Izrada mobilne aplikacije koristeći programski okvir Unity na studijskom slučaju mobilne igre

Sudar, Marko

VELEUČILIŠTE U ŠIBENIKU

ODJEL POSLOVNE INFORMATIKA

PREDDIPLOMSKI STRUČNI STUDIJ POSLOVNE INFORMATIKE

Marko Sudar

IZRADA MOBILNE APLIKACIJE KORISTEĆI PROGRAMSKI OKVIR UNITY NA STUDIJSKOM SLUČAJU MOBILNE IGRE

Završni rad

Šibenik, 2023.

VELEUČILIŠTE U ŠIBENIKU

ODJEL POSLOVNE INFORMATIKE

PREDDIPLOMSKI STRUČNI STUDIJ POSLOVNE INFORMATIKE

MOBILE APPLICATION DEVELOPMENT USING UNITY FRAMEWORK ON MOBILE GAME SHOWCASE

Završni rad

Kolegij: Izrada mobilnih aplikacija

Mentor: Marko Pavelić

Student: Marko Sudar

Matični broj studenta: 1219063863

Šibenik, 6 2023.

TEMELJNA DOKUMENTACIJSKA KARTICA

Veleučilište u Šibeniku Završni rad

Odjel Poslovne Informatike

Preddiplomski/Specijalistički diplomski stručni studij Poslovne Informatike

Izrada mobilne aplikacije koristeći programski okvir Unity na studijskom slučaju mobilne igre

Marko Sudar

Budačka Ulica 263F, markosudar3@gmail.com

Sažetak rada (opseg do 300 riječi)

Izgrađen je jednostavan koncept *Roguelike* videoigre s fokusom performanse na mnogim starijim i novim uređajima, koristeći se dostupnim alatima za definiranje izgleda i logike aplikacije. Napravljena je mapa otvorenog svijeta (*open world map)* s jednostavnim konceptom dostupnih moći igraču, mehanikom nivoa (*level)* i životnih bodova koji omogućuju napredak igrača kroz valove protivnika.

(70 stranica / 34 slika / 20 tablica / 23 literaturnih navoda / jezik izvornika: hrvatski)

Rad je pohranjen u digitalnom repozitoriju Knjižnice Veleučilišta u Šibeniku

Ključne riječi: Unity, C#, Mobilna Aplikacija, Game development

Mentor: Marko Pavelić

Rad je prihvaćen za obranu dana:

BASIC DOCUMENTATION CARD

Polytechnic of Šibenik Batchelor/Graduation Thesis

Department of Business informatics

Professional Undergraduate/Graduate Studies of Business informatics

Mobile Application in program Unity

Marko Sudar

Budačka Ulica 263F, markosudar3@gmail.com

Abstract (up to 300 words)

A simple concept of a Roguelike video game has been developed with a focus on performance on both older and newer devices, using available tools for defining the appearance and logic of the application. An open-world map has been created with a straightforward concept of available abilities for the player, level mechanics, and health points that enable the player to progress through waves of enemies.

(70 pages / 34 figures / 20 tables / 23 references / original in Croatian language)

Thesis deposited in Polytechnic of Šibenik Library digital repository

Keywords: Unity, C# Mobile Application, Game development

Supervisor: Marko Pavelić

Paper accepted:

Sadržaj

1. UVOD

Programski alat Unity je programski okvir korišten za razvoj aplikacija na raznim platformama napisan u programskom jeziku C++, koristeći programski jezik C# za pisanje skripti (*scripting API)* za programski alat Unity.

Tema mobilne aplikacije za operacijski sustav Android je „*Roguelike*“ mobilna igra. Cilj je postići što bolji krajnji rezultat iz svakog pokušaja. Započinjanje razvoja igre jednako je provođenju projekta, što bi značilo da treba paziti na vremenski limit tako što zaključujemo koliko vremena je potrebno za dijelove razvoja igre i koliko je potrebno resursa poput materijala za mobilnu aplikaciju.

Materijali za aplikaciju smatraju se teksturama (*texture)*, „*mesh materials*“, modeli, „*asset*“ paketi (Poput knjižnice *(librarya)* za programski jezik C++ i programski jezik C# samo su zapravo za programski alat Unity), animacije itd..

Programski alat Unity, kao i programski alat Unreal Engine, nudi besplatne verzije svog programa za nove programere i ljude koji se zanimaju za razvoj i dizajn aplikacija. Također pružaju besplatne lekcije (*courses*) za započinjanje svog prvog programa. Često programeri a pogotovo programeri fokusirani na izradu igara (*game developeri)* preuzimaju već gotove materijale ili modele da ubrzaju produkciju projekta. Kao rezultat omogućuje više projektnog vremena za druge dijelove projekta, a ponekad i uštedi na troškovniku projekta pošto je kreiranje novog materijala, tekstura ili modela vrlo zahtjevno kako vremenski tako i koristeći druge resursa projekta.

Programski alat Unity koristi dva programska jezika C++ i programski jezik C#. U većini slučajeva za jednostavne aplikacije i web programe se koristi programski jezik C++, dok se programski jezik C# više koristi specifično za igre to jest programiranje video igara. Prilikom razvoja mobilne aplikacije u programskom

alatu Unityu koristio se programski jezik C# radi jednostavnijeg razumijevanja koda za *game development*. Radi bolje podržanosti za programski jezik C# kao i optimizacije programskog jezika za programski alat Unity *engine*. Programski jezik C++ je osnovna razina programiranja što ga čini kompleksnim za primijeniti u nekim situacijama poput *timera* u *game developmentu* i radi osnovnih klasa poput *Update*, *Start*, *FixedUpdate* itd,. Programski jezik C++ se koristi kad je potrebno neki vanjski dio programa brzo prevesti natrag u programski alat Unity što dovodi do optimiziranije i brže aplikacije.

2. PROGRAMSKI ALAT UNITY

Programski alat Unity prvi je put na tržište izašao 2005. godini na Apple konferenciji „*Worldwide Developers Conference*“ kao Mac OS X *game engine*. Kasnije se proširio i na mnoge druge platforme poput računala, mobilnih uređaja, konzole i virtualnu realnost. (Dealessandri, 2020)

Programski alat Unity izgrađen je u programskom jeziku C++, koristi programski jezik C# za skripte unutar programskog alata Unity *skripting API-a*.

Slika 1-Unity Logo

Izvor 1: (Unity, 2023)

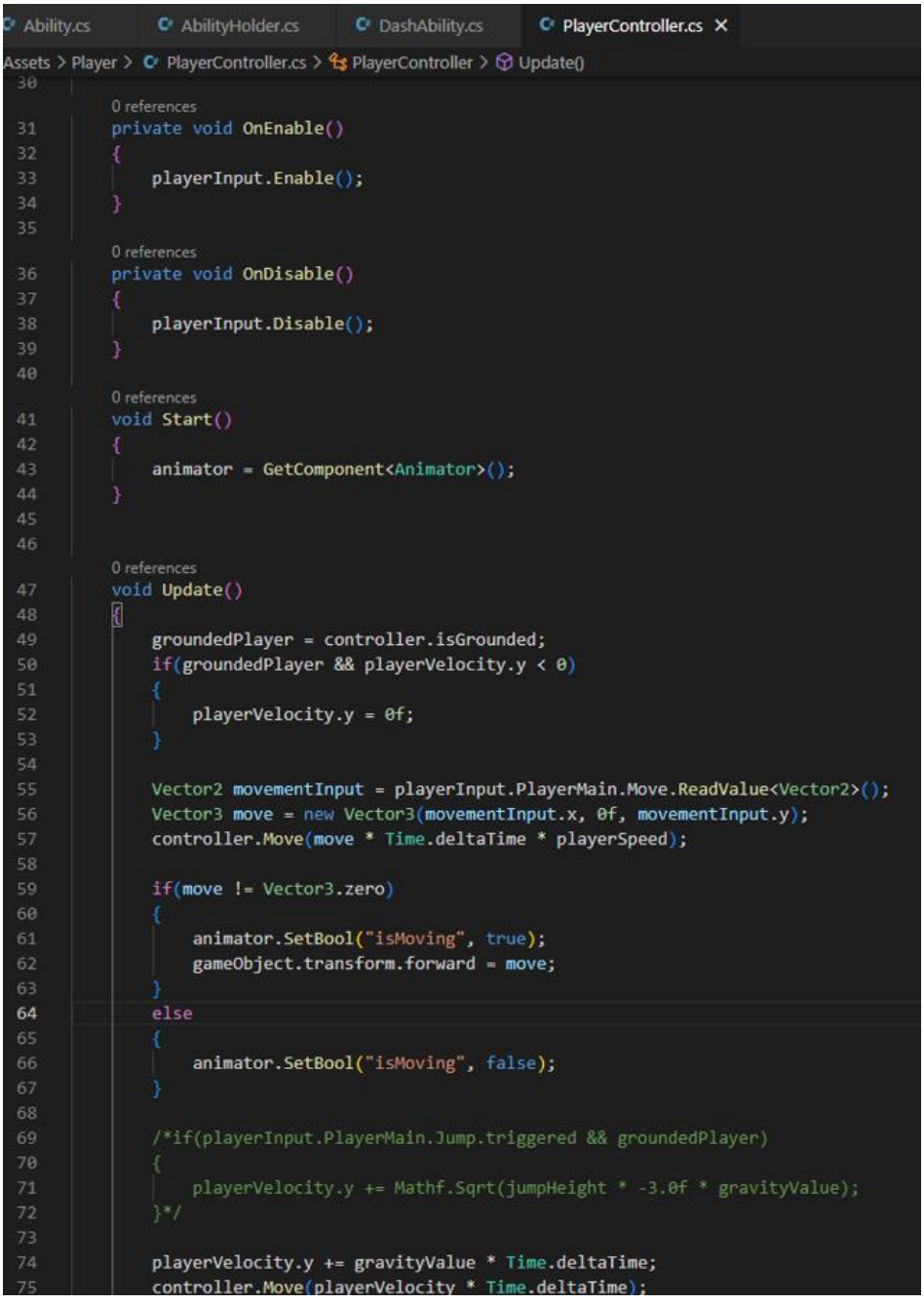
Programski alat Unity je besplatan za studente i nove programere preko kojeg mogu započeti kao i naučiti *game development* te programski jezik C#. Programski alat Unity također može koristiti i plaćenu licencu pomoću koje se ostvaruju benefiti za firme i grupe programera što olakšava objavljivanje gotovih projekata *online*. Može se i s besplatnom opcijom objaviti projekt dok bi programski alat Unity uzeo

neki dio dobitka od uspješnosti projekta. (Robertson, 2015)

2.1. SKRIPTE

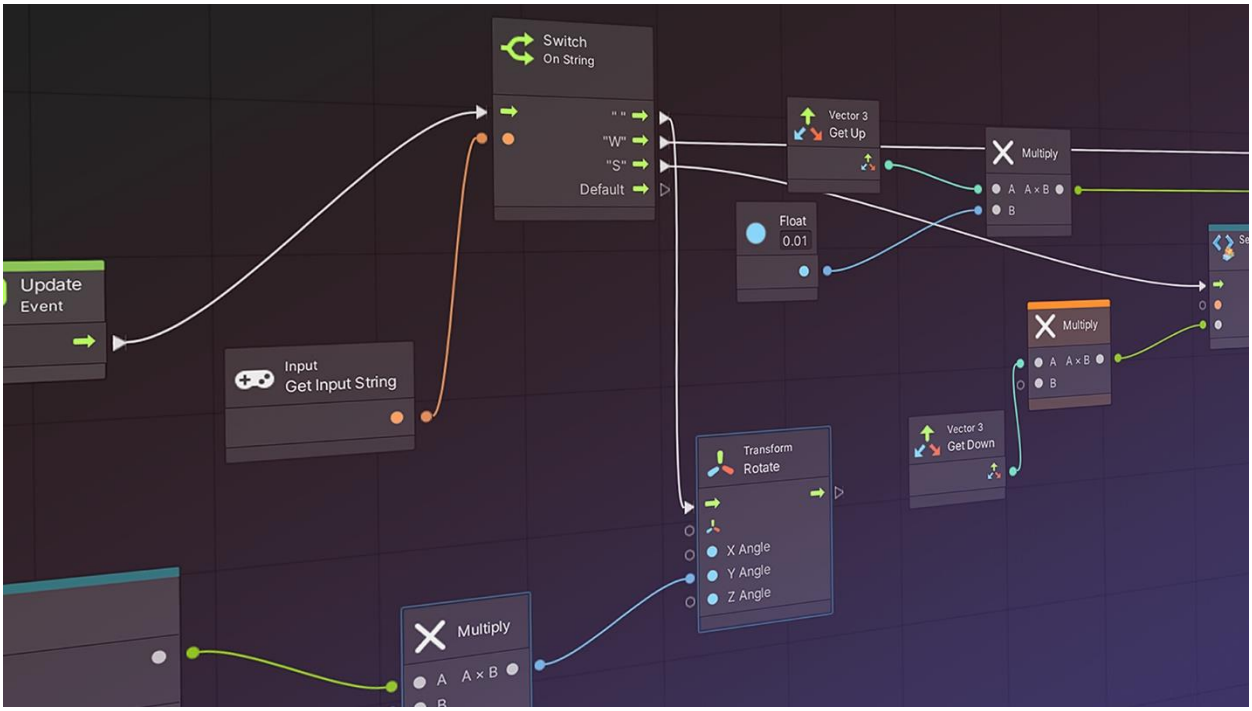
Skripte u programskom alatu Unity su sastav metoda i klasa pisano u programskom jeziku C#, koje provode određene funkcije za specifično određenu namjenu u *Sceni*. Uglavnom kreiraju i spremaju u mapi *Asset.* Nakon kreacije i pisanja skripte, ona se stavlja na *GameObject* da bi se aktivirala.

Slika 2- Primjer C# skripte



Programski alat Unity koristio je skripte kao način programiranja zbog više mogućnosti modifikacija. Dodatno su predstavili i *Visual Coding* kao opciju za nove programere. Kojima je lakše vizualno spajati i odrediti neke metode bez da moraju znati programirati koristeći programski jezik C#.

Slika 3- Visual Scripting



Izvor 2: (Unity, 2023)

3. PROGRAMSKI JEZIK C# i OPERACIJSKI SUSTAV ANDROID

Programski jezik C# je moderni objektno-orijentiran i komponentama-orijentiran programski jezik korišten u mnogim vrstama softvera i *game engina* i njihovih vrsta skripti. Zove se i *component-oriented* jer je napravljen u svrhu toga da bude što prirodniji za kreiranje i korištenje softverskih komponenti (Microsoft, 2023). Programski jezik C# također se koristi za razvoj mnogih aplikacija koje se pokreću u .NET-u te su vrlo zaštićene.

Programski jezik C# je iz porodice prepoznatljiv programerima koji su imali iskustva s programskim jezikom C, programski jezik C++, programski jezik Java i programski jezik JavaScript. Anders Hejlsberg je Danski programski inženjer koji radi u Microsoftu. Anders je dizajnirao i kreirao programski jezik C# te je trenutačno

i vodeći arhitekt za programski jezik C# u Microsoftu (Microsoft, 2006). Glavni je developer za TypeScript i također je poznat po programskim jezicima Turbo Pascal i Delphi. (GitHub, 2023)

Slika 4- Anders Hejlsberg

Izvor 3: (GStatic, 2023)

3.1. OS ANDROID

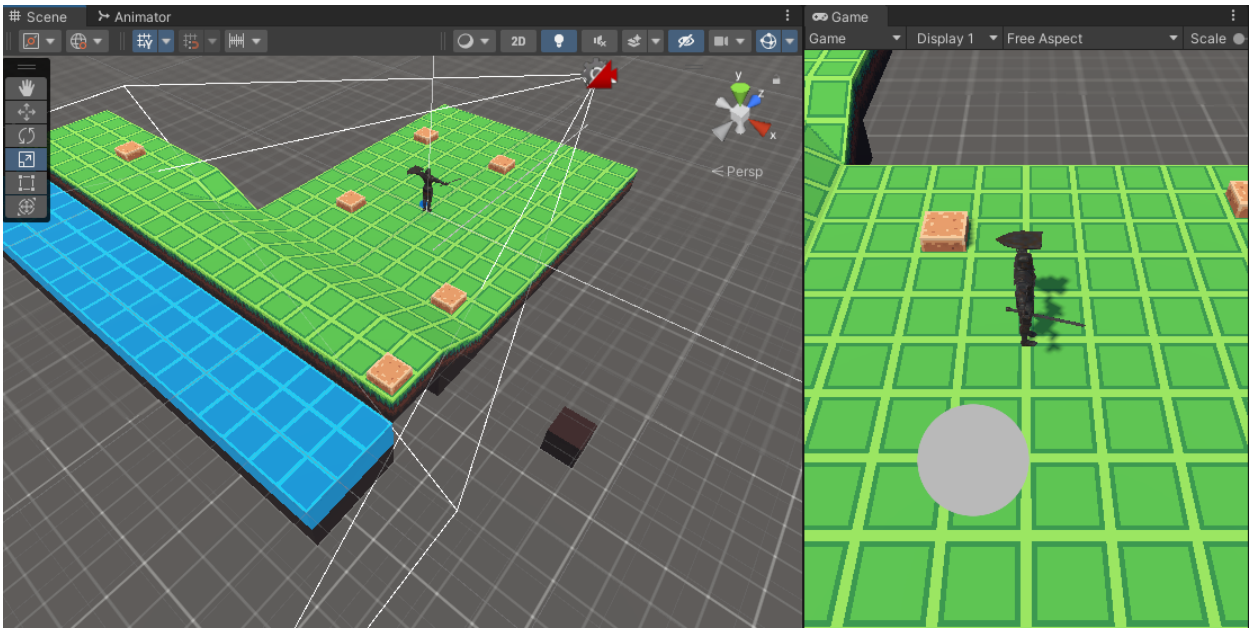
OS Android mobilni je operacijski sustav osnovan na modificiranim verzijama Linux kernel-a i drugih knjižnica otvorenog koda (*open-source API)* . Dizajniran specifično za ekrane osjetljive na dodir (*touchscreen)* i za mobitele, tablete te druge uređaje (Android, 2022).

Android su kreirali Andy Rubin, Nick Sears i Rich Miner. Rich Miner je poznat i po osnivanju Wildfire Communications. Andy Rubin je poznat i kao bivši Google *vice-president.* Nick Sears je poznat po tome što je sudjelovao u dosta revolucionarnih projekata tijekom 90-tih kad je Internet postajao popularniji (FAUguy, 2011).

4. IZGRADNJA SVIJETA

Svaki *game development* se prvo provodi sa objektima za čuvanje/rezerviranje mjesta (*placeholderima*) što su zapravo prazni objekti koji imitiraju neku buduću funkciju ili pripomažu izgradnji mehaničkog dijela igre. Uglavnom su bez teksture te sadrže osnovna područja sa interakcijom (*collidere)* sa jednobojnim (*solid color)* materijalom. Možemo vidjeti na slici 5. primjer *placeholdera* kao mali dio mape kreiran sa *colliderom* da imitira teren na kojem je igrač.

Slika 5- Primjer placeholder-a



Tijekom izgradnje svijeta u igri potrebno je uočiti potrebu proceduralne ili već određene veličine mape. S obzirom da se koristi mobilni uređaj kao platformu za aplikaciju moramo uzeti u obzir i hardversku snagu mobilnih uređaja. To nam određuje da je optimalno napraviti statičku mapu te ju optimizirati za rad.

Roguelike igre općenito imaju već prije definirane lokacije i mape na kojima korisnik to jest igrač može igrati te ih istraživati. Sadrže uglavnom prije definirane *Boss room's* ili neki ključni *event* u igri što bi aktiviralo *Bossa*. U ovom slučaju koristimo *Wave system* od 25 rundi kao ključni *event* za prijeći nivo igre što bi

uzrokovalo *event* instanciranja *bossa*.

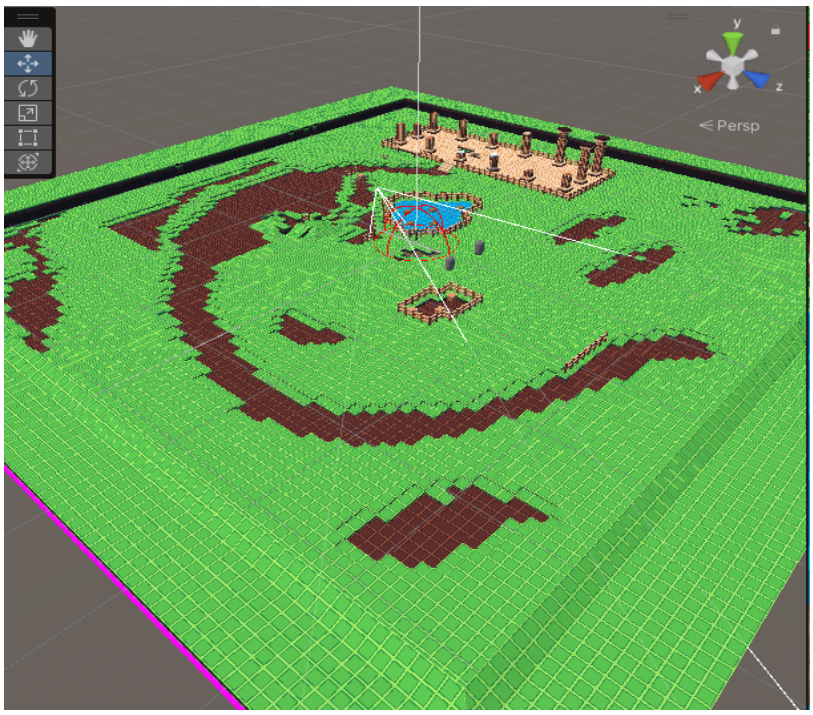
Nakon određenog cilja u igri kreiramo svijet koji će biti prikladan za takvu igru što bi značilo da svijet mora biti prije definiran i dovoljno velik da igrač može tijekom borbe istraživati i naći bolje lokacije sa svojim prednostima za preživjeti 25 rundi.

Mobilni uređaji ne mogu prikazati (*renderati)* kompleksne teksture i materijale te je fokus na *low-poly* objektima s *low poly* materijalima i teksturama. Na slici 6. možemo vidjeti razliku između *low-poly*(lijevo) i *normal poly* objekta(desno).

Slika 6-Primjer low-poly

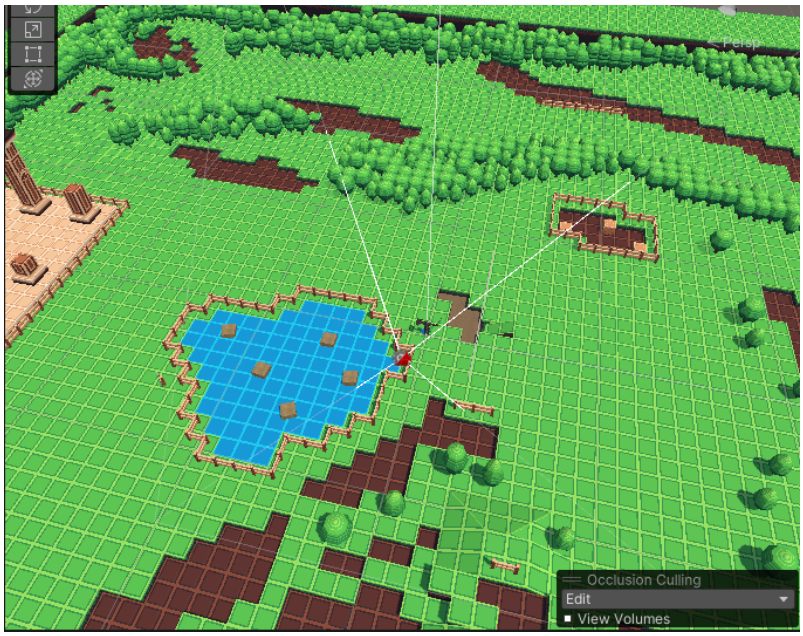
Definiramo veličinu mape te stavljamo osnovne teksture i materijale za prepoznavanje razlike u terenu na mapi. Na slici 7. prikazana je osnovna veličina mape kao i definicije terena poput dubina, glavnih objekata t.j. zgrada i područja za vodu.

Slika 7- Osnovna veličina mape



Na slici 8. stavljamo detalje(drveća, kamenja….) po mapi, ali ne finalne detalje već *placeholdere* koji služe za bolju vizualizaciju izgleda mape.

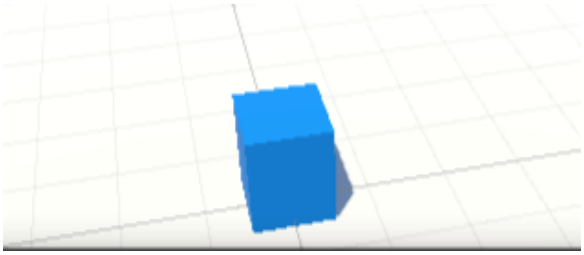
Slika 8- Detalji mape



Nakon što se dovrši izgled mape te osnovni detalji koje je potrebno imati, prelazi se na dizajn tekstura i materijala. Materijali sami po sebi mogu definirati i kompletno promijeniti izgled objekta bilo to po *tessellationu* (*tessallation* je redoslijed jedne ili više grafičkih oblika zajedno spojenih koji se ne preklapaju niti imaju rupe, stvarajući dojam dubine i/ili većeg detalja objekta (Freese, 2020)) objekta ili drugih efekata. Teksture se koriste za rezoluciju i sliku objekta. U kombinaciji s poželjnim materijalom može se proizvesti izgled dubine i detalja iz obične kocke, u objekt koji je prikazan na slici 9. u usporedbi s objektom t.j. kockom koja je prikazana na slici 10.

Slika 9- Primjer materijala i textura

Slika 10- Primjer objekta kocke



Na slici 11. možemo uočiti promijenjene teksture i materijale čitave mape u usporedbi s dizajnom mape na slici 8.

Slika 11- Primjer rekreacije textura i materijala

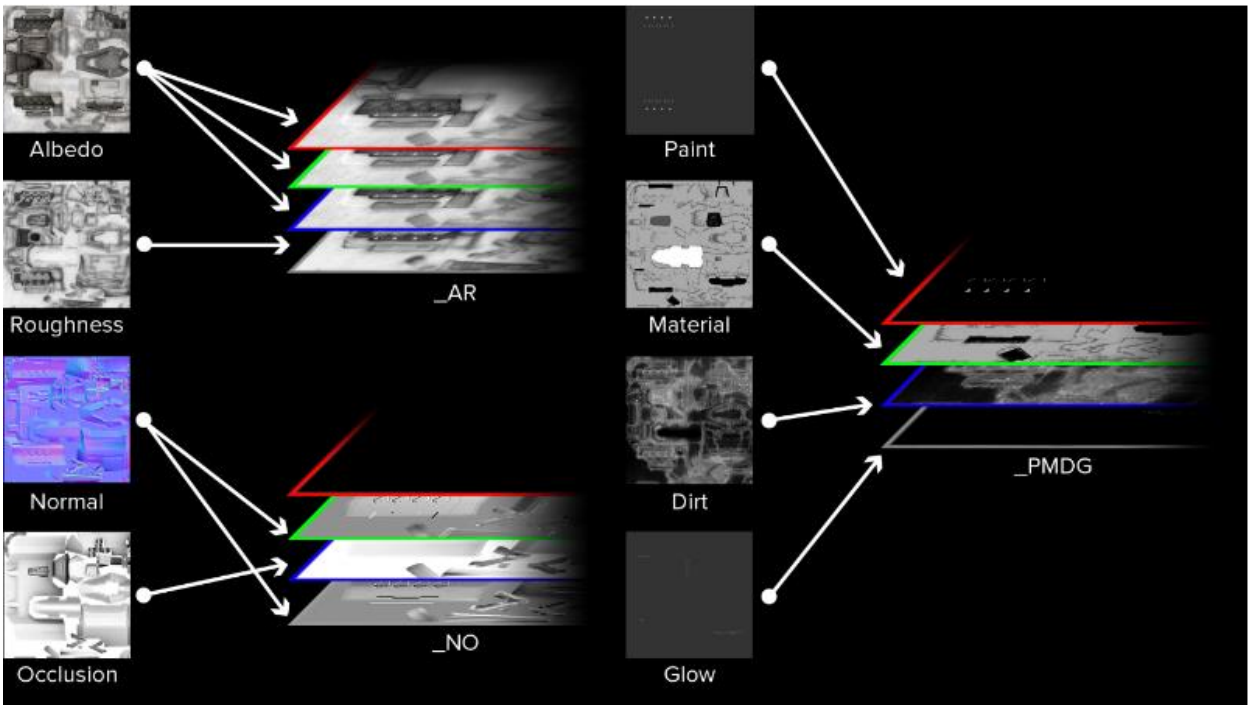
Izgled mape prelazimo na pokrivanje to jest maskiranje granica mape tako da igrač ne može naići na dio gdje može vidjeti rupu t.j. kraj svijeta u kojem se nalazi. Na slici 12. prikazano je 2D slike *(sprites)* drveća oko mape kako bi se stvorila iluzija lokacije u šumi i iluzija veće mape nego što jest.

Slika 12- Primjer maskiranja mape

5. TEKSTURE

Svako početno stanje objekta u programskom alatu Unity se sastoji od komponenti: *Transform*, *mesh renderer*, *material*. U materijalima nalazi se tekstura što određuje izgled objekta. Teksture se sastoje od slike i *lightmapsa* kao i ostalih dodatnih stvari za detalje tekstura. Na slici 13. možemo vidjeti primjer sastava tekstura.

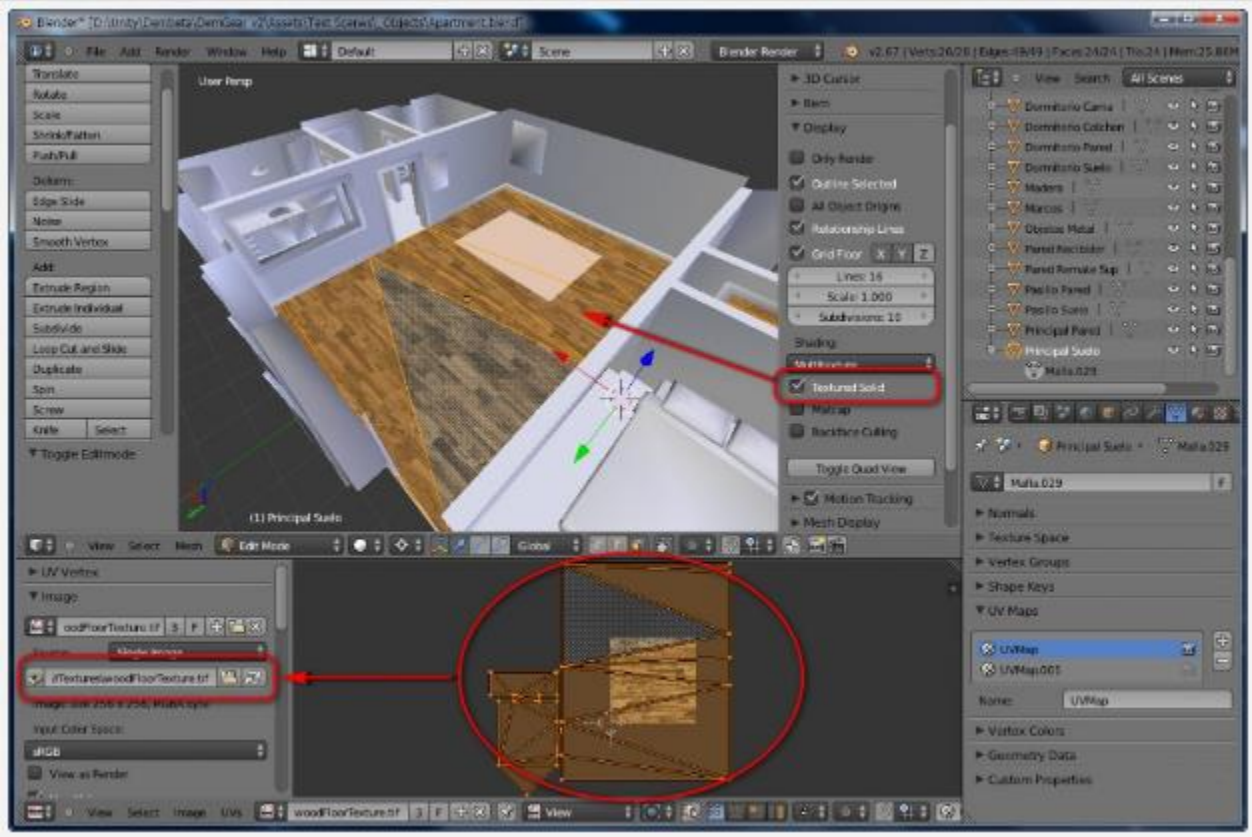
Slika 13- Primjer sastava teksture



Izvor 4: (EVE Online, 2023)

Nakon što se kreiraju teksture u aplikaciji Blender ili implementirali druge u svoj projekt, može se započeti s korištenjem izabrane teksture. Za koristiti teksture potrebno je isjeći objekt u Blenderu to jest *prefab* objekta i prilagoditi slici teksture tako da se poklapa s traženim izgledom, primjer možemo vidjeti na slici 14. Nakon mapiranja teksture na *prefab* potvrđujemo promjenu te dodajemo u materijale i dobivamo željeni izgled.

Slika 14- Primjer stavljanja teksture na objekt u Blender-u



Izvor 5: (EDY, 2014)

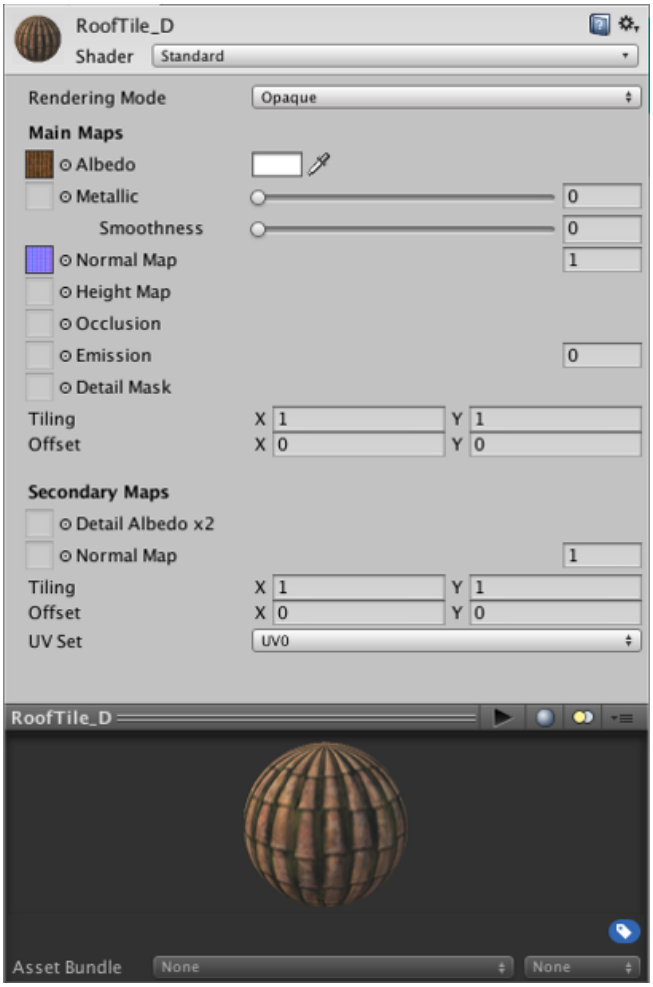
6. MATERIJALI U PROGRAMSKOM ALATU UNITYU

Materijali u programskom alatu Unity koriste se za definiranje izgleda scene pomoću materijala i sjena *(engl. shadersa)*. Primjer materijala možemo uočiti na slici 15., također možemo vidjeti da se sastoji od *shader-a* što bi bio *standard* prikaz ili URP (*Universal Pipeline Renderer*) *shader* te se sastoji od glavnih mapa što su teksture te dubine i definicije teksture da bi dobili iluziju 3D materijala za objekt.

Postoje osnovne izbore od „*Standard*“ *shader-a* i URP *shader-a* koji se često koriste za manje projekte poput mobilnih aplikacija. Imamo i HDRP (*High Definition Render Pipeline*) i SRP (*Scriptable Render Pipeline*) koji se koriste za puno veće aplikacije. Razlika *Standard-a* i URP je to što se URP može mijenjati i prilagoditi potrebi developera dok je *standard* neki osnovni izgled materijala.

U projektu se koristi URP *shader* kao način iscrtavanja (engl. *renderanja*) materijala radi potrebe za optimizacijom aplikacije za mobilne uređaje.

Slika 15- Primjer materijala



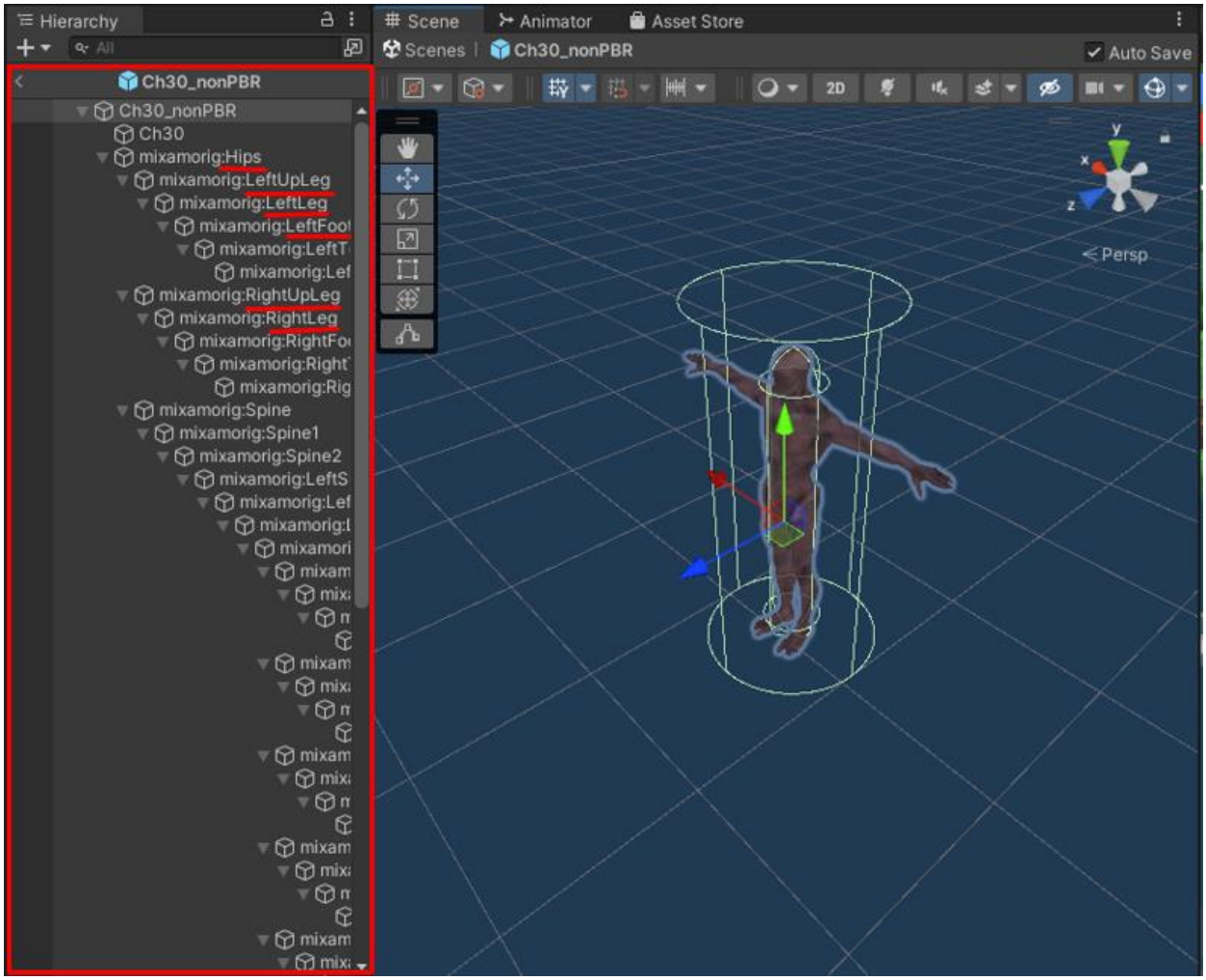
Izvor 6: (Unity, 2023)

7. MODELI U PROGRAMSKOM ALATU UNITY-U

Modeli za programski alat Unity često se izrađuju u aplikaciji Blender te bilo koja druga aplikacija koja u *export-u* podržava *fbx* format programskog alata Unity. Modeli se sastoje od dijelova po hijerarhiji bitnosti namijenjenog izgleda. Primjer bi bio ljudski model s vezom roditelj-dijete (*child-parent*) između ramena i gornjeg dijela tijela ili ostatka ruke s ramenom.

Na slici 16. Možemo vidjeti primjer *child-parent* veze dijelova modela te i sami model sa stavljenim i prilagođenim *colliderima*.

Slika 16- Primjer modela prefab 1

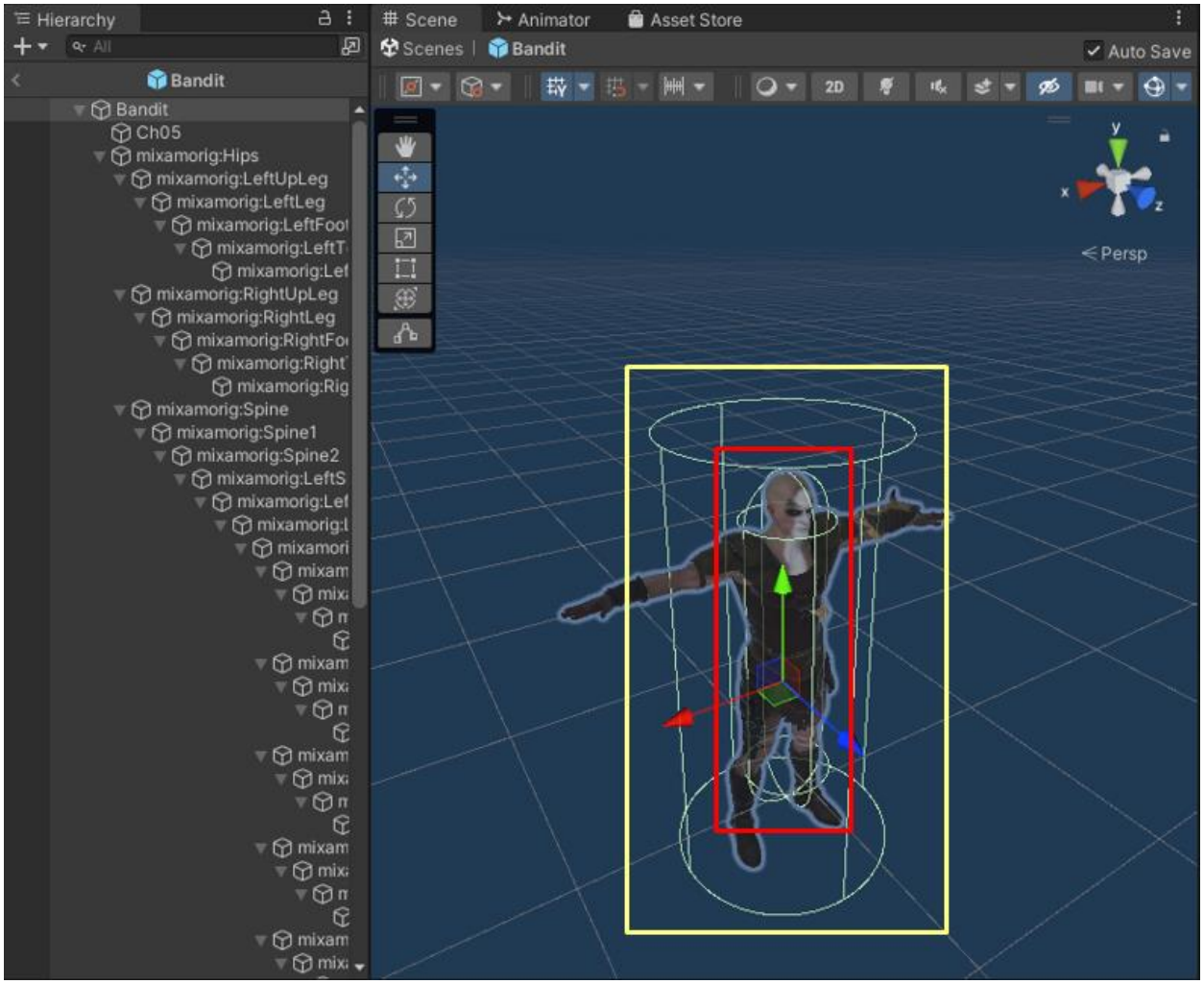


Koriste se tri *prefaba* protivnika za projekt. Dva su normalni *enemy* koje susrećemo tijekom igre te je treći *boss prefab* kojeg susretnemo na kraj igre. *Prefabi* dolaze s *transform* komponentom, *materials*, *mesh renderer* te je potrebno nadodati skripte za funkciju umjetne inteligincije (AI-a) protivnika, *NavMeshAgent*, *Collider* (u ovom slučaju *sphere collider*) da bi objekt postao funkcionalan.

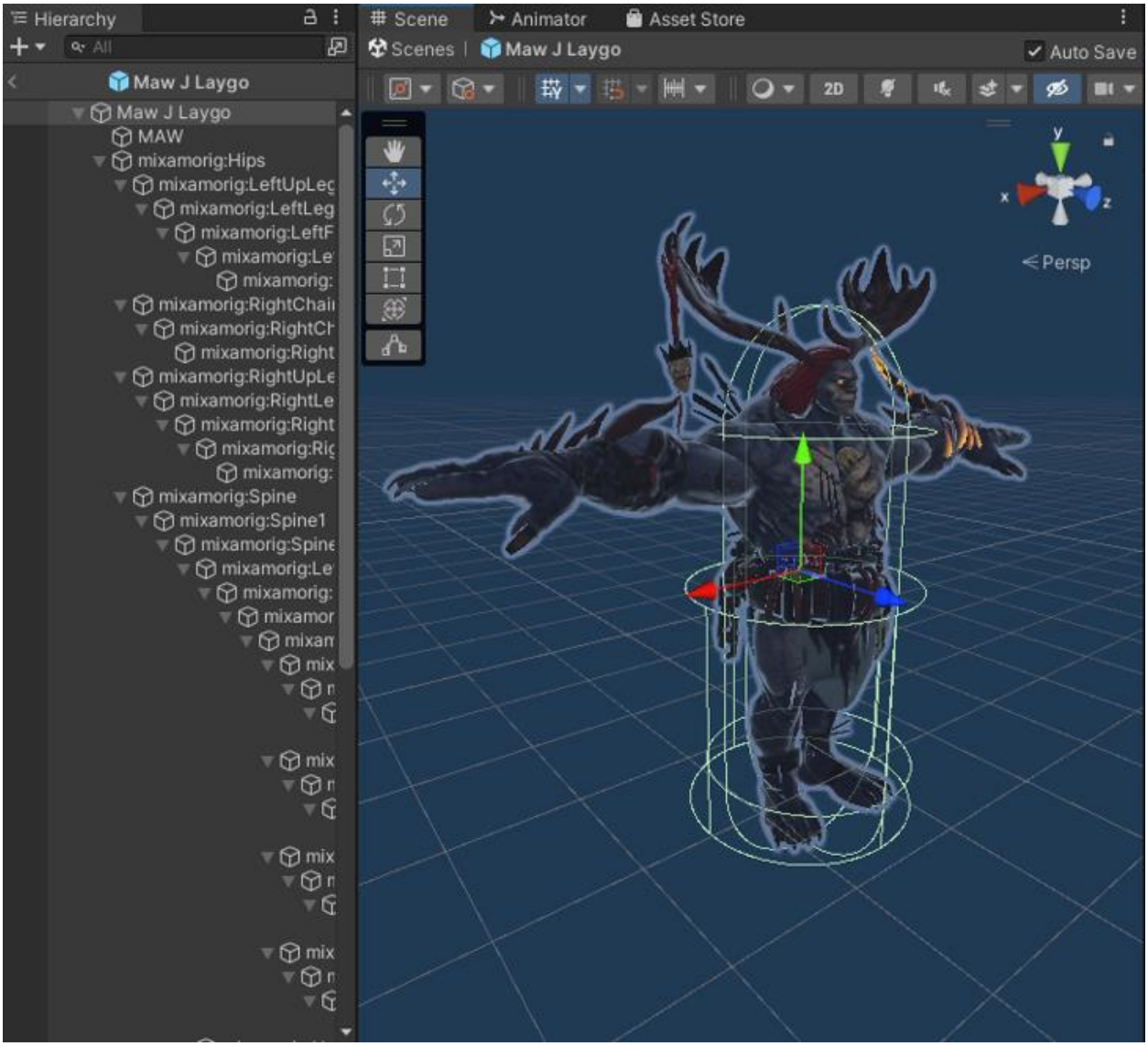
Na slici 17. Možemo uočiti dva *collidera* od kojih je jedan *NavMeshAgent* (žuta boja) koji omogućuje prepoznavanje terena te točno kretanje po terenu i

izmicanju prepreka. Koristi se *Sphere collider* (crvena boja) koji omogućuje interakciju igrača i protivnika. Primjer modela glavnog protivnika (*bossa)* možemo vidjeti na slici 18.

Slika 17- Primjer modela prefab 2



Slika 18-. Primjer boss prefab-a



8. SUSTAV IGRE

Nakon određivanja cilja igre, kreira se glavna mehanika poput kretanja igrača, određivanje igrača, kontrole igrača i koji objekt je igrač te kako se treba ponašati. Stavljamo objekt kojeg želimo da bude igrač kao „*Player*“ pod sloj(*Layer*) i oznaku (*Tag)* objekta. Kreiramo platno (*canvas*) što omogućuje prikaz tipki i drugih UI elemenata na ekranu bez da se prikažu u sceni svijeta aplikacije. Stavljaju se prve

kontrole igrača koje se upravljaju preko „*Stick shift-a*“ na ekranu. Kreirajući skriptu micanja objekta pomoću *stick-a* nam daje da mičemo objekt pomoću vektora što su vrijednosti x, y, z. Imamo dvije vrste *vectora* koji se koriste za kretanje to su *Vector2* i *Vector3*, gdje se *vector2* koristi za 2D aplikacije, a *vector3* koristi za 3D aplikacije. Premda se kod 3D aplikacije mogu obadva koristiti u specifičnim slučajevima. Za micanje objekta koristim „*transform*“ komande, koje nam *transformiraju* trenutačne vrijednosti vektora na objektu u vrijednosti određene na *stick shift-u.* Spremaju se ažurne vrijednosti natrag na objekt što omogućuje bilo kojem objektu sa skriptom kretanja da se pomiče za x, y, z udaljenost.

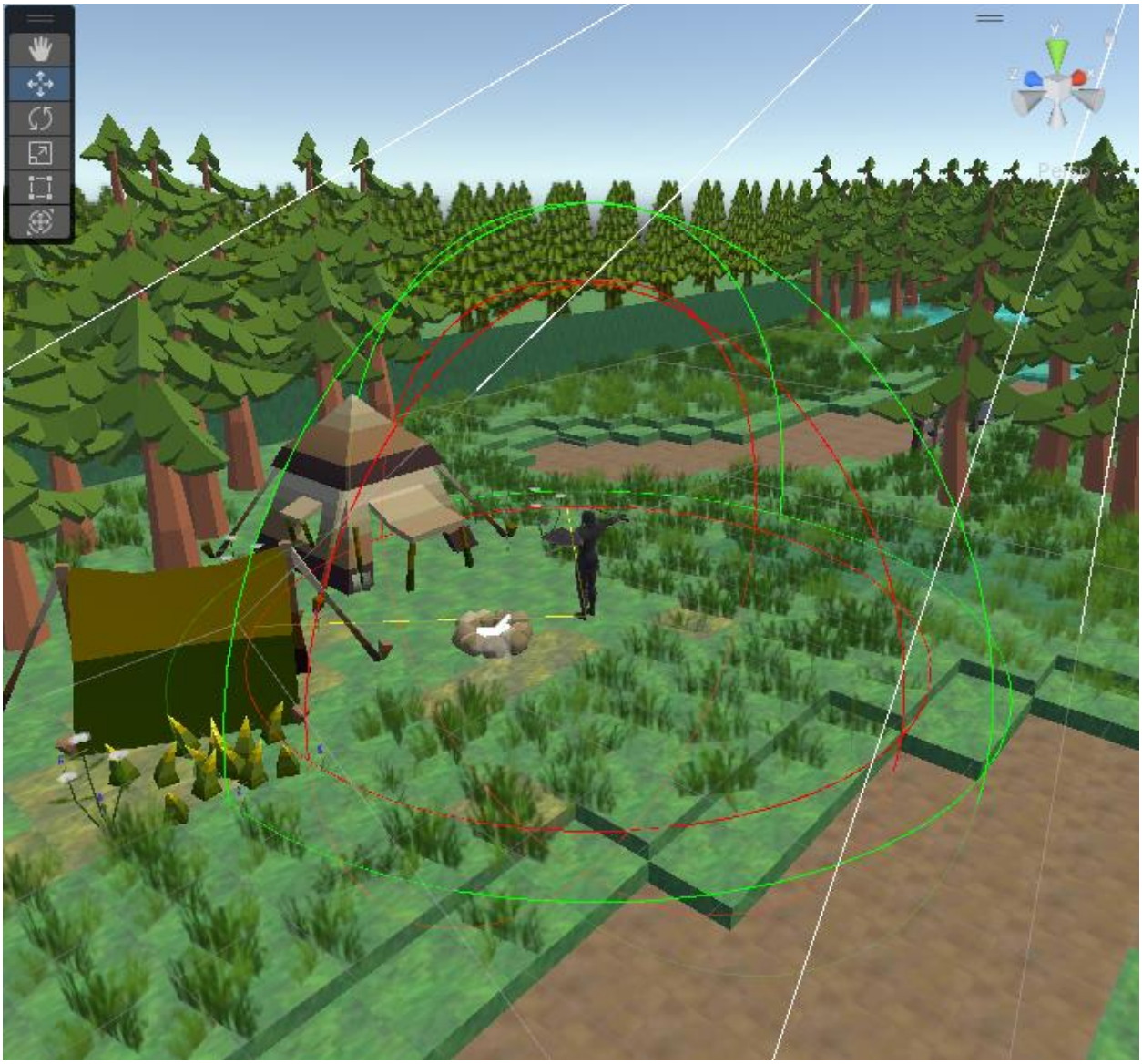
Objektu *player* dodijeljeno je više specifičnih skripti za određene funkcije, poput *Health, Player Controller, LevelXp* i *ability* skripte. Što omogućuje objektu da posjeduje „*Life points*“ koji određuju dali je igrač živ ili mrtav. Provjeravajući jeli vrijednost *float health* manja, jednaka ili veća od nule te od tuda proizlaze vrijednosti za vizualni prikaz *HealthBar-a* korisniku kao trenutačni postotak života.

LevelXp služi za provjeru trenutačnog i nadodanog iskustva igrača i dali je trenutačna vrijednosti veća ili jednaka od sto posto, ako je povećava broj *Level-a* igrača za jedan. Što omogućuje igraču da kroz *level-e* bude jači tako da otključa zadane moći. *Player controller* je od programskog alata Unity built in mobilni kontroler za igrača, provjerava na kojem objektu je skripta i kako se objekt treba ponašati tijekom micanja gumba na poziciji x, y, z.

Ability skripte se sastoje od četiri zasebne skripte s posebnim funkcijama, u kojima su tri skripte slične za moći igrača. Da bi aplikacija znala kad je igrač udario protivnika ili protivnik igrača trebamo provjeriti dali je *collider* igrača u razini *collidera* protivnika i obrnuto. Ako je oduzima zadanu vrijednost od životnih bodova igrača ili protivnika. Prva moć provjerava sa *colliderom* jeli protivnik pet vrijednosti x ili y ispred igrača ovisno o tome koja os je prednji dio igrača u radijusu od četrdeset i pet stupnjeva, kao što je prikazano na slici 19. Prikazan je radijus i daljina udarca

moći sa žutom bojom te s crvenom i zelenom su prikazani radijusi od ostalih moći.

Slika 19- Primjer AOE (Area Of Effect)

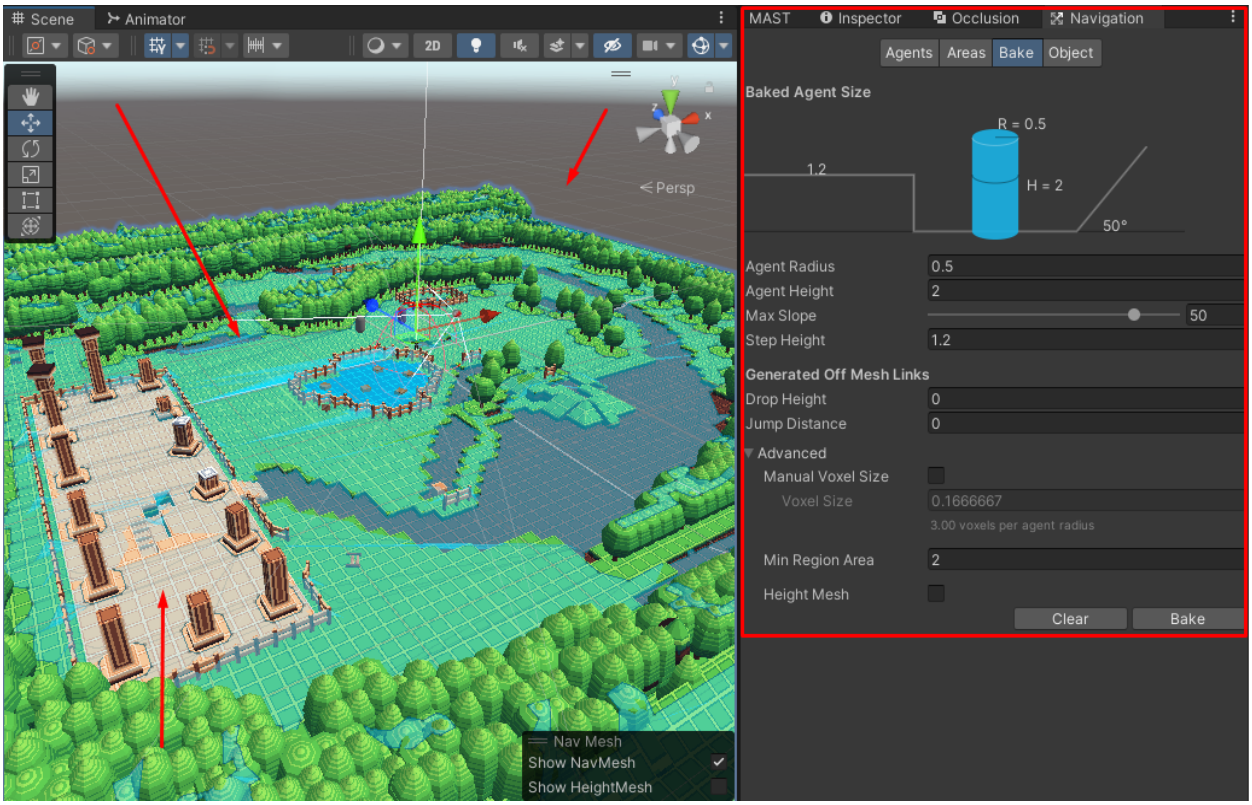


Na slici 19. crvena i zelena boja (u upotrebi je *gizmos* funkcija za vizualni prikaz interaktivnog prostora specifične funkcije) predstavljaju druge moći igrača od kojih su dvije na crvenoj razini i zadnja do koje je i najteže doći je na zelenoj razini. Da bi korisnik uočio napredak u igri potrebno je otključati moći kroz razine *levela* tijekom igre. Kreće se od najslabije moći do najjače.

Nakon uređivanja mape dodajemo *box collidere* na objekte mape s maknutom

gravitacijom te obrnuto s uključenom gravitacijom na objektima kojima je potrebna za kretanje i prepoznavanje razine terena. Da bi objekt *player* prepoznao teren potreban mu je *collider* koji ima stalnu interakciju s drugim objektima. Da bi umjetna inteligencija protivnika prepoznala teren potreban je *NavMeshAgent* koji *bake-a* te iscrta teren po kojem se može kretati bez smetnji, kao što uočavamo na primjeru slike 20.

Slika 20- Primjer navigacije NavMeshAgent-a



8.1. UMJETNA INTELIGENCIJA ZA UPRAVLJANJE NEPRIJATELJSKIM IGRAČIMA

Mehanika za kretanje protivnika je kreirana na sasvim drugi način, pošto nije potreban trenutačna ažurna lokacija već dali je došlo do te lokacije. Kao primjer *player* se kreće tako što gleda pomaknutu x, y, z putanje te ju sprema u *transform* objekta što korisniku zapravo izgleda da se objekt igrača kreće. Objekt enemy-a se

kreće tako da gleda trenutačnu poziciju i poziciju igrača te oduzima trenutačnu poziciju sve dok ne dođe do pozicije igrača. Da se ne zabija u igrača i traži baš njegovu poziciju dodajemo fiksni razmak pozicije igrača i trenutačne pozicije, u ovom slučaju za vrijednost jedan u x, y smjeru. Protivnik gleda u igrača tako da rotaciju zaključava na objekt igrača. Što nam je u prevedenom značenju prikazano na slici 21. s crvenom bojom kao trenutačnu poziciju protivnika te zelenom kao poziciju igrača i žutom je prikazana fiksna udaljenost koje se protivnik mora držati.

Slika 21- Primjer kretanja enemy AI-a

Da protivnik efektivno uspije napasti igrača koristi se *collider* za udarac sličan *collideru* sa slike 19., kad je *collider* aktivan deaktivira se sa *flag* (*flag* uglavnom

predstavlja neku vrijednost *booleana* koja se koristi kao ključ da pokrene ili zaustavi neku radnju. Umjesto *flaga* se može koristiti i *state* sustav) funkcijom pokret protivnika i pokreće se napad. Kad napad završi opet provjerava jeli u razini *collidera* ako nije traži poziciju igrača. Za efektivno kretanje po mapi i po terenu koristi *bake-ani NavMeshAgent-a* za putanju kao što je prikazan plavom bojom na slici 20.

8.2. KORISNIČKO SUČELJE

UI sustav za korisnika je prikazan s bitnim informacijama poput trenutačni broj protivnika, trenutačni *wave* protivnika, *level*, *level xp* kao bar na vrhu ekrana. *Health bar* na vrhu ekrana, s gumbom za kretanje i tipkama za moći na donjem dijelu ekrana sa tipkom za opcije u kutu ekrana (slika 22.).

Slika 22- Primjer UI

8.3. LEVEL I WAVE SUSTAV

Level sustav gleda početni nivo (*level)* igrača što je jedan, za svakih sto iskustva (*xp*-a) dodaje plus jedan *level*. Da bi *level sustav* znao kad je došao do sto posto *xp*-a gleda se trenutačni *xp* i dobiveni *xp*. Provjera dali trenutačni *xp* prelazi broj sto to jest varijablu maksimum *xp*, u slučaju da prelazi dodajemo jedan *level*. U slučaju da nam je ostalo viška *xp*-a od trenutačnog onda nadodajemo ostatak na novi *level*. *Leveli* su potrebni za otključavanje moći kao što je prikazano na slici 22. Moći olakšavaju preživljavanje igrača. Provjera potrebnog nivoa igrača za novu moć se provodi tako što se gleda jeli trenutačni *level* veći od određenog broja kojeg smo dobili.

Wave sustav funkcionira na način tako što uzima *prefab* protivnika i instancira ga po razini valova. Za svaki val nadodaje puta dva protivnika. *Wave* sustav funkcionira slično kao *health* i *level* sustav. Tako što gleda trenutačni val i maksimalni te provjerava koliko je instanciranih *gameObject*-a to jest protivnika preostalo aktivno u trenutačnom valu. Ako je broj objekata protivnika manji ili jednak nuli dodajemo plus jedan na val. Provedbom naredbe za nadodani val instanciramo novi broj protivnika, što kao primjer više nije dva košto bi bilo u valu jedan nego je četiri pošto je val dva. Kad *wave* sustav dosegne broj dvadeset pet i u isto vrijeme ako je broj protivnika u zadnjem valu manji ili jednak nuli, onda se isključuje *wave* sustav i instancira se zadnji *prefab* što je *boss prefab*.

8.4. GLAVNI IZBORNIK IGRE

Za vrijeme pokretanje aplikacije ulazimo u *main menu* igre kod kojeg se i automatski učitava ostatak igre. Pružen nam je broj opcija poput informacija u vezi moći i gumb za pokretanje igre. Možemo vidjeti primjer *main menu*-a na slici 23.

Slika 23- Primjer main menu-a

8.5. WIN I GAMEOVER SUSTAV

Kad igrač porazi zadnjeg *boss*a nakon dvadeset petog val-a, aplikacija provjerava jeli instanca tog objekta uništena. Ako je instanca uništena prikazuje *Win screen* ili ako su u bilo kojem trenutku životni bodovi igrača manji ili jednaki nuli prikazujemo *Game Over* ekran i restartamo nivo.

9. SKRIPTE U PROGRAMSKOM ALATU UNITY

Skripte započinju s glavnom klasom koja ima isto ime kao skripta. Nasljeđuje funkcije iz *MonoBehaviour* koji je potreban za funkciju komandi iz programskog alata Unity. U skripti dobivamo funkcije *start* i *update*. Na kodu 1. uočavamo primjer početka skripte.

Kod 1

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class CleaveAbility : MonoBehaviour
```

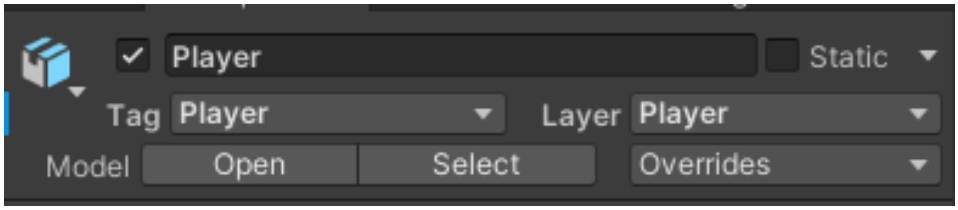
Na kodu 2. su nam prikazane potrebne varijable za funkciju moći. *ActivationButton* provjerava jeli tipka na UI-u pritisnuta preko *bool* vrijednosti što je *true* ili *false.* Varijabla *angle* koja nam provjerava pod koliko stupnjeva je širina udarca. *Range* varijabla koja provjerava udaljenost udarca i *power* varijabla koja daje snagu udarca. *DamageAmount* varijabla primjenjuje zadani broj kao vrijednost koja se oduzima od igračevih životnih bodova. *LayerMask* se koristi za provjeru jeli objekt igrač, primjer postavljanja *layermask*-a možemo vidjeti na slici 24. *Animator* povlači komponentu „*Animator*“ koja je zaslužna za pokretanje zadanih animacija na objektu, ako objekt ispunjuje uvjete potrebne što je *skeleton* objekta. *Bool isCleaving* nam služi kao *flag* da provjerimo dali se moć koristi. Referenca na komponentu *playerController* je potrebna za poziciju s koje će se moć aktivirati i u isto vrijeme se koristi kao referenca na igrača.

Kod 2

```
    public Button activationButton; // The UI button for
activating the ability
    public float angle = 90f; // The angle of the cleave effect
    public float range = 5f; // The range of the cleave effect
    public float power = 10f; // The power of the cleave effect
    public float damageAmount = 45f; // Amount of damage to
apply to enemies
    public LayerMask targetMask; // The layer mask for
selecting targets

    public Animator animator;

    public bool isCleaving = false; // Flag to indicate if the
```

```
ability is currently cleaving

    private PlayerController playerController; // Reference to
the PlayerController script
```

Slika 24- Tag i Layer primjer



U *start*-u na kodu 3. je potrebno staviti funkcije koje želimo da su uzete čim je pokrenuta aplikacija. U ovom slučaju skripta odmah pokreće i uzima vrijednosti *playerControllera,* za poziciju s koje će se aktivirati i provjerava jeli tipka već aktivirana. U slučaju da nije, stavljamo funkciju „*Listener*“ koji je stalno aktiviran kroz aplikaciju i provjerava dali je određena tipka aktivirana.

Kod 3

```
    private void Start()
    {
        // Get the PlayerController component from the player
object
        playerController = GetComponent<PlayerController>();

        // Add a listener to the activation button
        if (activationButton != null)
        {
            activationButton.onClick.AddListener(StartCleave);
        }
    }
```

Kad se aktivira određena tipka kao što je prikazano na kodu 3. onda pokrećemo *StartCleave* funkciju. *StartCleave* u sebi sadržava svu logiku moći. Na kodu 4. uočavamo primjer *StartCleave* metode. Započinjemo s provjerom jeli moć već aktivirana, ako nije stavi da je i pokreni redoslijed funkcija. Miče se mogućnost

kretanja igrača tako što deaktiviramo *playerController* da bi udarac bio realističniji. Koristio se *vector3* da nam pozicionira *collider*. Kad pišemo *forward* gleda se *front face* objekta što je trenutačno usmjereno ispred player objekta. *Quaternion* je rotacijska komanda koja po vektoru gleda rotaciju objekta, što je u primjeru prikazan kako gleda lijevi i desni *face* objekta. Od lijevog oduzima *angle* varijablu koja je 90f i od desnog nadodaje *angle*, te obadva kuta dijelimo s vrijednost dva da bi dobili udarac pod kutem od 45 stupnjeva. Množimo prednju vrijednost, što u prevedenom značenju kreira *collider* u obliku trokuta kao što je žutom bojom prikazano na slici 19. Nakon definiranja pozicije provjeravamo dali kut udarca se poklapa sa *colliderom* od protivnika što je sphere *collider* i provjeravamo dali se *sphere collider* protivnika poklapa s vrijednostima kuta moći. Ako se poklapa *collider* i ako protivnik sadrži skriptu pod imenom *enemyHealth*, onda šaljemo vrijednost *damageAmount* na funkciju *TakeDamage* od protivnika. U prevedenome znači da je protivnik primio štetu. U isto vrijeme ako je udarilo protivnika uzimamo mu *rigidbody* i udaljimo ga za malu vrijednost od igrača. Dok je moć aktivirana stavljamo da je animacija „*isAttacking*“ *true*, što nam pokreće animaciju koja je stavljena pod tim imenom i radi sve dok je moć aktivna. Na kraju imamo „*invoke*“ koji nam nakon određenog vremena pokreće funkciju *StopCleave*.

Kod 4

```
    private void StartCleave()
    {
        if (!isCleaving)
        {
            isCleaving = true;

            // Disable player movement
            playerController.enabled = false;

            // Perform the cleave effect
            Vector3 forward = transform.forward;
            Vector3 left = Quaternion.Euler(0f, -angle / 2f,
0f) * forward;
            Vector3 right = Quaternion.Euler(0f, angle / 2f,
```

```
0f) * forward;

            // Check for targets in the cleave range and within
the angle
            Collider[] colliders =
Physics.OverlapSphere(transform.position, range, targetMask);
            foreach (Collider collider in colliders)
            {
                Vector3 direction = collider.transform.position
- transform.position;
                float angleToTarget = Vector3.Angle(forward,
direction);

                if (angleToTarget <= angle / 2f)
                {
                    // Check if the collider belongs to an
enemy
                    EnemyHealth enemyHealth =
collider.GetComponent<EnemyHealth>();
                    if (enemyHealth != null)
                    {
                        // Apply damage to the enemy
                        enemyHealth.TakeDamage(damageAmount);
                    }

                    // Apply force to the collider
                    Rigidbody rb =
collider.GetComponent<Rigidbody>();
                    if (rb != null)
                    {
                        rb.AddForce(direction.normalized *
power, ForceMode.Impulse);
                    }
                }
            }

            animator.SetBool("isAttacking", true);

            // Stop the cleave after a delay
            Invoke("StopCleave", 0.8335f);
        }
    }
```

Kad je pokrenut *StopCleave* funkcija onda nam se provode komande na kodu 5. Stavljamo da je moć deaktivirana s *bool*om *false.* Aktiviramo natrag mogućnost

kretanja igrača i stavljamo da je animacija deaktivirana.

Kod 5

```
private void StopCleave()
{
    isCleaving = false;

    // Enable player movement
    playerController.enabled = true;

    animator.SetBool("isAttacking", false);
}
```

Izvor 7: Obrada autora

Da bi vizualno iscrtali *collide* zonu moći tako da bude lakše programeru vidjeti rezultat kao što je na slici 19. Onda nam je potrebno provesti komande koje su na kodu 6. *Gizmos* se koristi za vizualni prikaz određenih funkcija u *editoru* programskog alata Unity, da olakša *developeru* prikaz funkcija. Prikazane crte od *gizmos*a nisu prikazane tijekom igranja te su vidljive samo *developeru* u *editoru*. Na kodu 6. uzimamo postojeće vrijednosti iz ostatka dijela skripte i dodajemo *gizmos* za iscrtavanje oblika, linija po vrijednostima vektora, što nam je prikazano s crvenom, žutom i zelenom bojom na slici 19.

Kod 6

```
    private void OnDrawGizmos()
    {
        // Draw the cleave range using Gizmos
        Gizmos.color = Color.red;
        Gizmos.DrawWireSphere(transform.position, range);

        // Calculate the start and end directions of the cleave
effect
        Vector3 forward = transform.forward;
        Vector3 left = Quaternion.Euler(0f, -angle / 2f, 0f) *
forward;
        Vector3 right = Quaternion.Euler(0f, angle / 2f, 0f) *
```

```
forward;

        // Draw the lines representing the cleave angle
        Gizmos.color = Color.yellow;
        Gizmos.DrawLine(transform.position, transform.position
+ left * range);
        Gizmos.DrawLine(transform.position, transform.position
+ right * range);
    }
```

9.1. LEVELXP SKRIPTA

Levelxp skripta se vrti oko provjera trenutačnih vrijednosti. Sa deaktiviranjem specificiranih objekata tijekom zadovoljavanja uvjeta i ažuriranja teksta za level na UI-u kod 7.

Kod 7

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using TMPro;

public class LevelXp : MonoBehaviour
```

Varijable korištene u *levelxp* skripti su prikazane na kodu 8. *maxLevel* za limit levela igrača, *currentLevel* za provjeru trenutačnog levela igrača, *currentXP* kao provjeru trenutačnog *xp*-a. *XpToLevelUp* je maksimalan *xp* koji se može dobiti. *GameObject[]* je javna lista objekata koje smo stavili na skriptu preko *editora* i koje treba deaktivirati.

Kod 8

```
    public float maxLevel = 50;
    public float currentLevel = 1;
    public float currentXP = 0;
    public float xpToLevelUp = 100;

    public GameObject[] objectsToDisable1;
```

```
    public GameObject[] objectsToDisable2;
    public GameObject[] objectsToDisable3;

    public GameObject[] objectsToDisable4;
```

Na kodu 9. imamo funkciju *GainXP* koja prima vrijednost izvan skripte od protivnika. Tako što u *enemyHealth* skripti provjeravamo dali je objekt i dalje postojeć i dali su njegovi životni bodovi jednaki ili manji od nule. U slučaju da su manji od nule, šaljemo vrijednost dvadeset na *GainXP* funkciju. Kad primimo zadanu vrijednost nadodajemo ju na trenutačno iskustvo igrača i izjednačimo trenutačnu vrijednost s novom. Preko *while* provodimo funkciju *LevelUp* tako što gledamo dali je trenutačni *XP* veći ili jednak od maksimalnog i jeli trenutačni *level* manji od maksimalnog. U *levelXP* skripti nam nije potrebna *start* funkcija da bi započeli funkciju jer su nam sve varijable i funkcije javne. Javne varijable se mogu van skripte modificirati.

Kod 9

```
    public void GainXP(float amount)
    {
        currentXP += amount;

        while (currentXP >= xpToLevelUp && currentLevel <
maxLevel)
        {
            LevelUp();
        }
    }
```

LevelUp funkcija nam nadodaje *level* jer su već uvjeti ispunjeni za *levelUp.* Resetira *xp* tako što oduzima maksimalnu vrijednost što u isto vrijeme nam ostavlja preostali neiskorišteni *xp* za sljedeći *level*. *DebugLog* se koristi za poruke u terminalu koje samo *developer* vidi i u ovom slučaju nam *if* funkcija služi da *developer* vizualno utvrdi da je komanda prošla i da je aktivna. Kod 10.

Kod 10

```
    private void LevelUp()
    {
        currentLevel++;
        currentXP -= xpToLevelUp; // Subtract the XP required
for leveling up
        Debug.Log("Congratulations! You've reached level " +
currentLevel + "!");

        if (currentLevel < maxLevel)
        {
            Debug.Log("XP required for the next level: " +
(xpToLevelUp-currentXP));
        }
    }
```

FixedUpdate radi isto što i *Update* jedina razlika je što *Update* radi svaki *frame* u igri dok *FixedUpdate* radi svakih određenih sekundi. U ovom slučaju radi za nešto malo više od par *frameova.* Što znači da je više optimiziran i ne preopterećuje aplikaciju s manje bitnim provjerama. Uzimamo i mijenjamo *LevelNumberText* skriptu, da prikaže korisniku trenutačan *level playera.* Šaljemo vrijednost *currentLevel* i prikazujemo ju umjesto teksta na ekranu. Provjere za deaktiviranje objekata se vrše s *if* funkcijom te provjerava trenutačni *level* i zadani broj, ako je istinit uvjet pokrećemo funkcije *DisableGameObject*. Kod 11.

Kod 11

```
    void FixedUpdate()
    {
        // Update the level text using TextMeshPro
        LevelNumberText levelNumberText =
FindObjectOfType<LevelNumberText>();
        if (levelNumberText != null)
        {
            levelNumberText.UpdateLevelText(currentLevel);
        }

        if(currentLevel >= 4)
```

```
        {
            DisableGameObjects1();
        }
        if(currentLevel >= 12)
        {
            DisableGameObjects2();
        }
        if(currentLevel >= 20)
        {
            DisableGameObjects3();
        }
        if(currentLevel >= 50)
        {
            DisableGameObjects4();
        }


    }
```

Nakon provjerenih uvjeta u *FixedUpdate*u deaktiviramo objekte koji su stavljeni pod funkcijama na kodu 12.

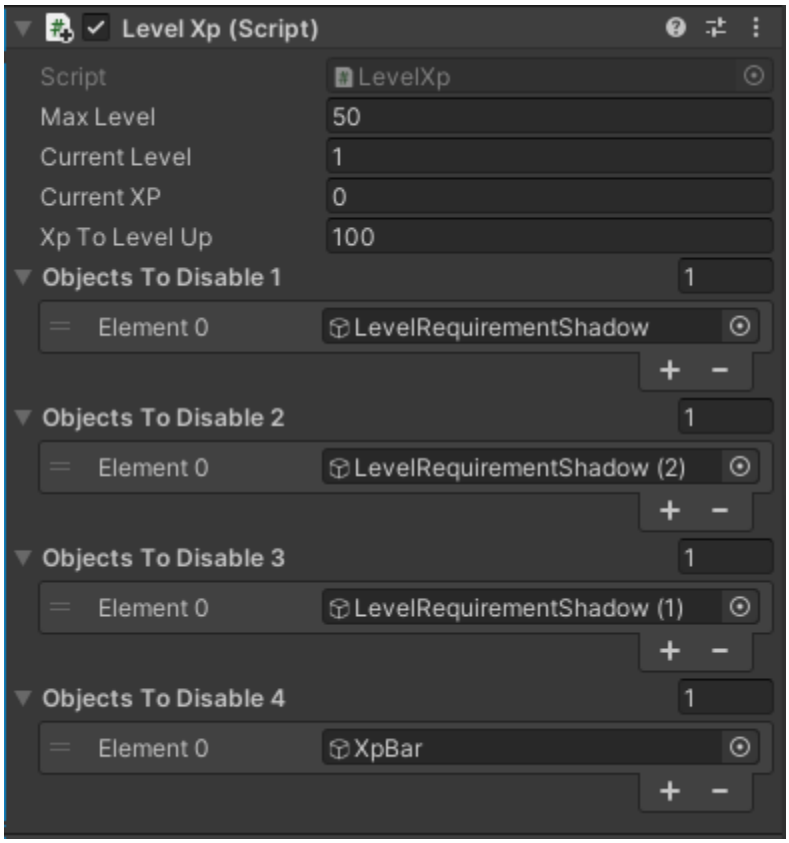
Kod 12

```
    public void DisableGameObjects1()
    {
        foreach (GameObject obj in objectsToDisable1)
        {
            obj.SetActive(false);
        }
    }
    public void DisableGameObjects2()
    {
        foreach (GameObject obj in objectsToDisable2)
        {
            obj.SetActive(false);
        }
    }
    public void DisableGameObjects3()
    {
        foreach (GameObject obj in objectsToDisable3)
        {
            obj.SetActive(false);
        }
    }
    public void DisableGameObjects4()
    {
```

```
        foreach (GameObject obj in objectsToDisable4)
        {
            obj.SetActive(false);
        }
    }
```

Objekte stavljamo u *editoru* pod skriptu u kojoj smo ih javno definirali, kao što je prikazano na slici 25.

Slika 25- LevelXp skripta u editor-u



9.2. FILLBAR I UI SKRIPTE

UI korišten za prikaz informacija korisniku poput iskustva i životnih bodova koristi *slider* objekt i skriptu za popunjavanje informacija, od varijabli na *slidere* kao vizualni prikaz životnih bodova i iskustva. Primjer na kodu 13. je skripta za ažuriranje *slidera* o životnim bodovima igrača. U navedenoj skripti koristimo knjižnicu programskog alata Unity za UI da bi preko skripte mogli mijenjati stanje

UI-a.

Kod 13

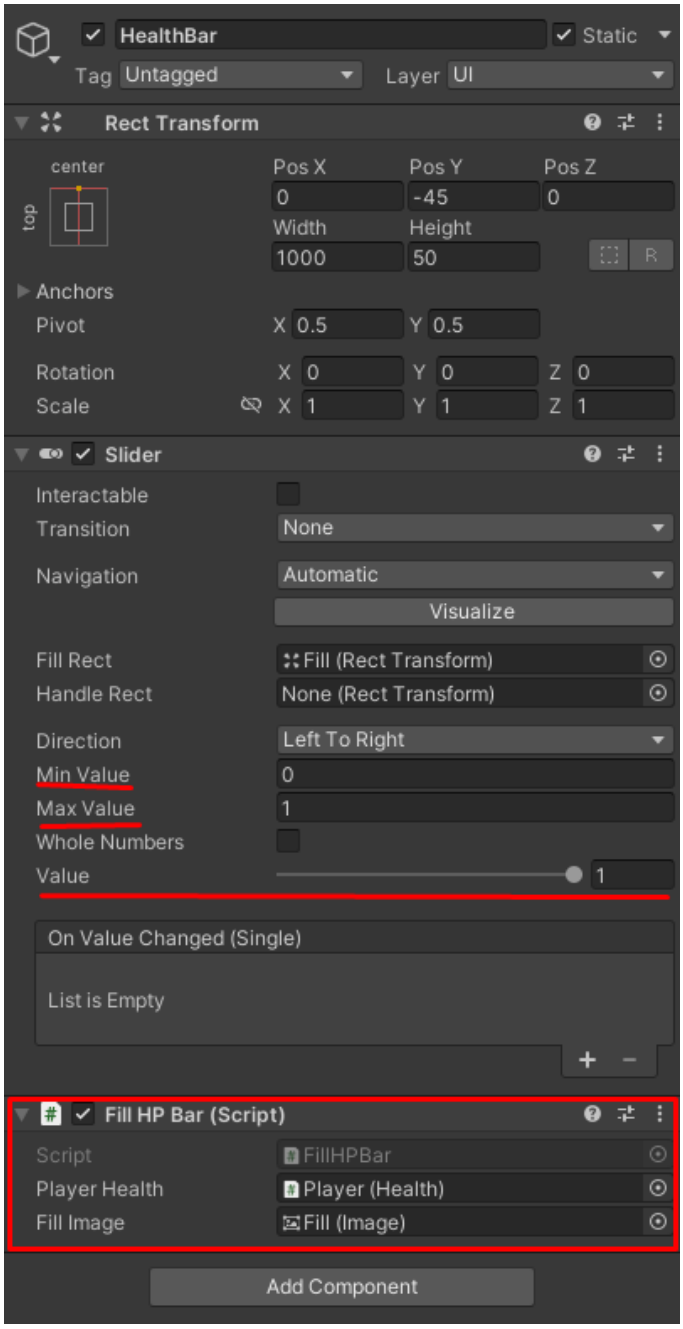
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class FillHPBar : MonoBehaviour
```

Varijable korištene u *FillHPBar* skripti su sve javne (*public*). Radi jednostavnijeg referenciranja potrebnih objekata i slika, kao što je prikazano na slici 26. i kodu 14. Koristimo referencu na *playerHealth* kako bi uzeli trenutačnu vrijednost životnih bodova igrača. Tražimo referencu *image* da se ručno pridruži skripti i da se može modificirati izgled. Tražimo referencu na komponentu *slider* kao vrijednost koju mijenjamo.

Kod 14

```
    public Health playerHealth;
    public Image fillImage;

    private Slider slider;
```

Slika 26- Primjer HealthBar-a u Canvasu



Koristimo funkciju *Awake* da bi tijekom pokretanja aplikacije skripta automatski našla i referencirala se na *slider* kao što je prikazano u kodu 15.

Kod 15

```
    void Awake()
    {
        slider = GetComponent<Slider>();
    }
```

Stavljamo glavnu logiku ažuriranja podataka o igraču u *update* funkciju koja se provodi svaki *frame* pošto je vrlo bitna informacija korisniku. U *update* funkciji stavljamo logiku provjere vrijednosti *slidera minValue*, *maxValue* i *Value* kao što je prikazano na kodu 16. Prvo definiramo kad se vrijednost puni tako što provjeravamo jeli trenutačna vrijednost *slidera* manja ili jednaka od minimalne. Ako je tvrdnja istinita deaktiviramo mogućnost punjenja i kao kontra gledamo dali je trenutačna vrijednost veća od minimalne. Ako je *fillImage* deaktiviran onda provodimo komandu za aktiviranje *fillImagea*, logiku *fillImagea* je potrebno provesti na ovaj način tako da aplikacija prepoznaje limite *slider*-a. Što je u ovom slučaju vrijednost od nula do jedan, u prevedenom značenju to je provjeravanje postotka trenutačnih životnih bodova a i ne sami broj bodova. Definiramo varijablu *fillValue* koja predstavlja trenutačne životne bodove, ali u postotcima tako što dijelimo trenutačni *hp* igrača i maksimalni *hp* igrača. Spremamo vrijednost u *fillValue* i poslije je nadodajemo da je to trenutačna vrijednost *slidera*. Za vizualni prikaz manjka i viška životnih bodova korisniku nadodajemo, da ako je manje životnih bodova od 33.33 posto onda pretvaramo boju *Hpbara* u žutu ili ako je veće od 33.33 posto pretvaramo u crvenu. Što olakšava prepoznavanje razine životnih bodova. Primjer kod 16. Za *slidere* u *canvasu* koristimo sličnu logiku skripte kao što je *FillHpBar* jedina razlika je korištena vrijednost.

Kod 16

```
    void Update()
    {
        if(slider.value <= slider.minValue)
        {
```

```
            fillImage.enabled = false;
        }
        if(slider.value > slider.minValue &&
!fillImage.enabled)
        {
            fillImage.enabled = true;
        }

        float fillValue = playerHealth.currentHealth /
playerHealth.maxHealth;


        if(fillValue <= slider.maxValue / 3)
        {
            fillImage.color = Color.yellow;
        }
        else if(fillValue > slider.maxValue / 3)
        {
            fillImage.color = Color.red;
        }

        slider.value = fillValue;
    }
```

Za ažuriranje teksta u brojeve na *canvasu* potrebno je referencirati tekst (*text*) koji želimo modificirati i ažurirati ga s vrijednosti koju želimo. Koristimo *TMPro* knjižnicu za modificiranje *TMPro* teksta, primjer na kodu 17. Od varijabli koristi se samo referenca na tekst, kod 18.

Kod 17

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using TMPro;

public class LevelNumberText : MonoBehaviour
```

Kod 18

```
    public TextMeshProUGUI levelText;
```

Za povlačenje reference na definiranu varijablu potrebno je u *awake* funkciji pozvati potrebnu komponentu, kod 19.

Kod 19

```
    private void Awake()
    {
        levelText = GetComponent<TextMeshProUGUI>();
    }
```

Koristi se trenutačna vrijednost *levela* iz skripte *LevelXp* i provodi se kroz *string* da bi broj prikazalo kao tekst na *TextMeshu*. Da bi se broj preveo u tekst potrebno je nadodati *ToString* na vrijednost *levela*, kod 20. Da bi prikazali ostale vrijednosti poput preostalih protivnika, valova preživjelih isto je potrebno referencirati se na izabranu trenutačnu vrijednost i prevesti ju u *string*.

Kod 20

```
    public void UpdateLevelText(float level)
    {
        levelText.text = level.ToString();
    }
```

10. OPTIMIZACIJA APLIKACIJE

Kod *export*-a aplikacije potrebno je obratiti pažnju na podršku Android uređaja i verzija Android-a na uređajima. Smatra se da mobilni uređaji ispod Android verzije 10 imaju u prosjeku 64gb prostora, ša je potrebno obratiti pažnju i na veličinu aplikacije. U sirovom formatu aplikacije mogu vrlo lako prijeći potreban prostor od 5gb do 10gb, pa ih je potrebno provesti kroz neku vrstu softverske kompresije. Programski jezik Unity za Android koristi tri osnovne metode *export*-a. One su Default, LZ4 i LZ4HC. Za trenutačnu aplikaciju koristi se LZ4 pošto je još u *development* stanju jer je LZ4 najbrža metoda *export-a* i testiranja aplikacija. Kad aplikacija postane spremna za *release build* prelazimo na metodu LZ4HC. LZ4HC je vrlo intenzivna metoda kompresije koja radi na *chunk-based* algoritmu. Gdje se samo vezane grupe objekata glavnog objekta dekompresiraju. Dekompresija se

dešava tijekom rada i nema vremenskog čekanja da se *build* aplikacije (*bundle*) dekompresira prije korištenja (Unity Docs, 2023).

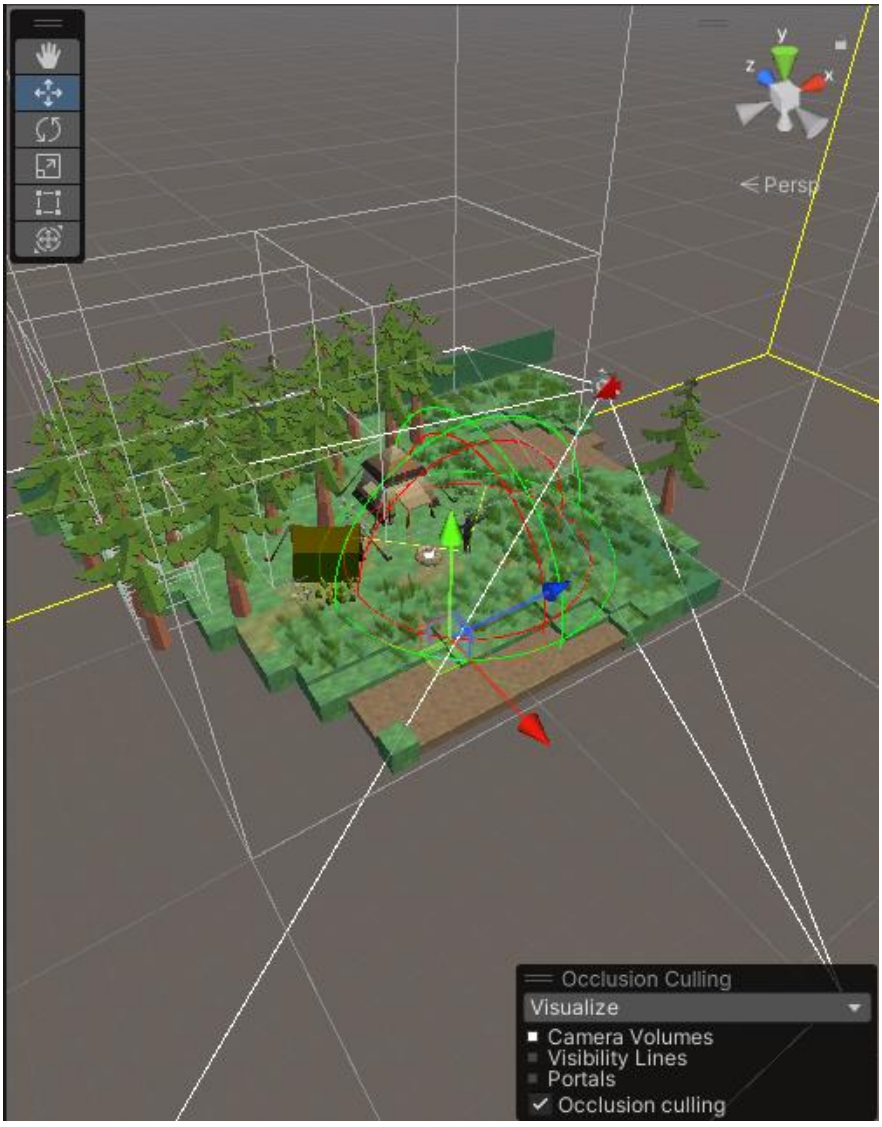
Aplikacija je podržana od verzije Android-a 8.1 do najnovijeg i potrebno je fokusirati optimizaciju oko uređaja koji se nalaze na starijim verzijama Android-a. Optimizacija se vrši na više načina, jedni od glavnih su *batching*, *occlusion*, *gpu instanciranje*, *mesh combiner* i kompresiranje tekstura. *Batching* i *occlusion* su odmah dostupni kao regularne opcije za aplikaciju. *Batching*(slika 28.) funkcionira tako da uzima sve definirane *static* objekte (neaktivne, bez funkcije) s istom teksturom, materijalima, *mesh* oblikom i stavlja ih u jednu grupu gdje se *renderiraju* s jednom definicijom objekta (definicija poput materijala, tekstura i *mesh* to jest *prefaba)*. Bez *batchinga* svaki objekt bi se isponova prikazao (*renderao*) posebno sa svojom definicijom, premda je ta definicija jednaka na drugim objektima. Što bi stvaralo preveliko opterećenje na procesoru. *Occlusion* koristi poziciju kamere i ono što glavna kamera vidi kao prostor koji je potreban prikazati (*renderati*). Pošto je to jedini prostor koji igrač vidi. Umjesto da aplikacija *rendera* čitavu scenu odjednom što bi bilo intenzivno na grafičkom procesoru, *occlusion rendera* pogled kamere kao što je na slici 27. Da bi *occlusion* radio na objektima potrebno ih je definirati da su *Occluder static* i *Occludee static* što bi značilo da su stavljeni u razdvojene grupe objekata koji su maknuti i prikazani.

Kako bi programski jezik Unity znao gdje se nalazi koji objekt tijekom *renderiranja* on sve objekte koji nisu u okviru igrača za *renderiranje* sprema kao informaciju objekta u memoriju. Što znači da svi *ne-renderirani* objekti se sastoje od informacija kako bi taj objekt trebao izgledati, što bi trebao imati i gdje se nalazi.

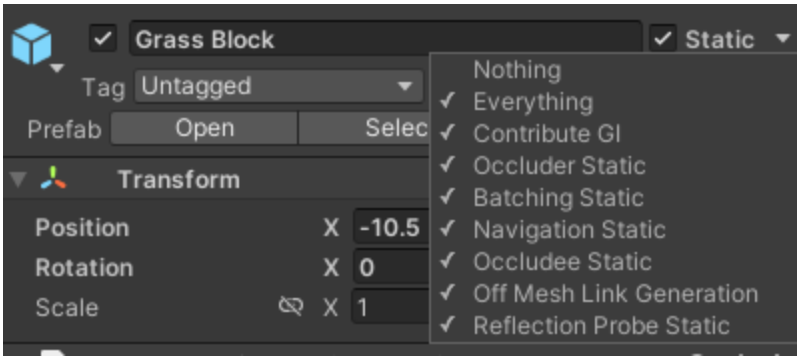
Osim kompresiranja same veličine aplikacije imamo i kompresiranje rezolucija tekstura. Teksture mogu biti vrlo intenzivne za mobilni grafički procesor pošto mogu biti u vrlo velikoj rezoluciji. Prosječna tekstura je u rezoluciji

2048x2048. Prosječni broj objekata u aktivnoj sceni je bliže 100. Što bi značilo da grafički procesor mobitela bi morao prikazivati (*renderati*) svaki objekt posebno po rezoluciji koja je malo veća od 2k. Prosječna tekstura po objektu bi uzimala 1mb do 2mb, što postaje preveliko i preteško za mobilni *hardware* da prikazuje *(rendera)*. Da bi se riješio problem prevelike aplikacije i grafički pre intenzivne aplikacije, teksture moramo kompresirati ili spojiti *mesh* terena i posebno postaviti teksture. U ovom slučaju smo smanjili rezoluciju teksture i način prikazivanja (*renderanja*) materijala objekta.

Slika 27- Primjer Occlusion-a



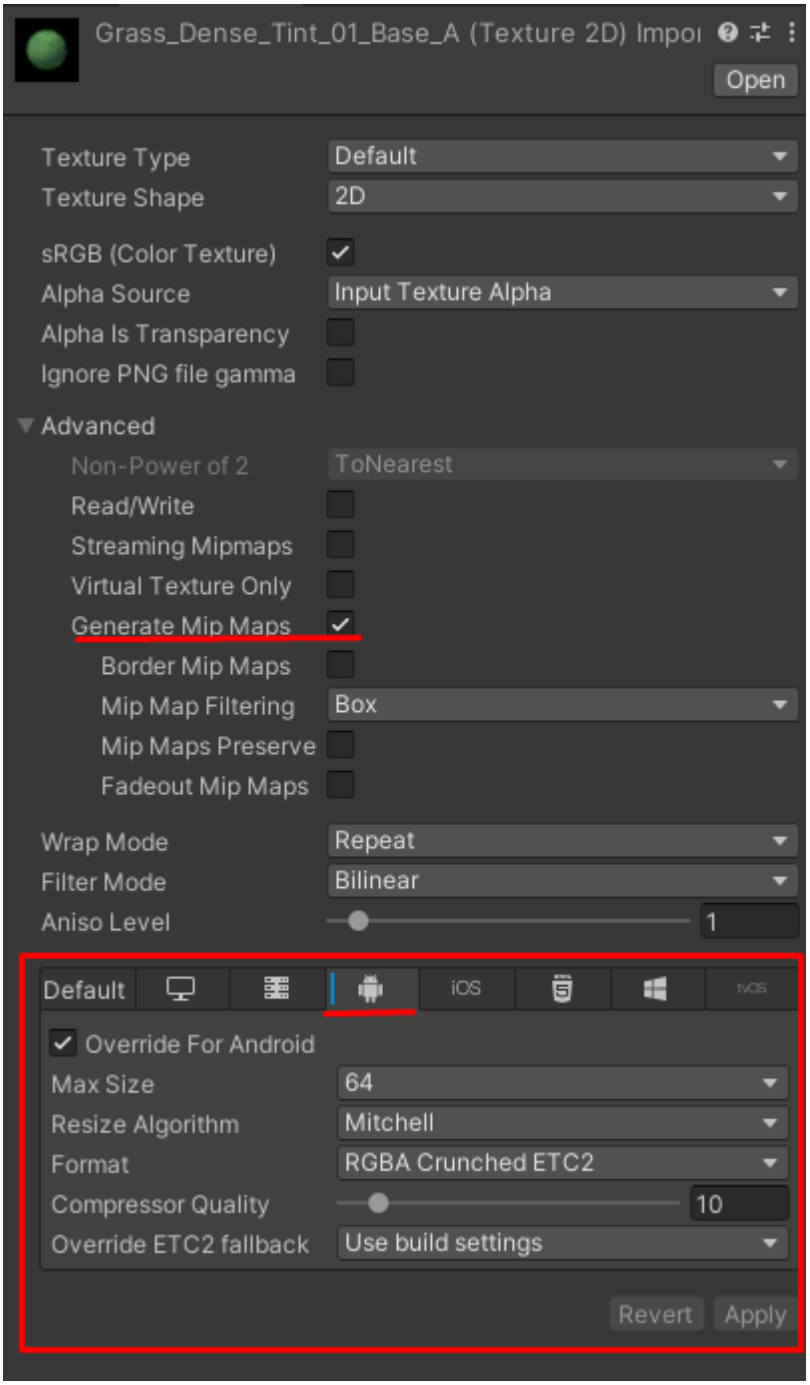
Slika 28- Primjer Static objekta



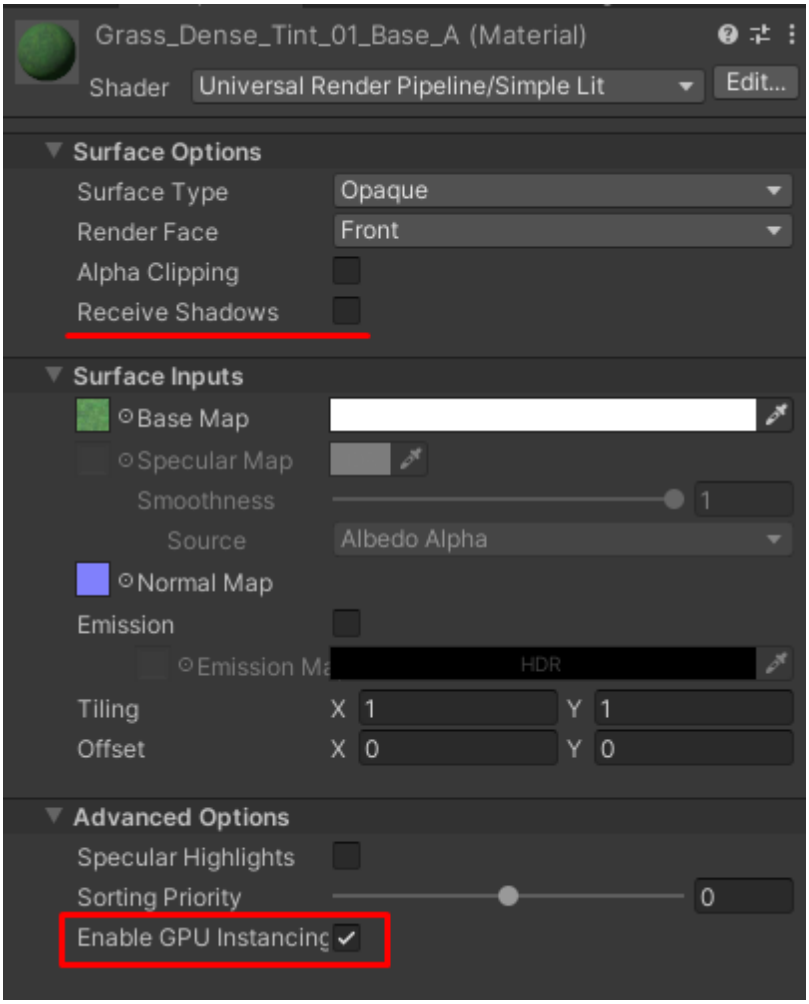
Da bi rezoluciju teksture smanjili moramo *overrideat* Android postavke za teksturu (slika 29.). Sa 2048 rezolucije smanjujemo na 64 kao maksimalnu rezoluciju. Uključujemo generiranje *mip* mapa i stavljamo najčešći način formata za Android kao RGBA *crunched* ETC2. Kvalitete kompresije su postavljenje na deset posto. Primjenjujemo način kompresije tekstura na sve objekte u sceni što kao rezultat smanjuje veličinu aplikacije za 150% i povećava performancu za 30*fps*-a.

Materijale je potrebno također modificirati tako da se isključe sjene koje mogu primati i uključi *gpu instancing* (slika 30.). *Draw calls* je metoda slanja zahtjeva za prikazivanje (*renderanje*) geometrije objekata na API od grafičkog procesora (Unity Docs, 2023). U sceni imamo opciju da vidimo performansu tijekom rada aplikacije i tamo nam se nalazi *draw calls. Draw calls* se vrlo lako može nakupiti i stvoriti problem performanse za aplikaciju. Da bi smanjili *draw calls* potreban nam je *gpu instancing*. *Gpu instanciranje* radi tako da prikaže (*rendera*) više objekata s istim *mesh*-om kao jedan *draw call* (Unity Docs, 2023). Što u sceni aplikacije smanjuje *draw calls* za duplo i povećava performansu aplikacije dok u isto vrijeme smanjuje teret na procesoru i grafičkoj.

Slika 29- Primjer kompresije Textura

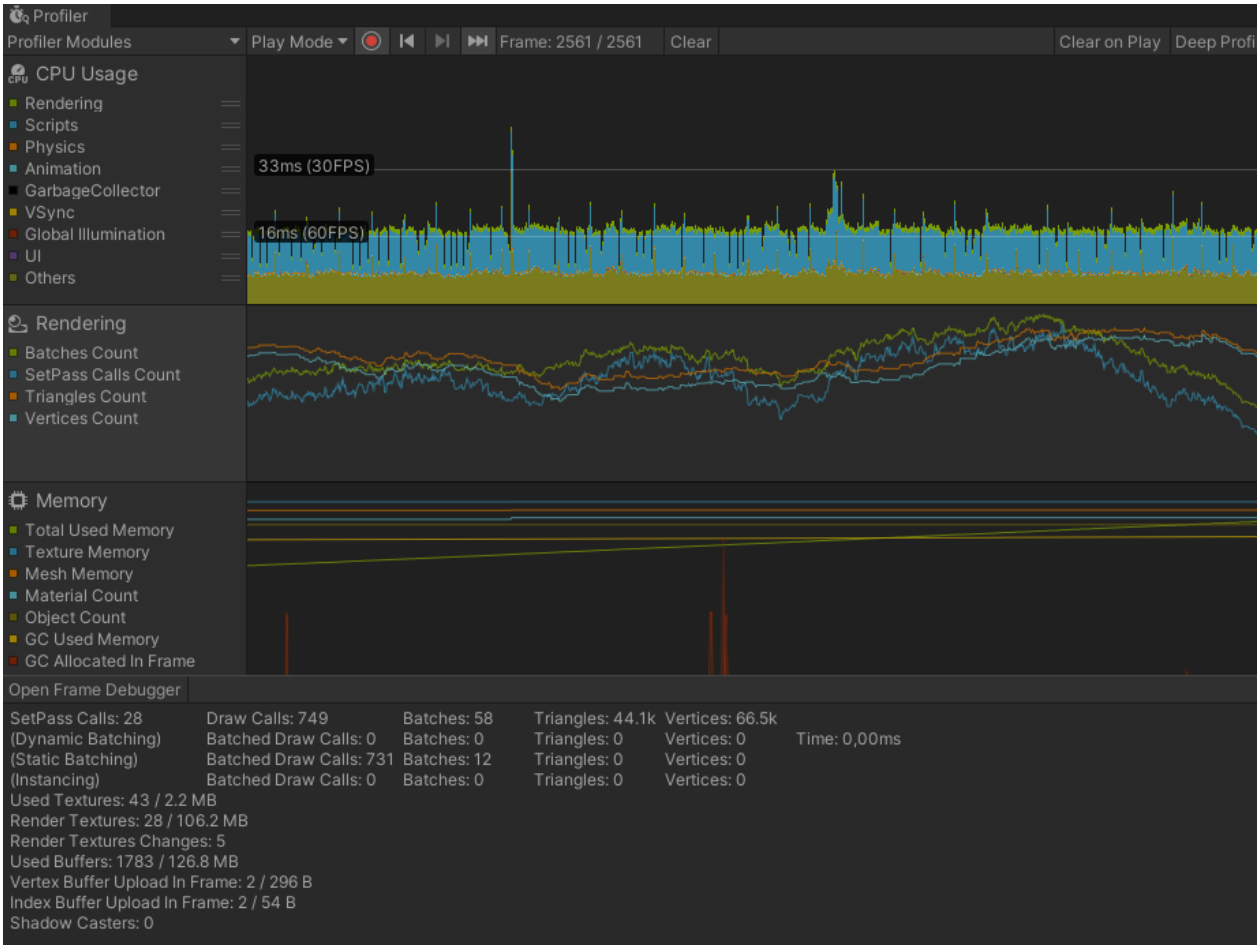


Slika 30- Primjer optimizacije materijala



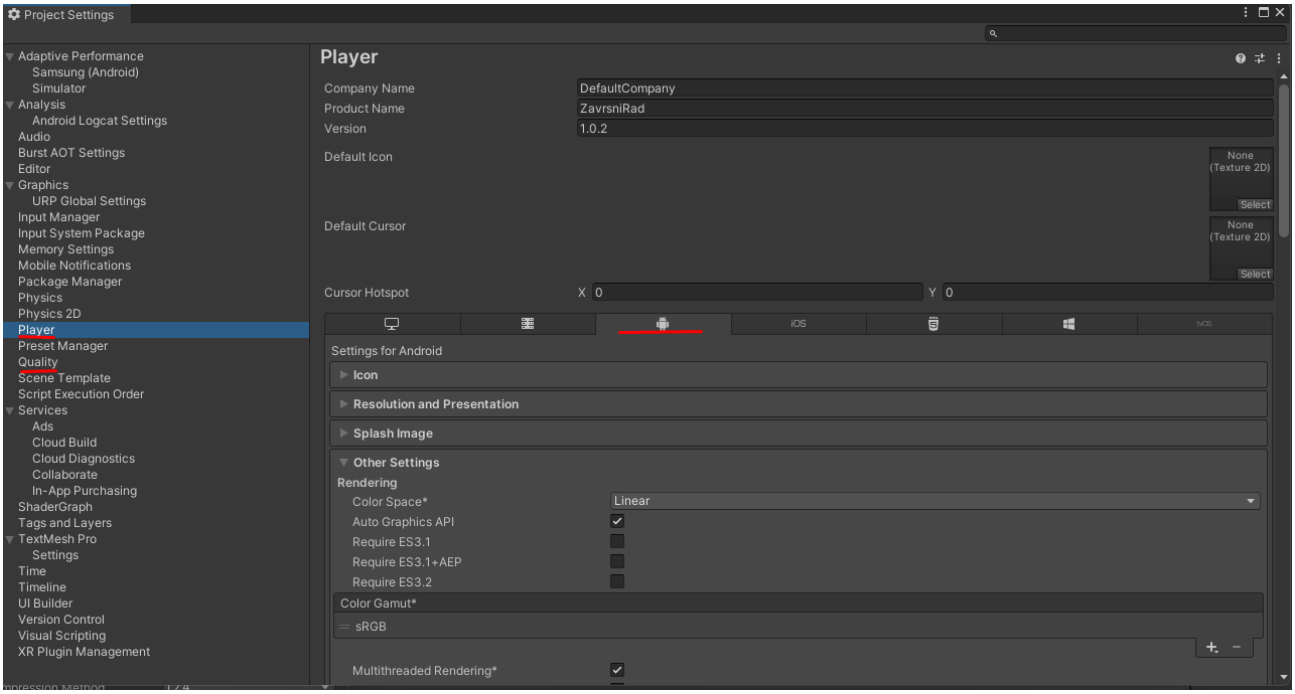
Da bi se saznalo gdje nam je aplikacija preopterećena potrebno je koristiti alat za profiliranje (*profiler).* Alat prikazuje resurse zadanog uređaja i gdje se oni koriste, u kojoj količini (Unity Docs, 2023). *Profiler* pokrećemo u *editor*-u ili na spojenom uređaju za prikaz performanse tijekom rada. Tokom uočavanja problema, *profiler* automatski prikazuje koliko se resursa koristi i gdje se nalazi problem. U slučaju trenutačne aplikacije najveće opterećenje su skripte na procesoru i prevelika količina trenutačnih objekata na ekranu. Oba slučaja je moguće optimizirati, no pošto aplikacija već sama po sebi dobro radi i nema curenja memorije (*memory leak),* zadovoljavajuće je za rad. Primjer *profiler*-a možemo uočiti na slici 31.

Slika 31- Primjer profiler-a



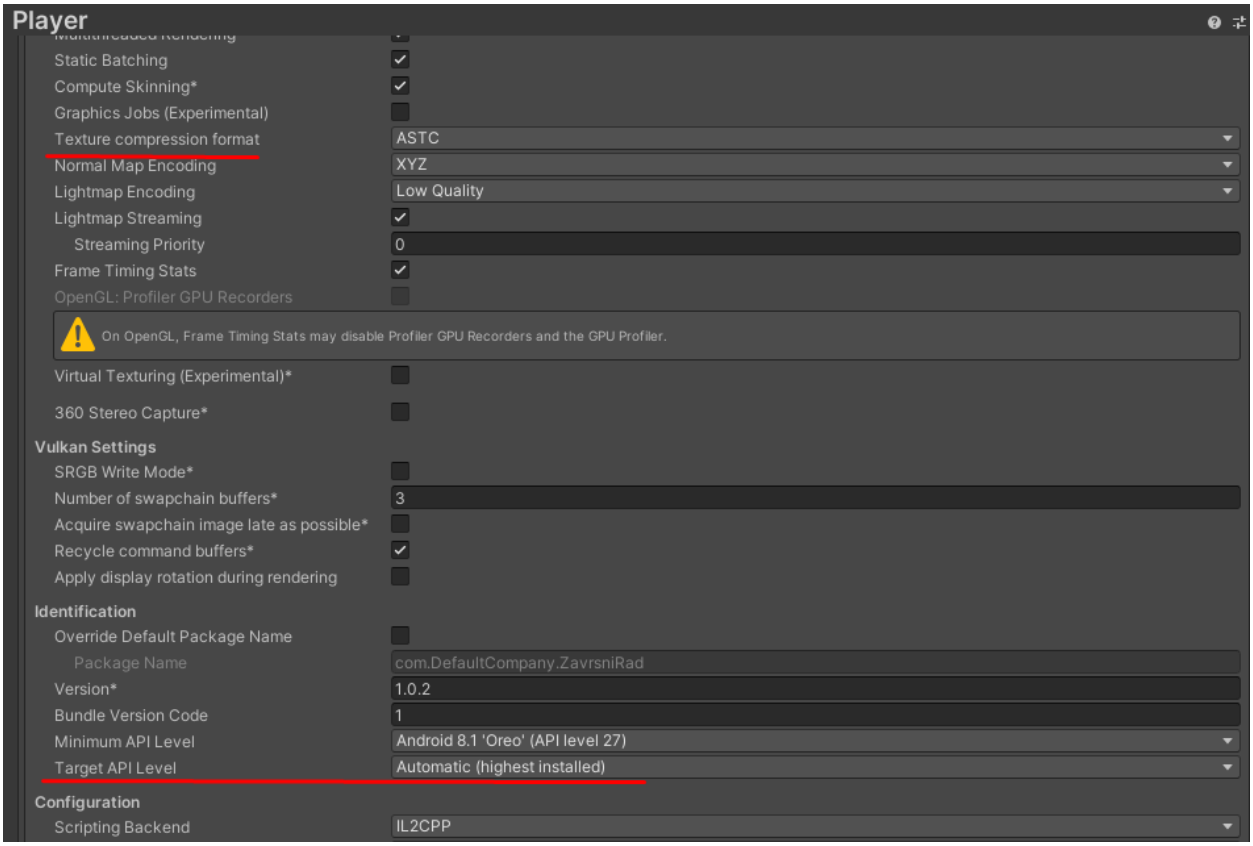
Player i *quality settings* pod *project settings* služi za dodatno i detaljnije optimiziranje aplikacije. *Player settings* se koristi za način kompresiranja, što će aplikacija od dostupne tehnologije koristiti, koji način *scripting*-a u *backend*-u će koristiti, na kojim uređajima i verzijama Android-a je dostupna (slika 32.). Izaberemo koju platformu koristimo i definiramo izgled i podržanost aplikacije.

Slika 32- Primjer project settings-a, player i quality



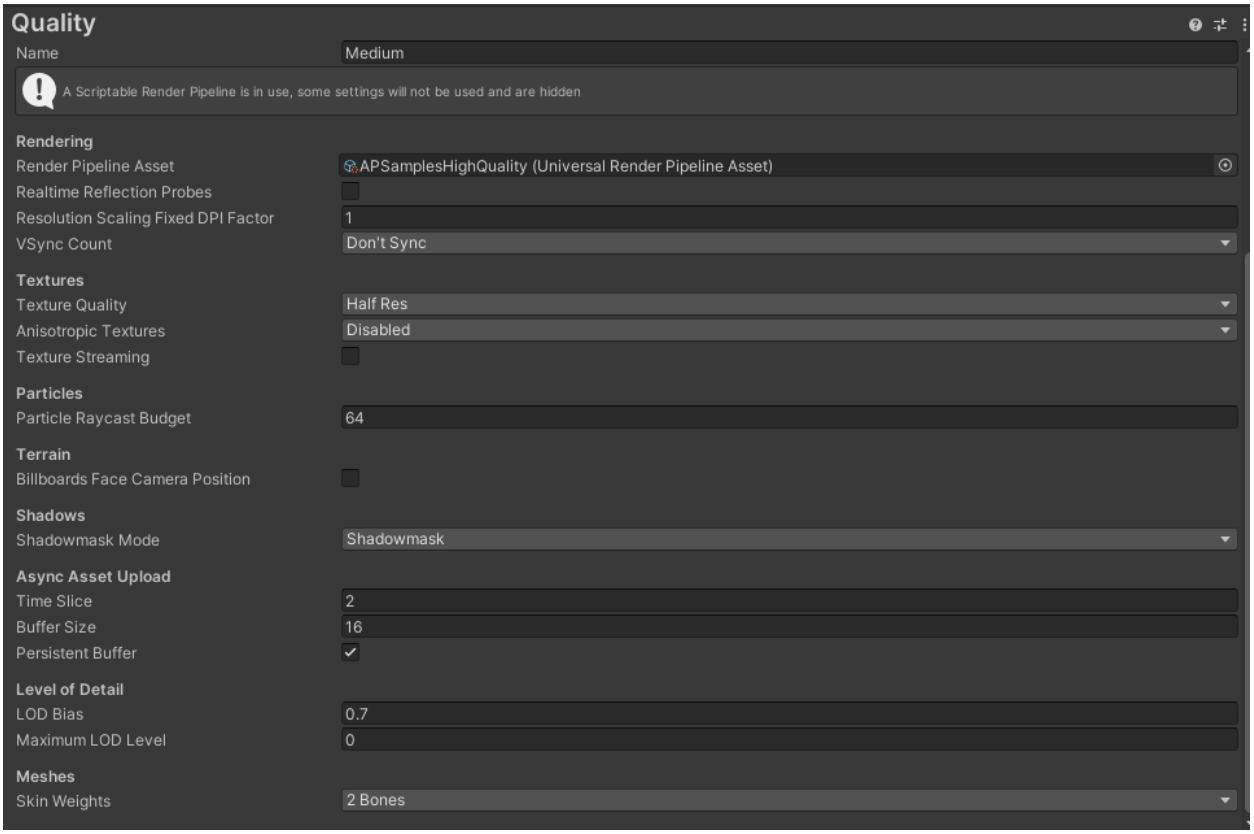
Za teksture koristi se format ASTC kao osnovna opcija za Android (slika 33.) i pretpostavljamo koje verzije Android-a podržavaju aplikaciju.

Slika 33- Primjer Player opcija



Opcije kvalitete se koriste za definiranje *render pipelinea*, koji koristimo u aplikaciji. *Render pipeline* je isto kao URP (*Universal Render Pipeline*) te ga je moguće po volji modificirati za traženi izgled i performancu aplikacije (slika 34.).

Slika 34- Primjer Quality opcija



11. ZAKLJUČAK

Programski jezik Unity jedan je od najboljih softvera za započeti izradu aplikacije (*development)*, radi svojih mnogih dokumentacija, videozapisa i tečajeva. Nude mnoge tečajeve za programski jezik C# te podržava mnoge platforme.

Izgrađen je za projekt jednostavan koncept *Roguelike* videoigre, koristeći se mnogim dostupnim alatima za definiranje izgleda i logike aplikacije. Napravljena je mapa otvorenog svijeta (*open world)* gdje igrač može manevrirati tijekom borbe te opcionalno istraživati detalje po mapi.

Sa jednostavnim konceptom dostupnih moći igraču, mehanikom *levela* i životnih bodova koji omogućuju napredak igrača kroz nivoe protivnika. Izgrađena logika protivnika i valova napada na igrača s postepenim povećavanjem težine igre.

Fokus projekta je jednostavna igra sa što efikasnijim i bržim dosezanjem cilja. Dostupna je na mnogim starijim i novim uređajima s ciljem performanse i užitka tijekom igranja.

Proučeno je mnogo novih načina programiranja za optimizaciju programski jezik C# skripti i alata za optimizaciju aplikacije, tehnologija poput *Occlusion*, *Batching* i *Profiler*-a.

Za pojednostavljenje *developmenta* igre primijenjeni su *mixamo* animacije i modeli, teksture , materijali i *prefabi* objekata s *stora* programskog alata Unity.

Za budući rad bi trebalo obratiti pažnju na pisanje koda u programskom jeziku C# preko skripti, s fokusom na pojednostavljenje kompleksnosti koda za bolje kompajliranje skripti. Također biti oprezan s kompleksijom *mesh-a* objekta i tekstura na objektu.

LITERATURA

Android. (16. 7 2022). *android*. Dohvaćeno iz android.com: https://www.android.com/what-is-android/

Dealessandri, M. (16. 1 2020). *Games Industry.biz*. Dohvaćeno iz www.gamesindustry.biz: https://www.gamesindustry.biz/what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you

EDY. (1. 1 2014). *EDY materials and textures from blender to Unity 3D*. Dohvaćeno iz edy.es: https://www.edy.es/dev/docs/materials-and-textures-from-blender-to-unity-3d/

EVE Online. (1. 1 2023). *EVE ONLINE*. Dohvaćeno iz eveonline.com: https://www.eveonline.com/fr/news/view/the-many-additions-of-v5plusplus

FAUguy. (17. 8 2011). *Phone Arena*. Dohvaćeno iz phonearena.com: https://www.phonearena.com/news/Googles-Android-OS-Past-Present-and-Future_id21273

Freese, T. (16. 12 2020). *PBS LearningMedia*. Dohvaćeno iz pbslearningmedia.org: https://www.pbslearningmedia.org/resource/tessellations/what-is-a-tessellation/

GitHub. (4. 7 2023). *GitHub*. Dohvaćeno iz github.com: https://github.com/microsoft/TypeScript/graphs/contributors

GStatic. (1. 1 2023). *gstatic*. Dohvaćeno iz tbn2.gsstatic.com: https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9GcQgcN_8houzjxaMF6rpreHTvUv2xOk_SrMk8ZlywGTPsFHHpOCz

Microsoft. (1. 1 2006). *Microsoft*. Dohvaćeno iz microsoft.com: https://www.microsoft.com/presspass/exec/techfellow/hejlsberg/default.mspx

Microsoft. (5. 4 2023). *Microsoft Documentation*. Dohvaćeno iz learn.microsoft.com: https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/

Robertson, A. (3. 3 2015). *The Verge*. Dohvaćeno iz theverge.com: https://www.theverge.com/2015/3/3/8142099/unity-5-engine-release

Unity. (1. 1 2023). *Unity*. Dohvaćeno iz unity.com: https://unity.com/

Unity. (1. 7 2023). *Unity Documentation*. Dohvaćeno iz docs.unity3d.com: https://docs.unity3d.com/Manual/Materials.html

Unity. (1. 1 2023). *Unity Visual Scripting*. Dohvaćeno iz unity.com: https://unity.com/features/unity-visual-scripting

Unity Docs. (1. 7 2023). *Unity Documentation*. Dohvaćeno iz docs.unity3d.com: https://docs.unity3d.com/ScriptReference/BuildCompression.LZ4.html#:~:text=LZ4(HC)%20is%20a%20%E2%80%9C,to%20be%20decompressed%20before%20use

Unity Docs. (1. 7 2023). *Unity Documentation*. Dohvaćeno iz docs.unity3d.com: https://docs.unity3d.com/Manual/optimizing-draw-calls.html

Unity Docs. (1. 7 2023). *Unity Documentation*. Dohvaćeno iz docs.untiy3d.com: https://docs.unity3d.com/Manual/GPUInstancing.html

Unity Docs. (1. 7 2023). *Unity Documentation*. Dohvaćeno iz docs.unity3d.com: https://docs.unity3d.com/Manual/Profiler.html

Mixamo, https://www.mixamo.com/#/ (pristup: 20.06.2023.)

Poliigon, https://www.poliigon.com/search?credit=0 (pristup: 14.06.2023.)

Unity Asset Store, https://assetstore.unity.com/ (pristup: 25.05.2023.)

Unity Documentation, https://docs.unity.com/ (pristup: 20.05.2023.)

Unity, https://unity.com/ (pristup: 20.05.2023.)

TABLICA SLIKA