Izrada avanturističke igre iz prvog lica u programskom alatu Unity

Matija, Kralj

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Matija Kralj

Izrada avanturističke igre iz prvog lica u programskom alatu Unity

DIPLOMSKI RAD

Varaždin, 2020.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Matija Kralj

JMBAG: 0016118416

Studij: Informacijsko i programsko inženjerstvo

Izrada avanturističke igre iz prvog lica u programskom alatu Unity

DIPLOMSKI RAD

Mentor:

Dr. sc. Mladen Konecki

Varaždin, studeni 2020.

Matija Kralj

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Ovaj rad se sastoji od pisanog i praktičnog dijela. Pisani dio se sastoji od opisa programskog alata Unity i žanra avanturističkih igara. Nakon toga slijedi opis računalne igre koja je izrađena u svrhu diplomskog rada te opis algoritama koji su korišteni u kreiranoj računalnoj igri. Praktični dio se sastoji od funkcionalnog prototipa računalne igre implementirane u programskom alatu Unity. Igra sadržava vatrena oružja s implementiranom balistikom osjetljivom na gravitaciju te na usporavanje projektila tijekom vremena, igra također sadrži agente neprijatelje protagonistu i koriste se umjetnom inteligencijom za takvu vrstu posla. Igra je većim dijelom izrađena uz pomoć tečaja na Udemy portalu.

Ključne riječi: računalna igra, algoritmi, Unity, igra iz prvog lica, avanturistička igra, programiranje

Sadržaj

1. Uvod

Ova tema s fokusom na izradu videoigre je odabrana radi autorove zainteresiranosti što se tiče ovog područja kao i iz razloga što je industrija videoigara sve cjenjenija u svijetu. Sljedeći razlog odabira ovakve teme je i prisutnost videoigara kod sve više ljudi u svakodnevnom životu te zbog sve veće pristupačnosti kreiranja videoigara svima, a ne samo velikim tvrtkama.

Ovaj rad će se sastojati od dva dijela, od teoretskog i praktičnog. Teoretski dio fokusirat će se najviše na alat u kojem je igra izrađena, Unity. Početkom rada bit će ukratko opisana povijest Unityja i ono što Unity zapravo jest. Nakon toga slijedi nekoliko poglavlja koja opisuju kako je raditi u alatu. Potonje podrazumijeva bit rada u Unityju, izgled alata i njegovi glavni dijelovi, podešavanje važnih postavki projekta te na kraju osvrt na programiranje (skriptiranje) koje je u skladu s Unityjem. Sljedeće, manje poglavlje koje također spada u teoretski dio, sastojat će se od žanrova i tipova videoigara. Pod navedeno poglavlje se podrazumijeva popis i opis žanrova te neke od poznatijih predstavnika svakog žanra. Nakon toga fokus se prebacuje na detaljniji opis avanturističkog žanra, glavne mehanike žanra i kako prepoznati videoigru ovog žanra. Bit će navedeno nekoliko igara i njihove igrače mehanike te što ih točno čini avanturističkim igrama.

Što se tiče praktičnog dijela, kreiran je prototip igre i on je opisan u poglavljima nakon navedenih u prethodnom odlomku. U ovom dijelu će detaljno biti opisani dijelovi igre kao što su neke važnije skripte, umjetna inteligencija, animacije i likovi. Kako animacije i likovi nikada ne mogu biti potpuni bez *animatora,* dijela koji međusobno povezuje animacije, geometriju i skripte, bit će riječi i o tome. Umjetna inteligencija će biti veliko poglavlje jer u navedeno područje spadaju navigacija i agenti zajedno sa skriptama te nekim njihovim dodatnim dijelovima.

2. Unity

Korišteni alat za izradu praktičnog dijela ovog rada zove se Unity te će u ovom poglavlju biti i detaljno opisan. Citat koji daje vrlo dobar uvod u temu je sljedeći: „Unity (poznatiji kao Unity3D) je sustav koji obuhvaća *game engine* i razvojnu okolinu (IDE) za kreiranje interaktivnih sadržaja, najčešće videoigara.“ (Haas, 2014). *Game engine* se neće doslovno prevoditi u ovom radu jer hrvatski jezik nema dovoljno dobar prijevod za taj pojam. Osnovni opis pojma je da je to pokretač računalnih igara, kako i riječ govori, to je „mašina za igre“, odnosno sve ono što je potrebno na programskoj razini (engl. *Software*) da se igra može izvoditi na računalu. U citatu se pojavljuje kratica IDE (engl. *Integrated development environment*) koja predstavlja skup potrebnih alata i pomoćnih komponenti u nekom alatu za razvoj nekog komada programskog proizvoda. Integrirana razvojna okolina (IDE) je programski proizvod koji služi za izgradnju aplikacija, ali obuhvaća uobičajene alate kao pomoć razvojnim developerima, integrirane u jednom grafičkom sučelju (engl. *Graphical User Interface* – GUI). IDE se najčešće sastoji od uređivača programskog koda, sustava za automatsku izgradnju programa i sustava za otklanjanje pogrešaka. (Henley, 2011).

Prva verzija alata objavljena je 2005. godine, kreiran je u Danskoj, a autori su David Helgason, Joachim Ante i Nicholas Francis (Haas, 2014; *Unity at 10: For better-or worse-game development has never been easier*, bez dat.). Također, važno je napomenuti da je cilj kreatora Unityja bio omogućiti razvoj videoigara svima, neovisno o budžetu i tehničkom znanju. Također je vrlo važno napomenuti da je Unity isprva bio ponajviše posvećen iPhone uređajima te je dobio i nagradu od Applea na „World Wide Developer Conference“ kao prvi potpuni *game engine* za spomenute uređaje. Tada je sadržavao osnovnu grafiku, fizičke kalkulacije i neke već ugrađene programske kodove (Axon, 2016).

Unity se, kao što možemo vidjeti, nije zadržavao samo na iPhone uređajima, već je postao popularan na svim ostalim uređajima koji su mogli podržavati ovakvu vrstu sadržaja. Danas je jasno vidljivo da je Unity ozbiljna konkurencija apsolutno svim ostalim alatima za razvoj videoigara i pruža podršku različitim platformama i uređajima (Shah, 2017).

Zbog toga što je Unity dostupan svima, a ne samo velikim timovima i kompanijama koje imaju budžeta, alat su jako počeli cijeniti pojedinci i govore kako je to alat za razvoj igara s kojim se može uspjeti i zaraditi. Baš zbog toga je u zadnje vrijeme došlo do naglog porasta dostupnih videoigara (Axon, 2016).

2.1. Rad u Unityju

Imajući na umu glavni cilj kod kreiranja samog Unityja, možemo zaključiti kako bi u alatu, naravno ukoliko je cilj postignut, trebalo biti relativno lagano raditi (*user friendly*). Također ukoliko je cilj kreatora bio stvoriti alat koji će koristiti svima, neovisno o tehničkom (pred)znanju, ako je cilj postignut, trebalo bi bit moguće kreirati nekakav sadržaj bez ikakvog znanja o programiranju, animiraju, dizajniranju scena, likova, itd. Ukoliko se pokaže da su ove tvrdnje većim dijelom istinite, možemo zaključiti kako su autori uspjeli u svom naumu. Uz navedeno, važno je i skrenuti pogled na sve mogućnosti koje Unity pruža te vidjeti koliko je fleksibilan za korisnikove nadogradnje i pogodan za velike timove i profesionalce, a ne samo za pojedince neovisno o znanju.

Gotovo sve što postoji u Unityju, a ima veze s kreiranjem igara ili sličnog sadržaja, bazirano je na povuci-ispusti principu (engl. *Drag and drop*) (Brodkin, 2013). Ovakav princip osigurava jednostavnost korištenja alata u smislu da ukoliko je neki objekt potrebno referencirati, primjerice u skripti, željeni objekt se jednostavno može povući i ispustiti na to predviđeno mjesto. Također Brodkin u članku spominje kako je jedan od autora (Helgason) rekao da je moguće kreirati računalnu igru u Unityju bez ikakvog pisanja programskog koda, ali da većina igara to ipak zahtijeva. To i jest istina jer postoje brojne već kreirane komponente u Unityju koje možemo dodati objektima, podesiti parametre prema potrebi i jednostavno isprobati kako rade. Jedan od brojnih primjera komponenti koje dolaze u standardnom Unity paketu je komponenta „FirstPersonController“ koju možemo jednostavno povući i ispustiti na objekt u igri te komponenta pruža rješenje za upravljanje likom, odnosno omogućuje hodanje i micanje kamere pomoću računalnog miša. Sve komponente koje postoje su u pozadini zapravo programski kod (skripte) koje korisnik može kreirati ili mijenjati, no o tome će biti riječi kasnije.

Kod rada u Unityju je važno napomenuti krivulju učenja rada u samom alatu koja je relativno strma (uz manji broj ponavljanja dolazi do većeg učinka) i to zbog toga što je alat, odnosno razvojno okruženje, intuitivan za korištenje i sve je na mjestu na kojem bi korisnik to i očekivao. Drugi i možda najvažniji razlog brzog učenja razvoja igara u Unityju je taj što Unity pruža vlastite početničke tečajeve za korisnike koji se još nisu nikada susreli ni s programiranjem, a kamoli s razvojem računalnih igara. Unity pruža tečajeve raznih težina koji imaju za cilj pripremiti korisnika na greške koje se često pojavljuju u razvoju te na ono što se u razvoju najčešće koristi. Treći razlog takve krivulje učenja je taj što je dostupna kompletna dokumentacija s primjerima za svoj kompletni API (engl. *Application programming interface*) (Craighead, Burke, & Murphy, 2008). Još jedan razlog je prisustvo velikog broja kvalitetnih

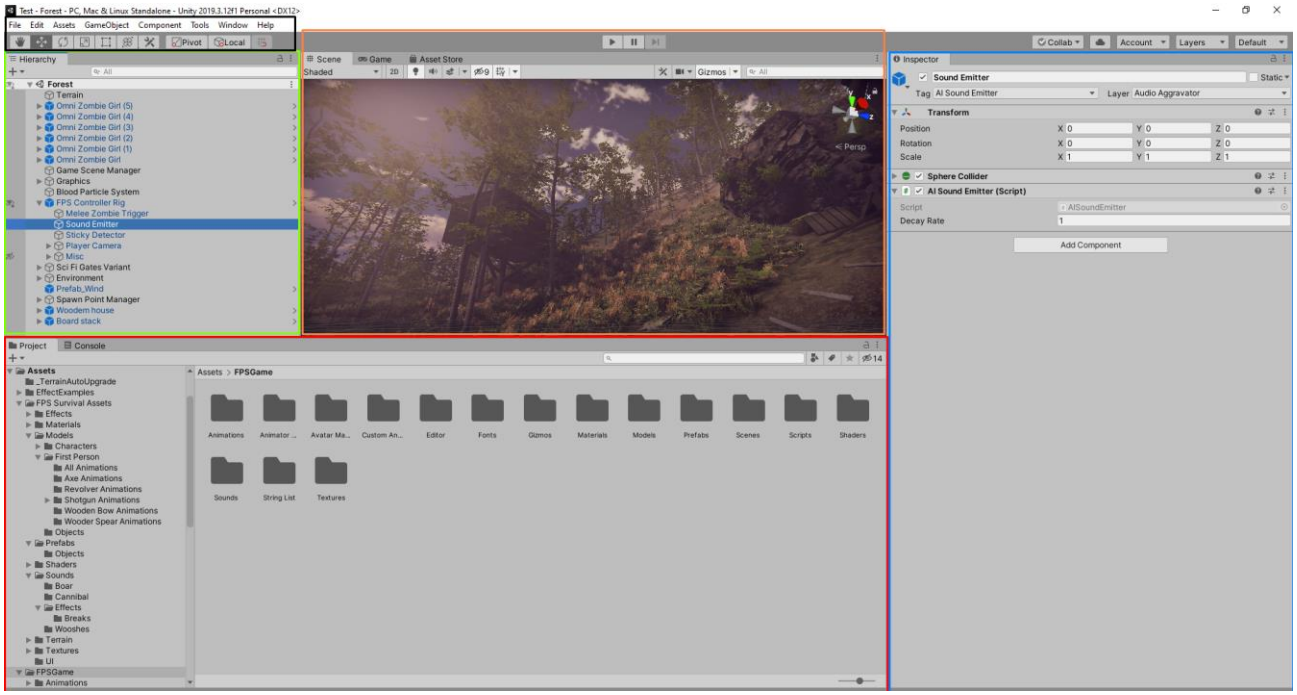
video materijala dostupnih na raznim web stranicama (Udemy, YouTube, itd.). Također je važno napomenuti da Unity sponzorira puno ovakvih materijala što daje ljudima veći povod za kreiranje korisnih video materijala. Na kraju, postoji još jedan razlog koji ubrzava učenje i smanjuje vrijeme provedeno u rješavanju problema, a to je da postoji on-line zajednica developera koji pomažu jedni drugima te korisnici u toj zajednici mogu predlagati poboljšanja i nove značajke za alat (Craighead i ostali, 2008).

Učenje bilo kakve nove vještine je samo po sebi zahtjevan posao, a kada govorimo o učenju vještine razvoja računalnih igara, razina težine se povećava. Na sreću, Unity je u ovom području dao sve od sebe, kao što se može vidjeti u tekstu iznad, da pomogne svim svojim korisnicima te zbog toga ima veliki plus kod početnika.

2.2. Razvojno okruženje

Kako bismo mogli bolje razumjeti kakav je rad u Unityju, trebali bismo najprije znati osnove o razvojnom okruženju alata. Potrebno je znati gdje se što nalazi, imena raznih prozora (engl. *Windows*) unutar alata i za što oni služe.

U nastavku slijedi slika alata i prikazuje kako on izgleda, uz iznimku što je ovaj projekt već popunjen elementima i raznim objektima te ima puno više mapa koje čuvaju važne grupirane podatke.

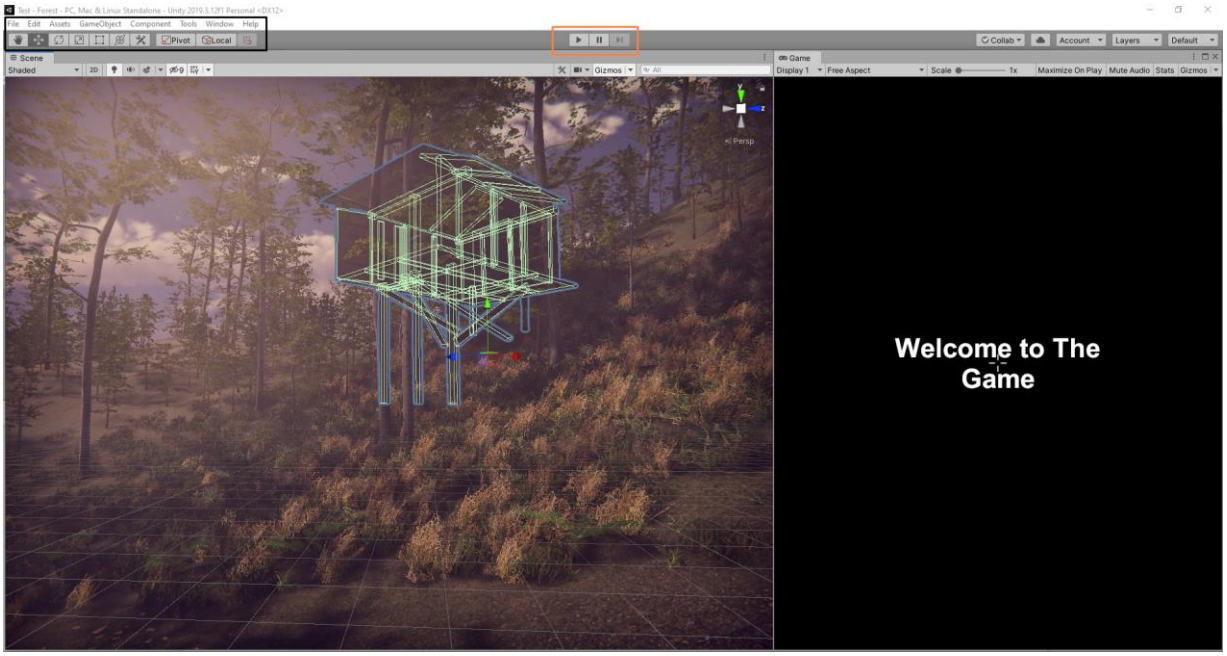


Slika 1. Unity razvojno okruženje

Dijelovi slike obojani su raznim bojama kako bi bilo lakše pratiti gdje se što nalazi te će sada biti svaki od ovih dijelova detaljnije opisan, ali će prozori biti odvojeni od ostatka alata kako bi bili što veći. Neki prozori će biti prikazani u paru kako bi se prikazao smisao i ovisnost jednog prozora o drugom. Važno je napomenuti da će većina opisa biti rezultat autorovog iskustva u radu s alatom, dok se detalji mogu pronaći na web stranici dokumentacije kreirane od strane Unity Technologies (*Working in Unity*).

2.2.1. Prozor igre i scene s alatima

Prvi prozor koji će biti opisan je prozor scene, zapravo i najvažniji prozor u alatu te uz njega u paru dolazi i prozor igre. Na slici iznad (Slika 1) prozor igre nije vidljiv, ali je vidljiv prozor scene i označen je narančastom bojom, a alati i izbornik su označeni crnom bojom kao i na slici koja slijedi.



Slika 2. Prozor igre i scene s alatima

Dva velika prozora lijevo i desno prikazuju prozor scene na lijevoj strani i prozor igre na desnoj. U prozoru scene možemo uređivati, pomicati i rotirati sve objekte koji postoje u sceni i to se radi pomoću alata u gornjem lijevom uglu, označenih crnom bojom. Alati redom predstavljaju: alat za pozicioniranje unutar scene (engl. *Hand tool*), za pomicanje objekata (engl. *Move tool*), za rotiranje objekata (engl. *Rotate tool*), za skaliranje objekata (engl. *Scale tool*), za skaliranje kao kvadrat (engl. *Rect tool*), za pomicanje, rotiranje ili skaliranje objekata

(engl. *Move, rotate or scale tool*), dostupni alati za prilagođeno uređivanje (engl. *Available custom editor tools*).

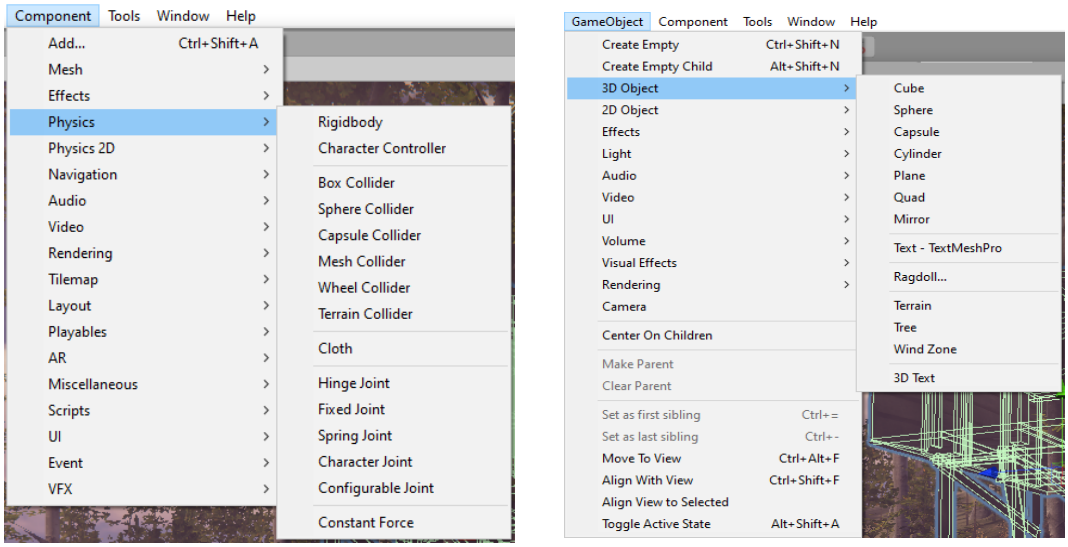
Na slici u prozoru scene možemo vidjeti odabrani objekt (engl. *Game object*) koji je skup velikog broja manjih objekata te tako tvori drvenu kuću. Svako zeleno obojano geometrijsko tijelo predstavlja jedan objekt, no u podnožju kuće mogu se vidjeti tri strelice, svaka označena drugom bojom te ispod njih kocku. Svaka od ovih strelica prikazuje koordinatnu os (crvena – x, zelena – y, plava – z). Ovisno o odabranom alatu, strelice se mijenjaju, primjerice u kružnice kod alata za rotiranje (rotiramo alat oko odabrane osi), u linije s kockom na njihovim krajevima ukoliko se radi o alatu za skaliranje, nestaju odabirom alata za pozicioniranje unutar scene i sl. U gornjem tekstu je nabrojano prvih sedam alata, preostala tri neposredno do njih su usko povezana s prvih sedam i sa spomenutim strelicama.

Prvi od tri alata je alat za odabir mjesta na kojem se strelice nalaze, a moguće opcije su da se nalaze u centru objekta ili na poziciji pivota (važno kod rotacije). Drugi alat govori o tome kako su strelice orijentirane, odnosno prikazuju li lokalne osi objekta ili osi svijeta u kojem se objekt nalazi. Trenutno je os y pozicionirana tako da prikazuje prema nebu te je isto tako i ukoliko je odabrani prikaz orijentacije u koordinatnom sustavu svijeta. Ukoliko bismo kuću rotirali za 180 stupnjeva oko x ili z osi, tada bi zelena strelica (y os) u lokalnom sustavu pokazivala prema terenu, dok bi koordinate svijeta i dalje ostale kako su i bile (rotirali smo objekt, a ne i svijet). Ovaj alat je jako koristan ukoliko želimo poravnati objekt sa svijetom, a ukoliko želimo objekte unutar nekog objekta poravnati sa svojim roditeljem (objektom koji ga sadrži), tada koristimo lokalne koordinate. Zadnji od tri alata je omogućen tek ukoliko se nalazimo u koordinatnom sustavu svijeta i omogućuje automatsko poravnanje objekta s linijama koordinatnog sustava (engl. *Toggle grid snapping*). Linije su vidljive na slici kao bijeli kvadratići pri dnu ekrana.

Kod alata je važno napomenuti traku s izbornicima, koja se nalazi tik iznad spomenutih alata. Izbornici imaju pregršt opcija, no najvažnije opcije se nalaze u izbornicima „Edit“, „GameObject“, „Component“, i „Window“. Prvi od navedenih će detaljno biti objašnjen kasnije zbog važnosti jedne od opcija u tom izborniku. „GameObject“ izbornik (Slika 3 - desno) pruža mogućnosti kreiranja različitih vrsta objekata, efekata, svjetla, itd. Jedna od važnijih i korisnijih opcija je mogućnost poravnanja trenutno odabranog objekta s pogledom u sceni (engl. *Align with view*). Prikaz ovog izbornika nalazi se na slici (Slika 3), a prikazuje sve kategorije i navedene mogućnosti izbora.

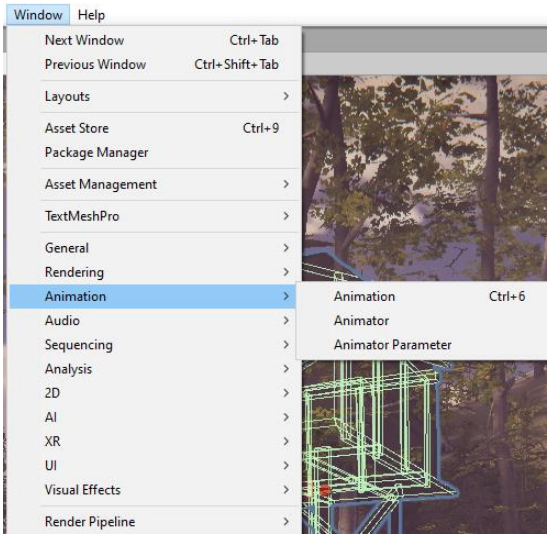
„Component“ izbornik (Slika 3 - lijevo) pruža mogućnosti dodavanja raznih komponenti na objekt, kao što su efekti, razne fizičke komponente, događaji, skripte i još mnogo toga.

Spomenuta slika prikazuje samo jednu od mnogih komponenti koje korisnik ima u arsenalu komponenti.



Slika 3. Component (lijevo) i GameObject (desno) izbornik

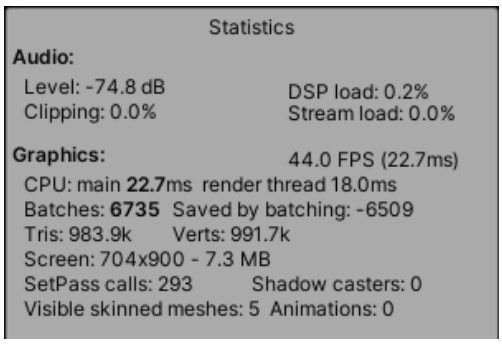
Na kraju preostaje izbornik „Window“ koji pruža mogućnost prikazivanja prozora koji se trenutno ne vide ili pak sakrivanja onih koji su aktivni. Važniji prozori su „Asset Store“, „Animation“, „Animator“, „Package Manager“ i „Visual Effects“. Osnovni prozori koji se koriste gotovo cijelo vrijeme nalaze se pod kategorijom „General“. Na slici ispod može se vidjeti skup prozora koje se može prikazivati ili sakrivati.



Slika 4. Window izbornik

Nakon opisa ovih svih alata, izbornika i prozora scene, preostaju još tri interaktivna gumba i prozor igre, koji su vrlo jednostavni. Na slici (Slika 2) su narančastom bojom označena spomenuta tri interaktivna gumba. Prvi od njih pokreće igru, drugi služi za pauziranje i ponovno pokretanje pauzirane igre, a treći je aktivan samo ukoliko je igra pauzirana i služi pokretanje igre ali samo za jedan korak - sliku (engl. *Frame*).

Prozor igre je jednostavni prozor i prikazuje ono što „vidi“ kamera prema svojim postavkama. Trenutno prozor prikazuje samo crni ekran s tekstom jer je to, iako nije interaktivno, korisničko sučelje (engl. *User interface*) i nalazi se na sloju iznad svega ostalog što kamera može vidjeti. Ovaj prozor također ima neke opcije koje se često koriste, a to su *maksimiziranje* prozora kod pokretanja igre (engl. *Maximize on play*) i statistika (engl. Stats). Statistika je jedan mali skočni prozor koji prikazuje razne detalje koji su korisni razvojnim programerima. Ovaj prozor prikazuje broj slika u sekundi koje se prikazuju (FPS – engl. *Frames per second*), glasnoću zvuka, koliko je ušteđeno i potrošeno resursa i sl. Slika ispod prikazuje opisani prozor.

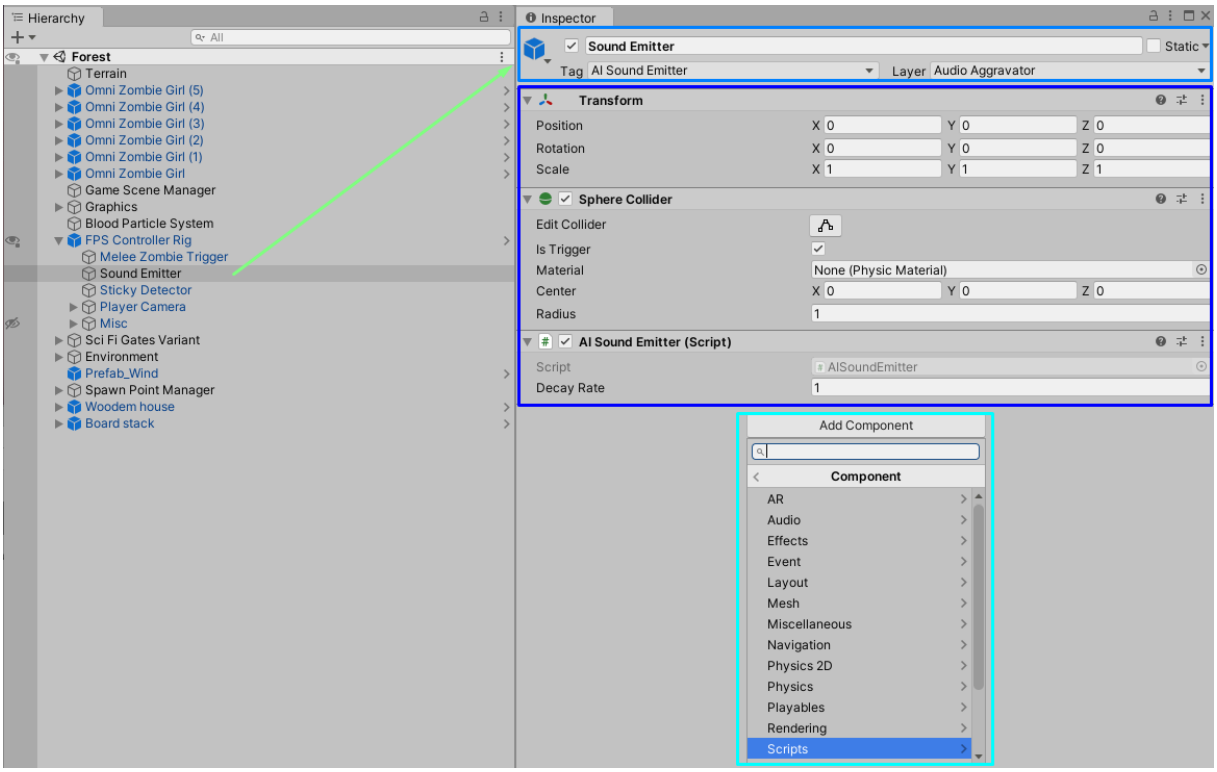


Slika 5. Statistika u prozoru igre

2.2.2. Prozor hijerarhije i inspektor

U ovome dijelu slijedi opis prozora koji se na slici, na početku ovog poglavlja (Slika 1) nalaze krajnje lijevo i označeni su zelenom bojom (prozor hijerarhije objekata), također i krajnje desno, označen plavom bojom (prozor inspektora). Slijedi slika koja prikazuje spomenuta dva prozora zajedno (Slika 6).

Razlog što će se spomenuta dva prozora opisivati u paru je taj što prozor inspektora (desno) direktno ovisi o prozoru hijerarhije (lijevo) i to na način da se na prozoru inspektora prikazuju atributi odabranog objekta iz hijerarhije. Objekt se može selektirati i na način da se odabere u prozoru scene, ali na taj način se on također odabire i u hijerarhiji, što rezultira slikom iznad.



Slika 6. Prozor hijerarhije i inspektor

Prozor hijerarhije objekata prikazuje sve objekte koji se nalaze na sceni, bilo da su oni vidljivi, nevidljivi, uključeni ili isključeni. Objekti mogu biti prazni i ne imati nikakvu ulogu osim grupiranja drugih objekata unutar sebe (biti roditelji objektima koje grupiraju) ili pak mogu biti bilo kojeg drugog tipa (vidi Slika 3 – desno). Objekt koji sadrži druge objekte možemo prepoznati prema trokutu koji se nalazi s lijeve strane imena objekta. Objekt roditelj može sadržavati druge objekte koji su roditelji. Primjer ovakvog objekta je objekt „FPS Controller Rig“ koji je na slici trenutno proširen (vide se potomci kojima je direktni predak) te je vidljivo da sadrži objekte koji su roditelji nekim drugim objektima („Player Camera“ i „Misc“).

Trenutno odabrani objekt u hijerarhiji je „Sound Emitter“ i njegovi svi detalji su sadržani u prozoru inspektora. Prozor inspektora podijeljen je na tri osnovna dijela, svaki označen drugom nijansom plave boje. Najgornji odjeljak prikazuje ime trenutno odabranog objekta, a s lijeve strane imena je indikator koji pokazuje da li je objekt aktivan (postoji li u sceni, odnosno u svijetu). Ukoliko je objekt roditelj deaktiviran, cijela hijerarhija tog objekta je također deaktivirana, bez obzira što su objekti potomci označeni kao aktivni (Unity Technologies, 2020d).

Ispod imena nalazi se oznaka (engl. *Tag*) objekta pomoću koje možemo objekte prepoznavati u skriptama, kao primjerice igrače, emitere zvuka, emitere svjetla, neprijatelje, itd. Jednu oznaku objekta možemo dodati jednom ili više objekata i svaki objekt može imati samo jednu takvu oznaku, a oznake su korisne kod upravljanja kolizijama između objekata. Primjer kolizije, koji je čest i relativno lagan za shvatiti, je skripta koja detektira može li igrač pokupiti objekt. Ako je objekt označen kao nešto što se može pokupiti (engl. *Collectable*), tada skripta daje korisniku do znanja da se objekt može pokupiti (Unity Technologies, 2020j).

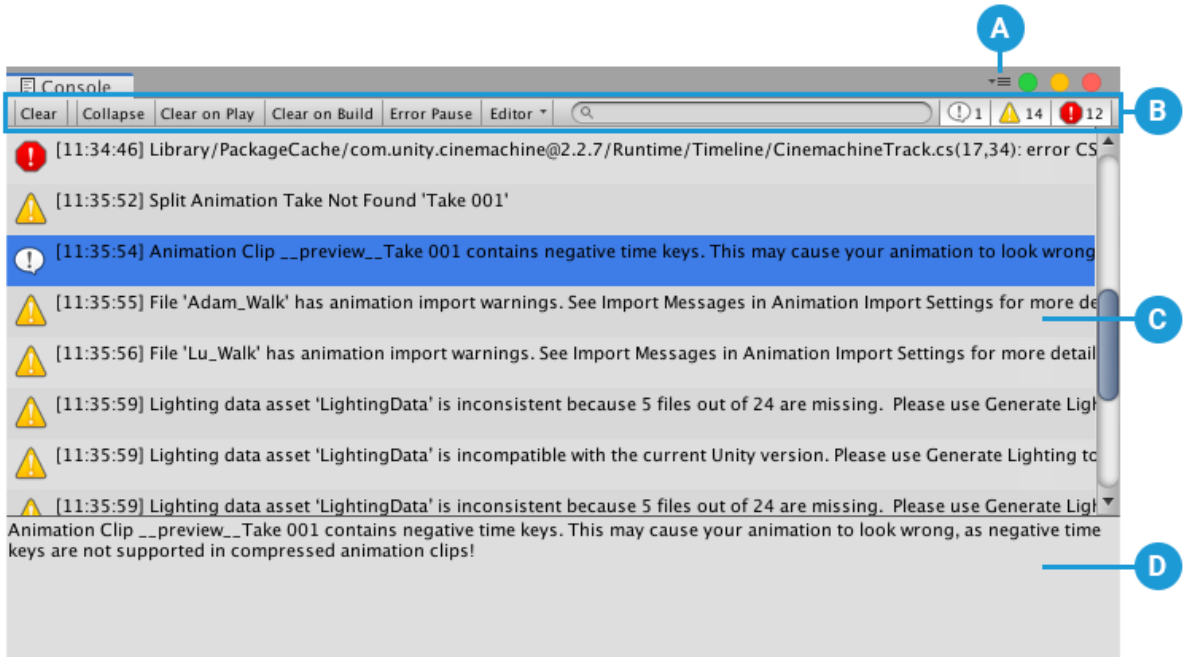
Desno od oznake nalazi se ime sloja kojem objekt pripada (engl. *Layer*). Pomoću sloja grupiramo objekte te slično kao oznake, objekt može pripadati samo jednom sloju i sloj može biti pridodan jednom ili više objekata. Slojevi se često koriste kod kamere da se označi što će se na kameri iscrtavati (engl. *Render*) i koriste se kod svjetla kako bi samo neki dijelovi scene bili osvijetljeni (Unity Technologies, 2020j). Slojevi se mogu koristiti i na sličan način kao i oznake kada se radi o detektiranju kolizija, no jedno jako važno svojstvo slojeva je što se može podesiti koji slojevi mogu generirati kolizije s drugim slojevima. Čest primjer korištenja slojeva su: voda, teren, prepreka hodanju i sl. Ukoliko igrač detektira koliziju sa slojem vode - animira se plivanje, ukoliko igrač detektira koliziju sa slojem „prepreka hodanju“ - onemogućuje se hodanje u tom smjeru i sl.

Korištenje slojeva i oznaka može biti korisno u sljedećem scenariju: u igri postoje razne stvari koje detektiraju kolizije sa projektilima koje korisnik izbacuje iz primjerice oružja te sve takve objekte možemo svrstati pod sloj koji detektira koliziju sa projektilima. Nadalje, sve takve objekte možemo podijeliti pomoću oznaka na primjerice, neprijatelje, eksplozivne objekte, objekte koji ne primaju štetu, objekte koji primaju štetu na specifičan način i sl. Na ovakav način uvijek kod detekcije kolizije sa slojem koji detektira projektile možemo znati s kakvom vrstom objekta radimo. S druge strane, na takav način možemo odrediti da je neki objekt eksplozivan, ali ne detektira koliziju s projektilima, nego se aktivira na neki drugi način.

2.2.3. Prozor projektnog repozitorija i konzola

Posljednji važniji prozori koji su preostali su prozor projektnog repozitorija i prozor konzole te će u nastavku biti opisani.

Prozor konzole (engl. *Console*) služi za ispis raznih programerovih vlastito stvorenih poruka u skriptama ili pak obavijesti, upozorenja i razne greške koje alat ispisuje samostalno (Unity Technologies, 2020b). Programer može također ispisati i upozorenje ili grešku, no to nije čest slučaj korištenja. Slijedi slika prozora konzole sa svojim dijelovima.



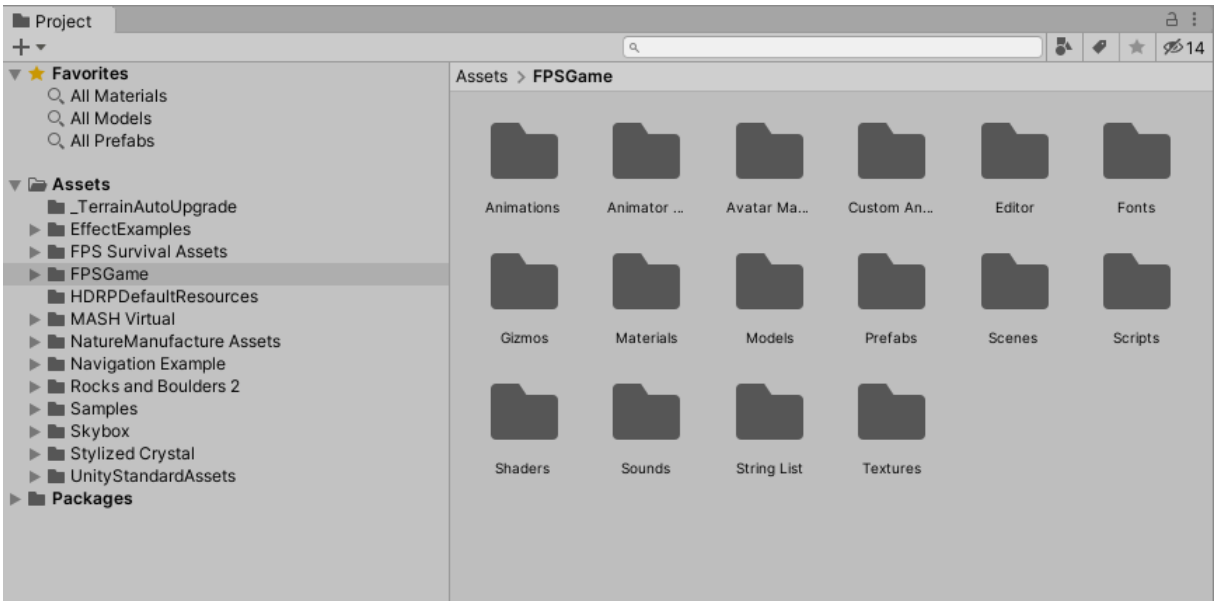
Slika 7. Prozor konzole s obavijestima (Unity Technologies, 2020b)

Označeni dijelovi na slici iznad su (Unity Technologies, 2020b):

- A – Prozor konzole.
- B – Alati konzole koji služe za upravljanje konzolom, od čišćenja svih obavijesti do pauziranja ispisa grešaka. Također pruža mogućnost za pretraživanje obavijesti te koliko ih je ispisano i kojeg su tipa.
- C – Dio u kojem se ispisuju obavijesti te se bilo koja obavijest može odabrati.
- D – Dio u kojem se ispisuju detalji odabrane obavijesti.

Prozor projektnog repozitorija je prikaz stvarnog repozitorija u kojem je spremljen projekt, no prikazuje samo mape i datoteke koje programer može koristiti u projektu ili koje je on kreirao. Ovaj repozitorij sastoji se od dva dijela kako će biti i prikazano na slici koja slijedi (Slika 8).

Prvi dio od kojeg se sastoji repozitorij je mapa resursa (engl. *Assets*), a drugi dio mapa paketa (engl. *Packages*). U mapi paketa nalaze se sve datoteke koje Unity sam preuzme ili kreira kod stvaranja novog projekta. Ukoliko se preuzima paket koji primjerice omogućuje rad s više vrsta kistova nego ih je inače dostupno, takav paket se sprema u ovu mapu te Unity o njemu vodi brigu.



Slika 8. Prozor projektnog repozitorija

U mapi resursa nalaze se svi resursi koji se mogu koristiti za kreiranje igre (scene), bilo preuzeti ili samostalno kreirani. U ovu mapu spremaju se sve skripte, zvukovi, teksture, materijali i sl. Također važno je napomenuti da se bilo koji objekt kreiran u sceni igre može kao cjelina spremiti u ovu mapu. Takav objekt se tada naziva *prefab* i važan je zbog toga jer održava referencu sa svim ovakvim objektima postavljenim u scenu. Promjenom nekog parametra u *prefabu* (objekt u mapi resursa), taj parametar se mijenja i na svim instancama tog objekta u sceni. U mapi resursa, trenutna odabrana mapa je FPSGame te se u toj mapi nalaze sve skripte koje je autor rada kreirao, dok su ostale mape nastale uvozom (engl. *Import*) tuđih resursa.

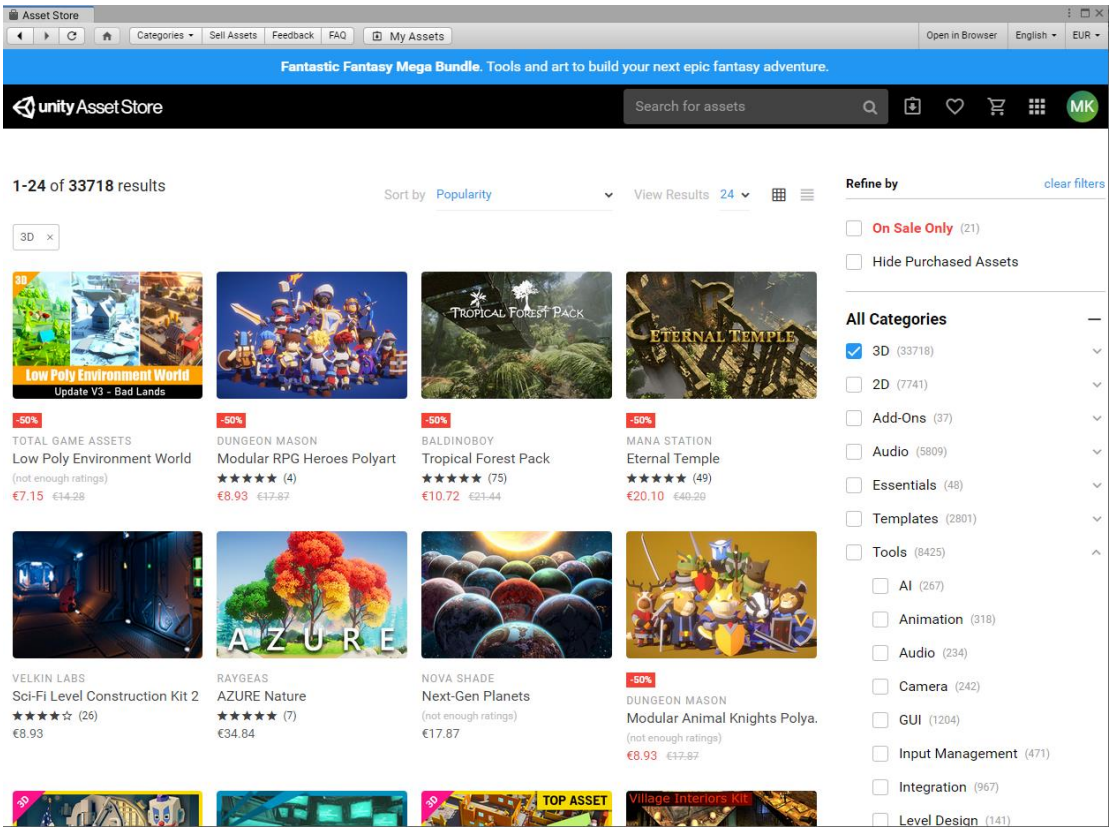
Drugi (tuđi) resursi se uvoze u projekt najčešće pomoću prozora „AssetStore“ koji će u sljedećem poglavlju biti ukratko opisan.

2.2.4. Asset Store i upravitelj paketima

Asset Store i upravitelj paketima su također prozori, ali koji služe za preuzimanje određenih resursa koji na neki način koriste programeru. Oba prozora najčešće nisu direktno odmah dostupna, već im se pristupa otvaranjem izbornika „Window“ i odabirom željene opcije, *Asset Store* ima i prečac za brzo otvaranje Ctrl+9.

Asset Store prozor služi za preuzimanje resursa koje programer može koristiti u svojim projektima. Resursi su kreirani od strane drugih razvojnih programera, raznih timova, tvrtki ili

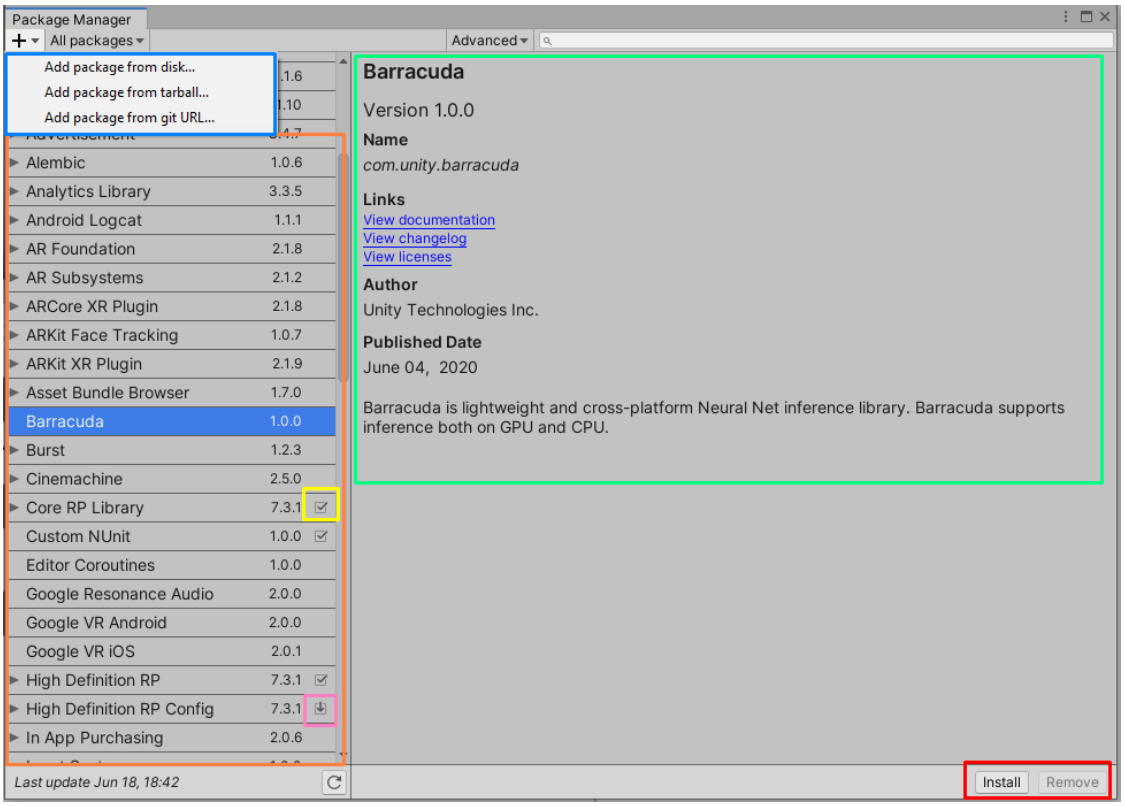
Unityjevog tima. Neki resursi su besplatni za korištenje svima i pružaju licence za korištenje, neke licence dozvoljavaju korištenje samo za osobne potrebe, dok drugi nemaju ograničenja. Postoje resursi koje je potrebno kupiti i takvi resursi najčešće pružaju licence bez ograničenja. Slijedi prikaz ovakvog prozora s prikazom raznih resursa (engl. *Assets*).



Slika 9. *Asset Store* prozor

Upravitelj paketima (engl. *Package Manager*) služi sličnoj namjeni kao i *Asset Store,* odnosno oba prozora imaju zadaću pridonijeti vrijednosti projekta na ovaj ili onaj način. Upravitelj paketima je prozor u kojem se nalaze dodatni alati koji bi mogli pomoći programeru u njegovom radu i svi paketi su besplatni. Slijedi slika prozora upravitelja paketima i kratki opis.

Na slici ispod (Slika 10) nalazi se upravitelj paketima te je označen četirima različitim bojama, svaka od njih prikazuje različite, ali važne dijelove prozora. Plavo označeni dio prikazuje opcije uvoza (engl. *Import*) paketa izvana te su moguće tri različite opcije. Odmah iznad plavo označenog dijela nalazi se izbornik u kojem je trenutno tekst „All packages“. Klikom na ovaj izbornik otvara se mogućnost pregleda samo onih paketa koji se trenutno nalaze u projektu, prikaz svih (trenutna opcija) ili prikaz ugrađenih paketa.

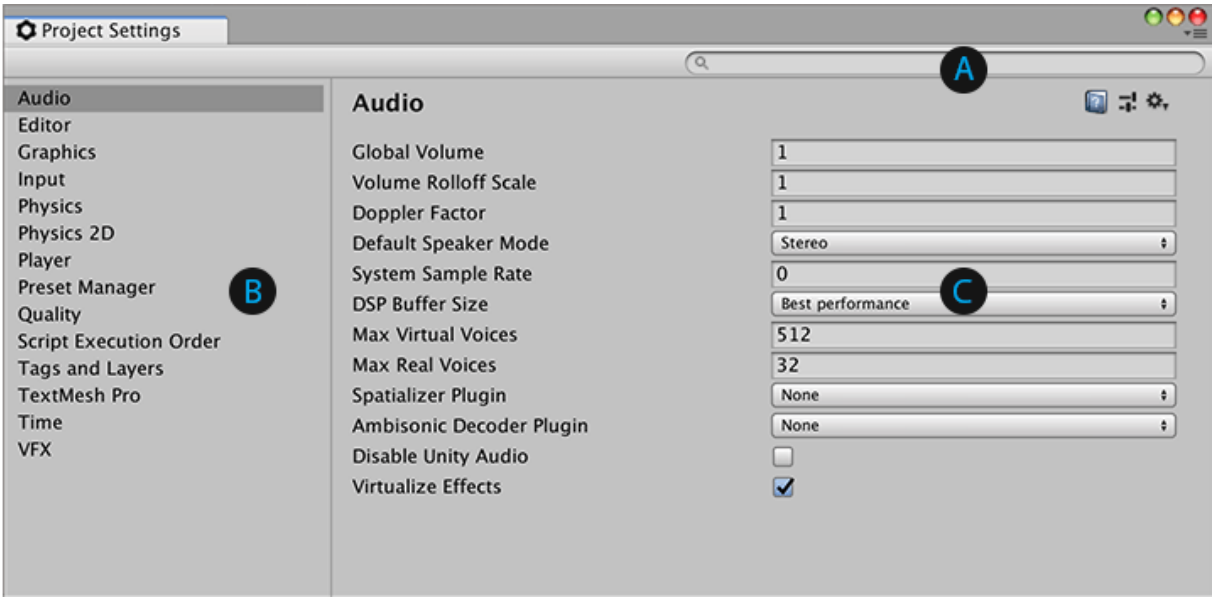


Slika 10. Prozor upravitelja paketima (engl. *Package Manager*)

Narančasti dio označuje popis paketa, ovisno o odabranom načinu prikaza. Trenutno je odabrani paket naziva „Barracuda“ čiji detalji su prikazani desno, u zelenom dijelu. Zeleni dio prikazuje između ostalih detalja i dostupnu verziju za preuzimanje, trenutno je to 1.0.0. Ukoliko je paket preuzet i nalazi se u projektu tada se ispisana verzija paketa odnosi na preuzetu verziju (ne na stvarnu najnovija verzija). U donjem desnom uglu, označeno crvenom bojom, nalaze se opcije za preuzimanje i instalaciju paketa (engl. *Install*) ili za uklanjanje paketa iz projekta (engl. *Remove*). Ukoliko je neki paket preuzet, u popisu paketa ima oznaku kao ikona označena žutom bojom, a ukoliko za njega postoji nova verzija, tada paket ima ikonu prikazanu unutar rozog okvira (paket „High Definition RP Config“).

2.3. Postavke projekta

„Postavke projekta“ (engl. *Project Settings*) je zapravo jedan od mogućih izbora u „Edit“ izborniku koji je već bio spomenut ranije. No zbog iznimne važnosti koju podešavanja postavki projekta ima na igru, ova opcija će biti detaljno opisana. Slijedi prikaz prozora koji prikazuje postavke projekta.



Slika 11. Postavke projekta (Unity Technologies, 2020h)

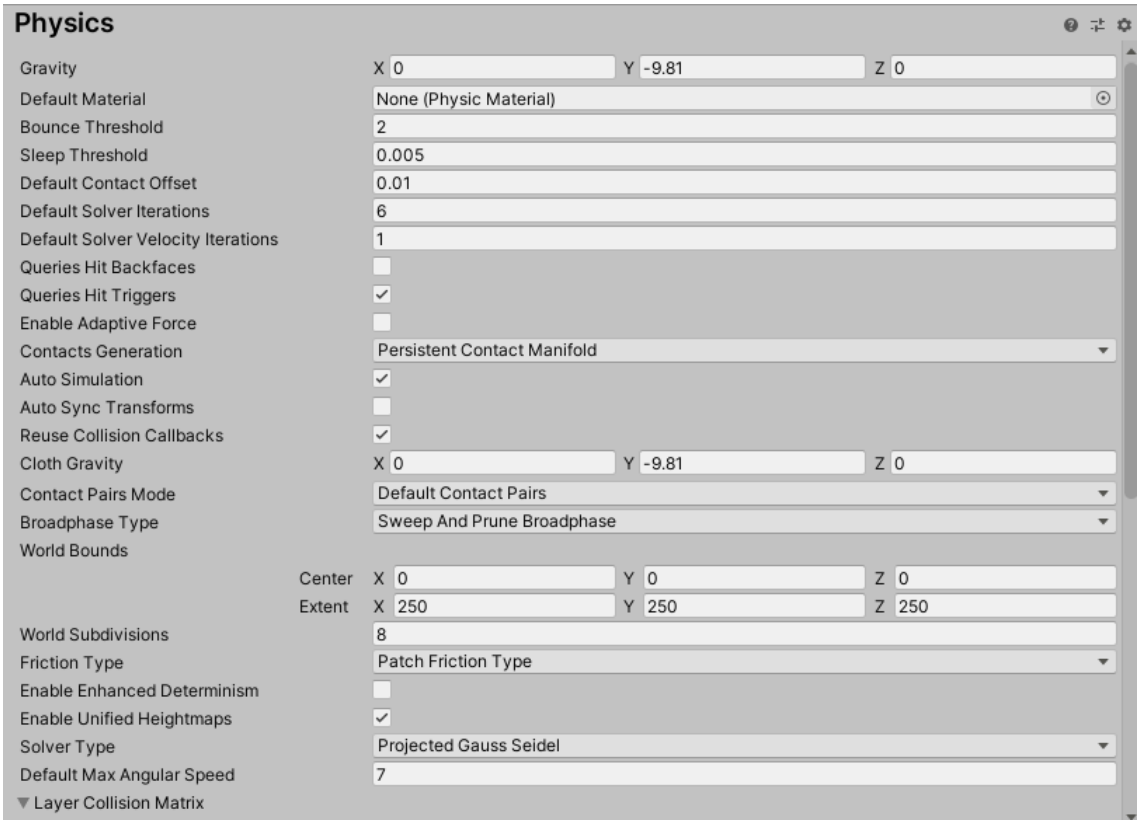
Prvim otvaranjem projektnih postavki pojavljuje se prozor kao na slici iznad, a sastoji se od tri osnovna dijela označena slovima (Unity Technologies, 2020h):

- A – Dio koji omogućuje pretraživanje kategorija postavki (B dio) i označavanje pojmova postavki C dijela prozora.
- B – Prikazuje grupirane postavke po kategorijama, ovisno o odabiru kategorije, na C dijelu se prikazuju dostupne postavke te kategorije (trenutno „Audio“ postavke).
- C – Prikazuje dostupne postavke odabrane kategorije te omogućuje njihovo uređivanje

Neke od najvažnijih postavki su možda postavke fizike (engl. *Physics*), kvalitete (engl. *Quality*), grafike (engl. *Graphics*) i korisničke kontrole (engl. *Input*). Slijedi opis postavki fizike i korisničkih kontrola jer se ove postavke najčešće koriste.

2.3.1. Postavke fizike

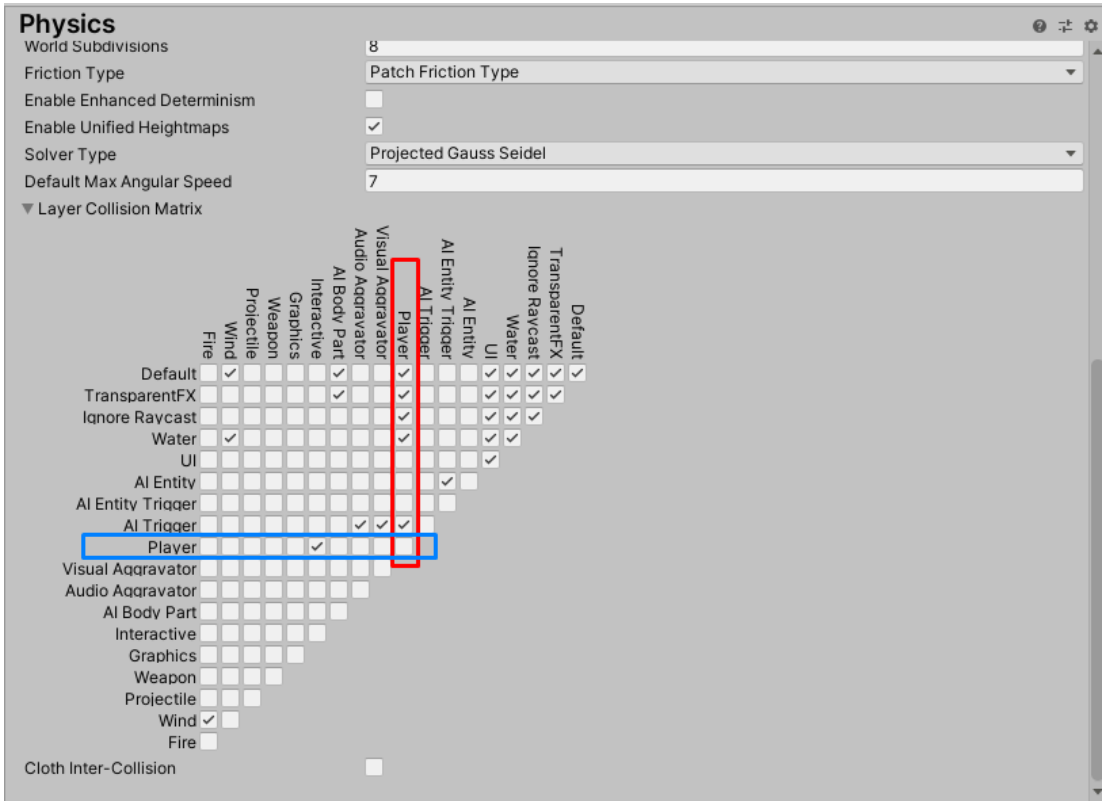
Prozor postavki fizike pruža mogućnost omogućavanja i onemogućavanja stvaranja kolizija između različitih slojeva, podešavanja gravitacijske sile i slično. Zbog jednostavnijeg shvaćanja podešavanja ovih postavki u nastavku slijede slike i njihovi opisi.



Slika 12. Postavke fizike 1. dio

U ovome dijelu najvažnija postavka je postavka gravitacije i nalazi se na samome vrhu. Kao što je i vidljivo na slici, gravitacija je trodimenzionalni vektor i djeluje konstantno na objekte koja sadrže komponente kao što je „Rigidbody“. Ukoliko radimo primjerice igru fantazije gdje je gravitacija možda manja u y-osi ili je čak pozitivna, ovdje nam mogu poslužiti i ostale dvije osi kako bismo simulirali neko gravitacijsko konstantno polje. Drugo polje, odnosno postavka materijala, omogućuje dodavanje materijala terena. Ovaj materijal nije običan materijal kakav se koristi za objekte da mu daju izgled, već služi za dodavanje fizičkih svojstava objektima, poput elastičnosti, odnosno mogućnosti odskakivanja od drugih objekata i sl.

Slijedi drugi dio postavki fizike koji predstavlja matricu kolizija između slojeva te je možda i važniji dio od maloprije navedenog.



Slika 13. Matrica kolizija slojeva (postavke fizike 2. dio)

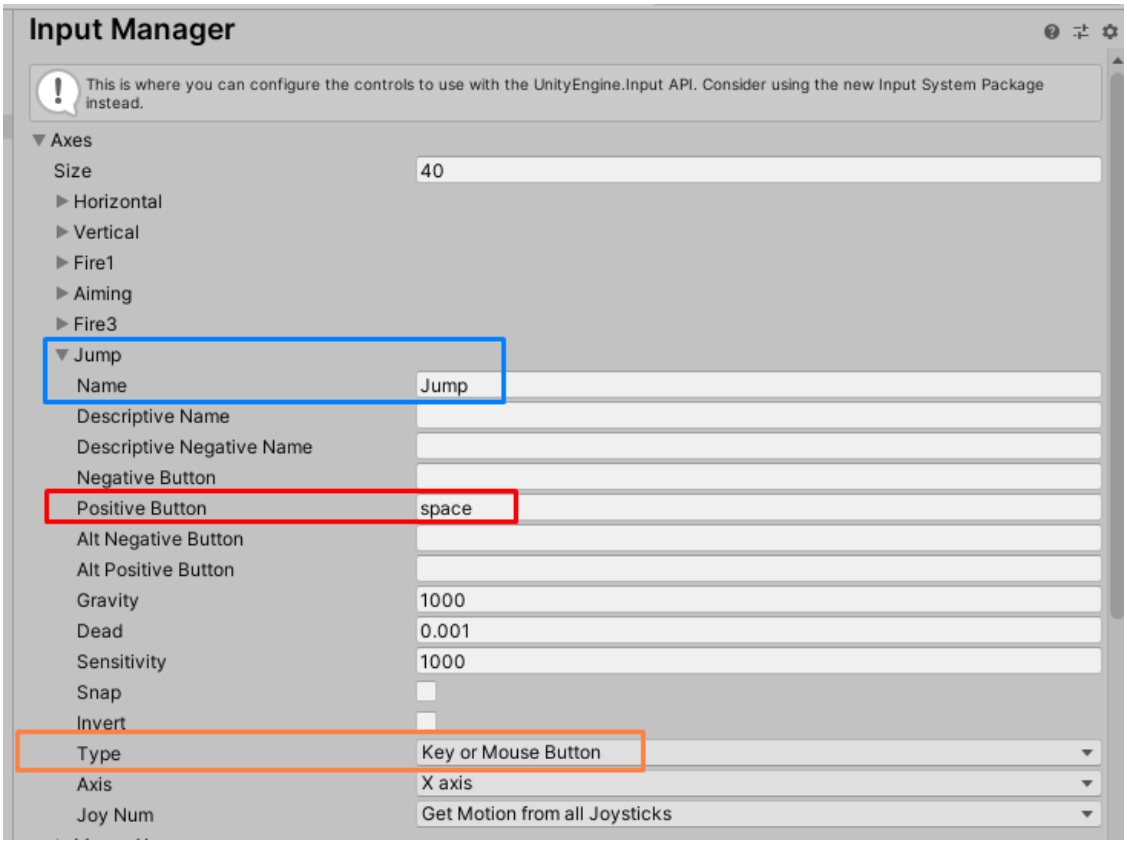
Matrica, kao na slici iznad, prikazuje sve slojeve i u horizontalnom i u vertikalnom smjeru na način da onaj sloj koji je u bilo kojem od ovih slojeva naveden prvi, u drugom je naveden zadnji. Ovim načinom se kreira matrica koja na kojoj se u dijagonali križaju slojevi istog imena (Unity Technologies, 2020e). Primjerice, mjesto križanja plavog i crvenog sloja je zapravo kolizija između dva ili više igrača te je trenutno ona isključena. Igrač (engl. *Player*) može generirati samo kolizije s četiri predefinirana sloja („Default“, „TransparentFX“, „Ignore Raycast“ i „Water“) te s interaktivnim (engl. *Interactive*) slojem i slojem „AI trigger“. Iz navedenog naravno slijedi i da ti navedeni slojevi mogu generirati kolizije sa slojem igrača.

2.3.2. Korisničke kontrole

Postavke korisničkog unosa (engl. *Input Manager)* omogućuju postavljanje vlastitih kontrola (engl. *Key bindings*, *Controls*) koje služe za upravljanje igrom. Tipične kontrole su „Horizontal“ i „Vertical“. Horizontalne kontrole su one koje najčešće služe za horizontalno kretanje u igri (lijevo – desno; tipke „a“ i „d“). Tipka koja služi za kretanje lijevo, kao rezultat pritiska daje negativan broj, a ona koja služi za kretanje desno daje pozitivan, a nula je

mirovanje. Što se tiče vertikalnog kretanja, sve je jednako kao kod horizontalnog, ali su tipke drukčije (naprijed – nazad; „w“ i „s“). Kod kontrola, gdje držanjem tipke želimo tijekom vremena dodavati pozitivnu ili negativnu vrijednost na rezultat držanja tipke, postavljamo osjetiljivost tipke (engl. *Sensitivity*) i gravitaciju (engl. Gravity) na niske vrijednosti (~3). Također je važno napomenuti mogućnost postavljanja opcije *snap* koja ukoliko je uključena, ako dođe do promjene u predznaku tipke (pozitivno -> negativno ili obrnuto), omogućuje odmah postavljanje rezultata na neutralnu poziciju te onda daljne napredovanje u željenom smijeru. Ukoliko je ova opcija isključena, tada se kod promjene predznaka rezultat pritiska (ili držanja) tipke postepeno približava neutrali (nuli) i nakon toga u određenom smjeru.

Slijedi slika nekih kreiranih korisničkih kontrola.



Slika 14. Korisničke kontrole

Prvo polje koje se može podešavati je broj postojećih kontrola (engl. *Size*). Ukoliko želimo dodati novu kontrolu, povećavamo broj kontroli s primjerice 40 na 41. Na slici su vidljiva i imena svake postojeće kontrole, trenutno je proširen prikaz kontrole skakanja (engl. *Jump*). Postavka skakanja je podešena tako da joj je dodano ime „Jump“ i na to ime se može u skripti referencirati, umjesto da se koristi sama tipka. Pozitivna tipka je razmaknica (engl. *Space Bar*).

Spomenuto referenciranje na kontrolu je vrlo važno jer pomoću referenciranja na skakanje programer ne ovisi o odabiru ili kasnijim promjenama tipki kontrola. Što se tiče tipa kontrole (engl. *Type*), označeno narančastom bojom, postoje opcije koje označuju radi li se o tipki miša ili tipkovnice, pokretu miša ili pak o osi komandne palice (engl. *Joystick*). Kod ove kontrole, osjetljivost i gravitacija su postavljeni na velike vrijednosti kako bi rezultat pritiska odmah porastao iz nule u maksimum.

2.4. Skriptiranje

Ono što igri u Unityju daje srž je programiranje. Programiranje u Unityju naziva se skriptiranje jer je bit ovog programiranja kreiranje kraćih skripti (komponenti) koje se dodaju objektima. Sve već ugrađene komponente („Rigidbody“, „Box Collider“, „Sphere Collider“, „Character Controller“ i druge) koje se mogu dodavati na objekte u igri su zapravo skripte. Često se ove komponente smatraju dodatnim i pomoćnim zatvorenim dijelovima i o njima se baš i ne razmišlja kao o skriptama i o tome kako rade u pozadini. „Iako se skriptiranje u Unityju odvija u C#, JavaScript ili Boo programskim jezicima pomoću tzv. Mono razvojnog okvira (verzija .NET okvira), Unity je sam po sebi zapravo pisan C++ jezikom“ (Brodkin, 2013). Prošla rečenica nije u potpunosti istinita u vrijeme pisanja ovog rada. Naime, u Unity Technologies navode kako je jedini preostali programski jezik za skriptiranje danas C#. Skripte koje se moraju odvijati vrlo brzo, kao kalkulacija fizike ili razne animacije, pisane su C++ programskim jezikom (Brodkin, 2013).

Kada se radi o skriptiranju u Unityju, važno je znati i napomenuti određene razlike između skriptiranja radi kreiranja ponašanja za objekte i drugih vrsta programiranja (kreiranja raznih aplikacija). Kod programiranja aplikacija (mobilne aplikacije, razne računalne aplikacije, i sl.) najčešće postoji jedna glavna datoteka koja sadrži komandu za pokretanje aplikacije te takva aplikacija ima početak i kraj i izvršava se u linearnom vremenskom slijedu. Ako se radi o skriptama koje objektima daju određeno ponašanje, u skriptama postoje komande koje se pokreću odmah kod inicijalizacije objekta na koji je skripta dodana, kod svake slike (engl. *Frame*) kontinuirano, netom nakon slike i sl. Dakle, ovakve skripte se kod pokretanja najčešće izvršavaju u petlji, kod svake slike ispočetka tako dugo dok objekt postoji.

Što se tiče otkrivanja pogrešaka (engl. *Debugging*) unutar skripti, jednako je kao kod otkrivanja pogrešaka bilo koje aplikacije. Potrebno je postaviti točku prekida (engl. *Breakpoint*), nakon što *game engine* kod izvršavanja linije po linije skripte dođe do točke prekida. Skripta se na toj liniji pauzira te je pauzirana dok korisnik ne omogući daljnje izvođenje (bilo samo do

sjedeće linije koda ili nastavak do sljedeće točke prekida). U međuvremenu dok je skripta pauzirana, developer može pregledavati razne varijable i njihove vrijednosti.

Slijedi prikaz jedne vrlo jednostavne skripte kako bismo na lakšem primjeru naučili osnovne stvari za shvaćanje većih i kompliciranijih skripti.

```
public class NewBehaviourScript : MonoBehaviour
{
    [SerializeField] private int _counter;
    void Awake(){
        _counter = 0;
    }
    void Start(){
        _counter=1;
    }
    void Update(){
        Debug.Log(_counter);
        _counter++;
    }
}
```

Ova skripta sadrži jednu klasu kojoj je dodijeljeno ime „NewBehaviourScript“ i ova klasa nasljeđuje klasu „MonoBehaviour“. Bilo koja skripta koja se dodaje na objekte mora naslijediti spomenutu klasu direktno ili indirektno (preko neke druge klase). „MonoBehaviour“ klasa pruža pristup ovim trima navedenim metodama „Awake“, „Start“, „Update“ (ali i mnogim drugim metodama) (Unity Technologies, 2020k).

Klasa unutar sebe sadrži i jednu globalnu, ali privatnu (engl. *Private*) varijablu „_counter“ koja može čuvati samo vrijednost nekog cijelog broja (engl. *Integer*). Privatnim varijablama ne može se pristupati iz drugih skripti, odnosno klasa. Također, privatne se varijable čak ni ne prikazuju u prozoru inspektora i prema tome ne može ih se ni uređivati na taj način. Ukoliko je neka globalna varijabla javna (engl. *Public*), tada joj se uvijek može pristupiti ukoliko postoji neka referenca na klasu koja je definira te je atribut moguće pregledavati i uređivati unutar Unityja u prozoru inspektora. Često postoji potreba da se neki atributi sakriju od svih ostalih dijelova igre, ali da se ipak prikažu u prozoru inspektora kako bi ih bilo lakše podesiti. Kod takvih situacija koristi se privatna ili zaštićena (engl. *Protected*) varijabla, ali se ispred nje dodaje oznaka „[SerializeField]“ (Unity Technologies, 2020l).

Spomenute tri metode izvršavaju se na način da se „Awake“ poziva prva, prije bilo kakvog drugog posla u skripti, drugim riječima, poziva se kod inicijalizacije skripte (kod

učitavanja). „Start“ metoda se poziva neposredno prije nego što se počne izvršavati prva slika (engl. *Frame*). Ova metoda se na jednoj instanci skripte poziva samo jednom. Ukoliko bi skripta na nekom objektu tijekom izvođenja postala deaktivirana te se nakon nekog vremena ponovno aktivirala, „Start“ i „Awake“ metode ne bi bile ponovno pozvane (kod takvih situacija se koriste „OnEnable“ i „OnDisable“). „Update“ metoda je najčešće i najvažniji dio skripte te se u ovome dijelu izvršava većina posla kojeg skripta obavlja. Ova metoda poziva se jednom po svakoj slici (engl. *Frame*). Postoje druge slične metode kao što je „Update“, primjerice „FixedUpdate“ metoda koja osigurava da će uvijek između neposredna dva poziva ove metode proći jednako vrijeme, obično 0.02 sekunde (Edit→Project Settings→Time). „FixedUpdate“ se najčešće koristi radi raznih kalkulacija koje su povezane s fizikom.

Kod učitavanja skripte, varijabla „_counter“ će se postaviti na nulu, osim ako to u prozoru inspektora nije promijenjeno (prozor inspektora nadjačava promjene vrijednosti varijable u „Awake“ metodi). Nakon toga, ali prije prvog poziva „Update“ metode, vrijednost varijable „_counter“ će se postaviti na vrijednost 1. Nakon toga, za svaku sliku u sekundi, vrijednost varijable će se povećati za jedan. Ukoliko se igra pokreće uz 100 slika u sekundi (engl *Frames per second*), vrijednost varijable će već nakon prve sekunde biti 101 te će se svaki put ispisati njezina vrijednost u prozor konzole. Često se u igrama, ako se nešto mora odvijati kroz vrijeme, primjerice ako je važno da se varijabla poveća svakih dvije sekunde za 20, koristi vrijednost „Time.deltaTime“. Spomenuta vrijednost je zapravo razlika u vremenu između početka prošlog poziva „Update“ metode i trenutnog poziva istog (Unity Technologies, 2020m).

Nadalje, važno se spomenuti i poslove (engl. *Coroutine,* u daljnjem tekstu korutina ili posao) u kojima se vrlo često koristi „Time.deltaTime“. Poslovi se mogu shvatiti kao metode (funkcije) te se i definiraju na isti način, ali imaju nekoliko različitosti. Metode (funkcije) moraju završiti u slici (engl. *Frame*) u kojoj su pozvane na izvršavanje, dok se poslovi (engl. *Coroutine*) mogu izvršavati kroz više slika (Unity Technologies, 2020c). Zbog toga se poslovi najčešće koriste kod zadataka koji se trebaju izvršavati kroz neko vrijeme, a brojač tog vremena je neka varijabla kojoj se dodaje vrijednost „Time.deltaTime“. Sljedeća različitost je ta što tip metode posla (engl. *Coroutine*) mora biti IEnumerator, dok bilo koje druge može biti „void“ ili bilo koji drugi tip podataka. Poslovi se pozivaju pomoću „StartCoroutine(coroutine)“ te se, ukoliko je potrebno, mogu se i prisilno zaustaviti pomoću „StopCoroutine(coroutine)“.

3. Žanr i tip igre

Cilj ovog poglavlja je naučiti što je to žanr videoigre te se fokusirati na avanturistički žanr i avanturističke igre, jer igra kreirana u svrhu ovog rada pripada navedenoj skupini. Također u radu će se često spominjati tip, a tip videoigre je zapravo podtip žanra te će iz primjera i opisa će to postati intuitivno i jasno.

3.1. Popis i opis žanrova

Slijedi popis žanrova, kratki opis i glavni predstavnici žanra te nakon toga detaljnija razrada avanturističkih igara. Svaki žanr sadrži još nekoliko tipova igara unutar sebe, zbog toga će najčešće biti spomenuti samo osnovni žanrovi ili najpoznatiji tipovi tog žanra.

Žanr	Opis	Predstavnici
Akcija (engl. *Action*)	Akcijske igre su kreirane na način da od igrača zahtijevaju koordinaciju ruku i očiju, motoričke sposobnosti i reflekse. Najpoznatiji tipovi akcijskih igara su definitivno pucačina (engl. *Shooter game*), igra šuljanja (engl. *Stealth game*), borilačke igre (engl. *Fighting games*) i vrlo popularne u zadnje vrijeme *Battle royale*.	Doom, Super Mario Bros, Guitar Hero, Call of Duty
Akcija-avantura (engl. *Action-adventure*)	Igre ovog žanra imaju fokus na istraživanju i često uključuju skupljanje raznih objekata, uključuju rješavanje problema, ali i elemente akcije poput borbe. Jedan od tipova koji ovdje spada je horor preživljavanje (engl. *Survival horror*).	The Legend of Zelda, Final Fantasy, Resident Evil
Avantura (engl. *Adventure*)	Ovaj žanr je jedan od najranije kreiranih igara, a prve igre su bile tekstualne avanture (engl. *Text adventure*). Ovakve igre nisu definirane pričom ili sadržajem, nego umjesto toga pružaju način igranja bez izazova kakve pružaju akcijske igre. Ovakve igre najčešće zahtijevaju	Myst, Colossal Cave Adventure, Life is Strange

	od igrača da rješava zagonetke i probleme u interakciji s okolinom i ljudima.	
Igra uloga (engl. *Role-playing game*)	Većina ovakvih igara stavlja igrača u položaj jednog ili više pustolova (engl. *Adventurer*) koji se specijalizira u specifičnom setu vještina dok napreduju kroz priču. Najpoznatiji tip ovog žanra je definitivno MMORPG (engl. *Massively multiplayer online role-playing game*), ovdje još pripadaju akcijski RPG (engl. *Action RPG*), RPG otvorenog svijeta (engl. *Open Worlk RPG; Sandbox RPG*).	World of Warcraft, The Elder Scrolls, Fallout, The Lord of the Rings Online
Simulacija (engl. *Simulation*)	Igre kreirane u ovom žanru kreirane su na način da se što više poklapaju sa stvarnim svijetom i na taj način simuliraju aspekte stvarne ili fikcijske realnosti. Postoje razni tipovi ovakvih igara, od simulatora građenja građevina do simulatora života likova i simulatora vozila (zrakoplovi, automobili, podmornice, i sl.)	The Sims, Euro Truck Simulator, Farming Simulator, Gran Turismo
Strategija (engl. *Strategy*)	Baziraju se na opreznom donošenju odluka, razmišljanju o mogućim potezima i strategiji kako bi se osigurala pobjeda. Ovakve igre mogu biti u realnom vremenu (engl. *Real-time*), gdje svi igrači donose odluke i upravljaju igrom u isto vrijeme ili igre bazirane na potezima (engl. *Turn-based*), gdje svaki igrač ima svoj potez određenog trajanja.	Civilization, Rise of Nations, Pocket Tanks, Worms, League of Legends, Dota 2, Age of Empires
Idle	*Idle* prevedeno s engleskog doslovno ima značenje „bez posla“. Ovakve igre se najčešće igraju same od sebe, računalo ili neka druga platforma odradi gotovo sav posao. Dakle ove igru pružaju slabu i gotovo nikakvu interakciju s igračem, pružaju igraču trivijalne zadatke gdje	Cookie Clicker, Adventure Capitalist

	se s nekoliko klikova završava zadatak i osvaja nagrada.	
Ostalo	Neke od ostalih žanrova koji su preostali su sportske igre, kartaške i igre na ploči (engl. *Bord games*), horor igre, kvizovi, edukacijske igre.	Are You Smarter than a 5th Grader?, Yu-Gi-Oh, Catan

Tablica 1. Popis žanrova, opis i igre predstavnici žanra (Wikipedia, 2020)

Kao što možemo vidjeti iz tablice, žanrova ima puno, a kada bi se nabrajali i svi tipovi pojedinog žanra, popis bi bilo ogroman. Razlog tome je što se sve češće u nekoj igri kombinira sve više raznih žanrova i tipova te na taj način nastaju novi tipovi.

3.2. Avanturistička igra

Sam žanr avanturističke igre već je ukratko opisan u tablici iznad, no u ovome poglavlju će biti dodatno razrađen kako bismo mogli usporediti žanr s igrom kreiranom u svrhu ovog rada. U ovom poglavlju će biti opisane mehanike avanturističkih igara općenito i nekih pojedinih istaknutijih.

Karhulahti (2011.) u svojem radu navodi skraćenu definiciju avanturističke igre i u suštini govori da je avanturistička igra ovisna o priči jer omogućuje igraču napredovanje kroz strukturu narativnih događaja. Dalje navodi da se izazovi u ovakvim igrama javljaju u obliku problemskih zagonetki, a ne u obliku aktivnih agenata (ljudskih ili računalnih) protiv kojih bi se igrač natjecao. Dakle, u avanturističkim videoigrama, problemske zagonetke se javljaju kao primarna mehanika igre, dok se u drugim žanrovima ne javljaju uopće ili se javljaju rijetko kao sekundarna mehanika.

Karhulahti (2011.) u svom radu također navodi da nisu sve videoigre kojima je glavna ili čak jedina mehanika rješavanje problema ili zagonetki avanturističke igre. Primjer ovakve igre je popularna igra „Tetris“ i ne može je se opisati kao avanturističku. Zbog tog problema, potrebno je imati bolju definiciju avanturističke videoigre koja detaljnije opisuje žanr. Jedna takva potpuna definicija glasi: „Avanturističke igre su igre vođene pričom koja potiče istraživanje i rješavanje zagonetki, mora sadržavati barem jednog lika te je osnovna interakcija s okolinom bazirana na interakciji s objektima i prostornom navigacijom.“ (Karhulahti, 2011).

Neke poznatije avanturističke igre su „Colossal Cave Adventure“ koja je prva igra definirana kao avanturistička, kao jedna od najrevolucionarnijih je igra „Myst“, a igra „The Talos Principle“ je poznata igra domaće proizvodnje koja je doživjela veliku popularnost. Što se tiče

„Colossal Cave Adventure“, razvijena je između 1975. i 1977. godine te umjesto da ponudi loše grafičko iskustvo kakvo je tada bilo dostupno, kreatori se odlučuju na kreiranje tekstualne igre. Igrač upravlja protagonistom na temelju jednostavnih tekstualnih naredbi, a opis događaja dobiva kao tekst u knjizi (Wikipedia, bez dat.-a). Sljedeća spomenuta igra je „Myst“, za razliku od prošle opisane igre, ova je donijela jednu vrstu revolucije grafičkim avanturama (Wikipedia, bez dat.-b). Ova igra je najviše fokusirana na rješavanje problemskih zadataka i putovanja po otoku Myst. Igra je postigla ogromnu popularnost na tržištu i postala je najprodavanija igra do 2002. godine (Wikipedia, bez dat.-b). Posljednja spomenuta igra „The Talos Principle“ razvijena je od strane hrvatske tvrtke „Croteam“. Ova igra je filozofsko remek djelo, ako je igrač razumije. Igra se iz prvog lica, duboko fokusirana na rješavanju kompleksnih problemskih zadataka i donošenju odluka koje se reflektiraju na igru (Croteam, 2014).

4. Računalna igra

U ovome poglavlju će fokus biti na kreiranoj igri. U prvome dijelu slijedi opis igre i priče koju igra posjeduje, glavne mehanike igre i agenata te opis istih.

Što se tiče same igre naziv joj je „Frequency and Energy“, a kreirana je u prvom licu, dakle igrač gleda svijet kroz oči protagonista. Avanturistički dio je postignut mističnim dijelovima igre kao što su zvukovi, misteriozni događaji, rješavanje problemskih zadataka i mogućnost istraživanja okoline u kojima se kriju tajne. Priča je zamišljena na način da protagonist predstavlja čovjeka koji si postavlja zadatak otkriti tajne energije i frekvencija te u svom snu kreće na put. Protagonist postaje svjestan da sanja i otkriva da se nalazi u šumi, tijekom svog lutanja, prate ga zvukovi zanimljivih frekvencija. Nakon kraćeg lutanja, protagonist prolazi kroz vrata i ulazi u podzemni tajni bunker, ali se vrata iznenadno zaglave. Unutar tajnog bunkera nalazi se tajanstveni energetski kristal koji sadrži energiju u obliku elektriciteta i frekvenciju kao zvuk. Protagonist odlučuje uzeti komad kristala kako bi ga proučio, no uslijed piezoelektričnog efekta kod diranja kristala, oslobađaju se ogromne količine energije. Ove energije i frekvencije probude određene uspavane ljude koji se nalaze skriveni unutar bunkera, te su zbog naglog i nespremnog buđenja predstavljeni kao zombiji. Protagonistu je jedini način izlaska iz bunkera koristeći dizalo, no da bi to bilo moguće, potrebno je riješiti niz problemskih zadataka.

Što se tiče izrade igre, kreirana je uz veliku pomoć Udemy tečaja, zapravo cijeli sustav umjetne inteligencije kreiran je pomoću tečaja. Također, što se tiče scene u tajnom bunkeru, veliki dio geometrije je već bio preuzet gotov te je postavljanje i rješavanje zagonetki kreirano pomoću tečaja. Ostale scene te sve što ima veze sa prikupljanjem kristala je kreirano samostalno. Unity projekt je kreiran pomoću HDRP (engl. *High Definition Render Pipeline*) sustava, u skladu s time je pomoću kupljenih resursa „NatureManifacture Assets“ kreirana scena šume i scena glavnog izbornik.

4.1. Sustav umjetne inteligencije

Umjetna inteligencija kod neigračih likova (NPC – engl. *Non-player character*) u igrama je od iznimne važnosti, pogotovo ako se radi o neprijateljskom liku. To je zbog samog pridonošenja uvjerljivosti igri, ukoliko igrač ima osjećaj da njegove akcije pridonose promjenama unutar igre, igra je uvjerljivija. Ovakve likove će se u daljnjem tekstu oslovljavati kao agenti jer se u kontekstu umjetne inteligencije često na taj način oslovljavaju subjekti

podvrgnuti umjetnoj inteligenciji. U igri kreiranoj u svrhu ovog rada, agenti nemaju umjetnu inteligenciju (AI – engl. *Artificial intelligence*) u modernom smislu ovog pojma. AI agenti u modernom smislu uče na svojim greškama i iskustvima, a u ovoj igri se promatrani agent uvijek ponaša isto, na temelju skripti koje im daju ponašanje i parametrima skripti. Što se tiče stvarnog učenja ovakvih agenata u Unityju, postoje neki sustavi koji omogućavaju kreiranje tzv. cjevovoda za učenje (engl. *Learning pipeline*), jedan od takvih sustava je „ML-Agents Toolkit“ (Juliani i ostali, 2018).

Sustav umjetne inteligencije sastoji se od agenata i njihovog ponašanja (što sve i kako utječe na odluke agenta) te od sustava navigacije za te agente (definicija terena po kojem se agenti mogu kretati i rute koje obilaze).

Često će se osim opisa agenata, skripti i navigacije iz završene igre, opisivati pojednostavljeni modeli istih. Primjerice, umjesto agenata iz stvarne igre (likovi i njihove animacije), pojavljivati će se jednostavni prikazi tih agenata kao jednostavni cilindri s jednostavnim kretanjem (bez ikakvih animacija) kroz navigacijsku plohu.

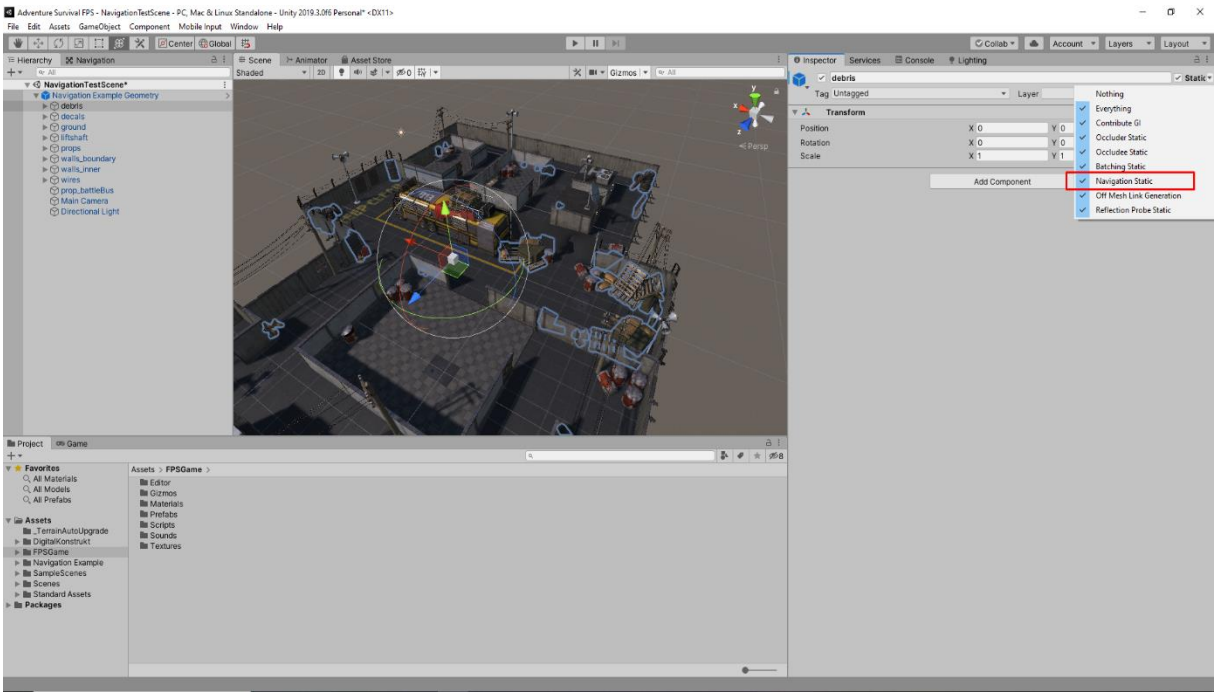
4.1.1. Sustav navigacije

Kako bi igra bila zanimljivija i teža, agenti se kroz scene kreću po zadanim rutama, neki agenti se kreću nasumično kroz rutu, a neki točno definiranim redom. Ovako složen sustav također pruža i dozu prosuđivanja i tjera igrača na promatranja kretnji agenata radi lakšeg i neprimjetnog prolaska do cilja.

Unity ima već implementiran sustav navigacije koji je samo potrebno podesiti i s jednostavnim agentima radi bez ikakvog problema. Da bi sustav navigacije radio u potpunosti, potrebna je navigacijska ploha po kojoj se agenti mogu kretati, potrebno je kreirati točke koje kreiraju rutu za kretanje po plohi, a prepreke na plohi su opcionalne.

4.1.1.1. Navigacijska ploha

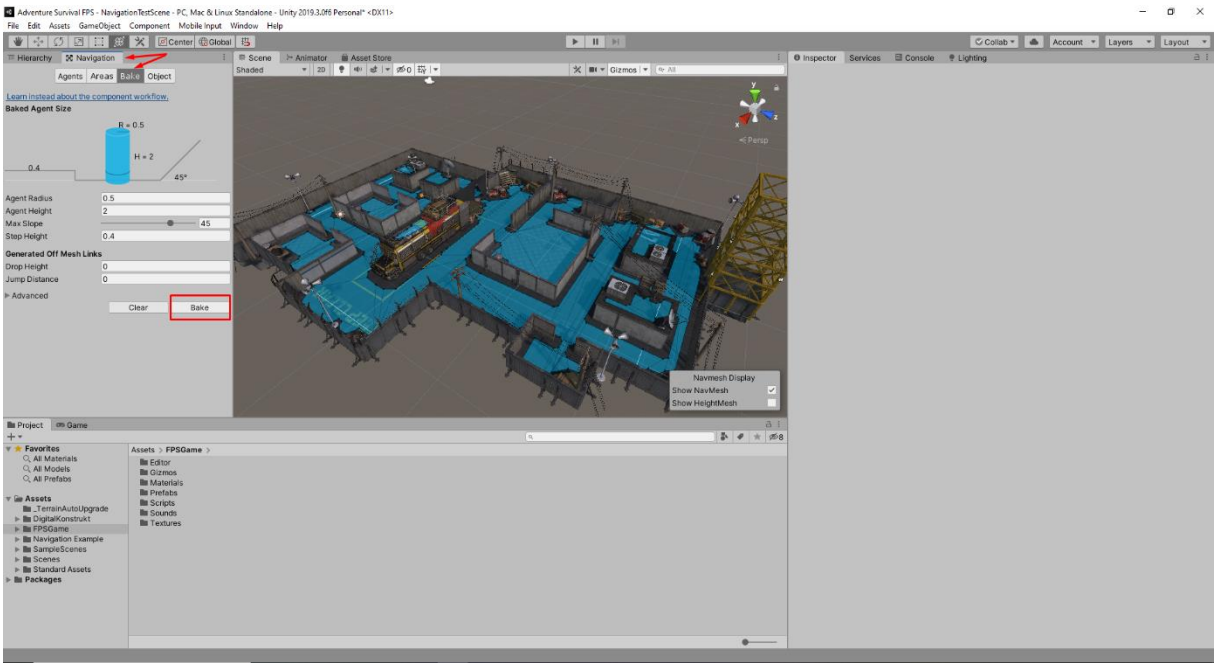
Kada se govori o navigacijskoj plohi, tzv. „NavMesh“, misli se na sav onaj prostor na sceni po kojem se agent može kretati. Navigacijska ploha se generira automatski prema parametrima agenta (visini, širini i drugim parametrima) i prema drugim objektima (eventualnim preprekama) koji se nalaze u sceni (na budućoj navigacijskoj plohi). Slijede slike kreiranja navigacijske plohe, opis postupka kreiranja te nakon toga slijede slike koje prikazuju detaljnije navigacijsku plohu i njene karakteristike.



Slika 15. Scena bez navigacijske plohe

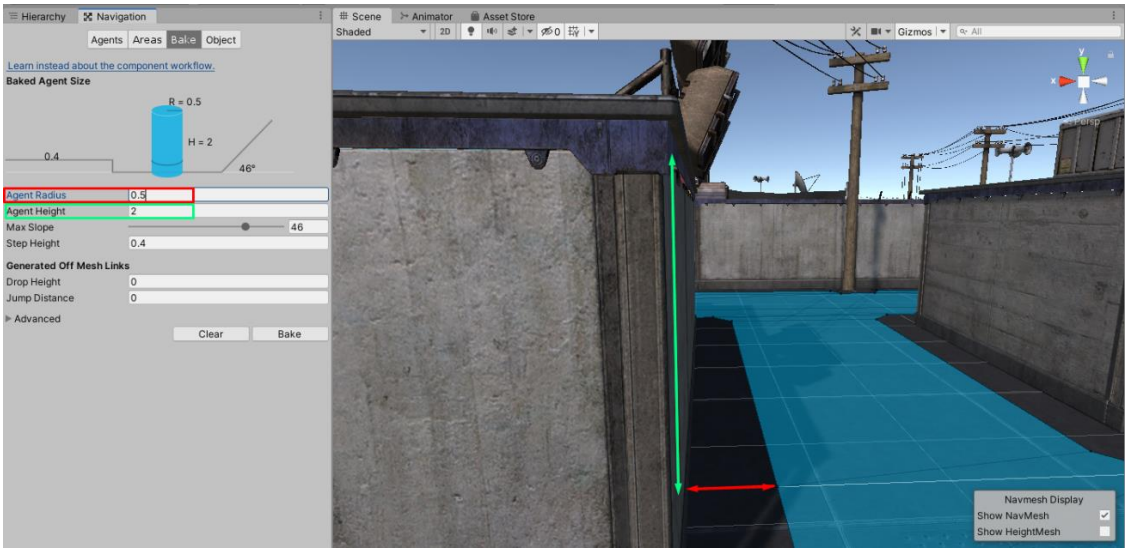
Slika iznad (Slika 15) prikazuje scenu koja će poslužiti kao primjer za kreiranje navigacijske plohe. Ova scena koja je vidljiva na slici ne sadrži navigacijsku plohu te je potrebno odabrati objekte, ukoliko ih ima, koji će biti ucrtani u istu. Objekte koje želimo ucrtati u plohu (u ovom primjeru je to grupa objekata „debris“) moramo označiti kao „Navigation Static“. Ovakvo označavanje objekata je prikazano na slici iznad (označeno crvenom bojom). Tako označeni objekti predstavljaju statične prepreke kroz koje agenti ne mogu prolaziti. Proces kreiranja navigacijske plohe, odnosno ucrtavanja objekata u plohu naziva se „NavMesh Baking“ (Unity Technologies, 2020a) te slijedi slika postupka.

Postupak kreiranja navigacijske plohe je vrlo jednostavan, prvo je potrebno otvoriti „Navigation“ prozor, ukoliko se već ne nalazi u razvojnom okruženju, može ga se uključiti odabirom izbornika „Window“, odabirom opcije „AI“ te nakon toga opcija „Navigation“. Nakon toga se jednostavno odabere „Bake“ grupa opcija kako prikazuju crvene strelice na slici ispod (Slika 16), a nakon toga se odabere opcija „Bake“ u donjem desnom uglu (označeno crvenom bojom). Nakon što se proces završi, dobivamo plavo označenu plohu koja predstavlja prohodni teren za agente. Na slici je vidljivo kako su primjerice krovovi zgrada označeni kao prohodni, iako agenti nemaju prohodan put do krovova, no više o tome kasnije.



Slika 16. Kreiranje navigacijske plohe

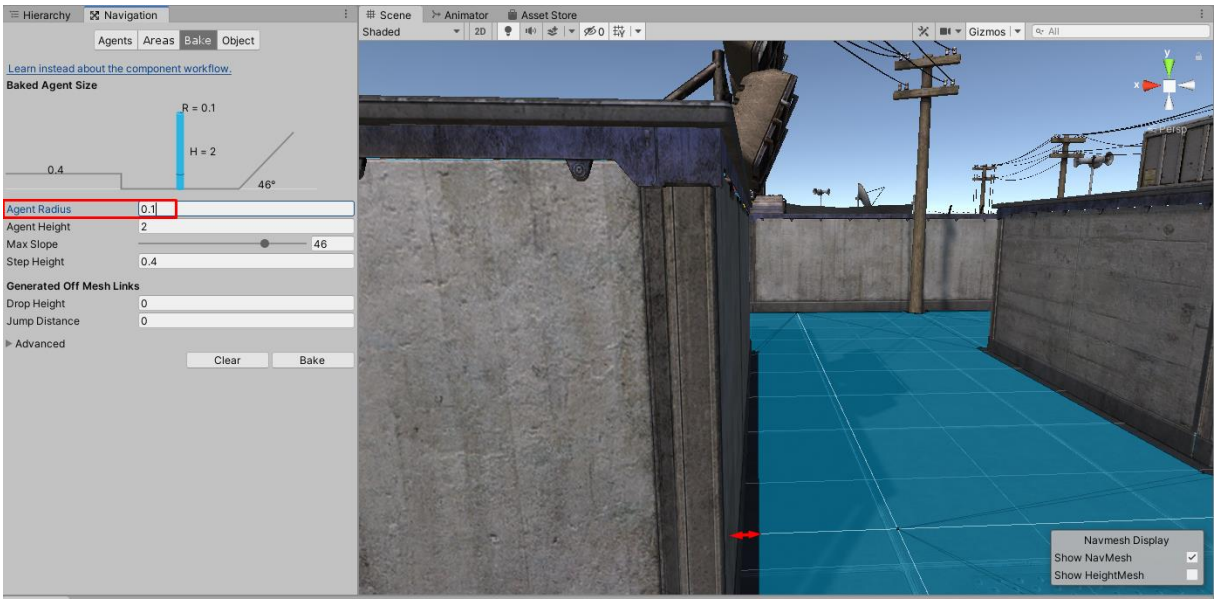
Slijedi niz slika koje prikazuju navigacijsku plohu i neprohodan teren iz drugog ugla te prikazuju kako dimenzije agenta utječu na prohodnost plohe.



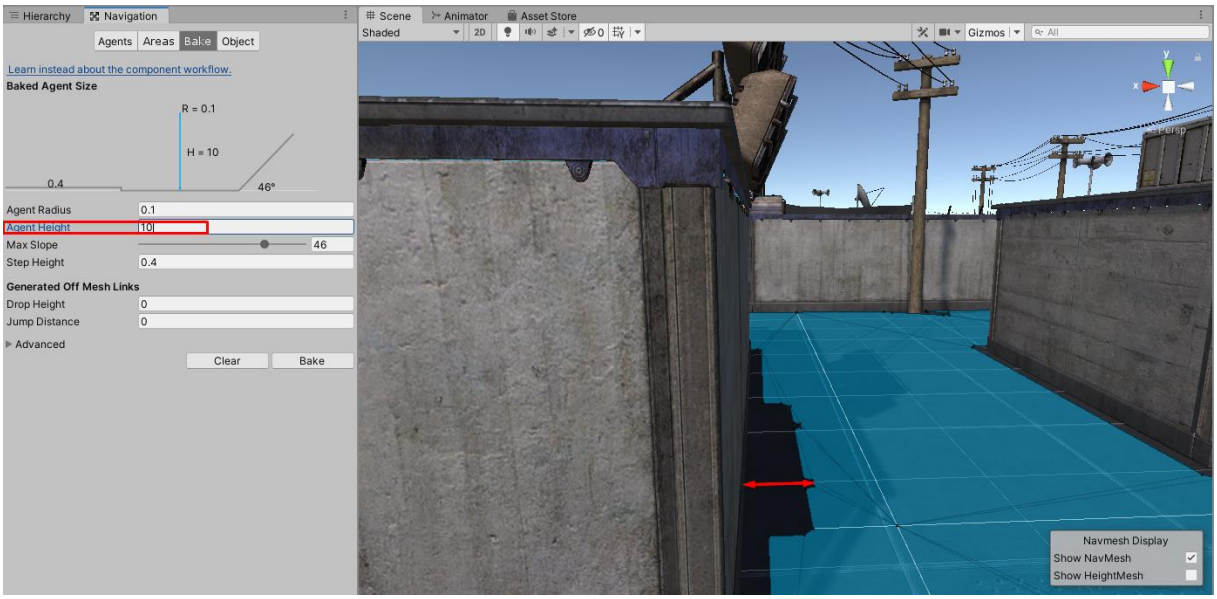
Slika 17. Originalne postavke agenta i odgovarajuća ploha

Zelena strelica prikazuje visinu agenta koji će se kretati plohom, a crvena širinu tih agenata. Visina agenta iznosi dvije mjerne jedinice, a širina pola jedinice. Uz ove postavke vidimo koliki

udio plohe je prohodan (plava ploha), a koliki dio je neprohodan (neobojani dio plohe). Ukoliko promijenimo radijus i visinu agenta, plohe izgledaju kao na slikama ispod.



Slika 18. Smanjeni radijus agenta i odgovarajuća ploha



Slika 19. Povećanje visine agenta i odgovarajuća ploha

Slike iznad (Slika 18 i Slika 19) prikazane su iz iste perspektive kao što to prikazuje Slika 17, ali su se promijenile postavke agenata, prema tome promijenjena je i sama navigacijska ploha. Na slici koja prikazuje smanjenje radijusa (Slika 18), radijus je postavljen na desetinu mjerne jedinice, dok je visina i dalje dvije mjerne jedinice, kao i u originalnim postavkama, no možemo vidjeti da se je udio neprohodne plohe drastično smanjio. No, na slici koja prikazuje povećanje

visine agenta na deset jedinica (Slika 19), ali uz ostavljanje radijusa na desetini mjerne jedinice, promjena navigacijske plohe je ponovno drastična. Zbog visine agenta, Unity je izračunao da agent ne može s tom visinom stajati ispod reflektora koji se nalaze na krovu kuće, kao ni ispod ruba krova. Prema tim kalkulacijama, iako je radijus jednak kao na slici prije (Slika 18), neprohodni dio plohe je veći. Radijus koji se spominje zapravo govori o tome kolika je najmanja moguća udaljenost od centra agenta do prepreke (Unity Technologies, 2020a). Ukoliko je stvarni radijus agenta (tijelo agenta) jedna mjerna jedinica, a u postavkama navigacije je postavljeno da je radijus agenta pola jedinice, tada to znači da agent može sa približno 50% svog tijela proći kroz prepreku. Zbog toga je uvijek bolje voditi se logikom da je radijus agenta kod navigacije nešto veći od stvarnog radijusa modela. Što se tiče visine agenta kod navigacija, definicija govori da je to najniža visina koja je potrebna agentu da prođe ispod prepreke (Unity Technologies, 2020a). Dakle, ukoliko se reflektor sa slike nalazi na tri mjerne jedinice, a visina agenta kod navigacije je pet mjernih jedinica, tada to znači da agent ne može proći ispod prepreka koje su niže od pet jedinica, odnosno ne može proći ispod reflektora.

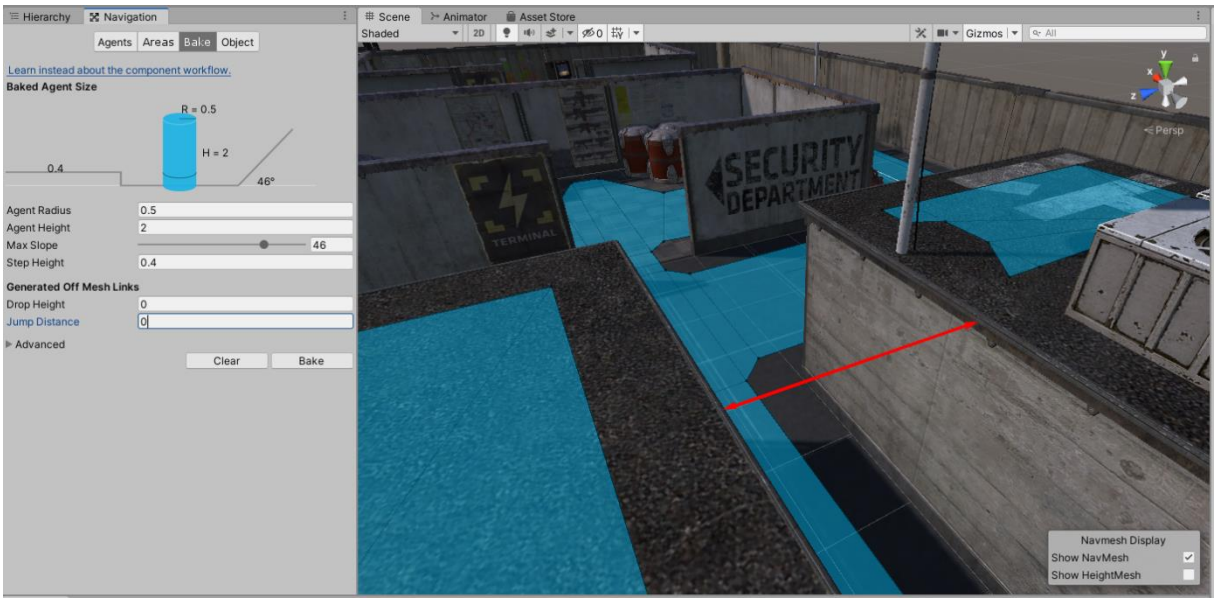
Da rezimiramo, postoji mogućnost podešavanja dimenzija svih agenata kao jedna univerzalna dimenzija za sve agente (bez obzira o stvarnim dimenzijama agenata). Nadalje, podešavanjem ove dimenzije se za sve agente predefinira i kreira navigacijska ploha koja je jedinstvena svim agentima te nema smisla da se za svakog agenta kreira njemu specifična ploha (svi agenti se kreću po jednoj plohi). Na ovaj način se definira zaobilaženja statične geometrije koja se nalazi na sceni (Unity Technologies, 2020g).

Postoji još nekoliko zanimljivih opcija, odnosno parametara koje se može podesiti kako bismo što više prilagodili navigacijsku plohu našim potrebama. Jedan od tih parametara je maksimalni nagib (engl. *Max Slope*), ovaj parametar određuje pod kojim najvećim kutom se agent može kretati. Primjerice, ukoliko je parametar podešen na 45°, tada agent ne može pristupati plohama gdje su kutovi veći od navedenog. Slijedi opis ostalih parametara koji su direktno povezani sa spomenutim prekidima veze jednog prohodnog dijela plohe s ostatkom plohe (krov kuće).

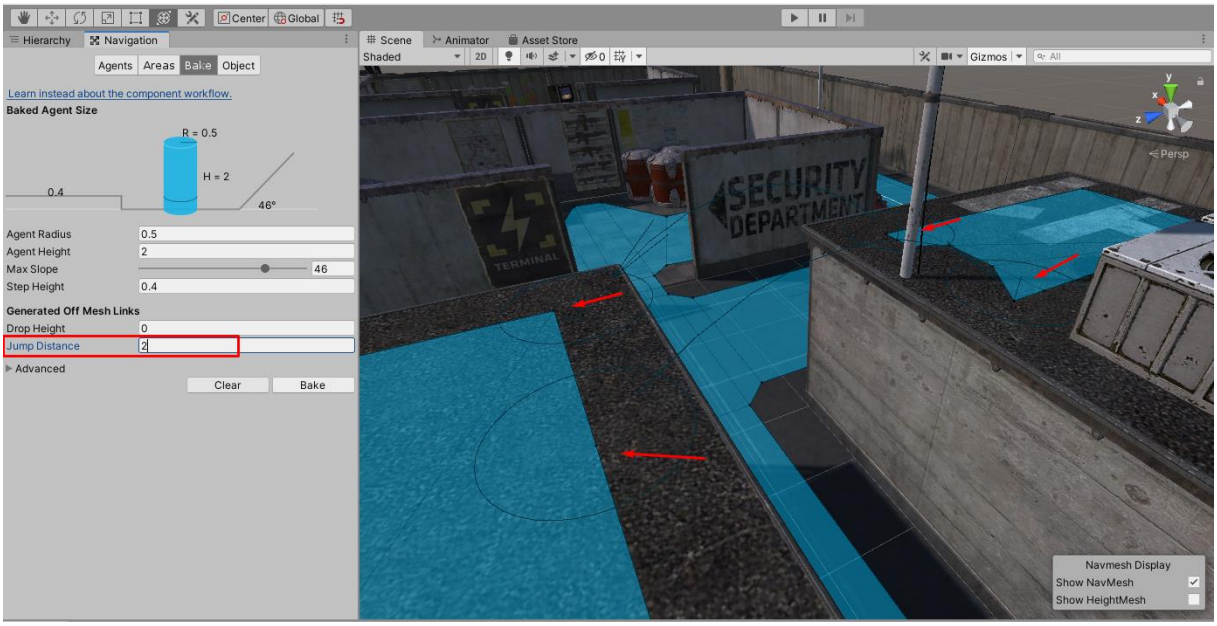
Kao što prikazuje slika koja slijedi (Slika 20), postoji razmak između dva krova kuće te ne postoji način da se s jednog krova pristupi drugom jer ne postoji prohodna ploha između njih. Na sreću, postoji opcija koja određuje maksimalnu daljinu skoka (engl. *Jump Distance*) agenata.

Nakon navedene slike, slijedi slika koja prikazuje podešavanje spomenutog parametra (Slika 21). Ukoliko postavimo duljinu skoka na dvije mjerne jedinice, kreiraju se ulazne i izlazne točke za agente, kao što je to i na slici prikazano. Točke mogu biti ili ulazne ili izlazne (tako da

dvije točke tvore par), ukoliko agent pristupi ulaznoj točki, tada se on teleportira na njezin par, tj. izlaznu točku. Postoji i opcija za podešavanje maksimalne visine pada (engl. *Drop Height*) te je identična kao i maloprije opisana opcija duljine skoka, ali služi za određivanje ulaznih i izlaznih točki s visine prema dolje (ne skok na povišeno).

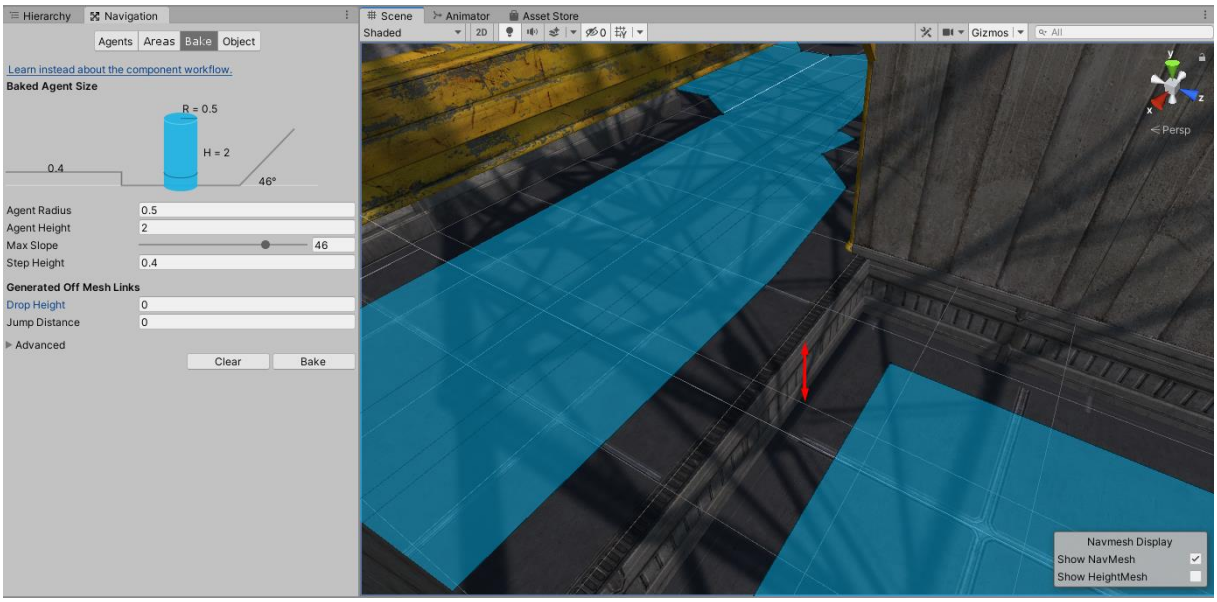


Slika 20. Nepovezani dijelovi navigacijske plohe



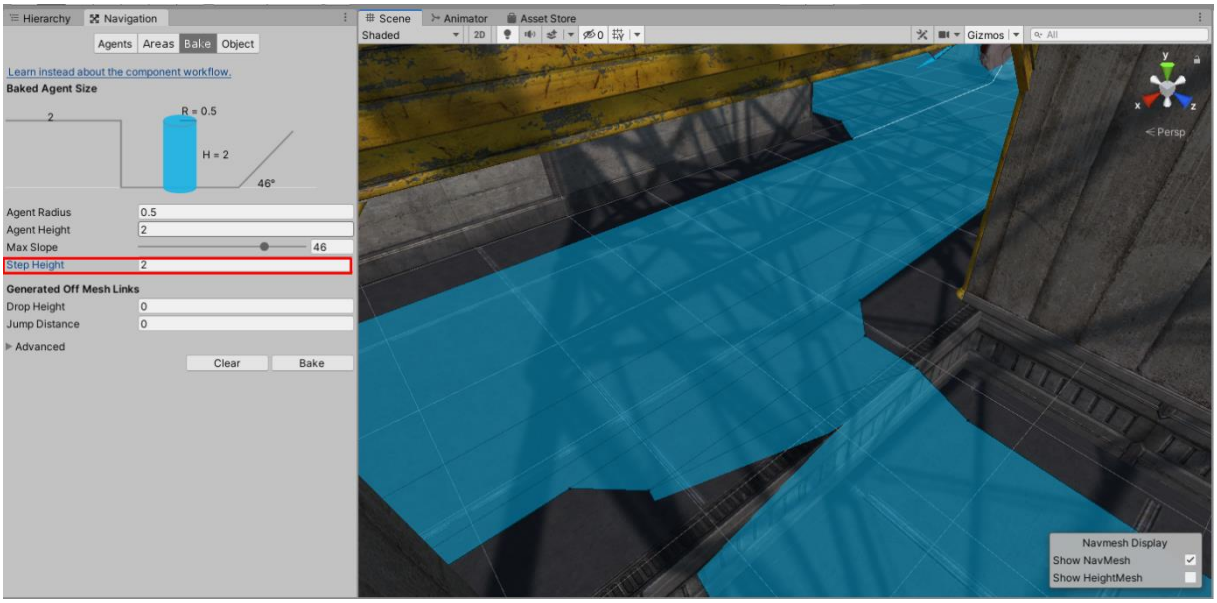
Slika 21. Duljina skoka agenta

Zadnja opcija koja će biti opisana je visina koraka (engl. *Step Height*) i određuje na koju maksimalnu visinu agent može zakoračiti (Unity Technologies, 2020a). Slijedi primjer nepovezane plohe te nakon toga i slika istog terena, ali uz podešenu visinu koraka.



Slika 22. Nepovezane plohe bez visine koraka

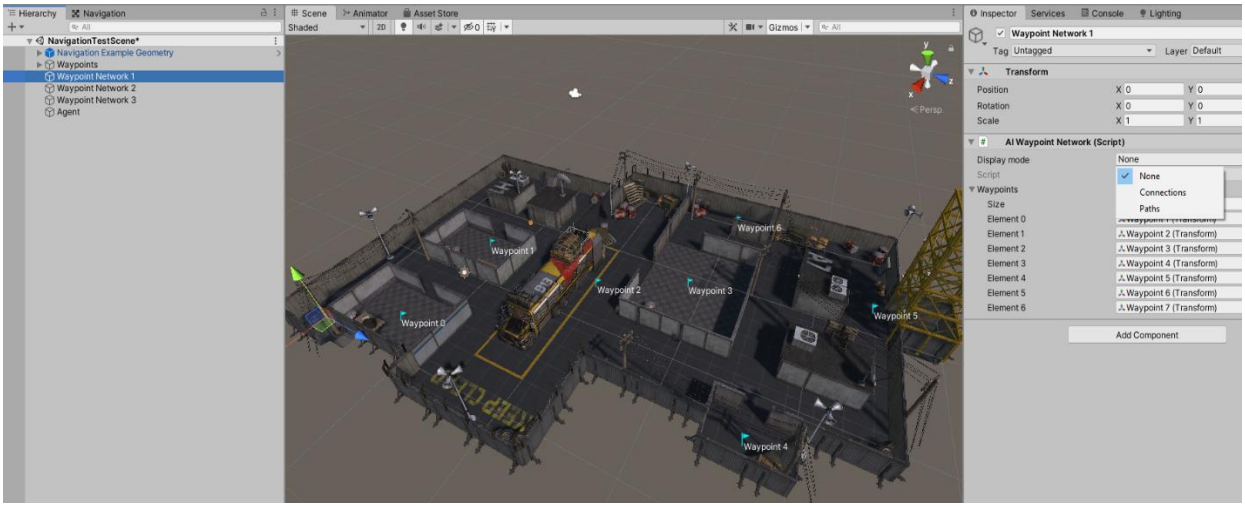
Kao što je vidljivo na slici iznad, postoji jasan prekid između dvije prohodne plohe (plavo označeno) zbog toga što postoji visoka stepenica koju agent ne može prijeći. Ovaj problem može se riješiti na tri načina, podešavanjem maksimalne visine pada, podešavanjem dužine koraka ili ručnim postavljanjem ulaznih i izlaznih točki. Slijedi izgled plohe nakon podešavanja visine koraka.



Slika 23. Plohe nakon podešavanja visine koraka

4.1.1.2. Točke rute

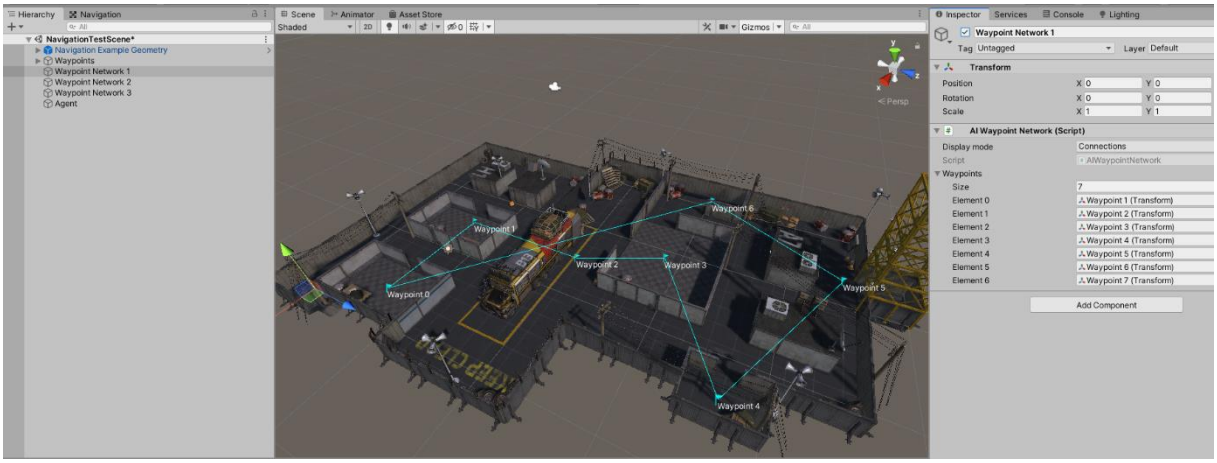
Ruta po kojoj se agenti kreću sastoji se od točaka rute kao što je to prikazano na slici koja slijedi.



Slika 24. Skup točaka rute koje čine rutu

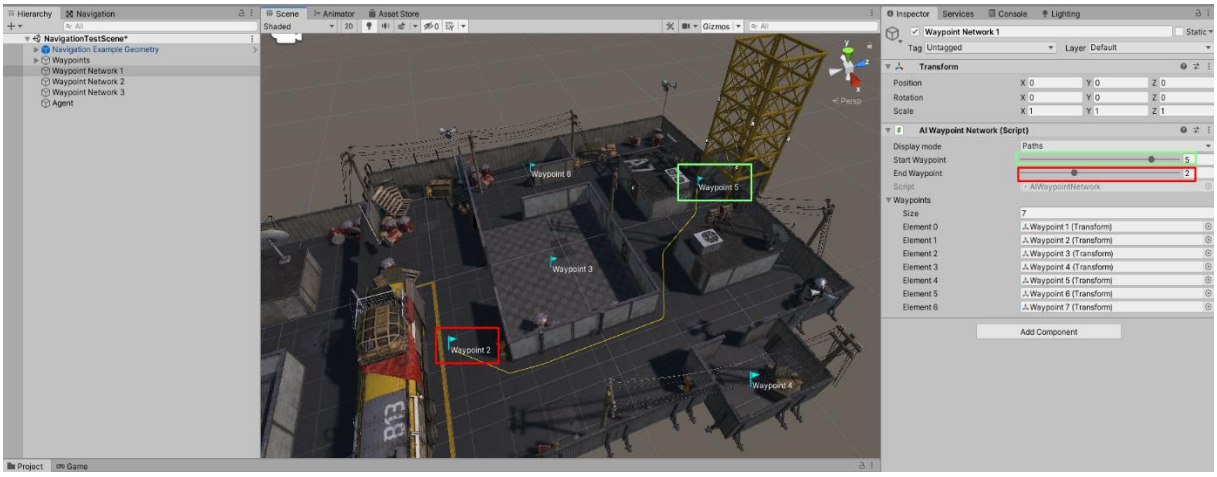
Na lijevom dijelu slike u prozoru hijerarhije vidljiv je označeni objekt „Waypoint Netowork 1“, a na desnom dijelu slike, u inspektoru je prikazana dodana kreirana komponenta (skripta) „AIWaypointNetwork“. Unutar skripte postoje dva polja „Display mode“ sa svoje tri opcije i „Waypoints“ koji je zapravo lista objekata, u ovom slučaju lista točki rute (engl. *Waypoint*). Lista je veličine sedam elemenata, odnosno sadrži sedam točaka. Navedenu načini prikaza (engl. *Display mode*) su: „ništa“ (engl. *None*), „povezana ruta“ (engl. *Connections*) i „put“ (engl. *Paths*).

Slijede slike koje prikazuju ostale dvije opcije načina prikaza rute.



Slika 25. Način prikaza – povezana ruta

Slika iznad prikazuje sve točke rute međusobno povezane tako da je element na trenutnoj poziciji povezan sa sljedećim i prethodnim elementima. Ukoliko je trenutni element zadnji, tada je sljedeći element onaj koji je zapravo prvi u listi.



Slika 26. Način prikaza – put

Slika iznad (Slika 26) prikazuje kako izgleda prikaz puta između dvije odabrane točke. Odabirom treće opcije prikaza rute, pojavljuju se dva nova klizna polja pomoću kojih se odabiru indeksi točke rute iz liste. Zelena boja označuje početnu točku, a crvena završnu.

Slijedi opis skripti korištenih za prethodno navedeni dio igre.

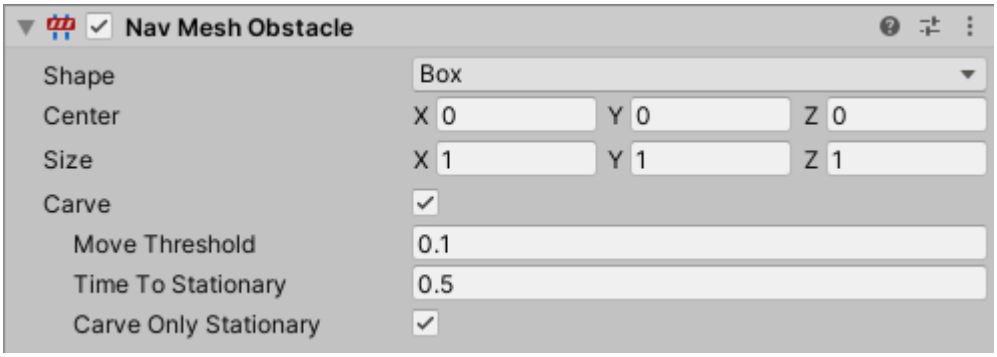
```
public enum PathsDisplayMode { None, Connections, Paths};
public class AIWaypointNetwork : MonoBehaviour
{
    [HideInInspector]
    public PathsDisplayMode DisplayMode = PathsDisplayMode.Connections;
    [HideInInspector]
    public int UIStart = 0;
    [HideInInspector]
    public int UIEnd = 0;

    public List<Transform> Waypoints = new List<Transform>();
}
```

Ova jednostavna skripta sadrži klasu „AIWaypointNetwork“ i enumeracije tipa prikaza rute te se kao takva dodaje na objekt koji će predstavljati mrežu točaka rute (potpunu rutu). Skripta ne posjeduje nikakvu logiku, jedina svrha joj je čuvati podatke o tome koje točke rute sadrži. Popis točaka rute je zapravo javna varijabla koja je tipa „List“, a može sadržavati bilo koje

podatke tipa „Transform“. Klasa „Transform“ sadrži podatke objekta koji govore o njegovim dimenzijama, poziciji i rotaciji. Elementi se listi dodaju pomoću prozora inspektora. Ostale varijable u skripti, odnosno klasi su javne, ali sakrivene u inspektoru. Razlog tome je taj što se ova skripta prikazuje na malo drukčiji način i to pomoću jedne druge skripte čija klasa ne nasljeđuje „MonoBehaviour“, već klasu „Editor“. Spomenuta skripta služi da temeljem odabranog načina prikaza („*None*“, „*Connections*“, „*Path*“), upravlja prikazivanjem i sakrivanjem atributa klase „AIWaypointNetwork“ u prozoru inspektora. Sve klase koje nasljeđuju klasu „Editor“ mogu biti smještene bilo gdje u projektu, ali moraju se nalaziti unutar mape „Editor“. Do sada opisivana skripta koja nasljeđuje „Editor“ klasu zove se „AIWaypointNetworkEditor“ te ni na kakav način ne utječe na samu igru i zbog toga neće biti detaljnije opisivana.

4.1.1.3. Prepreke

Kod navigacije, važno je spomenuti prepreke, pogotovo prepreke koje ne nastaju *bake* procesom, već su dinamičke. Agenti, kada se radi o navigaciji, mogu prolaziti kroz sve objekte koji nisu označeni kao „Nav Mesh Obstacle“, naravno ukoliko je to područje prohodno za agenta. Sam objekt koji sadrži „Nav Mesh Obstacle“ komponentu, sadrži nekoliko važnih opcija koje su prikazane na slici koja slijedi.



Slika 27. Nav Mesh Obstacle

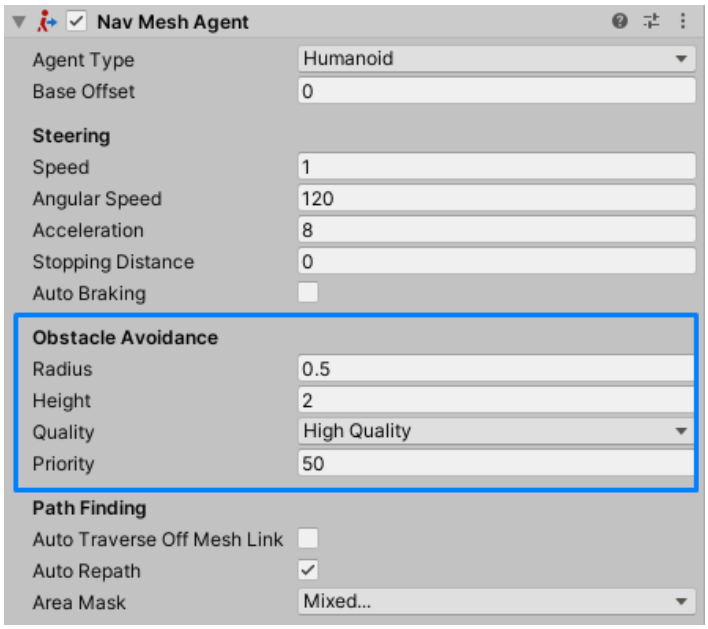
Dakle, važni atributi kod navedene komponente su oblik, njegove dimenzije i pozicija u odnosu na objekt na kojem se nalazi. Oblik „Nav Mesh Obstacle“ komponente može biti kutija ili kapsula. Vjerojatno najvažnija opcija je „Carve“, ukoliko je označena tada objekt u potpunosti prekida navigacijsku plohu na tom mjestu te ovaj prostor postaje neprohodan za agenta. Ukoliko je ova opcija isključena, tada agent jednostavno izbjegava koliziju s objektom, no i dalje se može gibati po tom dijeli plohe ukoliko postoji prohodni prostor (Unity Technologies, 2020f). Objekt bi trebao prekidati navigacijsku plohu samo kada je objekt stacionaran, u ovu

svrhu postoji opcija „Carve Only Stationary“. Uz navedenu opciju, dolaze i preostale dvije do sada neopisane opcije, jedna određuje količinu pomaka nakon kojeg objekt prestaje biti stacionaran („Move Threshold“), a druga vrijeme nakon kojeg objekt opet postaje stacionaran („Time To Stationary“) (Unity Technologies, 2020f).

4.1.2. Agenti

Agenti su, u navigacijskom smislu, kao što je već bilo spomenuto, objekti koji se na neki definirani način kreću po navigacijskoj plohi kroz prohodna područja. Objekti koji predstavljaju agente moraju sadržavati komponentu „Nav Mesh Agent“ te često i skriptu koja na temelju navedene komponente postavlja rutu kretanja (sljedeće odredište).

U dokumentaciji je navedeno da su agenti, pomoću navigacijske plohe („Nav Mesh“), „svjesni“ svijeta u kojem se nalaze te su u mogućnosti zaobilaziti jedni druge. Osim toga, mogu zaobilaziti ostale prepreke koje im se nalaze na putu do cilja (Unity Technologies, 2020g). Slijedi slika koja prikazuje komponentu, a nakon toga slijedi i prikaz skripte za upravljanje kretanja rutom.



Slika 28. Nav Mesh Agent

Vidjeli smo da se radijus agenata može podesiti kod kreiranja navigacijske plohe (poglavlje 4.1.1.1 Navigacijska ploha), ali ako pogledamo atribute ove komponente, možemo uočiti da postoji opcija podešavanja radijusa i visine agenta (označenom plavom bojom). Dakle, kod navigacijske plohe se definira zaobilaženje statične geometrije na sceni i zajedničko je svim

agentima bez obzira na njihove stvarne dimenzije. Kod ove komponente, dimenzije definiraju način zaobilaženja drugih agenata i prepreka spomenutih u odlomku 4.1.1.3 Prepreke (Unity Technologies, 2020g). Ostali atributi komponente upravljani su pomoću skripti i na to ćemo se vratiti kasnije, slijedi slika jednostavnog i osnovnog kretanja agenata po plohi.

```
public NavMeshPathStatus pathStatus;
private NavMeshAgent navAgent;

void Update()
{
	pathStatus = navAgent.pathStatus;

	if (navAgent.isOnOffMeshLink)
	{
		StartCoroutine(Jump(1.0f));
		return;
	}

	if ((!navAgent.hasPath && !navAgent.pathPending)
		|| pathStatus.Equals(NavMeshPathStatus.PathInvalid))
	{
		SetNextDestination(true);
	}
	else if (navAgent.isPathStale)
	{
		SetNextDestination(false);
	}
}
```

Ovaj dio skripte prikazuje osnovno kretanje agenta koji ne sadrži nikakve animacije, već samo „klizi“ po podlozi. Ova skripta nije korištena u završenoj igri, ali zbog jednostavnosti objašnjavanja i shvaćanja, tek će kasnije biti opisana prava skripta. Ova skripta, u suštini, svaki puta kada se pozove „Update“ metode, provjerava kakav je status puta na kojem se agent trenutno nalazi. Ukoliko se agent nalazi na vezi između dva nepovezana dijela plohe, tada odrađuje skakanje s trenutne poveznice na drugu. Nadalje ukoliko navedena tvrdnja nije istinita, tada ako agent već nema put i trenutno se ne kalkulira, ili pak ako je put nevažeći, tada mu se zadaje put. Ukoliko ni ovaj uvjet ne važi, potrebno je provjeriti neispravnost puta: put više nije optimalan ili je zastario. Ako je navedeno istinito (put je neispravan) agent gubi svoj put i potrebno mu je postaviti sljedeće odredište. Postavljanje puta, odnosno odredišta odrađuje se u metodi „SetNextDestination“. Navedena metoda je zapravo glavni dio i sadrži logiku kojom se postavlja agentov put. Metoda sadrži dva glavna dijela, a to su upravljanje rutom (prije opisanom klasom „AIWaypointNetwork“) i postavljanje odredišta (cilja) agentu. Metoda prima jedan argument koji indicira treba li se kod postavljanja odredišta promijeniti trenutna točka rute. Upravljanje rutom se zapravo svodi na upravljanje cirkularnim poljem

(slijed u polju od n elemenata: 0→1→…→n-1→n→0→…). Postavljanje odredišta odrađuje se na sljedeći način:

```
if (nextWaypointTransform != null)
{
      CurrentWaypointIndex = nextWaypointIndex;
      navAgent.SetDestination(nextWaypointTransform.position);
      return;
}
```

U gornjem dijelu koda možemo vidjeti da ukoliko je „Transform“ komponenta sljedeće točke („nextWaypointTransform“) rute valjana, tada se agentu postavlja pozicija te točke kao sljedeće odredište. Pozicija se postavlja pomoću agentove metode „SetDestination“.

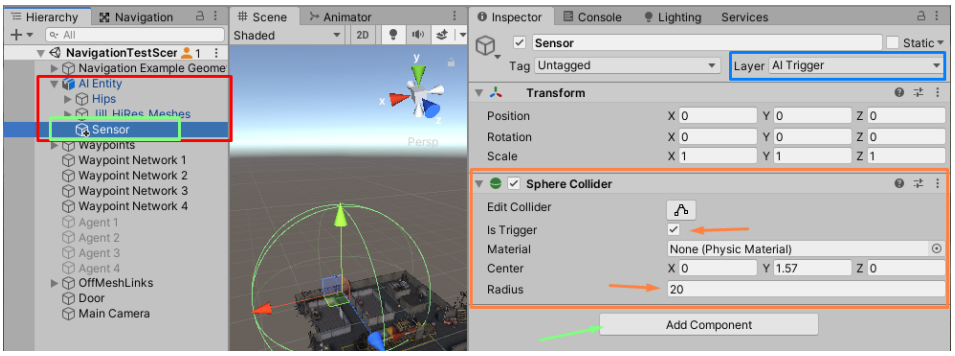
Što se tiče agenata i njima pripadajućih skripti korištenih u završnoj verziji igre, realizirani su pomoću nekoliko glavnih dijelova. Svaki agent unutar svog glavnog objekta (roditelj na najvišoj razini u hijerarhiji) mora sadržavati objekt koji se nalazi u „AI Entity Trigger“ sloju pomoću kojeg se zadaje cilj, a time i putanja do cilja. Unutar glavnog objekta nalazi se i posebni objekt koji sadrži svu geometriju i dijelove agenta te i jedan dodatni objekt koji mu služi kao osjetilo. Spomenuti objekt, kao i glavni objekt moraju se nalaziti u sloju „AI Entity“. Što se tiče dijelova agenta (geometrija, udovi, glava, torzo, itd.), moraju se nalaziti u sloju „AI Body Part“, a senzor (agentovo osjetilo) se mora nalaziti u sloju „AI Trigger“. Spomenuti dijelovi tijela „AI Body Part“ su dodatno označeni oznakama poput „Upper Body“, „Lowe Body“, „Chest“ ili „Head“. Pomoću ovakvog označivanja objekata je u skripti moguće zaključiti točno koji dio tijela je oštećen. Dijelovi tijela agenata pomoću kojih se može igraču nanijeti šteta se ne nalaze u sloju „AI Body Part“, već u sloju „AI Trigger“. Ovakvi objekti sadrže komponentu „Sphere Collider“ koja služi kao okidač (engl. *Trigger*). Uz navedenu komponentu ovakav dio tijela mora sadržavati i skriptu koja upravlja štetom koja se nanosi igraču te kreira čestice krvi kod udarca. Kako bi agentovo kretanje izgledalo prirodno, osim komponente „Nav Mesh Agent“ pomoću koje mu se zadaje destinacija, sadrži i komponentu „Animator“ koja će biti opisana kasnije.

Osim navedenog, agenti se sastoje i od tzv. „Ragdoll“ sustava koji je kreiran pomoću čarobnjaka za kreiranje spomenutog. Spomenuti sustav služi kako bi se dijelovi tijela agenata, kod radnje poput pada uslijed udarca, ponašali realistično i u skladu sa fizikom igre. Spomenuti sustav je skup komponenti na dijelovima agenta. Sustav se sastoji od tri glavne komponente koje čarobnjak za kreiranje kreira automatski i pozicionira ih prema kosturu agenta. Komponente koje čarobnjak kreira su sljedećih tipova: „Collider“, „Rigidbody“ i „Joint“ (Unity Technologies, 2020i). Komponenta „Collider“ na određenim dijelovima tijela prepoznaje kolizije s drugim dijelovima tijela i s ostalom geometrijom, a „Rigidbody“ služi za interakciju s fizikom

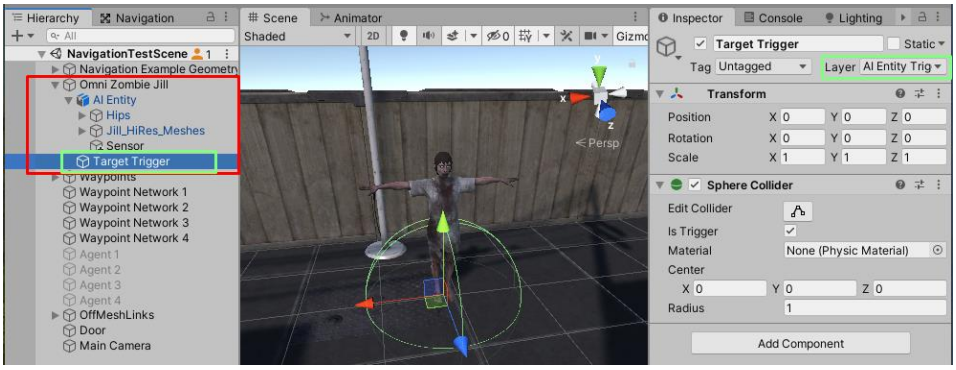
igre (gravitacija). Posljednja od tri komponente je „Joint“ i služi za stvaranje logičke veze između direktno povezanih dijelova tijela (podlaktica i nadlaktica kao lakat, nadlaktica i ruka kao rame, itd.). Spomenuta komponenta služi za definiranje u kojim se sve smjerovima dijelovi tijela mogu gibati na samim spojevima s drugim dijelovima (lakat, rame, koljeno, itd.).

4.1.2.1. Potrebni dodatni agentovi objekti

Agent, kao skup njegovih osnovnih dijelova (dijelovi tijela), nije sam po sebi dovoljan za potpunu funkcionalnost cijelog sustava. Zbog toga se mu potrebni neki dodatni elementi (objekti) postavljeni u njegovoj hijerarhiji (pripadaju agentu). Slijede slike koje detaljnije opisuju izgled objekta koji predstavlja agenta te opis zašto je baš na ovaj način posložena agentova hijerarhija.



Slika 29. Pregled agentove hijerarhije: „Sensor“



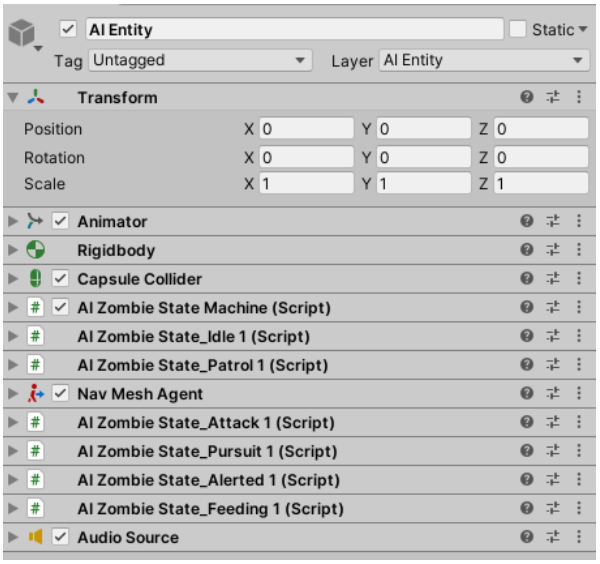
Slika 30. Pregled agentove hijerarhije: "Target Trigger"

Prva slika (Slika 29) prikazuje agentovu hijerarhiju (označeno crvenom bojom) i pregled objekta „Sensor“ (u hijerarhiji označen zelenom bojom). Sam objekt „AI Entity“ predstavlja agenta i sadrži sve njegove dijelove (skripte, geometriju, dijelove tijela, i sl.). Objekt „Sensor“ agentu služi kao osjetilo i prema tome ovaj objekt se mora gibati zajedno sa njim. Zbog navedenog razlog, ali i zbog organizacije objekata, osjetilo se nalazi u agentovoj hijerarhiji. Na

desnoj strani slike, plavom bojom označen je sloj kojem osjetilo mora pripadati, a narančasto su označene postavke osjetila. Kako bi osjetilo radilo prema očekivanom ponašanju, mora posjedovati komponentu „Sphere Collider“ koja očitava kolizije s ostalim objektima. Navedena komponenta ne smije djelovati na druge objekte ni na koji način, barem što se tiče fizičke strane. Drugim riječima, osjetilo ne bi nikada smjelo odgurivati druge objekte od svojeg plašta, već samo prolaziti kroz njih i očitavati kolizije s njima. Iz navedenog razloga, komponenta „Sphere Collider“ mora biti označena kao „Is Trigger“ (okidač) te mora imati neki radijus ovisno o potrebama. Kasnije je na objekt dodana skripta koja jednostavno registrira kolizije (događaje) i šalje informacije o istima nadležnoj skripti. Informacije mogu biti: ulazak objekta u kolizijsku sferu, izlazak ili zadržavanje u istoj. Objekt osjetila je referenciran u skripti „AIZombieStateMachine“ (spomenuta nadležna skripta), koja će biti opisana kasnije te preko nje navedena skripta otkriva poziciju izvora hrane ili igrača - zvukom, svjetlom iz svjetiljke, direktnim uočavanjem igrača ili „mirisom“ krvi ranjenog igrača.

Nadalje, što se tiče druge slike (Slika 30), ona prikazuje malo izmijenjenu agentovu hijerarhiju označenu crvenom te objekt „Target Trigger“ unutar hijerarhije (označeno zelenom bojom). Prvo što valja primijetiti u hijerarhiji je to da se „AI Entity“ objekt više ne nalazi na vrhu hijerarhije, nego je pomaknut jednu razinu ispod te je na vrhu „Omnie Zombie Jill“ objekt. Navedeni objekt služi samo kao omotač samog objekta agenta (koji je ostao nepromijenjen) i objekta „Target Trigger“. Kako bi bilo jasnije zašto je napravljena ovakva promjena u hijerarhiji, prvo je važno razumjeti funkciju „Target Trigger“ objekta. Navedeni objekt služi kao indikator agentu da je došao do cilja. Primjerice ukoliko agent pomoću svog osjetila („Sensor“) čuje igrača, to se registrira u „AIZombieStateMachine“ skripti, a nakon toga se agentu postavlja destinacija do izvora zvuka. Osim što se postavlja destinacija, na tu istu poziciju se postavlja i objekt „Target Trigger“, kolizijom agenta i navedenog objekta, indicira se uspješno završavanje puta. Sada je jasno da se jedan ovakav objekt ne smije nikako nalaziti unutar drugog objekta koji se kreće, već mora biti unutar stacionarnog objekta, a to pruža „Omnie Zombie“ objekt. Naime, da se „Target Trigger“ nalazi unutar objekta „AI Entity“, tada bi se kretanjem agenta kretao i spomenuti objekt te agentov put nikada ne bi mogao biti završen na prije opisan način. Gledajući od čega se ovaj objekt sastoji (Slika 30 - desna strana), vidi se kako je on zapravo vrlo jednostavan. Objekt je gotovo identičan kao i objekt iz slike prije (Slika 29), samo što se nalazi na „AI Entity Trigger“ sloju. Ovaj objekt je također referenciran u „AIZombieStateMachine“ skripti kako bi se upravljalo objektom te se registrirao trenutak agentovog dolaska do odredišta i postavljanje sljedećeg odredišta.

Slijedi prikaz komponenti koje agent sadrži u svojem objektu „AIEntity“ (Slika 31). Na navedenoj slici možemo vidjeti da ovaj agent sadrži sva stanja navedena ranije i sadrži skriptu „AIZombieStateMachine“ kako bi mogao funkcionirati. Sadrži komponentu „Animator“, izvor zvuka „Audio Source“ kako bi mogao emitirati zvukove, sadrži „Nav Mesh Agent“ kako bi se mogao koristiti navigacijskom plohom te „Rigidbody“ i „Capsule Collider“ kako bi mogao imati interakcije s okolinom.



Slika 31. Komponente "AIEntity" objekta

4.1.2.2. Stanja i ponašanje

Od skripti koje agenti sadrže, najvažnija skripta je „AIZombieStateMachine“ koja je zapravo specijalizacija šire skripte „AIStateMachine“, stoga će sve biti opisivano kao „AIZombieStateMachine“ (skripta specijalizacija sadrži sve element i funkcije skripte koju ona specijalizira). Specijalizacija je kreirana zbog rastavljanja dijelova koji se tiču konkretno agenata korištenih u ovoj igri, a to su zombiji. Ukoliko bi u igri postojali i drugačiji agenti s drugačijim svojstvima, primjerice roboti, bilo bi vrlo lako dodati specijalne zahtjeve takvog agenta. Skripta posjeduje informacije o trenutnoj agentovoj vizualnoj i audio prijetnji, patrolnoj ruti, trenutnom stanju u kojem se agent nalazi, upravlja samom jezgrom agentova ponašanja i sadrži parametre poput trenutnog zdravlja, inteligencije, agresivnosti i sl. Ova skripta upravlja stanjima agenta, a agent može imati nekoliko različitih stanja: bez posla (engl. *Idle*), patrola (engl. *Patrol*), napadanje (engl. *Attack*), potjera (engl. *Pursuit*), pripravnost (engl. *Alerted*) ili hranjenje (engl. *Feeding*). Svako stanje predstavljeno je skriptom i sadrži logiku o izmjeni stanja u neko drugo na temelju događaja u agentovoj okolini. Svaka od ovih skripti također upravlja parametrima agenta u ovisnosti o kontekstu stanja, promjena brzine kretanja,

postavljanje tipa napada, smanjivanje „osjećaja“ gladi tijekom hranjenja i sl. Važno je napomenuti da ne trebaju nužno agenti sadržavati sva stanja. Primjerice, pomoću ovih skripti može se kreirati agent koji uopće nije agresivan i ne napada igrača već samo otkriva njegovu ili svoju poziciju ostalim agentima. Slijedi prikaz „Update“ metode skripte „AIStateMachine“, a nakon toga samo opis „Update„ metode kod skripte „AIZombieStateMachine“.

```
protected virtual void Update()
{
    if (!_currentState || !_NavAgent || !_NavAgent.isActiveAndEnabled)
        return;

    AIStateType newStateType = _currentState.OnUpdate();
    if (newStateType != _currentStateType)
    {
        AIState newState;
        if (_states.TryGetValue(newStateType, out newState))
        {
            _currentState.OnExitState();
            newState.OnEnterState();
            _currentState = newState;
        }
        else if (_states.TryGetValue(AIStateType.Idle, out newState))
        {
            _currentState.OnExitState();
            newState.OnEnterState();
            _currentState = newState;
        }

        _currentStateType = newStateType;
    }
}
```

Ovaj dio koda djeluje vrlo jednostavno, zapravo i jest vrlo jednostavan, ali u pozadini se odvija jako puno raznih stvari koje su naoko nevidljive gledajući samo ovaj dio koda. U suštini, agentovo funkcioniranje sastoji se u najvećem dijelu od stanja u kojem se on trenutno nalazi te se baš u ovom dijelu skripte poziva promjena agentovog stanja. Odmah na početku metode, ali nakon rutinskih provjera gdje se provjerava ispravnost trenutnog stanja i komponente „Nav Agent“, poziva se metoda „OnUpdate“ nad trenutnim stanjem agenta. Dakle, bez obzira u kojem se agent stanju nalazi, ova metoda se poziva svaki put i iz ove skripte kako ne bi skripte stanja bezrazložno u sebi posjedovale „Update“ metodu. Čak i kada bi skripte stanja imale „Update“ metodu, ona bi se pozivala na svim skriptama u isto vrijeme, a ne zanima nas stanje u kojem se agent trenutno ne nalazi, stoga je sustav osmišljena na ovaj način. U metodu koja se poziva („OnUpdate“), trenutno stanje provjerava treba li agent i dalje ostati u tom stanju ili je vrijeme za promjenu, zbog svojeg „osjećaja gladi“ ili možda zbog akcije prouzročene igračem. Nakon što trenutno stanje provjeri koje je sljedeće stanje u kojem bi se agent trebao nalaziti i ako je to stanje različito od trenutnog, „AIStateMachine“ nad trenutnim stanjem poziva

metodu „OnExitState“ koja obavlja akcije potrebne prije izlaska iz stanja. Nakon toga se uzima novo stanje i poziva se „OnEnterState“ metoda koja se pak poziva kod samog ulaska u novo stanje. Na kraju se ovo novo stanje postavlja kao trenutno agentovo stanje.

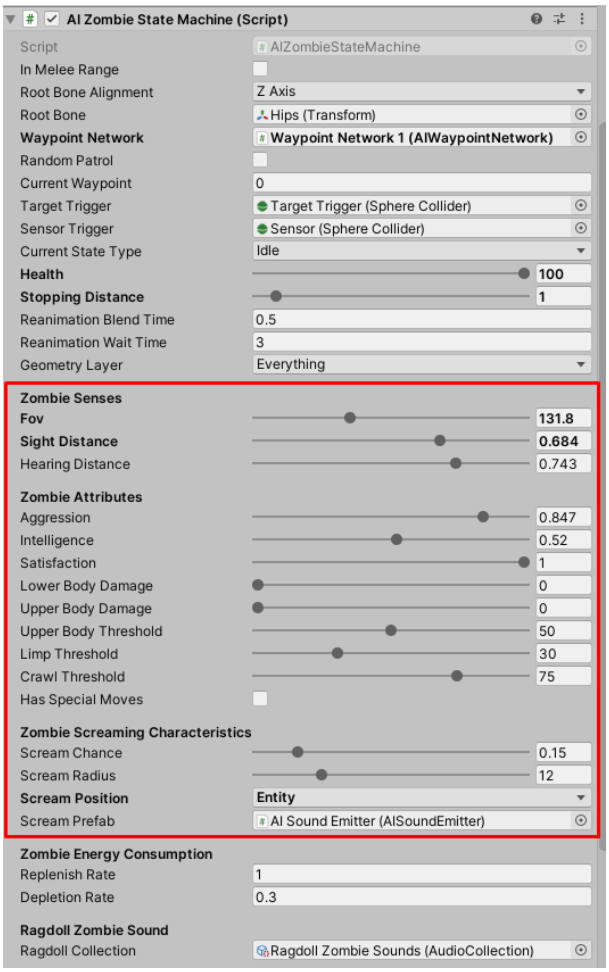
Što se tiče „AIZombieStateMachine“, „Update“ metoda neće biti opisana jer služi samo za komunikaciju sa animatorom te će metoda biti prikazana kod poglavlja 4.4 Animator. Kao što je bilo navedeno, ova skripta je specijalizacija „AIStateMachine“ skripte, stoga ova skripta može pozivati „Update“ metodu koja je bilo opisana u odlomku prije. Te se ovo radi na samom početku metode, bez ovog poziva roditeljeve metode, nikada se ne bi pozvala metoda koja provjerava agentova stanja. Kao što je bilo navedeno, ova metoda fokusira se samo na agentove animacije i to konkretno na one koje se odnose na zombije. Skripte stanja postavljaju brzinu kretanja, vrstu napada na igrača, indikator koji govori o tome da li se agent trenutno hrani, itd. Sve ove vrijednosti čuvaju se u varijablama „AIZombieStateMachine“ skripte i vrlo često se mijenjaju. Kod svakog poziva „Update“ metode, ali nakon provjere i promjene stanja (ako je potrebno), navedena skripta šalje vrijednosti svake od ovih varijabli *animatoru*. Animator je komponenta koja se nalazi na objektu agenta i služi za upravljanje animacijama.

Naravno, skripte „AIStateMachine“ i „AIZombieStateMachine“ sadrže puno više funkcionalnosti i metoda od spomenutih, no za razumjeti način na koji agenti funkcioniraju ovo je dovoljno. Kako bi detaljnije agent mogao biti opisan, slijedi prikaz cijele skripte „AIZombieStateMachine“ kako komponente na „AIEntity“ objektu.

Iz slike ispod (Slika 32) možemo vidjeti jako velik broj parametara koji se mogu podesiti i tako napraviti velike razlike između naizgled jednakih agenata. U dijelu označenom crvenom bojom, možemo vidjeti velik broj postavki koje definiraju agenta, a neki od tih postavki su: širina vidnog polja (*FOV* – engl. *Field of view*), daljina do koje agent „vidi“ (engl. *Sight distance*), agresivnost, inteligencija, granica zdravlja donjeg dijela tijela na kojoj agent počinje šepati te slično i za gornji dio tijela te granica na kojoj agent počinje puzati (ukoliko su mu noge oduzete), šansa za glasanjem (vrištanjem) ukoliko vidi igrača, udaljenost na koju se vrisak čuje te pozicija na koju se postavlja izvor prijetnje (na sebe ili na poziciju na kojoj je viđen igrač).

Stanja agenata su kreirana pomoću nasljeđivanja. U slučaju zombija, skripte su posložene na sljedeći način (od najgeneralnije do najspecijaliziranije skripte): MonoBehaviour → AIState → AIZombieState → AIZombieState_X. Kod zadnje navedene skripte slovo „X“ predstavlja naziv stanja, primjerice „Idle“, „Patrol“, itd. Skripte nasljeđuju klasu „MonoBehaviour“ samo iz razloga da se mogu dodati na objekte u igri, ne implementira se nijedna metode iz navedene klase (osim metode „Awake“ koja se bi inače mogla pozivati u konstruktoru bilo koje klase).

Što se tiče „AIState“ skripte, kako je ona najgeneralnija skripta u ovome dijelu, vrlo je apstraktna te posjeduje samo popis metoda koje bi ostale (specijaliziranije) skripte trebale implementirati. Popis navedenih apstraktnih metoda je sljedeći: „OnEnterState“, „OnExitState“, „OnAnimatorIKUpdate“, „OnTriggerEvent“, „OnDestinationReached“, „GetStateType“ i „OnUpdate“.



Slika 32. AIZombieStateMachine kao komponenta

Skripta koja dolazi sljedeća u nizu, „AIZombieState“ implementira metodu koja je za svako ponašanje zombija jednaka, a to je „OnTriggerEvent“. Ova metoda je implementirana na način da upravlja prijetnjama koje je zombi susreo na svom putu. Slijedi pregled metode:

```
override void OnTriggerEvent(AITriggerEventType eventType, Collider other){
    if (!_zombieMachine)
        return;

    if (eventType != AITriggerEventType.Exit)
    {
        AITargetType currentTargetType = _zombieMachine.VisualThreat.Type;
```

```
        if (other.CompareTag("Player")){
            ManagePlayerThreat(currentTargetType, other);
        }
        else if(other.CompareTag("Flashlight")
                  && currentTargetType != AITargetType.Visual_Player){
            ManageFlashlightVisuals(other);
        }
        else if(other.CompareTag("AI Sound Emitter")
                  && currentTargetType != AITargetType.Visual_Player){
            ManageAudioThreat(other);
        }
        else if(other.CompareTag("AI Food")
                  && currentTargetType != AITargetType.Visual_Player
                  && currentTargetType != AITargetType.Visual_Light
                  && _zombieMachine.AudioThreat.Type==AITargetType.None
                  && _zombieMachine.GetSatisfaction() <= 0.9f){
            ManageFoodThreat(other);
        }
    }
}
```

Dakle, metoda prima dva parametra koje joj prosljeđuje „AIStateMachine“ skripta pomoću agentovog objekta „Sensor“. Metoda provjerava tip događaja koji se je dogodio te prema tome upravlja prijetnjom. Primjerice ukoliko metoda zaključi da se je interakcija (kolizija) dogodila sa igračem direktno, tada se upravlja takvom prijetnjom pomoću metode „ManagePlayerThreat“. Gledajuću opisivanu metodu i njezinu funkcionalnost, u suštini je jednostavna i nije potrebno promatrati na koji način se odvija upravljanje prijetnjama. Ostale prijetnje koje postoje i koje agent pomoću metode može registrirati su: uočavanje svjetla iz svjetiljke, registriranje zvuka (bilo kojeg), uočavanje hrane ukoliko je agent gladan. Na kraju, važno je napomenuti da ove prijetnje imaju prioritete, uočavanje igrača (vizualno) ima najveću razinu prijetnje, a hranjenje najmanju od svih, svjetiljka je druga, a zvuk treća prijetnja prema prioritetu.

Kako bismo detaljnije mogli razumjeti agentova stanja, slijedi opis metode „OnUpdate“ u skripti „AIZombieState_Idle“ i kratki osvrt na samu skriptu.

```
public override AIStateType OnUpdate()
{
    if(!_zombieMachine)
        return AIStateType.Idle;

    if (_zombieMachine.VisualThreat.Type == AITargetType.Visual_Player)
    {
        _zombieMachine.SetTarget(_zombieMachine.VisualThreat);
        return AIStateType.Pursuit;
    }
    else if (_zombieMachine.VisualThreat.Type == AITargetType.Visual_Light)
    {
        _zombieMachine.SetTarget(_zombieMachine.VisualThreat);
        return AIStateType.Alerted;
    }
```

```
    else if (_zombieMachine.AudioThreat.Type == AITargetType.Audio)
    {
        _zombieMachine.SetTarget(_zombieMachine.AudioThreat);
        return AIStateType.Alerted;
    }
    else if(_zombieMachine.VisualThreat.Type == AITargetType.Visual_Food)
    {
        _zombieMachine.SetTarget(_zombieMachine.VisualThreat);
        return AIStateType.Pursuit;
    }

    _timer += Time.deltaTime;
    if (_timer >= _idleTime)
    {
        Vector3 waypointPosition = _zombieMachine.GetWaypointPosition(false);
        _zombieMachine._NavAgent.SetDestination(waypointPosition);
        _zombieMachine._NavAgent.isStopped = false;
        return AIStateType.Alerted;
    }

    return AIStateType.Idle;
}
```

Ova skripta, odnosno metode „OnUpdate“ ove skripte odabrana je jer je najkraća od ostalih skripti stanja, no jednako funkcionira kao i ostale. Glavna uloga ove skripte već je bila donekle opisana, no sada će to znanje biti malo produbljeno. Dakle, glavna zadaća „OnUpdate“ metode svake skripte stanja je da se po završetku rada metode, onoj metodi koja ju je poslala da do znanja koje će biti sljedeće stanje u koje agent ulazi. Naravno, ovo nije jedina zadaća koju skripta odrađuje, već su tu i poslovi poput postavljanja destinacije, ovisno o agentovom stanju. Dakle, uzmimo za primjer da se agent trenutno nalazi u besposlenom stanju (engl. *Idle*), odnosno iz „Update“ metode u skripti „AIStateMachine“, poziva se gore postavljena metoda skripte „AIZombieStateIdle“. Ova metoda radi na način kao da „želi“ izaći iz svog stanja, provjerava sve moguće scenarije, ne bi li agent promijenio stanje. Scenariji se provjeravaju redom po prioritetu, ukoliko je agentova trenutna prijetnja igrač (vizualno), tada se postavlja cilj na poziciju igrača i metoda odmah završava, a rezultat radnje je stanje „Pursuit“ (potjera). Stanje potjere (engl. *Pursuit*) je logičan odabir stanja, jer ukoliko agent „vidi“ igrača, najprirodnija reakcija (ukoliko se radi o zombijima) je lov plijena. Pod pojmom cilj, ovdje se misli na „AITargetTrigger“ objekt o kojem je već bilo riječi. Ostale provjere scenarija su identične, samo se provjeravaju drugi scenariji i rezultat su drugačija stanja. Na samom kraju metode postoji dio koji mjeri koliko dugo je agent proveo u besposlenom stanju, ukoliko to vrijeme prođe limit koji je zadan skriptom (različito za svakog agenta), tada se agenta forsira da promijeni stanje i to u stanje pripravnosti (engl. *Alerted*). Ukoliko agent dođe u ovakav scenarij, to znači da ga ništa nije omelo u njegovom besposlenom stanju, nakon promjene

stanja u stanje pripravnosti, tada to novo stanje preuzima odgovornost odlučivanja o sljedećim radnjama (u ovom slučaju je to početak patroliranja). U ovome slučaju možda nije logično da sljedeće stanje bude „Alerted“, ali je cijeli sustav stanja baziran na način da stanje pripravnost odrađuje najveći dio posla što se tiče promjena stanja. Razlog tome je taj što ovo stanje najviše po svojim osnovnim parametrima odgovara ostalim stanjima. Sve skripte stanja ove razine implementiraju metodu „OnEnterState“, a neke i „OnExitState“. Pomoću navedenih metoda se postavljaju parametri karakteristični za to stanje, primjerice u stanju potjere podešava se brzina agenta na veću vrijednost nego u drugim stanjima i sl. „OnExitState“ pak radi obrnuto, kod izlaska iz stanja, skripta obavlja resetiranje parametara koje je mijenjala na osnovne vrijednosti.

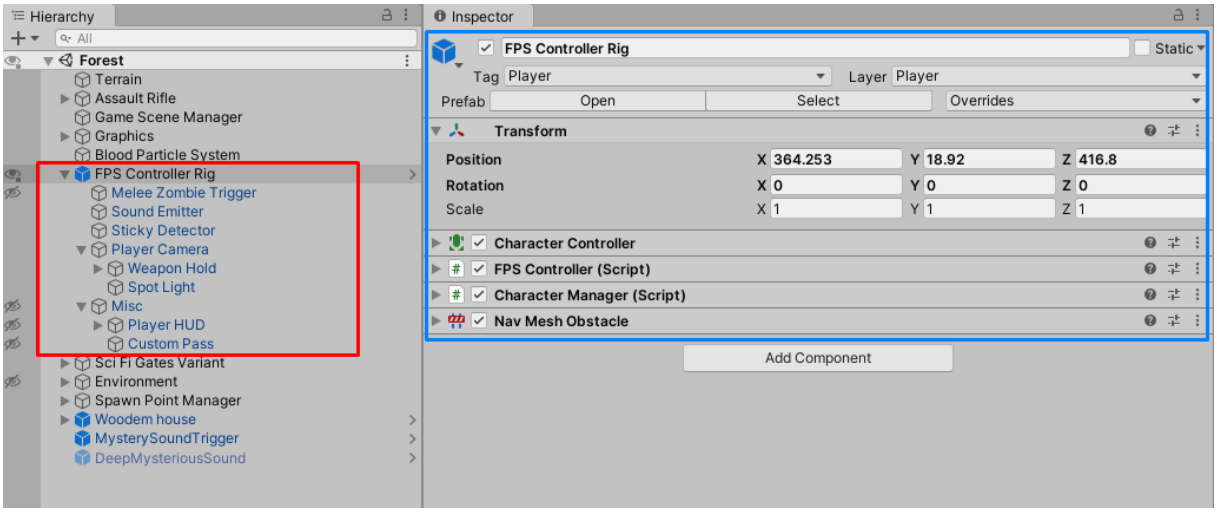
4.2. Objekt igrača

Ukoliko objekt igrača (igrač) postoji u igri, tada je on vjerojatno i osnovni dio igre, tako je i ovdje. Razlog zašto tako važna komponenta igre nije ranije opisana je taj što se najčešće komponenta igrača uzima „zdravo za gotovo“. Ukoliko igrač postoji, više – manje je svima jasno kako ova komponenta treba izgledati i kako većinom izgleda. Kako na tržište dolaze nove igre, tako se i objekt igrača usavršava, a kod svake nove generacije igara gotovo sve igre preuzimaju slično odrađene objekte koji predstavljaju igrača.

Objekt igrača u ovoj igri kreiran je pomoću već prije spomenutog tečaja, a u poglavljima koja slijede bit će opisane igračeve kretnje i kontrole. Uz igračev objekt usko su vezani interaktivni objekti kao i oružja s balistikom te će i ovi dijelovi biti opisani kasnije.

Ispod slijedi slika (Slika 33) koja prikazuje hijerarhiju igrača i njegove glavne skripte. Što se tiče hijerarhije i skupa objekata koji predstavljaju igrača, prikazani su lijevo na spomenutoj slici i označeni crvenom bojom. Dakle, igrač se sastoji od objekta „Melee Zombie Trigger“ i zadaća mu je obavijestiti agenta da se nalazi u dosegu igrača. Sljedeći objekti su „Sound Emitter“ koji služi za emitiranje zvukova te ga agenti mogu registrirati te „Sticky Detector“ koji uoči kolizije s agentom, igraču smanji brzinu kretanja. Nadalje, kamera je objekt pomoću kojeg se gleda scena, odnosno kamera predstavlja igračevu glavu, specifičnije oči. Također, unutar igračeve hijerarhije nalazi se i objekt „Misc“ koji sadrži grafičko sučelje (trenutno zdravlje i energiju, ispis tekstova kod događaja, i sl.) te „Custom Pass“. Navedeni objekti se inače ne moraju nalaziti unutar igračeve hijerarhije, ali su postavljeni na ovaj način iz razloga što se objekt „FPS Controller Rig“, odnosno igrač, po završetku scene seli iz jedne scene u drugu. „Custom Pass“ je objekt koji osigurava da se sva igračeva grafika uvijek iscrtava na ekranu iznad sve ostale

grafike. Sve ove navedene komponente su postavljene na ovaj način kako bi se u svakom trenutku nalazile na istoj poziciji u odnosu na igrala (kretanjem igrača i one se kreću u odnosu na ostatak svijeta, a lokalno su statične). Što se tiče objekata unutar kamere, oni su također pozicionirani ovako iz razloga da se miču zajedno i s kamerom i igračem. Možemo vidjeti da se unutar kamere nalaze objekt „Weapon Hold“ koji se sastoji od oružja te objekt „Spot Light“ koji predstavlja svjetiljku. Oružja i svjetlo iz svjetiljke trebali bi, osim kretanja zajedno s igračem, imati i rotaciju ovisno o smjeru u kojem gleda kamera te je to pravi razlog zašto je hijerarhija kreirana na ovaj način.



Slika 33. Hijerarhija igračevog objekta

Što se tiče komponenti od kojih se sastoji igrač, prikazane su s desne strane slike i označene plavom bojom. Igrač se sastoji od komponente „Character Controller“ i druge dvije najvažnije komponente (skripte): „CharacterManager“ i „FPSController“. Prva od skripti služi ponajviše za postavljanje stanja igrača i upravljanje njegovim parametrima te za proizvodnju zvukova i provjeru interaktivnih objekata. Što se tiče druge skripte, ona je zaslužna za registriranje korisničkog inputa s tipkovnice i miša. Navedena skripta („FPSController“) upravlja kretanjem igrača.

4.2.1. Kontrole i „FPSController“ skripta

U ovome dijeli bit će prikazana slika tipkovnice i sve kontrole kojima se igračem upravlja. Nakon ovoga, bit će prikazane i opisane neke skripte koje su u direktnoj vezi s kretanjem igračevog objekta. Slika ispod (Slika 34) prikazuje tipkovnicu i računalni miš s

oznakama tipki koje se u igri koriste kako bi se moglo kontrolirati objektom igrača i kako bi imao interakciju s drugim objektima. Nije potrebno detaljnije govoriti o slici, već će biti prikazani neki dijelovi skripti koji služe za postojanje navedenih kontrola.

Slika 34. Kontrole za igrača

Kako je već bilo spomenuto prije, za kretanje i čitanje korisničkog inputa, kreirana je skripta „FPSController“. Ova skripta će biti podijeljena u nekoliko dijelova kako bi bilo prikazano njezino djelovanje prema tim segmentima. Ova skripta osim glavne klase koja nasljeđuje „MonoBehaviour“, sadrži još dvije klase koje su pomoć kod dodavanja efekta klimanja glavom kod hodanja (engl. *Headbob*), funkciju „delegate“ koja služi kao odziv kod klimanja glavom i dvije enumeracije. Detaljnije će biti opisano kasnije te sada slijede osnovni dijelovi skripte.

```
protected void Update()
{
    if (_characterController.isGrounded)
        _fallingTimer = 0f;
    else
        _fallingTimer += Time.deltaTime;

    if (Time.timeScale > Mathf.Epsilon)
        _mouseLook.LookRotation(transform, _camera.transform);

    ManageInputs();

    DetermineMoveStatus();

    _previouslyGrounded = _characterController.isGrounded;

    if (_playerStatus == PlayerMoveStatus.Running)
        _stamina = Mathf.Max(_stamina - _staminaDepletion
                             * Time.deltaTime,0f);
    else if(_playerStatus == PlayerMoveStatus.NotGrounded)
        _stamina = Mathf.Max(_stamin - _staminaDepletion * 3f
                             * Time.deltaTime,0f);
```

```
    else
        _stamina = Mathf.Min(_stamina + _staminaRecovery
                             * Time.deltaTime,100f);

    _dragMultiplier = Mathf.Min(_dragMultiplier + Time.deltaTime,
                             _dragMultiplierLimit);
}
```

Dio koda iznad prikazuje „Update“ metode skripte „FPSController“ kojoj je uloga čitanje korisničkog inputa, postavljanje trenutnog stanja o kretanju igrača, upravljanje energijom i brzinom. U prvih nekoliko linija koda ove metode, upravlja se brojačem vremena pada. Ukoliko se igrač nalazi na terenu, tada se brojač zadržava na nuli, inače mu se dodaje vrijednost od „Time.deltaTime“. Odnosno dodaje mu se vrijeme od početka prethodne slike (engl. *Frame*) pa do početka trenutne, u milisekundama. Ova varijabla („_fallingTimer“) može se iskoristiti za zadavanje štete igraču ukoliko pada s prevelike visine. Nadalje, metoda „LookRotation“ koristi se za izglađeno rotiranje igrača, prema inputu s pozicije miša. Metoda se nalazi unutar Unityjeve skripte „MouseLook“ iz standardnih resursa za igre iz prvog lica. Ova skripta je nadograđena dijelom koji upravlja trzajem oružja, a metoda „LookRotation“ je nadograđena dijelom koji onda taj trzaj primjenjuje na kameru. Kod upravljanja korisničkih inputa, registriraju se inputi za upravljanje svjetiljkom, skok i promjenu položaja (čučanj). Promjena položaja u čučanj se odvija smanjivanjem igračeve visine na polovicu, a kod promjena položaja iz čučnja u stojeće stanje, pokreće se zasebni posao. Ovaj zasebni posao (engl. *Coroutine*) ima ulogu ponovno vratiti igračevu visinu na originalnu vrijednost, ali se to ne odvija odjednom (kao što to radi promjena u čučanj), već se odvija kroz vrijeme kako bi ta promjena izgledala glatko. Što se tiče upravljanja inputom za kretanje, nalazi se u metodi „FixedUpdate“ jer je povezan s kalkulacijama i upotrebom fizike (dobra praksa je postaviti ovakve dijelove u „FixedUpdate“). Metoda „DetermineMoveStatus“ provjerava u kojem bi se stanju igrač trenutno trebao nalaziti, primjerice ukoliko se u prošloj slici (engl. *Frame*) nije nalazio na podlozi, a u trenutnoj se nalazi, možemo zaključiti da je ili doskočio ili pao na podlogu. U ovom slučaju, status je „Landing“, a ukoliko igrač nema brzinu, tada je status „NotMoving“, itd. Nadalje, do kraja metode, odvija se upravljanje igračevom energijom ovisno o igračevom statusu i postavljanje opterećenja na igračevo kretanja ukoliko je u kontaktu s agentom.

Slijedi i opis „FixedUpdate“ metode navedene skripte jer je vrlo važna za igračevo kretanje.

```
protected void FixedUpdate()
{
    float horizontal = Input.GetAxis("Horizontal");
    float vertical = Input.GetAxis("Vertical");
    _isWalking = !Input.GetKey(KeyCode.LeftShift);
```

```
    float speed = ManagePlayerSpeed();

    Vector3 desiredMove = CalculateDesiredMove(horizontal, vertical);

    _moveDirection.x = !_freezeMovement ? desiredMove.x * speed *
                            _dragMultiplier : 0f;
    _moveDirection.z = !_freezeMovement ? desiredMove.z * speed *
                            _dragMultiplier : 0f;

    if (_characterController.isGrounded)
        DoIsGroundedJob();
    else
        _moveDirection += Physics.gravity * _gravityMultilier *
                            Time.fixedDeltaTime;

    _characterController.Move(_moveDirection * Time.fixedDeltaTime);

    Vector3 speedXZ = new Vector3(_characterController.velocity.x, 0.0f,
                            _characterController.velocity.z);

    if (speedXZ.magnitude > 0.01f)
        DoHeadbob(speedXZ);
    else
    {
        _camera.transform.localPosition = _localSpaceCameraPos;
        _weapon.transform.localPosition = _localSpaceWeaponPos;
    }
}
```

Ova metoda odvija se točno svakih 0.02 sekunde i unutar svakog takvog intervala odviju se sve radnje unutar metode. Među prvim linijama koda ove metode, učitavaju se korisnikovi horizontalni (tipke „A“ i „D“) i vertikalni (tipke „W“ i „S“) inputi. Vrijednosti ovih inputa su decimalni brojevo koji mogu biti negativni (tipke „A“ i „S“), pozitivni (tipke „W“ i „D“) ili 0 (nema inputa). Nadalje, provjerava se hoda li igrač (hoda ukoliko ne trči - nije pritisnuta tipka „LeftShift“) i prema tome se u sljedećoj liniji postavlja trenutna brzina igrača. Brzina ovisi o prije navedenoj vrijednosti „_isWalking“, osim toga, brzina ovisi i o tome da li se igrač nalazi u čučnju, nišani oružjem i koliko je trenutno ozljeđen. Metoda „CalculateDesiredMove“ uzima horizontalnu i vertikalnu komponentu inputa te iz njih kreira dvodimenzionalni vektor koji se normalizira (komponente vektora poprimaju vrijednosti u rasponu od 0 do 1). Nakon normalizacije, kreira se trodimenzionalni vektor „desiredMove“ kojem su komponente „naprijed“ i „desno“ pomnožene s komponentama dvodimenzionalnog normaliziranog vektora. Ovako kreiran vektor služi za kreiranje završnog vektora kretanja „_moveDirection“. Kako trenutno u vektoru „desiredMove“ postoje samo dvije komponente, iste se dodaju kao „X“ i „Z“ komponenta „_moveDirection“ vektora. Posljednja komponenta („Y“) dodaje se u metodi „DoIsGroundedJob“ ili u „else“ dijelu provjere. „DoIsGroundedJob“ metoda može biti pozvana samo ukoliko se igrač trenutno nalazi na podlozi. Ako je to istina, tada će se „Y“ komponenta

vektora „_moveDirection“ postaviti na negativnu vrijednost koja će igrača „lijepiti“ za podlogu, osim u slučaju kada igrač skače. Ukoliko igrač skoči, „Y“ komponenta je promijenjena na veličinu snage skoka koja je definirana skriptom. Ako se igrač ne nalazi na podlozi („else“ dio provjere), tada se kao „Y“ komponenta vektora dodaje gravitacija. Do vrijednosti gravitacija dolazi se pozivom „Physics.gravity“ i ova vrijednost je trodimenzionalni vektor (trenutno predefiniran na vrijednost x: 0, y: -9.81, z: 0). Napokon slijedi metoda „Move“ komponente „Character Controller“, koja pokreće igrača svakih „fixedDeltaTime“ sekundi i to u smjeru „_moveDirection“. Na kraju se iz „Character Controller“ komponente uzimaju konačne „X“ i „Z“ komponente te se kreira vektor „speedXZ“ koji služi za dodavanje efekta klimanja glavom kod hodanja (ukoliko igrač hoda). Ako igrač stoji, pozicija kamere i oružja se vraćaju na originalne vrijednosti kako se ne bi događala odstupanja zbog decimala i zaokruživanja vrijednosti kod kalkulacija. Također, važno se prisjetiti „delegate“ funkcije odziva klimanja glavom. Kod inicijalizacije skripte, navedena funkcija odziva se prijavljuje klasi koja izračunava klimanje glavom te kod svakog igračevog koraka funkcijom odziva može se emitirati zvuk koraka.

4.2.2. „CharacterManager“ skripta

U ovome dijelu ćemo se nadovezati na upravljanje objektom koji predstavlja igrača. Pod upravljanje stanjem igrača misli se na upravljanje njegovim parametrima koji ovise o samim akcijama igrača, ali i o ostalim interakcijama s objektima tijekom igre. Pod ostalim objektima podrazumijevaju se ponajprije agenti, ali i interaktivni objekti poput vrata, generatora, objekti koje je moguće pokupiti, itd. Ovdje će biti objašnjena skripta „CharacterManager“ te slijedi i prikaz nekih metoda.

```
protected void Update()
{
    SearchForInteractiveItems();
    if (_fpsController != null && _soundEmitter != null)
    {
        ManageSoundEmitters();
        _fpsController.SetDragMultiplierLimit(Mathf.Max(_health / 100f, 0.20f));
    }
    if (Input.GetButtonDown("Taunt"))
    {
        if(Time.time > _nextTauntSound)
            DoTaunt();
    }
    if (_playerHUD)
        _playerHUD.Invalidate(this);
}
```

Početkom svake slike (engl. *Frame*), poziva se početak metode „SearchForInteractiveItems“ koja pretražuje interaktivne objekte koji se nalaze ispred igrača. Preciznije, pretraživanje radi

na principu kolizije zrake koja se „ispaljuje“ iz sredine ekrana prema naprijed sa objektima koji su označeni kao interaktivni. Kako bismo detektirali samo one predmete koji se nalaze u neposrednoj blizini, ograničava se duljina ispucane zrake. Nakon ispucavanja zrake, dobiva se skup kolizija koji se može spremiti u polje ili listu tipa „RaycastHit“. Ukoliko je zraka pogodila neke interaktivne objekte, potrebno je prolaziti kroz sve objekte koji se nalaze u polju i pronaći onog koji ima najviši prioritet. Kako bi ove radnje bile efikasnije, na početku svake scene se svi interaktivni objekti pohranjuju u skripti „GameSceneManager“ u obliku mape (rječnika) kojoj je ključ jedinstveni identifikator objekta („instance ID“). Nakon što se pronađe interaktivni objekt s najvećim prioritetom (ako ga ima), tada se postavlja prikazivanje teksta interaktivnog objekta na ekranu. Osim toga, provjerava se i korisnički input za tipku „Use“ (tipka „E“) te ukoliko je pritisnuta u trenutnoj slici, poziva se aktivacija objekta. A ukoliko nema nikakvog objekta, miče se tekst s ekrana. Nakon što smo provjerili interaktivne objekte, slijedi dio koji upravlja emiterom zvuka kojeg na sebi nosi igrač. Ovaj emiter služi za emitiranje zvuka kojeg agenti mogu „čuti“. Metoda „ManageSoundEmitters“ koja se poziva je vrlo jednostavna. Naime, jedina zadaća spomenute metode je postaviti novi radijus za objekt „AISoundEmitter“ koji se nalazi na igraču. Radijus se računa na temelju igračevog statusa kretanja, ali i na temelju igračevog zdravlja. Što manje zdravlja igrač ima, to ga agenti lakše mogu „namirisati“. Prvobitni radijus se postavlja na vrijednost prema zdravlju, a nakon toga ukoliko trči, radijus se postavlja na upravo izračunani radijus ili na radijus trčanja (ovisno o tome koja vrijednost je veća), isto vrijedi i za skakanje. Na kraju metode, šalje se izračunani novi radijus objektu „AISoundEmitter“ te je metoda za postavljanje radijusa zadužena za to hoće li uzeti novi radijus u obzir ili ne. Ukoliko je novi radijus manji od trenutnog radijusa sfere (odnosno samog objekta kojeg agenti detektiraju), tada se radijus ne mijenja. Radijus može biti izveden na način da se sfera poveća za određenu vrijednost ili da se jednostavno samo postavi na novu vrijednost (kod brzog gađanja oružjem, radijus se zbraja). Nakon upravljanja emiterom zvuka, prije opisanoj skripti „FPSController“ se šalje informacija o tome koliko se opterećenje treba dodati na igračevo hodanje i to na temelju trenutnog zdravlja (ukoliko je igrač ranjen, sporije hoda). Nakon toga, provjerava se korisnički input zvukova ometanja (engl. *Taunt*), pomoću ove funkcionalnosti igrač može omesti agenta na način da mu promijeni trenutni posao ili radnju. Agentima je važno reagiranje na zvukove u nadi da će uloviti igrača. Metoda „DoTaunt“ može se pozvati samo ukoliko je od zadnjeg poziva iste prošlo određeno vrijeme. Što se tiče same metode, zadaća joj je emitirati pogrdni zvuk ili zvižduk, postaviti radijus „AISoundEmitter“ objekta (koji tada prema prije spomenutom kriteriju upravlja radijusom) te na kraju postavlja sljedeće

vrijeme mogućeg pozivanja metode. Zadnja linija koda u ovoj metodi je ažuriranje korisničkog sučelja s ispravnim podacima o energiji i zdravlju igrača.

Ova skripta je zadužena i za primanje štete nastale od strane agenata, za umiranje igrača, za završetak igre i scene te pokretanje poslova zaslužnih za učitavanje sljedeće scene (pomoću „ApplicationManager“ skripte). Što se tiče funkcionalnosti za primanje štete, izvedena je vrlo jednostavno. Agent igraču zadaje štetu tako dugo dok mu dio tijela (primjerice ruka, noga ili usta) ima doticaj s agentom, ali dok je agent u izvođenju napada. Takvih kolizija može biti vrlo mnogo te je primanje štete napravljeno na način da se jačina štete (udarca) pomnoži s „Time.deltaTime“. Nakon što se od trenutnog zdravlja oduzme ovakva vrijednost, ako je zdravlje manje od nule, ono se postavlja na nulu kako bi se na kraju izveo posao umiranja igrača. No prije toga, igrač emitira bolni jauk iz svoje kolekcije adekvatnih zvukova, a osim toga, emitira i zvuk udarca. Naravno, kako bi se ograničila prevelika učestalost igračevih jauka, kreira se varijabla koja predstavlja najranije sljedeće emitiranje ovakvog zvuka.

4.2.3. Oružje i balistika

Oružje je u igri implementirano samostalno što se tiče kreiranja animacija ciljanja, opaljivanja i promjene oružja. Također, samostalno su dodani zvukovi opaljenja i pokupljanja oružja, no sami zvukovi su preuzeti. Što se tiče grafike oružja i ruku koje nose to oružje, preuzeti su s tečaja „Awesome Tuts“ (https://www.awesometuts.com). Nadalje, balistika je kreirana pomoću YouTube videa (https://www.youtube.com/watch?v=d7pwmO6IS2I), a parametri projektila su podešeni na način da reprezentiraju gotovo jednake energije i brzine kao u stvarnom svijetu na adekvatnim udaljenostima.

U ovome dijelu većina skripti ili dijelovi skripti neće biti detaljno opisani, već će biti samo spomenute funkcionalnosti i zadaće skripti jer nema potrebe za daljnjim objašnjavanjem nekih od njih. Sustav oružja se sastoji od tri skripte: skripta za projektil („Projectile“), skripta oružja („Weapon“) i skripta koja upravlja oružjem („WeaponManager“). „WeaponManager“ skripta sadrži listu oružja koje igrač posjeduje, sadrži objekt koji predstavlja ciljnik, kameru i informacije o ciljanju (povećanje i vrijeme potrebno za dostizanje povećanja). Skripta provjerava korisnički input za pucanje (lijeva tipka miša), ciljanje (desna tipka miša), promjenu oružja (tipka „1“), punjenje oružja (tipka „R“). Osim što se provjerava korisnički input kod pritiska tipke, registrira se (provjerava) i trenutak puštanja tipke. Kao što znamo, sva oružja koja igrač posjeduje su referencirana u listi oružja te u skripti pratimo koje je trenutno aktivno (odabrano) oružje. Ukoliko igrač radi bilo koje radnje s oružjem, to se odvija na način da se u skripti poziva adekvatna metoda implementirana u samoj skripti oružja („Weapon“). Implementacija je takva

ponajprije zbog toga što svako oružje nije isto (ne koristi isti animator ni zvukove, itd.), ali i zbog toga da se napravi distinkcija između zadaća skripti. Zadaća „WeaponManager“ skripte je samo upravljati oružjem i slušati input s ulaznih jedinica (tipkovnice i miša) te reagirati na njih prijenosom odgovornosti na skriptu „Weapon“.

Spomenuta skripta „Weapon“ također prenosi odgovornost sa sebe u dijelu opaljivanja oružjem i to na način da samo instancira novi projektil na određenom mjestu i s određenom orijentacijom. Oružje se definira na način da mu se zadaju parametri o tipu oružja (automatsko ili jedinični pucanj), broju pucnjeva kroz vrijeme (engl. *Rate of fire*), veličina spremnika streljiva i ukupna maksimalna količina streljiva koju igrač može sadržavati. Oružju se, osim navedenog, dodaje odgovarajući projektil kojeg će instancirati, dodaju mu se zvukovi pucnja i podaci o povratnom udarcu oružja nakon pucnja (engl. *Recoil*). Za rad funkcionalnost ciljanja upravlja se animatorom na način da se postavi varijabla „Aiming“ na vrijednost „true“. Slično se odrađuje i za promjenu oružja, samo što je funkcija za promjenu oružja zapravo korutina koja čeka na završetak animacije za promjenu oružja. Što se tiče pucanja, slijedi prikaz koda koji se poziva kod istog.

```
public void Shoot(int shootHash, Transform cameraTransform, bool isAiming)
{
    if (!_animator || !_projectile || !_muzzlePoint)
        return;

    Quaternion shootingRotation = cameraTransform.rotation;
    if (!isAiming)
    {
        float xAngle = shootingRotation.eulerAngles.x;
        float yAngle = shootingRotation.eulerAngles.y;
        xAngle += UnityEngine.Random.Range(-3f, 3f);
        yAngle += UnityEngine.Random.Range(-3.5f, 3.5f);
        shootingRotation.eulerAngles = new Vector3(xAngle, yAngle,
                                shootingRotation.eulerAngles.z);
    }

    Instantiate(_projectile, _muzzlePoint.position, shootingRotation);
    --_currentAmmoInMagazine;

    _animator.SetTrigger(shootHash);
    PlayShootingSound();
}
```

Prvo se uzima trenutna rotacija kamere i vrijednost se zapisuje u „shootingRotation“ varijablu te se nakon toga provjerava da li igrač nišani ili puca bez ciljanja. Ukoliko igrač ne nišani, tada se varijabli „shootingRotation“ mijenjaju „X“ i „Y“ komponente na način da im se dodaju nasumične vrijednosti u rasponu od -3 do 3, odnosno od -3.5 do 3.5. Na ovaj način se pokušavaju simulirati greške koje nastaju ovakvim pucanjem iz oružja. Nakon ovih provjera,

odnosno kalkulacija (ukoliko su bile potrebne), slijedi instanciranje projektila. Projektil se instancira na poziciji cijevi oružja te mu se zadaje rotacija koju bi trebao imati kod početka kretanja, a to je „shootingRotation“. Nakon toga, iz spremnika se oduzima jedan metak i pokreće se animacija pucanja te se emitira zvuk opaljenja.

Što se tiče projektila i skripte „Projectile“, ona sadrži informacije o projektilu poput početne brzine u metrima po sekundi, mase projektila u gramima, balistički koeficijent i sila kojom projektil pomakne agenta kod pogotka. Osim ovih informacija, potreban je i podatak o broju koraka predikcije po slici (engl. *Frame*). Ova skripta je najviše zadužena zapravo za predikciju kolizije s objektom te zadavanje štete tom objektu ukoliko je on agent. Dakle, kolizija se ne detektira između objekata pomoću raznih komponenti, npr. „Sphere Collider“, jer je ovakav način prespor za projektile ovakve brzine. Iz navedenog razloga, kreirana je skripta koja radi predikciju pomoću stvaranja zrake nekoliko puta u svakoj slici te se tako detektira kolizija između zrake i objekta koji se projektilu nađe na putu.

```
private void FixedUpdate()
{
    Vector3 point1 = this.transform.position;
    RaycastHit hitInfo;
    float deltaTime = Time.fixedDeltaTime;

    _airDragVector = _bulletSpeed;
    _airDragVector *= _airDrag;

    float stepTimeLenght = _stepLength * deltaTime;
    Vector3 bulletSpeedSlowdown = (Physics.gravity + _airDragVector) *
stepTimeLenght;

    for (float step = 0; step <= 1f; step += _stepLength)
    {
        _bulletSpeed += bulletSpeedSlowdown;
        Vector3 point2 = point1 + _bulletSpeed * stepTimeLenght;

        Vector3 pointsDelta = point2 - point1;
        Ray ray = new Ray(point1, pointsDelta);
        if (Physics.SphereCast(ray, transform.localScale.x, out hitInfo,
                               pointsDelta.magnitude, _rayLayerMask))
        {
            GameObject target = hitInfo.collider.gameObject;

            if (target.layer == _bodyPartLayerHash)
            {
                float energyAtImpact = (_bulletWeightGrams *
                          Mathf.Pow(_bulletSpeed.magnitude, 2))/(2000.6f);
                DoDamage(ray, hitInfo, energyAtImpact);
                DestroyImmediate(this.gameObject, true);
                return;
            }
            else
            {
```

```
                DestroyImmediate(this.gameObject, true);
                return;
            }
        }
        point1 = point2;
    }
    this.transform.position = point1;

    float verticalPos = this.transform.position.y;
    if (verticalPos < 0 || verticalPos > 2000)
        DestroyImmediate(this.gameObject, true);
}
```

Glavni dio ove skripte odvija se unutar „FixedUpdate“ metode i jer se koriste kompleksne kalkulacije, a uz to i fizika (npr. gravitacija). Prvo što se odradi kod svakog poziva metode je određivanje trenutne početne točke „point1“. Sljedeće se postavlja trenutni otpor vjetra projektilu adekvatan trenutnoj brzini, ali pomnožen s konstantom koja predstavlja otpor zraka. Nakon toga je potrebno definirati koliko će projektil usporiti u trenutnoj slici i to na temelju zadanog broja koraka u predikciji. Ova vrijednost računa se na način da se kreira trodimenzionalni vektor „bulletSpeedSlowdown“ i postavi mu se vrijednost na zbroj gravitacije i trenutnog otpora vjetra projektilu te se dobivena vrijednost pomnoži sa dužinom koraka i „Time.fixedDeltaTime“. Dužina koraka je zapravo vrijednost broja koraka predikcije po slici s potencijom -1. Nakon toga, dolazi „for“ petlja koja kreće od nula do jedan i u svakoj iteraciji se povećava za prije spomenutu dužinu koraka. Trenutnoj brzini projektila se kod svake iteracije dodaje prije spomenuta vrijednost „bulletSpeedSlowdown“ kojoj su zapravo sve komponente negativne. Određuje se druga točka „point2“, odnosno završna točka u ovoj provjeri. Nakon toga se kreira zraka s početkom u „point1“ i duljinom koja je predstavljena razlikom između završne („point2“) i početne točke. Ukoliko je zraka detektirala koliziju i ukoliko je objekt kolizije u sloju „AIBodyPart“, kalkulira se energija koju će kod kolizije projektil prenijeti agentu te se agentu javlja da je pogođen. Agentu se javlja s koje strane je pogođen, koja je pozicija s koje je bilo pucano te energija u trenutku pogotka. Nakon toga ili ukoliko je projektil pogodio nešto što ne pripada sloju „AIBodyPart“, projektil se uništava. Ukoliko kod prošle provjere nije došlo do kolizije, sada završna točka postaje početna točka za sljedeću predikciju, odnosno sljedeći korak, a iteriranje se nastavlja.

4.3. Interaktivni objekti

Osnovna skripta kod interaktivnih objekata je „InteractiveItem“. Svaki interaktivni objekt mora sadržavati navedenu skriptu. Također, od iznimne je važnosti postaviti objekte u interaktivni sloj. Objekti zapravo ne sadrže spomenutu skriptu direktno, već neku drugu skriptu koja nasljeđuje, odnosno specijalizira klasu „InteractiveItem“. Neke specijalizacije ove klase su: „InteractiveInfo“, „InteractiveGenericSwitch“, „InteractiveDoor“, „InteractiveKeypad“, „InteractiveSound“ i „SceneSwitchDoor“. Također, postoje neke skripte koje služe sličnoj svrsi kako i ove prije navedene, ali ne nasljeđuju „InteractiveInfo“ skriptu, a to su: „CollisionGenericSwitch“ i „LockdownTrigger“. Ovakve skripte nisu direktno interaktivne jer ih igrač nesvjesno aktivira. Ovisno o potrebi i kontekstu, bira se adekvatna skripta koja će odrađivati posao. Primjerice „InteractiveInfo“ skripta nema nikakvu funkcionalnost osim da prikazuje tekst prilikom detekcije interaktivnog objekta, koristi se na objektima koji se ne mogu ni aktivirati ni deaktivirati. „InteractiveSound“ skripta služi sličnoj svrsi kao i „InteractiveInfo“, odnosno ne postoji nikakva akcija osim prikazivanja informacije o objektu, no kod ovakvih objekata, aktivira se zvuk kod pokušaja aktivacije skripte (kod zaključane kutije se samo emitira zvuk). Što se tiče „CollisionGenericSwitch“, skripta ima vrlo sličnu zadaću kao i „InteractiveGenericSwitch“, ali prva od navedenih skripti postavlja se na objekt koji samo treba detektirati koliziju te tako aktivira skriptu. Slijedi opis osnovne, najgeneralnije skripte („InteractiveItem“).

```
public class InteractiveItem : MonoBehaviour
{
    [SerializeField] protected int _priority = 0;
    protected GameSceneManager _gameSceneManager = null;
    protected Collider _collider = null;
    public int GetPriority()
    {
        return _priority;
    }
    public virtual string GetText() { return null; }
    public virtual void Activate(CharacterManager characterManager) { }
    protected virtual void Start()
    {
        _gameSceneManager = GameSceneManager.GetInstance();
        _collider = GetComponent<Collider>();
        if(_gameSceneManager && _collider)
        {
           _gameSceneManager.RegisterInteractiveItem(
                        _collider.GetInstanceID(), this);
        }
    }
}
```

Ova skripta je većim dijelom apstraktna, implementirane su samo dvije metode: „GetPriority“ i „Start“. Što se tiče „Start“ metode, potrebno ju je pozvati unutar „Start“ metode svake skripte koja ju nasljeđuje. U navedenoj metodi se svaki interaktivni objekt registrira u već prije spomenutoj skripti „GameSceneManager“ te se objekti spremaju u mapu interaktivnih objekata. Metode „GetText“ i „Activate“ potrebno je implementirati u skriptama koje nasljeđuju navedenu skriptu.

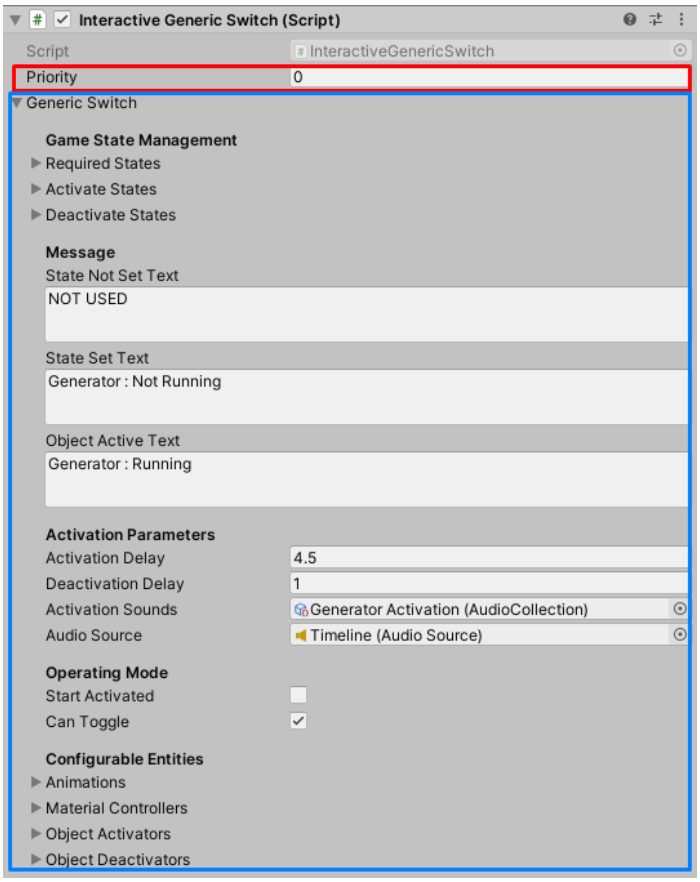
Jedna od najkompleksnijih skripti je „InteractiveGenericSwitch“, skripta sama po sebi zapravo nije toliko kompleksna, ali osnovni objekt koji ova skripta sadrži je tipa „GenericSwitch“ i on je čini kompleksnom. „GenericSwitch“ klasa, ujedno i skripta, sadrži sve važne funkcionalnosti i atribute potrebne za normalno funkcioniranje „InteractiveGenericSwitch“, ali i „CollisionGenericSwitch“. Slijedi prikaz skripte „InteractiveGenericSwitch“ koja se koristi kod aktivacije i deaktivacije generatora električne energije u tajnom bunkeru, a nakon toga i opis „GenericSwitch“ skripte. Razlog prikaza „InteractiveGenericSwitch“ skripte je taj što ćemo lakše uočiti korištenje ove druge skripte te kasnije i lakše shvatiti kako „GenericSwitch“ funkcionira u pozadini.

```
public class InteractiveGenericSwitch : InteractiveItem
{
    [SerializeField]
    private GenericSwitch _genericSwitch = new GenericSwitch();
    protected override void Start()
    {
        base.Start();
        _genericSwitch.OnStart();
    }
    public override string GetText()
    {
        if (!enabled)
            return string.Empty;
        return _genericSwitch.OnGetText();
    }
    public override void Activate(CharacterManager characterManager)
    {
        _genericSwitch.Activate(characterManager);
        if (_genericSwitch._coroutine != null)
            StopCoroutine(_genericSwitch._coroutine);
        _genericSwitch._coroutine = _genericSwitch.DoDelayedActivation();
        StartCoroutine(_genericSwitch._coroutine);
    }
}
```

Ova skripta sadrži, kako je bilo prije napomenuto, atribut tipa „GenericSwitch“ koji je serijaliziran (prikazan) u prozoru inspektora kako bismo mogli dodati potrebne atribute ovom objektu. Atributi ove skripte u alatu će biti prikazani na slici kasnije. U prvom dijelu koda kod „Start“ metode, možemo vidjeti kako se odmah u prvoj liniji poziva „Start“ metoda

„InteractiveItem“ klase. Na ovaj način se objekt koji sadrži skriptu registrira u kolekciji. Nakon navedenog se poziva „OnStart“ metoda „GenericSwitch“ objekta. Što se tiče metode „GetText“, također se poziva metoda iz „GenericSwitch“ objekta. Slijedi glavni dio skripte, a to je aktivacija. U „Activate“ metodi se poziva ponovno metoda iz prije spomenutog objekta, ali ovaj put metoda je „Activate“. Nakon navedenog, slijedi pokretanje posla (engl. *Coroutine*) koji upravlja animacijama koje se pokreću aktiviranjem ili deaktiviranjem objekta. Slika ispod (Slika 35) prikazuje komponentu koja se nalazi na objektu generatora te služi za upravljanje stanjima i objektima kod aktiviranja ili deaktiviranja objekata. U plavo označenom dijelu na slici, nalaze se svi atributi „GenericSwitch“ klase, a na samom dnu postoji atribut „Animations“. U navedenom atributu nalaze se spomenute animacije koje se pokreću u „Activate“ metodi pomoću posla (engl. *Coroutine*). Taj objekt, odnosno atribut je zapravo lista tipa „AnimatorConfigurator“ koja u sebi sadrži još nekoliko dodatnih atributa, a neki od njih su: animator u kojem se izvršava animacija i lista parametara za upravljanje animacijama. Zadnje navedeni parametar (lista) sadrži skup objekata tipa „AnimatorParameter“. Navedena klas sadrži parametre koji označuju tip i ime varijable definirane u animatoru nad kojim se vrši promjena, te sadrži i vrijednost na koju će se varijabla postaviti.

Kao što je već bilo spomenuto u tekstu iznad, plavi dio označava atribute „GenericSwitch“ klase (skripte), a crveni dio, odnosno atribut prioriteta (engl. *Priority*) dolazi iz skripte „InteractiveItem“. Skripta se sastoji od nekoliko dijelova (skupina atributa): Upravljanje stanjem igre (engl. *Game State Management*), upravljanje poruka za igrača (engl. *Message*), parametri aktivacije (engl. *Activation Parameters*), način rada (engl. *Operation mode*), podesivi entiteti (engl. *Configurable Entities*). Prva skupina atributa koristi se za postavljanje stanja igre koja moraju biti zadovoljena da bi se objekt mogao aktivirati. Uz navedeno, sadrži i stanja koja će biti promijenjena prilikom aktivacije i vrijednosti koje će poprimiti, ali sadrži isto i za stanja kod deaktivacije. Sljedeća skupina služi za definiranje poruka koje se prikazuju korisniku ukoliko postoje nezadovoljena stanja za aktivaciju, tekst prije i nakon aktivacije objekta. Unutar aktivacijskih parametara, nalaze se opcije za podešavanje kašnjenja aktivacije i deaktivacije (najčešće radi animacija) te još dodatno i zvukovi aktivacije uz pripadajući izvor koji ih emitira. Nadalje, postoji dio u kojem se određuje početno stanje objekta (aktivan ili neaktivan) te opcija koja određuje može li se objekt uključivati i isključivati ili samo jedno od navedenog.



Slika 35. Interactive Generic Switch komponenta

Na kraju, nalazi se atribut koji je već bio opisan u tekstu iznad, a riječ je o animacijama kod aktivacije objekta. Atribut „Material Controllers“ omogućuje promjenu svojstava raznih materijala (kod aktivacije) koji se kod deaktivacije objekta vraćaju u originalne postavke. Posljednja dva atributa sadrže skup objekata koji će se aktivirati te skup objekata koji će se deaktivirati kod aktivacije interaktivnog objekta.

Ostale skripte koje nasljeđuju „InteractiveItem“, pa čak i one koje ne nasljeđuju, rade na gotovo jednak način kao gore opisana skripta, prema tome nema ih potrebe opisivati.

4.4. Animator

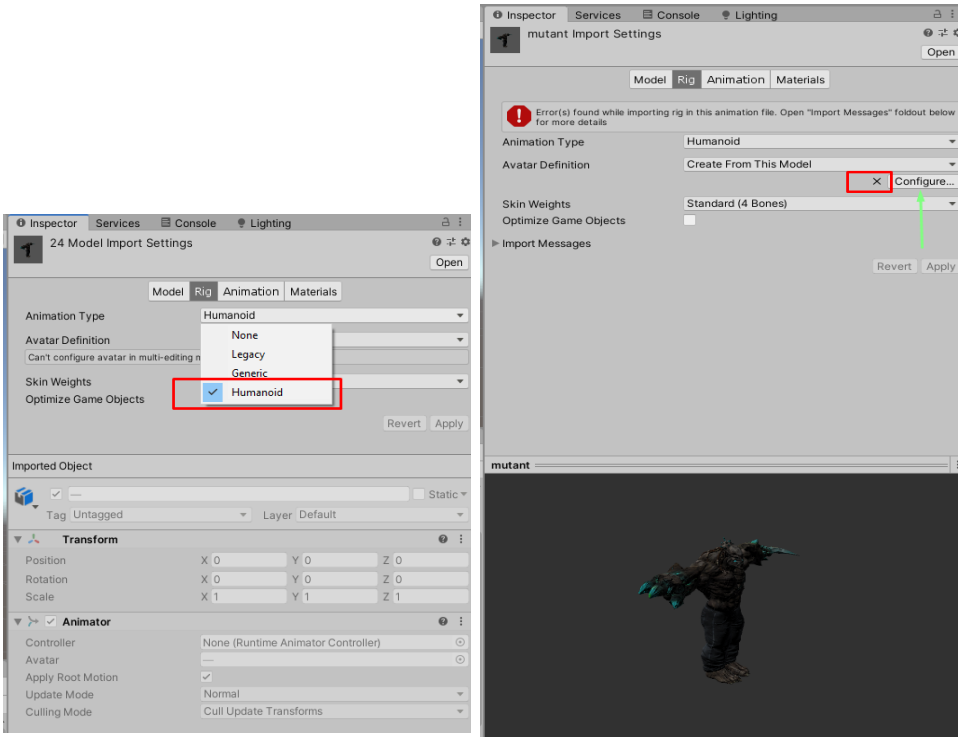
U ovom poglavlju bit će riječi o animacijama i kako se pomoću animatora one pokreću te upravljaju modelom. Animacije, modeli i sam animator će ponajviše biti fokusirani na agente (u ovom slučaju zombije). Već smo se u radu dotakli animatora te prikazali skriptiranje istog, ali nije bilo objašnjeno na nižoj razini, navedeno je zapravo cilj ovog poglavlja. Kako bismo

mogli uopće koristiti animacije nad nekim modelom, potrebno je podesiti model na ispravan način. Osim navedenog, u ovome dijelu će biti prikazano osnovno upravljanje animacijama, tj. podešavanje animacija. Govoreći o animiranju, vrlo je važno spomenuti slojeve koji se mogu nadodati animatoru.

Važno je napomenuti da je većina animacija i likova (modela) korištenih za izradu ovog rada preuzeto s Mixamo web stranice (https://www.mixamo.com). Osim što se može upravljati i preuzeti tuđe animacije, moguće je i snimiti svoje. U kreiranoj igri postoje neke vlastito snimljene animacije, a to su: animacija ciljanja i simulacija udarca puške kod opaljenja (engl. *Recoil*) te animacija pokretanja dizala. Vlastito snimanje animacija nije od velike važnosti za ovaj rad i za samu igru, prema tome neće ni biti odrađeno u tekstu.

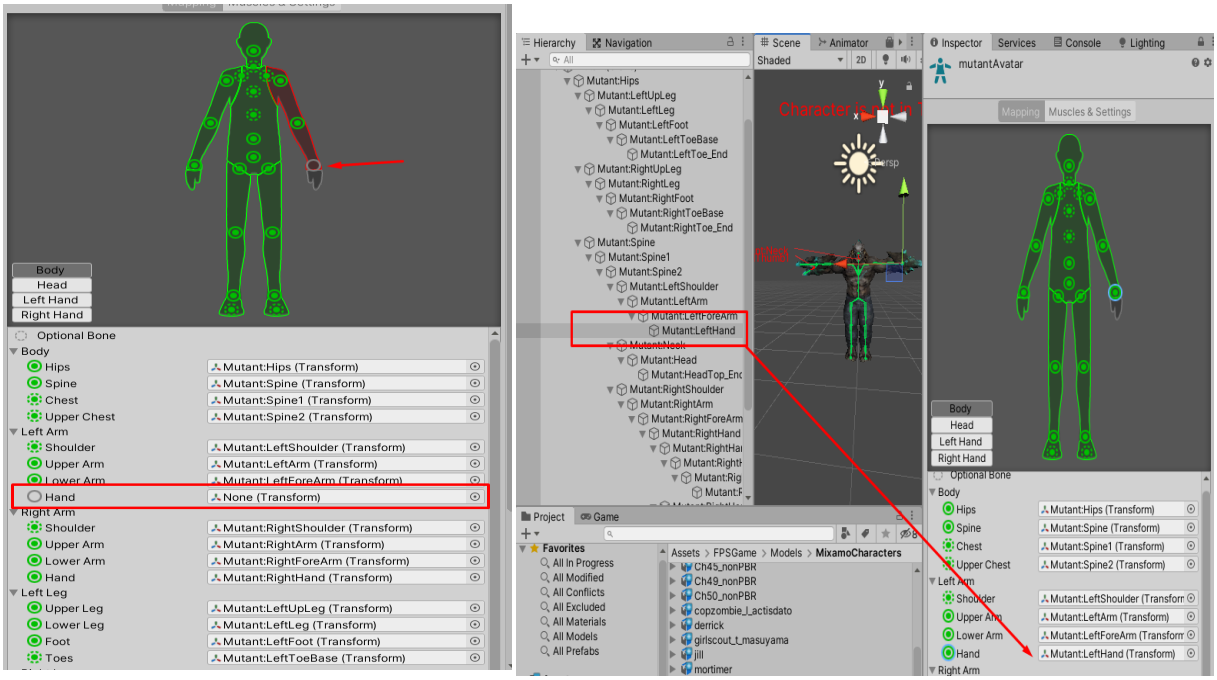
4.4.1. Priprema modela i animacije

Kako bi animacije određenog modela funkcionirale, model mora odgovarati animacijama. U ovome radu najveći je fokus na humanoidne modele i animacije. Kod preuzimanja ili pregledavanja dostupnih modela i animacija, odmah znamo radi li se o humanoidnim tipovima (modeli imaju ruke, noge, tijelo, glavu – poput čovjeka) te je potrebno podesiti ove postavke u Unityju. Slijede slije koje prikazuju postavljanje modela i rezultat istog (Slika 36 lijevo i desno). Podešavanje se zapravo radi unutar kartice „Rig“.



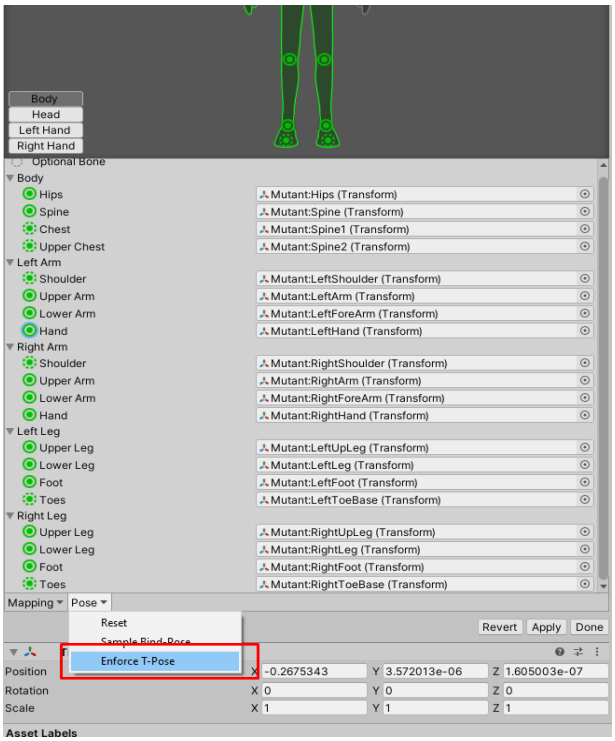
Slika 36. Postavljanje tipa modela (lijevo) i rezultat postavljanja (desno)

Slika lijevo prikazuje pretvaranje odabranih modela (u ovom slučaju 24 modela) u tip „Humanoid“, dok desna prikazuje jedan odabrani model nakon podešavanja tipa. Fokusirajmo se sada na desnu sliku i uočimo kako postoji greška kod modela te je i crvenom bojom označen dio koji prikazuje oznaku „X“. Ovakva oznaka prikazuje da model nije korektno definiran i potrebno ga je ručno konfigurirati odabirom opcije „Configure…“ (zelena strelica). Odabirom ove opcije otvara se prozor kao na slici koja slijedi.



Slika 37. Konfiguracija modela (lijevo) i dodavanje reference koja nedostaje (desno)

Slika lijevo prikazuje jednu od pogreški zbog kojih je nastao problem predstavljen u tekstu iznad. Na slici se može uočiti kako humanoidu nedostaje referenca jednog dijela ruke. Kod humanoida ruka se sastoji od četiri dijela, u ovome slučaju su tri dijela automatski referencirana, ali nedostaje referenca na početak šake (zglob). Referenca se jednostavno može dodati iz hijerarhije samog modela (Slika 37 desno). Nakon odrađenih akcija, ukoliko kostur samog modela (Slika 37 desno, model pokraj crvenog pravokutnika) nema crveno označenih dijelova možemo spremiti promjene i provjeriti postoji li još uvijek oznaka „X“ kao na slici prije (Slika 36 desno). Ukoliko se model (lik) ne nalazi u „T“ pozi (kao na slici iznad desno) animacije neće u potpunosti funkcionirati prema očekivanju, na sreću postoji lagano rješenje. Jedno rješenje se nalazi na slici koja slijedi (Slika 38), no ukoliko i ovo ne pomogne (u većini slučajeva se problem u potpunosti ovako riješi), može se model ručno podesiti u ovu pozu.

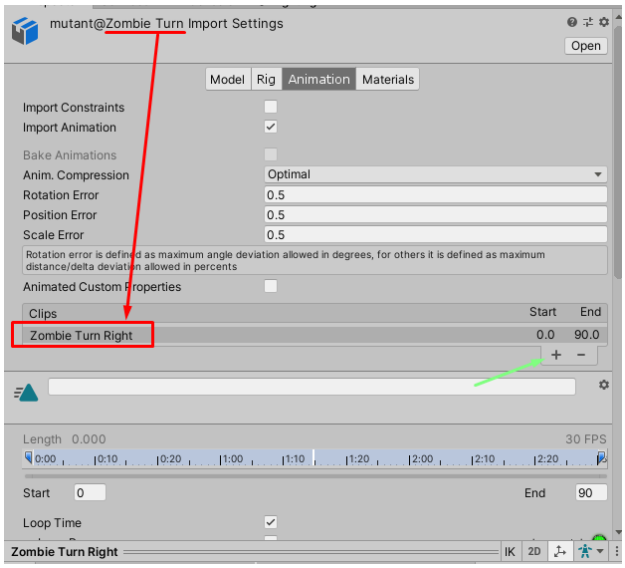


Slika 38. Postavljanje modela u „T“ pozu

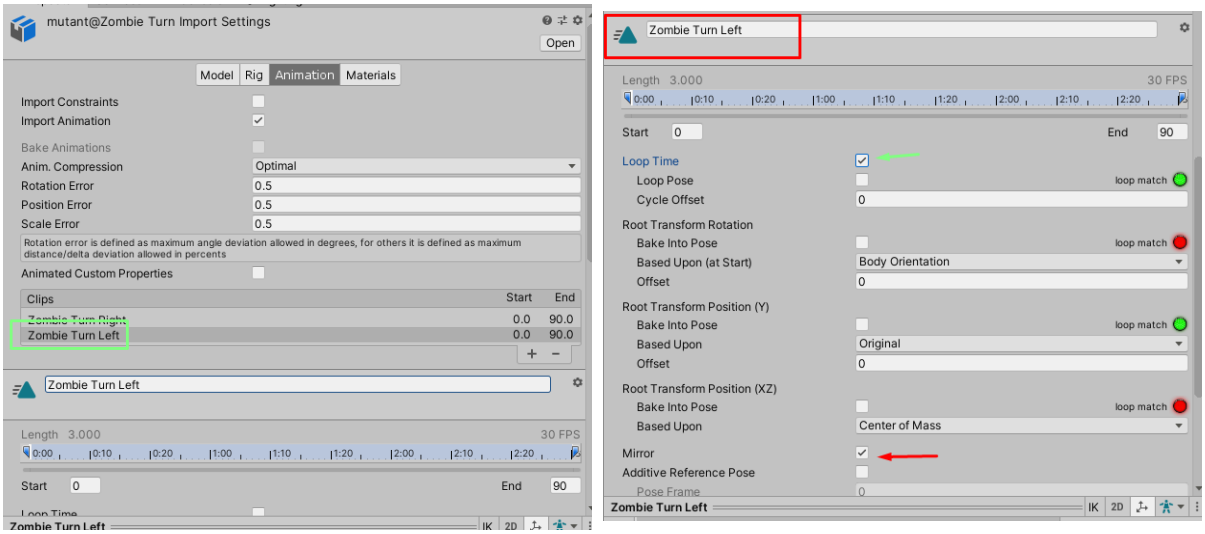
Krenimo sada na animacije i neke od osnovnih mogućnosti kod upravljanja animacijama. U Unityju je objekt animacije zapravo predstavljen kao skup svih animacija koje pripadaju tom objektu. Drugim riječima, animacija okretanja modela u lijevu stranu i animacija okretanja u desnu stranu pripadaju objektu koji se zove „okretanje“.

Na slici ispod (Slika 39) možemo vidjeti objekt „mutant@Zombie Turn“ i jednu njegovu animaciju (video klip, engl. *Clip*) koja se zove „Zombie Turn Right“ (označeno crvenom bojom). Nadalje, odabirom znaka „+“ (označeno zelenom bojom) možemo dodati novu animaciju u ovaj objekt. Ovako dodana animacija bit će identična već postojećoj, ali ju možemo promijeniti prema našim potrebama. Primjerice, na slici ispod prikazuje se kreiranje animacije skretanja u lijevu stranu iz već postojeće animacije skretanja u desnu stranu (vidi Slika 40). Navedena slika (Slika 40 lijevo) prikazuje novokreiranu animaciju koja je identična animaciji „Zombie Turn Right“, ali je preimenovana u „Zombie Turn Left“. Podešavanje ove animacije da stvarno radi ono što joj ime sugerira, a to je okretanje modela u lijevu stranu, nalazi se na slici desno (Slika 40 desno). Zelena strelica na slici prikazuje da bi opcija „Loop Time“ trebala biti aktivirana (barem što se tiče ove animacije). Ova opcija omogućava da se animacija ponovno pokrene ispočetka, ali s zadržavanjem trenutne pozicije modela (agenta). Ovo je korisno kod višestrukog okretanja agenta, primjerice ukoliko se agent treba okrenuti za 360 stupnjeva, a

potrebno je više okreta za dostizanje ovog učinka. Također, u ovome primjeru vrlo je važno odabrati opciju „Mirror“ (označeno crvenom strelicom). Ova opcija zrcali animaciju te se zbog toga ona ne izvodi više u desnu stranu, već u lijevu.



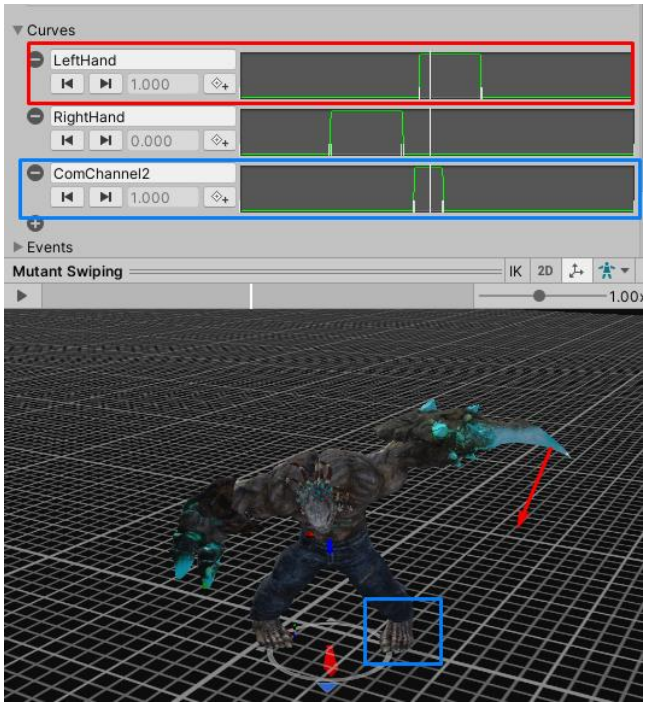
Slika 39. Objekt "Zombie Turn" s jednom animacijom



Slika 40. Novokreirana animacija (lijevo) i podešavanje animacija (desno)

Postoji mnogo više opcija za rad s animacijama, primjerice može se iz jedne animacije kreirati identična animacija, ali ubrzana, usporena ili da se izvodi unatrag. Moguće je uzeti samo dio (ili dijelove) neke animacije i na taj način iz jedne duge animacije kreirati mnogo kraćih. Na animacije se mogu dodavati krivulje i događaji, a to su vrlo korisne funkcionalnosti. Kod implementacije zvukova hodanja agenata, zvukova vikanja i definiranja kada neki dio tijela

agenta predstavlja opasnost za igrača, korištena je funkcionalnost krivulja. Slijedi prikaz istoga pomoću slike i kratkog opisa.



Slika 41. Krivulje na animaciji napada

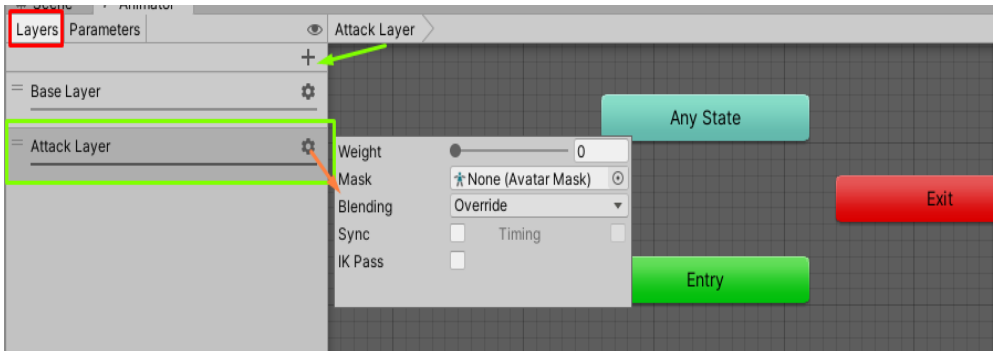
Krivulje i događaji (engl. *Events*) kod animacija su gotovo ista stvar, ali su krivulje jednostavnije. Događaji ili krivulje kod animacija određuju vremenske intervale i trenutke u animaciji kada se odvija (ili dogodi) nešto što je potrebno dalje razraditi. U ovakvim događajima, određene varijable poprimaju zadane vrijednosti. Razlika između krivulja i događaja je ta da krivulja može zadati kod jedne krivulje vrijednost samo jednoj varijabli, a za upravljanje većim brojem varijabli je potrebno kreirati više krivulja. Događaji su malo napredniji od krivulja i omogućuju pozivanje funkcije u trenutku nekog događaja.

Krenimo sada na praktični opis krivulja prema slici iznad (Slika 41). Crvenom bojom označeno je sve što ima veze s lijevom rukom, dok plava boja označuje pokret kod kojeg se emitira zvuk koraka. Kod crveno označenog kvadrata nalazi se krivulja (graf zelene boje) koja prikazuje vrijednosti varijable „LeftHand“ od početka do kraja trajanja animacije. Unutar pravokutnika je prikazana i bijela vertikalna linija koja prikazuje trenutno vrijeme animacije. Gledajući ovu liniju možemo vidjeti kako je animacija trenutno na samom početku promjene vrijednosti varijable „LeftHand“. Također, kod modela (agenta) je prikazana crvena strelica koja označuje pokret lijevom rukom, odnosno zamah u smjeru strelice. U navedenom trenutku animacija počinje odrađivanje napada te se vrijednost varijable „LeftHand“ iz 0 mijenja u 1.

Vrijednost varijable može se očitati u skriptama te tako možemo znati ukoliko se je dogodila, za igrača, štetna kolizija s agentovom lijevom rukom. Dakle, ako se dogodi kolizija, važno je otkriti je li ona štetna za igrača ili ne, ako je vrijednost varijable 1 tada se šteta računa, inače ne. Što se tiče plavo označenog dijela, on prikazuje krivulju za varijablu „ComChannel2“ te ova varijabla služi za emitiranje zvuka. Ova varijabla se postavlja na vrijednost 1 malo prije varijable lijeve ruke, jer baš u tom trenutku agent napravi mali korak unaprijed. U ovom trenutku, emitira se zvuk koraka.

4.4.2. Slojevi animatora i maska sloja

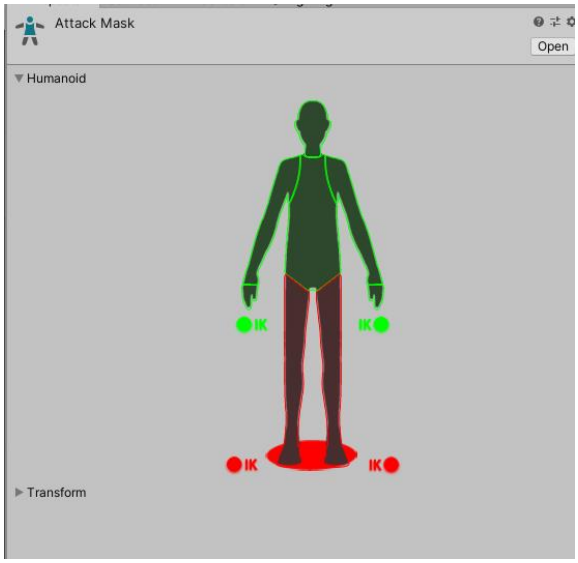
Osnovni dio svakog animatora je sloj, a u svakom sloju se može kreirati logika za upravljanje animacijama. Kod kreiranja animatora već postoji jedan osnovni sloj koji se zove „Base Layer“. Osim tog sloja, mogu se kreirati i drugi koji upotpunjuju osnovni sloj. Slijedi slika koja prikazuje dodavanje novog sloja, izgled novokreiranog sloja te izgled prozora animatora.



Slika 42. Slojevi animatora

Dodavanje sloja odrađuje se odabirom kartice „Layers“ (označeno crvenom bojom), a nakon toga klikom na „+“ (zelena strelica). Na ovaj način se kreira novi sloj, u ovom slučaju sloj je preimenovan u „Attack Layer“. Odabirom postavki za taj sloj otvaraju se opcije (vidi narančastu strelicu) u kojima se može podesiti koliko ovaj sloj utječe na slojeve ispod sebe (engl. *Weight*). Osim navedenog podešavanja, sloju se može dodati maska i način stapanja (engl. *Blend*) sa slojevima ispod njega. Način stapanja može biti: nadjačavanje (engl. *Override*) ili aditivno (engl. *Additive*). Dodatne mogućnosti su omogućavanje sinkronizacije ovog sloja s nekim drugim te opcija omogućavanja IK prolaska (engl. *IK Pass*). IK prolazak omogućuje korištenje „OnAnimatorIK“ metode, osim toga omogućuje da se preko skripte odredi primjerice smjer gledanja agenta kod obavljanja neke animacije. Nadalje, novokreirani sloj je prazan što se tiče animatorovih stanja (engl. *States*).

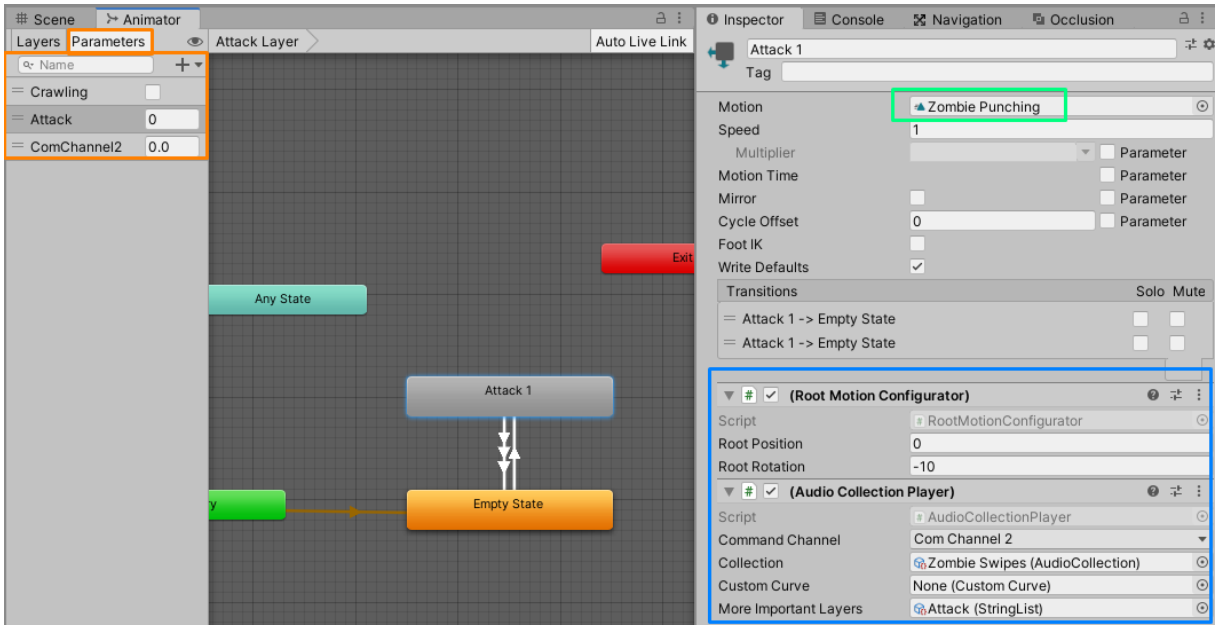
U ovome poglavlju nećemo odraditi sve slojeve koji su kreiranu u svrhu igre već samo jedan sloj i to „Attack Layer“, odnosno sloj napadanja. Prvo će biti objašnjeno kreiranje stanja i tranzicije iz stanja u stanje, a na kraju će biti prikaz završenog animatora sa svim slojevima. U sloju „Base Layer“ (sloj koji se nalazi ispod sloja napadanja) nalazi se logika kretanja agenata. Kako se sloj napadanja nalazi iznad sloja kretanja i kompletno ga nadjačava, pojavljuju se neki problemi. Ti problemi su vidljivi tek kada agent pokušava trčati i napadati u isto vrijeme. Naime takva radnja je za njega nemoguća jer animacije napadanja imaju veći prioritet od hodanja (hijerarhija slojeva – slojevi koji se u hijerarhiji nalaze niže su važniji). Na sreću ovaj problem se može riješiti na vrlo jednostavan i interesantan način, a to je korištenjem maske sloja. Maska sloja (engl. *Avatar Mask*) predstavlja dijelove tijela humanoida koje promatrani sloj može koristiti za svoje animacije, odnosno koje ne može. Slijedi slika maske sloja za sloj napadanja.



Slika 43. Maska sloja

Dakle, slika maske sloja odnosi se na sloj napadanja, a na liku humanoida možemo vidjeti dijelove koji su zeleni i one koji su crveni. Zeleni dijelovi su na promatranom sloju dozvoljeni, a crveni nisu. Naravno, logično je da sloj za napadanje koristi samo gornji dio tijela, dok svi slojevi ispod ovog sloja mogu slobodno koristiti sve one druge dijelove koji su preostali. Ova posljednja rečenica je uvjetna, zapravo svi slojevi ispod ovog sloja mogu slobodno koristiti cijelo agentovo tijelo, ali samo dok je navedeni sloj u praznom stanju (ne koristi se). Jednom kada se aktivira (koristi) sloj kojem ova maska pripada tada se svim nižim slojevima zabranjuje koristiti zeleno označene dijelove tijela.

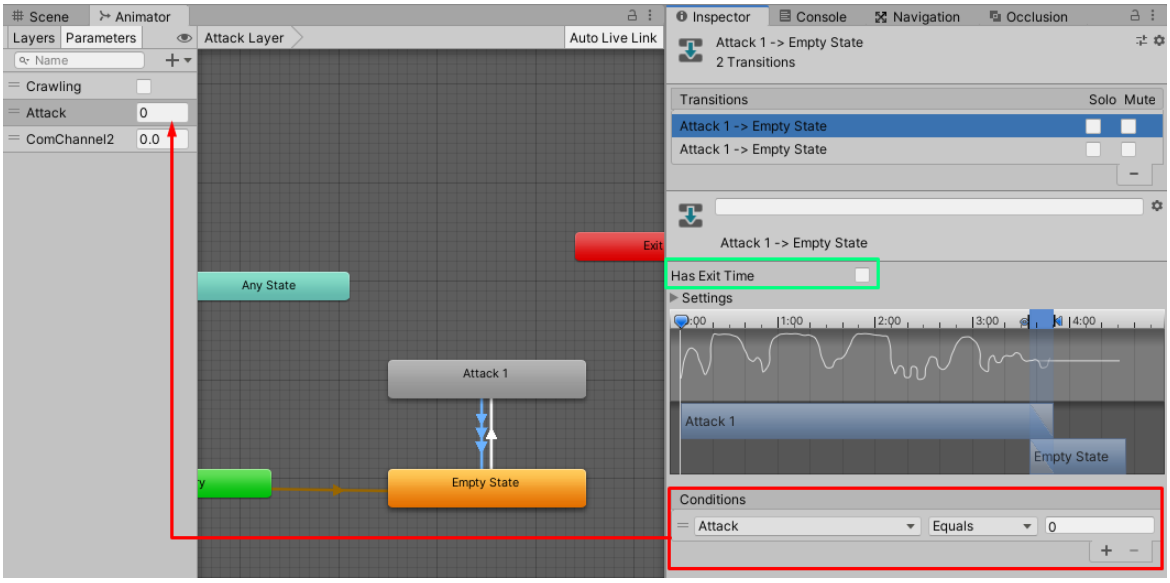
Nakon dodavanja maske, možemo krenuti na kreiranje stanja sloja za napadanje. Slijedi slika koja prikazuje potrebne komponente, animaciju i parametre za kreiranje sloja, a sama slika će biti opisana detaljnije.



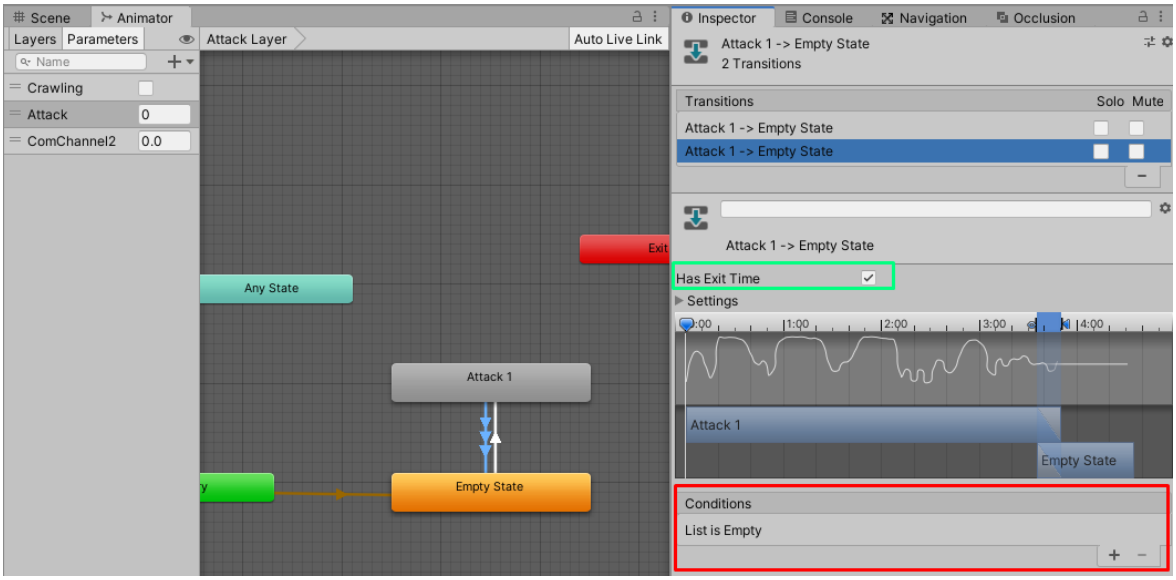
Slika 44. Početna stanja sloja napadanja

Slika prikazuje sloj napadanja s dva stanja: prazno stanje (engl. *Empty State*) i „Attack 1“. Sloj se nalazi u praznom stanju onda kada je neaktivan, odnosno kada agent ne napada. Označeno narančastom bojom (lijevo), nalaze se parametri potrebni za kreiranje logike tranzicija između stanja. Desno, označeno zelenom bojom, nalazi se referenca na animaciju koja se izvršava kod ulaska u stanje „Attack 1“. Što se tiče plavo označenog dijela, ovdje se nalaze skripte koje nećemo detaljno objašnjavati. Skripta „Audio Collection Player“ služi za osluškivanje vrijednosti varijable „ComChannel2“ kako bi se emitirao zvuk zamaha rukom, dok druga skripta služi za upravljanje pozicijom i rotacijom kod ulaska i izlaska u ovo stanje („Attack 1“). Strelice između dva navedena stanja predstavljaju tranzicije, odnosno logiku prelaska iz jednog stanja u drugo. Ove tranzicije određene su uvjetima koji se postižu kontroliranjem parametara animatora. Slijede dvije slike koje prikazuju uvjete prelaska iz stanja „Attack 1“ u prazno stanje („Empty State“). Obratimo prvo pozornost na izgled strelica tranzicije (Slika 45), tranzicija označena plavom bojom je ona koja se trenutno prikazuje u prozoru inspektora. Dalje možemo vidjeti kako je plavo označena tranzicija zapravo skup triju strelica, a bijela se sastoji samo od jedne. Razlog tome je taj što se označena tranzicija sastoji od više tranzicija, ne jedne tranzicije od više uvjeta nego od više tranzicija koje su grupirane iz tog stanja u ciljano stanje.

Korištenjem ovakvog skupa tranzicija postiže se kreiranje logičkih ILI vrata (engl. *OR gate*), dok skup uvjeta (označeno crvenom bojom na slikama) predstavlja logička I vrata (engl. *AND gate*).



Slika 45. Tranzicija iz „Attack 1“ u prazno stanje s uvjetom

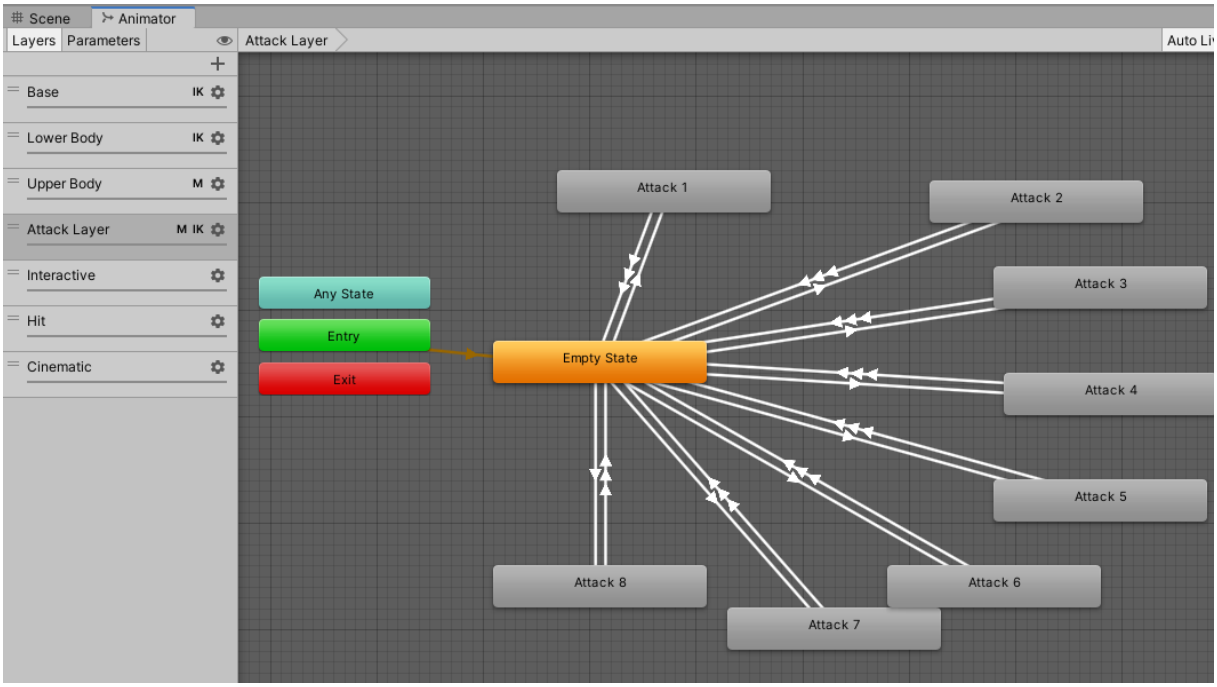


Slika 46. Tranzicija iz „Attack 1“ u prazno stanje bez uvjeta

Dakle, ILI vrata omogućuju promjenu stanja ukoliko je zadovoljena barem jedna od tranzicija čiji su svi uvjeti zadovoljeni. Plavo označena tranzicija, odnosno skup tranzicija, sastoji se od dvije tranzicije (Slika 45 i Slika 46). Kod prve slike (Slika 45), tranzicija će se okinuti samo onda kada varijabla „Attack“ iz skupa parametara poprimi vrijednost 0 (crveno označeni dijelovi na slici). Što se tiče zeleno označenog dijela na prvoj slici, opcija „Has Exit Time“ trebala bi biti

isključena kako animator ne bi čekao završetak animacije. Ovo je iznimno važno iz razloga kako bi se animacija zaustavila (dogodila tranzicija) odmah kada se ispuni uvjet, a ne u trenutku kada se ispuni uvjet i kada animacija završi. Na drugoj slici je situacija malo drugačija, uvjeta završetka animacije uopće nema, ali je „Has Exit Time“ opcija uključena. To znači da će se u ovom slučaju tranzicija okinuti samo kada animacija završi. Kada bi se obje tranzicije gledale zajedno u jednoj široj perspektivi, promjena stanja iz „Attack 1“ u prazno stanje će se dogoditi ili ukoliko animacija završi samostalno ili ukoliko je varijabla „Attack“ poprimila vrijednost 0. Što se tiče tranzicije iz praznog stanja u „Attack 1“, postoje 3 uvjeta: da je vrijednost parametra „Attack“ veće od 0, ali i manje od 21 te da agent ne puže. Agent generira slučajni broj od jedan do sto i ta vrijednost se zadaje parametru „Attack“, a kako u završnom sloju imamo više različitih napada (stanja), svako stanje ima svoj raspon i prema tome vjerojatnost izvršavanja tog napada.

Slijedi slika svih slojeva na završenom animatoru sa prikazom stanja u sloju napada.



Slika 47. Završni prikaz animatora

4.4.3.Skriptiranje *animatora*

Pod pojmom skriptiranja animatora podrazumijeva se upravljanje vrijednostima parametara animatora koji su bili spomenuti u prošlom poglavlju. Pod upravljanjem parametara spada promjena vrijednosti parametra te čitanje njegove trenutne vrijednosti. Svaki agent, zapravo bilo koji objekt, može imati svoj animator koji može biti različit od svih drugih. Iz tog

razloga za upravljanje animatorom pomoću skripte potrebna je referenca na taj animator. Osim reference na animator, potrebno je znati ime i tip varijable kojom želimo upravljati kroz skriptu. Najčešće se kod pokretanja skripte („Start“ metoda) iz statične klase animatora, na temelju imena varijable, generira jedinstvena *hash* vrijednost tražene varijable i ista se sprema u memoriju. Svaki animator kod kreiranja parametra unutar prozora animatora radi to isto. Na ovaj način animator može puno brže određivati o kojoj je varijabli riječ jer se jednostavno uspoređuju *hash* vrijednosti teksta (cijeli broj) nego da se provjerava sam tekst (ime varijable). Slična stvar kao i s parametrima se može učiniti s imenom sloja, samo što se sloj čita iz animatora i pretvara u indeks koji označuje njegovu poziciju unutar hijerarhije slojeva tog animatora. Često se navedena vrijednost koristi za upravljanje slojem na način da se primjerice podešava utjecaj promatranog sloja na ostale (engl. *Weight*). Slijede neki dijelovi skripte „AIZombieStateMachine“ koji prikazuju upravljanje animatorom.

```
…
private readonly int _speedHash = Animator.StringToHash("Speed");
private readonly int _feedingHash = Animator.StringToHash("Feeding");
private readonly int _attackHash = Animator.StringToHash("Attack");
…
protected override void Update()
{
    base.Update();

    if (_Animator)
    {
        _Animator.SetFloat(_speedHash, _speed);
        _Animator.SetBool(_feedingHash, _feeding);
        _Animator.SetInteger(_seekingHash, _seeking);
        _Animator.SetInteger(_attackHash, _attackType);
        _Animator.SetInteger(_stateHash, (int)_currentStateType);
        if (_hasSpecialMoves)
        {
            _Animator.SetInteger(_specialMoveHash, _specialMoveType);
        }
        _isScreaming = IsLayerEnabled("Cinematic") ?
                        0f : _Animator.GetFloat(_screamingHash);
    }
   …
}
```

U prve tri linije koda vidimo kreiranja *hash* vrijednosti za tri različita parametra animatora. S parametrom „Attack“ smo se već susreli u prošlom poglavlju. Nakon pohranjivanja ovih vrijednosti, iste su pohranjene u varijable. Navedene varijable sadrže ove vrijednost do kraja životnog ciklusa skripte i nema potrebe za ponovnim kreiranjem novih vrijednosti unutar tog ciklusa. Unutar „Update“ metode, provjerava se postoji li ispravna referenca na animator („_Animator“). Ukoliko postoji, postavljaju se vrijednosti svim navedenim parametrima ovog animatora. Ove vrijednosti su mijenjane od strane drugih skripti, točnije od skripti ponašanja

(stanja). Primijetimo kako za postavljanje vrijednosti bilo kojem parametru moramo pozvati animatorovu metodu koja se odnosi na tip varijable kojoj će vrijednost biti dodijeljena. Primjerice ukoliko je parametar tipa cijeli broj (engl. *Integer*), poziva se metoda „SetInteger“, ukoliko je decimalni broj „SetFloat“, itd. Nadalje, nakon postavljanja svih vrijednosti, čita se decimalna vrijednost parametra „Screaming“ koja se odnosi na agentovo glasanje (vikanje). Ukoliko je „Cinematic“ sloj omogućen, čita se decimalna vrijednost s animatora pomoću „GetFloat“ i ona se zapisuje u varijablu „_isScreaming“.

4.5. Upravljanje igrom i scenama

U ovome poglavlju će biti riječi o jednostavnim skriptama koje su se već spominjale u prethodnim poglavljima. Skripte o kojima će se govoriti su „ApplicationManager“ i „GameSceneManager“. Obje navedene klase su izvedene kao *singleton* uzorak dizajna, a skripta „ApplicationManager“ se dijeli kroz scene, dok se druga skripta kod učitavanja scene popunjava ispočetka.

Krenimo prvo s „ApplicationManager“ skriptom te njezinim atributima i funkcionalnostima. Jedini javno dostupni atribut koji se može postaviti prije samog pokretanja igre je lista početnih stanja u igri. Ova lista sadrži objekte klase „GameState“. Navedena klasa ne sadrži funkcionalnosti, već samo dva atributa: ključ (ime stanja) i vrijednost. Slijedi prikaz „Awake“ metode navedene skripte.

```
private void Awake()
{
    DontDestroyOnLoad(gameObject);
    ResetGameStates();
    _audioManager = FindObjectOfType<AudioManager>();
}
```

Prisjetimo se da je „Awake“ metoda pokrenuta samo jednom u životnom ciklusu skripte na određenom objektu i to kod učitavanja scene u kojoj se prvo pojavljuje. U navedenoj skripti je odmah kod inicijalizacije određeno da se objekt na kojem se skripta nalazi ne smije uništiti kod promjene scena (više se nikad ne poziva „Awake“ metoda). Nadalje, poziva se metoda za resetiranje stanja igre. Metoda ima jednostavni posao, a to je da prođe kroz listu stanja (javno dostupni atribut) te iz svakog objekta u ovoj listi uzme ključ i njegovu vrijednost i postavi ih u mapu (rječnik). Osim toga, pronalazi se objekt koji sadrži skriptu „AudioManager“ (također *singleton*) koja služi za upravljanje zvukom. Ostale funkcionalnosti ove skripte su dohvaćanje vrijednosti željenog stanja igre iz mape (rječnika), postavljanje vrijednosti postojećeg stanja na temelju ključa u neku drugu vrijednost (ili kreiranje novog stanja). Također, osim navedenog,

skripta ima mogućnost na temelju dane liste tipa „GameState“ provjeriti jesu li navedena stanja u rječniku postavljena. Skripta može učitati glavni izbornik, učitati početnu scenu (šuma), učitati scenu na temelju imena ili pak ugasiti igru.

Kako maloprije navedena skripta sadrži stanja igre koja se dijele kroz sve scene, tako skripta „GameSceneManager“ sadrži popis objekata koji su vezani za trenutnu scenu. Slijedi opis „GameSceneManager“ skripte, ali i prikaz podataka koje ona sadrži.

```
private Dictionary<int, AIStateMachine> _stateMachines;
private Dictionary<int, PlayerInfo> _playerInfos;
private Dictionary<int, InteractiveItem> _interactiveItems;
private Dictionary<int, MaterialController> _materialControllers;
```

Dakle, atributi koje ova skripta sadrži, o kojima vodi brigu i pruža sučelja za dohvaćanje podataka istih su zapravo mape (rječnici). Svaki od ovih rječnika ima ključ tipa broj (engl. *Integer*), a sprema se jedinstveni identifikator objekta. Mape se koriste za spremanje velikog broja objekata iz razloga što je često potrebno dohvatiti točno određeni objekt. Dohvaćanje točno određenog objekta je moguće i pomoću liste, ali tada kompleksnost istog raste s porastom broja objekata u listi. Ukoliko se objekt traži kroz scenu pomoću metoda klase „MonoBehaviour“ tada je opet potrebno koristiti mnogo računalnih resursa za pronalazak objekta (prvo se traže objekti točno određenog tipa, a nakon toga onaj objekt kojeg trebamo dohvatiti). Iz navedenih razloga se kod učitavanja scene objekti sami registriraju u rječnike. Iz registriranih objekata se uzimaju jedinstveni identifikatori i pohranjuju u rječnike. Rječnici se pretražuju prema ključu (jedinstvenom identifikatoru) i uvijek postoji samo jedna vrijednost za navedeni ključ. Kompleksnost pretrage rječnika se ne mijenja porastom broja objekata u rječniku. Dakle, rječnici koje ova skripta sadrži su: agenti, informacija o igračima (ukoliko bi igra bila namijenjena za više igrača), interaktivni objekti i popis materijala kojima će se mijenjati parametri (pomoću „InteractiveGenericSwitch“ skripte). Svi objekti navedenih tipova („AIStateMachine“, „PlayerInfo“, „InteractiveItem“ i „MaterialController“) se moraju registrirati kod „GameSceneManager“ skripte kako bi sve pravilno funkcioniralo. Ovo registriranje objekata se najčešće odrađuje u „OnStart“ metodi svake skripte navedenog tipa. Svi ovi atributi (rječnici) su privatne varijable kako ne bi mogle biti promijenjene izvana (iz druge skripte ili klase) već samo pomoću ove skripte. Zbog toga što su varijable privatne, potrebna su javna sučelja koja osiguravaju drugim skriptama dohvaćanje objekata iz rječnika na temelju jedinstvenog identifikatora. Osim ovakvog sučelja, potrebno je i sučelje kojim druge skripte registriraju svoje objekte. Slijedi prikaz sučelja za registraciju agenata i za dohvaćanje agenta na temelju identifikatora.

```
public AIStateMachine GetAIStateMachine(int key)
{
```

```
    if(_stateMachines.ContainsKey(key))
    {
        return _stateMachines[key];
    }

    return null;
}
public void RegisterAIStateMachine(int key, AIStateMachine sMachine)
{
    if (!_stateMachines.ContainsKey(key))
    {
        _stateMachines.Add(key, sMachine);
    }
}
```

Gore naveden kod nije potrebno detaljnije objašnjavati zbog njegove jednostavnosti. Na kraju, važno je napomenuti da je u ovoj skripti implementirana metoda „OnDestroy“. Navedena metoda osigurava da se kod promjene scene (kod uništenja objekta na kojem se ova skripta nalazi) svim materijalima vrate originalne postavke (rječnik objekata „MaterialController“). Također, postavlja se vrijednost *singleton* instance „GameSceneManager“ na „null“ kako bi kod učitavanja sljedeće skripte rječnici bili prazni.

5. Zaključak

Korištenjem Unity alata za razvoj videoigara i VisualStudio alata za kreiranje skripti potrebnih za igru kreirana je 3D igra iz prvog lica. Igra je napravljena u duhu avanturističke igre s elementima horora i preživljavanja. Videoigra je kreirana uz veliku pomoć tečaja kupljenog na Udemy web stranici. Za izradu jedne ovakve igre potrošeno je više od 320 sati. U ukupan broj sati ubraja se i dizajn svih scena, dodatno implementirane samostalne ideje te vrijeme potrebno za praćenje video tečaja. Osim navedenog, u sate se ubraja i velika količina grešaka koje su se pojavljivale tijekom izrade igre zbog razlika u verzijama alata između trenutne Unity verzije i one korištene u tečaju. Također, pojavljivalo se je puno grešaka i zbog toga što je ova igra izrađena pomoću HDRP (engl. *High Definition Rendere Pipeline*) sustava. Ovakav sustav sadrži visokokvalitetne materijale koji nisu dostupni u standardnom sustavu i materijali se u svojim opcijama i mogućnošću upravljanja kroz skripte uvelike razlikuju. Što se tiče same igre, nije završena do krajnjih granica, naime ova igra je samo prototip.

Prije dokumentiranja praktičnog dijela (igre) u ovome radu, bilo je potrebno odraditi teoretski dio kako bi se čitatelja upoznalo s okolinom u kojoj je igra kreirana. Osim okoline, potrebno je donekle poznavati način skriptiranja (programiranja) u Unityju. Tek se nakon detaljnih teoretskih opisa može krenuti na opisivanje ovako kompleksnog sustava kao što je videoigra. No prije opisivanja videoigre, odrađeno je poglavlje o žanrovima i vrstama videoigara te opis avanturističkog žanra. Ovo poglavlje je moralo postojati prije opisa praktičnog dijela kako bi čitatelj znao što može očekivati od igre, koje su mehanike koje bi igra trebala sadržavati i sl. Kako je sustav koji tvori videoigru vrlo kompleksan te čak i neorganiziran, nije bilo moguće opisati svaku skriptu i svaki detalj igre. Postoji velik broj skripti koje nisu opisane u ovom radi, već su samo opisane one skripte koje na neki način definiraju samu igru i bez kojih igra uopće ne bi mogla funkcionirati. Drugim riječima, opisane su one skripte koje čine najveću razliku u odnosu na druge igre, tj. nije bilo potrebno opisivati široko korištena i dobro poznata rješenja.

Posao kreiranja videoigara (engl. *Game Development*) je vrlo težak posao zbog kompleksnosti sustava koji se svakom skriptom i svakim objektom sve više povećava. Vrlo je teško vratiti se nakon određenog vremena u projekt i odmah znati gdje se što nalazi. No, bez obzira na navedeno, kreiranje igara može biti vrlo interesantno i zanimljivo, što zbog samih problema koje treba riješiti skriptama, što zbog toga što za jedan problem postoji bezbroj mogućih rješenja. Također, ono što ovaj posao čini drugačijim od drugih vrsta programiranja

je taj što je potrebna visoka doza kreativnosti kod rješavanja problema i kreiranje scena. Osim toga, kod ovakvog vrsta programiranja, potrebno je razmišljati na drugačije načine nego kod programiranja drugih sustava. Skripte kreirane u svrhu videoigre ne mogu biti pisane na način da imaju početak i kraj kao linearni vremenski slije, već se nekoliko metoda vrti u krug kod svake nove slike (engl. *Frame*). U ovakav način vremenskog slijeda uvjerili smo se kod „Update“ i „FixedUpdate“ metoda. Osim navedenog, nije dovoljno samo pisanje programskog koda, već je važno stalno razmišljati o performansama napisanog koda.

Popis literature

Axon, S. (2016). Unity at 10: For better-or worse-game development has never been easier. Preuzeto 06. lipanj 2020., od Arstechnica website: https://arstechnica.co.uk/gaming/2016/09/unity-at-10-easy-game-development/

Brodkin, J. (2013). How Unity3D Became a Game-Development Beast. Preuzeto 06. lipanj 2020., od Dice website: http://insights.dice.com/2013/06/03/how-unity3d-become-a-game-development-beast/

Craighead, J., Burke, J., & Murphy, R. (2008). Using the Unity Game Engine to Develop SARGE : A Case Study. *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems Simulation Workshop*, (January). Preuzeto od https://www.researchgate.net/profile/Jeffrey_Craighead/publication/265284198_Using_the_Unity_Game_Engine_to_Develop_SARGE_A_Case_Study/links/55d33fdf08aec1b0429f32f4/Using-the-Unity-Game-Engine-to-Develop-SARGE-A-Case-Study.pdf

Croteam. (2014). The Talos Principle. Preuzeto 14. srpanj 2020., od http://www.devolverdigital.com/games/view/the-talos-principle

Haas, J. (2014). *A History of the Unity Game Engine - An Interactive Qualifying Project*. (March), 44. Preuzeto od https://web.wpi.edu/Pubs/E-project/Available/E-project-030614-143124/unrestricted/Haas_IQP_Final.pdf

Henley, L. (2011). What is an IDE ? Preuzeto 08. lipanj 2020., od https://www.redhat.com/en/topics/middleware/what-is-ide

Juliani, A., Berges, V.-P., Teng, E., Cohen, A., Harper, J., Elion, C., … Lange, D. (2018). *Unity: A General Platform for Intelligent Agents*. 1–28. Preuzeto od http://arxiv.org/abs/1809.02627

Karhulahti, V.-M. (2011). *Mechanic/aesthetic videogame genres*. 71. https://doi.org/10.1145/2181037.2181050

Shah, V. (2017, kolovoz 21). Reasons Why Unity3D Is So Much Popular In The Gaming Industry. Preuzeto 12. lipanj 2020., od https://medium.com/@vivekshah.P/reasons-why-unity3d-is-so-much-popular-in-the-gaming-industry-705898a2a04

Unity Technologies. (bez dat.). Scripting in Unity for experienced C# & C++ programmers | Unity. Preuzeto 25. lipanj 2020., od Unity Documentation website: https://unity.com/how-to/programming-unity

Unity Technologies. (2020a). Unity - Manual: Building a NavMesh. Preuzeto 27. lipanj 2020.,

od Unity Documentation website: https://docs.unity3d.com/Manual/nav-BuildingNavMesh.html

Unity Technologies. (2020b). Unity - Manual: Console Window. Preuzeto 16. lipanj 2020., od Unity Documentation website: https://docs.unity3d.com/Manual/Console.html

Unity Technologies. (2020c). Unity - Manual: Coroutines. Preuzeto 03. kolovoz 2020., od https://docs.unity3d.com/Manual/Coroutines.html

Unity Technologies. (2020d). Unity - Manual: Deactivating GameObjects. Preuzeto 13. lipanj 2020., od Unity Documentation website: https://docs.unity3d.com/Manual/DeactivatingGameObjects.html

Unity Technologies. (2020e). Unity - Manual: Layer-based collision detection. Preuzeto 23. lipanj 2020., od Unity Documentation website: https://docs.unity3d.com/Manual/LayerBasedCollision.html

Unity Technologies. (2020f). Unity - Manual: Nav Mesh Obstacle. Preuzeto 15. srpanj 2020., od Unity Documentation website: https://docs.unity3d.com/Manual/class-NavMeshObstacle.html

Unity Technologies. (2020g). Unity - Manual: NavMesh Agent. Preuzeto 18. srpanj 2020., od https://docs.unity3d.com/Manual/class-NavMeshAgent.html

Unity Technologies. (2020h). Unity - Manual: Project Settings. Preuzeto 18. lipanj 2020., od Unity Documentation website: https://docs.unity3d.com/Manual/comp-ManagerGroup.html

Unity Technologies. (2020i). Unity - Manual: Ragdoll Wizard. Preuzeto 21. srpanj 2020., od https://docs.unity3d.com/Manual/wizard-RagdollWizard.html

Unity Technologies. (2020j). Unity - Manual: Tags. Preuzeto 13. lipanj 2020., od Unity Documentation website: https://docs.unity3d.com/Manual/Tags.html

Unity Technologies. (2020k). Unity - Scripting API: MonoBehaviour. Preuzeto 01. srpanj 2020., od https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

Unity Technologies. (2020l). Unity - Scripting API: SerializeField. Preuzeto 01. srpanj 2020., od https://docs.unity3d.com/ScriptReference/SerializeField.html

Unity Technologies. (2020m). Unity - Scripting API: Time.deltaTime. Preuzeto 01. srpanj 2020., od https://docs.unity3d.com/ScriptReference/Time-deltaTime.html

Unity Technologies. (2020n). Working in Unity. Preuzeto 13. lipanj 2020., od Unity Technologies website: https://docs.unity3d.com/Manual/CreatingGameplay.html

Wikipedia. (bez dat.-a). Colossal Cave Adventure - Wikipedia. Preuzeto 14. srpanj 2020., od Wikipediai website: https://en.wikipedia.org/wiki/Colossal_Cave_Adventure

Wikipedia. (bez dat.-b). Myst - Wikipedia. Preuzeto 14. srpanj 2020., od Wikipediai website: https://en.wikipedia.org/wiki/Myst

Wikipedia. (2020). List of video game genres. Preuzeto 11. srpanj 2020., od Wikipediai website: https://en.wikipedia.org/wiki/List_of_video_game_genres#First-person_party-based_RPG%0D

Popis slika

Popis tablica

Prilozi

Opis riječi i kratica

Game engine	Pokretač računalnih igara, kako i riječ govori, to je „mašina za igre“, odnosno sve ono što je potrebno na programskoj razini (engl. Software) da se igra može izvoditi na računalu.
Software	Programski proizvod koji se može izvoditi na računalu.
IDE	IDE (engl. Integrated development environment), koja predstavlja skup potrebnih alata i pomoćnih komponenti u nekom alatu za razvoj nekog komada programskog proizvoda.
User friendly	Često kažemo da je neki alat *user friendly* onda kada ima grafičko sučelje koje korisniku pruža mogućnost rada na brz i jednostavan način te koji nakon kraćeg učenja korisniku omogućuje intuitivan rad
API	API (engl. Application programming interface) je sučelje za programiranje aplikacija, skup pravila, specifikacija i procedura koje razvojni programeri slijede kako bi se mogli pravilno služiti dostupnim resursima i uslugama unutar programa.
Assets (resursi)	*Asseti* su skup preuzetih ili samostalno kreiranih resursa koji se mogu koristiti za izradu igre u Unityju.
Prefab	Objekt u Unityju koji ne samo da se nalazi u sceni, nego je postavljen i u projektni

	direktorij (postao je resurs). Na ovaj način, objekt postaje referenca svim objektima u sceni koji su nastali od *prefaba*.
NPC (agenti)	NPC (engl. *Non-palyer character*) likovi kojima u igri upravljaju skripte, a ne drugi igrači (stvarni ljudi)
Singleton uzorak dizajna	Klasa implementirana na način da se može instancirati samo jednom. U svakom trenutku može postojati samo jedan objekt ove klase i on je uvijek isti.
Mapa (rječnik)	Varijabla koja može spremiti mnogo podataka u obliku ključ, vrijednost. Vrijednost se iz rječnika dohvaća pomoću referenciranja na jednoznačni ključ.
Instancirati / instanciranje	Stvaranje objekta (instance) neke domene (klase). Objekt (instanca) – pas, mačka, riba. Klasa (domena) - životinja