unity okruženja.

6 Android platforma

Industrija igara iz godine u godinu sve se više razvija i širi, a poseban zamah ostvaruju mobilne igre. Prihodi se već odavno mere u milijardama dolara, i ovakvom trendu se ne vidi kraj. Mobilne igre zarađuju na najmanje dva načina, prodajom u online prodavnicama, i putem mikrotransakcija u samoj igri. Sve popularniji način zarade je preko takozvanih freemium igara,

koje se zvanično besplatno preuzimaju, ali je u njima vrlo teško postići išta značajno bez plaćanja i otključavanja novih elemenata u samoj igri. Ono što je takođe bitno napomenuti je da danas sve popularne igre ciljaju veći broj platformi, a Unity je odličan game engine koji omogućava prilično lak prelazak sa jedne platforme na drugu, što ćemo mi upravo demonstrirati na primeru prelaska sa Windows platforme na Android.

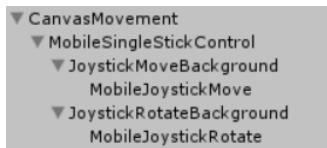
6.1 Kontrola kretanja

Prvi korak pri prelasku na drugu platformu su Build Settings podešavanja kojim se pristupa iz File menija, ili prečicom Ctrl+Shift+B na tastaturi. Naša prethodno zapamćena podešavanja odnose se na Windows platformu. Sada iz liste ponuđenih biramo Android, i to potvrđujemo klikom na dugme Switch Platform. Prebacivanje projekta sa jedne platforme na drugu obično traje od nekoliko sekundi do nekoliko minuta, u zavisnosti od veličine i složenosti samog projekta. U desnom delu Build Settings prozora nalaze se dodatne opcije. Texture Compression opcija se koristi kada ciljamo određenu hardversku arhitekturu. Ukoliko se igra objavljuje u Google Playstore prodavnici, moguće je build-ovati igru za svaki format kompresije posebno, a prodavnica će različitim modelima telefona nuditi odgovarajuću verziju instalacije u zavisnosti od prepoznatog hardvera. Google Android Project opcija omogućava generisanje projekta koji je moguće otvoriti iz Android Studio razvojnog okruženja. Ukoliko želimo da vršimo Debug igre na Android telefonu, i pratimo izvršenje na računaru, potrebno je čekirati opciju Development Build, nakon čega dobijamo dve dodatne opcije - AutoConnect Profiler i Script Debugging, koje omogućavaju Debug koda i korišćenje Profiler-a.

Nakon podešavanja nove platforme, zaista nemamo puno posla. Sva logika ostaje apsolutno ista, nepromenjena. Jedina stvar koju moramo na neki način rešiti je kontrola kretanja glavnog igrača i pucanje iz puške. Za ovu namenu na Windows platformi koristili smo tastaturu i miša. Na Android telefonima nemamo standardno ove uređaje (mada ih na više načina možemo priključiti), pa je Touch jedina preostala, a nekako i logična, opcija. U ovu svrhu najčešće se koriste džojstici, koji kako izgledom, tako i funkcionalnošću podsećaju na prave džojstike igračkih konzola. Za našu igru potrebna su dva džojstika. Jeden za kontrolu kretanja glavnog lika, a drugi za kontrolu njegovog usmerenja i pucanje iz puške.

Najpre uvozimo *CrossPlatformInput* paket, koji dolazi kao deo Unity Standard Assets kolekcije, a može se pronaći u Assets / Import Package meniju, ili preuzeti iz Asset prodavnice. Istoimeni folder ovog paketa nam je dovoljan za nastavak, tako da ostale foldere i fajlove možemo da dečekiramo pri uvozu. Kreiramo novi Canvas objekat u hijerarhijskom panelu - *CanvasMovement*, i podešavamo UI Scale Mode na Scale With Screen Size, kako bi čitav Canvas na različitim rezolucijama izgledao približno isto. Iz foldera *CrossPlatformInput* / *Prefabs* biramo *MobileSingleStickControl* i prevlačimo na kreirani Canvas. U okviru ovog objekta nalaze se *MobileJoystick* i *JumpButton* elementi. *JumpButton* odmah uklanjamo jer nam neće biti potreban. *MobileJoystick* postavljamo pod roditeljski objekat kome u pozadini postavljamo sliku okvira džojstika, nakon čega ovaj objekat dupliramo, jer kao što smo napomenuli, potrebna su nam dva ovakva džojstika. Njihove nazive postavljamo na *MobileJoystickMove* i *MobileJoystickRotate*. Ovi objekti sadrže Image komponentu sa

standardnom sličicom, koju menjamo i postavljamo po želji. Mi smo u ovu svrhu kreirali dve različite slike džojstika, uz pomoć Adobe Photoshop aplikacije.



Slika 6.1: Hjerarhija *CanvasMovement* objekata

Hijerarhijski prikaz novog Canvas objekta vidimo na slici 6.1. Ukoliko se u ovom trenutku jave određeni problemi i dodate komponente nisu prikazane na sceni, potrebno je manuelno uključiti Mobile Input, iz istoimenog padajućeg menija, klikom na opciju Enabled.

Džojstike postavljamo u donjem levom i donjem desnom uglu, gde su najpristupačniji korisniku kada je aplikacija u Landscape režimu. Kako na ovom mestu trenutno stoje pokazivač preostale energije i broj osvojenih poena, ove objekte *2DCanvas*-a pomeramo na gornji deo ekrana. Primetimo da je *MobileJoystick* objektima priključena *Joystick* skripta. Kao public promenljive, koje možemo podešavati direktno iz Inspector-a, dostupne su MovementRange, koja predstavlja opseg pomeranja džojstik objekta, zatim AxesToUse, što se odnosi na ose koje se koriste za očitavanje komandi džojstika (mi koristimo obe), i Horizontal i Vertical Axis Name, koje predstavljaju nazive osa. Kod *MobileJoysticMove* objekta, naziv osa ostaje nepromenjen, standardni: Horizontal i Vertical. Kod *MobileJoysticRotate* objekta, nazive osa menjamo na Horizontal2, i Vertical2, mada oni mogu biti praktično bilo šta. MovementRange u oba slučaja postavljamo na 60.

Prvi džojstik, namenjen za kontrolu kretanja, u ovom trenutku spreman je za dalje korišćenje i implementaciju njegove funkcionalnosti kroz pisanje programskog koda.

Drugi džojstik služi kako za usmeravanje glavnog karaktera, tako i za pucanje. Naime, dok god je ovaj džojstik pritisnut i registruje dodir, ispaljuju se meci iz puške. Da bismo ovo postigli, moramo malo prilagoditi i izmeniti ponuđenu skriptu. Dodajemo public static bool promenljivu rotatePressed. Uz pomoć nje vodimo evidenciju o tome da li je drugi džojstik pritisnut ili nije. U ovoj zamisli će nam pomoći događaji *OnPointerUp* i *OnPointerDown*, *Joystick* skripte. U okviru ovih funkcija postavljamo odgovarajuću vrednost novododatac bool promenljive, i to jedino ukoliko se radi o drugom džojstiku. Koji džojstik je registrovao dodir možemo zaključiti na osnovu naziva osa. Evo i definicija funkcija:

```

public static bool rotatePressed = false;
public void OnPointerUp(PointerEventData data) {
    transform.position = m_StartPos;
    UpdateVirtualAxes(m_StartPos);
    if (horizontalAxisName == "Horizontal2")
        rotatePressed = false;
}
public void OnPointerDown(PointerEventData data) {
    if (horizontalAxisName == "Horizontal2")
        rotatePressed = true;
}
  
```

Na slici 6.2 možemo da vidimo kako izgleda Gameplay nakon postavljanja džojstika, i promena u izgledu i lokaciji elemenata *2DCanvas*-a.

Bez obzira na to što je platforma promenjena, testiranje igre i dalje je moguće iz Game View prozora unutar Unity okruženja. Pri ovome, Touch event se simulira klikom miša. U ovom trenutku pokrećemo igru kako bismo videli rezultat uživo. Džoystici se pokreću, međutim kontrola glavnog lika i dalje nije implementirana.



Slika 6.2: Gameplay

Ažuriramo *PlayerMovement* i *PlayerShooting* skripte. U telu funkcije **Awake()** skripte *PlayerMovement* dodajemo:

```
GameObject canvasMovement = GameObject.Find("CanvasMovement");
if (Application.platform == RuntimePlatform.Android) {
    canvasMovement.SetActive(true);
    Screen.sleepTimeout = (int)SleepTimeout.NeverSleep;
}
else {
    canvasMovement.SetActive(false);
}
```

Na osnovu *Application.platform* atributa imamo informaciju o tome na kojoj platformi je igra pokrenuta. Ukoliko se radi o Android platformi, aktiviramo *CanvasMovement* objekat, dok ga u suprotnom deaktiviramo. Linija koja podešava *sleepTimeout* zapravo daje naredbu Android telefonu da preinači standardna podešavanja vezana za gašenje display-a nakon određenog perioda nekorišćenja.

Postojeću funkciju *Turning()* koja usmerava glavnog karaktera u smeru pozicije miša, preimenujemo u *Turning_Windows()*, zato što sada implementiramo novu funkciju *Turning_Android()*.

```
void Turning_Android() {
    float x = CrossPlatformInputManager.GetAxis("Horizontal2") * 50;
    float z = CrossPlatformInputManager.GetAxis("Vertical2") * 50;
    if (x == 0f && z == 0f)
        return;

    Vector3 floorPoint = new Vector3(x, 0f, z);
    Vector3 playerToFloorPoint = floorPoint - transform.position;
    playerToFloorPoint.y = 0f;

    Quaternion newRotation = Quaternion.LookRotation(playerToFloorPoint);
    playerRigidbody.rotation = newRotation;
}
```

Da bismo uopšte bili u mogućnosti da očitavamo vrednosti džoystika, na početku skripte obavezno stavljamo:

```
using UnityEngine.StandardAssets.CrossPlatformInput;
```

Logika Turning_Android() funkcije se neće razlikovati od početne funkcije. Dakle, potrebna nam je tačka ka kojoj ćemo zarotirati igrača. Koordinate ove tačke dobijamo na osnovu vrednosti koje nam pruža drugi džoystik. Kako su ove vrednosti iz opsega [-1, 1], moramo ih pomnožiti nekom konstantom, tako da uvek dobijemo dovoljno udaljenu tačku na mapi igre. Nastavak funkcije i dodeljivanje rotacije objektu glavnog karaktera je isto kao u postojećoj Turning_Windows() funkciji.

Nova i glavna funkcija Turning() na osnovu prosleđene informacije o platformi poziva odgovarajuću funkciju, pa je sada definisana na sledeći način:

```
private void Turning(RuntimePlatform platform) {
    if (platform == RuntimePlatform.Android)
        Turning_Android();
    else
        Turning_Windows();
}
```

Ažuriramo i FixedUpdate() funkciju *PlayerMovement* skripte. Izmene su minimalne. Na osnovu platforme očitavamo vrednosti Input-a, a ostatak funkcije ostaje isti.

```
void FixedUpdate() {
    if (!GameOverManager.gamepaused) {
        float h, v;

        if (Application.platform == RuntimePlatform.WindowsEditor ||
            Application.platform == RuntimePlatform.WindowsPlayer) {
            h = Input.GetAxisRaw("Horizontal");
            v = Input.GetAxisRaw("Vertical");
        }
        else if (Application.platform == RuntimePlatform.Android) {
            h = CrossPlatformInputManager.GetAxisRaw("Horizontal");
            v = CrossPlatformInputManager.GetAxisRaw("Vertical");
        }
        else
            return;
    }

    Move(h, v);
    Turning(Application.platform);
    Animating(h, v);
}
}
```

U okviru *PlayerShooting* skripte, potrebno je ažurirati Update() funkciju. Ukoliko je istekao tajmer, a radi se o Windows platformi i pritisnut je levi taster miša, ili se radi o Android platformi i pritisnut je džoystik za pucanje, to je znak da treba ispaliti metak i pozvati funkciju Shoot().

```
void Update() {
    if (!GameOverManager.gamepaused) {
        timer += Time.deltaTime;
        bool isWinPlatform = Application.platform == RuntimePlatform.WindowsEditor ||
                            Application.platform == RuntimePlatform.WindowsPlayer;
        bool isAndPlatform = Application.platform == RuntimePlatform.Android;
        bool timerElapsed = timer >= timeBetweenBullets;
        bool mouseClicked = Input.GetButton("Fire1");
        bool rotJoyPressed = Joystick.rotatePressed;

        if ((timerElapsed) &&
            (isWinPlatform && mouseClicked) || (isAndPlatform && rotJoyPressed)))
    }
}
```

```
        Shoot();

        if (timer >= timeBetweenBullets * effectsDisplayTime)
            DisableEffects();
    }
}
```

Napomenimo da će sa ovim izmenama igra i dalje savršeno raditi na Windows platformi, ukoliko se ikad na nju vratimo. Na osnovu svega, možemo zaključiti da je prilično lako kreirati jedinstveni Unity projekat koji bez problema i sa minimalno modifikacijom radi na većem broju platformi.

6.2 Facebook integracija

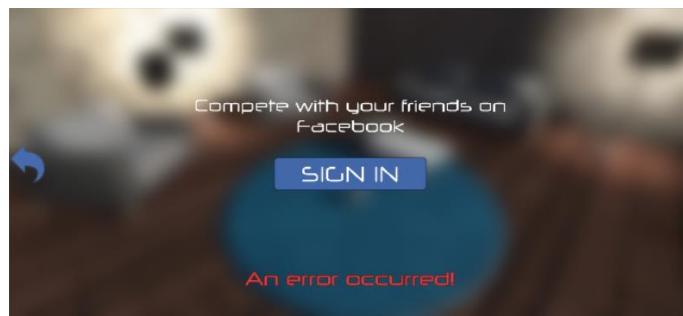
Integracija funkcionalnosti koje nude društvene mreže u igru je dobra praksa. Dobit je višestruka. Najpre, omogućena je brza i jednostavna autentikacija korisnika. I sa strane korisnika je ovo pozitivno, s obzirom da nije potrebno pamtitи još jedan username / password. Zatim, većina API-a društvenih mreža pruža mogućnost međusobnog nadmetanja između prijatelja, kroz praćenje zajedničke liste najboljih rezultata. Ovo motiviše korisnike da duže igraju igru, i ostvare što veći broj poena, kako bi nadmašili rezultate svojih prijatelja. Funkcionalnosti društvenih mreža ne zavise od platforme, pa je ovo najjednostavniji način za povezivanje korisnika aplikacije različitih platformi. Kako je Facebook trenutno najpopularnija društvena mreža, sa najvećim brojem korisnika, i ima odličnu podršku za Unity, odlučujemo se za implementaciju funkcionalnosti koje ona nudi.

Nekoliko je koraka u implementaciji Facebook API-a. Prvi jeste kreiranje Facebook Developer naloga na stranici <https://developers.facebook.com>. Ovaj nalog može biti vezan i za postojeći privatni nalog. Potrebno je ostaviti broj telefona nakon čega se dobija poruka sa kodom za verifikaciju. Drugi korak je kreiranje Facebook aplikacije. Nakon logovanja sa Developer nalogom, iz menija My Apps biramo opciju Add A New App. Pojavljuje se prozor sa ponuđenim platformama, među kojima biramo Android, upisujemo naziv aplikacije, kontakt email adresu i kategoriju. Sledеćih nekoliko stranica korisno je pročitati jer se tiču korišćenja Facebook SDK, implementacije, i drugih podešavanja. Ove kratke smernice se mogu preskočiti klikom na dugme Skip Quick Start, nakon čega se otvara glavna Dashboard stranica. Najbitniji podatak koji ćemo odavde koristiti je App ID - jedinstveni identifikacioni broj aplikacije. Sa stranice <https://developers.facebook.com/docs/unity/> potrebno je preuzeti Facebook SDK i izvršiti uvoz u Unity. Klikom na Assets / Import Package / Custom Package biramo preuzeti fajl i završavamo uvoz. U Player Settings Inspector-u treba podesiti Bundle identifier, Company Name i Product Name. Uvozom Facebook SDK dobija se novi padajući meni - Facebook. Iz ovog menija biramo opciju Edit Settings, nakon čega se sa desne strane otvara Facebook Settings Inspector. Ovde popunjavamo polja App Name i App ID, sa vrednostima koje se nalaze na Developer stranici Facebook aplikacije. S druge strane, potrebno je iz Android Build grupe podešavanja ovog Inspector-a kopirati vrednosti Package Name, Class Name, i Debug Android Key Hash, i nalepiti ih u odgovarajuća polja na Developer stranici. Ovde otvaramo Settings / Basic podešavanja. Klikom na Add Platform dodajemo Android platformu, u okviru koje popunjavamo Google Play Package Name, Class Name i Key Hashes sa kopiranim vrednostima

iz Unity projekta. Promene pamtimo klikom na dugme Save Changes. U ovom trenutku Facebook izbacuje poruku da polje Google Play Package Name nije i ne može biti verifikovano, a razlog leži u tome što igra nije javna niti objavljena na Google PlayStore-u, no nama ovo neće predstavljati nikakav problem u implementaciji.

Developer stranica Facebook aplikacije nudi sijaset opcija, za upravljanje, analizu, uloge, obaveštenja, i druga podešavanja i alate. Na Dashboard stranici možemo pratiti pozive ka našoj aplikaciji, broj korisnika, greške, prosečno vreme odziva, aktivnost korisnika kroz vreme, trendove, a tu su i dodatne opcije za objavljivanje reklama u igri i potencijalnu zaradu. Settings stranica sadrži sva ostala podešavanja, koja se tiču naziva, domena, kontakt mejlova, linkova ka polisama privatnosti i uslovima korišćenja (ukoliko je igra javna i ima svoju web stranu). Tu su i specifična podešavanja aplikacije za različite platforme, kao i podešavanje ikone, kategorije kojoj aplikacija pripada, i, opcionalno, potkategorije. Napomenimo da je za naše potrebe vrlo bitno pravilno odabrati kategoriju - Games. U naprednim podešavanjima su opcije za Age i Country Restriction, Security, način autorizacije, analitike, migracije i kreiranja posebne Facebook Application stranice za korisnike i fanove. Roles stranica je posebno bitna programerima. Naime, na ovoj stranici imamo tri grupe korisnika: Administratore, Developere i Testere, svaka grupa sa različitim nivoom pristupa. Mi ćemo koristiti grupu Testera, i u okviru nje dodati nekoliko Tester korisnika. Ova grupa korisnika ima pristup funkcijama Facebook API-a i pre nego što igra bude javno objavljena za sve korisnike.

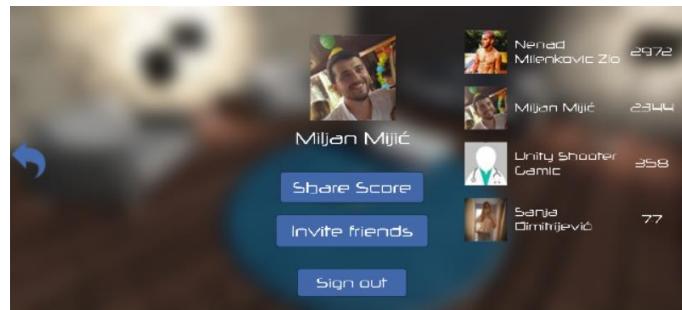
Da bismo uspešno implementirali funkcije Facebook API-a, potrebna nam je nova skripta, dva nova Canvas objekta, ali i ažuriranje postojećeg *MainMenuCanvas* objekta i *GameOverManager* skripte. Najpre kreiramo nove Canvas-e, a to su *FBNotLoggedInCanvas* i *FBLoggedInCanvas*. Prvi Canvas je aktivan za neulogovanog korisnika, i sadrži dva Text UI objekta za prikaz poruka dobrodošlice i greške, Button objekat za logovanje, i Button objekat za povratak na prethodni (glavni) meni.



Slika 6.3: *FBNotLoggedInCanvas*

Drugi Canvas je malo složeniji. Na njemu najpre treba prikazati ime logovanog korisnika, zajedno sa profilnom slikom, za šta koristimo Text i Image objekte. Na isti način na koji smo kreirali skrolabilnu listu najboljih rezultata, kreiramo još jednu listu sa najboljim rezultatima, ali na kojoj se sada prikazaju ostvareni poeni prijatelja sa Facebook-a, nezavisno od platforme na kojoj igraju igru. Implementiramo i funkcionalnost deljenja postignutih rezultata korišćenjem objava na profilnoj stranici, sa prijateljima, ili potpuno javno, klikom na dugme Share Score. Pored toga, korisniku omogućavamo i slanja pozivnica za instalaciju igre

prijateljima koje sam odabere, klikom na dugme Invite Friends. Na kraju, dodajemo dugmiće za Sign out, i povratak na prethodni meni.



Slika 6.4: *FBLoggedInCanvas*

Kreiramo novu skriptu *FBScript* koja će sadržati implementaciju Facebook API funkcija, i koju obevezno priključujemo nekom objektu koji je sve vreme aktivan na sceni, za šta možemo iskoristiti npr. *Main Camera* objekat.

FBscript skripta:

```
public class FBscript : MonoBehaviour
{
    public GameObject loggedInUI, notLoggedInUI, friendObj;
    public Text loadingUI_errText;
    public string myName = "", myScore = "";
    bool errorOrNot = false;

    void Awake() {
        noErrMsg();
        if (!FB.IsInitialized)
            FB.Init();
    }

    public void logIn() {
        if (!FB.IsLoggedIn) {
            List<string> permissions = new List<string>();
            permissions.Add("user_friends, publish_actions");
            noErrMsg();
            FB.LogInWithReadPermissions(permissions, logInCallback);
        }
    }

    public void logOut() {
        if (FB.IsLoggedIn) {
            FB.LogOut();
            loggedInUI.SetActive(false);
            notLoggedInUI.SetActive(true);
        }
    }

    public void showUI() {
        noErrMsg();
        if (FB.IsLoggedIn) {
            notLoggedInUI.SetActive(false);
            FB.API("me?fields=name", HttpMethod.GET, getNameCallback);
            FB.API("me/picture?width=100&height=100", HttpMethod.GET, getPictureCB);
            FB.API("/app/scores?field=score,user.limit=30", HttpMethod.GET, scoresCB);
            if (!errorOrNot)
                loggedInUI.SetActive(true);
        }
    }
}
```

```

        }
    else {
        loggedInUI.SetActive(false);
        notLoggedInUI.SetActive(true);
    }
}

public void hideUI() {
    loggedInUI.SetActive(false);
    notLoggedInUI.SetActive(false);
}

private void showErrMsg() {
    errorOrNot = true;
    notLoggedInUI.SetActive(true);
    loggedInUI.SetActive(false);
    loadingUI_errText.text = "An error occurred!";
}

private void noErrMsg() {
    errorOrNot = false;
    loadingUI_errText.text = "";
}

public static void setScore(int Score) {
    if (FB.Initialized && FB.LoggedIn) {
        string scoreCmd = "/me/scores?field=score";
        FB.API(scoreCmd, HttpMethod.GET, delegate (IGraphResult result) {
            if (result.Error == null) {
                var res = result.ResultDictionary["data"] as List<object>;
                if (res.Count > 0)
                {
                    var entry = (IDictionary<string, object>)res[0];
                    int oldScore;
                    if (int.TryParse(entry["score"].ToString(), out oldScore))
                        if (Score > oldScore)
                            setNewScore(Score);
                }
                else
                    setNewScore(Score);
            }
        });
    }
}

private static void setNewScore(int Score) {
    Dictionary<string, string> scoreData = new Dictionary<string, string>();
    scoreData["score"] = Score.ToString();
    FB.API("/me/scores", HttpMethod.POST, delegate (IGraphResult result)
    {
        Debug.Log(result.RawResult);
    }, scoreData);
}

public void share() {
    string name = "Li'l Killer Game";
    string link = "https://drive.google.com/open?id=0BwyV-ALBQFETQXh1UXRBZmF1Njg";
    var pic = new Uri("http://oi65.tinypic.com/2nu1728.jpg");

    bool scoreExists = !String.IsNullOrEmpty(myScore) && myScore.Length > 0;
    string scoreText = "Can you beat my highscore - " + myScore + "?!";
}

```

```

        string noScoreText = "Li'l Killer is a great Unity shooter game.";
        string msgText = scoreExists ? scoreText : noScoreText;

        FB.ShareLink(new Uri(link), name, msgText, pic);
    }

    public void invite() {
        string msg = "You should really try this game out";
        string tit = "Li'l Killer is a great game!!!";
        FB.AppRequest(message: msg, title: tit);
    }

    private void createFriend(string name, string id, string score) {
        GameObject fr = Instantiate(friendObj);
        Transform parent = loggedInUI.transform.FindChild("ListContainer").
                           FindChild("FriendList");
        fr.transform.SetParent(parent, false);
        fr.GetComponentsInChildren<Text>()[0].text = name;
        fr.GetComponentsInChildren<Text>()[1].text = score;
        FB.API(id + "/picture?width=50&height=50", HttpMethod.GET, delegate
        (IGraphResult result) {
            if (result.Error == null)
                fr.GetComponentInChildren<Image>().sprite =
                Sprite.Create(result.Texture, new Rect(0,0,50,50), new Vector2(0.5f,0.5f));
        });
    }

    private void logInCallback(ILoginResult result) {
        if (result.Error == null)
            showUI();
        else
            showErrMsg();
    }

    private void getNameCallback(IGraphResult result) {
        if (result.Error == null) {
            IDictionary<string, object> res = result.ResultDictionary;
            myName = res["name"].ToString();
            loggedInUI.transform.FindChild("Name").GetComponent<Text>().text = myName;
        }
        else
            showErrMsg();
    }

    private void getPictureCB(IGraphResult result) {
        if (result.Error == null) {
            Texture2D image = result.Texture;
            var s = Sprite.Create(image,new Rect(0,0,100,100),new Vector2(0.5f,0.5f));
            var pic = loggedInUI.transform.FindChild("ProfilePicture");
            pic.GetComponent<Image>().sprite = s;
        }
        else
            showErrMsg();
    }

    private void scoresCB(IGraphResult result) {
        if (result.Error == null)
        {
            var scoresList = result.ResultDictionary["data"] as List<object>;
            var children = loggedInUI.transform.FindChild("ListContainer").
                           FindChild("FriendList").transform;

```

```

        foreach (Transform child in children)
            GameObject.Destroy(child.gameObject);
        foreach (var friendScore in scoresList) {
            var entry = (IDictionary<string, object>)friendScore;
            var user = (IDictionary<string, object>)entry["user"];
            var score = entry["score"].ToString();
            createFriend(user["name"].ToString(), user["id"].ToString(), score);
            if (user["name"].ToString() == myName)
                myScore = score.ToString();
        }
    }
    else
        showErrMsg();
}
}

```

Skripta referencira Canvas objekte logged in i logged out stanja, kao i objekat koji predstavlja element skrolabilne liste rezultata. Vodi se i evidencija o tome da li je došlo do neke greške, npr. usled izgubljene Internet veze, a čuva se i tekst greške.

Awake() funkcija inicijalizuje Facebook podatke ukoliko to već nije urađeno. Funkcija **logIn()** pokušava da uloguje korisnika, a navedene permisije su potrebne za pravilan rad funkcionalnosti o kojima smo govorili. Povratna informacija Facebook-a se obrađuje u **logInCallback** funkciji. Ukoliko je došlo do greške prikazuje se poruka o tome, u suprotnom prikazuje se *FBLoggedInCanvas*. Funkcija **logOut()** radi suprotno. Ona pokušava da razloguje trenutnog korisnika ukoliko je on logovan, deaktivira *FBLoggedInCanvas* i prikazuje *FBNotLoggedInCanvas*. Funkcija **showUI()** je najbitnija funkcija. Ukoliko je korisnik logovan, ona traži od Facebook-a njegovo ime, profilnu sliku, i 30 najboljih rezultata prijatelja koji igraju ovu igru. Nakon toga prikazuje odgovarajući Canvas. Funkcija **getNameCallback()** obrađuje rezultat zahteva za imenom, i ukoliko nije došlo do greške, prikazuje ime korisnika u predviđeni Text objekat. Funkcija **getPictureCB()** postavlja profilnu sliku korisnika koji se loguje u odgovarajući Image objekat, podešavanjem sprite atributa. Funkcija **scoresCB()** uništava postojeće podatke iz liste najboljih rezultata, i učitava najsvežije, pozivima funkcije **createFriend()**. Ova funkcija pojedinačno kreira elemente liste u koje upisuje ime i broj ostvarenih poena, a pored toga, zahteva i profilnu sliku prijatelja, nakon čega je postavlja u odgovarajući Image objekat. Funkcija **hideUI()** potpuno sakriva Facebook Canvas-e. Funkcije **showErrMsg()** i **noErrMsg()** prikazuju, odnosno sakrivaju, poruku o grešci. Funkcija **setScore(int)** upoređuje trenutni rezultat sa prethodnim zapamćenim, i ukoliko je ostvaren veći broj poena, poziva funkciju **setNewScore(int)** koja upisuje novi rezultat na Facebook. Primetimo da je u ovoj funkciji prvi put upotrebљena `HttpMethod.POST` metoda, umesto dosadašnje `HttpMethod.GET`. Sada se od Facebook-a ne traže podaci, već se šalju njemu na čuvanje. Funkcija **share()** postavlja Facebook objavu na zid logovanog korisnika sa zadatim nazivom, linkom, slikom, i porukom, naravno, nakon što korisnik za to da odobrenje. Funkcija **invite()** šalje zahtev odabranim prijateljima za instalaciju aplikacije.

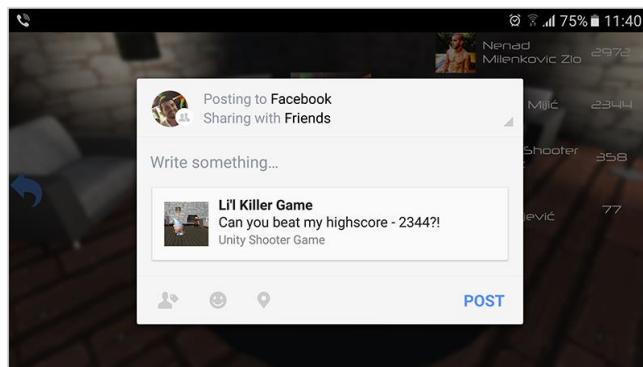
Na *MainMenuCanvas* objektu dodajemo novo dugme za prikaz odgovarajućeg Facebook Canvas-a, i ažuriramo skriptu *GameOverManager*. U **Update()** funkciji ove skripte dodajemo `FBscript.setScore(score);`; čime se ostvaren rezultat, ukoliko je on veći od postojećeg, postavlja i čuva na Facebook-u. Kao akcije prethodno kreiranog dugmeta postavljamo funkcije `FB.showUI()` i `GameOverManager.deactivateMainMenuCanvas()`, a

kasnije ovde dodajemo još jednu - `ButtonsManager.setFbUIOnState()` funkciju, kada je budemo napisali. Dakle, klikom na ovo dugme se pokreće i prikazuje odgovarajući Facebook Canvas, deaktivira `MainMenuCanvas`, a nakon implementacije poslednje funkcije ažuriraće se i trenutno stanje `ButtonsManager` skripte.

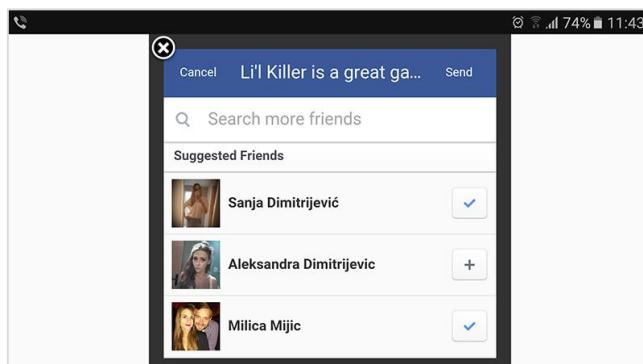
`LoginButton` dugme `FBNotLoggedInCanvas`-a izvršava `FBScript.login()`. Dugmićima `FBLoggedInCanvas`-a za Share i Invite dodajemo akcije poziva istoimenih funkcija - `share()` i `invite()`, dok `Logout` dugmetu dodajemo `logout()` funkciju `FBScript`-e. Dugme za povratak na prethodni meni oba Facebook Cavnas-a treba da izvršava funkciju `hideUI()` skripte `FBScript` i funkciju `activateMainMenuCanvas()` `GameOverManager` skripte.

U toku testiranja iz Unity okruženja i Game View prozora, logovanje na Facebook vrši se uz pomoć generisanog tokena koji se preuzima sa Facebook Developer stranice, umesto regularnog popunjavanja username / password polja. Odavde bez problema rade sve funkcionalnosti osim Share i Invite, koje Unity test okruženje ne podržava. Ove funkcije rade samo pri izvršavanju na ciljanoj platformi.

Za funkcionisanje svih implementiranih funkcionalnosti na Android telefonu, dok igra ne bude javno objavljena, korisniku je potrebna Tester rola i instalacioni .apk fajl igre. Tester rolu dodeljuje administrator ili kreator Facebook aplikacije, dodavanjem korisničkog imena ili identifikacionog broja profila korisničkog Facebook naloga. Logovanje na mobilnim uređajima ide direktno preko Facebook aplikacije, ukoliko je ona instalirana, i tu se pri prvom pokretanju potvrđuje saglasnost davanja navedenih permisija. Svako naredno logovanje preko istog naloga ide automatski, i ne zahteva ponovno davanje saglasnosti.



Slika 6.5: Facebook opcija - Share Score



Slika 6.6: Facebook opcija - Invite Friends

6.3 Optimizacija i korisničko iskustvo

U ovom trenutku možemo reći da je igra završena i da smo postigli zacrtane ciljeve. Igra je spremna i funkcioniše kako na Windows, tako i na Android platformi. Međutim, uvek treba obraćati pažnju na sitnice i obogatiti korisničko iskustvo, tako da aplikacija bude što više User-Friendly. Pojedine funkcionalnosti su zajedničke skoro svim aplikacijama određenog operativnog sistema, pa i one koje sami kreiramo ne treba da budu izuzetak. Ono što je zajedničko mobilnim telefonima koje pokreće Android operativni sistem su Back i Home tasteri. Pritiskom na taster Back, korisnik očekuje povratak na prethodno stanje ili stranicu. Duplim pritiskom na Back, korisnik očekuje izlazak iz aplikacije. Pritiskom Home tastera, igra treba da uđe u pauzirano stanje, tako da pri povratku korisnik može da se pripremi pre nego što nastavi partiju. U verziji Unity-a koju koristimo, ne postoje standardni event-i koji se pokreću pritiskom na pomenute tastere, tako da sve ovo manuelno implementiramo.

Nastavljamo sa implementacijom *ButtonsManager* skripte. Back taster na telefonu može imati različite funkcionalnosti u zavisnosti od toga u kom je igra stanju, ili koji UI meni je trenutno pokrenut. Podatak o ovome čuva promenljiva *currentUIState*. Funkcija *Update()* očekuje pritisak Back tastera, koji je mapiran kao Esc taster na tastaturi, i na osnovu vrednosti *currentUIState* promenljive se njegovim pritiskom igra pauzira, gasi, ili se pokreće odgovarajući UI meni.

Ažurirana ButtonsManager skripta:

```
public class ButtonsManager : MonoBehaviour {  
    ...  
    public enum uiState { NoBackButton, FbUIOn, HighScoresUIOn, NewScoreUIOn };  
  
    public static uiState currentUIState;  
  
    public FBscript fbScript;  
    public GameOverManager gomScript;  
    public HighScoreManager hscScript;  
  
    int numEscPressed = 0;  
  
    void Start() {  
        ...  
        currentUIState = uiState.NoBackButton;  
    }  
  
    public void setFbUIOnState() {  
        currentUIState = uiState.FbUIOn;  
    }  
    public void setHighScoresUIOnState() {  
        currentUIState = uiState.HighScoresUIOn;  
    }  
    public void setNewScoreUIOnState() {  
        currentUIState = uiState.NewScoreUIOn;  
    }  
  
    void Update() {  
        if (Input.GetKeyUp(KeyCode.Escape)) {  
            switch (currentUIState) {
```

```

        case uiState.FbUIOn:
            fbScript.hideUI();
            gomScript.activateMainMenuCanvas();
            numEscPressed = 0;
            break;
        case uiState.HighScoresUIOn:
            gomScript.hideScores();
            numEscPressed = 0;
            break;
        case uiState.NewScoreUIOn:
            hscScript.hideEnterNewHighscore();
            gomScript.activateMainMenuCanvas();
            numEscPressed = 0;
            break;
        default:
            if (GameOverManager.gamepaused || gomScript.gameover) {
                if (numEscPressed >= 1)
                    Application.Quit();
                else
                    numEscPressed++;
            }
            else
                gomScript.setPaused();
            break;
        }
        currentUIState = uiState.NoBackButton;
    }
}

void OnApplicationPause(bool pauseStatus) {
    if (!DontDestroyScript.showStartScreenOnGameStart &&
        !GameOverManager.gamepaused && !gomScript.gameover)
        gomScript.setPaused();
}
}

```

Definisan je enumerator koji predstavlja moguća stanja. Promenljiva numEscPressed vodi evidenciju o broju uzastopnih klikova tastera Back, pa ukoliko je ovaj broj veći ili jednak od 2, igra se gasi. Tu su i reference *FBScript*, *GameOverManager* i *HighScoreManager* skripti i javne metode za postavljanje trenutnog stanja koje koristimo kao dodatne akcije dugmića UI menija. *Update()* je glavna funkcija koja usled pritiska Back tastera ispituje trenutno stanje, i po potrebi gasi prikazani Canvas, i prikazuje odgovarajući.

Funkcija *OnApplicationPause(bool)* je u stvari događaj koji se pokreće kada aplikacija sistemski uđe u pauzirano stanje, a to se dešava u slučaju kada npr. izgubi fokus. U slučaju da igra nije završena, pauzirana, niti tek pokrenuta, pritiskom na Home dugme, poziva se metoda *GameOverManager* skripte za prelazak u stanje pauze.

Funkcije *setFbUIOnState()* i *setHighScoresUIOnState()* treba da budu pozvane pri pokretanju Facebook i Highscore Canvas-a, pa ih kao dodatne akcije On Click () događaja dodelujemo odgovarajućim dugmićima *MainMenuCanvas* objekta. Takođe, ažuriramo *GameOverManager* skriptu, i unutar njene *showEnterNewHighscore()* metode, dodajemo funkciju *setNewScoreUIOnState()*.

Kada je reč o User-Friendly funkcionalnostima, vrlo je važno da korisnik uvek bude u toku šta se trenutno dešava u igri. Konkretno, ukoliko se učitava scena, ili nivo, koji sadrži puno

detalja, pa je za to potrebno više vremena, vrlo je poželjno na neki način obavestiti korisnika da se nešto u pozadini dešava. Ovo se obično prikazuje kroz neki Progress Bar, ili zanimljivu animaciju. Kako se naša scena prilično brzo učitava, za ovako nešto ipak nema potrebe.

U okviru PlayerSettings podešavanja možemo podesiti ikonicu, naziv igre, podržanu verziju Android operativnog sistema, standardnu orientaciju (gde biramo Auto Rotation - Landscape Right & Left), Loading Indicator, kao i Splash Screen sliku ukoliko imamo Pro verziju Unity-a. Sve su ovo sitnice koje korisnik primeti, i dobra je praksa podesiti ih kako valja.

Na kraju, osvrnimo se na performanse naše igre. Na Windows platformi i laptopu koji ima dosta superiorniji hardver u odnosu na pametne telefone, igra ima odličan frame rate. Na Android telefonima visoke klase igra i dalje ima odlične performanse, i za ljudsko oko razlike su neprimetne u poređenju sa izvršenjem na računaru. Na telefonima srednje klase frame rate već osetno pada, međutim to i dalje ne utiče previše na korisničko iskustvo, dok kod starijih telefona i telefona niže klase, gotovo je nemoguće normalno igrati. Napomenimo samo da je igra testirana na više od deset različitih modela Android telefona i tablet uređaja svih klasa. Problem koji se javlja na telefonima slabijeg hardvera donekle može da se reši, ali ipak ne u potpunosti. Nakon prelaska na Android platformu, objekte koji čine okruženje, neprijatelje i glavnog lika, kao i njihove materijale, shader-e i teksture nismo menjali. Teksturama možemo isključiti posebne efekte, smanjiti veličinu, i odabratи odgovarajući tip kompresije. Materijalima treba postaviti mobilne verzije Shader-a, na šta nas Unity i sam upozorava. Objektima scene možemo uključiti ili isključiti funkcionalnost primanja i bacanja senki, što se takođe odražava na perfomanse. Ova podešavanja se nalaze u okviru Mesh Renderer komponenti objekata, kao opcije Cast Shadows i Receive Shadows. Dalja podešavanja vezana za izgled i performanse pronalazimo u Lighting / Scene Tab-u gde podešavamo tip, kompresiju i intenzitet refleksije i druge efekte koji se odnose na osvetljenje scene. Takođe, u Player Settings Inspector-u pronalazimo razna podešavanja vezana za rezoluciju, kompresiju Vertex-a, Prebake funkcionalnosti i korišćenje komponenti hardvera. Na kraju, u okviru Edit / Project Settings / Quality Inspector-a možemo podešavati nivo kvaliteta grafike, renderovanja, kvaliteta tekstura, podešavanja jačine, rezolucije i daljine senki, Anti Aliasing i druge efekte, a sva ova podešavanja direktno utiču na perfomance aplikacije.

Iz Build Settings prozora, biramo opciju Build. Instalacioni .apk fajl prebacujemo na mobilni uređaj i pokrećemo instalaciju. Nakon završene instalacije, startujemo igru.

7 Zaključak

U ovom radu su obrađeni osnovni koncepti Unity tehnologije. Dat je opis interfejsa i razvojnog okruženja kroz upoznavanje sa osnovnim elementima, komponentama i alatima. Predstavljen je GameObject, kao osnovna jedinica građe Unity aplikacija, i njemu odgovarajuća Game Objects - Components - Variables struktura, na kojoj se zasniva čitav razvoj složenih objekata i njihovih osobina. Paralelno sa uvođenjem osnovnih koncepata, u radu je dat opis razvoja složene 3D igre. Proces razvoja započet je idejama, i preko opisa funkcionalnosti, uvođenja novih pojmove, upoznavanja načina rada i dostupnih alata, dolazi se do implementacije rešenja i konačne realizacije. Izložene su mogućnosti multiplatformske

distribucije, i demonstrirana je migracija projekta sa Windows platforme na Android. Predstavljena je konstrukcija mašine stanja, i naglašena njena uloga u kreiranju animiranog sadržaja. Kao jedan od načina povezivanja korisnika različitim platformi, dat je opis integracije sa društvenim mrežama na primeru Facebook-a. Na kraju, date su smernice za poboljšanje korisničkog iskustva, optimizaciju rada i postizanje visokih performansi.

Logični nastavak razvoja bio bi proširenje funkcionalnosti, rad na novim elementima igre, kreiranje novih nivoa, unapređenje grafike i vizuelnih efekata. Sledi prebacivanje na Web platformu, igranje iz pretraživača i pristup direktno sa Facebook-a. Treba omogućiti dodatne načine za međusobno nadmetanje između prijatelja, razmisliti o marketingu, ali i o načinu zarade - preko reklama ili mikrotransakcija u igri. Da bi sve ovo funkcionalno, igra mora biti javno objavljena i dostupna kao freemium aplikacija u Google Playstore prodavnici.

8 Literatura

- [1] Sue Blackman, *Beginning 3D Game Development with Unity 4: All-in-one, multi-platform game development*, Apress 2013.
- [2] Ryan Henson Creighton, *Unity 4.x Game Development by Example Beginner's Guide*, Packt Publishing 2013.
- [3] Will Goldstone, *Unity 3.x Game Development Essentials 2nd Edition*, Packt Publishing 2011.
- [4] Alan Thorn, *Learn Unity for 2D Game Development*, Apress 2013.
- [5] Jate Wittayabundit, *Unity 3 Game Development HOTSHOT*, Packt Publishing 2011.
- [6] <http://docs.unity3d.com/Manual/index.html>
- [7] <https://developers.facebook.com/docs/unity>



**ПРИРОДНО - МАТЕМАТИЧКИ ФАКУЛТЕТ
НИШ**

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, РБР:	
Идентификациони број, ИБР:	
Тип документације, ТД:	монографска
Тип записа, ТЗ:	текстуални / графички
Врста рада, ВР:	мастер рад
Аутор, АУ:	Миљан Мијић
Ментор, МН:	Марко Петковић
Наслов рада, НР:	Unity : Основни концепти и развој 3D игре
Језик публикације, ЈП:	српски
Језик извода, ЈИ:	енглески
Земља публиковања, ЗП:	Р. Србија
Уже географско подручје, УГП:	Р. Србија
Година, ГО:	2016.
Издавач, ИЗ:	авторски репринт
Место и адреса, МА:	Ниш, Вишеградска 33.
Физички опис рада, ФО: (поглавља/страна/цитата/табела/слика/графика/прилога)	81 стр. ; граф. прикази
Научна област, НО:	рачунарске науке
Научна дисциплина, НД:	развој игара
Предметна одредница/Кључне речи, ПО:	unity, развој 3D игара
УДК	005.311.7 004.42 519.83
Чува се, ЧУ:	библиотека
Важна напомена, ВН:	
Извод, ИЗ:	У раду је представљена Unity платформа за развој интерактивних апликација и видео игара. Обрађени су основни концепти, и описано развојно окружење са доступним алатима. Дат је процес развоја 3D игре који обухвата креирање сцене и окружења, рад са асетима, креирање карактера и њиховог понашања, креирање UI менија, анимација и машина стања. Имплементиране су функције друштвене мреже Facebook, и игра је објављена за Windows и Android платформу.
Датум прихватања теме, ДП:	
Датум одбране, ДО:	
Чланови комисије, КО:	Председник: Члан: Члан, ментор:



**ПРИРОДНО - МАТЕМАТИЧКИ ФАКУЛТЕТ
НИШ**

KEY WORDS DOCUMENTATION

Accession number, ANO:	
Identification number, INO:	
Document type, DT:	monograph
Type of record, TR:	textual / graphic
Contents code, CC:	university degree thesis (master thesis)
Author, AU:	Miljan Mijić
Mentor, MN:	Marko Petković
Title, TI:	Unity : Basic concepts and 3D game development
Language of text, LT:	Serbian
Language of abstract, LA:	English
Country of publication, CP:	Republic of Serbia
Locality of publication, LP:	Serbia
Publication year, PY:	2016
Publisher, PB:	author's reprint
Publication place, PP:	Niš, Višegradska 33.
Physical description, PD: (chapters/pages/ref./tables/pictures/graphs/applications)	81 p. ; graphic representations
Scientific field, SF:	Computer science
Scientific discipline, SD:	Game development
Subject/Key words, S/KW:	Unity, 3D game development
UC	005.311.7 004.42 519.83
Holding data, HD:	library
Note, N:	
Abstract, AB:	In this thesis, Unity is described as a platform for development of video games and other interactive applications. Thesis discusses basic concepts, and describes IDE with all the available tools. Also, the whole process of 3D game development is presented - creating new scenes and environment, working with assets, building characters and controlling its behavior, creating UI menus, animations and state machines. Facebook API for Unity is implemented, and the game is published for Windows and Android platform.
Accepted by the Scientific Board on, ASB:	
Defended on, DE:	
Defended Board, DB:	President: Member: Member, Mentor: