Izrada akcijske igre u prvom licu u Unity Game Engine

Vugrinec, Tomislav

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Tomislav Vugrinec

IZRADA AKCIJSKE IGRE U PRVOM LICU U UNITY GAME ENGINE

DIPLOMSKI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Tomislav Vugrinec

Matični broj: 44915/16–R

Studij: Baze podataka i baze znanja

IZRADA AKCIJSKE IGRE U PRVOM LICU U UNITY GAME ENGINE

DIPLOMSKI RAD

Mentor:

Prof. dr. sc. Danijel Radošević

Varaždin, rujan 2022.

Tomislav Vugrinec

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

Kroz ovaj rad ispitao sam mogućnosti i potencijalna ograničenja Unity i Blender alata. U uvodu rada kratko sam se dotaknuo karakteristika odabranog žanra igre. U metodama i tehnikama rada nabrojao sam i kratko opisao sve korištene alate i preuzete resurse, dok sam u razradi teme opisao i objasnio procese izrade video igre i 3D modela koji ta video igra koristi. Razrada teme podijeljena je na 2 velika dijela. Prvi dio naslova Blender bavi se kompletnom izradom modela, materijala i animacija potrebnih kako bi izrađeni model mogli koristiti unutar izrađene interaktivne akcijske igre. Drugi dio započinje poglavljem Unity i bavi se izradom akcijske igre prvog lica korištenjem osnovnih alata koji dolaze unutar Unity programa, ali i dodatnim proširenjima koje sam naknadno preuzeo sa Unity asset store webstranice u svrhu ubrzavanja procesa izrade video igre, ali i kako bih mogao preispitati i neke dodatne mogućnosti koje Unity alat nudi.

Ključne riječi: Unity, Blender, Unity plugin, računalna igra, programiranje

Sadržaj

1. Uvod

1992. godine tvrtka ID Software, u partnerstvu s izdavačem Apogee Software, izradila je Wolfenstein 3D igru koja je za svoje doba nudila pregršt inovacija na tada vrlo limitiranim hardverskim komponentama, od čega je najznačajnija inovacija bila upravo pogled igre iz prvog lica. Spomenuta igra zainteresirala je širok spektar novih igrača i time definirala temelje FirstPersonShooter (u nastavku kraće FPS) žanra koji će u narednim godinama dodatno ugladiti igre Doom i Quake. [1] Moj prvi susret s igrom u prvom licu bio je u ranim đačkim danima kada bih sa školskim prijateljima svakodnevno igrao Counter Strike 1.6 mod Half-life igre na raznim balkanskim serverima s ljudima iz čitavog Balkana. Nakon spomenutih igara brojne igre slijedile su ranije postavljenu shemu igranja u prvom licu te upotrebom vatrenog oružja kako bi otklonili novo stvorenu prijetnju. U FPS žanru okušale su se i hrvatske kompanije - poput tvrtke Croteam koja je 2001. godine izdala igru "Serious Sam: The First Encounter" koja je doživjela svjetski uspjeh. Sam žanr dvadesetak godina kasnije i dalje je jedan od najpopularnijih žanrova video igra te danas se dosta igara tog žanra odlučuje za održavanje raznih natjecanja u obliku e-sportova kako bi uključile i zainteresirale čim više igrača. Suvremeni predstavnici FPS žanra danas su igre poput Counter-Strike: Global Offensive, MetroExodus i Doom Eternal igara. Osim ranije spomenutih video igra koje se drže strogo isprobane formule FPS žanra, novije igre sve češće u FPS žanr ukomponiraju dodatne mehanike igre kako bi spojili više žanrova i stvorili nove zanimljive pod žanrove igra. Kroz ovaj rad izradio sam igru za jednog igrača koja sadrži osnovne mehanike FPS žanra definirane najranijim predstavnicima FPS žanra kako bih spoznao mogućnosti Unity i Blender alata koji se danas koriste u industriji izrade profesionalnih video igra. Tijekom izrade ove igre uvelike mi je pomoglo stečeno fakultetsko znanje, no putem sam još morao naučiti brojne stvari poput vještine izrade 3D modela i animacija te korištenja Unity alata i njegovih proširenja.

2. Metode i tehnike rada

Čitav kod igre implementiran je unutar Unity game engine alata razlomljen na manje Unity skripte koje sam kodirao unutar Microsoft Visual Studio 2019. integrirane razvojne okoline *(eng. Integrated Development Environment)*. Implementirane skripte pisane su upotrebom C# programskog jezika uz korištene biblioteke Unity alata koje skriptama dodaju razne funkcionalnosti i mogućnosti. Tijekom procesa izrade igre koristio sam brojna besplatna, ali i plaćena proširenja *(eng. plugin)* Unity alata poput Animancer, DoTween i NodeCanvas proširenja koja dodatno ubrzavaju i olakšavaju izradu igre tako da pojednostave, dodaju i/ili promjene način na koji Unity rješava određeni problem ili proces. Za ovu igru izradio sam i animirao model pištolja korištenjem Blender besplatnog alata za izradu 3D modela i vizualizacija, a kako bi stekao vještine potrebne za snalaženje unutar Blender alata, kupio sam i završio kurs naziva "Blender Character Creator for Video Games Design" na Udemy platformi za e-učenje. Ovaj projekt također sačinjavaju i kupljeni resursi za izradu video igre *(eng. video game assets)* koji će na kraju ovog rada biti detaljno popisani zajedno sa izvorima preko kojih sam iste preuzeo.

3. Razrada teme

Tema ovog rada je ispitati mogućnosti Blender i Unity alata za izradu video igra stoga je u nastavku struktura nadolazećih potpoglavlja sljedeća. Tema je razdijeljena na 2 velika dijela prvi od kojih opisuje Blender alat , brojne procese u izradi samog modela, izradu modela u Blender alatu te na posljetku pripremanje modela u format pogodan za korištenje unutar Unity game engine alata. Drugi veliki dio opisuje Unity alat i njegove brojne sustave koji korisniku omogućuju obavljanje brojnih kompleksnih radnji poput animiranja 3D modela, kreiranje glavnog meni-a igre ili primjerice simulaciju fizike nad objektima unutar scene u samom programu. Osim osnovnog Unity alata koristio sam besplatna i plaćena proširenja Unity alata kojih ću se dotaknuti u poglavlju "Korištena Unity proširenja".

3.1. Procesi izrade video igre

Izrada igre dugotrajan je proces koji uključuje mnoge discipline i znanosti, stoga igru najčešće realiziraju manji, srednji ili veliki timovi ljudi raznih kompetencija iako postoji par iznimka gdje su vrlo talentirani individualci sami kreirali kvalitetnu i komercijalno uspješnu igru. Proces izrade video igre podijeljen je na tri dijela: procesi prije same izrade igre, proces izrade igre te procese poslije izrade igre. Kroz procese prije izrade same igre prvi naznaci igre formiraju se kao koncepti ideja koji zajedno sa okolišem *(eng. environment)* unutar igre, samom pričom igre i raznim mehanikama igre *(eng. gameplay mechanics)* zajedno tvore skup definiranih ideja unutar dokumenta dizajna igre *(eng. game design document)* koje igru sačinjavaju. Pri smišljanju ideja važno je prije prepoznati žanr, potencijalnu zainteresiranu publiku, perspektivu kamere odnosno sam pogled na igru te platforme za koje će se igra proizvesti. (prema [2]) Dokument dizajna igre označava početne specifikacije za izradu same igre, a kraj njegove izrade, označava kraj procesa prije same izrade igre. Proces izrade igre traje najdulje od triju spomenutih procesa. Kako bi se igra realizirala od ideje do vizualno interaktivnog iskustva, potreban je određen skup vještina i ljudi s jasno definiranim zadaćama unutar tima. Iako se procesi izrade video igre pa tako i podjela odgovornosti razlikuju od tvrtke do tvrtke, većina poslova može se kategorizirati u par općenitih kategorija. Tako pri izradi igre postoji umjetnički odjel *(eng. art department)* čija je zadaća izraditi resurse koji će se koristiti unutar igre, ali i smisleno međusobno povezati izrađene resurse igre u cjelinu kako bi se očuvala kohezija između izrađenih komponenti. Poslovi umjetnika za izradu 3D modela, tekstura ili posao animatora svi spadaju u spomenuti odjel. Odjel dizajna brine se osmišljanju korisničkih iskustva te definiranju samog osjećaja, dubine i napredovanja *(eng. game progression)* igrača kroz igru. Tipični poslovi odjela dizajna igre obuhvaćaju poslove dizajnera mehanika igre *(eng. gameplay designer)*, dizajnera nivoa *(eng. level designer)* te dizajnera korisničkih iskustva *(eng. User experience designer)*. Poslovi odjela za izradu zvuka koji igri dodaju audio komponentu obuhvaćaju dizajnere zvuka *(eng. sound designer)* koji izrađuju zvučne efekte poput zvuka pucnja unutar igre, skladatelje glazbe *(eng. music composer)* i audio programere koji implementiraju izrađenu glazbu i zvučne efekte unutar igre. Odjel programera zadužen je za tehničku implementaciju svih dogovorenih specifikacija i ideja dogovorenih dokumentom dizajna igre te realizaciju igre.

Budući da je izrada igre veliki tehnički pothvat postoji mnogo specijalizacija unutar ovog odjela poput programera mehanika igre *(eng. gameplay programmer)* koji putem koda u igru implementira mehanike igre dogovorene dokumentom dizajna igre, mrežni programer *(eng. network programmer)* koji implementira protokole kako bi igrači mogli međusobno djelovati i komunicirati te programer umjetne inteligencije *(eng. artificial intelligence programmer)* zadužen da agenti kojima upravlja računalo djeluju pametno i smisleno komuniciraju s igračem. Odjel za osiguranje kvalitete *eng. quality assurance department)* testira i prijavljuje greške koje su se potkrale prilikom tehničke implementacije igre, dok odjel produkcije *(eng. production department)* je zadužen za marketing i promidžbu igre, te vodi brigu o financijskom aspektu izrade igre kako bi se igra izradila u dogovorenom vremenskom periodu unutar okvira predviđenog budžeta igre. (prema [3]) Osim spomenutih poslova tvrtke nerijetko osmišljavaju razne dodatne poslove poput "Dev-ops" operativnih poslova koji obuhvaćaju programere čiji je posao ubrzati postojeće procese izrade virtualnog proizvoda optimizacijom postojećih procesa ili osmišljavanjem novih bržih procesa unutar tvrtke. Izrada igre ogroman je pothvat stoga i ne čudi podatak da mnogo igra probije svoje planirane budžete te se nikada do kraja ne realizira. Procesima poslije izrade igre izrađena igra dodatno se optimizira kako bi se dodatno unaprijedilo korisničko iskustvo te kako bi igra bila dostupna većem spektru igrača podržana na što većem broju računala različitih hardverskih komponenti. Kvalitetno odraditi sve spomenute poslove gotovo je nemogući zadatak stoga u ovom projektu ,kako bi pojednostavio pojedine procese izrade igre, koristim gotove kupljene 3D modele te besplatna ali i kupljena proširenja *(eng. plugin)* Unity alata koje ću kasnije detaljnije opisati.

3.2. Prepoznati elementi izrađene video igre

Kroz izradu ovog rada prepoznao sam elemente većih cjelina bez kojih izrađeni projekt ne bi mogli nazvati igrom. Kroz ovaj rad objasnit ću mogućnosti koje Unity nudi, a paralelno tome i elemente koji sačinjavaju izrađenu igru. Preostale neobjašnjene elemente objasnit ću u poglavlju preostalih implementiranih funkcionalnosti. Kako bih ostvario ovaj projekt koristio sam brojne plaćene preuzete resurse kojih ću se u nastavku ukratko dotaknuti, a poveznice na relevantne materijale dodati kao dodatak na kraju ovog rada.

3.2.1. Izbornik pri ulasku u igru

Pri pokretanju igre igrača najčešće dočeka početni meni koji nudi nekoliko opcija poput pokretanja nivo-a igre, promjene opcija senzitivnosti kontrola i tipku za izlaz iz igre. U ovom radu meni igre realiziran je kao zasebna scena koja se učita odmah pri pokretanju igre, a pritiskom na tipku play unutar izbornika pokrećemo drugu scenu koja sadrži izrađeni nivo, igrača i neprijatelje.

3.2.2. Nivo igre

Nivo igre *(eng. game level)* predstavlja tlo po kojem se kreću igrač i neprijatelj. Da vizualno dočaram tlo kao zemljinu koru, koristio sam slike i teksture preuzete sa ambientCG web stranice. Na izrađeno tlo dodatno sam još postavio drveće i travu kako bi nivo postao što više realističan. Izradu nivoa igre odradio sam korištenjem gaia 2021 plaćenog proširenja Unity alata jer sam pomoću istog mogao automatizirati proces bojanja terena i postavljanja drveća i trave.

3.2.3. 3D modeli i animacije

Za izradu igre koristio model pištolja koji sam izradio unutar Blender alata. Osim izrađenog modela pištolja u igri sam koristio i plaćene preuzete modele poput Polygon Zombie paketa modela koje sam koristio za prikaz neprijatelja unutar igre. Za model drveća i vlasi trave koristio sam modele koji su uključeni u gaia 2021 proširenje. Kako bih modelima neprijatelj udahnuo dašak života i učinio ih dinamičnim unutar izrađene scene, koristio sam besplatno preuzete animacije sa Mixamo web stranice, dok sam animacije za model pištolja sam izradio unutar Blender alata.

3.2.4. Kamera i upravljač za kretanje igrača

Igra je interaktivno iskustvo stoga kako bih čitao igračeve pritiske miša i tipkovnice koristio sam novi Unity input sustav. Kako bi realizirao osnovne mehanike igre koje igrač može koristiti unutar igre, kreirao sam više skripti funkcionalnosti. Za kretanje igrača kroz nivo izradio sam upravljačku skriptu koja pročitanom korisničkom unosu pridodaje odgovarajuće akcije. Na posljetku kako igrač ne bi izgubio model kojim upravlja, izradio sam i skriptu kamere koja slijedi igrača kroz nivo.

3.2.5. Funkcionalnosti i kompleksno ponašanje neprijatelja

Kompleksno i inteligentno ponašanje neprijatelja izradio sam putem manjih skripti akcijskih i uspoređujućih zadataka koje neprijatelj u svakom trenutku obavlja. Neprijatelji u sceni imaju mogućnost primijetiti metu tj. igrača ukoliko se on nalazi u njihovom vidnom polju. Ako je neprijatelj ugledao metu, on počinje trčati prema meti, a ako meta izađe van njegova vidnog polja, on ju može pratiti osjetom mirisa sve dok i taj trag ne iščezne. Ukoliko neprijatelj nije locirao svoju metu nasumično luta terenom ili stoji i provjerava okolinu. Tok odlučivanja radnji neprijatelja, odnosno inteligentnog ponašanja, izradio sam korištenjem NodeCanvas plaćenog proširenja za Unity alat dok sam animacije neprijatelja izvodio korištenjem Animancer plaćenog proširenja Unity alata.

3.3. Blender

Blender je besplatan program otvorenog koda za stvaranje 3D modela *(eng. 3D creation suite)*. Kompletno podržava tok izrade *(eng. pipeline)* 3D modela odnosno procese kroz koje model kroz svoju izradu prolazi. Blender podržava modeliranje, namještanje armature *(eng. rigging)*, animacije, simulacije, prikazivanja *(eng. rendering)*, kompozicije i praćenja pokreta *(eng. compositing and motion tracking)*, uređivanja videa pa čak i kreiranja igre. Blender kao profesionalni alat svakodnevno nadograđuje zajednica programera, programera volontera i korisnika samog alata čiji je zajednički cilj omogućiti svakome korisniku besplatan i profesionalan alat za izradu 3D modela.[4] Zbog toga Blender alat koristi GNU GPL licencu *(eng. GNU General Public Licence)* koja diktira da čitav kod Blender alata mora biti dostupan za čitanje, proučavanje i modifikaciju svakome koga zanima struktura i kod projekta. Prednost ovog pristupa je da korisnik također vrlo lako može sudjelovati u ispravcima postojećih grešaka unutar programa, unaprijediti postojeću dokumentaciju programa ili dodati neku novu naprednu mogućnost koja je do tada nedostajala unutar Blender programa. Potrebno je još napomenuti kako sam prilikom izrade modela koristio Blender verzije 3.2.1 budući da je alat s verzijom 2.8 iznimno napredovao te poprimio intuitivniji dizajn i izgled od prethodnih verzija.

3.3.1. Kompletan proces izrade 3D modela za video igru

Ova sekcija opisuje proces izrade skulpturiranog modela zajedno sa svim procesima potrebnim da se model kompletno realizira kao resurs za video igru. Pri izradi svakog modela najprije u scenu moramo dodati jedan ili više jednostavnih geometrijskih likova te opcionalno pozadinsku sliku kao referencu prema kojoj izrađujemo model. Cijelu referentnu sliku prekrijemo primitivnim likovima kvadra razdvajajući ih pri zglobovima nacrtanog lika. Ovu fazu nazivamo fazom skiciranja *(eng. blocking out)*. Nakon toga dodanim kockama moramo dodati više detalja kako bi ih pripremili za proces skulptiranja. Kako bi dobili više detalja (više vrhova i ploha) na kvadrima primjenjujemo Subdivision modifikator. Nakon toga model skulptiramo pomoću specijalnih alata za skulptiranje unutar blendera. Kod skulptiranja sam model više je sličan glini *(eng. clay)* koju alatima za skulptiranje oblikujemo u konačan model. Kada smo zadovoljni izgledom i detaljima modela, najčešće model sadrži prevelik broj poligona *(eng. polygon)* odnosno ploha, što bi loše utjecalo na performanse igre unutar koje koristimo izrađeni model. Stoga nakon skulptiranja slijedi proces retopologije, odnosno pojednostavljivanja oblika na oblik koji je oblikom vrlo sličan originalnom, ali sadrži puno manje poligona. Postoji puno načina izrade topologije od kojih su neki i potpuno automatizirani, no potrebno je napomenuti da najbolje rezultate još uvijek daje ručna retopologija *(eng. manual retopology)*. Kako bi na skulptirani model mogli kasnije dodati obojene plošne teksture kroz proces razmotavanja modela u UV prostor *(eng. UV unwrapping)* na modelu označujemo porubove preko kojih će Blender kasnije virtualno razrezati model kako bi 3D model rastavio na skup kompleksnih oblika u 2D UV prostoru. Pravilno razrezan 3D model sprječava razne artefakte prilikom bojanja kao primjerice rastezanje materijala na plohama ili vizualni prekid tekstura odnosno prelaz s kraja prve na početak druge teksture. Na kraju teksture detaljno obojenog modela moramo prenijeti na manje kompleksan model koji smo izradili procesom retopologije. Time detalje detaljnog

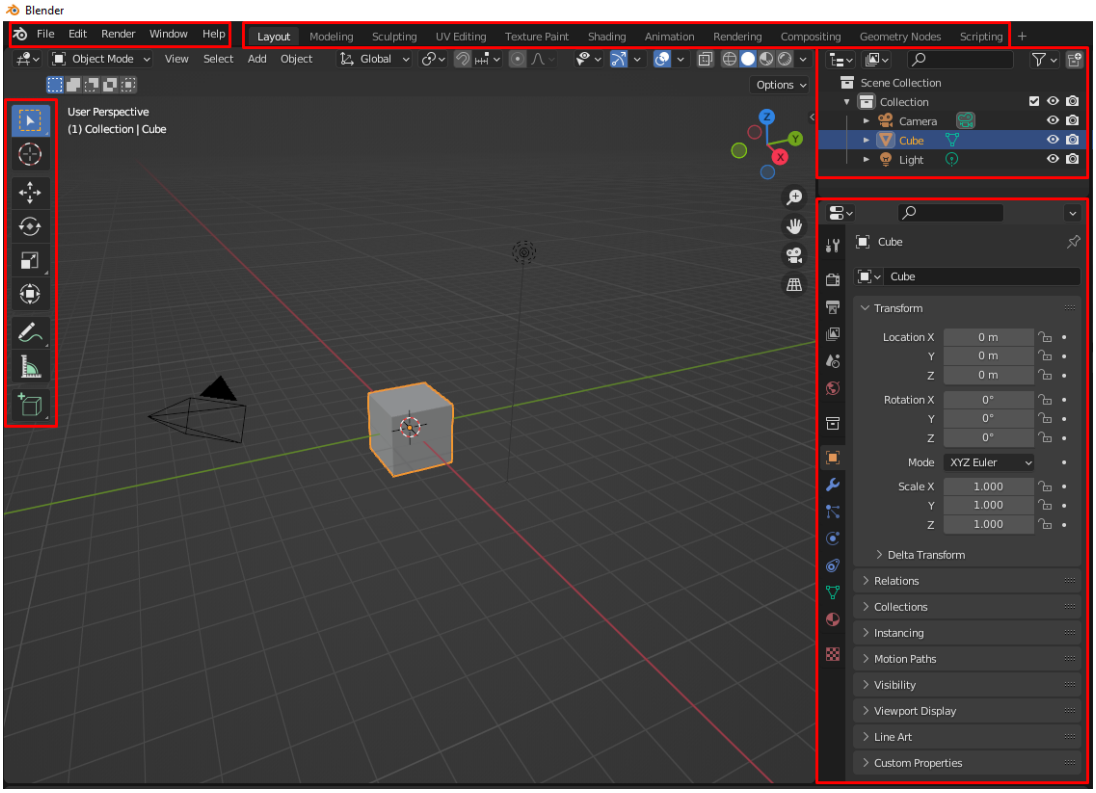
modela također prenosimo na jednostavniji model koji sadrži puno manje poligona te smo ovim postupkom "pečenja tekstura" *(eng. texture bake)* dobili jednako detaljan model koji sadrži znatno manje poligona od originalnog detaljnog modela. Posljednji proces pri izradi organskog modela za igru jest izrada animacija i pokreta koje će model izvoditi unutar igre. (prema [5])

Slika 1: Primjer dobre retopologije kompleksnog modela (izvor [6])

Slika 1. prikazuje tri modela dobivenih u različitim fazama izrade umjetničke kreacije. Model s lijeve strane je jednostavan model dobiven procesom retopologije detaljnog modela. Detaljan model u sredini između 2 modela dobiven je procesom skulptiranja i vrlo je detaljan što se može primijetiti i po samim ornamentima na oklopu detaljnog modela. Detaljan model posjeduje previše poligona što bi utjecalo na performanse video igre stoga umjetnik ovdje također koristi proces "pečenja tekstura i detalja" s detaljnog modela na jednostavan model i konačan rezultat vidljiv je na desno obojenom modelu. Model je očuvao sve detalje kompleksnog modela, a koristi plohe izrađenog pojednostavljenog modela.

3.3.2. Korisničko sučelje Blender alata

Korisničko sučelje blender alata nudi mnoštvo mogućnosti stoga je samo korisničko sučelje *(eng. user interface)* iznimno dobro organizirano. U nastavku ću ugrubo opisati organizaciju opcija i alata unutar samog Blender programa, a detaljnijem opisu alata pristupit ću kroz izradu modela pištolja koji će se koristiti unutar video igre.



Slika 2: Početni pogled Blender alata

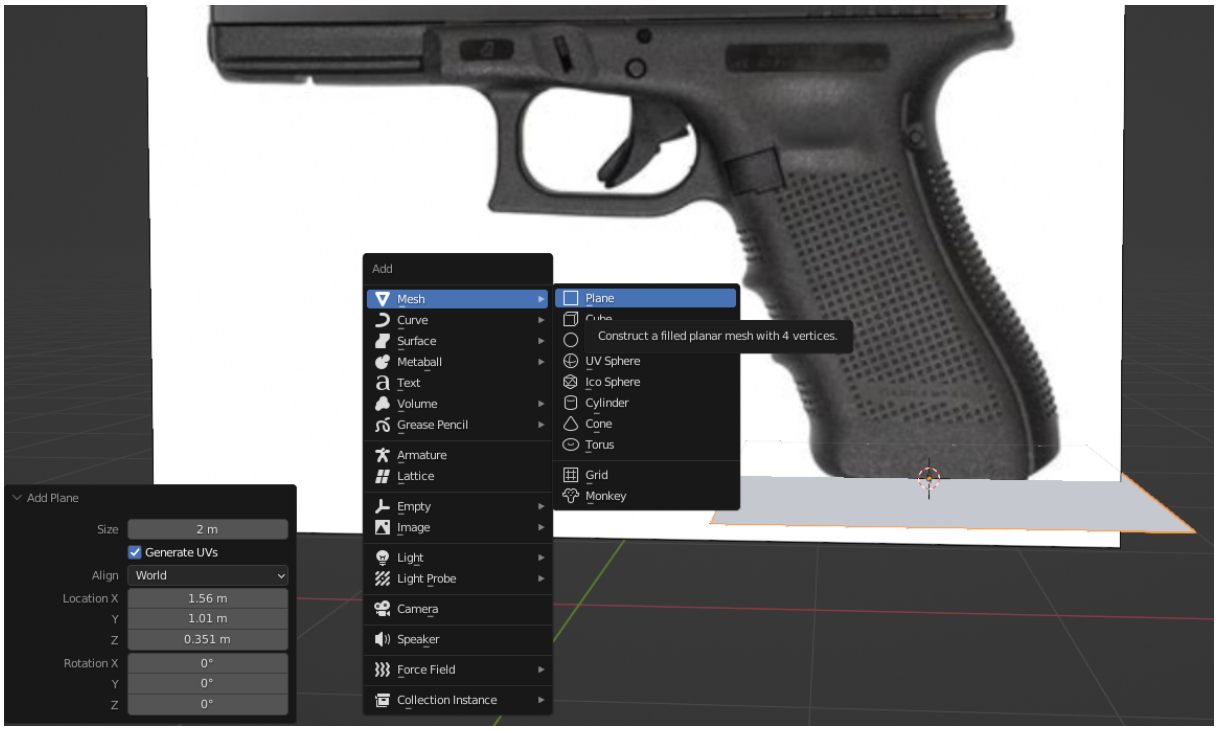
Na prikazanoj slici vidljivo je odvojeno pet cjelina uokvirenih unutar crvenih pravokutnika. Počevši od gore lijevo najprije imamo Blender meni sa "File Edit Render Window Help" opcijama. Pritiskom na "File" opciju otvaramo nove opcije poput spremanja izrađenog projekta unutar alata, uvoz *(eng. import)* postojećih modela, ne nužno u blender formatu već i ostalim formatima za dopremu 3D modela te izvoz opciju *(eng. export)* unutar koje biramo kako želimo zapakirati model za daljnju uporabu s drugim programima. Ostale opcije mogu se ostvariti i putem prečaca *(eng. shortcut)* i drugih dijelova korisničkog sučelja stoga ih ovdje neću pobliže objašnjavati. Sljedeća grupa alata desno od blender menija nazivamo radnim prostorima *(eng. workspace)*. Svaki radni prostor sastoji se od područja koji sadržavaju uređivače *(eng. editor)* i namijenjeni su obavljanju specifičnih zadataka poput modeliranja 3D objekta, animiranja animacija i slično. [7] Kroz izradu modela pištolja koristit će se "Layout" , "Animation" i "Shading" koje ću kasnije detaljnije objasniti. Lijevo od Blender menija svaki radni prostor sadrži alatnu traku ispunjenu alatima *(eng. tool)* koji se najčešće koriste za specifični proces za koji su namijenjeni.

S desne strane nasuprot alatne trake nalaze se dvije zasebne cjeline. Njih nazivamo panelima, gore desno je "Outliner" područje koje hijerarhijski prikazuje sve objekte unutar scene.

Osim u središnjem vizualnom dijelu , objekte također možemo i odabrati i preko "Outliner" područja klikom na objekt u hijerarhiji koji želimo odabrati. Naposljetku ispod "Outliner" područja nalazi se "Properties" područje koje je podosta kompleksnije od svih ostalih područja budući da unutar sebe sadrži dodatne kartice od kojih svaka prikazuje određene informacije aktivnog objekta. Glavana namjena područja svojstva *(eng. Properties area)* je prikazivanje i modificiranje postojećih svojstava odabranog (aktivnog) objekta poput pozicije, rotacije i dimenzija samog objekta ako korisnik selektira "Object properties" karticu ili prikaz svih modifikatora nad aktivnim objektom ako korisnik odabere "Modifier properties" karticu i slično.

3.3.3. Izrada oblika modela pištolja

3D izgled modela pištolja modeliran je unutar Layout načina rada unutar Blender alata. Kao što sam ranije opisao, svaki radni prostor sastoji se od više zasebnih područja koji zajedno objedinjuju operacije potrebne za izvođenje specifičnog procesa izrade 3D modela za koji je taj radni prostor namijenjen. Layout radni prostor koristio sam kako bih uvijek imao uvid u vizualni izgled modela kojeg sam izrađivao. Sama izrada ovog modela smisleno slijedi isti ranije opisan postupak izrade organskog modela, no budući da je ovdje riječ o izradi modela tvrdih površina *(eng. hard surface modeling)*, pojedini procesi će biti pojednostavljeni ili kompletno preskočeni ponajprije zbog jednostavnije izrade modela pištolja. Prije nego što sam započeo s izradom samog modela , pronašao sam referencu pištolja koji sam odlučio izmodelirati, u ovom slučaju odlučio sam izraditi kultno vatreno oružje imena glock17 koje je do sada doživjelo već pet generacija, preinaka i poboljšanja. Nakon što sam pronašao zadovoljavajuću sliku, unio sam je u blender te pozicionirao za proizvoljni negativni pomak po y osi kako bi slika bila u pozadini dok sam izrađivao 3d model.



Slika 3: Dodavanje objekata unutar scene

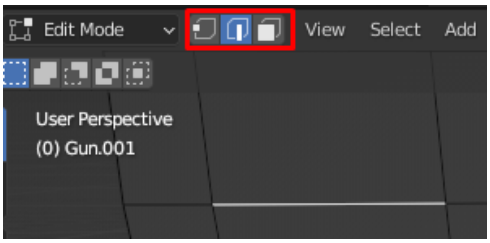
Prilikom izrade novog 3D modela najprije unutar scene moramo dodati barem jedan jednostavan objekt poput kocke, valjka ili kugle koji zatim raznim operacijama unutar radnog okvira prilagodimo da izgledom što više nalikuje na sliku koju smo odabrali. Miš mora biti pozicioniran unutar središnjeg vizualnog prikaza scene te pritiskom SHIFT + A tipki na tipkovnici otvaramo izbornik gdje biramo objekt koji želimo dodati u scenu na kojoj trenutno radimo. U ovom slučaju odabrao sam opciju "Mesh > Plane" kako bih dodao ravnu plohu unutar scene. Kao što je vidljivo sa slike 3. sama ploha pogrešno je rotirana te prodire kroz pozadinsku sliku pištolja što nije neobično budući da nad novo dodanom plohom nismo radili nikakve izmjene. Najčešće je slučaj s novo dodanim objektima da njihove dimenzije, rotacija i veličina ne odgovaraju ostalim dijelovima modela stoga nakon dodavanja u scenu dodanim elementima mijenjamo njihova osnovna svojstva. Kako bi bili brži od otvaranja izbornika pa odabira operacije, Blender definira prečace *(eng. shortcut)* u obliku kombinacija tipki za najčešće korištene operacije unutar samog alata, dok poneki prečaci ovise o kontekstu odabranog aktivnog radnog okvira. Pa tako ukoliko želimo provesti operaciju translacije objekta možemo pritisnuti tipkovnu kraticu "G" , za operaciju rotacije slovo "R", a operaciju skaliranja *(eng. scale)* pritisnemo slovo "S". Nadalje prilikom obavljanja operacija ako pritisnemo X,Y ili Z limitiramo operaciju samo na određenu odabranu os pa tako ako pritisnemo G + X, aktivni objekt možemo translatirati samo po X-osi. Ukoliko pritisnemo S te zatim SHIFT + X obavljamo operaciju skaliranja nad svim osima izuzev X osi. Sljedeća slika prikazuje najčešće korištene prečace Blender alata.

Edit Mode	
1	Vertex mode
2	Edge mode
3	Face mode
P	Create separate object out of selection
M	Merge selection
GG	Vertex and edge slide
E	Extrude geometry
F	Fill face
Ctrl + R	Loop cut
Scroll wheel after Ctrl + R; Move mouse to choose orientation	Add loop cut divisions
Left click	Lock in loop cut
I	Add inset faces to selection
Ctrl + B	Add bevel to selection

Object Mode	
Tab	Toggle between Object mode and Edit mode
Shift + A	Add new object
Shift + Tab	Toggle snapping
Ctrl + A	Apply transformations
Alt + G	Clear position
Alt + R	Clear rotation
Alt + R	Clear scale
Ctrl + J	Join selected objects
Ctrl + 1, 2, 3, 4, etc	Add subdivision set
Shift + C	Reset 3D cursor to center

Slika 4: Najčešće korišteni prečaci blender alata

Osim manipulacije plohe kao zasebne cjeline pritiskom na tab tipku možemo promijeniti način modifikacije aktivnog objekta s objektnog načina u uređivački način. Uređivački način *(eng. edit mode)* omogućuje nam da postojeći aktivni objekt uređujemo s detaljnijom granularnosti.Pa tako kada smo unutar uređivačkog načina možemo još dodatno odabrati želimo li aktivni objekt rastaviti na skupinu njegovih vrhova, bridova ili želimo objekt gledati kao skupinu ploha koje ga omeđuju kao što je prikazano horizontalnim izbornikom na slici 5. Budući da smo odabrali granularnost u pogledu bridova jednim klikom miša označujemo kliknuti brid. Ukoliko želimo označiti više bridova držimo tipku SHIFT, a ako smo pogrešno selektirali jedan od bridova držimo tipku CTRL i mišom odaberemo pogrešno označen brid.

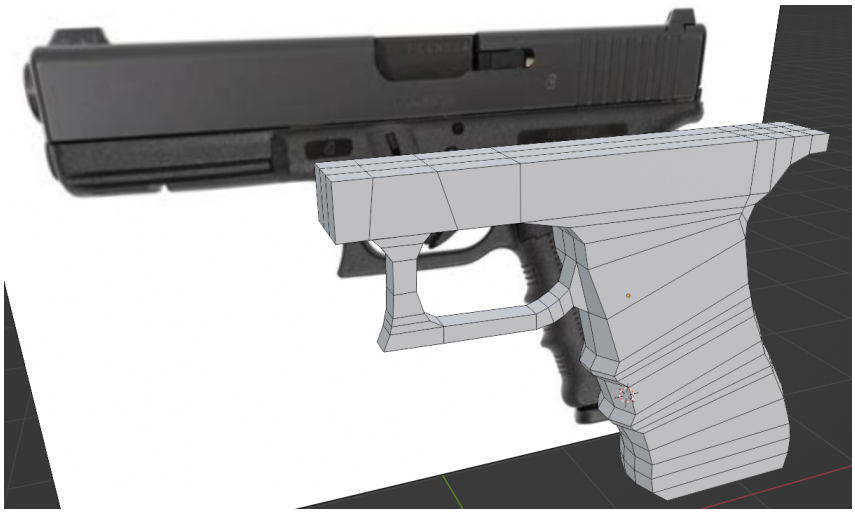


Slika 5: Granularnost uređivačkog načina rada

Glavnina modeliranja 3D modela odrađuje se unutar uređivačkog načina rada budući da omogućava detaljniju granularnost od objektnog načina rada, dok objektni način više služi, u početku za grubi, a kasnije i precizni raspored izmodeliranih elemenata na sceni. Plohu koju smo dodali u scenu najprije u objektnom načinu rada rotiramo oko x osi za 90 stupnjeva kako bi ploha bila paralelna sa slikom. Translatiramo i skaliramo plohu da ugrubo prekriva dršku pištolja.Sam vizualni prikaz promijenimo u pogled žičanog okvira *(eng. wireframe view)* kako bi mogli pratiti i referentnu sliku iza dodane plohe. Prebacimo se u uređivački način rada. Dodamo nove bridove unutar same plohe korištenjem "Loop cut" naredbe. Promijenimo granularnost na vrhove modela te ručno pomičemo točke kako bi ugrubo slijedile obrub postavljene slike.

Slika 6: Popločavanje drške pištolja u uređivačkom načinu rada

Postojeći oblik izmijenjene plohe možemo proširiti korištenjem "Extrude" naredbe koja projektira i kreira novi vrh, brid ili plohu , ovisno o tome što smo prije toga označili. Kako bi proširili plohu da prekriva čitavu površinu drške pištolja izabrat ćemo posljednji brid i projicirati ga više puta, ukoliko nismo zadovoljni detaljem izrađenog oblika, možemo dodati nove bridove unutar korištenjem već spomenute "Loop cut" naredbe. Na kraju kada smo zadovoljni oblikom koji pokriva izrađena ploha , označit ćemo cijelu plohu i sve njezine dijelove i izvršiti "Extrude" naredbu kako bi izrađenoj plohi dali širinu, odnosno kako bi ploha poprimila 3D oblik. Dobiveni oblik nazivamo "Low poly" verzijom finalnog modela. Izreka "Low poly" u žargonu kreatora 3D modela znači da je ovo tek prva, gruba verzija modela čija je svrha obuhvatiti glavni oblik modela koji izrađujemo.

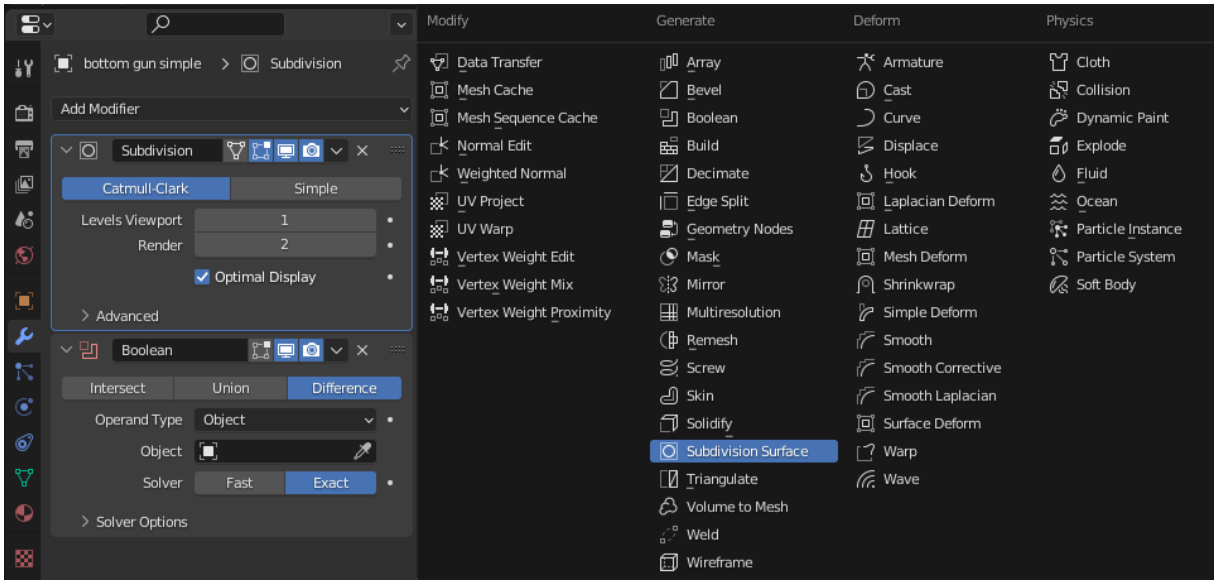


Slika 7: Gruba verzija drške pištolja

Ostale dijelove pištolja jednako tako možemo izraditi ranije spomenutim tehnikama: dodavanjem novih oblika u objektnom načinu rada, te modifikacijom tih novih oblika u uređivačkom načinu rada korištenjem operacija projekcije, dodavanja bridova tj. rezanjem elementa na manje dijelove te skaliranjem, translacijom i rotacijom. Pritom spomenute operacije možemo izvršavati na potrebnoj nam granularnosti: nad vrhovima, bridovima ili plohama objekta.

Postavlja se pitanje zašto prvo raditi grubu verziju modela umjesto da odmah pristupimo izradi detaljnog modela? Prije svega, ovakav pristup, iako naizgled u izradi uzme više vremena kad umjesto jednog modela radimo dva modela - jedan niske rezolucije *(eng. Low poly)* i drugi detaljan model koji je puno kompleksnije naravi koji vjerodostojnije prikazuje izrađeni 3D model, ovakav pristup dolazi s mnogo prednosti. Puno je lakše, a i brže izmijeniti i modificirati oblik objektu koji je definiran s pedesetak ploha, nego tu istu preinaku raditi nad detaljnim modelom koji ima preko pet tisuća ploha. Druga prednost vezana je uz kompleksnost izrađene igre. Naime kompleksnost igra konstantno raste, dok unapređenja hardverskih komponenti sve sporije napreduju, stoga video igre konstantno zahtjevaju sve više i više računalnih resursa. Kako bi kreatori igra doskočili tome problemu, u industriji igara koriste se brojni trikovi optimizacija nad igrom, ali i nad resursima korištenim unutar igre. Naime procesorska jedinica *(eng. Central processing unit* u jednom trenutku može manipulirati određenim brojem poligona *(eng. polygon)* odnosno ploha, a smanjenje broja ploha direktno utječe na kvalitetu i detaljnost samog modela. Tako za 3D modele postoji proces pečenja svjetlosti *(eng. bake light)* nad posebnom teksturom mape normala *(eng. normal map)*. Kako bi se zadržao broj poligona jednostavnog "Low poly" modela i istovremeno očuvali detalji koje ima detaljan model, procesom izrade mape normala na mapu zapečemo smjer odbijanja svjetlosti kod detaljnog modela. Tu mapu normala (2D sliku) koristimo kao informaciju za odbijanje svjetlosti na jednostavnom modelu. Sada jednostavan i grubi model izgledaju više manje identično detaljnom modelu budući da prividnim lomom svjetlosti, jednostavan model poprima detalje detaljnog modela, a zadržava minimalan broj poligona i time minimalno utječe na performanse igre. Nadalje operacije nad jednostavnim i grubim modelom izvršavaju se puno brže od istih operacija nad identičnim kompleksnijim modelom.

Budući da smo sada izradili jednostavan model pištolja, duplicirat ćemo taj model, original spremiti i sakriti van vidnog polja te kako bi iz jednostavnog modela izradili kompleksniji model. Za povećanje detalja i rezolucije modela koristit ćemo modifikator "Subdivision surface". Sve modifikatore aktivnog objekta možete pronaći na slici 8. S lijeve strane je pogled i opcije već dodanih modifikatora na aktivni objekt, dok je s desne strane izbornik svih dostupnih modifikatora koji se otvara pritiskom na opciju "Add Modifier" u postavkama modifikatora za trenutno aktivni objekt.

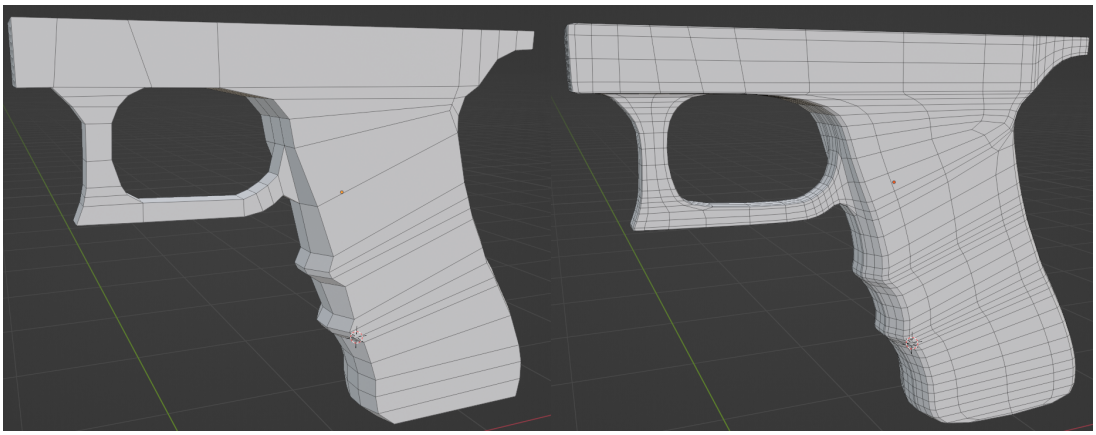


Slika 8: Postavke i lista svih modifikatora blender alata

Kako bi "Subdivision surface" modifikator ispravno radio model mora zadovoljiti ograničenje da sve plohe koje ga omeđuju moraju biti međusobno povezane te svaka ploha mora biti omeđena s četiri stranice. Ukoliko model ne zadovoljava ograničenje omeđenosti plohama s četiri stranice, eventualno povećanje detalja morali bi zasebno dodavati razlomljivanjem ploha pomoću nož alata *(eng. knife tool)*. Pomoću nož alata dodajemo detalje tako da odabranu plohu raspolovimo na dvije manje stranice. Srećom pri izradi modela pretežito smo projicirali brid pa smo time i zadovoljili ovo ograničenje. Unutar postavki modifikatora vidljivih na slici 8. zatim biramo algoritam podjele ploha te koliko puta želimo povećati detalje modela. Iako su već sada promjene vidljive na samom modelu, "Subdivision surface" modifikator još uvijek nije primijenjen na sam model. Izrađeni model i dalje je jednostavan i sve njegove plohe su očuvane, no ako proširimo izbornik i odaberemo opciju primjeni *(eng. apply)* tada bi trajno primjenili odabrani modifikator nad aktivnim dijelom modela i više ne bismo imali mogučnost povratka na jednostavniji model. Zato smo prije izvođenja bilo kakvih modifikatora originalni jednostavan model duplicirali i sakrili kako ukoliko su nad modelom potrebne neke veće promjene, ili smo se odlučili na viši ili manji stupan detalja, ponovo možemo početi od iste polazne točke tj. jednostavnog originalnog modela.

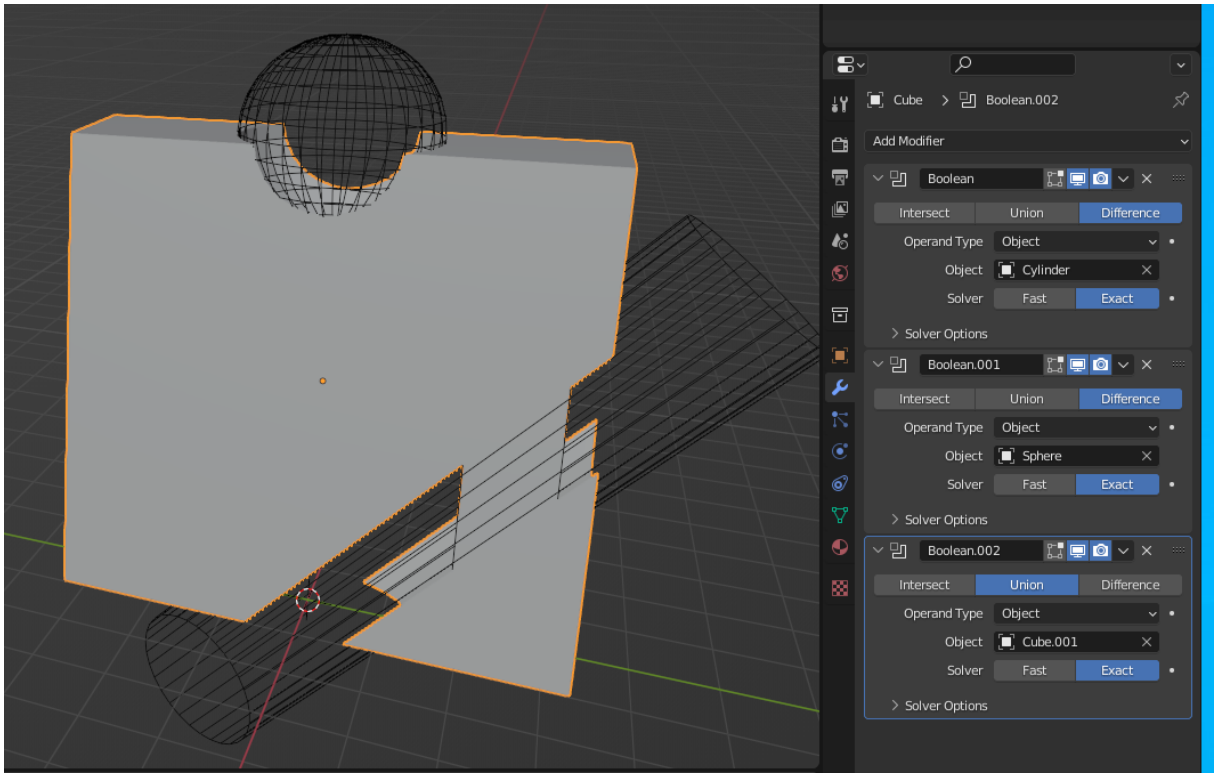
Trajnu primjenu modifikatora nad aktivnim elementom nazivamo destruktivnim načinom izrade modela, odnosno svaka primijenjena operacija nad modelom trajno i bespovratno mjenja model koji izrađujemo. Kada bolje razmislimo, do sada sve operacije koje smo radili nad modelom bile su destruktivne naravi koje smo mogli ispraviti samo netom nakon što smo ih

izvršili koristeći naredbu poništavanja *(eng. undo command)*. Naravno prije svake promjene možemo spremiti model nad kojim radimo destruktivnu operaciju, no ubrzo bi uvidjeli da hijerarhija scene sadrži mnogo nepotrebnih i nedovršenih objekata te je navigacija kroz čitav projekt znatno otežana i usporena.



Slika 9: Model prije i nakon korištenja "Subdivision surface" modifikatora

Osim destruktivnog postoji i ne destruktivan način izrade modela u blenderu koji naravno ima svoje prednosti, ali i nedostatke. Glavna premisa nedestruktivnog načina izrade modela jest korištenje Boolean modifikatora nad aktivnim elementom kojem modeliramo oblik.



Slika 10: Nedestruktivan način izrade modela korištenjem niza boolean modifikatora

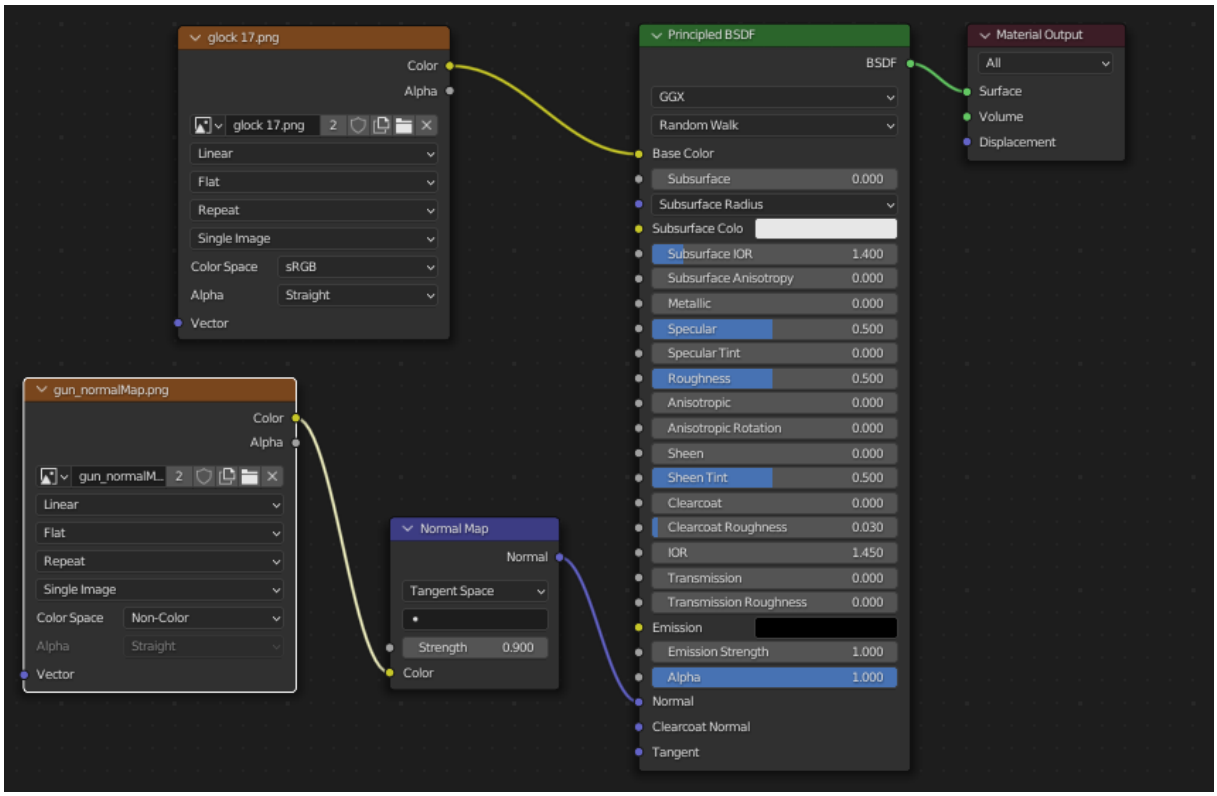
Umjesto manipuliranja vrhova, bridova ili ploha, nad aktivnim elementom radimo uniju ili presjek s drugim dijelovima pritom tijekom izrade nikad trajno ne primjenjujemo boolean modifikator jer bi tada promjene postale trajne i nepromjenjive. Svaki od oblika koji radi presjek ili

uniju s aktivnim objektom možemo u bilo kojem trenutku modificirati i promjene će biti odmah vidljive i na glavnom objektu na kojem koristimo niz boolean modifikatora. Nedestruktivnim načinom vrlo lako manipuliramo oblikom i izgledom modela. Na kraju kada smo zadovoljni izgledom modela jednostavno trajno primijenimo sve modifikatore koje smo koristili, pritom treba obratiti pažnju da je također važan i poredak samih modifikatora koje se koriste nad odabranim objektom. Modifikatori pri vrhu prvi se primjenjuju na pogled, dok oni niže se primjenjuju tek nakon što su se modifikatori iznad njih primijenili. Stoga ukoliko trajnim primjenjivanjem koristimo drugačiji redoslijed primjenjivanja modifikatora finalan objekt vrlo vjerojatno neće poprimiti isti izgled kao prikaz oblika prije trajnog primjenjivanja njegovih modifikatora.

Problem vezan uz nedestruktivan način rada je neuredna topologija koja proizlazi iz tako izrađenog modela. Izrađen model nakon trajnog primjenjivanja modifikatora omeđen je plohama sa raznim brojem stranica što onemogućava daljnji rad sa brojnim alatima. Ukoliko kasnije uvidimo da model nije dovoljno detaljan i želimo primijeniti subdivision surface modifikator kako bi povećali rezoluciju modela, vrlo vjerojatno bi primjenjivanjem modifikatora dobili neželjene artefakte unutar i/ili izvan izrađenog modela. Također loša topologija modela utječe na sjenčanje modela i odbijanje svjetlosti. Ploha omeđena s više od četiri stranice odbijala bi svjetlost na drugačiji način i opet stvorila artefakt krivog loma svjetlosti nad modelom. Ti artefakti događaju se zato što se na kraju poligoni - odnosno plohe izrađenog modela triangu-liraju *(eng. mesh triangulation)* odnosno pretvaraju u trokute. Zato je uvijek poželjno pri izradi modela koristiti plohe s tri ili četiri stranice budući da računalo plohu omeđenu s četiri stranice zapravo dijagonalno podijeli na dva trokuta.

3.3.4. Izrada materijala za izrađeni model pištolja

Izrađeni model pištolja poprimio je oblik nalik na preuzetu referentnu sliku, no obojan je jednoličnom sivom bojom. Model pištolja sastavljen je od međusobno razdvojenih dijelova drške pištolja, cijevi za ispaljivanje metaka, čekića za ispaljivanje metaka, gornjeg oklopa pištolja te okidača. Iako sam pri početku spomenuo kako procesu bojanja modela prethodi proces rezanja dijelova modela na plohe unutar 2D UV prostora *(eng. UV unwrapping)* taj proces ćemo ovdje preskočiti. Svaki razdvojeni dio izrađenog vatrenog oružja obojat ću jednobojno i time izbjeći nepoželjno rastezanje teksture nad samim modelom koje se javlja kada plohe modela nisu ispravno postavljene u UV prostor. Kako bi mogli kreirati nove materijale i vizualno istovremeno vidjeti promjene nad izrađenim modelom prebacit ćemo se u radni prostor sjenčanja *(eng. shading workspace)*. Prostor je podijeljen na dva pogleda, iznad se prikazuje model zajedno s materijalima koje smo na njega primijenili, dok se ispod prikazuje uređivač sjenčanja *(eng. shader editor)* za materijal primijenjen na aktivni dio modela. Pri početku kada model još nema niti jedan pridružen materijal uređivač sjenčanja je prazan budući da scena ne prikazuje ni jedan materijal. Kreiranjem materijala automatski se unutar uređivača prikazuju dva međusobno povezana čvora grafa*(eng. graph node)*. Čvor Principled BSDF koji svoj sadržaj prenosi u čvor naziva Material Output. Na sljedećoj slici prikazan je postav čvorova kompleksnijeg materijala koji kao boju koristi sliku "glock17.png" i kao mapu normala "gun_normalMap.png".



Slika 11: Materijal teksture zajedno sa mapom normala

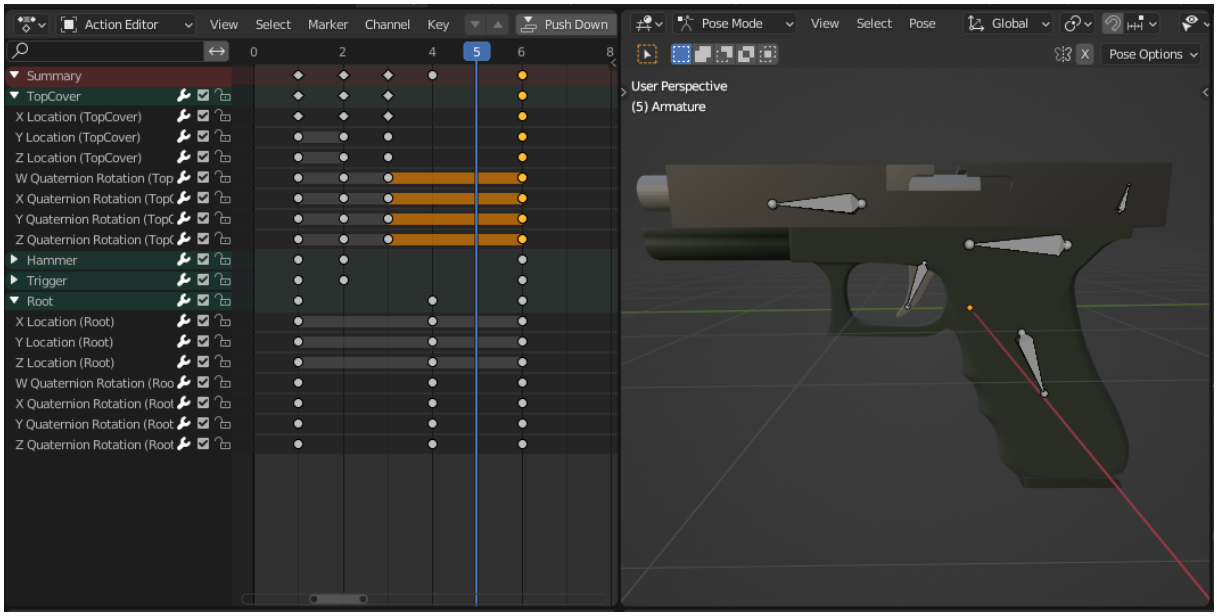
Čvorovi imaju svoje ulaze i izlaze prikazane različito obojenim točkama. Boje točaka označuju tip podataka koji atribut očekuje kako bi mogao ispravno prikazati materijal. Tako primjerice u gornjem primjeru, sliku mape normala najprije moramo pretvoriti pomoću čvora "Normal Map" kako bi je mogli ispravno koristiti unutar prikazanog materijala. Materijali za izrađeni model vatrenog oružja jednostavnije su naravi, stoga za njihovu izradu koristimo isključivo Principled BSDF i Material Output čvorove. Principled BSDF je glavni čvor koji modelu daje boju promjenom "Subsurface color" atributa, te razna fizička svojstva materijala poput grubosti samog materijala *(eng. roughness)* koja utječe na lom i odbijanje svjetlosti , metalnog svojstva *(eng. metalic property)* koji materijalu dodaje metalni odsjaj i drugih. Principled BSDF čvor spaja se na čvor naziva "Material Output". To je poseban čvor koji unesene vrijednosti ostalih čvorova primjenjuje na aktivni materijal. Unutar Principled BSDF čvora pažnju ćemo usmjeriti na promjenu "Subsurface radius", "Metalic" i "Roughness" atributa. Promjenom tih atributa dobit ćemo različite materijale koje primjenjujemo nad dijelovima izrađenog modela.

3.3.5. Izrada animacija za izrađeni model pištolja

Izrađeni model trenutno je statičan, dok u stvarnom svijetu pištolj ima pokretne dijelove primjerice prilikom pucnja pištolj ispaljuje metak, a opruge unutar pištolja ublažuju silu ispaljivanja metka i izbacuju iskorištenu čahuru ispaljenog metka. Kako bi model realističnije prikazivao stvaran pištolj izradit ćemo osnovne animacije koje ćemo kasnije koristiti unutar Unity alata.

Kako bi dijelove pištolja mogli pomicati dijelovima pištolja potrebno je pridružiti pripadajuće kosti *(eng. bone)* armature. Armatura, odnosno kosti u scenu dodaju se na isti način kao

i osnovni geometrijski likovi - pritiskom na tipke SHIFT + A otvara se padajući izbornik elemenata koje možemo dodati u scenu te iz padajućeg izbornika odaberemo opciju armature. Kosti približno pozicioniramo i orijentiramo u smjeru dijelova kojima ih želimo pridružiti. Nakon toga označimo najprije dio oružja te nakon toga kost koju dijelu želimo pridružiti te koristeći operaciju roditeljstva objekata *(eng. object parenting* združimo dio s njemu pripadajućom kosti. U novo kreiranom odnosu objekata dio pištolja nazivamo djetetom združene mu kosti, odnosno kost je roditelj pridruženog dijela pištolja. Sada svaki pomak ili rotacija kosti također pomiču i povezani dio pištolja. Na isti prikazani način pridružimo ostale dijelove ostalim kreiranim kostima. Sada kada svi dijelovi pištolja imaju združene njima pripadajuće kosti potrebno je još i kosti objediniti u veću cjelinu koju nazivamo armaturom modela. Sam redoslijed hijerarhije itekako je bitan, budući da pomak ili rotacija roditelj utječu na pomak i rotaciju njemu pridružene djece. Analiziramo li model pištolja možemo uvidjeti kako su svi dijelovi povezani na dršku pištolja pa tako ćemo kost drške pištolja učiniti roditeljem svih ostalih kostiju. Prva kost u hijerarhiji armature često se naziva korijenom *(eng. root)* stoga sam je ovdje i ja tako nazvao. Ostale kosti spajamo na kost korijena ponovo korištenjem već spomenute operacije roditeljstva objekata čime model dobiva funkcionalnu armaturu koja je spremna za izradu animacija. Same kosti unutar blendera prikazane su kao piramide koje na svakom kraju imaju 2 male kugle koje označavaju početak, kraj i orijentaciju kosti. Sam izgled armature modela prikazan je na slici 12.



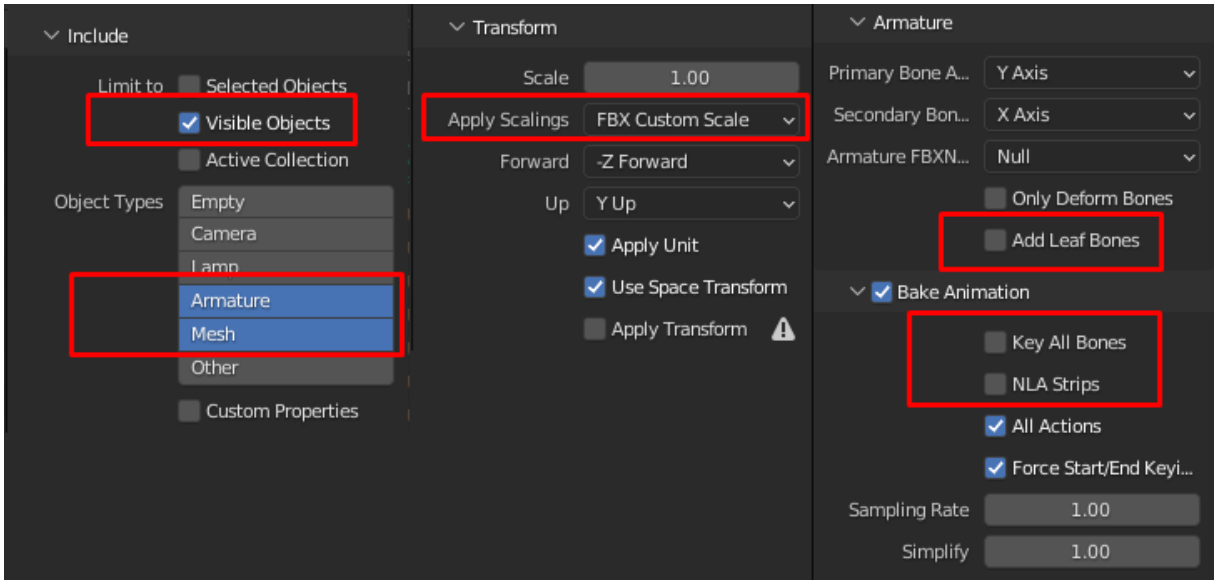
Slika 12: Prikaz animacije pucanja iz pištolja glock17

Slika 12. prikazuje dva pogleda, lijevo je prikazan uređivač akcija *(eng. action editor)* dok je desno prikazan model pištolja s pripadajućom armaturom. Uređivač akcija trenutno izvršava izrađenu animaciju pucanja koja se sastoji od šest koraka *(eng. keyframe)*. Svaki korak traje jednako dugo kao prethodni, odnosno sljedeći korak. Proces animiranja modela zapravo je bilježenje svojstava odabranih kostiju u određenom vremenskom koraku, dok privid pokreta dobivamo procesom interpolacije između koraka postepenim mijenjanjem svojstava kostiju prolaskom vremena kako bi svojstva dostignula vrijednosti nadolazećeg koraka. Svaka animacija izvodi se određenom brzinom, u ovom konkretnom slučaju animacije za pištolj izvode

se brzinom od dvadeset pet koraka u sekundi, no kada bi kompleksnost animacije zahtjevala detaljniju granularnost ta se brzina može povećati.

3.3.6. Izvoz modela pištolja u format prikladan za rad s Unity alatom

Blender izrađene modele sprema u .blend datoteke pa je tako i finalni model zajedno s pripadajućim materijalima i animacijama također spremljen u .blend datoteku. Međutim Unity ne podržava izravan rad sa .blend datotekama. Kako bi bili sigurni da će se model ispravno prikazati i animirati unutar Unity alata spremljenu .blend datoteku moramo izvesti *(eng. export)* u .fbx format s kojim Unity interno radi. Međutim limitacije fbx formata zahtijevaju nekoliko koraka prilagodbe modela prije same pretvorbe u fbx format. Ishodište modela *(eng. model origin)* mora biti u ishodištu koordinatnog sustava unutar blender alata, u suprotnom animacije modela izvodit će se i rotirati s pozicijskim i rotacijskim odmakom jednakim udaljenosti između ishodišta modela i ishodišta koordinatnog sustava. Model mora biti rotiran tako da gornja strana modela gleda prema +Y koordinatnoj osi, dok prednja strana modela odnosno cijev pištolja gleda prema +Z koordinatnoj osi. Tako rotiranom modelu potrebno je primijeniti rotaciju te nakon toga model dodatno rotiramo oko x koordinatne osi za 90 stupnjeva. Ako se prilikom izrade materijala koriste kompleksni grafovi koji kombiniraju primjerice osnovnu teksturu, metalnu mapu *(eng. metalic map)* i mapu šupljina *cavity map* njih je potrebno objediniti unutar jedne teksture kako bi Unity prepoznao i automatski prikazao materijal nad izvedenim modelom. Nakon provjere da izrađeni model zadovoljava gore navedena ograničenja klikom na Blender meni File > Export > fbx otvaramo prozor s dodatnim opcijama spremanja modela u fbx format.



Slika 13: Postavke izvoza modela u FBX format

Slika 13. prikazuje dodatne opcije koje moramo označiti kako bi se model ispravno spremio u fbx datotečni format. Budući da smo tijekom izrade modela povremeno radili duplikate modela kako bi očuvali originalni model prije značajnih izmjena nad njime, najprije moramo označiti opciju limitiranja izvoza samo na dijelove koji su vidljivi unutar blender alata. Nadalje prilikom izrade modela izradili smo oblik , materijale i animacije stoga tipove podataka koje že-

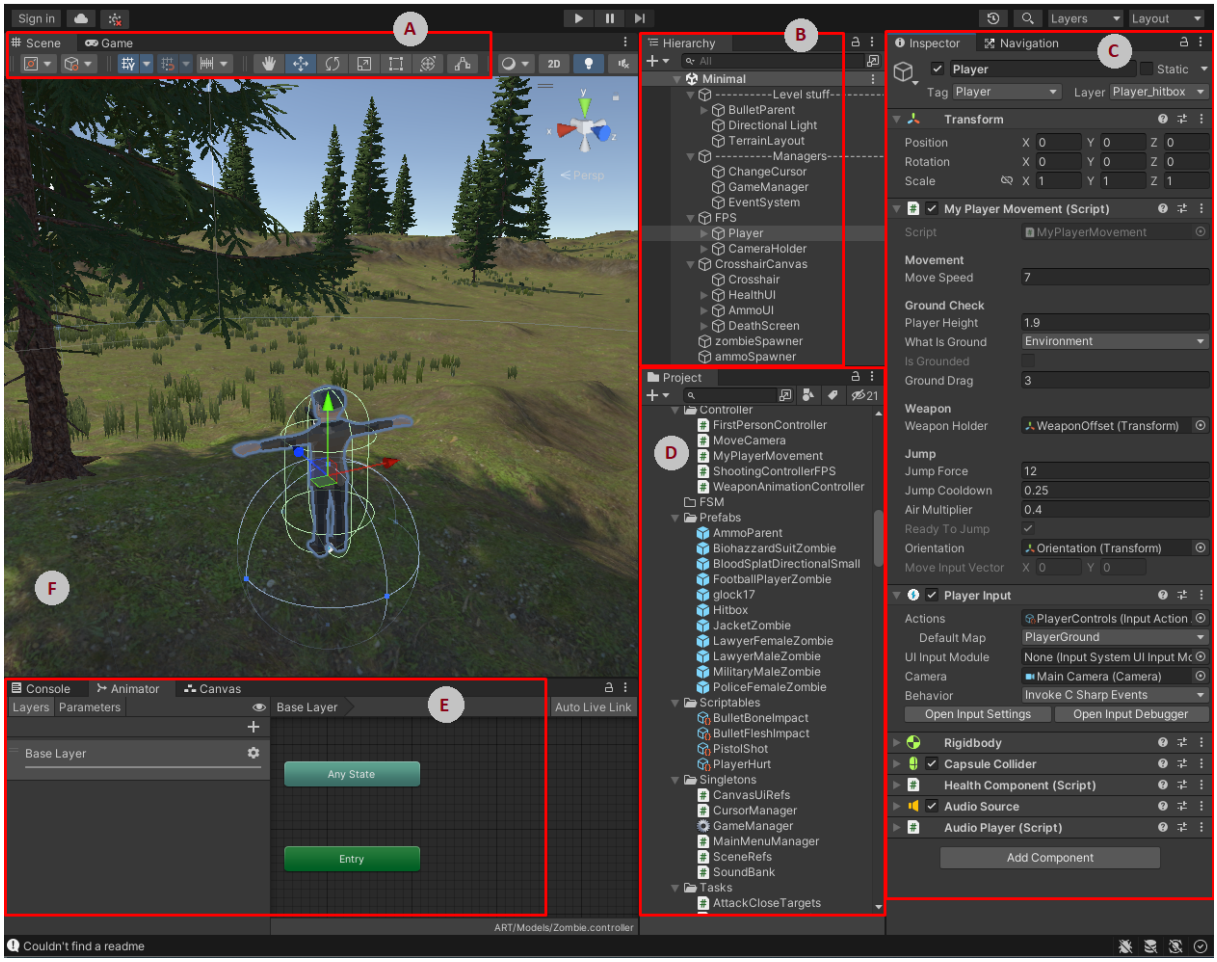
limo izvesti iz .blend datoteke označujemo mesh koji sprema oblik i pripadajuće mu materijale, te armaturu koja sadrži hierarhiju kostiju i animacije definirane nad njima. Ostale opcije koje je potrebno primjeniti označene su crvenim okvirom na slici 13. Ispravnim pretvaranjem izrađenog modela u fbx format prikladan za korištenje u Unity alatu, model je spreman za korištenje unutar Unity alata.

3.4. Unity

Unity kao alat za izradu video igra i interaktivnih iskustva unutar virtualne stvarnosti prvi je puta objavljen 2005. godine kao alat za izradu video igra na Mac OS X operativnom sustavu. Danas projekti realizirani unutar Unity alata imaju mogućnost izvođenja na brojnim platformama od Microsoft Windows, Mac OS i Linux računalnih operativnih sustava, Android i iOS mobilnih operativnih sustava, PlayStation, XBox i ostalih igračih konzola pa čak i unutar web pretraživača pomoću WebGL programske podrške.[8] Unity se konstantno unaprjeđuje kako bi ostao relevantan u prostoru izrade video igra i interaktivnih solucija u kojem djeluje. Glavne funkcionalnosti i servisi Unity alata izrađeni su u C++ programskom jeziku dok aplikacijsko programsko sučelje *(eng. application programming interface* koristi C# programski jezik koji je po mišljenju mnogih dosta jednostavniji za korištenje od C++ programskog jezika. Unity alat ima više licenca prilagođenih krajnjem korisniku na odabir. Svatko može koristiti osobnu licencu Unity alata ukoliko na proizvodima izrađenim u Unity alatu godišnje ne zarađuje preko sto tisuća dolara, u suprotnom tu je i Unity Plus, Pro ili Enterprise verzija koja se pretplaćuje na godišnjoj bazi. Unity alat svojom iznimno pogodnom osobnom licencom pretežito interesira hobi programere i dizajnere igra, no danas brojne profesionalne tvrtke za izradu video igra također odabiru Unity kao glavni alat za razvoj svojih proizvoda kako bi ubrzali proces izrade igra i uštedjeli na troškovima održavanja postojećih sustava. Tijekom izrade ovog projekta koristio osobnu licencu Unity alata te Microsoft Visual Studio kao glavni uređivač koda Unity skripata.

3.4.1. Unity korisničko sučelje

Slika 14. prikazuje korisničko sučelje Unity alata. Na samom vrhu iznad označenih dijelova nalazi se traka koja omogućuje prijavu u korisnički račun, a osim toga sadrži i play, pause i step tipke za pokretanje igre unutar Unity alata, dok su ostale opcije raspodijeljene unutar otvorenih prozora. Unity alat iznimno je modularan te korisnicima omogućava proizvoljnu promjenu dimenzija, pozicije i vidljivosti svakog otvorenog prozora. Prozor označen slovom A sadrži alate za upravljanje objektima unutar scene iako za iste naredbe postoje prečaci kao kombinacije specifičnih tipaka na tipkovnici. B dio jest prozor koji prikazuje hijerarhiju objekata igre *(eng. game object)* unutar aktivne scene koju uređujemo. Važno je napomenuti kako je poredak elemenata u hijerarhiji bitan budući da promjena na vrhu hijerarhije utječe na sve njemu ugniježđene objekte *(eng. nested object)*. U inspektor prozoru označenim slovom C uglavnom provodimo najviše vremena. Unutar inspektor prozora smještene su sve komponente koji aktivni objekt igre sadrži zajedno sa svim izloženim *(eng. exposed)* atributima. Unutar samog unity alata najčešće sam koristio upravo prozor inspektora kako bih ispravno prilagodio odre-



Slika 14: Korisničko sučelje Unity alata

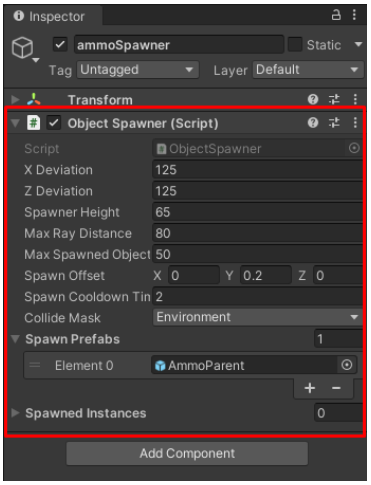
đene komponente i objekte unutar igre. Važno je spomenuti da su promjene unutar inspektor prozora trajne ako promjene radimo nad igrom prije njezina pokretanja, no ukoliko najprije pokrenemo igru klikom na gumb Play pa zatim mijenjamo komponente i atribute, nakon zaustavljanja igre te promjene se neće spremiti. Stoga kada nismo sigurni kakve točno promjene želimo, možemo eksperimentirati unutar pokrenute igre, promjene će biti istovremeno odražene unutar igre te možemo biti sigurni da slučajno načinjene promjene neće utjecati na prijašnje postavke objekata unutar igre. D dio naziva se prozorom projekta. Ovaj prozor sadrži sve resurse *(eng. asset)* ubačene u naš projekt. Neki od tipičnih resursa koje možemo pronaći unutar prozora projekta primjerice su zvučne datoteke, 3D modeli, izrađene Unity skripte, skriptabilni objekti *(eng. scriptable object)*, "Prefab" objekti, materijali, isječci animacija, Unity programska proširenja te svi ostali objekti multimedije koji se mogu, ali i ne moraju koristiti unutar projekta. Lijevo dolje nalaze se dodatne kratice prozora na koje možemo prebaciti pogled. Naposljetku unutar prozora označenog slovom F nalazi se vizualna reprezentacija scene, odnosno igre ovisno koja kratica iznad je aktivna. Ovaj pogled najčešće koristimo kako bi pozicionirali elemente unutar scene na odgovarajuće mjesto ili testirali igru prilikom pokretanja igre. Osim spomenutih prozora mnogi prozori i opcije nisu prikazani jer ostale prozore otvaramo i zatvaramo prema potrebi, primjerice ako trebamo pregledati, kreirati ili izmijeniti određenu animaciju nad objektom otvorit ćemo prozor za animacije.

3.4.2. Osnovna arhitektura sustava unity alata

Nove funkcionalnosti igre kodiramo unutar zasebnih Unity skripata korištenjem C# programskog jezika i biblioteka. Najčešće jedna Unity skripta sadrži metode i atribute jedne klase derivirane iz "MonoBehaviour" ili "ScriptableObject" klase, no Unity skripta nije limitirana na samo jednu klasu niti je bitan tip klase koju skripta definira. Međutim klase naslijeđene *(eng. inheritance)* iz MonoBehaviour ili ScriptableObjekt klasa dobivaju određene naslijeđene metode i funkcionalnosti. U nastavku ću objasniti osnovne Unity klase koje sam koristio prilikom izrade arhitekture sustava igre te ću ukratko objasniti kreiranu arhitekturu projekta.

3.4.2.1. Klasa MonoBehaviour

Sve klase koje nasljeđuju od MonoBehaviour klase Unity sustava prikazuju se kao komponente unutar inspektor prozora koji omogućava promjenu vrijednosti vidljivih atributa čak i prilikom izvođenja igre, a promjene atributa vidljive su unutar igre u stvarnom vremenu. MonoBehaviour klasa također ima svojstvo dodavanja svoje funkcionalnosti na bilo koji objekt unutar scene kao komponenta tog objekta. Sljedeća slika prikazuje "ammoSpawner" objekt unutar scene te dvije njemu pridružene komponente. Transform komponenta dodaje se svakom novokreiranom objektu kako bi objekt mogli pozicionirati unutar scene, dok druga komponenta naziva Objekt spawner je unity skripta koju sam izradio kako bi ovaj objekt unutar igre imao funkcionalnost dodavanja municije za pištolj na nivo u kojem se igrač nalazi.



Slika 15: Prikaz ObjectSpawner klase kao komponente game objekta u inspektor prozoru

Kada nam je potrebna dodatna funkcionalnost, izradimo novu Unity skriptu koja nasljeđuje iz MonoBehaviour klase, definiramo metode koje obavljaju traženu funkcionalnost te skriptu pridružimo objektu putem "Add Component" tipke pri dnu inspektor prozora. Još bolja stvar je da tu jednom izrađenu Unity skriptu možemo pridružiti proizvoljnom broju objekata unutar scene. Tako sam istu "ObjectSpawner" skriptu koristio kako bi na teren tokom izvršavanja igre mogao dodavati neprijatelje koji traže igrača.

Osim navedenih svojstva, MonoBehaviour klasa definira funkcije životnog ciklusa *(eng. lifecycle)* koje Unity automatski poziva pri izvršavanju pokrenute igre. U nastavku ću navesti i

opisati neke od najčešće korištenih funkcija životnog ciklusa komponente unutar igre izrađenog putem nasljeđivanja iz MonoBehaviour klase.

```
using UnityEngine;

public class TestKlasa : MonoBehaviour {
    private void Awake() {
        // used to initialize variables or states before the application starts
    }
    private void OnEnable() {
        // called every time object is enabled
    }
    private void OnDisable() {
        // called every time object is disabled
    }
    void Start() {
        // Start is called before the first frame update
    }
    void Update() {
        // Update is called once per visual frame update
        // events inside unity are listening inside update loop
    }

    private void FixedUpdate() {
        // called independently of framerate at fixed intervals
    }

    private void LateUpdate() {
        // called after all Update functions have been called
    }

    private void OnCollisionEnter(Collision collision) {
        // called when physics system detects the collision with this game object
        // one of the colliders needs to have rigidbody attached for collision to be
             registered by physics engine
    }
}
```

Awake funkciju Unity poziva kod svih aktivnih objekata unutar scene prilikom pokretanja te scene u trenutku učitavanja komponenti pridruženih aktivnom objektu. Awake funkcija poziva se samo jednom isključivo kod učitavanja početne scene prije bilo koje druge funkcije životnog ciklusa ili prilikom dodavanja novih objekata unutar scene korištenjem Instantiate() naredbe. Najčešće Awake funkciju koristimo kako bi dohvatili reference na druge skripte/komponente čije metode izrađena skripta koristi. Ako je objekt kod učitavanja scene neaktivan, Awake funkcija neće se izvršiti nad njemu pridruženim komponentama.

OnEnable i OnDisable naredbe pozivaju se svaki puta kada se pripadajuća komponenta isključi ili uključi na objektu u sceni. Najčešće ovim funkcijama dodajemo ili brišemo slušače događaja *(eng. event listener)* na koje izrađena klasa izvodi određene akcije.

Start naredba pokreće se nad aktivnim objektima scene nakon što se je izvršila Awake naredba. Start naredbu možemo i privremeno odgoditi tako da početni objekt bude neakti-

van prilikom početnog učitavanja scene, pa zatim kasnije taj objekt aktiviramo kako bi izvršili Start naredbu. Budući da se prilikom izvođenja Start naredbe Awake naredba izvršila nad svim objektima, a awake naredbu koristimo za međusobno referenciranje komponenti, unutar start naredbe najčešće inicijaliziramo privatne atribute i vrijednosti skripti koje nisu prikazane u inspektoru na njihove početne vrijednosti.

MonoBehaviour klasa sadrži i tri vrste update naredbi: Update, FixedUpdate i LateUpdate koje se pozivaju u različitim periodičkim intervalima ukoliko objekt kome je ta skripta pridodana aktivan. Update naredba poziva se prilikom osvježenja vizualnog okvira *(eng. frame)* igre. Update naredba ne izvodi se u pravilnim vremenskim ciklusima budući da kompleksnost akcija varira prilikom prikaza svakog vizualnog okvira. Broj izvođenja Update naredbe unutar jedne protekle sekunde limitiran je snagom računalnih komponenti na kojima se igra izvršava budući da snažnije komponente brže obrađuju i pripremaju podatke potrebne za prikaz sljedećeg vizualnog okvira. Unutar Update naredbe najčešće stavljamo kod koji konstantno provjerava određene parametre komponenti te izvršava određene funkcionalnosti ukoliko su neki od tih parametra zadovoljeni. Također često unutar update naredbe koristimo provjere pritiska određenih tipki kako bi komponenta prilikom pritiska određene tipke, bez ikakvog zastoja na sljedećem osvježenju vizualnog okvira, izvršila odgovarajuće akcije. [9]

FixedUpdate funkcija životnog ciklusa izvodi se u pravilnim definiranim intervalima. Ukoliko nismo dirali početne postavke projekta, FixedUpdate naredba izvodit će se pedeset puta u sekundi. Unutar FixedUpdate naredbe najčešće stavljamo kod koji izvršava neke fizičke simulacije. Razlog tome je taj da se FixedUpdate naredba, za razliku od Update naredbe, izvršava unutar pravilnih vremenskih intervala, fizički izračuni često koriste razmak proteklog vremena za izračunavanje fizičkih simulacija. Također i cijeli sustav fizike Unity programa izvršava se unutar FixedUpdate funkcije zbog istog navedenog razloga.

LateUpdate naredba poziva se nakon što su se sve update naredbe izvršile. Najčešće ovdje stavljamo kod koji ovisi o vrijednostima koje se mijenjaju unutar Update naredbi nad drugim komponentama budući da je redoslijed izvršenja komponenti i objekata unutar scene neodređen. Dobar primjer ispravnog korištenja ove naredbe bio bi pozicioniranje kamere koja prati igrača, dok igrač mijenja svoju poziciju izvođenjem akcija unutar Update naredbe. [10]

Funkcija OnCollisionEnter nad komponentom se poziva kada ugrađeni Unity sustav fizike detektira sudar dvaju tijela unutar scene. Kako bi sustav mogao detektirati sudar jedno tijelo mora posjedovati "Collider" komponentu , dok drugo tijelo mora sadržavati "Rigidbody" komponentu. Collision atribut koji se funkciji prosljeđuje sadrži dodatne informacije o detektiranom sudaru primjerice pozicije dodira sudarenih objekata i slično. [10] Kako ovu komponentu poziva sustav fizike, detekcija samih sudara provjerava se prilikom izvođena FixedUpdate naredbe Unity sustava fizike.

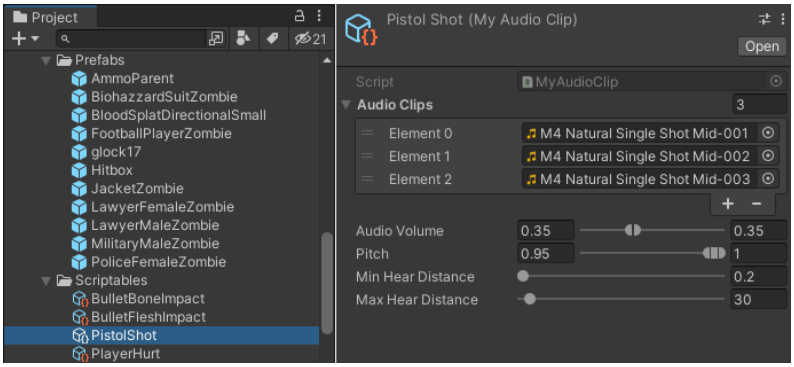
3.4.2.2. Klasa ScriptableObject

Klasa ScriptableObject sadrži par posebnosti koje nasljeđivanjem također prenašamo na našu klasu. ScriptableObject klasa ne može se dodati kao komponenta nekom objektu u sceni kao što to može MonoBehaviour klasa, no posebnost ScriptableObject klase je mo-

gućnost kreiranja unutar prozora projekta kao .asset datoteka. Unutar asset datoteke možemo unaprijed definirati vrijednosti koje takva datoteka sadržava dok MonoBehaviour klasa najčešće definira tipove podataka, a svaka komponenta skripte inicializira vrijednosti putem inspektora. ScriptableObject klasa od ranije objašnjenih funkcija životnog ciklusa posjeduje samo Awake, OnEnable i OnDisable funkcije. Atributi promjenjeni unutar ScriptableObject klasa trajno čuvaju promjene neovisno mijenjamo li njene vrijednosti prilikom izvođenja igre ili unutar samog Unity uređivača prije pokretanje igre. Unity alat u pozadini ScriptableObject i MonoBehaviour klasu reprezentira istom C++ klasom stoga čudi činjenica da ScriptableObject klasa sadrži reducirani set mogućnosti MonoBehaviour klase. Određene funkcije životnog ciklusa namjerno su izostavljene iz ScriptableObject klase budući da je ScriptableObject klasa pretežito zamišljena kao resurs skupnih definiranih podataka i metoda unutar izrađenog projekta prema čijim podacima MonoBehaviour klase čitaju i pišu podatke te izvršavaju određene akcije. Osim kao projektni resurs ScriptableObject klasu možemo instancirati i dinamički, prilikom izvođenja igre, no ona će se tada prilikom zatvaranja programa automatski uništiti. [11] Snalažljivom upotrebom ScriptableObject klase unutar Unity alata možemo izraditi skalabilan i modularan sustav, no u ovome radu, ScriptableObject klasu koristio sam za izradu prilagođene klase video zapisa čiji primjer slijedi u nastavku.

```
[CreateAssetMenu(menuName = "ScriptableObject/Audio Clip")]
public class MyAudioClip : ScriptableObject {

    public List<AudioClip> audioClips;
    [MinMaxSlider(0f, 1f)]   // NaughtyAttributes
    public Vector2 audioVolume = new Vector2(0.5f,0.5f);
    [MinMaxSlider(0f, 1f)]   // NaughtyAttributes
    public Vector2 pitch = new Vector2(1f,1f);
    [Range(0f, 1000f)]
    public float minHearDistance;
    [Range(0f, 1000f)]
    public float maxHearDistance = 30f;
}
```

Klasa "MyAudioClip" sadrži listu audio isječaka, te atribute za definiranje glasnoče i visine zvuka. CreateAssetMenu nareba kreira stavku u padajućem izborniku pod navedenom putanjom pa klasu kreiramo kao resurs unutar projektnog prozora odabirom odgovarajuće opcije iz padajućeg izbornika.

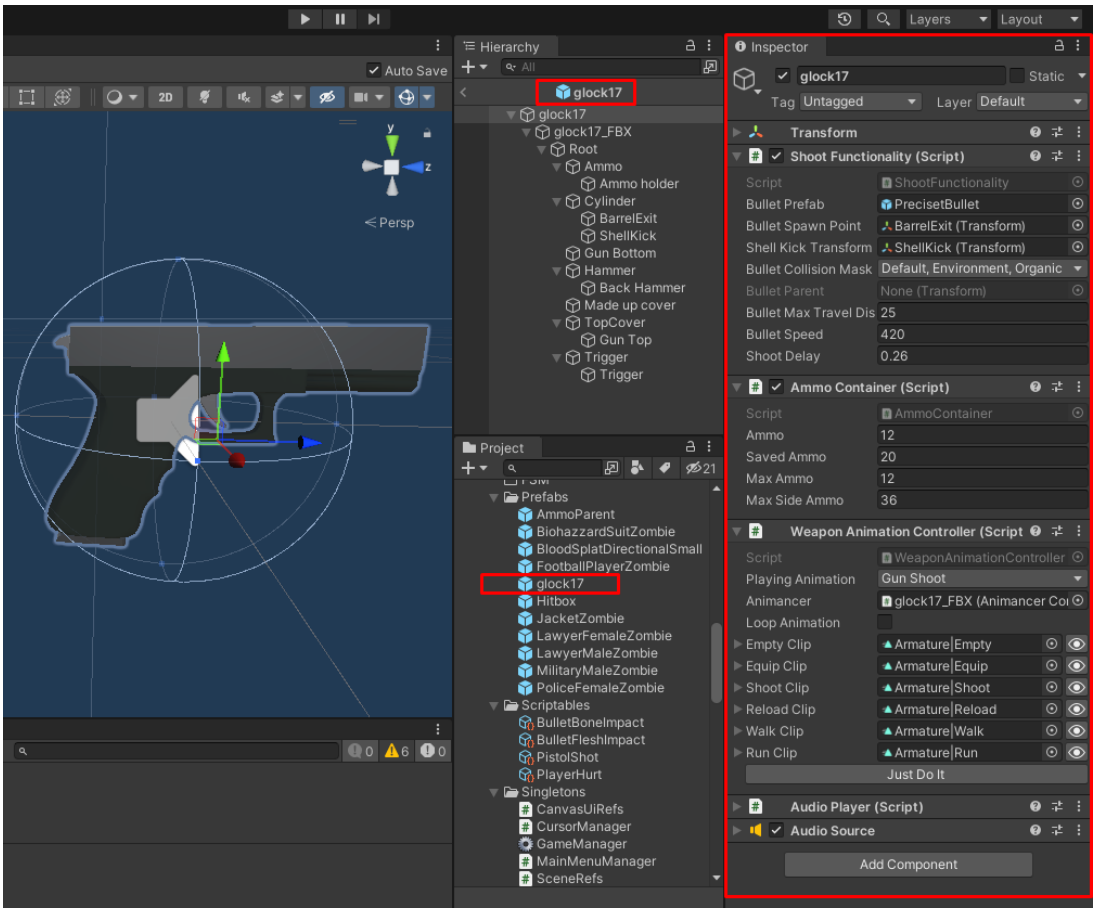


Slika 16: Prikaz skriptabilnog objekta u projektnom i inspektor prozoru

Kako bih zvukove unutar igre mogao definirati na razini čitavog projekta, a ne samo na razini komponente, odlučio sam koristiti klasu ScriptableObject. Druga važna dobivena fleksibilnost korištenjem ScriptableObject klase je pamćenje promijenjenih vrijednosti prilikom izvođenja igre. Naime glasnoća zvuka ispituje se kada se igra izvodi stoga kada bi koristili MonoBehaviour klasu, promjene nad prilagođenim atributima resetirale bi se prilikom prestanka izvođenja igre, dok korištenjem skriptabilnog objekta promjene se spremaju prilikom svake promjene atributa.

3.4.2.3. Prefab strukture

Prefab strukturu koristimo kako bi enkapsulirali izrađeni objekt unutar scene *(eng. game object, u nastavku kraće game objekt)* zajedno sa svim njemu pridruženim komponentama kao ponovno iskoristivi resurs unutar projekta. Osim toga prefab struktura podržava enkapsulaciju game objekta koji sadrži druge ugniježđene game objekte.



Slika 17: Prefab izrađenog modela pištolja i izrađenom funkcionalnosti

Strukturu prefaba možemo gledati kao šablonu *(eng. template)* iz koje kreiramo nove objekte u sceni inicijalizirane na vrijednosti definirane unutar prefab strukture. Također prefab struktura omogućava i ugnježđivanje prefab struktura unutar sebe i kreiranje prefab varijanti. Objekti kreirani preko prefab strukture mogu prebristati pojedine vrijednosti svojim prilagođenim vrijednostima putem inspektor prozora. Prefab strukture najčešće koristimo za instanciranje objekata kako svaki puta objekt ne bi morali izgraditi iznova prilikom pokretanja igre. Objekt

zajedno sa svim komponentama, njihovim vrijednostima pa čak i podređenom hierarhijom objekata spremamo kao prefab strukturu unutar projekta. Kasnije unutar igre, gotove unaprijed izgrađene kompleksne objekte instanciramo jednostavnim pozivom Instantiate() naredbe koja će izgraditi objekt kao identičnu kopiju pripadajuće mu prefab strukture. Prethodno izrađeni model pištolja u scenu igre stavljamo učahurenog unutar glock17 prefab strukture. Sama prefab struktura pištolja izgleda dosta kompleksno, no u suštini izrađeni model pištolja ugnijezdili smo unutar "glock17" praznog objekta hijerarhije. Izrađeni model morali smo smjestiti unutar praznog objekta zato što nad pozicijom i rotacijom izrađenog modela pištolja upravljaju njegove animacije, a kako bi model mogli pomicati unutar scene koristimo pomicanje glavnog objekta hijerarhije "glock17" budući da pomicanje i rotiranje glavnog objekta hijerarhije utječe na sve njemu podređene objekte pa tako i na poziciju podređenog mu modela pištolja. Osim toga, glock17 objektu na vrhu hierarhije dodali smo i nekoliko komponenti koje modelu dodaju mogućnost pucanja, prate razinu municije pištolja i izvršavaju prikladne animacije ovisno o stanju u kojem se pištolj unutar igre nalazi. Kako prilikom postavljanja pištolja u novu scenu ponovo ne bi morali kreirati opisanu hierarhiju objekata i njima pridružiti odgovarajuće skripte, čitavu izrađenu strukturu pretvorili smo u prefab strukturu - unaprijed izrađenu šablonu koju postavljamo u scenu ukoliko unutar scene želimo dodati izrađeno vatreno oružje.

3.4.2.4. Arhitektura izrađene igre

Čitava igra izrađena je korištenjem prethodno opisanih klasa i struktura. ScriptableObject koristi se za spremanje audio isječaka. MonoBehaviour klasa, odnosno komponente koriste se za dodavanje funkcionalnosti kreiranom objektu igre, dok sam pomoću prefab strukture unaprijed izradio kompleksne game objekte poput igraća , neprijatelja, pištolja te metka koji se stvaraju u sceni prilikom pucanja iz pištolja.

Ponekad nam je i potrebna komunikacija između različitih game objekata. Kako bi doskočio tome problemu, kroz ovaj projekt koristim singleton uzorak dizajna koji sprema vrijednosti referenci važnih izrađenih game objekata i komponenti kako bi bile dostupne ostalim objektima u sceni. Prednost ovakvog pristupa je jednostavnost budući da singleton klasi preko koda može pristupiti bilo koja klasa te je ovakav uzorak doista primjeren za izradu prototipa igre. No u stvarnoj produkciji igra, singleton uzorci koriste se rijetko zato što sa sobom često vuku ovisnosti na druge komponente unutar igre i ne omogućavaju zadovoljavajući stupanj modularnosti i ponovne iskoristivosti koda.

```
public class SoundBank : MonoBehaviour {

    private static SoundBank _Instance;
    public static SoundBank Instance {
        get {
            if (_Instance == null)
                _Instance = FindObjectOfType<SoundBank>();
            return _Instance;
        }
    }
    ...
}
```

Unutar projekta koristio sam SoundBank singleton koji sadrži skriptabilne objekte audio isječaka kako bi u bilo kojoj skripti mogao reproducirati zvučne efekte jednostavnim referenciranjem na globalno dostupnu klasu SoundBank. Unutar scene mora postojati samo jedan objekt sa njemu pridruženom SoundBank komponentom kako bi naredba FindObjectOfType<SoundBank>() pronašla i spremila referencu SoundBank komponente. Svaki sljedeći put singleton vraća spremljenu referencu pronađenog objekta.

3.4.3. Korištena Unity proširenja

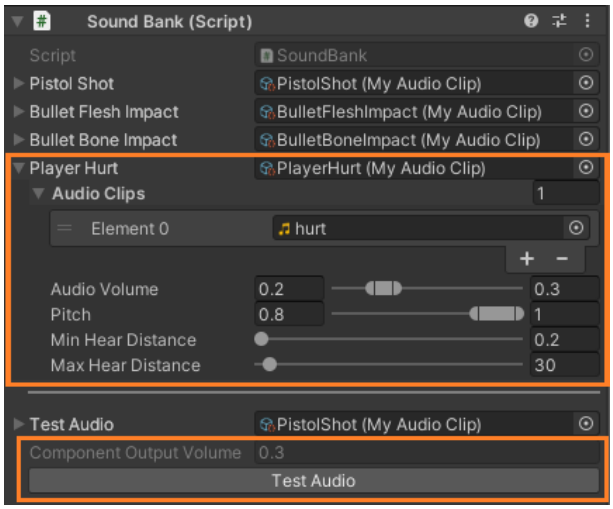
Prilikom izrade projekta koristio sam brojna besplatna, ali i kupljena programska proširenja Unity alata. Programska proširenja preuzimao sam sa GitHub i Unity asset store web stranica. Unity asset store je trgovina koja sadrži razne besplatne, ali i plaćene resurse za izradu igre. Ponuda obuhvaća veliki broj modela, animacija , tekstura, programskih proširenja Unity alata pa čak i izrađene čitave projekte i primjere. Kupljena i besplatna proširenja u projekt sam dodao preko Unity menadžera paketa. Kako bi dodali kupljene pakete najprije se moramo prijaviti u svoj Unity račun te jednim klikom na gumb Import , proširenje dodajemo unutar projekta. U nastavku ću opisati korištena preuzeta proširenja Unity alata, usporediti Unity s i bez korištenog proširenja te navesti razlog korištenja svakog od korištenih proširenja.

3.4.3.1. Naughty Attributes

NaughtyAttributes besplatno je proširenje Unity alata koje omogućuje dodatne opcije prikazivanja atributa unutar inspektora. [12] Kroz ovaj projekt, NaughtyAttributes proširenje sam koristio gotovo u svakoj Unity skripti, najčešće kako bih prikazao privatne variable unutar inspektora, ali zabranio promjenu njihove vrijednosti. Nadalje NaughtyAttributes proširenje koristio sam kako bih dodao tipke unutar inspektora kojima sam brzo mogao testirati određenu funkcionalnost te kako bih skriptabilne objekte mogao proširiti unutar drugih komponenti koje ih koriste. Korištenje NaughtyAttributes biblioteke naredbi jako je jednostavno i intuitivno, a NaughtyAttributes biblioteku ukratko ću objasniti nad ranije spomenutom SoundBank singleton klasom.

```
using UnityEngine;
using NaughtyAttributes;

public class SoundBank : MonoBehaviour {
    [Expandable]   //NaughtyAttributes
    public MyAudioClip pistolShot;
    [Expandable]
    public MyAudioClip bulletFleshImpact;
    ...
    [ReadOnly]
    public float componentOutputVolume = 0.3f;

    [Button]  //NaughtyAttributes
    public void TestAudio() {
        ...
}}
```

Kako bi skriptabilne objekte mogli prikazati i proširiti na njegove atribute unutar druge komponente koristimo atribut [Expandable] prije atributa koji referencira skriptabilni objekt. Provjeru glasnoće zvuka možemo isprobati prilikom izvođenja igre, no to također možemo učiniti i putem tipke "TestAudio" unutar inspektora koju dodajemo [Button] atributom. Atribut componentOutputVolume dodao sam kako bih demonstrirao treću vrlo korisnu i često korištenu [ReadOnly] naredbu NaughtyAttributes biblioteke koja atribut prikazuje unutar inspektora, ali onemogućava pisanje u njega jer ponekad nam je vrlo važno vidjeti stanje atributa unutar komponente, dok taj atribut primjerice skripta koristi za svoje unutarnje procese stoga taj atribut nije namijenjen izmjenjivanju vrijednosti putem inspektora. Sam promijenjen izgled inspektora prikazan je sljedećom slikom.

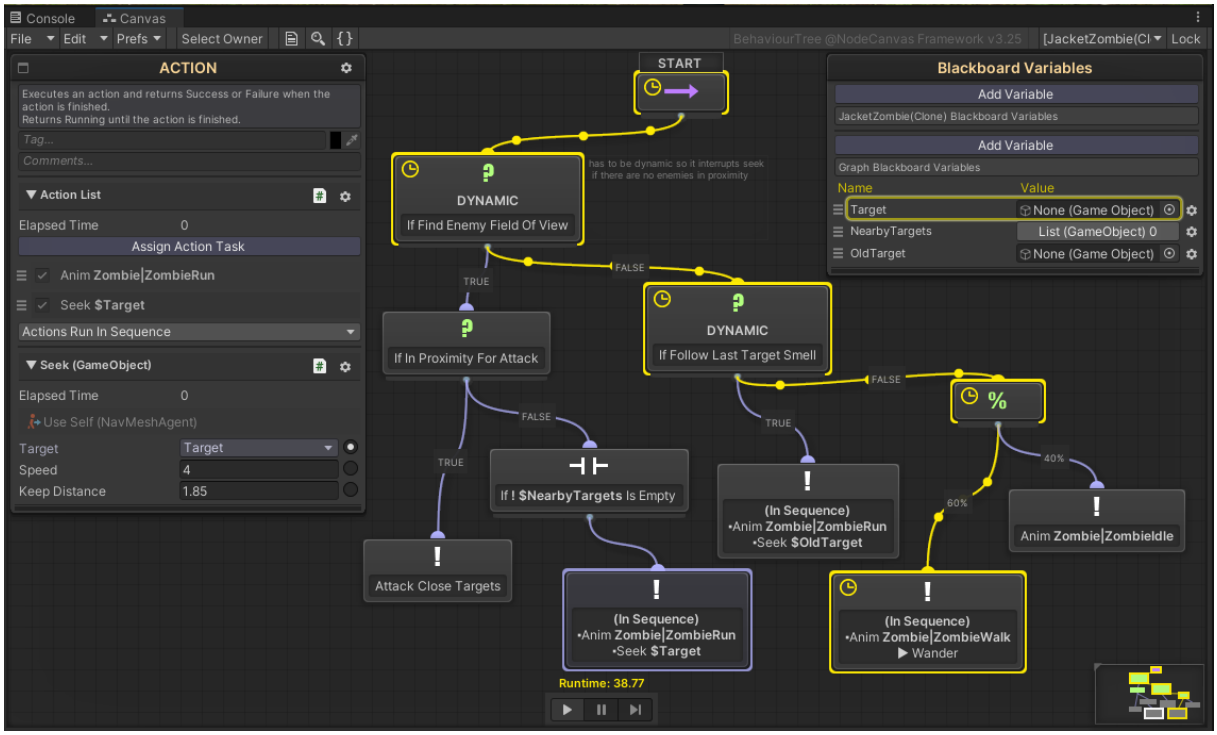


Slika 18: Izgled inspektora SoundBank komponente primjenom NaughtyAttributes proširenja

3.4.3.2. NodeCanvas

NodeCanvas plaćeno je proširenje koje nam omogućuje izrađivanje stabla ponašanja *(eng. behaviour tree)*, konačnih strojeva stanja *(eng. finite state machine)* te izradu ne linearnih dijaloga unutar Unity alata. Kroz ovaj projekt NodeCanvas proširenje koristio sam kako bih pomoću stabla ponašanja izradio kompleksno ponašanje neprijatelja unutar igre. Kako bi objektu unutar igre dodali stablo ponašanja najprije objektu moramo dodati Behaviour Tree Owner komponentu s kojom se automatski nadodaje i Blackboard komponenta. Behaviour Tree Owner komponenta posjeduje prostor za izradu dijagrama ponašanja, dok blackboard komponentu koristimo za spremanje korisnih vrijednosti koje se koriste prilikom izvršavanja stabla ponašanja. Glavna prednost korištenja vizualnog stabla ponašanja lakše je testiranje ponašanja neprijatelja jer kroz graf u svakom trenutku vidimo akcije i "razmišljanje" neprijatelja na sceni. Ukoliko unutar scene primjetimo bilo kakav problem, vrlo lako možemo locirati izvor gdje je ta greška nastala. Proširenje dolazi i sa pregršt unaprijed definiranih zadataka koje možemo dodati unutar NodeCanvas prozora. NodeCanvas osim vizualnog stabla ponašanja dolazi s čitavom bibliotekom skripata i naredbi koje možemo koristiti kako bi putem koda izradili svoje prilagođene zadatke koje želimo dodati u stablo ponašanja. Za ovaj projekt izradio sam nekolicinu takvih skripata: "AttackCloseTargets", "InProximityForAttack", "FindEnemyFieldOfView",

"FollowLastSmell" te "PlayAnimancerAnimation". Stablo ponašanja diktira redoslijed izvršavanja radnji dok se same radnje, enkapsulirane u zasebne skripte, mogu ponovno iskoristiti prilikom izrade drugih stabla ponašanja.



Slika 19: Stablo ponašanja za JacketZombie prefab unutar NodeCanvas prozora

Na prikazanoj slici s lijeve strane možemo vidjeti izbornik koji prikazuje zadatke trenutno odabranog čvora. Unutar tog izbornika čvoru možemo pridružiti dodatne nove akcije ili dodatno podesiti korištene parametre odabranog zadatka. S desne strane nalazi se Blackboard izbornik koji sadrži vrijednosti na koje se referenciraju određeni zadaci unutar čvorova. U prikazanom primjeru stablo ponašanja neprijatelja pamti ugledanu metu unutar Target vrijednosti, osjeća mete u neposrednoj okolini i zapisuje ih u listu NearbyTargets, a ukoliko u vidnom polju izgubi ugledanu metu preko OldTarget vrijednosti prati miris zadnje ugledane mete.

U središnjem dijelu nalazi se stablo ponašanja prikazano grafom s međusobno povezanim čvorovima zadataka i uvjeta za njihovo izvršavanje. Svako ponašanje počinje čvorom Start. Nakon start slijedi provjera postoji li meta u vidokrugu neprijatelja te ako postoji slijed grafa prelazi na true granu gdje proganja ugledanu metu koju i napada ukoliko je meta dovoljno blizu, u suprotnom se izvršava grana false. Neprijatelj traži i trči prema mirisu neprijatelja ako postoji neprijatelj kojega je zapamtio, inače nasumično hoda unutar nivo-a ili stoji i promatra okolinu ovisno o statistički definiranoj šansi. Na slici žutim konturama prikazan je trenutno odabran slijed akcija promatranog neprijatelja u sceni.

Čvorovi koje dodajemo unutar grafa kategorizirani su u grupe složenih čvorova *(eng. composite)*, dekoratora *(eng. decorator)*, pod grafova i krajnjih čvorova grafa koje nazivamo listovima. List je čvor koji se nalazi na dnu hijerarhije stabla ponašanja kojim završava grana grafa. Dekorator čvorovi su čvorovi s točno jednom vezom na sljedeći čvor. Oni služe ukrašavanju odnosno promjeni funkcionalnosti čvora koji slijedi nakon njih kao primjerice dekorator

koji zabranjuje pristup zadatku nakon određenog broja pristupa ili proteklog vremena. Čvorovi podgrafova omogućuju dodavanje čitavih stabla ponašanja ili strojeva stanja unutar postojećeg grafa enkapsuliranih unutar jednog čvora. Kompozitne čvorove koristimo za grananje stabla ponašanja budući da su to jedini čvorovi grafa koji mogu prosljeđivati podatke na više čvorova djece. [13] Svaki čvor osim čvorova podgrafa, mogu sadržavati jedan ili više zadataka koji se obavljaju kada je čvor aktivan. NodeCanvas razlikuje dva tipova kreiranih zadataka, zadatak akcije koji izvršava definirane akcije i zadatak uvjeta koji provjerava ispunjenje određenih uvjeta te ovisno o tome preusmjerava tok izvođenja grafa i njegovih čvorova. U nastavku ću ukratko opisati i prokomentirati izrađen zadatak akcije i zadatak uvjeta kako bi dobili detaljniju sliku mogućnosti ovog Unity proširenja.

```
using NodeCanvas.Framework;
using ParadoxNotion.Design;
using UnityEngine;
namespace NodeCanvas.Tasks.Conditions{

public class FindEnemyFieldOfView : ConditionTask {

        public BBParameter<float> radius;
        [Range(0,359)]
        public BBParameter<float> viewAngle;
        public BBParameter<LayerMask> targetMask;
        public BBParameter<LayerMask> obstructionMask;
        public BBParameter<GameObject> target;
        ...
        //Called once per frame while the condition is active.
        //Return whether the condition is success or failure.
        protected override bool OnCheck(){

        Collider[] targetsInProximity = Physics.OverlapSphere(agent.transform.
            position, radius.value, targetMask.value);

        if (targetsInProximity.Length > 0) {
        ...
        for (int i = 0; i < targetsInProximity.Length; i++) {
        ...
        // if that something is inside viewing angle
        if (Vector3.Angle(agent.transform.forward, directionToTarget) < viewAngle.
            value / 2f) {
        if (!RayBlockedByObstacle(directionToTarget, targetDistance)) {
            ...
                target.value = targetsInProximity[i].gameObject;
                blackboard.SetVariableValue("OldTarget", target.value);
        }}}
        if (target.value != null)
                return true;
        else
                return false;
        }
    ...
}}}
```

Odlučio sam prikazati samo dijelove "FindEnemyFieldOfView" skripte relevantne za objašnjavanje funkcionalnosti izrađenog rješenja jer bi u suprotnome skripta bila vrlo duga i nezgrapna za čitanje. Pri samom početku prikazana skripta nasljeđuje od ConditionTask klase NodeCanvas sustava. Time skriptu označujemo kao uvjetni zadatak za stablo ponašanja. Nakon toga naredbom BBParameter (skraćeno od Blackboard parametar) definiramo atribute koje izrađena skripta koristi kako bi mogla donijeti ispravan sud o tome da li je meta vidljiva unutar neprijateljeva vidokruga ili ne. Nakon definiranih atributa unutar OnCheck metode nalazi se logika kojom zadatak utvrđuje ranije spomenuti sud. Logika OnCheck metode najprije dohvaća objekte koji se nalaze u neposrednoj blizini koji se nalaze na targetMask sloju *(eng. layer)* igre, dok smo unutar prikaza grafa stabla kao sloj odabrali sloj na kojem se nalazi igrač. Ako postoje mete u blizini, za svaku metu računamo kut od kuta gledanja neprijatelja te ukoliko se meta nalazi unutar vidokruga neprijatelja putem RayBlockedByObstacle metode još provjeravamo da li je meta sakrivena iza neke prepreke. Ako meta nije sakrivena iza prepreke i nalazi se u vidokrugu neprijatelja OnCheck metoda, čvoru unutar stabla ponašanja unutar kojeg se izvodi, vratiti će istiniti sud, u suprotnom sud će smatrati neistinitim.

```
using NodeCanvas.Framework;
using ParadoxNotion.Design;
using UnityEngine;
using Animancer;

public class AttackCloseTargets : ActionTask<AnimancerComponent> {

    [RequiredField]
    public BBParameter<AnimationClip> animationClip;
    public BBParameter<float> radius;
    public BBParameter<LayerMask> targetMask;

    //This is called once each time the task is enabled.
    //Call EndAction() to mark the action as finished, either in success or failure.
    protected override void OnExecute(){
        Collider[] targetsInProximity = Physics.OverlapSphere(agent.transform.
            position, radius.value, targetMask.value);

        AnimancerState animState = agent.Play(animationClip.value);  // animancer
        animState.NormalizedTime = 0f;
        animState.Events.Add(0.3f, CheckProximityAndDealDamage);
        animState.Events.OnEnd += () => { EndAction(true); };
    }

    public void CheckProximityAndDealDamage() {
        Collider[] targetsInProximity = Physics.OverlapSphere(agent.transform.
            position, radius.value, targetMask.value);
        foreach(Collider c in targetsInProximity) {
        HealthComponent hp = c.gameObject.GetComponent<HealthComponent>();
        if(hp != null) {
            // deal damage
            ...
    }}}
}
```

AttackCloseTargets je skripta zadatka akcije, zato što nasljeđuje ActionTask skriptu. Štoviše u ovom primjeru klasa nasljeđuje od ActionTask<AnimancerComponent>. Ovakvim nasljeđivanjem skripti dajemo do znanja da objekt koji će izvoditi ovu skriptu će kao jednu od svojih komponenti imati pridruženu komponentu AnimancerComponent stoga objekt agenta možemo tretirati kao AnimancerComponent klasu. Ponovo najprije kao i u prijašnjoj skripti definiramo blackboard atribute koje će skripta koristiti, dok njihove vrijednosti pridružujemo unutar prozora za izradu stabla stanja. Dok zadatak uvjeta nakon izvršavanja vraća true ili false, zadatak akcije može se duže izvršavati i vratiti status da još uvijek izvršava zadani zadatak stoga unutar skripte moramo definirati kada određeni zadatak akcije završava. U ovom primjeru dohvaćamo mete u neposrednoj blizini te pomoću naredbe Play() animancer biblioteke pokrećemo animaciju napada. Nakon proteklih trideset posto vremena animacije izvršit će se CheckProximityAndDealDamage naredba koja će ozlijediti mete koje se nalaze u neposrednoj blizini neprijatelja. Na kraju same animacije dodajemo poziv EndAction(true) koji označuje da je zadatak akcije uspješno izvršen i da graf može dalje nastaviti sa zadacima koji slijede.

3.4.3.3. Animancer

Cilj animancer proširenja je smanjiti stupanj apstrakcije koji je prisutan kada koristimo mecanim animacijski sustav ugrađen u Unity alat. Prednosti koje ostvarujemo korištenjem animancer proširenja najbolje tumači sljedeća tablica.

Tablica 1: Problemi Unity mecanim animacijskog sustava i prednosti animancer rješenja

Mecanim	Animancer
Kako bi objekt animirali unutar igre najprije moramo nadodati animator komponentu, izraditi AnimatorController resurs te unutar posebnog animator prozora dodati i povezati željene animacije.	Preko AnimationClip ili TransitionClip strukture, referencu isječka animacije direktno prosljeđujemo do animancer komponente.
Sljedeća velika mana mecanim sustava je otežano testiranje i uočavanje problema budući da mecanim sustav vodi brigu o internim vrijednostima i stanjima u kojima se nalazi.	Animancer komponenta radi što joj se naredi preko koda, ukoliko se dogodi greška korisnik dobiva status greške *(eng. error)* kojeg može slijediti do dijela koda koji ga je prouzročio.
Logika animacije jednog lika unutar igre mora biti kompletno definirana unutar jednog Animator Controller resursa.	Korištenjem AnimationClip i TransitionClip struktura unutar izrađenih skripata, sami organiziramo u kojim skriptama će se koristiti određene animacije.
Upravlja svojim unutarnjim stanjem stoga ignorira ili odgodi zahtjeve izvršavanja novih animacija ukoliko trenutno izvodi neku animaciju sve dok god se ta animacija izvodi.	Prilikom poziva izvršavanja novih animacija prestaje s izvršavanjem stare animacije i započinje izvršavanje nove animacije.

(Izvor: [14])

Nadalje, skripta koja koristi mecanim animator Play("IHopeThereIsAnimation") naredbu, očekuje da animator komponenta unutar sebe sadrži definiranu animaciju naziva IHopeThereIsAnimation koju želi izvršiti. Ukoliko animator komponenta unutar sebe ne sadrži animaciju IHopeThereIsAnimation, prešutno će nastaviti svoj rad i ignorirati traženu play naredbu skripte. Također Animator komponenta može definirati mnoštvo animacija, no izvršavanje animacija ovisi o skripti koja poziva play naredbu jer animacije se izvršavaju samo pozivom animator.Play() naredbe. Korištenjem animancer proširenja smanjujemo vanjske ovisnosti skripte prema animator resursu. Komponenti dodajemo referencu na Animancer komponentu kao i reference na animacije koje želimo izvoditi pomoću izrađene skripte. Ako je referenca na animaciju prazna prilikom izvršavanja dobiti ćemo status greške koja objašnjava kako je animacija koju smo željeli koristiti prazna.

Animancer proširenje zapravo koristimo kako bi izbjegli korištenje mecanim animacijskog sustava unutar Unity alata zbog ranije spomenutih prednosti. Dodatna prednost koja dolazi korištenjem animancer proširenja jest ponovna iskoristivost koda, dok resurs mecanim AnimatorController ponovno možemo iskoristiti samo ako objekt igre koristi identične animacije i uvjete tranzicije između njih.

```
public class WeaponAnimationController : MonoBehaviour
{
    public AnimancerComponent animancer;
    public ClipTransition shootClip;
    public ClipTransition reloadClip;
    ...

    public void PlayAnimation(ClipTransition clip) {
        animancer.Play(clip);
    }
    ...
}
```

WeaponAnimationController skriptom vrlo elegantno izvršavamo razne animacije poput animacije ispaljivanja metka iz pištolja ili animacije dodavanja nove municije u spremnik pištolja. Za svaku potrebnu animaciju dodajemo ClipTransition atribut te putem inspektor prozora u taj atribut proslijedimo odgovarajući isječak animacije.

Kako bi mogućnost jednostavnog animiranja animancer proširenjem koristio unutar NodeCanvas proširenja, izradio sam i jednostavnu skriptu PlayAnimancerAnimation.

```
using NodeCanvas.Framework;
using ParadoxNotion.Design;
using UnityEngine;
using Animancer;

[Category("Animation")]
public class PlayAnimancerAnimation : ActionTask<AnimancerComponent> {

    [RequiredField]
    public BBParameter<AnimationClip> animationClip;
    [SliderField(0, 1)]
    public float crossFadeTime = 0.1f;
```

```
    public bool waitActionFinish = true;
    ...

    protected override void OnExecute() {
        AnimancerState animState = agent.Play(animationClip.value, crossFadeTime);
        animState.Time = 0;

        if (!waitActionFinish) {
            EndAction(true);
        } else {
            animState.Events.OnEnd = NotifyForAnimationEnd;
        }
    }
    private void NotifyForAnimationEnd() {
        EndAction(true);
    }
}
```

3.4.3.4. DoTween

DoTween proširenje dodaje nove naredbe i funkcionalnosti unutar Unity alata pomoću kojih, kroz zadani interval, možemo postepeno mijenjati vrijednosti odabranih atributa. Unutar projekta koristio sam besplatnu inačicu DoTween proširenja kako bih unutar menija igre dodao jednostavne animacije i tranzicije.

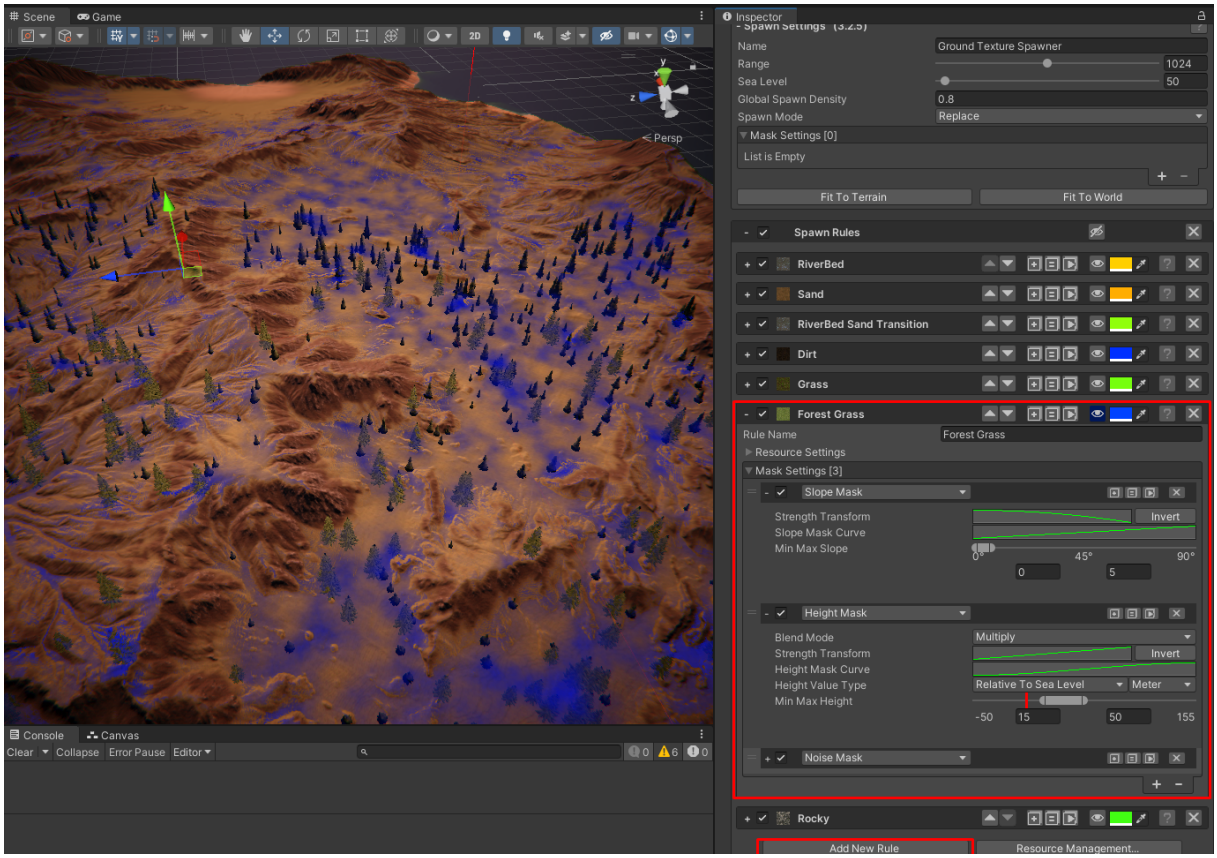
```
using DG.Tweening;
public class MainMenuManager : MonoBehaviour {
    ...

public void OpenView(Transform view, float animationExecutionTime = .2f, float
    initialDelay = .1f) {
        DOTween.Kill("MenuTransitionAnimation");

        foreach(CanvasGroup cg in viewsCG) {
            cg.DOFade(0f, animationExecutionTime).SetUpdate(true);
            cg.blocksRaycasts = false;
            cg.interactable = false;
        }

        CanvasGroup viewCG = view.GetComponent<CanvasGroup>();

        viewCG.DOFade(1f, animationExecutionTime).SetUpdate(true).SetDelay(
            initialDelay)
            .OnComplete(()=> { viewCG.blocksRaycasts = true; viewCG.interactable =
                true; })
            .SetId("MenuTransitionAnimation");
    }
}
```

Prikazana open view metoda koristi se prilikom pritiska na tipku unutar početnog izbornika igre. DoFade naredbom nad CanvasGroup elementima izvršavamo animaciju nestanka

pogleda, odnosno kroz definirani vremenski interval "animationExecutionTime" postepeno mjenjamo vidljivost svih pogleda na nula. Sljedećom DoFade naredbom postepeno povećavamo vidljivost odabranog ekrana na koji pritisnuta tipka vodi, a kada je pogled do kraja vidljiv, OnComplete naredba omogući interakciju s novo otvorenim pogledom. Samoj naredbi na kraju dajemo ime pomoću kojeg je možemo ranije uništiti ukoliko poželimo dodati nove tranzicije budući da DoTween proširenje svoje naredbe izvršava u zasebnim dretvama kako ne bi prekidao glavni tok igre. Iste tranzicije mogli smo ostvariti korištenjem SmoothDamp funkcije unutar Update() životnog ciklusa klase, no korištenjem DoTween proširenja pojednostavili smo proces izrade jednostavnih animacija i dobili veću kontrolu nad njima.

3.4.3.5. Gaia 2021

Gaia 2021 plaćeno je proširenje Unity alata koje omogućuje izradu terena i okoliša pomoću unaprijed definiranih pravila. Samo proširenje koristio sam za izradu terena, bojanje terena određenim teksturama pijeska, zemlje, trave i kamena te za postavljanje drveća i vlasi trave na izrađen teren. Prilikom izrade nivoa , otvara se prozor Gaia Manager kroz koji odabiremo nekolicinu parametara: veličinu nivoa, razlučivost i detaljnost terena u igri, gustoću objekata stavljenih unutar scene, ekosustav koji želimo koristiti *(eng. biome)* te želimo li nasumično generiran teren ili ćemo teren oblikovati sami korištenjem žigova *(eng. stamp)* za teren. Ja sam kao opcije odabrao mali nivo, bez ekosustava te sam odabrao postavku izrade terena korištenjem žigova jer time imam veću kontrolu nad samim izgledom nivoa.



Slika 20: Prikaz inspektor opcija za spawner komponentu gaia 2021 proširenja

Odabrao sam jednostavan žig i primijenio ga na području čitavog terena i time nad terenom oblikovao planine, nizine i ostala udubljenja, no teren je još uvijek bio nebojan. Kako bih obojio teren i na njega postavio drveće i ostale objekte odlučio sam izraditi svoj ekosustav koji će gaia koristiti pri izradi mojeg terena. Konkretan ekosustav koji sam izradio sastoji se od nekoliko Spawner komponenti koje pri primjenjivanju ekosustava odrađuju određene zadatake. GroundTextureSpawner komponenta definira pravila bojanja terena teksturama pijeska, zemlje, trave i kamenja. TreeSpawner izrađena komponenta na travnata i šumska područja dodaje drveće , dok GrassSpawner na ista ta područja dodaje vlasi trave. Na slici iznad prikazan je GroundTextureSpawner sa svim definiranim pravilima. S prikazane slike vidimo kako ova spawner komponenta ima 7 definiranih pravila za bojanje terena: "RiverBed", "Sand", "RiverBedSand Transition", "Dirt", "Grass", "ForestGrass" i "Rocky". Također ForestGrass pravilo prošireno je unutar izbornika te vidimo da je samo pravilo iako se izvodi na području cijelog terena, ograničeno triju maskama: maskom nagiba *(eng. slope mask)*, visinskom maskom *(eng. height mask)* i maskom šum-a *(eng. noise mask)*. Maskom nagiba ograničavamo bojanje terena samo u područjima koji su pod nagibom od nula do pet posto. Visinskom maskom određujemo da se forest grass tekstura primjenjuje samo na područjima od petnaest do pedeset metara nadmorske visine. Te na kraju noise maskom zabranjujemo postavljanje teksture na nasumično odabranim mjestima kako bi obojeni teren dobio prirodniji dojam. Teren koji će se obojati šumskom travom na slici je vidljiv plavom bojom. Ako teren želimo obojati novom teksturom, klikom na "Add New Rule" gumb, dodajemo novo pravilo gdje definiramo teksturu koju želimo koristiti te maske koje dodatno limitiraju gdje će se ta tekstura unutar terena postaviti. Isti proces odradio sam za TreeSpawner i GrassSpawner samo umjesto tekstura pravilu sam pridružio objekte koji želim stvoriti unutar terena. Nakon što sam izradio ekosustav s pridruženim mu spawner komponentama, isti sam i spremio kako ubuduće kada ću izrađivati novi nivo mogu u nekoliko klikova generirati novi nivo korištenjem pravila definiranih unutar tog ekosustava koji primjenjujem nad proizvoljno oblikovanim terenom. Ponovna iskoristivost pravila generiranja terena jedna je od prednosti gaia alata. Teren i čitav nivo mogao sam izraditi i bez gaia proširenja, no oblikovanje, bojanje terena pa i postavljanje drveća i vlasi trave morao bih ručno odraditi, a kada bih poželio dodati novi nivo ponovo bi cijeli proces izrade nivo-a morao odraditi od početka na isti način.

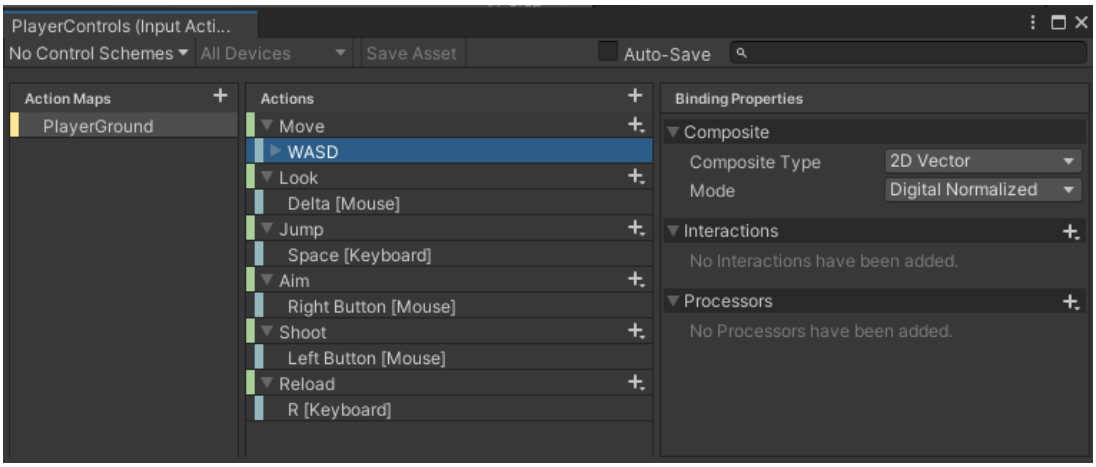
Iako sam gaia proširenje koristio samo za generiranje terena nivoa, ono posjeduje još mnoštvo drugih mogućnosti poput generiranja vodenih površina, generiranje vjetra, dodavanja ugođaja jutra, zalaska sunca ili noći i slično. Najveća mana koju sam primijetio korištenjem ovog proširenja je ta da proširenje zauzima mnogo prostora na disku i time povećava vrijeme otvaranja projekta. Također upravo zbog toga što proširenje zauzima puno prostora , onemogućuje prenos projekta na GitHub server stoga sam sve promjene nad igrom nakon dodavanja ovog proširenja, imao samo lokalno korištenjem Git sustava verzioniranja. Iako smatram da je gaia proširenje izuzetno kvalitetno izrađeno, mislim da bih bio barem nekoliko puta brži da nisam koristio ovo rješenje budući da izrađeni nivo igre malih dimenzija.

3.4.4. Preostale implementirane funkcionalnosti

Kroz ovo poglavlje opisat ću skripte koje su od velike važnosti za izrađenu igru budući da se glavne mehanike igrača razvijaju baš implementacijom ovdje prikazanih skripti. Kako bi igrač mogao obavljati radnje unutar igre unutar Unity alata moramo definirati radnje koje igrač može obavljati pritiskom na određene tipke perifernih uređaja, a o tome više govori nadolazeće poglavlje.

3.4.4.1. Unity sustav za očitavanje korisničkog unosa

Igre koje se izrađuju unutar Unity alata imaju mogućnost odabira između dva sustava za detekciju korisničkog unosa (u nastavku skraćeno input sustav). Stariji input sustav podosta je limitiran pogotovo kada bi željeli mogućnost da igra podržava razne periferne uređaje poput upravljača Xbox konzole što danas nije rijetka pojava. Zbog toga igra prezentirana u ovom radu koristi novi input sustav koji zahtjeva par dodatnih koraka, no podržava razne periferne uređaje i uistinu je modularan. Kako bi koristili novi input sustav najprije u projektnom prozoru moramo kreirati resurs naziva "InputActions". Dvostrukim klikom na novo izrađeni resurs otvaramo prozor unutar kojeg definiramo koje ulazne akcije želimo motriti kroz novi input sustav.

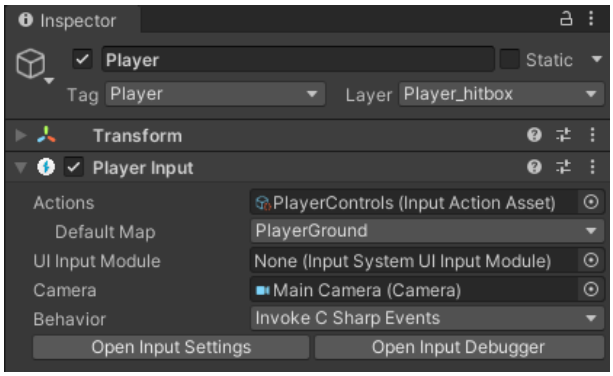


Slika 21: Prikaz prozora unutar Input Actions Unity resursa

Slika prikazuje prozor s već definiranim akcijama koje se izvršavaju kada korisnik pritisne neku od definiranih veza *(eng. binding)* s perifernim uređajima. U ovom primjeru akcija Jump koristi "Space [keyboard]" vezu na periferni uređaj tipkovnice odnosno Jump akcija izvršava se prilikom svakog pritiska tipke space na tipkovnici. Novi input sustav također ima dodatne opcije poput procesiranja i promjene detektiranog unosa dodavanjem interakcija i procesora informacija, no ni jednu od tih dodatnih opcija nisam imao potrebe koristiti. Kako bi Input Actions resurs koristili unutar igre objektu u sceni potrebno je još dodati Player Input komponentu.

Prije samog komentiranja koda kretanja igrača važno je prokomentirati odabranu "Behavour" opciju Player Input komponente. Naime Player Input komponenta definira nekoliko različitih načina komuniciranja detektiranog unosa igrača. U ovom projektu odlučio sam unos igrača ostalim skriptama oglasiti korištenjem C Sharp Events načina koji prilikom novog de-

tektiranog korisničkog unosa oglašava događaj *(eng. event)* pomoću kojeg obavještava ostale skripte o pritisnutom unosu korisnika.



Slika 22: Player input komponenta pridružena objektu igrača unutar scene

3.4.4.2. Skripta upravljanja kretanja igrača

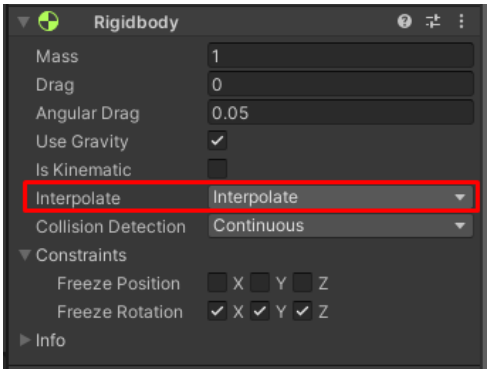
Skripta za upravljanje kretanja igrača vrlo je važan element igre stoga ću je ovdje rastaviti na dva dijela kako bih tu podužu skriptu mogao opisati u cijelosti. Prva nadolazeća skripta opisat će kretnje smjera igrača , dok skripta nakon nje opisuje funkcionalnost skoka igrača unutar igre. Kod kretanja i skoka igrača, uz male preinake, izrađen je po uzoru na [15].

```
[RequireComponent(typeof(Rigidbody))]
public class MyPlayerMovement : MonoBehaviour {
    [Header("Movement")]
    public float moveSpeed;
    private PlayerInput playerInput;
    public Transform orientation;

    [ReadOnly]
    [SerializeField]
    private Vector2 moveInputVector;
    Vector3 moveDirection;
    Rigidbody rb;
    ...
    private void Awake() {
        rb = GetComponent<Rigidbody>();
        rb.freezeRotation = true;
        playerInput = GetComponent<PlayerInput>();
    }
    private void OnEnable() {
        playerInput.onActionTriggered += CallAppropriateCallback;
    }
    private void OnDisable() {
        playerInput.onActionTriggered -= CallAppropriateCallback;
    }
    private void CallAppropriateCallback(InputAction.CallbackContext context) {
        if (context.action == playerInput.actions["Move"]) {
            OnMove(context);
        }
```

```
        if (context.action == playerInput.actions["Jump"]) {
            OnJump();
    }}

    private void OnMove(InputAction.CallbackContext context) {  // called by
        PlayerInput
        moveInputVector = context.ReadValue<Vector2>();
    }

    private void FixedUpdate() {
        MovePlayer();
    }

    private void MovePlayer() {
        moveDirection = orientation.forward * moveInputVector.y + orientation.right
            * moveInputVector.x;
        if (isGrounded) {  // on ground
            rb.AddForce(moveDirection.normalized * moveSpeed * 10f, ForceMode.Force)
                ;
        }
        else if (!isGrounded) {  // in air
            rb.AddForce(moveDirection.normalized * moveSpeed * 10f * airMultiplier,
                ForceMode.Force);
}}}
```

RequireComponent naredbom naznačujemo da skripta na game objektu još zahtjeva i Rigidbody komponentu. Ukoliko prilikom dodavanja izrađene skripte objekt ne sadrži Rigidbody komponentu, ona će se automatski dodati. Nakon toga definiramo variable potrebne za izvršavanje MyPlayerMovement skripte. Awake metoda izvršava se automatski prilikom prvog učitavanja skripte. Unutar Awake metode pomoću GetComponent<Rigidbody> parametrizirane naredbe dohvaćamo Rigidbody komponentu game objekta na kojem se nalazi MyPlayerMovement komponenta. Ukoliko GetComponent<> naredba ne pronađe traženu komponentu, umjesto reference tražene komponente, naredba bi vratila null vrijednost, no u ovom slučaju to se ne može dogoditi jer prvom naredbom ReqireComponent garantiramo da će objekt kojem se ova skripta pridruži imati i pridruženu rigidbody komponentu. Na isti način unutar awake funkcije dohvaćamo ostale vrijednosti. Svaki puta kada aktiviramo MyPlayerInput komponentu izvršit će se OnEnable metoda unutar koje se skripta upisuje za slušanje novih događaja korisničkog unosa. Kada PlayerInput komponenta oglasi novi događaj, tj. detektira da je korisnik pritisnuo tipku, prikazana skripta pokrenuti će CallAppropriateCallback metodu koja provjerava da li korisnik svojim unosom želi skočiti ili pokretati igrača i zatim će pozvati odgovarajuću metodu. Svaki puta kada deaktiviramo MyPlayerMovement komponentu na nekom game objektu, pozvat će se OnDisable naredba unutar koje prekidamo slušanje novih player input događaja korisničkog unosa pošto je komponenta postala neaktivna. Korisnički unos bilježi se unutar Update petlje kako bi odmah po pritisku tipke mogli reagirati na događaj. Kao što smo ranije napomenuli simulaciju fizike odrađuje Unity sistem fizike i to unutar FixedUpdate petlje koja se odvija u pravilnim vremenskim intervalima neovisno od izvršavanja Update petlje. Zbog toga što korisnički unos pratimo unutar Update petlje, a rigidbody tijelo pokreće sustav fizike unutar FixedUpdate petlje, važno je istaknuti kako OnMove naredba ne pomiče igrača već samo

sprema smjer kretanja unutar moveInputVector dvodimenzionalnog vektora, dok se samo kretanje igrača događa unutar FixedUpdate metode koja spremljene očitane vrijednosti smjera koristi kako bi dodala silu koja djeluje na Rigidbody komponentu igrača. Drugim riječima korištenjem moveInputVector variable ogradili smo logiku čitanja korisničkog unosa Update petlje od logike FixedUpdate petlje koja u pravilnim vremenskim intervalima izvršava fizičke izračune nad rigidbody komponentom i pokreće ga.



Slika 23: Postavke rigidbody komponente nad Player game objektom

Također kako bi primijenjene kretnje igrača unutar FixedUpdate petlje uskladili sa vizualnim osvježavanjem ekrana koje se događa unutar Update petlje, unutar Rigidbody komponente moramo uključiti opciju interpolacije. Učinjene promjene nad Rigidbody komponentom možemo vidjeti na slici 23.

Skriptu kretanja mogli smo implementirati i korištenjem CharacterController komponente. Budući da ta komponenta nije dio sustava fizike ne bismo trebali koristiti FixedUpdate petlju već bi čitanje korisničkog unosa i potrebne kretnje odradili unutar Update petlje direktno mijenjajući poziciju Transform komponente game objekta kojemu skripta pripada, no tada na igrača ne bi djelovala sila gravitacije, niti bi ga neprijatelji mogli odgurnuti ili blokirati - osim naravno ukoliko sve te funkcionalnosti ne bi dodatno implementirali popratnim Unity skriptama.

```
[RequireComponent(typeof(Rigidbody))]
public class MyPlayerMovement : MonoBehaviour {
    ...
    [Header("Ground Check")]
    public float playerHeight;
    public LayerMask whatIsGround;
    [ReadOnly]
    [SerializeField]
    private bool isGrounded;
    public float groundDrag;

    [Header("Jump")]
    public float jumpForce;
    public float jumpCooldown;
    ...

    private void Update() {
        isGrounded = Physics.Raycast(transform.position, Vector3.down, playerHeight
            * 0.5f + 0.2f, whatIsGround);
```

```
        if (isGrounded)
            rb.drag = groundDrag;
        else
            rb.drag = 0;
    }

    private void CallAppropriateCallback(InputAction.CallbackContext context) {
        ...
        if (context.action == playerInput.actions["Jump"]) {
            OnJump();
    }}

    private void OnJump() {
        if(readyToJump && isGrounded) {
            readyToJump = false;
            Jump();

            Invoke(nameof(ResetJump), jumpCooldown);
    }}

    private void Jump() {
        // reset y velocity
        rb.velocity = new Vector3(rb.velocity.x, 0f, rb.velocity.z);
        rb.AddForce(transform.up * jumpForce, ForceMode.Impulse);
    }

    private void ResetJump() {
        readyToJump = true;
    }
}
```

Ovdje prikazani nastavak MyPlayerMovement skripte implementira funkcionalnost skoka igrača unutar igre kako bi igraču omogućili preskakanje prepreka i bolje manevriranje u prostoru. Za razliku od koda kretanja gdje smo jasno odvojili čitanje korisničkog unosa unutar Update te pomicanje Rigidbody komponente unutar FixedUpdate petlje, OnJump metodu skoka igrača pozivamo istovremeno s korisnikovim pritiskom tipke. No u ovom slučaju to je u redu budući da kao silu na Rigidbody objekt dodajemo silu tipa ForceMode.Impulse čime označavamo da se sila nad objektom primjenjuje instantno, a ne postepeno kroz korake fizičke simulacije. Samu mehaniku skakanja igrača dodatno smo ograničili uvjetom readyToJump kako bi nakon skoka igraču trebao predah prije nego ponovno može skočiti i isGrounded kako bi mogućnost skakanja igrač imao samo ako je prizemljen na tlu.

U ovoj skripti još bih nakratko skrenuo pažnju na Physics.Raycast() metodu. Pomoću raycast metode unutar igre stvaramo virtualnu zraku koja se proteže zadanim smjerom pravca sve dok se ne sudari s game objektom koji sadrži Collider ili Rigidbody komponentu. U ovom primjeru Raycast metodom stvaramo virtualnu zraku počevši od sredine igračeve trenutne pozicije u smjeru Vector3.down - odnosno prema dolje. Ako zamislimo da se transform komponenta nalazi u sredini game objekta igrača, polovicu visine igrača zraka mora proći da dospije do dna igračevih stopala, a ostalih 0.2f je dužina zrake koja izvire iz igračeva stopala prema dolje. Tu

virtualnu zraku koristimo kako bi u svakom koraku Update petlje mogli odrediti nalazi li se igrač na tlu ili je u zraku i pada prema tlu, ovisno o tome siječe li zraka teren ili ne. Ako je igrač prizemljen, rb Rigidbody komponenti dodajemo groundDrag odnosno silu trenja koja usporava kretnju igrača. Ukoliko je igrač u zraku, tada sile trenja nema.

3.4.4.3. Kamera

Skriptu kamere kao komponentu smo postavili na zaseban game objekt od game objekta igrača. Kad bi skripta kamere i skripta kretanja igrača obje bile na istom objektu, pojavljivale bi se razne greške poput nasumičnog trzanja kamere i slično. Ti artefakti se događaju zato što Unity ne daje prednost izvođenju jedne komponente ispred druge, već životne cikluse komponenti izvodi redom kako završi s prijašnjim poslom.

```
public class MoveCamera : MonoBehaviour
{
    [Header("PlayerInput")]
    public PlayerInput playerInput;

    public float sensX;
    public float sensY;
    public Transform orientation;
    public Transform cameraPositionOnPlayer;
    public Transform cameraTransform;
    float xRotation;
    float yRotation;

    private void Start() {
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
    }

    private void Update() {
        Vector2 mouse = playerInput.actions["Look"].ReadValue<Vector2>(); // get
            mouse input
        mouse = new Vector2(mouse.x * Time.deltaTime * sensX, mouse.y * Time.
            deltaTime * sensY);

        yRotation += mouse.x;
        xRotation -= mouse.y;
        xRotation = Mathf.Clamp(xRotation, -70f, 70f);

        transform.rotation = Quaternion.Euler(xRotation, yRotation, 0);  // rotate
            cam holder
        orientation.rotation = Quaternion.Euler(0, yRotation, 0);  // update player
            orientation
    }

    private void LateUpdate() {
        cameraTransform.position = cameraPositionOnPlayer.position;
    }
}
```

Unutar Start() metode zaključavamo i sakrivamo pokazivač miša budući da unutar igre koristimo nacrtani ciljnik *(eng. crosshair)*. Unutar Update() metode konstantno provjeravamo promjenu pozicije miša. Ovdje ne trebamo pratiti događaj pomaka miša , već pomak miša očitavamo svakim prolaskom kroz Update petlju budući da Input Actions resurs, vrijednost pomaka miša prebrisuje upravo nakon svakog koraka Update naredbe. Zabilježeni pomak miša množimo s odgovarajućom vrijednosti senzitivnosti i spremamo u "mouse" dvodimenzionalni vektor. X koordinata mouse vektora označava horizontalni pomak miša, dok Y koordinata označava vertikalni pomak miša. Zabilježeni horizontalni pomak dodajemo na rotaciju oko Y osi, dok vertikalni pomak oduzimamo od X rotacijske osi. Na kraju vertikalni vidokrug limitiramo na 70 stupnjeva gore i 70 stupnjeva dolje od prednje strane objekta. Korištenjem Quaternion.Euler() primjenjujemo izračunatu rotaciju na rotaciju game objekta skripte , ali i na rotaciju modela igrača unutar igre. Kako se igrač u bilo kojem trenutku može pomaknuti, a redoslijed Update naredbi nije unaprijed određen , nemoguće je osvježiti poziciju kamere i biti siguran da se igrač prvo pomaknuo i nova pozicija kamere je nova pozicija igrača ili je kamera očitala staru vrijednost igrača nakon koje se je igrač pomaknuo. Kako bi izbjegli ovaj nepovoljan scenarij koristimo LateUpdate metodu. LateUpdate metoda počet će se izvršavati tek nakon što se izvrše sve Update naredbe pa time možemo biti sigurni da je nova pozicija kamere na poziciji igrača nakon njegova pomaka.

3.4.4.4. Mehanika pucanja

Sljedećom skriptom igraču dodajemo mehaniku pucanja unutar igre. Kada igrač potroši svu municiju unutar pištolja više ne može pucati, no novu municiju može nadopuniti sakupljanjem nasumično stvorene municije unutar izgrađenog nivoa.

```
public class ShootFunctionality : MonoBehaviour {
    ...

    private void Awake() {
        cameraTransform = Camera.main.transform;
        wac = GetComponent<WeaponAnimationController>();
        weaponAudio = GetComponent<AudioPlayer>();
        ammoContainer = GetComponent<AmmoContainer>();
    }

    private void OnEnable() {
        playerInput.onActionTriggered += OnShoot;
        playerInput.onActionTriggered += ManualReload;
    }
    ...

    private void OnShoot(InputAction.CallbackContext context) {  // called by
        PlayerInput component (bcz it's set to Behaviour: SendMessages)
        if (context.action != playerInput.actions["Shoot"]) return;
        if (!GameManager.Instance.isPlayerAlive) return;
        if ( ammoContainer.isAmmoEmpty()) {
            ...
            return;
```

```
        }
        if (((lastShootTime + shootDelay) > Time.time))  return;

        GameObject bullet = Instantiate(bulletPrefab, bulletSpawnPoint.position,
            Quaternion.identity, bulletParent);
        weaponAudio.PlayAudio(SoundBank.Instance.pistolShot);

        PreciseProjectile projectile = bullet.GetComponent<PreciseProjectile>();  //
             assumes each bullet has PreciseProjectile component , can also do with
            for example bullet interface ?

        Vector3 bulletDirection;
        if (Physics.Raycast(cameraTransform.position, cameraTransform.forward, out
            RaycastHit hit, Mathf.Infinity))
            bulletDirection = hit.point - bullet.transform.position;
        else
            bulletDirection = (cameraTransform.position + cameraTransform.forward *
                projectile.maxTravelDistance) - bullet.transform.position;

        bullet.transform.rotation = Quaternion.LookRotation(bulletDirection);
        projectile.BulletVelocity = bulletSpeed;
        ammoContainer.ShootBullet(1);
        wac.PlayAnimation(wac.shootClip);
        lastShootTime = Time.time;
}}
```

Na isti način kao i u prijašnjim skriptama unutar Awake metode tražimo reference na komponente čije metode koristimo u skripti. OnEnable i OnDisable metodama kao i u ranijim skriptama pratimo događaje korisničkog unosa te ovisno o korisničkom unosu izvršavamo OnShoot ili ManualReload metodu. ManualReload metoda koristi se za promjenu praznog šaržer-a pištolja stoga nam implementacija te metode nije važna pri objašnjavanju mehanike pucanja. Kompletna mehanika pucanja izvršava se unutar OnShoot metode koja kao argument prima InputAction.CallbackContext budući da input sustav za pritisnutu tipku tipkovnice ili miša koristi taj događaj. Zbog toga najprije provjerimo je li korisnik uistinu pritisnuo tipku miša ili možda neku drugu tipku. Nakon toga provjeravamo da li je igrač živ. Zatim da li je spremnik pištolja prazan. Konačno na kraju provjeravamo još da li je prošao dovoljan odmak vremena od prošlog ispaljivanja metka. Ovaj zadnji uvjet koristimo kako igrač ne bi ispaljivao metke jedan iza drugoga bez da se izvrši kompletna izrađena animacija pucanja iz pištolja. Ako su sve provjere zadovoljene pozivom Instantiate() naredbe, unutar scene, na kraju cijevi pištolja stvorimo prefab metka. Izrađena struktura prefab-a metka unutar sebe sadrži i PreciseProjectile komponentu koju dohvaćamo pomoću GetComponent naredbe. Pomoću Physics.Raycast naredbe sa sredine ekrana, odnosno sredine kamere bacamo virtualnu zraku u smjeru nišana. Ako se pod nišanom nalazi neki objekt, tu točku sudara koristimo kako bi definirali smjer kojim novo stvoreni metak putuje, u suprotnom ukoliko u dometu virtualne zrake nismo sudarili niti jedan objekt, završetak zrake koristimo kao destinaciju putanje stvorenog metka. Nakon što smo ispalili metak oduzimamo jedan metak iz spremnika , pokrećemo animaciju pucanja pištolja i osvježujemo vrijeme ispaljivanja zadnjeg metka. Kretanje samog prefab metka odrađuje učahurena PreciseProjectile komponenta budući da smo metku zadali smjer i brzinu kretanja.

3.4.4.5. Dinamičko stvaranje neprijatelja prilikom izvođenja igre

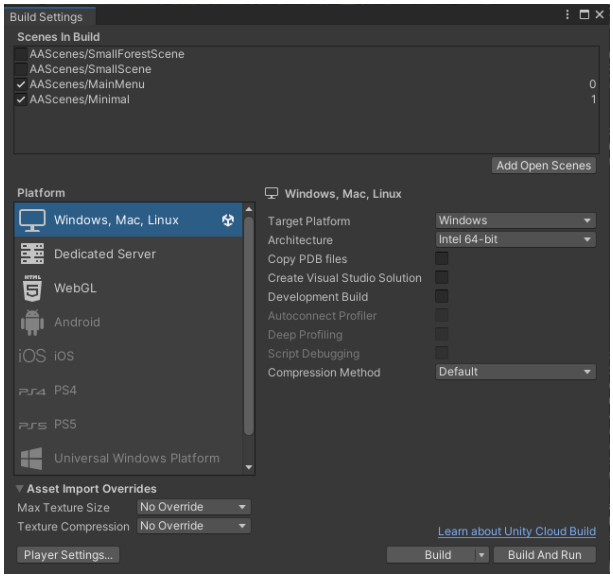
Kako bi igra imala osjećaj nepredvidljivosti, neprijatelje unutar izgrađenog nivoa postavljamo nasumično od početka izvršavanja igre i nadomještamo kada igrač ubije neprijatelja kako bi nivo ostao ispunjen istim brojem neprijatelja.

```
public class ObjectSpawner : MonoBehaviour {
    ...
    [SerializeField]
    private List<GameObject> spawnPrefabs;
    public List<GameObject> spawnedInstances = new List<GameObject>();

    private void Update() {
        if (oldSavedTime + spawnCooldownTime < Time.time) {
            oldSavedTime = Time.time;

            if (spawnedInstances.Count < maxSpawnedObjects) {
                GameObject obj = SpawnObject();
                if (obj != null)
                    spawnedInstances.Add(obj);
    }}}

    private GameObject SpawnObject() {
        float randomX = Random.Range(-xDeviation, xDeviation);
        float randomZ = Random.Range(-zDeviation, zDeviation);
        GameObject objToSpawn = spawnPrefabs[Random.Range(0, spawnPrefabs.Count)];
        Vector3 maybeSpawnPos = transform.position + new Vector3(randomX, 0, randomZ
            );
        maybeSpawnPos.y = spawnerHeight;

        RaycastHit hit;
        Physics.Raycast(maybeSpawnPos, Vector3.down, out hit, maxRayDistance,
            collideMask);
        Vector3 spawnPos = hit.point + spawnOffset;

        if (hit.collider != null)
            return Instantiate(objToSpawn, spawnPos, Quaternion.identity, this.
                gameObject.transform);
    }
}
```

U samoj suštini ova skripta prolaskom vremena unutar igre u definiranim vremenskim intervalima poziva SpawnObject metodu. SpawnObject metoda najprije odabere nasumične vrijednosti devijacije po X i Z osi. Nasumično iz liste objekata, odnosno prefab-a, odabere jedan objekt te iz zraka baci virtualnu zraku prema terenu ispod. U ovom slučaju, bačena virtualna zraka može se sudariti samo s game objektima koji se nalaze na collideMask sloju kako se neprijatelj ili municija ne bi mogli stvoriti iznad glave igrača ili iznad drveća nivoa. Ukoliko se zraka sudari s terenom na mjestu sudara virtualne zrake i terena stvara se nasumično odabrani prefab. Ovu jednostavnu skriptu koristim kako bih na nepravilnom terenu prilikom izvršavanja igre mogao stvoriti municiju i neprijatelje.

3.4.5. Izgradnja Unity aplikacije

Iako izrađenu igru možemo igrati unutar Unity alata, izrađena igra još uvijek je samo projekt, odnosno trenutno je možemo drugima poslati samo kao Unity projekt te da bi je druge osobe mogle isprobati također bi morale preuzeti i instalirati Unity alat sa svim proširenjima koje smo i sami koristili. Kako bi izrađenu igru s drugima podijelili kao zapakiranu aplikaciju, igru moramo izgraditi *(eng. build)*, odnosno zapakirati kao Unity aplikaciju. Srećom proces izgradnje aplikacije vrlo je jednostavan te klikom i provjerom na nekoliko opcija možemo započeti s izgradnjom aplikacije.



Slika 24: Postavke izgradnje čitavog projekta

Slika iznad prikazuje korištene postavke za izgradnju igre za windows platformu na kojoj je ovaj projekt i izrađen. Kao što smo ranije napomenuli, za ovu igru izradili smo dvije scene, prva scena naziva "MainMenu" prikazuje početni izbornik unutar kojeg igrač pokreće ili izlazi iz igre, dok je scena "Minimal" scena glavnog izrađenog nivoa igre. Obratimo pažnju na poredak scena. Budući da želimo da igrača pri ulasku u igru dočeka pogled na početni izbornik igre MainMenu sceni pridružujemo izgradni indeks *(eng. build index)* 0 , dok glavnoj sceni igre "Minimal" pridružujemo indeks 1. Klikom na build započinjemo izgradnju Unity aplikacije.

Tijekom procesa izgradnje, Unity sustav za izgradnju, u aplikaciju će uključiti sve korištene resurse, no neće uključiti alate i proširenja koje smo koristili za izradu igre. U mojem slučaju igra kao Unity projekt zauzima dvadeset tri gigabajta memorije, dok izgrađena Unity aplikacija zauzima skromnih dvjesto megabajta memorije. Krajnji korisnici odnosno igrači igre, igru će moći pokrenuti bez posjedovanja Unity alata jednim klikom na izrađenu Unity aplikaciju.

3.4.6. Finalni izgled igre

Slika 25: Prikaz korisničkog pogleda unutar izrađene igre

Kao što je prikazano na slici 25. dolje s lijeve strane igrač u svakom trenutku može vidjeti svoje životne bodove i broj metaka u pištolju. U sredini ekrana je ciljnik koji igraču vizualno prikazuje smjer ispucavanja metka iz pištolja. Kako bi se igrač kretao kroz prostor koristi WASD tipke. Pomoću space tipke skače, a klikom na miš puca iz pištolja. Ukoliko municija pištolja nije skroz puna pritiskom na tipku R igrač može dodati municiju u pištolj, a za vrijeme dodavanja municije ne može pucati iz pištolja.

4. Zaključak

Kroz izradu ovog rada unutar Blender i Unity alata spoznao sam da je izrada video igre zanimljiv ali i dug proces. Kroz Blender alat upoznao sam se procesom izrade i pripreme resursa za upotrebu untar video igre. Prilikom izrade modela pratio sam kurs profesionalne izrade modela, a Blender kao alat niti u jednom trenutku nije me sputavao pri izradi. Štoviše spoznao sam da je Blender kao alat sposoban podržati profesionalne procese izrade modela s lakoćom. Unity alat proučavao sam gledanjem raznih videa na youtube platformi, čitanja relevantnih blogova te preko službene dokumentacije Unity alata. Pri početku sam se teško snalazio sa svim alatima koje Unity nudi, no ubrzo sam pohvatao osnove i igra je polako poprimala izgled koji sam zamislio. Kako sam izrađivao igru saznao sam za brojna Unity proširenja, od kojih sam neka i sam kupio i koristio unutar projekta. Nakon što sam izradio igru u Unity alatu uvjeren sam kako Unity ima sve potrebne mogućnosti i alate da se njime proizvede profesionalna komercijalna igra. Osim toga Unity alat izvrsno je integrirao C# programski jezik i razvijanje programskih rješenja unutar Unity alata uistinu se odvija s lakoćom, a sve izrađene skripte uredno su strukturirane u komponente ili resurse projekta koje možemo ponovno upotrijebiti na različite nove načine. Na poslijetku bih još samo kostatirao da iako rad opisuje potpuno funkcionalne skripte, smatram da sam najviše naučio na greškama koje sam proizveo prije nego sam došao do finalnog programskog rješenja. Unity kao alat za izradu igra uistinu pruža pregršt mogučnosti te u kombinaciji sa Blender alatom smatram da gotovo ne postoji problem u području video igra koji se tim alatima ne bi mogao riješiti.

Popis literature

[1] K. T. Jensen, *The Complete History Of First-Person Shooters*, https://www.pcmag.com/news/the-complete-history-of-first-person-shooters, Dostupno dana: 02-09-2022, 2017.

[2] A. Jadallah, *The Game Development Pipeline*, https://www.youtube.com/watch?v=hKfV3YtK6Hc, Dostupno dana: 30-08-2022, 2020.

[3] *Screen Skills - Careers in the games industry*, https://www.screenskills.com/media/4379/games-a2-current-inclusivity-web.pdf, Dostupno dana: 30-08-2022, 2022.

[4] *Blender about*, https://www.blender.org/about/, Dostupno dana: 26-08-2022.

[5] G. Abbitt, *Udemy - Blender Character Creator for Video Games Design*, https://www.udemy.com/course/blendercharacters, Dostupno dana: 28-08-2022.

[6] J. Pu, *ArtStation - Game Character Female Elf*, https://www.artstation.com/artwork/4yNK8, Dostupno dana: 28-08-2022.

[7] *Blender documentation*, https://docs.blender.org/manual/en/latest/interface/index.html, Dostupno dana: 26-08-2022.

[8] *Wikipedia - Unity*, https://en.wikipedia.org/wiki/Unity_(game_engine), Dostupno dana: 30-08-2022, 2022.

[9] J. French, *Game dev beginner - How to use Fixed Update in Unity*, https://gamedevbeginner.com/how-to-use-fixed-update-in-unity/, Dostupno dana: 31-08-2022, 2022.

[10] *Unity documentation - MonoBehaviour*, https://docs.unity3d.com/ScriptReference/MonoBehaviour.html, Dostupno dana: 31-08-2022.

[11] R. Hipple, *Unite Austin 2017 - Game Architecture with Scriptable Objects*, https://www.youtube.com/watch?v=raQ3iHhE_Kk, Dostupno dana: 31-08-2022, 2017.

[12] *Github - NaughtyAttributes*, https://github.com/dbrizov/NaughtyAttributes, Dostupno dana: 31-08-2022.

[13] *NodeCanvas - Documentation*, https://nodecanvas.paradoxnotion.com/documentation/, Dostupno dana: 30-08-2022.

[14] *Animancer - Why replace Mecanim?* https://kybernetik.com.au/animancer/docs/introduction/mecanim-vs-animancer/why/, Dostupno dana: 31-08-2022.

[15] Dave, *FIRST PERSON MOVEMENT in 10 MINUTES - Unity Tutorial*, https://www.youtube.com/watch?v=f473C43s8nE, Dostupno dana: 31-08-2022, 2022.

Popis slika

Popis tablica

5. Popis korištenih resursa

5.1. Alati

Unity 2021.3.5f1 Personal - https://unity.com/download dostupno dana 05.09.2022.

Blender 3.2.1 - https://www.blender.org/download/ dostupno dana 05.09.2022.

5.2. Proširenja Unity alata (eng. plugin)

Gaia 2021 - Terrain & Scene Generator - https://assetstore.unity.com/packages/tools/terrain/gaia-2021-terrain-scene-generator-193509 dostupno dana 05.09.2022.

NaughtyAttributes - https://github.com/dbrizov/NaughtyAttributes dostupno dana 05.09.2022.

Animancer Pro - https://assetstore.unity.com/packages/tools/animation/animancer-pro-116514 dostupno dana 05.09.2022.

DoTween - http://dotween.demigiant.com/download.php dostupno dana 05.09.2022.

NodeCanvas - https://assetstore.unity.com/packages/tools/visual-scripting/nodecanvas-14914 dostupno dana 05.09.2022.

5.3. 3D modeli i animacije

Modeli neprijatelja (POLYGON - City Zombies Pack) - https://syntystore.com/products/polygon-city-zombies-pack dostupno dana 05.09.2022.

Model municije (POLYGON - Icons Pack) - https://syntystore.com/products/polygon-icons-pack dostupno dana 05.09.2022.

Animacije neprijatelja: Zombie Idle, Zombie Running, Zombie Attack, Zombie Walk, Zombie Dying - https://www.mixamo.com/ dostupno dana 05.09.2022.

5.4. Slike i teksture

RiverBed Texture -Ground 022 https://ambientcg.com/view?id=Ground022 dostupno dana 05.09.2022.

Sand - Ground 049 A https://ambientcg.com/view?id=Ground049A dostupno dana 05.09.2022.

Dirt - Ground 044 https://ambientcg.com/view?id=Ground044 dostupno dana 05.09.2022.

Grass - Grass 003 https://ambientcg.com/view?id=Grass003 dostupno dana 05.09.2022.

ForestGrass - Grass 004 https://ambientcg.com/view?id=Grass004 dostupno dana 05.09.2022.

Rocky Texture - Rock 033 https://ambientcg.com/view?id=Rock033 dostupno dana 05.09.2022.

HUD HealthUI - https://cdn2.vectorstock.com/i/1000x1000/35/76/green-cross-medical-symbol-vector-20783576.jpg dostupno dana 05.09.2022.

HUD AmmoUI - https://cdn5.vectorstock.com/i/1000x1000/54/64/bullet-ammunition-icon-simple-style-vector-10885464.jpg dostupno dana 05.09.2022.

Bullet Decal - https://opengameart.org/content/bullet-decal dostupno dana 05.09.2022.

crosshair152.png - https://kenney.nl/assets/crosshair-pack dostupno dana 05.09.2022.

5.5. Zvuk

Zvučni efekti (KROTOS - Ammo & Reloads Sound Effects Library):

- M4 Natural Single Shot Mid-001.wav
- M4 Natural Single Shot Mid-002.wav
- M4 Natural Single Shot Mid-003.wav

https://www.krotosaudio.com/products/ammo-reloads-sound-effects-library/ dostupno dana 05.09.2022.

Zvučni efekti (SoundMorph Gore Sound Effects Library):

- Flesh Hit Impact Thud Punch-004.wav
- Flesh Hit Impact Thud Punch-006.wav

https://www.krotosaudio.com/products/gore-sound-effects-library/ dostupno dana 05.09.2022.