

**SVEUČILIŠTE U SPLITU**  
**SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE**  
**Odsjek za informacijsku tehnologiju**

## **Dionyzus Unity3D igra**

**Split, 10. lipnja 2019.**

# Sadržaj

<b>Sažetak</b>	<b>1</b>
<b>1 Uvod</b>	<b>3</b>
<b>2 Pogonski alat Unity</b>	<b>5</b>
2.1 Upoznavanje sa sučeljem . . . . .	6
2.2 Rad s dodacima . . . . .	9
2.3 Glavni prozori . . . . .	11
2.4 Stvaranje igre . . . . .	22
2.5 Grafika i optimizacija . . . . .	33
2.6 Fizika, animacije, navigacija i platno . . . . .	38
2.7 Skripte . . . . .	48
<b>3 Dionyzus</b>	<b>56</b>
3.1 Osnovne skripte za upravljanje s likom . . . . .	56
3.2 Energija i sistem oružja . . . . .	65
3.3 Funkcionalnosti neprijatelja . . . . .	72
3.4 Implementacija ostalih funkcionalnosti . . . . .	79
<b>4 Zaključak</b>	<b>85</b>
<b>Literatura</b>	<b>87</b>
<b>Dodaci</b>	<b>87</b>

## Sažetak

Cilj ovog završnog rada bio je predstaviti stranu programiranja koja omogućava zabavu korisnika, odnosno igrača kao i samog programera. Prikazuje spajanje različitih znanja unutar jedinstvenog projekta, potiče na razvijanje kreativnosti koja je potrebna kako bi se napravio što zanimljiviji dizajn, animacije, atmosfera, opisali likov i priča, sve to je potrebno kako bi se postigla zainteresiranost, ali i povezanost igrača s igrom. Tijekom izrade igre programer se upoznaje s ljudima iz različitih grana industrije u svrhu prikupljanja potrebnih informacija, bilo to vezano za potrebe izrade igre ili jednostavno povratne informacije osoba koje testiraju igru, istražuju eventualne probleme, daju određene savjete i slično. Konkretno ovim projektom je prikazano kako se s besplatnim resursima može izraditi sustav koji u konični pruža sate i sate užitka. U radu je opisan način izrade igrice korištenjem okruženja za izradu igara Unity3D (engl. *Unity3D game engine*) te programskog jezika C# koji se koristi u pogonskom alatu Unity3D. Ime igre je Dionyzus, žanr je otvoreni svijet, igra se iz trećeg lica te se radi o 3D igrici.

Unutar igre je prikazan život Leona te njegov put prema otkrivanju tko stoji iza namještajke u kojoj njegov najbolji prijatelj umire. Osim odradivanja zadatka i izazova koji su predstavljeni igraču, moguće je slobodno istraživati grad te obavljati različite poslove neovisno o glavnoj priči.

Ključne riječi: Termin pogonski alat (engl. *Game engine*) u radu predstavlja kompletan Unity, dakle i editor i C# biblioteku. Razvojni okvir Unity (engl. *Unity framework*) predstavlja C# biblioteku. 3D igra (Igrica u 3 dimenzije), C# (Programski jezik C sharp), Dionyzus (Naziv igrice).

## Summary

### Dionyzus Unity3D Game

This graduate work is a game, made in Unity3D game engine. It is 3D game, open world, sandbox game based of GTA (Grand theft auto) series. This game was made as a demo which shows what can be done in Unity. Purpose of this graduation work is to enjoy and have fun while testing, adding custom story line, as well as programming and adding some original features in world of gaming. It combines different types of knowledge, it encourages

developing of different set of skills such as visuals and animation design, music composing, creating interesting atmosphere, describing characters and story line, all that is necessary to make future player interested in the game as well as connected to it. Throughout the game develop, programmer meets people from different industries with a purpose to gather all kinds of information, it could be anything connected to the game develop, getting feedback when it comes to bugs as well as listening to the different advises etc. With this project it was shown how with no investment it's possible to create a game which offers hours and hours of fun time. Game shows life of Leon and his path to find out who is behind the set-up in which his best friend dies. Beside doing tasks and surviving through the challenges which lead to finishing story, player can freely explore the city and do side jobs which aren't related to main story.

Keywords: Game engine Unity which refers to complete Unity, editor and C# library, framework Unity which refers to C# library, C# programming language, Dionyzus - name of the game, 3D game - game in 3 dimensions.

# 1. Uvod

U radu je obrađena izrada 3D igre u Unity3D. Motivacija za rad je uvod u granu informacijskih tehnologija (IT) koja se bavi izradom igara pomoću modernog pogonskog alat za izradu igara (engl. *Game engine*). Pristup izradi igre je bio da se omogući što lakše ponovno korištenje programskih rješenja unutar igre, te da se stvori sustav koji se jednostavno može proširiti.

Obrađena su ključna poglavlja za izradu velikog svijeta te sustava potrebnog za funkcioniranje igre:

- teren, odnosno područje gdje će se odvijati radnja,
- sustav za upravljanje igračem,
- osnovno ponašanje neprijatelja,
- sustav napretka kroz igru.
- odrađivanje misija,
- zarada te potrošnja novca,
- prikupljane predmeta,
- izrada video isječka,
- zvučni efekti te muzika,
- upravljanje vozilima.

Svi modeli, ljudi, okolina, oružja itd. su preuzeti iz Unity centra za kupovanje podataka za igru. Koriste se isključivo u svrhu uljepšavanja same igre, a ključni izvori će biti navedeni u radu.

U prvom dijelu rada govorit će se ukratko o Unity okruženju te programskom jeziku C# u kojem su napisane sve skripte potrebne za funkcioniranje igre. Nakon toga opisane su najvažnije prednosti rada u Unity, u odnosu na izradu igre od početka, implementaciju fizike, grafike i slično. Isto tako prikazani su i opisani neki ključni segmenti Unityja koji su bili korišteni u ovom radu.

U sljedećem poglavlju opisan je osnovni pristup prilikom izrade skripti, te glavne komponente koje su potrebne da bi iste funkcionirale u igri. U trećem i četvrtom poglavlju su

prikazane scene te opisana radnja igre. Isto tako su navedena rješenja koja su izradile druge osoba, a koja su korištena u ovom radu. Na samom kraju naveden je zaključak te prijedlozi kako proširiti sustav.

Igra započinje scenom u vlaku gdje glavni lik Leon razgovara sa svojim prijateljem Vladislavom o njihovoј trenutnoј životnoј situaciji, koja i nije baš najbolja. Oba lika su u teškoј financijskoј situaciji te im je potrebna promjena kako bi usmjerili svoje živote u pravom smjeru. Vladislav je snalažljiv lik koji na razne načine dolazi do zarade, to su uglavnom sitne prevare, iznuđivanja i slično, dok je Leon vješt lik koji je prošao specijalnu vojnu obuku, ali je zbog incidenta razriješen dužnosti. Leonu se ne sviđa to što Vladislav radi, ali nerijetko mu pomaže izbaviti ga iz nevolje, naravno zbog njihovog velikog prijateljstva. Vladislav ovaj put iznosi opasan plan pljačke banke, gdje bi Leon trebao biti samo pomoćnik, odnosno vozač automobila za bijeg.

Leon, iako svjestan opasnosti, unatoč negodovanju i ovaj put pristaje pomoći, što će na kraju skoro platiti vlastitim životom. Čudom Leon preživljava, saznaje da je riječ o podvali, uvjerenja je da je Vladislav ubijen te se odlučuje na osvetu gdje će mu sva obuka koju je prošao pomoći na putu ostvarivanja iste. Tijekom igre Leon uz suradnju s drugim prijateljima doznaje različite informacije te otkriva zamršeni mafijaški lanac kojem odluči stati na kraj.

Kako se radi o otvorenom svijetu, igrač ima mogućnost istraživanja svijeta te slobodnu kretnju preko cijele mape. Pri tom sakuplja različite predmete, kao što su oružja, municija, energetski paketi te različite stvari koje su mu potrebne kako bi odradio misiju.

## 2. Pogonski alat Unity

Unity je softver koji pruža stvaraocima igara sve potrebne dodatke kako bi se omogućila što brža i efektivnija izrada igara. Pogonski alat za igre (engl. *Game engine*) omogućava upotrebu 2D i 3D modela te slika iz drugih programa kao što su Maya, Blender ili Photoshop. Isto tako omogućava jednostavno sastavljanje svih važnih komponenti koje su potrebne za jednu igru, kao što su svjetlost, zvuk, specijalni efekti, fizika, animacije itd.

Rad u Unity možemo podijeliti na dva dijela. Jedan je rad unutar razvojnog okruženja Unity koji obuhvaća upotrebu svih mogućnosti pogonskog alata. Drugi dio je programiranje, odnosno razvijanje softvera koristeći Microsoft Visual Studio ako se radi unutar operacijskog sustava Windows ili MonoDevelop unutar operacijskog sustava Linux. U nastavku će biti opisan rad u jednom i u drugom dijelu.

Kada se govori o grafici, to podrazumijeva kvalitetnu arhitekturu sa sveukupnim mogućnostima vizualizacije visokih performansi kao i brzi pristup grafičkom sučelju, kako bi se omogućila što vjernija vizualna kvaliteta igre.

Zvuk se može stvarati i uređivati izvan Unityja, odnosno koristeći aplikacije treće strane. Nakon toga se gotovi zvuk integrira u igru, bilo kao dio atmosfere, efekta određenih predmeta ili kao dio nekog događaja igre.

Jedna od najvažnijih komponenti pogonskog alata je fizika. Fizika je već integrirana u alat te uz samo nekoliko linija kôda moguće je postići vjerodostojne efekte, odnosno ponašanje različitih objekata unutar igre. To omogućava dizajneru igre da manje vremena potroši na mukotrpno stvaranje fizike, a više na integriranje vlastitih ideja kako napraviti igru što zanimljivijom.

Isto tako omogućeno je jednostavno stvaranje grafičkog sučelja za samog igrača unutar igre, što uključuje različite izbornike, statistiku, kao i prikaz najvažnijih podataka tijekom igranja, kao što su rezultat, zdravlje, navigacijska karta itd.

Unity podržava programiranje u C#, isto tako dolazi s određenim standardnim skriptama koje omogućuju korisniku brzi početak rada, te stvaranja prve igre bez previše kodiranja. Od velike pomoći su gotovi projekti koji se nalaze na stranicama Unityja koji omogućuju jednostavno upoznavanje s procesom rada te ključnim segmentima izrada igara.

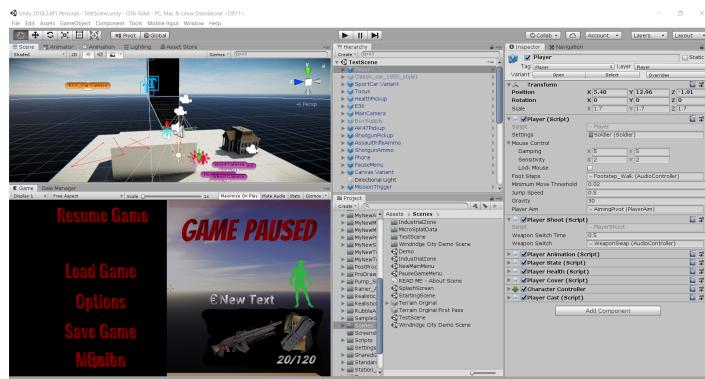
Unity je više-platformski što znači da se može koristiti, te se isto tako konačni

produkt može prevesti i pokrenuti na različitim operacijskim sustavima. Omogućava izradu više vrsta projekata, to uključuje:

- kompletna 3D igra,
- ortografska 3D igra,
- kompletan 2D,
- 2D način igranja s 3D grafikom,
- 2D način igranja i grafika, s kamerama različitih perspektiva.

## 2.1. Upoznavanje sa sučeljem

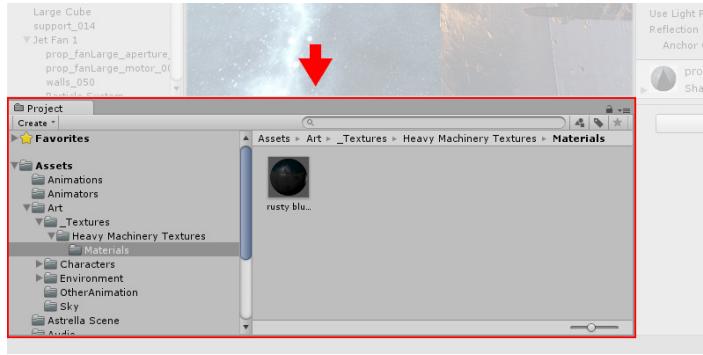
Prije početka rada dobro je se upoznati s razvojnim sučeljem. Glavni zaslon se može prilagoditi tako da se poslože glavni prozore za rad prema vlastitim sklonostima, a to uključuje kompletну promjenu rasporeda, dodavanje i uklanjanje različitih prozora i tabova. Glavni prozor je prikazan na slici 1.



Slika 1: Radno sučelje Unity

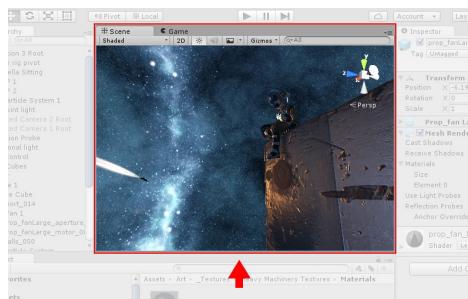
Glavni prozor uključuje alatnu traku, prikaz scene, nadgledni prozor (engl. *Inspector window*), projektni prozor i hijerarhijski prikaz objekata.

Projektni prozor (slika 2) prikazuje kompletnu biblioteku dodataka koji su dostupni u projektu. Prilikom preuzimanja dodataka sa stranica trgovine s dodacima (engl. *Asset store-a*), napravi se direktorij koji sadrži taj dodatak unutar ovog prozora.



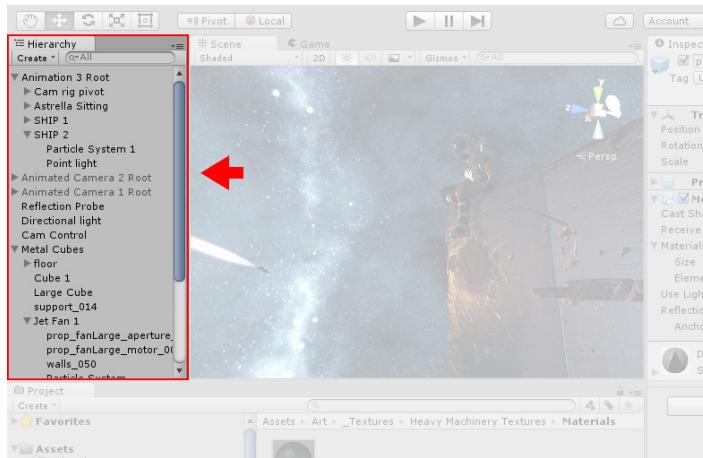
**Slika 2:** Projektni prozor

Prikaz scene (slika 3) je uz hijerarhijski prikaz, najkorišteniji prozor unutar samog sučelja. Omogućava pomicanje objekata, rotiranje, navigaciju po sceni itd. Moguće su 3D ili 2D perspektive ovisno o tome o kakvom se projektu radi i radi li se grafičko sučelje za igru, odnosno HUD-u (engl. *Head-up display*). Isto tako su u prikazu scene vidljive promjene osvijetljena ovisno o tome jesu li uključene postavke za to.



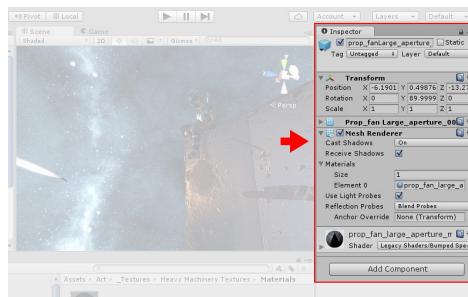
**Slika 3:** Prikaz scene

Hijerarhijski prikaz (slika 4) je hijerarhijski tekstualni prikaz svakog objekta u sceni. Svaki objekt unutar scene se nalazi i u ovom prozoru, a hijerarhija prikazuje kako su objekti međusobno povezani. Unutar ovog prozora moguće je preimenovati različite objekte, postavljati vidljivost, mijenjati raspored te naravno brisati i dodavati objekte unutar scene.



**Slika 4:** Hijerarhijski prikaz

Nadgledni prozor (slika 5) omogućuje prikaz i promjenu svojstava trenutno izabranog objekta. Kako različiti objekti imaju različita svojstva, sami prikaz i sadržaj ovog prozora će se razlikovati ovisno o tome. Ovaj prozor se najviše koristi za promjenu položaja objekta unutar scena ako su poznate koordinate ili je potrebno fino pomicanje, a isto tako služi za promjenu veličine objekta, kao i za dodavanje različitih komponenti kao što fizička svojstva, grafički prikaz, materijali te skripte.



**Slika 5:** Nadgledni prozor

Alatna traka (slika 6) omogućava pristup najvažnijim komponentama za rad. Na lijevoj strani se nalaze osnovni alati za upravljanje scenom i objektima unutar scene. U sredini su kontrole za pokretanje, stopiranje i pauziranje igre. Na desnoj strani se nalaze botuni koji omogućavaju pristup Unity računu i servisima unutar Unity oblaka (engl. *Unity Cloud Services*). Tu su još komande za promjenu prikaza sučelja te spremanje vlastitih postavki.



**Slika 6:** Alatna traka

## 2.2. Rad s dodacima

Dodaci (engl. *Assets*) su svi elementi koji se mogu koristiti unutar projekta. To mogu biti modeli, audio datoteke, slike ili bilo koje druge datoteke koja se može koristiti unutar Unityja. Osim toga, postoje dodaci koji se mogu kreirati unutar Unity, kao što su kontroler animacije (engl. *Animator controller*), mikser zvukova (engl. *Audio mixer*) i učitavač tekstura (engl. *Render texture*).

Unity podržava sve standardne formate slika kao što su BMP, TIF, JPG, PNG, kao i sve standardne formate audio datoteka.

Kada je riječ o modelima, uglavnom se radi o FBX formatu datoteka, a u slučaju da korisnik spremi datoteku u drugom formatu ona se uvozi u Unity kao FBX datoteka.

Standardni dodaci koji dolaze s Unityjem su uglavnom elementi, odnosno skripte koje se koriste u svim projektima kao što su postavke kamere, određeni likovi, više-platformski unos, efekti i okolina, sistem čestica i vozila.

Kao što postoje primitivni tipovi u programskim jezicima tako postoje određeni primitivni objekti unutar Unityja. To su:

- Kocka, koja osim što predstavlja kocku, kada se rastegne može predstavljati i zidove, okidače, kao i privremeni držač za neki objekt.
- Kugla, koja se, osim logične upotrebe kao što su planet, lopta i slično, može koristiti naprimjer kao prikaz radijusa nekog efekta.
- Kapsula, koja najčešće služi kao privremeni držač za neki objekt.
- Valjak.
- Ravnina, služi uglavnom kao ravna površina, pod ili zid.
- Četverokut, za prikaz slika i slično.

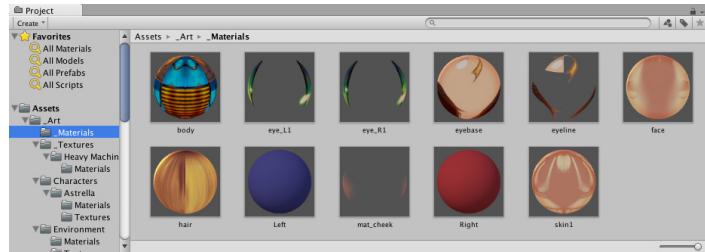
Kada je riječ o korištenju kompletno izrađenih dodataka od strane neke treće osobe, odnosno projekata koji se nalaze u trgovini za dodatke, na stranicama trgovine (engl. *Asset store*) nalazi se mnoštvo projekata, bilo različitih modela, potrebnih mehanizama u igri, skripti koje olakšavaju izradu različitih komponenti igre, kao i kompletne igre koje mogu biti korištene unutar vlastitog projekta. Neki projekti su besplatni, dok je neke, vjerojatno kvalitetnije, potrebno kupiti. Međutim, čak i upotrebotom besplatnih mogu se napraviti odlične

igre. Unutar Dionyzus igre korišteni su isključivo besplatni dodaci.

Dodatak se prvo skine na računalo te nakon toga uvozi u projekt. Mogu se uvesti samo potrebni dijelovi te u slučaju potrebe nadograditi. Isto tako je moguće izvoziti vlastite dodatke, tako što se napravi vlastiti paket za izvoz.

## 2.3. Glavni prozori

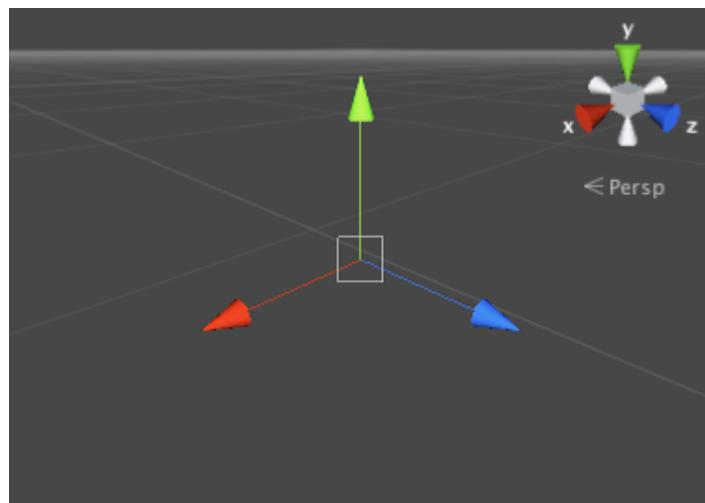
U ovom dijelu je nastavljeno upoznavanje s radnim okruženjem, te će biti detaljnije opisan svaki od prozora. Prvi je prozor projekta (slika 7).



Slika 7: Prozor projekta

S lijeve strane nalaze se direktoriji projekta prikazani u hijerarhijskoj listi. Klikom na neki od direktorija prikazuje se sadržaj tog direktorija na desnoj strani. Pojedini elementi su prikazani odgovarajućim ikonama tako da se jednostavno može pronaći potrebnu datoteku. Strukturu ovog prozora je lako prilagoditi vlastitim željama. Moguće je povećati i smanjiti ikone, promijeniti vrstu prikaza, spremiti određene datoteke u favorite. Sve datoteke mogu se pretraživati unosom filtera, odnosno imena datoteke. Osim standardnog prikaza datoteka koje odgovaraju filteru pretraživanja, moguće je dodatno filtrirati rezultate pretraživanjem po tipu ili labelama. Klikom na datoteku prikazuju se detalji u nadglednom prozoru.

U prozoru scene nalazi se set navigacijskih kontrola koje olakšavaju rad unutar scene. U gornjem desnom kutu nalazi se naprava (engl. *Gizmo*) vidljiv na slici 8.



Slika 8: Naprava (engl. *Gizmo*)

S njim se mijenja perspektiva, odnosno pogled unutar scene, te se na taj način mogu koristiti odgovarajuće postavke ovisno o tome što se trenutno radi u sceni. Vrste pogleda su:

- perspektivni model,
- ortogonalni model,
- prednji pogled,
- topografski pogled.

Alat za rad s objektima (slika 9) nalazi se u gornjem desnom kutu. Prva ikona predstavlja alat s kojim se kreće po sceni, takozvani *Hand tool*. Tu su i ostali botuni s kojima možemo rotirati, pomicati, te povećavati, odnosno smanjivati objekte.



**Slika 9:** (engl. *UI-ViewTool*)

Neki od prečaca koji mogu ubrzati rad unutar scene nalaze se na slici 10.

Action	3-button mouse	2-button mouse or track-pad	Mac with only one mouse button or track-pad
<b>Move</b>	Hold Alt+middle mouse button, then drag	Hold Alt+Control+left-click, then drag	Hold Alt+Command+left-click, then drag
<b>Orbit</b> (Not available in 2D mode)	Hold Alt+left-click, then drag	Hold Alt+left-click, then drag	Hold Alt+left-click, then drag
<b>Zoom</b>	Use the scroll wheel, or hold Alt+right-click, then drag	Hold Alt+right-click, then drag	Use the two-finger swipe method to scroll in and out, or hold Alt-Control+left-click, then drag

**Slika 10:** Prečaci za rad u sceni

Jedna od jako korisnih stvari je *Unit snapping* koji se koristi na način da se pritisne kontrolnu tipku (Control-CTRL) te koristeći se alatom za pomicanje ili transformiranje (kombinacija pomicanja, rotiranja i mijenjanja veličine) mijenjamo vrijednost za onu koja je postavljena u postavkama (engl. *Snap settings*).

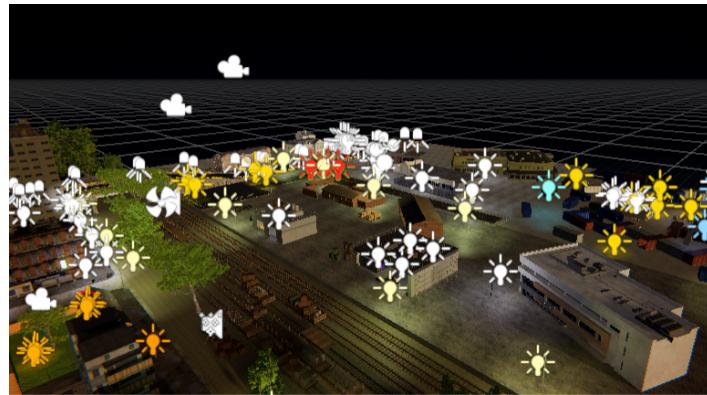
Tijekom rada korisnik može uključiti, odnosno isključiti određene efekte unutar scene. Ti efekti su prisutni jedino tijekom rada na projektu, te nemaju nikakvi utjecaj na završni proizvod, a to su:

- crtaće metode
  - *Shaded* - prikazuje površine s teksturama,
  - *Wireframe* - prikazuje strukture objekata u obliku linija različitih boja,
  - *Shaded Wireframe* - kombinacija prethodna dva navedena.
- razno
  - vidljivost sjena,
  - prikaz načina učitavanja tekstura,
  - prikaz transparentnih objekata,
  - prikaz idealne veličine tekstura ovisno o objektu (*Mipmaps*).
- prikaz vrsta materijala,
- globalno osvjetljenje.

Postoji još nekoliko različitih mogućnosti što se tiče postavki tijekom rada, kao što su mijenjanje aktivnog osvjetljena, uključivanje, isključivanje ikoni i mijenjanje veličina ovisno o tome što vam je trenutno potrebno za rad na projektu. Na slici 11 je prikazan izgled scene s aktivnim sličicama te rešetkama.

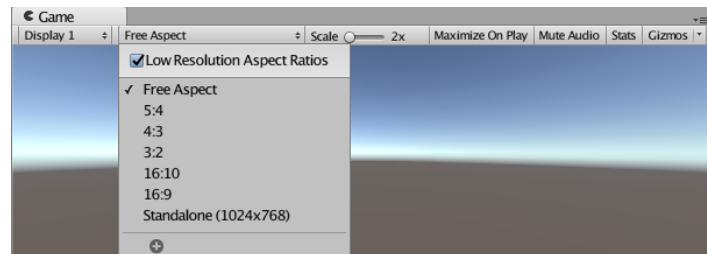
Botun	Opis
Prikaz	Odabir kamere ukoliko imate više kamere unutar igre.
Odnos(engl. <i>Aspect</i> )	Pregled izgleda na različitim veličinama ekrana.
Odabir niske rezolucije	Odaberite ovo ako želite vidjeti kako bi igra izgledala na starijim ekranima.
Povećalo	Mogućnost uvećanja, odnosno smanjenja prikaza.
Maksimiziranje ekrana	Ukoliko je ovo odabrano slika se automatski maksimizira tijekom pokretanja igre.
Zvuk	Gašenje zvuka po potrebi.
Statistika	Prikazuju se podaci o kvaliteti performansa tijekom igranja, podaci o grafici, memoriji, procesoru, jačini zvuka i slično.

**Tablica 1:** Opcije unutar prozora igre.



**Slika 11:** Rešetke i sličice (engl. *Grid and gizmo icons activated*)

Prozor igre je prozor u kojem se vidi konačni proizvod i koji predstavlja konačno stanje igre. Da bi se moglo prikazati što se događa u igri potrebne su jedna ili više kamere, koje trebaju biti kontrolirane tako da korisnik može vidjeti, odnosno igrati igru. Kontrolni botuni igre omogućavaju pokretanje, pauziranje te zaustavljanje igre. Na slici 12 prikazane su mogućnosti, odnosno postavke unutar prozora igre.



**Slika 12:** Mogućnosti prozora igre

U tablici 1 opisane su pojedine postavke.

Početni izgled prozora hijerarhije nalazi se na slici 13. Prilikom stvaranja novog projekta tu se nalazi naziv scene, glavna kamera te glavni izvor osvjetljena u sceni.



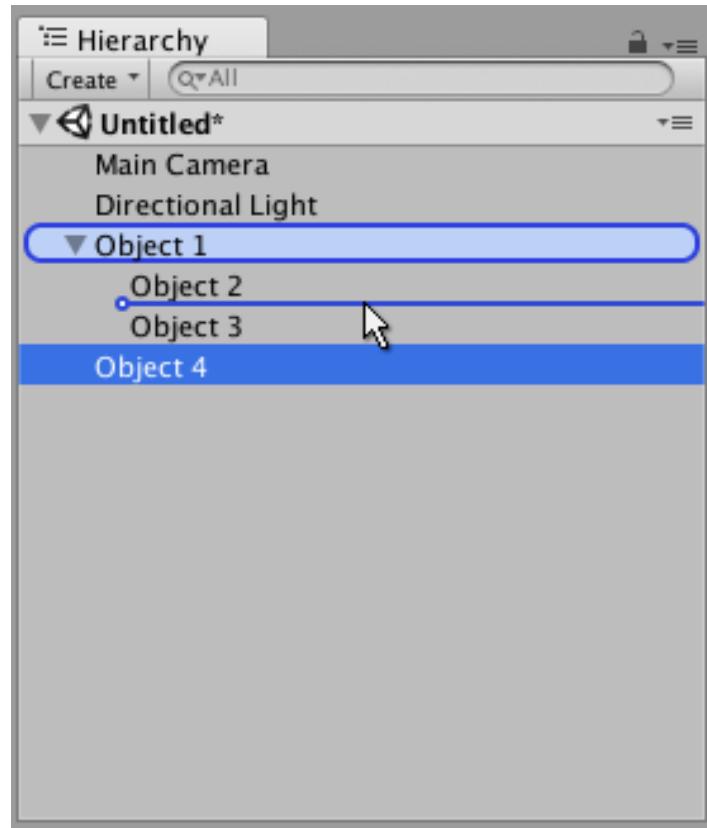
**Slika 13:** Početni izgled hijerarhijskog prozora

Ovaj prozor sadrži listu svih objekata koji se nalaze unutar trenutne scene, tzv. (engl.*GameObject*). Neki su obični 3D modeli dok su neki, tzv. (engl.*Prefabs*), odnosno objekti koji se mogu spremiti te tako ponovo koristiti unutar scene, ili neke druge scene. Kada se objekti uklone ili dodaju iz scene tako se brišu, odnosno dodaju unutar hijerarhijskog prikaza. Po potrebi korisnik može mijenjati raspored objekata te isto tako mijenjati odnos dijete ili roditelj. Na slici 14 prikazan taj odnos, objekt koji je prvi u hijerarhiji, odnosno koji sadrži neke druge objekte je roditelj, a objekti koji su sadržani unutar roditelja su djeca.



**Slika 14:** Prikaz odnosna objekata

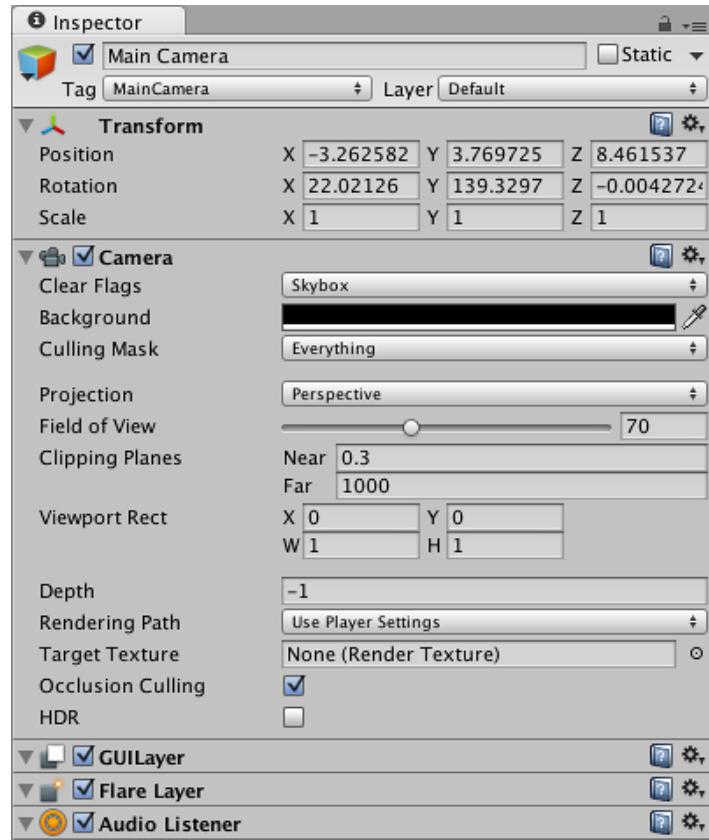
Da bi mijenjali odnos objekata jednostavno se povuče objekt na budući objekt roditelj. Više objekata na istoj razini su braća i sestre. Primjer je prikazan na slici 15, objekt 1 je roditelj, objekti 2 i 3 su djeca. Njihov raspored se isto tako može mijenjati.



Slika 15: Prikaz djece

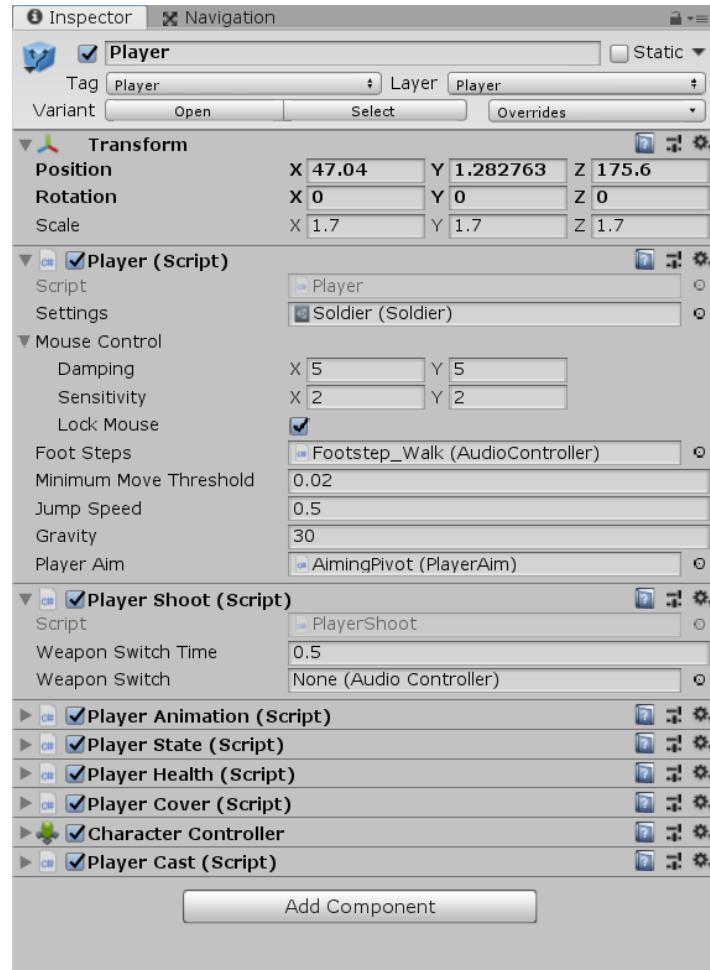
Raspored objekata se može promijeniti tako da budu abecedno poredani, sve što je potrebno je promijeniti postavke izgleda hijerarhije. Isto tako je omogućeno uređivanje, odnosno prikaz više od jedne scene unutar iste hijerarhije.

Nadgledni prozor prikazuje detaljne informacije trenutno odabranog objekta, uključujući sve komponente koje su dodane na taj objekt, njihova svojstva te omogućava promjenu njihove funkcionalnosti u sceni. Na slici 16 prikazan je nadgledni prozor s primjerom postavki te komponenti nekog objekta.



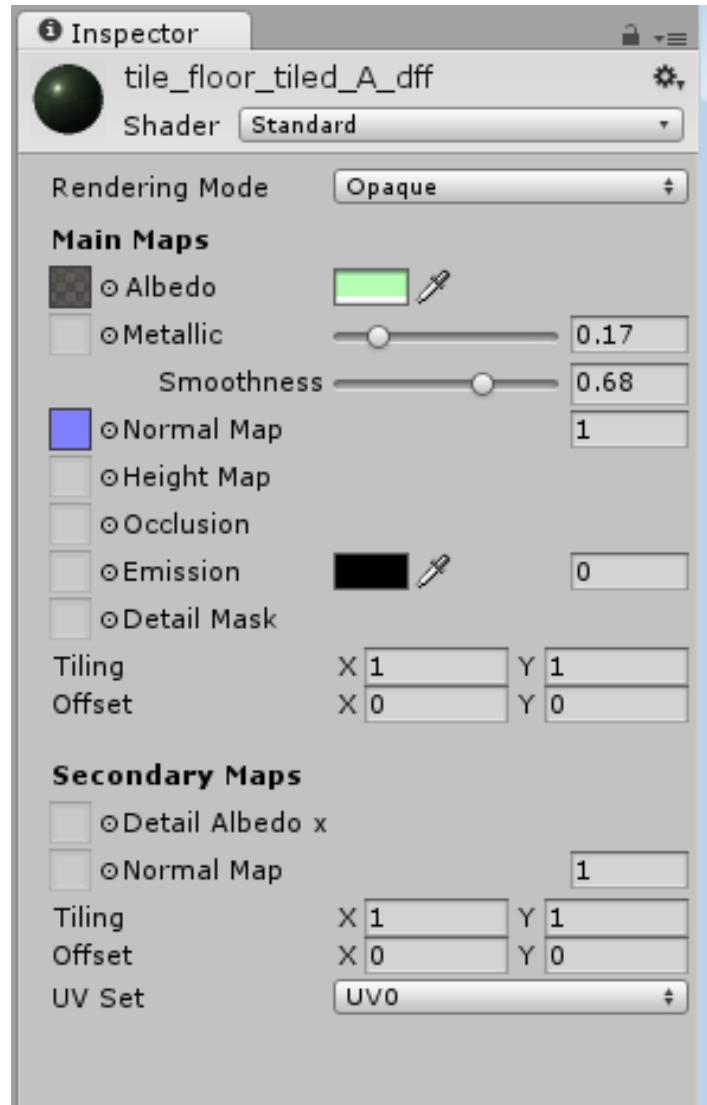
Slika 16: Nadgledni prozor

Ovaj prozor se koristi za promjenu svojstava gotovo svega unutar Unityja, od objekata, materijala pa sve do postavki samog editora. Kada odaberete neki objekt unutar hijerarhije ili scene, nadgledni prozor prikazuje sva svojstva koja opisuju taj objekt, na slici 17 prikazan je izgled prozora kada je odabran objekt igrač (engl. *Player*) osim podataka o položaju i veličini tu su prikazane i sve ostale komponente vezane za taj objekt, podaci o tome je li u pitanju fizički objekt, sadrži li neke od standardnih komponenti unutar Unityja kao što je sudarač (engl. *Collider*) i slično te naravno skripte koje pripadaju tom objektu, npr. "Player" skripta koja je prikazana na slici. Osim imena skripte tu su prikazane sve varijable koje su javne (engl. *Public*), nazivi te vrijednosti.



Slika 17: Nadgledni prozor igrač (engl. *Player*)

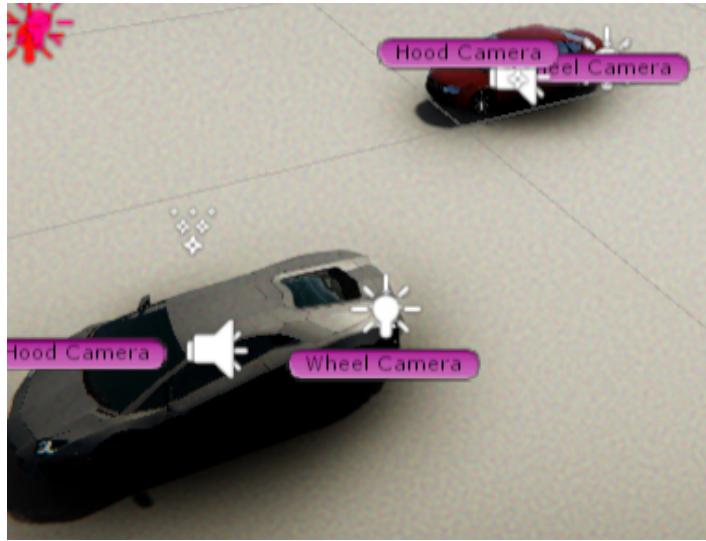
Na slici 18 prikazane su postavke za svojstvo objekta materijal. Najvažnije su postavke načina prikazivanja u sceni te dodavanja glavnih tekstura za objekt (engl. *Maps*).



Slika 18: Prikaz postavki za materijal

Komponente se na objekt dodaju vrlo jednostavno, samo treba pritisnuti botun dodaj komponentu (engl. *Add components*) te izabratи ono шto je potrebno, a kada je riječ o dodavanju skripti tada je vrlo jednostavno možete povući iz projektnog prozora na sami objekt. Vrlo je važno paziti na ovisnosti o ako postoje, zbog toga što skripte koje ovise o nekim drugim dodaju automatski te skripte na objekt, a isto tako te skripte koje su dodane ne mogu se izbrisati jer postoji ovisnost s glavnom skriptom.

Kako bi se lakše snalazili u sceni, moguće je objektu pridružiti boju, odnosno sličicu koja se prikazuje iznad objekta, tako moguće je lakše vidjeti gdje se nalaze svi objekti tog tipa u sceni. Isto tako bilo koja slika koja je uključena u projekt može predstavljati sličicu. Primjer postavljene ljubičaste sličice za komponente kamere koje su vezane za vozilo, prikazan je na slici 19.



**Slika 19:** Primjer prikaza sličica s nazivom za objekte istog tipa

Svaki objekt ima svoja svojstva, te se vrijednosti mogu mijenjati ovisno o potrebi. Kada se o objektima koji se koriste u svrhu uređenja scene, to su najčešće veličina, položaj, ima li objekt sudarač, njegov materijal i slično, dakle objekti koji su uglavnom statični. Isto tako ako se radi o objektima čija je svrha poboljšanje ugođaja igre, onda ti objekti najčešće imaju komponentu svjetlost, gdje možemo podesiti intenzitet, boju, radius osvjetljenja i slično ili zvuka koji se može aktivirati prilikom nekog događaja, ili biti aktiviran konstantno tijekom igre. Druga vrsta objekata su objekti koji s kojima je omogućena interakcija, dakle objekti koji uglavnom sadrže neku skriptu preko koje im je definirana uloga unutar scene, odnosno igre, to mogu biti stvari, likovi, i slično. Prilikom igranja, aktiviraju se funkcionalnosti skripti koje su dodane na te objekte, to može biti kretanje, govor, uporaba alata itd.

Još jedna opcija nadglednog prozora je izbor normalnog načina rada ili rada za pronalaženje grešaka (engl. *Debug mode*). Način rada za pronalaženje grešaka se koristi kao i u svim programskim jezicima, služi za praćenje promjena stanja varijabli, možemo vidjeti i privatne varijable, ali ih ne možemo mijenjati.

Neki drugi prozori koji će naknadno biti opisani su:

- Konzolni prozor,
- prozor animacija,
- prozor za praćenje performansi,

- prozor za osvjetljenje,
- Prozor za upravljanje s postavkama optimizacije (engl. *Occlusion Culling*).

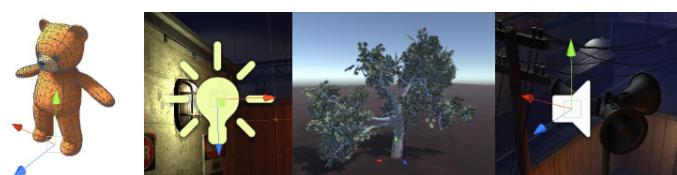
## 2.4. Stvaranje igre

Ono što je jedna od posebnosti Unityja je što nije potrebno imati godine iskustva u kodiranju da bi napravili igricu. Uz praćenje i razumijevanje jednostavnih procesa izrade igre, u vrlom kratkom roku započet ćete s izradom igara.

U ovom poglavlju će biti opisan općeniti pristup radu na igri, bit će izdvojene glavne komponente, koje se najčešće koriste tijekom rada unutar Unityja, odnosno tijekom stvaranja igre. Većina ovih koncepata su bazirana na izradi skripti, što će kasnije biti detaljnije obrađeno.

Tijekom izrade igre, sva okolina i izbornici su sadržani unutar scena. Može se reći da je svaka scena jedinstvena razina (engl. *Unique level*). Unity sprema scene unutar direktorija sa svim dodacima, odnosno unutar prozora projekta gdje se nalaze svi ostali dodaci. Tijekom izrade igre važno je paziti na urednost projekta, tako da je korisno napraviti direktorij koji će sadržavati sve scene vezane za taj projekt.

Već je mnogo puta spomenut izraz objekt igrice (engl. *Game object*), to je naravno, s razlogom. Objekti su najvažniji dio Unityja, od likova, stvari pa do efekata, kamere i slično. Međutim, objekt sam po sebi ne može ništa raditi ako nema dodanu komponentu skripte u kojoj je programirana logika za taj objekt. Na slici 20 prikazana su četiri različita tipa objekata. Objektima se trebaju pridružiti odgovarajuće komponente kako bi se postigla funkcionalnost koju je korisnik zamislio.



**Slika 20:** animirani lik, svjetlost, stablo, izvor zvuka

Neke od najčešćih komponenti koje se koriste tijekom izrade igre su:

- podaci o položaju i veličini (engl. *Transform*) ,
- kruto tijelo (engl. *Rigidbody*),

- svjetlost,
- zvuk,
- kontroler lika,
- sudarač,
- učitavač oblika (engl. *Mesh renderer*),
- kontroler animacije,
- C# skripte.

Svi objekti se mogu aktivirati ili deaktivirati unutar scene preko nadglednog prozora, ali i unutar samih skripti, što je vrlo korisno, bilo da je u pitanju neki efekt koji je pridružen objektu te se treba aktivirati samo u određenom trenutku, bilo da je riječ o objektu koji se može uništiti, ali će se nakon nekog vremena ponovo pojaviti u igri.

Objektu se može pridružiti i tag, referentni naziv za određene objekte. Primjerice, ako je potrebno, moguće je napraviti tag "Igrač" za sve likove kontrolirane od strane igrača te tag "Neprijatelj", ako se radi o protivniku. To je vrlo korisno jer je tako moguće napisati logiku unutar kôda, gdje će se različiti objekti ponašati na različiti način ovisno o tagu.

Na jednostavan način provjeravamo je li se objekt odgovarajućeg taga sudario s objektom kojeg je moguće skupiti, i ako je, igrač će prikupiti stvar, a dalje je programski riješeno ponašanje ovisno o prikupljenoj stvari. Primjer kôda je dan u ispisu 1.

```
public class PickupItem : MonoBehaviour {

    void OnTriggerEnter(Collider collider)
    {
        if (collider.tag != "Player")
        {
            return;
        }
        Pickup(collider.transform);
    }

    public virtual void OnPickup(Transform item)
    {
```

```

        print("colliding");
    }

void Pickup(Transform item)
{
    OnPickup(item);
}
}

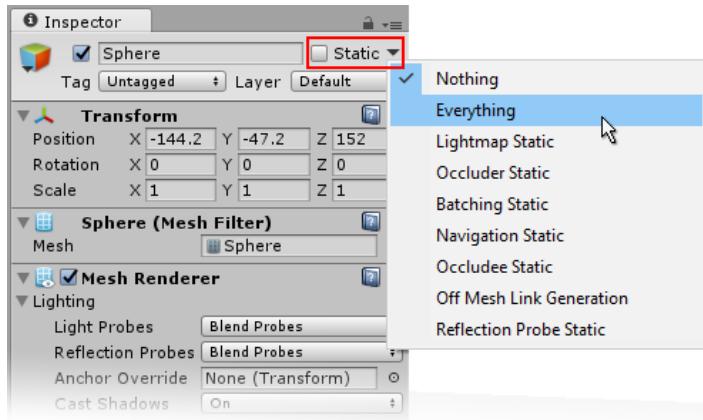
```

### Ispis 1: Uporaba taga

Nove tagove je vrlo jednostavno dodati preko nadglednog prozora, gdje se preko padajućeg izbornika pristupa tagovima, te odabere opcija "dodaj tag". Svaki objekt može imati samo jedan tag. Unity ima određene tagove već napravljene, to su uobičajeni tagovi koji se koriste tijekom izrade igre:

- Untagged,
- Respawn,
- Finish,
- EditorOnly,
- MainCamera,
- Player,
- GameController.

Važno svojstvo objekata je jesu li statični ili ne. To svojstvo je bitno kad je u pitanju optimizacija, a govori nam može li se objekt micati ili ne. Na taj način neka svojstva, kad je riječ o performansama, mogu biti unaprijed izračunata, odnosno optimizirana. To se u pozadini izvršava tako da se više statičnih objekata poveže u jedan veliki objekt zvan gomila (engl. *Batch*). Na slici 21 je prikazan nadgledni prozor s padajućim izbornikom gdje imamo različite opcije kada su u pitanju postavke statičnih objekata.

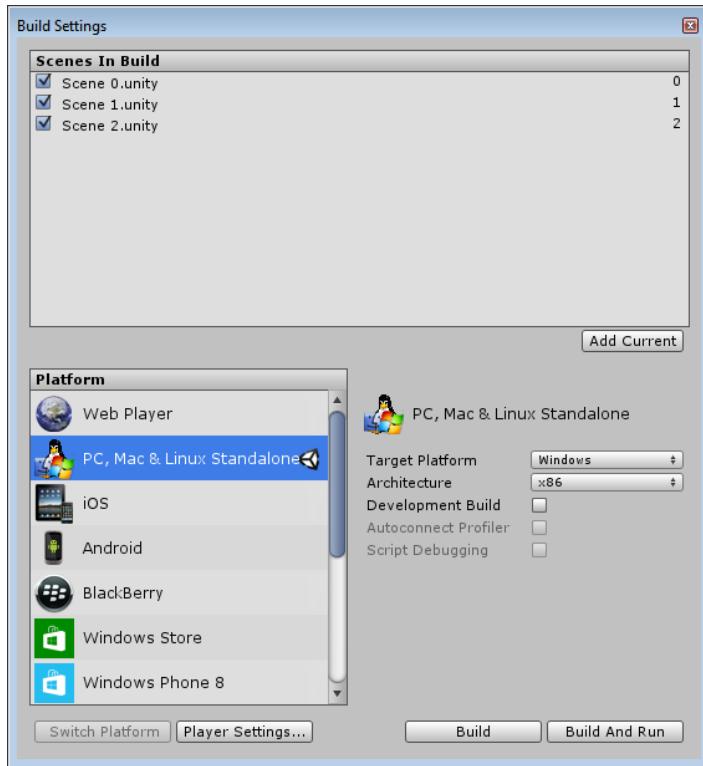


**Slika 21:** Izbornik za odabiranje različitih svojstva statičnih objekata

Tijekom rada na projektu, u nekom trenutku će biti potrebno napraviti izvršnu datoteku za testiranje dosadašnjeg rada. U tom slučaju je potrebno pristupiti postavkama za izradu projekta, dakle za stvaranje izvršne datoteke. Potrebno je dodati sve scene koje korisnik želi uključiti u datoteku, te ih poredati redoslijedom kakvim želite da se učitavaju. Svaka scena ima, osim naziva i indeksa, koji se inkrementira kako se dodaju scene. To je bitno kada se u programskom rješenju želi pristupiti određenoj sceni u trenutku izvršavanja određene radnje.

Kad se radi projekt koji se može pokretati na različitim platformama treba uzeti u obzir da se igra napravljena za računalo, koja ima stabilne performanse, ne mora jednako ponašati na mobilnim uređajima, iz jednostavnog razloga što mobiteli nemaju jednaku procesorsku moć kao računala.

Na slici 22 prikazan je izgled prozora prilikom stvaranja izvršne datoteke. Važno je napomenuti da treba izabrati odgovarajuću platformu za koju želite napraviti izvršnu datoteku.

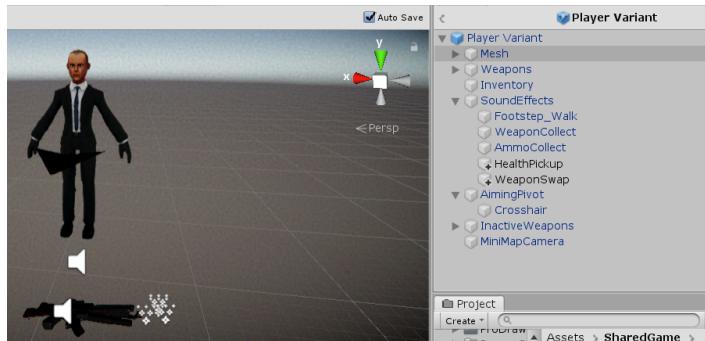


Slika 22: Prozor s postavkama za izradu izvršne datoteke

Unity ima sistem koji omogućava stvaranje već gotovih uzoraka koji se kasnije mogu jednostavno upotrebljavati kada god su potrebni, sa svom djecom, komponentama te svojstvima koji opisuju taj objekt. To su tzv. uzorci (engl. *Prefabs*).

Da bi se objekt koji je konfiguriran na određeni način, npr. lik koji nije kontroliran od strane igrača (engl. *NPC-non-player character*), ili možda određeni objekt koji je dio garniture ponovno upotrijebio u nekim drugim scenama onda je preporučljivo taj objekt pretvoriti u uzorak, tako je osigurano da je taj objekt u svim svojim pojavljivanjima jednak. To je puno bolje od standardnog kopiranja objekta, jer se svaka promjena na uzorku propagira na sve uzorke gdje god da se pojavljuju. Isto tako, moguće je određenu instancu prilagoditi ovisno o potrebi scene te napraviti varijantu tog uzorka.

Uz već navedene primjere korištenja uzoraka, uobičajeni primjeri su još instanciranje projektila ili glavnog lika u igri, npr. uzorak igrača može se nalaziti na polaznoj točki svake scene. Na slici 23 prikazan je uzorak Igrača.



**Slika 23:** Uzorak Igrača

Stvaranje uzorka je vrlo jednostavan proces, željeni objekt se povuče iz prozora hijerarhije u prozor projekta i stvoren je uzorak tog objekta. Ako već postoji uzorak tog objekta, onda je moguće odabrati hoće li to biti varijanta uzorka ili novi uzorak. Taj uzorak je onda moguće povući ili direktno u scenu ili u hijerarhiju te će se instancirati taj objekt. Objekti koji su uzorci, prikazani su plavom bojom. Duplim klikom na objekt ulazi se u prozor za uređivanje uzorka, po završetku, odnosno izlaskom iz prozora za uređivanje promjene se automatski spremaju. Opcija automatskog spremanja se može isključiti te je tada potrebno ručno spremiti sve promjene kako bi se one propagirale.

Osim u prozoru za uređivanje uzorka, promjene se mogu propagirati i u samoj sceni, odnosno u nadglednom prozoru, tako da se potvrde sve promjene na instanci tog uzorka, a ako je potrebno moguće je i vratiti objekt u početno stanje. Uzorke je moguce i raspakirati, s tim je izbrisani uzorak, ali ne i sami objekt. Desnim klikom na objekt pritisnite raspakiraj i objekt više neće biti uzorak.

Prethodno je objašnjeno kako koristiti mogućnosti uzorka na razini editora, u nastavku će biti opisani neki primjeri kada je korisno koristiti uzorke i unutar skripti. Jedan od primjera je izgradnja zida. Skripta 2 prikazuje kako pristupiti izradi zida, bez korištenja uzorka.

```
public class Instantiation : MonoBehaviour
{
    void Start()
    {
        for (int y = 0; y < 5; y++)
        {
            for (int x = 0; x < 5; x++)
```

```
        {
            GameObject cube = GameObject.CreatePrimitive(PrimitiveType.Cube);
            cube.AddComponent();
            cube.transform.position = new Vector3(x, y, 0);
        }
    }
}
```

## Ispis 2: Skripta za izradu zida

Skripta se doda na prazan objekt te nakon što se pokrene igra, izgradi se zid. Problem je u tome što je ovakvim pristupom, svaka promjena na zidu, bilo to dodavanje tekstura, mijenjanje fizičkih svojstava, trenja i slično, nova linija kôda. Koristeći uzorke prvo se uredi objekt po želji te se onda u skripti oslanjamo na taj uzorak. Primjer takve skripte je u ispisu 3.

```
//Instantiate accepts any component type, because it instantiates the  
GameObject  
//brick is reference to prefab model  
  
public Transform brick;  
  
void Start()  
{  
    for (int y = 0; y < 5; y++)  
    {  
        for (int x = 0; x < 5; x++)  
        {  
            Instantiate(brick, new Vector3(x, y, 0), Quaternion.identity);  
        }  
    }  
}
```

**Ispis 3:** Skripta za izradu zida koristeći uzorke

Ovim pristupom, ne samo da je skripta urednija, već sve promjene koje želimo

napraviti na uzorku, ne radimo preko kôda već direktno na tom uzorku, te prilikom novog pokretanja igre, zid nastaje sa svim novim svojstvima.

Drugi primjer je instanciranje projektila. Projektil može imati različita svojstva, kao što su trag koji ostavlja iza sebe, zvuk, trag koji nastaje nakon sudara, svjetlost i slično. Sve se to može i programski riješiti, međutim puno je jednostavnije napraviti uzorak sa svim potrebnim efektima, a unutar skripte samo instancirati taj objekt kada je potrebno (skripta 4).

```
void CheckDestructable(RaycastHit hitInfo)
{
    var destructable = hitInfo.transform.GetComponent();

    destination = hitInfo.point + hitInfo.normal * .01f;

    if (hitInfo.transform.tag == "Enemy" || hitInfo.transform.tag == "Player")
    {
        Transform blood = Instantiate(bloodImpact, destination,
            Quaternion.LookRotation(hitInfo.normal) * Quaternion.Euler
            (0, 180f, 0));

        flag = true;
    }
    else if (flag == false)
    {
        Transform hole = Instantiate(bulletHole, destination, Quaternion
            .LookRotation(hitInfo.normal) * Quaternion.Euler(0, 180f, 0))
        ;
        hole.SetParent(hitInfo.transform);
    }

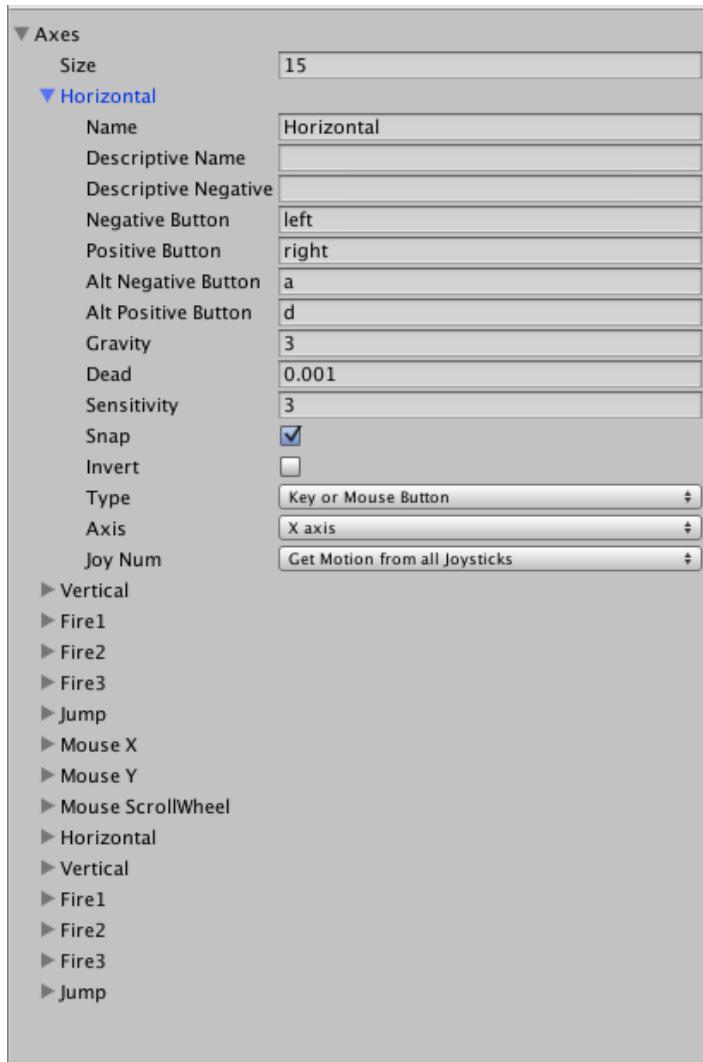
    if (destructable == null)
    {
        return;
    }
    destructable.TakeDamage(damage);
    flag = false;
}
```

---

#### Ispis 4: Instanciranje metka

Treći primjer je korištenje lutki (engl. *Ragdoll*), odnosno smrt neprijatelja i slično. Dakle, igrač ubije neprijatelja, koji sadrži razne skripte i komponente koje su potrebne za funkcioniranje, i u slučaju smrti trebamo smisliti logiku što s tim likom. Jedna od opcija je deaktiviranje s komponenti, međutim, puno bolji pristup je brisanje tog objekta te instanciranje uzorka koji će npr. pokrenuti animaciju smrti, te će predstavljati mrtvo tijelo. Slična logika se može primijeniti kada su u pitanju vozila itd. Ovim pristupom puno je manja vjerojatnost neočekivanog ponašanja objekata u igri, čišći je kôd i što je najvažnije jednostavno ga je ponovno upotrijebiti.

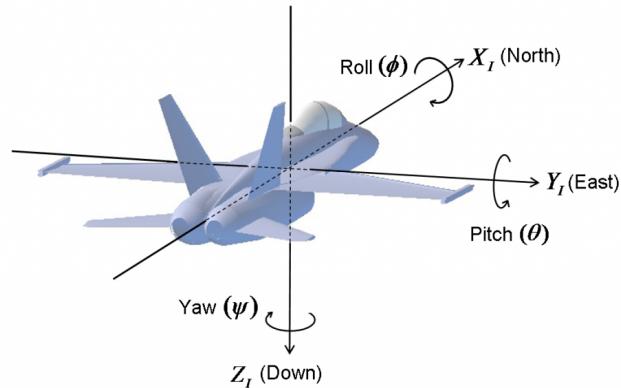
Unity podržava sve standardne uređaje za unos. Kada je riječ o kontrolama igre, unos može biti preko tipkovnice, kontrolera, miša te svi standardni načini unosa u mobilnim igrama. Osim standardnih postavki za kontrole, moguće je definirati vlastite kontrole. Za to je potrebno promijeniti vrijednost u postavkama na slici 24 te se referirati na taj botun preko skripte.



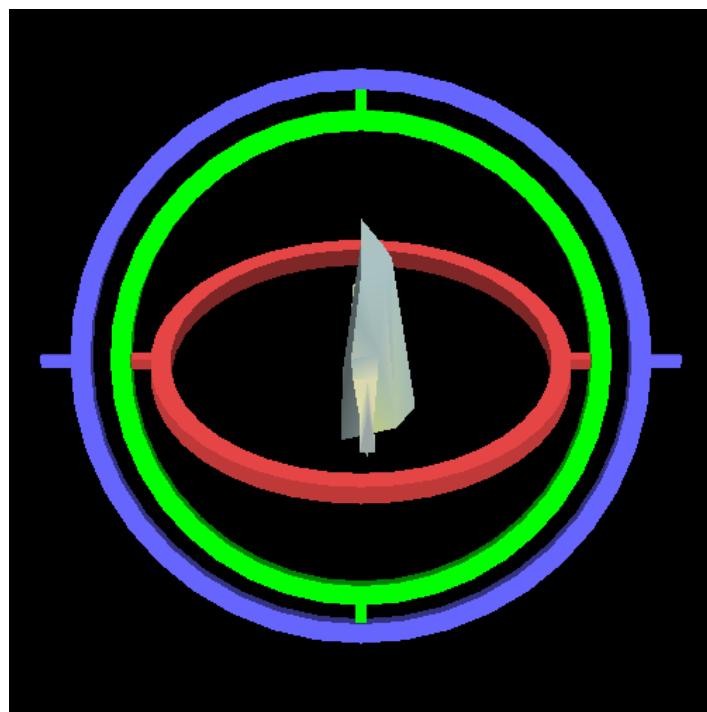
**Slika 24:** Postavke kontrola

Za prikazivanje rotacije i orijentacije Unity koristi dvije tehnike, to su kvaternioni i Eulerovi kutovi. Obe tehnike imaju svoje prednosti i mane. Eulerovi kutevi su način prikazivanja 3D orijentacije objekta koristeći kombinaciju rotacija oko triju različitih osi. Na slici 25 prikazana su usmjerenja pojedinih osi kada je objekt u neutralnom položaju. Postoje tri vrste rotacije, to su Yaw rotacija, odnosno rotacija oko osi z za vrijednost kuta  $\Psi$ , Pitch oko osi y za vrijednost  $\Theta$ , i Roll rotacija oko osi x za vrijednost  $\Phi$ . Orientacija se mijenja tako da se vrijednosti rotacija dodjeljuju redom oko svake osi. Eulerovi kutovi su jednostavnii intuitivni za analiziranje te je njima moguće prikazati okrete veće od 180 stupnjeva. Problem Eulerovih kutova je gimbalno zaključavanje, to je kada se orijentacija ne može jedinstveno prikazati, odnosno izračunati koristeći Eulerove kutove. Događa se kada se dvije rotacije vrte oko iste osi te su približno vrijednosti 90 stupnjeva. Primjer zaključavanja prikazan je na slici 26, dva prstena su u istoj ravnini te je jedan stupanj slobode izgubljen, odnosno

promjene na plavom i crvenom prstenu stvaraju jednaku rotaciju objekta.



**Slika 25:** Prikaz orijentacijskih osi objekta



**Slika 26:** Primjer gimbalnog zaključavanja

Kvaternioni se mogu koristiti kako bi se prikazala orijentacija ili rotacija objekta. Kvaternion je vektor s četiri elementa koja mogu predstavljati bilo koju rotaciju u 3D koordinatnom sustavu. Kvaternion je sastavljen od jednog realnog i tri kompleksna elementa, to su X, Y, Z i W vrijednosti. X, Y i Z se mogu opisati kao vektorski dio oko kojih bi se trebala događati rotacija, dok je W skalarni dio koji označava vrijednost rotacije koja bi se trebala izvršavati oko vektorskog dijela. Kvaternion može prikazivati orijentaciju i rotaciju

te nema problema vezanih za gimbalno zaključavanje. Izborom krivulje kao što je sferna linearna interpolacija kvaterniona, gdje jedan kraj predstavlja početnu rotaciju, a drugi predstavlja željenu krajnju rotaciju postiže se glatka, postepena rotacija koja je potrebna kako bi se dobili što vjerodostojniji rezultati u video igrama. Nedostaci kvaterniona su nemogućnost prikazivanja rotacija većih od 180 stupnjeva te težina očitavanja vrijednosti. Zbog toga su sve vrijednosti rotacija pohranjene kao kvaternioni, ali su predstavljene Eulerovim kutovima u nadglednom prozoru.

Prilikom obrađivanja rotacije u skriptama, preporučljivo je koristiti Quaternion klasu. Unutar klase nalazi se mnoštvo korisnih metoda kao što su:

- Quaternion.LookRotation,
- Quaternion.AngleAxis,
- Quaternion.FromToRotation,
- Quaternion.Slerp,
- Transform.Rotate,
- Transform.RotateAround.

## 2.5. Grafika i optimizacija

Unity ima set različitih postavki kvalitete grafike u namjeri da se poboljša učitavaњe i performanse. Općenito govoreći, visoka kvaliteta znači i lošije performanse na starim slabim računalima i mobilnim uređajima i zato nije uvijek pametno ciljati na najbolji izgled igre.

U postavkama za kvalitetu nalazi se matrica gdje su stupci različite platforme, a retci razina kvalitete, što je prikazano na slici 27. Moguće je dodati i vlastite postavke, kao i urediti ili izbrisati već postojeće.



Slika 27: Prikaz postavki grafike

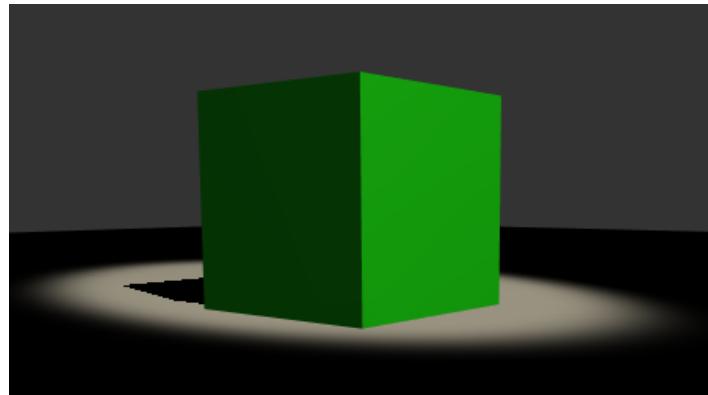
Kada se dodaju vlastite postavke grafike, treba podešiti učitavanje, odnosno postavke kao što su broj piksela svjetla, kvaliteta tekstura, uglađeni rubovi, refleksije i slično, zatim postavke vezane za sjene, kao što su kvaliteta sjena, rezolucija, projekcija udaljenost itd. Ostale postavke su vezane za broj kostiju lika koje sudjeluju u animaciji, osvježavanje, maksimalan broj zraka za aproksimiranje prašine, razine detalja i slično.

Grafika je vrlo bitan segment izrade igara, ne samo po pitanju izgleda igre, koja treba lijepo izgledati i biti pristupačna i zanimljiva igraču, već treba voditi računa o ograničenjima platformi za koje je igra napravljena. Samim time u pitanju je vrlo širok pojam. U ovom poglavlju opisani su samo najvažniji pojmovi koji su bili korišteni u projektu kao što su modeli, svjetlost, efekti te na kraju malo i optimizacija.

Poželjno je da su modeli koji se koriste unutar projekta isto tako optimizirani. To se najviše odnosi na teksture. Kvalitetno odrađeni modeli najčešće imaju jednu veliku tzv. atlas mapu koja je napravljena od više različitih tekstura. To omogućava da se određeni dijelovi okoline mogu prikazati kao jedan veliki objekt, ali i dalje sa svim svojim detaljima. Korištenjem više jedinstvenih teksutra za prikazivanje pojedinog objekta postiže se detaljniji prikaz, međutim na taj način igra je lošije optimizirana. Sve se teksture trebaju posbeno obraditi, kao i kompletni objekti što zahtjeva veliku procesorsku snagu te dolazi do lošijih performansa igre što može pokvariti doživljaj igranja.

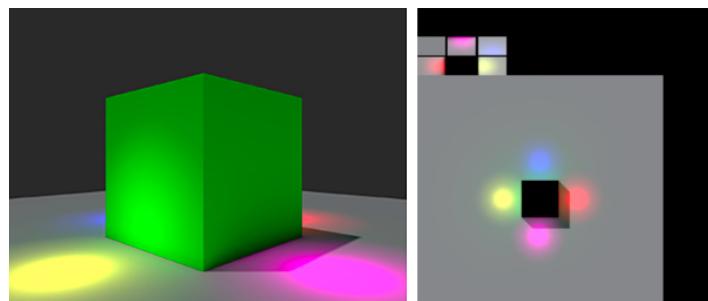
Svjetlost djeluje na način da osvjetljuje predmet, te taj predmet reflektira svjetlost, a u pozadini stvara sjenu. Baš kao u stvarnom svijetu, nije baš jednostavno postići realnu scenu, pa treba voditi računa o više stvari. Postoje dva ključna pojma po tom pitanju, a to su predizračunata i svjetlost u stvarnom vremenu. Tri su vrste izvora svjetlosti, to su direktna svjetlost, reflektor te snop. Oni spadaju u svjetlost u stvarnom vremenu i koriste se za osvjetljavanje objekata. Problem je što ovim osnovnim principom ne možemo postići dovoljno visoku kvalitetu osvjetljenja, sjene su kompletno crne i pikseliziranih rubova, nema reflek-

tiranog svjetla i samo površine neposredno uz objekt su pod utjecajem, što je prikazano na slici 28.



**Slika 28:** Osnovno osvjetljenje kocke

Unity može izračunati kompleksne statičke svjetlosne efekte korištenjem tehnike globalne iluminacije te ih sprema u referentnu teksturnu mapu zvanu svjetlosna mapa (engl. *light-map*). To je tzv. prženje svjetlosti (engl. *baking*). Prilikom ovog procesa, efekti svjetlosti koji nastaju djelovanjem na objekte u sceni koji su označeni kao statični izračunati su te upisani u teksturu koja je preko cijele scene što stvara geometriju svjetlosnih efekata. To je prikazano na slici 29. Lijevo je prikazana jednostavna scena gdje je primijenjena svjetlosna mapa, a desno tekstura koja je generirana. Koristeći ovu metodu, efekti svjetlosti na odbijane objekte se ne mogu mijenjati.



**Slika 29:** Efekti svjetlosne mape

Ako želimo postići stvarno vremensku promjenu efekta svjetlosti, dakle dinamičko osvjetljenje, trebamo koristiti predizračunatu stvarno vremensku globalnu iluminaciju.

Dodatni alati koji pospješuju osvjetljenje su svjetlosne i reflektivne sonde. Svjetlosne sonde pojačavaju efekt svjetlosti, odnosno prikupljaju informacije o praznom prostoru gdje prolazi svjetlost. Reflektivne sonde prikupljaju informacije o refleksijama koje različite

površine stvaraju, npr. u prostoriji gdje je mnoštvo metalnih površina, stakala i slično. Korištenjem ove tehnike moguće je postići realnu scenu.

Sve ovo bilo bi beskorisno da na neki način ne možemo prikazati scenu. Naravno, za to su potrebne kamere. Svaka scena, da bi se mogla prikazati, treba imati najmanje jednu kameru. Standardna svojstva kamere su pozicija, rotacija, kut prikaza, način na koji obrađuje svjetlost i slično. Kamera se može koristiti kao objekt koji osluškuje zvukove, a ono što je jako bitno svojstvo kamere je mogućnost obrađivanja samo onog što je u kutu pregleda, dok se ostali dijelovi scene procesorski ne obrađuju. To svojstvo se naziva blokada prikaza(engl. *Occlusion culling*).

Kamera sama po sebi prikazuje scenu bez dodatnih efekata, međutim dodavanjem komponente za naknadno obrađivanje slike (engl. *Post-processing effect*) moguće je stvoriti mnoštvo zanimljivih efekata. Ovisno o stilu igre koji se želi dobiti, mijenjanjem različitih postavki ove komponente moguće je zaista oživjeti scenu, kompletno dočarati željenu atmosferu u određenom dijelu igre. Ono što će sigurno pomoći tijekom izrade igre je činjenica da što kamera ne vidi, nije vidljivo niti igraču. Korištenjem ovog znanja, moguće je zamaskirati različite situacije unutar igre.

Vrijedno je navesti još jednu komponentu koja se može koristiti kako bi se dodatno uljepšalo scenu, a to je sistem prašine. Osim onog što sami naziv sugerira, ova komponenta se može koristiti za predstavljanje bilo kakvog dinamičkog efekta. U samom projektu, korišten je za izradu efekta pucanja, prašine u skladištu, efekta eksplozije, krvi itd.

Naravno, kao što je već spomenuto, performanse igre su vrlo bitne, a u želji da se postigne što bolje osvjetljenje i efekte treba paziti da to ne uzrokuje veliko pogoršanje. Unity ima ugrađen alat za praćenje performansi tijekom igranja, ali isto tako postoje vrlo dobri alati od treće strane. Jedan od njih je Graphy, besplatni alat koji je korišten tijekom izrade ovog projekta i koji se može preuzeti sa stranica Unity trgovine s dodacima <https://assetstore.unity.com/packages/tools/gui/graphy-ultimate-fps-counter-stats-monitor-debugger-105778>, autor je Tayx.

Lista ispod navodi ključne točke kada su u pitanju performanse te s tim završava poglavljje vezano za grafiku.

- Držati broj vrhova između 200 tisuća i 3 milijuna.
- Održavati što manju kompleksnost i manji broj materijala.

- Postaviti zastavicu za statičke objekte.
- Pržiti svjetlosne mape.
- Koristiti jedno direktno svjetlo.
- Koristiti kompresirane teksature te teksturne mape.
- Gdje god je moguće koristiti blokadu prikaza (engl. *Occlusion culling*).
- Koristiti manje kompleksne modele i animacije gdje je moguće.

Osim o grafičkoj kvaliteti igre, važno je voditi računa i o veličini završnog projekta. Načelno, najviše memorije zauzimaju teksture, zvukovi i animacije, dok skripte manje utječu na veličinu projekta. Prvi korak prema manjoj veličini projekta je kompresija tekstura, a osim toga moguće je i smanjiti fizičku veličinu teksture, odnosno smanjiti broj piksela. U postavkama je potrebno smanjiti maksimalnu veličinu te pri tom paziti da objekt koji koristi texture bude prepoznatljiv. Osim tekstura, moguće je komprimirati animacije, te tako dodatno uštedjeti na prostoru, ali treba uzeti u obzir da je to samo smanjenje veličine datoteke.

Kako bi spremili veliku količinu podataka, vrlo je dobra i korisna praksa koristiti skriptirane objekte (engl. *ScriptableObject*). To su podaci koji nisu vezani za instancu klase. Skriptirani objekti se najčešće koriste kada se želi uštedjeti na memoriji. Umjesto stvaranja duplicitiranih vrijednosti i spremanja istih u standardne skripte, moguće je napraviti skriptirani objekt za spremanje tih vrijednosti, pa tako svi objekti preko reference pristupaju tim podacima. Osim toga, skriptirani objekti su korisni kada je potrebno spremiti određene podatke tijekom igranja. Skriptirani objekti se ne pridružuju objektima igre poput standardnih skripti, već se spremaju kao dodaci unutar projekta. Primjer korištenja je prikazan u skripti ispod, skripta 5 prikazuje kako se stvara skriptirani objekt, a kako bi koristili te objekte, sve što je potrebno je referencirati se na taj objekt unutar skripte gdje ćemo koristiti te podatke.

```
using UnityEngine;

[CreateAssetMenu(fileName = "Soldier", menuName = "Data/Soldier")]
public class Soldier : ScriptableObject {

    public float RunSpeed;
    public float WalkSpeed;
    public float CrouchSpeed;
```

```
    public float SneakSpeed;  
}
```

### Ispis 5: Stvaranje skriptiranog objekta

Osim stvaranja skriptiranog objekta, na ovaj način smo dodali u Unity editor direktni pristup izradi nove datoteke koja će sadržavati potrebne postavke. Isto tako je moguće stvoriti vlastite prozore. Sve što je potrebno je napraviti skriptu koja nasljeđuje klasu EditorWindow, te implementirati grafičko sučelje. Iako ova mogućnost nije korištenja u projektu, jako je korisna, jer je na ovaj način moguće napraviti prozore koji se ponašaju slično kao npr. nadgledni prozor, i gdje se mogu pregledavati i uređivati svojstva važna za projekt.

## 2.6. Fizika, animacije, navigacija i platno

Kako bi se postigla realna fizika potrebno je postići da objekt ubrzava ispravno, da na njega utječe kolizija, gravitacija i druge sile. Unity ima već ugrađene komponente koje se brinu baš o tim segmentima. Samo s nekoliko postavki moguće je postići realno ponašanje. Kontroliranjem fizike preko skripti, moguće je objektu dodjeliti dinamike vozila, stroja, pa čak i komada odjeće.

Unity ima dva različita sistema obrađivanja fizike, 3D i 2D fizika. Glavni koncepti su uglavnom isti, međutim implementirani su različitim komponentama, npr. za 3D postoji Rigidbody komponenta, te analogno tome Rigidbody 2D za 2D fiziku.

Čvrsto tijelo (engl. *Rigidbody*) (u nastavku će se koristiti engleski naziv), je glavna komponenta koja omogućava fiziku objekta. Dodjeljivanjem ove komponente objektu, automatski je pod utjecajem gravitacije, a ako se doda i sudarač (engl. *Collider*) onda je i pod utjecajem kolizije s ostalim objektima.

Iz skripte kontroliramo ove objekte dodavanjem sile koja gura objekt, što znači da nije poželjno mijenjati poziciju preko skripte, već će se ugrađeni alat za obrađivanje tih rezultata pobrinuti o novoj poziciji objekta. Naravno nekad ćemo htjeti sami pomaknuti objekt na određenu poziciju, ali kada je riječ o standardnim kretanjama koje želimo postići sa objektom, potrebno je samo navesti vrijednost sile koja će djelovati na taj objekt, te odrediti smjer.

Kao što postoje primitivni oblici tako postoje i primitivni sudarači, a to su kocka,

kugla i kapsula. U većini slučajeva ovi sudarači su dovoljni, odnosno moguće ih je koristiti za većinu objekata u sceni. Dakle u slučaju da želimo omogućiti sudaranje s drugim objektima moramo dodati ovu komponentu. Nekada, ako je objekt previše kompleksan i želimo postići što realnije sudaranje, trebamo koristiti kompleksni sudarač koji se temelji na obliku samog objekta za kojeg želite koristiti ovaj sudarač (engl. *Mesh collider*). Ovi sudarači su naravno teži za obrađivanje te je gdje je god moguće poželjno koristiti primitivne sudarače.

Sudarači se koriste za tlo, zid, objekte kroz koje nije moguće proći i slično. Ovo su tzv. statički sudarači, dakle objekti koji se ne mogu pomicati u sceni. Ako se radi o objektu koji se može micati, odnosno ima komponentu Rigidbody, tada se radi o dinamičkim sudaračima. Efekt sudaranja ovisi o tome jesu li statički ili dinamički.

Kada se dogodi interakcija između sudarača, onda ovisno o površini i svojstvima iste, rezultat tog sudaranja je različit. Npr. ako se radi o površini kao što je led, onda će biti vrlo skliska, a ako je riječ o gumenoj površini, onda bi trebala pružiti dosta trenja itd. Postizanje ovih efekata nije baš jednostavno te zahtjeva dosta testiranja, ali moguće je postići jako realne rezultate. Sudarači su, osim uloge koju imaju vezano za fiziku, jako korisni kada su u pitanju programska rješenja različitih situacija u igri. Svaki sudarač ima svojstvo okidanja (engl. *Trigger*). Ovo svojstvo kada je postavljeno, tada sudaranjem s istim moguće je aktivirati različite događaje. Npr. ulaznjem u okidač moguće je aktivirati ispis nekog teksta na ekranu ili pokretanje kratkog filma unutar igre i slično. Baš ovo svojstvo je korišteno u projektu na velik broj različitih načina te će to biti predstavljeno u dijelu gdje će biti opisana igra i korištena programska rješenja. Lista navodi metode koje su dostupne kada su u pitanju okidači.

- OnCollisionEnter,
- OnCollisionStay,
- OnCollisionExit,
- OnTriggerEnter,
- OnTriggerStay,
- OnTriggerExit.

Svaki lik kontroliran od strane igrača, kao i ostali koji se korise u igri trebaju neku vrstu fizike bazirane na sudaranju. Npr. mogućnost kretanja, onemogućavanje propadanja

kroz tlo ili prolazanje kroz zidove. Kada se radi o 3D fizici, ovo ponašanje može se stvoriti korištenjem komponente kontroler lika (engl. *Character Controller*). Ova komponenta daje liku jednostavni sudarač u obliku kapsule koji je uvijek usmjeren prema gore te sami kontroler ima vlastite specijalne funkcionalnosti kao što su brzina i smjer. Objekt koji sadrži ovu komponentu ne može prolaziti kroz statičke sudarače, može detektirati uzvišene površine, pomicati ostale objekte koji na sebi imaju Rigidbody komponentu, ali neće mu se mijenjati akceleracija ovisno o tome sudari li se s nekim objektom itd. Ova komponenta je korištena na glavnom liku igre.

Sistem animacija unutar Unityja je napredan sustav s potpunom kontrolom animacija tijekom igranja, pozivanje eventa pomoću animacija, sofisticirani sistem stanja (engl. *State machine*), tranzicija, povezivanja više stanja u jednu animaciju itd. Osim toga, cijeli sistem je vrlo jednostavan i intuitivan, uz mogućnost pregledavanja animacija i prijelaza unutar pomoćnog prozora, povezivanje animacija s jednostavnim dodavanjem prijelaza, te opisivanja istih koristeći pomoćne varijable, kao što su provjera vrijednosti istina i laž, okidači te vrijednosti parametara kada je u pitanju pokretanje itd. Animacije se mogu uvoziti u Unity. Moguće je koristiti generičke i čovjekolike (engl. *Humanoid*) animacije, te jednostavno napravljene kontrolere animacija koristiti na više različitih likova.

Rad na animacijama se uglavnom vrši unutar dva prozora u Unityju, a to su prozor animacija, gdje je moguće napraviti ili urediti vlastite animacije i prozor kontroler animacija (engl. *Animator controller*) gdje se radi na stanjima, stvaranju logike kompleksnog sustava prijelaza među stanjima.

Za objekt koji želimo animirati potrebno je dodati jednu od tih dviju komponenti ili obje, te osim standardnih svojstava koje svaka od tih komponenti ima koje upravljuju animacija, moguće je unutar skripti pisati logiku. Unity prepoznaje je li model ljudski preko avatara, te je zbog sličnosti strukture kostiju različitih likova, vrlo jednostavno ponovno iskoristiti već napravljenu logiku.

Kada se stvara nova animacija, potrebno je paziti na nekoliko ključnih elemenata, a to su model, momenat u kojem se događa promjena, vrsta promjene, bilo pozicija, rotacija i slično, trajanje te promjene, te završna pozicija. Nakon toga sve što je potrebno je, dodati komponentu animacija objektu koji želimo animirati, te odabratи hoće li se animacija pokretati automatski, hoće li se ponavljati ili će se jednostavno sva logiku implementirati preko

skripte.

Kada se radi s kontrolerom animacija, najčešće su u pitanju stanja, tranzicije i mišanje različitih animacija. Sve što je potrebno su model i animacija. Mašina stanja je bazirana na dodavanju različitih stanja, te stvaranje tranzicije među istim. Tranzicija je prelazak iz jednog stanja u drugo uslijed promjene odgovarajućeg parametra, dakle potrebno je dodati parametar, napraviti tranziciju te unutar skripte napisati logiku, što je i prikazano u dijelu kôda za animacije 6.

```
animator.SetFloat("Vertical", GameManager.Instance.InputController.  
    Vertical);  
  
animator.SetFloat("Horizontal", GameManager.Instance.  
    InputController.Horizontal);  
  
animator.SetBool("IsFalling", GameManager.Instance.LocalPlayer.  
    PlayerState.IsFalling);  
  
animator.SetBool("EnterCar", GameManager.Instance.InputController.  
    enterCar);  
animator.SetBool("ExitCar", GameManager.Instance.InputController.  
    exitCar);  
  
animator.SetBool("IsReloading", GameManager.Instance.InputController  
    .Reload);  
animator.SetBool("WeaponSwitch", GameManager.Instance.  
    InputController.MouseWheelDown || GameManager.Instance.  
    InputController.MouseWheelUp);  
animator.SetBool("IsJump", GameManager.Instance.InputController.  
    Jump);  
animator.SetBool("IsRunning", GameManager.Instance.InputController.  
    IsRunning);  
animator.SetBool("IsSneaking", GameManager.Instance.InputController  
    .IsSneaking);  
animator.SetBool("IsCrouched", GameManager.Instance.InputController  
    .IsCrouched);  
animator.SetBool("IsFiring", GameManager.Instance.InputController.  
    Fire1);  
animator.SetBool("IsAiming", GameManager.Instance.LocalPlayer.
```

```

PlayerState.WeaponState == PlayerState.EWeaponState.AIMING ||
GameManager.Instance.LocalPlayer.PlayerState.WeaponState ==
PlayerState.EWeaponState.AIMEDFIRING);

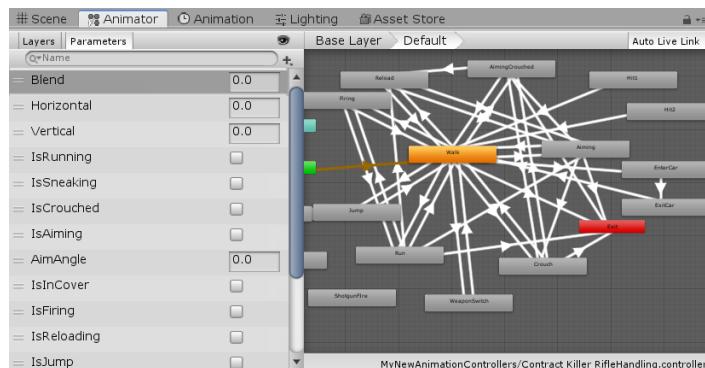
animator.SetFloat("AimAngle", playerAim.GetAngle());

animator.SetBool("IsInCover", GameManager.Instance.LocalPlayer.
PlayerState.MoveState == PlayerState.EMoveState.COVER);

```

### Ispis 6: Skripta animacija

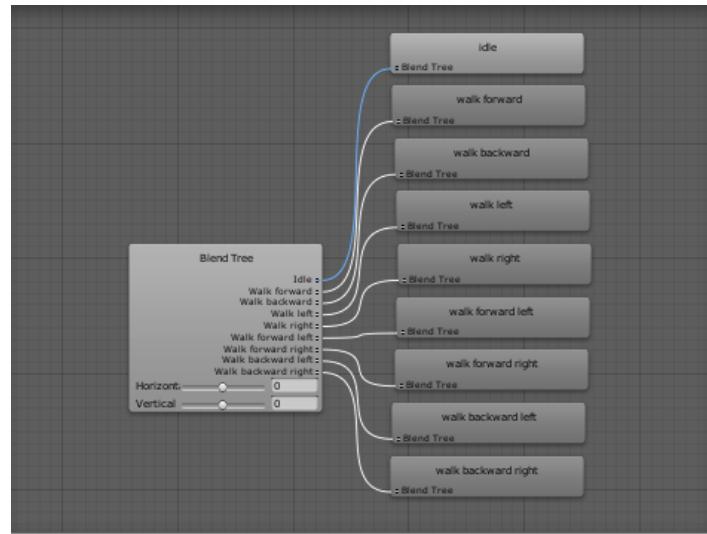
Kao što je vidljivo, pristupa se kontroleru animacija preko animatora, te se postavlja odgovarajuća vrijednost, npr. postavljanjem vrijednosti "IsReloading" govorimo kontroleru animacija da postavi parametar tog naziva te se, ako je dozvoljeno, događa prijelaz iz prethodnog stanje u stanje punjenja oružja. Na slici 30 je prikazan prozor kontrolera animacija za igrača.



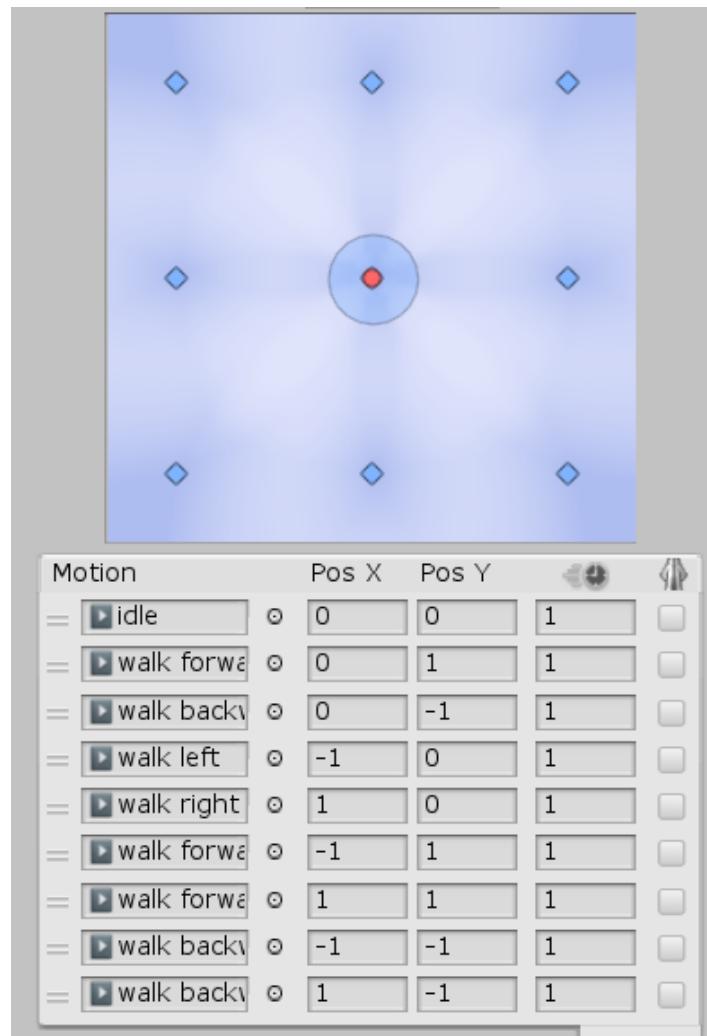
Slika 30: Kontroler animacija

Jedan od standardnih zadataka kada su u pitanju animacije je miješanje jedne ili više sličnih animacija. Za to nam služi stablo miješanja animacija (engl. *Blend tree*). Možda najpoznatiji primjer je miješanje između hodanja i trčanja ovisno o brzini lika. Drugi primjer je naginjanje lika lijevo ili desno dok trči itd. Važno je razlikovati tranzicije između stanja i stabla miješanja, iako i jedno i drugo služi za glatki prijelaz između animacija, koriste se u različitim situacijama. Dok tranzicije služe za prijelaz između stanja unutar nekog perioda vremena, miješanje se događa korištenjem više animacija te ovisno o vrijednosti određenih parametara dobiva se konačni izgled animacije. Može se odabrat 1D ili 2D miješanje, ovisno o odabranom, bit će potrebno mijenjati vrijednost jednog ili više parametara. Osim toga, rad sa stablom je sličan radu sa stanjima, a to je dodavanje stanja unutar stabla, te dodavanjem

animacije. Na slikama 31 i 32 prikazan je prozor stabla mijеšanja animacija.



Slika 31: Nazivi animacija



Slika 32: Animacije i parametri

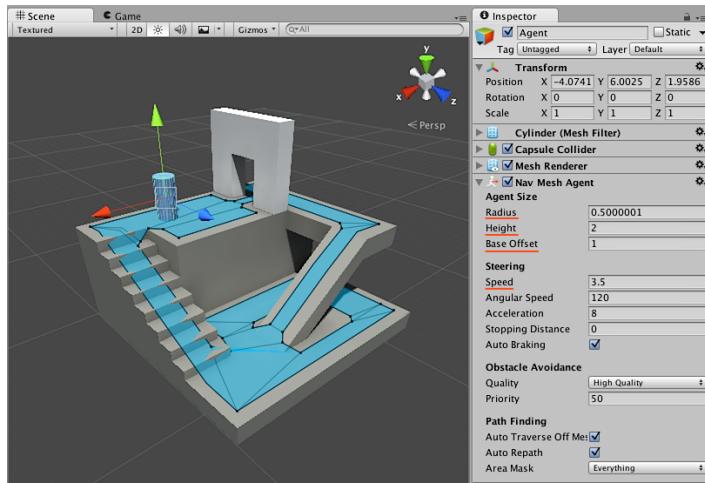
Ovisno o tome koliko je kompleksan izrađeni sustav animacija, moguće je stvoriti više slojeva animacija, te onda razdvojiti slojeve ovisno o tome u kojem je stanju lik. Dakle, moguće je napraviti npr. sloj nenaoružan lik, dodati odgovarajuća stanja, te ako je lik naoružan, sloj gdje će biti stanja sukladno tome.

Sistem navigacije u Unityju omogućava stvaranje prohodnog područja na osnovu geometrije unutar scene. Tako je moguće dodavati likove koji će se korištenjem sistema pronalaženja puta kretati unutar scene prema određenim pravilima, što uključuje zaobilaženje prepreka, zaustavljanje na rubovima, skakanje i slično.

Ovaj sustav se sastoji od idućih komponenti:

- NavMesh je struktura podataka koja opisuje površine gdje se može kretati, te omogućava pronalaženje puta iz jednog područja gdje je dopušteno kretati se u drugo. Ovaj podatak se automatski generira na osnovi geometrije levela.
- NavMesh Agent je komponenta koja pomaže pri stvaranju likova koji mogu izbjegavati jedan drugog tijekom kretanja prema cilju, kao i izbjegavanje bilo kojih drugih prepreka
- Off-Mesh Link je komponenta koja omogućava korištenje prečaca tijekom kretanja, a koji ne mogu biti prikazani kao površina gdje je omogućeno kretanje, kao što je skakanje preko ograde, ili otvaranje vrata itd.
- NavMesh Obstacle je komponenta kojom je moguće opisati prepreke koje bi lik trebao izbjegavati tijekom kretanja kroz svijet, a da se kreću prema liku, ili ako je riječ o statičnom objektu onda omogućava jednostavno pronalaženje drugog puta

Korištenje ovog sustava u projektu se radi tako da se doda prozor za navigaciju, potom se ispeče (engl. *Bake*), dakle napravi se mapa (NavMesh) bazirana na geometriji levela, potom se objektu za kojeg želite da se kreće po ovoj mapi doda komponenta NavMesh Agent te se konfiguriraju postavke vidljive na slici 33.



**Slika 33:** NavMesh mapa i agent

Plavom bojom je obojeno područje gdje se agent može kretati, a u postavkama se konfigurira sami agent, veličina, inteligencija, pronalaženje puta i brzina. Sve što je potrebno sada da se agent kreće je napisati logiku unutar skripte. U ispisu 7 dan je jednostavan primjer kako možemo koristiti komponentu NavMesh Agent.

```
using System.Collections;

public class MoveTo : MonoBehaviour {

    public Transform goal;

    void Start () {
        NavMeshAgent agent = GetComponent ();
        agent.destination = goal.position;
    }
}
```

**Ispis 7:** Korištenje navigacije

Za napraviti NavMesh Obstacle, odnosno prepreku, treba dodati istoimenu komponentu objektu kojeg je potrebno izbjjeći, potom je potrebno podesiti određene postavke, odabrati oblik te dodati Rigidbody komponentu i na kraju postaviti Carve kako bi agent mogao pronaći put pored prepreke. Jednako se postupa prilikom stvaranja Off Mesh Linka. Doda se pozicija gdje je omogućeno preskakanje ograde ili skakanje s visine, postavi se po-

zicija gdje završava ta radnja, objektu koji se nalazi na startnoj poziciji doda se istoimena komponenta te postave potrebne vrijednosti, odnosno startna i krajnja pozicija. Agent će tijekom kretanja iskoristiti tu mogućnost ako je to brži put prema cilju.

Kako bi se dobila što preciznija mapu kretanja potrebno je omogućiti računanje visinske mape unutar postavki za pečenje. Moguće je postaviti različite vrijednosti određenih pozicija mape, tako da ako agent nađe na površine s većom vrijednosti, pokušat će pronaći neki drugi put. Unity koristi A\* algoritam za računanje najkraćeg puta do cilja, a radi tako da koristi graf povezanih čvorova. Algoritam započinje s najbližim čvorom početku puta te posjećuje povezane čvorove sve dok ne dođe do cilja. Vrijednost, odnosno cijena kretanja između dva čvora ovisi o udaljenosti te cijeni vezanu za područje gdje se agent kreće. Ako je cijena npr. 2.0, tada će se ukupna vrijednost računati kao dvostruko duža od 1.0, tako da je moguće postaviti različite cijene na mapi ako želimo specificirati put kojim će se agent kretati.

Unity omogućava vrlo jednostavnu izradu korisničkog sučelja u igri, bilo to vezano za različite izbornike ili prikaz osnovnih informacija tijekom igranja . Glavna komponenta korisničkog sučelja (UI) je platno (engl. *Canvas*). To je prostor gdje bi se trebali nalaziti svi elementi grafičkog sučelja. To je objekt igre koji sadrži komponentu Canvas, a svi ostali elementi su djeca platna. Platno se unutar scene prikazuje kao pravokutnik te je najjednostavnije raditi s platnom tako da se postavi 2D prikaz unutar scene. Elementi se crtaju na platnu redoslijedom kako se pojavljuju u hijerarhiji, što znači da ako se dva elementa preklapaju, onaj kasnije naveden bit će nacrtan preko prijašnjeg.

Tri su načina učitavanja platna, a to su prevlačenje, kamera i prostor svijeta. Prevlačenje znači da ako je promijenjena rezolucija, promijenit će se i veličina platna. Kamera način znači da se učitava platno ovisno o tome koliko je ono udaljeno od određene kamere i o postavkama kamere. A treći način, prostor svijeta, je kada se platno ponaša kao bilo koji drugi objekt unutar scene, gdje moguće je ručno mijenjati veličinu svih elemenata, te će se učitavati ovisno o tome jesu li ispred ili iza nekog objekta u sceni.

Pozicija elementa je definirana preko sidrišta (engl. *Anchor*). Element se usidri na određenoj poziciji, npr. gore desno, te će uvijek, neovisno o veličini ekrana i rezoluciji, gravitirati toj poziciji. Tako smo sigurni da će, bez obzira na postavke igranja, element uvijek biti prisutan na očekivanoj poziciji. Kao i kod ostalih objekata, promjene nad roditeljem,

utječu i na djecu.

Neke od standardnih komponenti platna su:

- Tekst, kao što i samo ime sugerira, element koji sadrži određeni tekst.
- Slika, nije obična tekstura već (engl. *Sprite*), može služiti kao pomicna slika za prikaz stanja lika i slično.
- Standardna slika (engl. *RawImage*), prima tekstuру za prikaz.
- Botuni, koriste se za standardne radnje kao što su pokretanje različitih događaja, mijenjanje postavki itd.
- Klizači, kao i botuni, mogu se koristiti za promjene postavki, a u kombinaciji sa slikama služe za izradu stanja energije (engl. *Health bar*).

Svim navedenim elementima može se pristupiti u nadglednom prozoru, ali i preko skripti što znači da je po potrebi moguće implementirati logiku vezanu za pojedini element. To uključuje animacije, događaje koji se okidaju pritiskom na neki od botuna, prikazivanje podataka potrebnih igraču, glatke tranzicije između scena i slično. Na slici 34 prikazano je grafičko sučelje Dionyzus igre sa svim najvažnijim podacima potrebnim tijekom igranja.



**Slika 34:** Grafičko sučelje

## 2.7. Skripte

U ovom poglavlju će biti opisani glavni koncepti vezani za pisanje skripti koje je potrebno razumijeti kako bi se mogle implementirati željene funkcionalnosti u igri. Skripte su nužan dio svih igara. Čak i u najjednostavnijim igram na je potrebno napisati skripte koje će opisivati kako će igra reagirati prilikom unosa korisnika, kako i zašto će se dogoditi određeni događaji itd. Osim toga skripte se mogu koristiti za stvaranje grafičkih efekata, za kontroliranje ponašanje objekata kao i za implementiranje inteligencije likova u igri.

Ponašanje objekata igre kontroliraju komponente koje sadrže taj objekt. Iako je s ugrađenim komponentama moguće stvoriti raznovrsne elemente igre, vrlo brzo dolazi do potrebe stvaranja vlastite logike. Skripte se stvaraju tako da se unutar dodataka doda nova C# skripta te joj se dodjeli ime. Nakon stvaranja skripte automatski se generira kostur unutar skripte prikazan u dijelu kôda ispod 8.

```
using UnityEngine;
```

```

using System.Collections;

public class MainPlayer : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}

```

### Ispis 8: Nova skripta

Dakle svaka novostvorena skripta sadrži istoimenu klasu koja implementira klasu MonoBehaviour. MonoBehaviour je nacrt za stvaranje nove komponente koja se može dodavati na bilo koji objekt igre. Važno je da su naziv klase i datoteke jednaki kako bi se omogućila funkcionalnost te skripte.

Nadalje, generirane su dvije metode bez tijela. Unutar Update se piše kôd koji će upravljati ažuriranjem objekta igre kroz vrijeme. To može biti kretanje, okidanje različitih akcija ili reagiranje na korisnikov unos. Start metoda se poziva prije početka igre, odnosno prije prvog poziva Update metode, te je idealna za bilo kakvu vrstu inicijalizacije. Nije potrebno pisati konstruktoare jer se o tome brine Unity editor, a pisanjem vlastitih konstruktora može se uzrokovati poremećaj normalne funkcionalnosti Unityja. Kako bi aktivirali logiku napisanu unutar skripte potrebno ju je dodati na objekt gdje želimo da se primjeni napisana logika.

Kao i sve ostale komponente skripte mogu imati svojstva koje je moguće mijenjati unutar nadglednog prozora, što je prikazano na slici 35.



Slika 35: Prikaz skripte s javnom varijablom

Kôd 9 sadrži logiku skripte35. Dodana je javna varijabla myName te je unutar startne metode jednostavno ispisana koristeći Debug.Log metodu.

```
using UnityEngine;
using System.Collections;

public class MainPlayer : MonoBehaviour
{
    public string myName;

    // Use this for initialization
    void Start ()
    {
        Debug.Log("I am alive and my name is " + myName);
    }
}
```

### Ispis 9: Dodavanje javne varijable

Unity dopušta mijenjanje vrijednosti tijekom igranja, što je jako korisno za testiranje različitih efekata ili prilagođavanje određenih pozicija i slično. Ono što je moguće preko skripte, a ne i u editoru je mijenjanje vrijednosti tijekom vremena. Najčešći i najjednostavniji način pristupanja komponentama unutar kôda je prikazan je ispod 10.

```
void Start ()
{
    Rigidbody rb = GetComponent();

    // Add a force to the Rigidbody.
    rb.AddForce(Vector3.up * 10f);
}
```

### Ispis 10: Dodavanje javne varijable

Na ovaj način se može dohvatiti bilo koja komponenta koju sadrži objekt. Moguće je imati više od jedne skripte na jednom objektu. Ako je potrebno koristiti neki objekt igre unutar

skripte, potrebno je deklarirati objekt tipa `GameObject`, te mu onda pridružiti odgovarajuću vrijednost. To može biti javna varijabla kojoj će se dodijeliti odgovarajući objekt u samom editoru ili dohvaćanje objekta preko imena ili taga.

Ono što je karakteristično za skripte je što Unity daje kontrolu skripti tako što poziva funkcije koje su deklarirane u samoj skripti, a u trenutku kada funkcija završava, kontrola se predaje nazad Unityju. Neke od tih funkcija su `Start` i `Update`, to su tzv. funkcije događaja, pošto se aktiviraju kao odgovor prilikom određenog događaja tijekom igranja. `Update` se poziva prije učitavanja trenutka i prije računanja animacije dok se `FixedUpdate` poziva prije ažuriranje fizike. Pošto se fizika i trenutak vremena ne ažuriraju istom frekvencijom, precizniji se rezultati dobivaju ako se kôd vezan za fiziku napiše unutar `FixedUpdate` metode. Ako je potrebno napraviti dodatne promjene nakon `Update` ili `FixedUpdate` funkcija tada je najbolje koristiti `LateUpdate` metodu.

Osim navedenih postoji i `Awake` metoda koja se poziva u trenutku učitavanja scene, što znači da će sve `Awake` metode biti pozvane prije prve `Start` metode. Tada je moguće u `Start` metodi napraviti logiku koja će koristiti neku od inicijalizacija napravljenih u `Awake`.

Osim standardnih metoda za upravljanje događajima, postoje i one za upravljanje grafičkim sučeljem. Neke od metoda su `OnGUI`, a one što su korištene i u samom projektu su `OnMouseOver` i `OnMouseDown`. Ovim metodama možemo kontrolirati što će se dogoditi kada je pokazivač miša usmjeren prema određenom objektu, npr. za okidanje događaja koji će prikazati tekstualni sadržaj ili omogućiti pritisak nekog botuna kako bi se pokrenuo neki događaj.

Kada je riječ o pokretanju objekata i upravljanju vremena, važno je voditi računa o tome da frekvencija učitavanja nije uvijek ista. Dakle nije moguće postaviti fiksne vrijednosti koliko prostora će neki lik proći u određenom periodu vremena, npr. metara u sekundi, već je potrebno uzeti u obzir i frekvenciju. Dakle da bi postigli željene rezultate potrebno je koristiti svojstvo `Time.deltaTime`.

U kratkom primjeru 11 prikazano je kako se koristi ovo svojstvo, odnosno koji je ispravan pristup kada je riječ o upravljanju objekata tijekom vremena u igri.

```
using UnityEngine;
```

```

using System.Collections;

public class ExampleScript : MonoBehaviour {
    public float distancePerSecond;

    void Update() {
        transform.Translate(0, 0, distancePerSecond * Time.deltaTime);
    }
}

```

### Ispis 11: Upravljanje vremenom

Moguće je mijenjati fiksni pomak vremena `Time.fixedDeltaTime`. Ako je zadan veliki posao za stroj fizike, onda je moguće smanjiti početnu vrijednost, ali to zahtjeva veći rad procesora. Moguće je povećati vrijednost, pritom tijela koja su pod utjecajem fizike neće savršeno reagirati na promjene, ali to nije toliko primjetno te je prihvatljivo s obzirom na to da to može poboljšati performanse igre.

Postoji još jedan način upravljanja vremenom a to je korištenje `Time.timeScale` svojstva. Tim svojstvom mijenjamo koliko brzo ide vrijeme unutar igre s obzirom na stvarno vrijeme. Ovime je moguće ubrzati ili usporiti kretnje objekata u igri, npr. moguće je napraviti sporije animacije (engl. *Slow motion*), ubrzati neke radnje tijekom kratkih filmova te isto tako kompletno pauzirati igru.

Navedene opcije mogu se mijenjati i globalno unutar prozora Time, odnosno vrijeme, ali je puno bolji pristup upravljanje ovim svojstvima unutar skripti. Primjer kôda 12

```

using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    void Pause() {
        Time.timeScale = 0;
    }

    void Resume() {
        Time.timeScale = 1;
    }
}

```

```
    }  
}
```

### Ispis 12: Upravljanje vremenom

Upravljanje objektima često je vezano za njihovo pojavljivanje i brisanje iz scene, za to je moguće koristiti metode kao što su Instantiate za stvaranje objekta u određenim situacijama, npr. metak. Suprotno tome je metoda Destroy, koja kao što joj i naziv govori uništava objekt, odnosno briše ga kompletno iz scene. Tako se moguće riješiti nepotrebnih objekata, te optimizirati igru. Ako je potrebno samo trenutno onesposobiti neki objekt ili aktivirati samo u određenim situacijama, tada je moguće koristiti metodu setActive koja prima boolean true ili false.

Za određene efekte, kao što su zacrnjivanje ekrana ili jednostavno tok teksta koji se prikazuje na ekranu, standardne metode nisu dovoljno dobre. Svaka metoda se odradi unutar jednog trenutka vremena te je nemoguće prikazati promjene trenutak po trenutak. U tom slučaju potrebno je koristiti metode koje mogu odraditi dio skripte, potom dati kontrolu Unityju, te nakon određenog perioda opet odraditi nastavak skripte. Takve metode su (engl. *Coroutine*), a nalaze se u Ienumerator sučelju. Primjer korištenja je u skripti ispod 13.

```
public int waitTimeForCredits = 3;  
public float waitTimeForCameraSwitch = 5f;  
public float waitForThirdCamera = 3f;  
  
// Use this for initialization  
void Start () {  
    StartCoroutine(CameraSwitcher());  
}  
  
IEnumerator CameraSwitcher()  
{  
    yield return new WaitForSeconds(waitTimeForCredits);  
    credLeadDesigner.SetActive(true);  
  
    yield return new WaitForSeconds(waitTimeForCredits);
```

```

        credStory.SetActive(true);

        yield return new WaitForSeconds(waitForThirdCamera);
        thirdCamera.SetActive(true);
        firstCamera.SetActive(false);

        yield return new WaitForSeconds(2);
        credEngine.SetActive(true);

        yield return new WaitForSeconds(waitTimeForCameraSwitch);
        secondCamera.SetActive(true);
        thirdCamera.SetActive(false);
        credGameName.SetActive(true);

        yield return new WaitForSeconds(6);
        secondCamera.SetActive(false);
        fourthCamera.SetActive(true);

        yield return new WaitForSeconds(9.5f);
        fourthCamera.SetActive(false);
        fifthCamera.SetActive(true);
    }
}

```

### Ispis 13: Upravljanje vremenom

Korištenjem metode `WaitForSeconds` odgadamo vrijeme te dajemo kontrolu Unityju, a nakon što završi taj period vremena, opet se izvršava kôd u skripti. Kao što je vidljivo u skripti ovo je korisno ako želimo uključiti ili isključiti određene objekte u sceni, te za stvaranje različitih efekata. Osim toga korištenjem `IEnumerator` moguće je različite zahtjevne zadatke rasporediti te tako poboljšati performanse igre.

Vektorska aritmetika je osnova za 3D grafiku, fiziku i animaciju, Unity ima implementirane sve najvažnije funkcionalnosti za rad s vektorima, a to su zbrajanje, oduzimanje, množenje i normalizacija vektora i slično. Tako se oduzimanje vektora može koristiti kada je potrebno izračunati udaljenost između dva objekta, a za određivanje smjera se koristi normalizacija vektora. Još jedan od primjera korištenja vektora je prilikom mjerjenja brzine

automobila. To se uglavnom radi tako da se mjeri brzina rotiranja kola. Međutim, ako automobil ne ide pravo, odnosno klizi sa strane tada se na ovaj način ne može dobiti točan rezultat brzine. Taj problem se rješava metodom za računanje skalarnog produkta vektora. Ta operacija je vrlo jednostavna što znači da ne zahtijeva mnogo procesorske moći te je dobar izbor kada je potrebno konstantno ažurirati rezultat kao u ovom slučaju.

Za traženje grešaka moguće je postaviti nadgledni prozor u Debug način rada, a isto tako Visual Studio ima ugrađeni program za pronalaženje grešaka (engl. *Debugger*). Osim toga Unity ima svoj konzolni prozor gdje se ispisuju pogreške kao i upozorenja, te je isto tako moguće ispisivati poruke unutar konzole koristeći `Debug.Log` funkcionalnost.

Više o samom pisanju skripti bit će opisano u dijelu opisivanja samog projekta, gdje će se prikazati pristup prilikom rješavanja konkretnih problema tijekom izrade igre.

Ovim završava opisivanje općenitog rada u Unityju te će u nastavku biti opisan sami projekt.

### 3. Dionyzus

U ovom poglavlju je opisana igra, odnosno kompletan projekt, najvažniji koraci prema izradi, te su objašnjeni neki ključni koncepti koji su korišteni tijekom izrade igre.

#### 3.1. Osnovne skripte za upravljanje s likom

Glavna skripta koja je zadužena za stvaranje, odnosno inicijalizaciju svih potrebnih komponenti unutar igre je `GameManager`. To je skripta koja ne implementira `Monobehaviour` klasu, što znači da je nije potrebno pridružiti niti jednom objektu. `GameManager` je zadužen za inicijalizaciju komponenti kao što su `InputController`, odnosno skripta koje implementira logiku unosa igrača, `Timer`, koji je korišten za potrebe stvaranja vremenskog perioda koji je potreban da se neka radnja izvrši, `Respawner`, koji je korišten za ponovno instanciranje objekata unutar scene ako je to potrebno te `EventBus` koji implementira logiku različitih događaja koji se okidaju tijekom igranja. Jedan od tih eventa je i instanciranje glavnog igrača. Dio te skripte je prikazan u ispisu 14.

```
public event System.Action<Player> OnLocalPlayerJoined;  
private GameObject gameObject;  
  
public bool IsPaused { get; set; }  
  
private static GameManager m_Instance;  
public static GameManager Instance  
{  
    get  
    {  
        if (m_Instance == null)  
        {  
            m_Instance = new GameManager();  
            m_Instance.gameObject = new GameObject("_gameManager");  
            m_Instance.gameObject.AddComponent<InputController>();  
            m_Instance.gameObject.AddComponent<Timer>();  
            m_Instance.gameObject.AddComponent<Respawner>();  
        }  
        return m_Instance;  
    }  
}
```

```

    }

    private InputController m_InputController;
    public InputController InputController
    {
        get
        {
            if (m_InputController == null)
            {
                m_InputController = gameObject.GetComponent<InputController>();
            }
            return m_InputController;
        }
    }
}

```

#### Ispis 14: Menadžer igre

Kao što je već navedeno Dionyzus je 3D igra, što znači da su sav okoliš, glavni lik te ostali likovi koji se pojavljuju u igri u tri dimenzije. To zahtjeva rad s 3D modelima te je potrebno realizirati njihovu mehaniku. Igranje igre također je iz trećeg lica što znači da je potrebno koristiti vjerodostojne animacije glavnog lika kako bi se uspješno dočarale sve radnje tijekom igranja.

S tim rečenim, prvi zadatak je realizirati kretanje unutar svijeta igre. Kretanje su omogućene korištenjem komponente kontroler lika (engl. *CharacterController*). Ta komponenta koristi fiziku te omogućava kretanje djelovanjem sile na objekt. Prije prikaza skripte kratko će biti opisane sve mogućnosti likavezane za kretanje.

Implementirano je šetanje, trčanje i kretanje u sagnutom položaju. Početno stanje je mirovanje, pritiskom odgovarajućih tipki na tipkovnici prelazi se u neko od prethodno navedenih stanja. Svaka od ovih mogućnosti je pokrivena odgovarajućim animacijama. Sve animacije korištene u igri su preuzete sa stranice Mixamo <https://www.mixamo.com>. Potrebno je registrirati se, učitati lika na kojem želite da se odradi prilagođavanje animacija, odabiranje odgovarajuće animacije te spremanje lika unutar projekta.

Sve najvažnije komponente koje su potrebne kako bi lik pravilno funkcionirao nalaze se u skripti `Player`. Ispis 15 prikazuje `Move` i `LookAround` metode u kojima je

napisana potrebna logika kako bi se prethodno navedene funkcionalnosti omogućile. Ono što se može istaknuti je provjera je li lik na zemlji, to je potrebno kako bi se omogućilo skakanje lika.

```
void LookAround()
{
    mouseInput.x = Mathf.Lerp(mouseInput.x, playerInput.MouseInput.x, 1
        f /
    MouseControl.Damping.x);

    mouseInput.y = Mathf.Lerp(mouseInput.y, playerInput.MouseInput.y, 1
        f /
    MouseControl.Damping.y);

    transform.Rotate(Vector3.up * mouseInput.x * MouseControl.
        Sensitivity.x);

    playerAim.SetRotation(mouseInput.y * MouseControl.Sensitivity.y);
}

void Move()
{
    float moveSpeed = settings.WalkSpeed;
    if (playerInput.IsRunning)
    {
        moveSpeed = settings.RunSpeed;
    }
    if (playerInput.IsSneaking)
    {
        moveSpeed = settings.SneakSpeed;
    }
    if (playerInput.IsCrouched)
    {
        moveSpeed = settings.CrouchSpeed;
    }
    if (PlayerState.MoveState == PlayerState.EMoveState.COVER)
    {
        moveSpeed = settings.WalkSpeed;
    }
}
```

```

    }

    if (!MoveController.isGrounded)
    {

        PlayerState.IsFalling = true;

    }

    if (MoveController.isGrounded)
    {

        // We are grounded, so recalculate
        // move direction directly from axes
        PlayerState.IsFalling = false;
        moveDirection = new Vector3(playerInput.Horizontal, 0.0f,
playerInput.Vertical);
        moveDirection = transform.TransformDirection(moveDirection);
        moveDirection = moveDirection * moveSpeed;

        if (playerInput.Jump)
        {
            moveDirection.y = jumpSpeed;
        }

    }

    if (Vector3.Distance(moveDirection * Time.deltaTime,
        previousPosition) >
minimumMoveThreshold)
    {

        footSteps.Play();
    }

    // Apply gravity
    moveDirection.y = moveDirection.y - (gravity * Time.deltaTime);

    // Move the controller
    MoveController.Move(moveDirection * Time.deltaTime);
}
}

```

### Ispis 15: LookAround i Move metode

Osim omogućavanja kretnji potrebno je i zabilježiti iste te znati gdje se lik kreće. Za to nam je naravno potrebna kamera. ThirdPersonCamera skripta implementira logiku kamere. Unutar skripte traži se objekt AimingPivot koji se nalazi na glavnom liku, omogućeno je mijenjanje udaljenosti s koje će se promatrati lik, brzina kojom će kamera pratiti kretnje lika, te isto tako promjena odstojanja ovisno o tome je li lik stoji ili je sagnut. Osim toga kako bi se spriječila pogreška prolaska vidnog polja kroz zidove, dodana je metoda gdje se provjerava sudara li se vidno polje kamere, odnosno linija koju odašilje kamera (engl. *Raycast*) s nekim od čvrstih objekata, u slučaju da je, sprječava se daljnji pomak kamere te se rotira u suprotnom smjeru. Sva logika kamere je napisana unutar LateUpdate metode (ispis 16).

```

CameraRig cameraRig = defaultCamera;

if (localPlayer.PlayerState.WeaponState == PlayerState.EWeaponState
    .AIMING
    || localPlayer.PlayerState.WeaponState ==
    PlayerState.EWeaponState.AIMEDFIRING)
{
    cameraRig = aimCamera;
}

float targetHeight = cameraRig.CameraOffset.y +
(localPlayer.PlayerState.MoveState == PlayerState.EMoveState.CROUCHING ?
cameraRig.CrouchHeight : 0);
Vector3 targetPosition = cameraLookTarget.position +
localPlayer.transform.forward * cameraRig.CameraOffset.z +
    localPlayer.transform.up * targetHeight
    + localPlayer.transform.right * cameraRig.CameraOffset.x;
Quaternion targetRotation = cameraLookTarget.rotation;

Vector3 collisionTargetPoint = cameraLookTarget.position +
localPlayer.transform.up * targetHeight - localPlayer.transform.forward *
10f;
HandleCameraCollision(collisionTargetPoint, ref targetPosition);

transform.position = Vector3.Lerp(transform.position,

```

```

        targetPosition,
cameraRig.Damping * Time.deltaTime);

    transform.rotation = Quaternion.Lerp(transform.rotation,
cameraLookTarget.rotation, cameraRig.Damping * Time.deltaTime);

}

private void HandleCameraCollision(Vector3 toTarget, ref Vector3
fromTarget)
{
    RaycastHit hit;
    if (Physics.Linecast(toTarget, fromTarget, out hit))
    {
        Vector3 hitPoint = new Vector3(hit.point.x + hit.normal.x * .2f,
hit.point.y, hit.point.z + hit.normal.z * .2f);
        fromTarget = new Vector3(hitPoint.x, fromTarget.y, hitPoint.z);
    }
}

```

### Ispis 16: Logika kamere

Ono što je vidljivo u obje skripte je `PlayerState`. To je skripta koja sadrži sva moguća stanja lika tijekom igranja. Stanja su podijeljena na dva glavna, a to su naoružan (engl. *Armed*) i nenaoružan (engl. *Unarmed*). Ta stanja su realizirana preko `Enum` tipa podatka. Kôd 17 prikazuje sva moguća stanja lika.

```

public class PlayerState : MonoBehaviour
{
    public enum EMoveState
    {
        WALKING,
        RUNNING,
        CROUCHING,
        SNEAKING,
        COVER,
        FALLING
    }
}

```

```

public enum EWeaponState
{
    IDLE,
    FIRING,
    AIMING,
    AIMEDFIRING
}

```

### Ispis 17: Stanja lika

Ove funkcionalnosti zaokružujemo s dvije vrlo važne skripte, a to su `InputController` i `PlayerAnimation`. Skripte su međusobno povezane tako da igra osluškuje koje tipke korisnik pritisne, a potom se aktiviraju boolean vrijednosti, dakle istina ili laž, koje svojom aktivacijom utječu na logiku napisanu unutar skritpe `PlayerAnimation`. Rad unutar prozora kontroler animacije je već prethodno opisan, ali, za napomenu, te su boolean vrijednosti potrebne kako bi se omogućio prijelaz iz jednog stanja u drugo stanje lika. Skripta `InputController` nalazi se u ispisu 18.

```

void Update()
{
    if (EnterCar.InCar)
    {
        exitCar = Input.GetKey(KeyCode.L);
    }
    if (EnterCar.CanEnter)
    {
        enterCar = Input.GetKeyDown(KeyCode.Return);
    }
    Action = Input.GetKeyDown(KeyCode.E);
    Escape = Input.GetKey(KeyCode.Escape);
    Vertical = Input.GetAxis("Vertical");
    Horizontal = Input.GetAxis("Horizontal");
    MouseInput = new Vector2(Input.GetAxisRaw("Mouse X"),
    Input.GetAxisRaw("Mouse Y"));
    Fire1 = Input.GetButton("Fire1");
    Reload = Input.GetKeyDown(KeyCode.R);
}

```

```

IsRunning = Input.GetKey(KeyCode.LeftShift);
IsCrouched = Input.GetKey(KeyCode.C);
IsSneaking = Input.GetKey(KeyCode.V);
IsAiming = Input.GetButton("Fire2");
CoverToggle = Input.GetKeyDown(KeyCode.F);
Jump = Input.GetKeyDown(KeyCode.Space);

if (IsAiming)
{
    Vertical = 0;
    Horizontal = 0;
}

MouseWheelUp = Input.GetAxis("Mouse ScrollWheel") > 0;
MouseWheelDown = Input.GetAxis("Mouse ScrollWheel") < 0;
}

```

### Ispis 18: Kontroler unosa korisnika

Sva logika se odvija unutar `Update` metode gdje se konstantno osluškuje unos korisnika i, ako je odgovarajuća tipka pritisnuta, mijenja se status pripadajuće varijable. Skripta 19 opisuje funkcionalnost animacija lika.

```

void Update()
{
    if (GameManager.Instance.IsPaused)
    {
        return;
    }

    animator.SetFloat("Vertical", GameManager.Instance.InputController.
        Vertical);

    animator.SetFloat("Horizontal",
        GameManager.Instance.InputController.Horizontal);

    animator.SetBool("IsFalling",
        GameManager.Instance.LocalPlayer.PlayerState.IsFalling);

    animator.SetBool("EnterCar", GameManager.Instance.InputController.
        enterCar);
}

```

```

        animator.SetBool("ExitCar", GameManager.Instance.InputController.
            exitCar);

        animator.SetBool("IsReloading", GameManager.Instance.InputController
            .Reload);
        animator.SetBool("WeaponSwitch",
GameManager.Instance.InputController.MouseWheelDown ||

GameManager.Instance.InputController.MouseWheelUp);
        animator.SetBool("IsJump", GameManager.Instance.InputController.
            Jump);
        animator.SetBool("IsRunning", GameManager.Instance.InputController.
            IsRunning);
        animator.SetBool("IsSneaking",
GameManager.Instance.InputController.IsSneaking);
        animator.SetBool("IsCrouched",
GameManager.Instance.InputController.IsCrouched);
        animator.SetBool("IsFiring", GameManager.Instance.InputController.
            Fire1);
        animator.SetBool("IsAiming",
GameManager.Instance.LocalPlayer.PlayerState.WeaponState ==
PlayerState.EWeaponState.AIMING ||
GameManager.Instance.LocalPlayer.PlayerState.WeaponState ==
PlayerState.EWeaponState.AIMEDFIRING);
        animator.SetFloat("AimAngle", playerAim.GetAngle());

        animator.SetBool("IsInCover",
GameManager.Instance.LocalPlayer.PlayerState.MoveState ==
PlayerState.EMoveState.COVER);
    }
}

```

### Ispis 19: Animacije lika

SetBool je jednostavna metoda, gdje se na temelju boolean vrijednosti mijenja odgovara-juća, odnosno istoimena varijabla unutar kontrolera animacija. Svim varijablama se pristupa preko instance GameManagera. Ovim skriptama je omogućeno instanciranje igre, glavnog lika te glavnih komponenti koje su potrebne kako bi igrač imao kontrolu nad likom te bio u mogućnosti da događa nešto unutar igre.

### 3.2. Energija i sistem oružja

U ovom poglavlju nastavlja se opisivanje funkcionalnosti igre. Opisan je sistem oružja te mogućnost lika da umre unutar igre. Kada bi se lik mogao nesmetano kretati kroz svijet, odnosno bez opasnosti po vlastiti život, to i ne bi bila zanimljiva igra otvorenog svijeta. Taj problem je riješen na standardan način da glavni lik kao i neprijatelji imaju energiju (engl. *Health points*), ali isto tako i bilo koji drugi objekt za koji je omogućeno da bude uništen. Glavna funkcionalnost je opisana unutar skripte `Destructable`, gdje se objektu koji sadrži ovu skriptu dodjeljuje energija, metode koje omogućuju oduzimanje energije, te mogućnost umiranja ako je vrijednost energije jednaka nuli. Ranjavanje i umiranje je riješeno korištenjem događaja (engl. *Event*) koji su sastavni dio C# jezika. Kod 20 prikazuje deklaraciju događaja te glavne metode koje omogućuju prethodno opisane funkcionalnosti. Osim toga, dodan je i prikaz energije, što će detaljnije biti opisano u dijelu grafičkog sučelja igre. Sve što je potom potrebno napraviti je naslijediti ovu skriptu s pojedinom skriptom koja će pripadati različitim likovima unutar igre kako bi se postigle varijacije ako to igra zahtjeva.

```
[SerializeField] float hitPoints;
[SerializeField] Image healthBar;

public event System.Action OnDeath;
public event System.Action OnDamageReceived;

float damageTaken;

public virtual void Die()
{
    if (IsAlive)
        return;

    if (OnDeath != null)
    {
        OnDeath();
    }
}

public virtual void TakeDamage(float amount)
{
```

Skripta	Opis
Shooter	Sadrži logiku pucanja.
WeaponController	Sadrži logiku opskrbljivanjem s oružjem, odnosno korišteњem opreme.
WeaponReloader	Sadrži logiku ponovnog punjenja oružja.
WeaponRecoil	Sadrži logiku uzorka pucanja.
Projectile	Sadrži logiku instanciranja odgovarajućeg projektila te efekta istog, kao i provjeravanje mjesta sudara.
PlayerAim	Sadrži logiku ciljanja oružjem.
Crosshair	Vizualno prikazuje nišan.

**Tablica 2:** Sistem oružja

```

if (!IsAlive)
{
    return;
}

if (amount > 0)
{
    RandomHitAnimation();
}

damageTaken += amount;

if (OnDamageReceived != null)
{
    OnDamageReceived();
}

if (HitPointsRemaining <= 0)
{
    Die();
}
}
}

```

### Ispis 20: Energija lika

Kao što je vidljivo metode su virtualne, dakle moguće je pregaziti funkcionalnost istih.

Sistem oružja je izведен preko nekoliko skripti koje su prikazane u tablici 2.

Shooter omogućava pucanje. Dodavanjem ove skripte na objekt, potrebno je postaviti vrijednost frekvencije pucanja, dodati odgovarajući projektil kojeg će instancirati određeno oružje, te je potrebno da objekt sadrži dijete Muzzle. To je potrebno kako bi

se omogućili efekti pucanja s oružjem te da bi se postavila ogledna točka iz koje će se usmjeravati projektil. Glavna metoda unutar skripte je virtualna metoda `Fire`, koja se može mijenjati ovisno o potrebi različitog oružja. Metoda se oslanja na broj metaka u spremniku i frekvenciju pucanja. U slučaju da je spremnik prazan ili nije prošao period frekvencije, onemogućeno je pucanje. U slučaju da je moguće pucati, instancira se projektil na lokaciji već spomenutog objekta `Muzzle`. Projektil se kreće ovisno o tome gdje je usmjeren model oružja određenom brzinom. Put projektila, odnosno ono što predstavlja smjer projektila i što provjerava s čime se sudara je zraka (engl. *Ray*) koja je spomenuta i kod kamere. Osim toga, prilikom pucanja se aktiviraju efekti kao što su zvuk i svjetlost kako bi pucanje bilo što stvarnije. Kod 21 prikazuje opisanu metodu.

```
public virtual void Fire()
{
    light.SetActive(false);
    canFire = false;
    if (Time.time < nextFireAllowed || ShooterBlock.blockShooter ==
        true)
        return;
    if (reloader != null)
    {
        if (reloader.IsReloading)
        {
            return;
        }

        if (reloader.RoundsRemainingInClip == 0)
        {
            emptyClip.Play();
            return;
        }
        reloader.TakeFromClip(1);
    }
    nextFireAllowed = Time.time + rateOfFire;
    bool isLocalPlayerControlled = AimTarget == null;
    if (!isLocalPlayerControlled)
    {
```

```

        muzzle.LookAt(AimTarget.position + AimTargetOffset);
    }

    Projectile newBullet = (Projectile) Instantiate(projectile, muzzle.
        position,
    muzzle.rotation);

    if (isLocalPlayerControlled)
    {
        Ray ray = Camera.main.ViewportPointToRay(new Vector3(.5f, .5f,
            0));

        RaycastHit hit;
        Vector3 targetPosition = ray.GetPoint(500);

        if (Physics.Raycast(ray, out hit))
        {
            targetPosition = hit.point;

        }
        newBullet.transform.LookAt(targetPosition + AimTargetOffset);
    }
    if(this.WeaponRecoil)
    {
        this.WeaponRecoil.Activate();
    }
    FireEffect();
    audioFire.Play();
    canFire = true;
}

```

### Ispis 21: Skripta pucanja

Sljedeća skripta je `WeaponController` u kojoj je opisana logika korištenja oružja koje je dostupno igraču ako ga posjeduje, te promjena oružja. Unutar skripte su deklarirana dva niza tipa `Shooter`, dakle objekti koji sadrže skriptu `Shooter`. Kada započne igra, provjeravaju se djeca objekta glavnog lika te se oružja razvrstavaju ovisno o tome posjeduje li ih trenutno igrač ili ih tek treba prikupiti. To je riješeno tako da se deak-

tiviraju sva oružja koja trenutno nisu dopuštena za korištenje, koja igrač ne posjeduje te će se aktivirati unutar niza omogućenih oružja u trenutku kada ih prikupi. Osim toga skripta sadrži metode koje omogućavaju aktiviranje oružja kojeg igrač posjeduje, metoda Equip te SwitchWeapon koja omogućuje promjenu oružja. Ako je moguće korisnik prilikom rotiranja kotačića miša mijenja oružje te mu je omogućeno pucanje iz tog oružja, a osim toga aktivira se i odgovarajuća slika koja predstavlja to oružje. Kod 22 prikazuje dio navedenih funkcionalnosti.

```
internal void SwitchWeapon(int direction)
{
    CanFire = false;
    currentWeaponIndex += direction;

    if (currentWeaponIndex > weapons.Length - 1)
    {
        currentWeaponIndex = 0;
    }
    if (currentWeaponIndex < 0)
    {
        currentWeaponIndex = weapons.Length - 1;
    }
    GameManager.Instance.Timer.Add(() =>
    {
        Equip(currentWeaponIndex);
    }, weaponSwitchTime);
}

internal void Equip(int indexOfWeapon)
{
    DeactivateWeapons();
    CanFire = true;

    indexOfWeapon = CheckInactive(indexOfWeapon);

    m_ActiveWeapon = weapons[indexOfWeapon];
    m_ActiveWeapon.Equip();
```

```

        weapons [indexOfWeapon] .gameObject.SetActive (true) ;
        weapons [indexOfWeapon] .GetComponent () .WeaponImage.GetComponent () .
            enabled =
        true;
        weapons [indexOfWeapon] .GetComponent () .BulletImage.GetComponent () .
            enabled =
        true;
        weaponSwitch.Play ();
        if (OnWeaponSwitch != null)
        {
            OnWeaponSwitch (m_ActiveWeapon);
        }
    }
}

```

### Ispis 22: Kontroler oružja

WeaponReloader je jednostavna skripta koja omogućava ponovno punjenje oružja pritiskom odgovarajuće tipke. Osim toga osluškuje promjenu broja metaka u spremniku, dakle prilikom pucanja taj broj se smanjuje, a nakon punjenja povećava. Kako pucanje ne bi bilo previše jednostavno, koristi se skripta WeaponRecoil, to je skripta koja sprječava igrača da puca u istom pravcu ako konstantno puca, dakle ovisno o parametrima mijenja smjer u kojem će se projektil kretati, tzv. trzaj oružja. Taj smjer nije kompletno nasumičan već je određen uzorkom koji je definiran za pojedino oružje. Način implementacije nalazi se u ispisu 23.

```

void Update()
{
    if (nextRecoilCooldown > Time.time)
    {
        recoilActiveTime += Time.deltaTime;
        float percentage = GetPercentage();

        Vector3 recoilAmount = Vector3.zero;
        for (int i = 0; i < layers.Length; i++)
        {
            recoilAmount += layers [i].direction *
            layers [i].curve.Evaluate (percentage);
    }
}

```

```

        }

        this.Shooter.AimTargetOffset = Vector3.Lerp(Shooter.
            AimTargetOffset,
        Shooter.AimTargetOffset + recoilAmount, strength * Time.deltaTime);
        this.Crosshair.ApplyScale(percentage * Random.Range(strength *
            7,
        strength * 9));
    }

    else
    {
        recoilActiveTime -= Time.deltaTime;
        if (recoilActiveTime < 0)
        {
            recoilActiveTime = 0;
        }
        this.Crosshair.ApplyScale(GetPercentage());
        if (recoilActiveTime == 0)
        {
            this.Shooter.AimTargetOffset = Vector3.zero;
            this.Crosshair.ApplyScale(0);
        }
    }
}

float GetPercentage()
{
    float percentage = recoilActiveTime / recoilSpeed;
    return Mathf.Clamp01(percentage);
}

```

### Ispis 23: Trzaj oružja tijekom pucanja

Pucanjem se instancira projektil, odnosno metak te ovisno o tome s čim se sudari, može napraviti štetu, odnosno smanjiti energiju mete ili je uništiti. Glavna logika je implementirana unutar metode `CheckDestructable` koja provjerava vrstu pogodene mete, u slučaju da se radi o ljudskoj meti poziva se metoda `TakeDamage`. Uz to je implementiran i efekt koji nastaje u trenutku pogotka, to opet ovisi o vrsti mete. U slučaju ljudske mete,

Skripta	Opis
EnemyState	Stanja neprijatelja.
EnemyPlayer	Glavna skripta koja upravlja svim ostalim komponentama.
EnemyPatrol	Skripta koja iskorištava PathFinder i Scanner komponente.
EnemyShoot	Kontrolira pucanje neprijatelja.
EnemyHealth	Energija neprijatelja.
EnemyAnimation	Upravlja animacijama neprijatelja ovisno o stanju.

**Tablica 3:** Skripte koje implementiraju neprijatelja

pokrene se odgovarajuća animacije te se instancira krv, a u drugim slučajevima trag udara metka. PlayerAim prati pokrete miša kako bi bilo omogućeno pucanje u odgovarajućem smjeru, Crosshair skripta iscrtava nišan na odgovarajućoj poziciji.

### 3.3. Funkcionalnosti neprijatelja

U ovom dijelu su opisane sve funkcionalnosti koje su dodijeljene neprijatelju kao što su kretanje, pretraživanje, sukob s igračem i slično.

Skripte koje opisuju neprijatelja su navedene i kratko opisane u tablici 3. Najvažnija skripta neprijatelja je `EnemyPlayer`. Osim inicijalizacije svih ostalih komponenti, ona sadrži i logiku kako će se lik ponašati u različitim situacijama. Kako se radi o neprijateljima, početno stanje svakog je patroliranje, a nakon što detektira igrača napada ga. Stanja neprijatelja su opisana u skripti `EnemyState` gdje su moguća dva stanja `Aware` i `Unaware`, stanje kada je neprijatelj svjestan prisutnosti glavnog igrača i stanje kada nije. Navedene funkcionalnosti riješene su preko događaja koja su već prije opisana, osluškuju se stanja te ovisno o događaju izvršava se logika metoda. Metode su vrlo jednostavne, pretraživanje područja te prepoznavanje glavnog igrača koji je definiran vlastitim tagom. Kod 24 prikazuje način kako se automatski sve potrebne skripte pridružuju objektu kada se doda `EnemyPlayer` skripta.

```
[RequireComponent(typeof(PathFinder)) ]
[RequireComponent(typeof(EnemyHealth)) ]
[RequireComponent(typeof(EnemyState)) ]
public class EnemyPlayer : MonoBehaviour
{
    [SerializeField] public Scanner playerScanner;
    [SerializeField] Soldier settings;
```

```
PathFinder pathFinder;
```

### Ispis 24: Automatsko pridruživanje ovisnih skripti

Ključna riječ je `RequireComponent`, prima tip komponente koja će se automatski pridružiti objektu, te je potrebno inicijalizirati unutar editora sve što te skripte očekuju. To može biti pridruživanje nekog drugog objekta, neke variable i slično. Osim toga, prikazana je inicijalizacija dvije dodatne komponente, `Scanner` i `PathFinder`. `Scanner`, u prijevodu pretraživanje, je skripta koja sadrži logiku na koji će način neprijatelj pretraživati područje. Odredi se radius pretraživanja, brzina te maska (engl. *LayerMask*). Maska označava koje će područje biti uključeno u skeniranje. Objektu se u nadglednom prozoru doda tag maske te je s tim određeno ono što neprijatelj treba detektirati. Uz to se provjerava nalazi li se prepreka ispred neprijatelja, odnosno smanjuje li išta vidno polje. Ovdje je za primjer korištena i mogućnost ekstenzija, a to je vrlo koristan način na koji se mogu proširiti funkcionalnosti postojećih klasa. Metode trebaju biti static, dakle vrijede za cijelu klasu te se pozivaju bez inicijalizacije objekta. Prvi parametar metode mora biti `this` te potom naziv Klase koja se proširuje. Ta je mogućnost korištena za Unity tip `Transform`, a metoda je `IsInLineOfSight` koja provjerava nalazi li se tražena meta unutar vidnog pogleda. Na ovaj način je vrlo jednostavno ponovno iskoristiti ovu metodu u bilo kojem drugom projektu.

`PathFinder` skripta koristi Unity komponentu `NavMeshAgent` te jednostavno omogućava objektu koji sadrži ovu skriptu da se kreće unutar svijeta. Tu se nalazi i `Soldier` skriptirani objekt koji je već prethodno opisan, ali kao podsjetnik, to je skripta koja sadrži informacije, postavke igre i slično.

Unutar ove skripte, ali i unutar kompletnog projekta često je vidljiv atribut `[SerializeField]` Unity serijalizira samo polja koja su deklarirana kao javna, ali ako želite serijalizirati i privatna polja potrebno je dodati navedeni atribut. Serijalizacija je proces transformiranja podataka u format koji Unity može spremiti te rekonstruirati kada je to potrebno. Kod 25 prikazuje dio `Scanner` skripte, gdje je prikazan način na koji se koristi ekstenzija za `Transform`, a 26 skriptu gdje je prikazan način na koji se proširuju klase.

```
void OnDrawGizmos ()  
{
```

```

        Gizmos.color = Color.green;
        Gizmos.DrawLine(transform.position, transform.position +
GetViewAngle(fieldOfView / 2) * GetComponent().radius);
        Gizmos.DrawLine(transform.position, transform.position +
GetViewAngle(-fieldOfView / 2) * GetComponent().radius);
    }

    Vector3 GetViewAngle(float angle)
    {
        float radian = (angle + transform.eulerAngles.y) * Mathf.Deg2Rad;
        return new Vector3(Mathf.Sin(radian), 0, Mathf.Cos(radian));
    }

    public List ScanForTargets()
    {
        print("Scanning");
        List targets = new List();
        Collider[] results = Physics.OverlapSphere(transform.position,
ScanRange);

        for (int i = 0; i < results.Length; i++)
        {
            var player = results[i].transform.GetComponent();

            if (player == null)
            {
                continue;
            }
            if (!transform.IsInLineOfSight(results[i].transform.position,
fieldOfView, layerMask, Vector3.up))
            {
                continue;
            }
            else
            {
                targets.Add(player);
            }
        }
        PrepareScan();
        return targets;
    }
}

```

---

### Ispis 25: Pretraživanje područja

Gizmos je klasa koja iscrtava linije, različite oblike i slično. To je vidljivo samo tijekom rada unutar scene, ali ne i u igri.

```
namespace SharedGame.Extensions
{
    public static class TransformExtensions
    {
        /**
         * Check if the target is within the line of sight
         */

        /**
         * transform origin
         * target direction
         * field of view
         * check against layers
         * transform origin offset
         * yes or no
         */
        public static bool IsInLineOfSight(this Transform origin, Vector3 target, float fieldOfView, LayerMask collisionMask, Vector3 offset)
        {
            Vector3 direction = target - origin.position;

            if (Vector3.Angle(origin.forward, direction.normalized) <
                fieldOfView / 2)
            {
                float distanceToTarget = Vector3.Distance(origin.position,
                    target);
                //view blocked
                if (Physics.Raycast(origin.position + offset + origin.forward
                    * .3f,
                    direction.normalized, distanceToTarget, collisionMask))
                {
                    return false;
                }
            }
        }
    }
}
```

```
        }
```

```
    return true;
```

```
}
```

```
return false;
```

```
}
```

```
}
```

```
}
```

## Ispis 26: Proširivanje klasa

Kao što je već navedeno, `EnemyPatrol` koristi gore navedene skripte, `Scanner` i `PathFinder`. Osim toga koristi i `WaypointController`. To je skripta s kojom se na jednostavan način određuje kojim će se prostorom kretati neprijatelj tijekom patrole. Sve što je potrebno napraviti je pridružiti objektu roditelj `WaypointController` skriptu te dodati broj djece kojima se dodaje skripta `Waypoint` te tako određujemo prostor kojim će se neprijatelj kretati. `EnemyPatrol` skripta određuje nasumičan smjer kojim će se neprijatelj kretati kao i period vremena koliko će se zadržavati na svakoj od pozicija kako bi se postiglo što realnije ponašanje neprijatelja. Kod 27 prikazuje implementaciju navedenog.

```
private void EnemyPlayer_OnTargetSelected(Player obj)
{
    if (pathFinder.Agent.isActiveAndEnabled)
    {
        pathFinder.Agent.isStopped = true;
    }
}

private void EnemyHealth_OnDeath()
{
    if (pathFinder.Agent.isActiveAndEnabled)
    {
        pathFinder.Agent.isStopped = true;
        enemyHealthBar.SetActive(false);
        transform.GetComponentInChildren().enabled = false;
    }
}

private void WaypointController_OnWaypointChanged(Waypoint waypoint)
{
```

```

        pathFinder.SetTarget(waypoint.transform.position);
    }

    private void PathFinder_OnDestinationReached()
    {
        //patroller
        GameManager.Instance.Timer.Add(waypointController.SetNextWaypoint,
Random.Range(waitTimeMin, waitTimeMax));
    }

```

### Ispis 27: Patroliranje neprijatelja

EnemyHealth i EnemyAnimation su implementirani na identičan način kao već opisane skripte za iste funkcionalnosti igrača te zbog toga nisu detaljnije prikazane.

EnemyShoot skripta implementira ponašanje neprijatelja u trenutku kad detektira igrača. Nakon što neprijatelj otkrije igrača prestaje s patroliranjem te ulazi u stanje svjesnosti te napada igrača, počinje ciljati i pucati. To je riješeno tako da puca u nasumičnim intervalima i u kratkim rafalima. Iako vrlo precizno prema igraču, ipak uz mali pomak kako bi pucanje bilo što realnije. Osim toga neprijatelj se saginje te ponovno puni oružje ako potroši sav spremnik. U slučaju da igrač nije više u vidnom polju neprijatelja, neprijatelj mijenja stanje u nesvjesno te nakon kraćeg perioda nastavlja s patroliranjem. Kôd skripte nalazi se u ispisu 28.

```

private void EnemyPlayer_OnTargetSelected(Player target)
{
    print("OnTargetSelected");
    ActiveWeapon.AimTarget = target.transform;
    ActiveWeapon.AimTargetOffset = Vector3.up * 1.0f;
    StartBurst();
}

void CrouchState()
{
    bool takeCover = Random.Range(0, 2) == 0;

    if (!takeCover)
    {
        return;
    }
}

```

```

        }

        float distanceToTarget = Vector3.Distance(transform.position,
ActiveWeapon.AimTarget.position);

        if (distanceToTarget > 15)

        {

            enemyPlayer.GetComponent().IsCrouched = true;

        }

    }

void StartBurst()

{

    if (!enemyPlayer.EnemyHealth.IsAlive && !CanSeeTarget())

    {

        return;

    }

    CheckReload();

    CrouchState();

    shouldFire = true;

    GameManager.Instance.Timer.Add(EndBurst, Random.Range (

        burstDurationMin,

        burstDurationMax));

}

void EndBurst()

{

    shouldFire = false;

    if (!enemyPlayer.EnemyHealth.IsAlive)

    {

        return;

    }

    CheckReload();

    CrouchState();

    if (CanSeeTarget())

    {

        print(CanSeeTarget());

        GameManager.Instance.Timer.Add(StartBurst, shootingSpeed);

    }

}

bool CanSeeTarget()

```

```

    {
        if (!transform.IsInLineOfSight(ActiveWeapon.AimTarget.position, 90,
enemyPlayer.playerScanner.layerMask, Vector3.up))
        {
            enemyPlayer.ClearTargetAndScan();
            return false;
        }
        return true;
    }

    void CheckReload()
    {
        if (ActiveWeapon.reloader.RoundsRemainingInClip == 0)
        {
            CrouchState();
            ActiveWeapon.Reload();
        }
    }

    void Update()
    {
        if (!shouldFire || !CanFire || !enemyPlayer.EnemyHealth.IsAlive)
        {
            return;
        }
        else
        {
            ActiveWeapon.Fire();
        }
    }
}

```

**Ispis 28:** Ponašanje neprijatelja

### 3.4. Implementacija ostalih funkcionalnosti

Kako bi se omogućilo prikupljanje stvari i korištenje istih napravljen je sistem spremnika za stvari. Uz to su napravljene skripte za prikupljanje stvari. Kako bi se omogućilo ponovno prikupljanje stvari napravljena je skripta koja uklanja stvar kada se prikupi, ali je isto tako ponovno aktivira nakon određenog perioda vremena ako je to potrebno. Time

je opisana pomoćna klasa `Timer` koja je korištena na više mesta kako bi se omogućilo praćenje određenih vremenskih perioda kao što je vrijeme potrebno za ponovno aktiviranje stvari, ponovno pucanje i slično.

Prva skripta je `Container`. To je skripta kojom je definirano koje je stvari moguće prikupiti, da pri tom ostanu kod igrača. S njom je definirano ime stvari, identifikacija spremnika te maksimalan broj određene stvari unutar spremnika. Unutar skritpe su implementirane metode za dodavanje stvari u spremnik, potrošnju ako dođe do nje. Primjerice, ponovnim punjenjem oružja smanjuje se broj metaka u spremniku, a prikupljanjem metaka tijekom igranja taj se broj povećava.

`Container` je referenciran unutar `WeaponLoader` skripte, tako da je za svaki objekt koji sadrži ovu skriptu potrebno pridružiti odgovarajući spremnik.

Prikupljanje stvari iz prostora je riješeno pomoću `PickupItem` skritpe. Unutar skripte implementirana je virtualna metoda `OnPickup`, a svaka skripta koja implementira prikupljanje stvari nasljeđuje ovu klasu. Kod je vrlo jednostavan, svaki objekt koji je moguće prikupiti sadrži komponentu sudarač te se provjerava sudara li se sa igračem preko taga. Ponašanje nakon prikupljanja je definirano u skriptama koje implementiraju različite stvari koje se mogu prikupiti kao što su `WeaponPickup`, `AmmoPickup` itd. Nakon što je stvar prikupljena deaktivira se te, ako je to definirano, aktivira se nakon određenog perioda vremena što je opisano skriptama `Respawner` i `Timer`. `Respawner` koristi `Timer` na način da jednostavno aktivira objekt nakon što prođe definirani period vremena. `Timer` skripta koristi C# događaje te izvršava događaje unutar definiranog vremena. To znači da, ako je potrebno za neku radnju nekoliko sekunda prije nego se ponovno može pokrenuti, ista se dodaje u listu događaja te se definira broj sekundi koliko će se radnja izvršavati. Kod skritpe `Timer` prikazan je u ispisu 29.

```
public class Timer : MonoBehaviour
{
    private class TimedEvent
    {
        public float TimeToExecute;
        public Callback Method;
    }
    private List<TimedEvent> events;
```

```

public delegate void Callback();

void Awake()
{
    events = new List<TimedEvent>();
}

public void Add(Callback method, float inSeconds)
{
    events.Add(new TimedEvent
    {
        Method = method,
        TimeToExecute = Time.time + inSeconds
    });
}

void Update()
{
    if (events.Count == 0)
    {
        return;
    }

    for (int i = 0; i < events.Count; i++)
    {
        var timedEvent = events[i];
        if (timedEvent.TimeToExecute <= Time.time)
        {
            timedEvent.Method();
            events.Remove(timedEvent);
        }
    }
}

```

### Ispis 29: Odbrojavač

Vožnja automobila nije implementirana unutar ovog projekta već je korišten gotov projekt koji je preuzet sa stranica trgovine za dodatke. Stranica tvorca navedenog projekta je <http://www.bonecrackergames.com>. Korištenje ove funkcionalnosti je vrlo jednostavno,

potrebno je na vozilo dodati `RealisticCarController` skriptu, `RealisticCarCamera` skriptu te potom aktivirati sudarače za kotače vozila, tako što se odabere odgovarajući model kotača na vozilu. Uz to, s ovim projektom dolazi i sistem za jednostavno dodavanje zvukova, svjetla kao i različitih vizualnih efekata automobila tijekom vožnje te grafičko sučelje kada je igrač unutar vozila.

Ono što je implementirano unutar Dionyzus projekta je ulazak i izlazak iz automobila na način da je to prikazano preko animacija. Skripte su vrlo jednostavne, te neće biti detaljno prikazane. Implementirane su na način da se aktiviraju i deaktiviraju potrebne komponente ovisno o tome ulazi li igrač u automobil ili izlazi. Osim toga, definirane su različite pozicije kako bi se što realnije mogle izvršiti animacije koje prikazuju navedene aktivnosti.

Tijekom igre igrač obavlja određene zadatke, odnosno misije te se kada je vrijeme za to aktiviraju kratki filmovi (engl. *Cutscenes*) koji prikazuju scenu bez aktivnosti korisnika. To su filmovi koji se odvijaju unutar perioda vremena, odnosno u određenim trenutcima se pokreću različite animacije, pomak kamere, mijenjanje kuta prikaza, prikaz dijaloga i slično. Na taj način je omogućeno upoznavanje s pričom igre koja je predefinirana. Te funkcionalnosti unutar ovog projekta su riješene korištenjem `IEnumerator` sučelja. To je sučelje koje implementira vrijeme koje je potrebno da se određene aktivnosti odviju. Dakle korisnik ne utječe na iste već se odvijaju prema definiranom redoslijedu. Sve što je potrebno unutar skripte je definirati što će se dogoditi nakon određenog vremenskog perioda. To može biti:

- Izmjenjivanje različitih kamera kao i promjena kuta gledanja istih.
- Pojavljivanje teksta koje može dodatno opisati scenu.
- Aktiviranje različitih zvukova kako bi se dočarala atmosfera unutar scene .
- Aktiviranje dijaloga između likova unutar igre.
- Definiranje automatskog kretanja lika do određene destinacije.
- Aktiviranje i deaktiviranje obavijesti.

Navedene mogućnosti su korištene ne samo za kratke filmove već i kada igrač treba dobiti dodatne informacije o trenutnoj misiji i slično.

Implementiranje grafičkog prikaza filma radi se pomoću sučelja. Potrebno je dobiti u scenu glavne aktere, postaviti kamere, urediti prostor gdje će se odvijati film, ubaciti

različite zvukove, animirati likove, dati im glas i slično.

Implementacija misija je izvršena preko zasebnih skripti gdje svaka definira različite informacije ovisno o tome što igrač treba raditi kako bi je izvršio. Glavna skripta je `MissionManager` koja prati tijek igre tako što svaka pojedina misija ima vlastiti broj koji se povećava ovisno o tome koja je misija trenutno aktivna. Svaka misija isto tako može imati dodatne zadatke koji su sadržani unutar pojedinačne misije te se zatvaraju po završavanju zadatka. U trenutku kada je izvršena kompletna misija, broj trenutne misije se povećava za jedan te se aktivira iduća misija. Na ovaj način je vrlo jednostavno moguće dodati nove misije neovisno o ostalima. Kod skripti za svaku pojedinu misiju neće biti prikazan zato što se radi o vrlo jednostavno slaganju različitih sekvenci događja gdje se aktiviraju, odnosno deaktiviraju različiti objekti po potrebi te se daju dodatne informacije igraču kako bi znao što raditi. Problem pronalska odgovarajućih lokacija riješen je na način da je igraču dodan pokazivač koji je uvijek usmjeren prema idućem zadatku. Na taj način korisnik uvijek zna što sljedeće treba napraviti, a osim toga pokretanjem izbornika pauze u svakom trenutku može ponovno pročitati opis zadatka.

Grafičko sučelje je vrlo važan element svake igre. Osim što je vizualna prezentacija određenih parametara unutar igre, može imati i važnu ulogu kada je u pitanju zainteresiranost igrača za igru. Ako sučelje nije dovoljno intuitivno moguće je pokvariti igračima iskustvo igranja igre, a ako je nedovoljno interesantno napravljeno može ostaviti utisak kao da se ne radi o dobro razrađenom projektu.

Grafičko sučelje u Dionyzus projektu možemo podijeliti na dva dijela. Prvi dio je prezentacija osnovnih podataka tijekom igranja, kao što su status spremnika, trenutno oružje u uporabi i energija lika, te drugi dio gdje su glavni izbornik i izbornik pauze. U osnovi su oba dio `Canvasa`. To je objekt igre koji sadrži sve ostale elemente grafičkog sučelja, odnosno elementi se dodaju kao djeca `Canvasa`.

Dizajniranje sučelja se radi unutar prozora scene. Najčešće je u projekt potrebno dodati određene teksture koje će predstavljati različite elemente. Tako je npr. za prikaz spremnika upotrebljena slika torbe za oružje te su iznad toga prikazana odgovarajuća oružja. Isto tako su prikazani i dinamički elementi kao što su energija i broj metaka preostao u spremniku, ali je promjena tih podataka, kako vizualno tako brojevno napisana unutar skripti.

Osim osnovnih elemenata, tijekom igranja igraču se prikazuju tzv. uputstva, od-

nosno naredbe koje je potrebno napraviti kako bi se određena aktivnost izvršila, npr. ulazak i izlazak iz automobila, započinjanje razgovora i slično.

Izrada izbornika moguća je na više načina. Osnovni pristup je preko Canvasa gdje se dodaju elementi kao što su slika koja predstavlja pozadinu, tekstualne prezentacije različitih mogućnosti unutar izbornika, i slično. Drugi način je izraditi trodimenzionalni prostor gdje je onda moguće dodati različite objekte koje je potom moguće i animirati kako bi se postigli zanimljivi vizualni efekti. Osim toga moguće je dodati i zvučne efekte. To može biti glazba u pozadini, kao i različiti zvučni efekti koji se aktiviraju pritiskom na neku od opcija.

Funkcionalnosti različitih botuna moguće je opisati unutar skripti, ali isto tako i unutar samog sučelja. Jednostavne funkcionalnosti kao što su zatvaranje i otvaranje nekog od prozora moguće je jednostavno implementirati unutar nadglednog prozora.

## 4. Zaključak

Ovim projektom predstavljen je rad unutar velikog sustava kao što je Unity3D. Opisano je kako je na jednostavan način moguće izraditi igre korištenjem vrhunskih mogućnosti sučelja za izradu igara u kojem su glavne komponente kao što su fizika, animacije i grafika već implementirane. Opisane su i mogućnosti kako proširiti navedene funkcionalnosti te na koji način su iste upotrebljene u projektu Dionyzus. Ovaj projekt je izrađen tako da je što više komponenti moguće ponovno upotrijebiti u novim projektima sličnih tema, ali isto tako predstavlja osnovne koncepte koje svaka 3D igra treba implementirati. Može se reći da su ovim projektom postavljeni temelji igre otvorenog svijeta te iako se radi o kratkoj i jednostavnoj priči, omogućeno je da se na vrlo jednostavan način proširi. To se najviše odnosi na zadatke koje igrac mora odraditi kako bi otkrio priču, ali isto tako i istražio kompletno područje kojem može pristupiti tijekom igranja te na inteligenciju neprijatelja, odnosno likova koji nisu kontrolirani od strane igraca. To može uključivati osluškivanje okoline, intelligentnije pretraživanje područja kao i ponašanje tijekom sukoba.

Unutar skripti cilj je bio postići konzistentnost pisanja kôda. To se najviše odnosi na nazive varijabli, metoda, klasa i slično te na održavanje osnovnih koncepata objektno orijentiranog programiranja.

U ovom projektu nisu obrađene kompleksne teme kao što su kompletna optimizacija performansi igre, izrada modela, animacija i slično zato što je projekt izrađen s onim mogućnostima Unityja koje su ponuđene svakom korisniku bez obzira na iskustvo u izradi igara. To se odnosi na preuzimanje gotovih modela iz Unity trgovine s dodacima gdje su svi preuzeti modeli besplatni, što podrazumijeva lošije optimizirane teksture, dimenzije i slično, a izbor objekata je temeljen na potrebama igre.

Animacije su preuzete sa stranice <https://www.mixamo.com>, a zvukovi sa <https://freesound.org> te su dodatno obrađeni u slučaju potrebe. Projekt je izrađen uz pomoć <https://www.youtube.com> vodiča, gdje su pronađene informacije za izradu kompleksnijih elementa igre. A za snalaženje unutar Unity sučelja je korištena službena dokumentacija.

## **Literatura**

[1] Unity User Manual(2019.\*),

<https://docs.unity3d.com/Manual/index.html>

# **Dodaci**

**SVEUČILIŠTE U SPLITU**  
**SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE**

Preddiplomski stručni studij Informacijska tehnologija

**Ivan Odak**

**Z A V R Š N I   R A D**

**Dionyzus Unity3D igra**

Split, lipanj 2019.

**SVEUČILIŠTE U SPLITU**  
**SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE**

Preddiplomski stručni studij Informacijska tehnologija

**Predmet:** Objektno orijentirano programiranje

**Z A V R Š N I   R A D**

**Kandidat:** Ivan Odak

**Naslov rada:** Dionyzus

**Mentor:** Ljiljanja Despalatović, dipl. ing.

Split, lipanj 2019.