Izrada 3D računalne igre u Unity razvojnom okruženju

Arlović, Matej

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Preddiplomski stručni studij

Izrada 3D računalne igre u Unity razvojnom okruženju

Završni rad

Matej Arlović

Osijek, 2018.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac Z1S: Obrazac za imenovanje Povjerenstva za obranu završnog rada na preddiplomskom stručnom studiju

Osijek, 17.09.2018.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za obranu završnog rada na preddiplomskom stručnom studiju

Ime i prezime studenta:	Matej Arlović
Studij, smjer:	Preddiplomski stručni studij Elektrotehnika, smjer Automatika
Mat. br. studenta, godina upisa:	A4286, 23.09.2017.
OIB studenta:	53024601678
Mentor:	Doc.dr.sc. Ivica Lukić
Sumentor:	
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	Izv. prof. dr. sc. Krešimir Nenadić
Član Povjerenstva:	Doc.dr.sc. Mirko Köhler
Naslov završnog rada:	Izrada 3D računalne igre u Unity razvojnom okruženju
Znanstvena grana rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak završnog rada	Izraditi 3D računalnu igre u Unity razvojnom okruženju koja omogućuje korisniku dodavanje slika na osnovu kojih će se automatski izraditi teren za igru. Pomoću podataka sa slike odrediti će se zapreke i slobodni dijelovi terena u kojima će se glavni like računalne igre sukobljavati s neprijateljima.
Prijedlog ocjene pismenog dijela ispita (završnog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	17.09.2018.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 28.09.2018.

Ime i prezime studenta:	Matej Arlović
Studij:	Preddiplomski stručni studij Elektrotehnika, smjer Automatika
Mat. br. studenta, godina upisa:	A4286, 23.09.2017.
Ephorus podudaranje [%]:	4%

Ovom izjavom izjavljujem da je rad pod nazivom**: Izrada 3D računalne igre u Unity razvojnom okruženju**

izrađen pod vodstvom mentora Doc.dr.sc. Ivica Lukić

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.

Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

Sadržaj

1. UVOD

Unity je razvojno okruženje koje omogućava izrađivanje video igrica u 2D ili 3D grafici. Osim što nudi pokretač igrica (eng. *game engine*) Unity nudi set alata koji pomažu pri izradi video igrica kao što je lako animiranje karaktera, prilagođen grafički motor (eng. *custom graphics engine*), vizualni efekti i tako dalje. Omogućava vrlo laki izvoz video igre za više od 25 platformi od Androida do PlayStationa.

1.1. Zadatak završnog rada

Izraditi 3D računalnu igre u Unity razvojnom okruženju koja omogućuje korisniku dodavanje slika na osnovu kojih će se automatski izraditi teren za igru. Pomoću podataka sa slike odrediti će se zapreke i slobodni dijelovi terena u kojima će se glavni like računalne igre sukobljavati s neprijateljima.

1.2. Opis video igre u završnom radu

U ovome radu će se prikazati izrada 3D video igre za Android operacijski sustav. Rad će biti podijeljen na 4 poglavlja u kojima ćemo objasniti tehnologije korištene u izradi igre, probleme koji su se pojavili prilikom izrade video igre, te primjeri iz same igre od modela i grafičkih stvari do primjera koda. Glavna razlika između ove video igre i drugih je ta što će se teren za igru generirati automatski iz neke slike koju korisnik unese. Igra će posebnim algoritmima za obradu digitalnih slika obraditi unesenu sliku i na temelju toga generirati 3D model terena. Osim toga koristiti ćemo novu tehnologiju iz Unity-a koja omogućava višenitno programiranje (eng. *multithreading*) čime se potrošnja baterije i zagrijanost mobitela spušta na minimum. Višenitno programiranje će omogućiti da se digitalna slika što brže obradi kako korisnik ne bi čekao dugo.

2. TEHNOLOGIJE KORIŠTENE U IZRADI IGRE

U ovome poglavlju će se spomenuti tehnologije koje su korištene u izradi ove video igre, ali i općenito prilikom izrade 3D video igara. Za razvoj kvalitetne 3D video igre potrebno je izraditi kvalitetnu računalnu grafiku, API za prikaz računalne grafike i stvoriti unikatan ugođaj same igre, ali i pisanje kvalitetnog programskog koda koji će oživjeti tu igru. Unity3D donosi 3 API-ja koji omogućuju prikaz računalne grafike na uređaj: Direct3D, OpenGL i Vulkan. Za igru napravljenu u ovome radu odabran je OpenGL core jer Vulkan API nije komatibilan sa verzijama Androida manjim od 7.0. Važno je napomenuti da većina stvari iz računalne grafike već dolazi uz sam pokretač igrica (eng. *game engine*), te ukoliko želite unikatniji ugođaj potrebno je znati kako konfigurirati pokretač igrica.

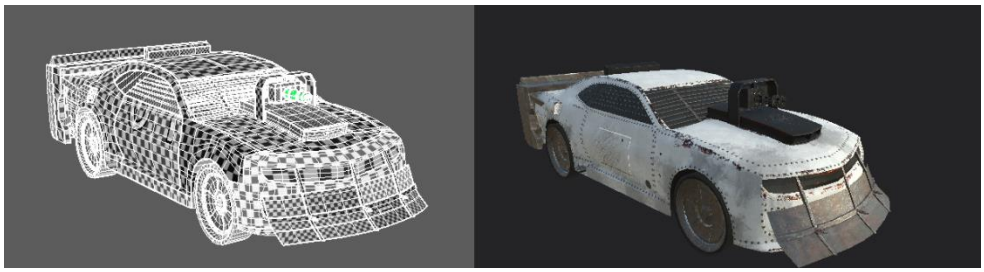
2.1. 3D modeli i modeliranje

Modeliranje 3D modela je proces kreiranja virtulanog trodimenzionalnog objekta putem nekog od programa za modeliranje. 3D modeli se ne koriste samo u video igricama već i u 3D printanju, marketingu, znanosti, dizajn potpomognut računalom (CAD) i proizvodnja potpomognuta računalom (CAM). Pod modeliranje 3D modela se podrazumjeva pravljenje i uređivanje objekata, dodavanje tekstura ili boja (ovisno o tipu igre), te promjenom gradivnih svojstava. Postoji više načina modeliranje 3D modela:

- Modeliranje pomoću mnogokuteva (eng. *polygons*)
- Modeliranje korištenjem primitivnih 3D modela (kocka, piramida, stožac, cilindar, kulga)
- Modeliranje korištenjem krivulja (eng. *spline curves*)
- Modeliranje putem NURBSa – Objekti koji su definirani sa Bézier krivuljama[1]

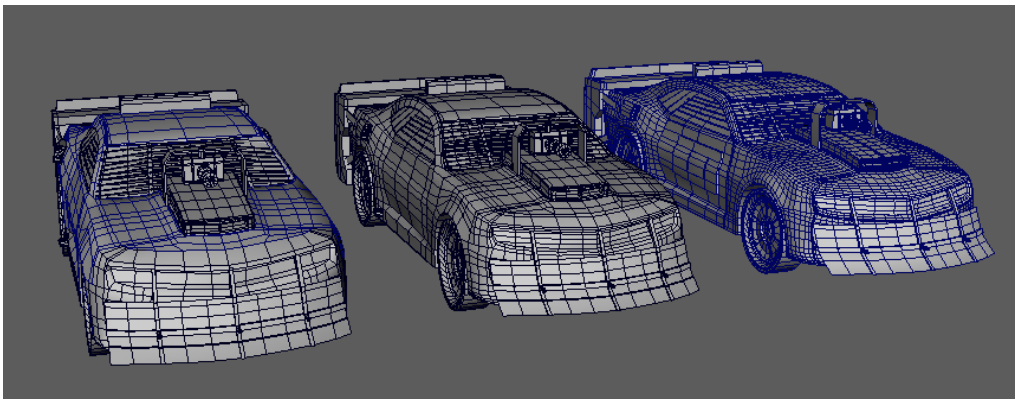
Svaki 3D model je napravljen od mnogokuta ili trokutova, što je 3D objekt kompleksniji njihov broj će se povećavati. Unutar pokretača igrica svaki se mnogokut dijeli na više trokutova i tada se svaki trokut zasebno prikazuje na igračevom ekranu. Stoga što je 3D model kompleksniji on će izgledati lijepo na igračevom ekranu, ali će zahtjevati puno jače sklopovlje (eng. hardware) koje će što brže prikazati 3D model. Da bi se igraču prikazao kompleksan objekt, a da ne šteti performansama igre tada se koriste trikovi sa normal i ambient occlusion (AO) mapama.

Nakon što se odradi 3D model objekta koji se želi staviti u video igru potrebno je dodati teksturu ili boju. Da bi se moglo teksturirati objekt potrebno je odrediti UV mape. Kreiranje UV mapa je zapravo proces projekcije 2D slike na površinu 3D modela. Slova U i V unutar naziva mapa označavaju osi koje se koriste unutar mapa.



Sl. 2.1. Prikaz UV mape i teksture na 3D modelu.

Najčešće pokretači igrica koriste UV mape unutar kvadrata u rasponu od 0 do 1. Kada se napravi 3D model potrebno je generirati UV koordinate svakog tjemena (eng. *vertex*) unutar modela. Jedan od načina je tzv. razmotavanje (eng. *unwrapping*) trokutova objekta na 2D sliku. Kada se objekt razmota tada osoba koja modelira objekt može nanijeti teksturu na isti. Potrebno je paziti da se UV mape ne preklapaju, da šavovi (eng. *seams*) ne budu preočiti, te da teksture ne budu razvučene. Zato se najčešće koristi kockasta tekstura gdje osoba koja modelira može vidjeti kvalitetu UV mape na samom objektu[2].

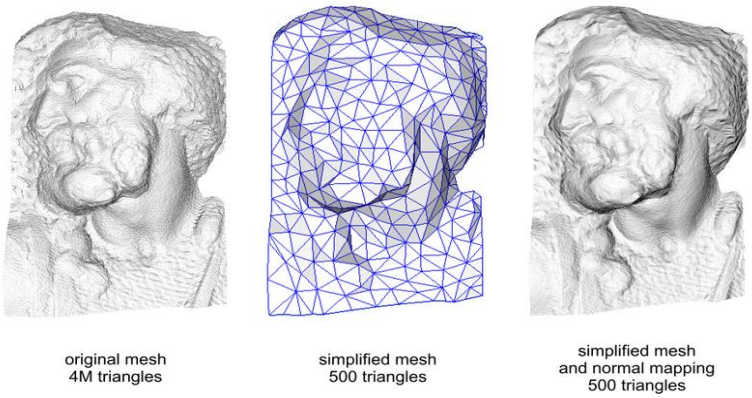


Sl. 2.2. Prikaz 3D modela od onoga sa najviše mnogokuta (desno) do onog spremnog za igru (lijevo).

3D model je skup podataka koji omogućavaju vizualni prikaz modela u računalnoj grafici (eng. *rendering*). Stoga se 3D modeli mogu napraviti u računalnim programima, ali i putem algoritama koji generiraju podatke potrebne za prikaz na ekranu (trokute, normale i UV mape).

U računalnoj igrici koja se radi u ovom završnom radu se koriste oba načina prikaza 3D modela[3].

Normal mapanje (eng. *normal mapping*) je tehnika u računalnoj grafici koja „lažira“ osvjetljenje na povišenjima i udubinama 3D modela. Time se jednostavnom modelu dodaju detalji kao kod kompleksnog modela. Da bi se dobila normal mapa potrebno ju je kreirati u programu za uređivanje slika kao što je GIMP ili Photoshop ili se može generirati iz kompleksnog objekta putem xNormal programa.



Sl. 2.3. Prikaz kompleksnog modela, jednostavnog modela (bez normal mape) i jednostavnog modela (sa normal mapom).

Ambient Occlusion je tehnika u računalnoj grafici gdje se računa koliko je izložen tjemen modela ambijentalnom svijetlu. Ambient Occlusion mape (često nazivane još i AO mapama) su 2D prikaz ambient occlusion proračuna odnosno sjena modela bez utjecaja direktnog osvjetljenja. AO mape se „peku“ (eng. *baking*) iz samog 3D modela koristeći se raspoređivanjem zraka (eng. *ray casting*) po geometriji modela[4].

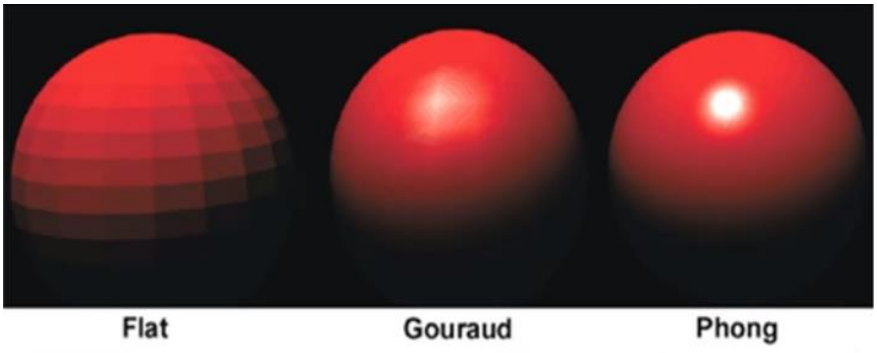
Sl. 2.4. Prikaz modela sa AO mapama (lijevo) i bez (desno).

2.2. Shaderi

Shader je računalni program koji se izvodi na grafičkom procesoru (eng. GPU) , a služi za sjenčanje (eng. shading) 3D modela tijekom prikaza 3D modela na igračevom ekranu. Oni trebaju napraviti prikladnu razinu svjetla i sjena na 3D objektu. Također se mogu koristiti za specijalne efekte i post processing efekte. Postoje četiri vrste shadera:

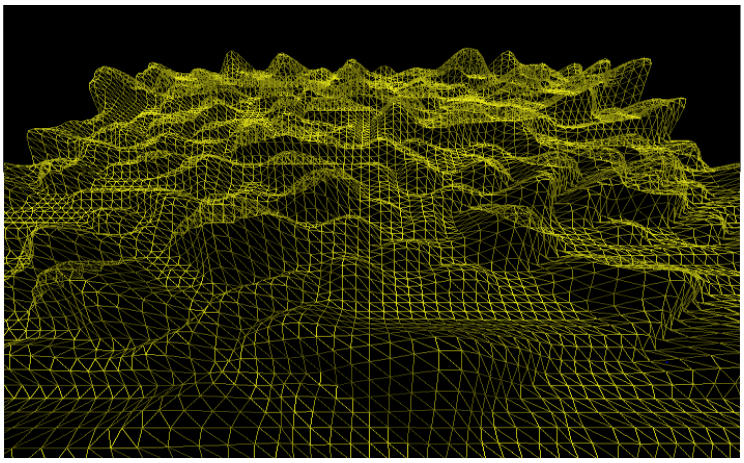
- Tjemeni shaderi (eng. *vertex shaders*)
- Geometrijski shaderi (eng. *geometry shaders*)
- Teselacijski shaderi (eng. *tessellation shaders*)
- *Compute* shaderi

Tjemeni shaderi su najčešće korišteni shaderi u svijetu video igrica. Oni se izvode jednom za svaki tjemen (eng. *vertex*) kojem opisuju svojstva tjemena: pozicija, koordinate, boja, dubinske koordinate slika unutar Z-buffera i tekstura. Njihova svrha je transformirati 3D poziciju tjemena unutar virtualnog svijeta na 2D koordinate koje se nalaze na ekranu.



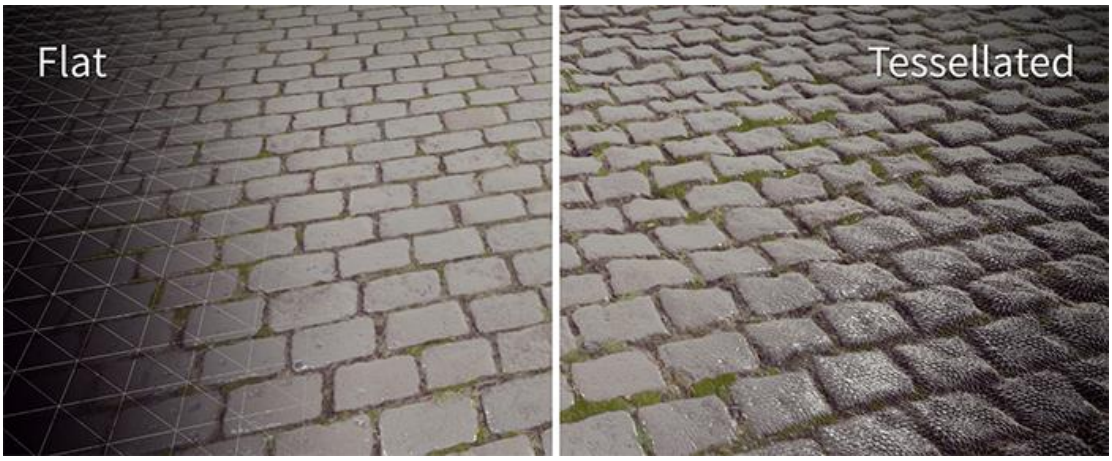
Sl. 2.5. Prikaz tjemenog shadera u različitim verzijama sjenčanja.

Geometrijski shader je nova vrsta shadera predstavljena u Direct3D 10 i OpenGL 2.0+. Ovaj shader program upravlja grafičkim primitivima kao što su: linije, trokuti i točke. Oni se izvode nakon tjemenih shadera, a kao ulaz koriste jedan ili više primitiva 3D modela s kojim se tada manipulira. Oni nisu obavezni, ali se koriste kod teselacije na 3D modelu, automatskog upravljanja kompleksnim objektima, ali i kod boljeg generiranja sjena.



Sl. 2.6. Prikaz geometrijskog shadera u prilikom generiranja terena.

Teselacijski shaderi su najnovija vrsta shadera koji su predstavljeni u OpenGL 4.0 i Direct3D 11. Oni omogućuju jednostavnim 3D modelima da se podijele u kompleksnije modele tijekom izvođenja igre (eng. *at runtime*) putem matematičke formule. Najčešće varijable unutar te formule je razina detalja (eng. *level of details*) i udaljenost kamere od objekta. Kada je kamera blizu 3D modela sa teselacijskim shaderom tada je on kompleksan i ima puno detalja, a kada je udaljena tada on bude sve jednostavniji i jednostavniji čime se performanse igre povećavaju. Jedina mana ovih shadera je ta što se ne mogu korisititi na starijim uređajima i mobitelima[5].



Sl. 2.7. Prikaz teselacijskog shadera na primjeru kaldrme.

2.3. Obrada slike

U računalnoj znanosti digitalna obrada slika je korištenje računalnih algoritama kako bi se digitalne slike obradile. Digitalnom obradom slika se želi poboljšati ili se žele izvući korisne informacije iz nje. Poboljšanjem digitalne slike želimo spriječiti nepravilnosti na njoj poput neutralizacije jake svijetlosti, uklanjanje mrtvih piksela ili šumova[6]. Svaka digitalna slika je

matrica realnih brojeva koji predstavljaju svjetlinu piksela od nula do jedan. Prilikom obrade slike ta se matrica mijenja na temelju algoritama koji se koriste za obradu slike. U ovom radu će se pisati o algoritmima za neutralizaciju osvjetljenja na slici, za pretvorbu obojane slike (eng. *RGB*) u crno bijelu sliku (eng. *grayscale*), te algoritmu za usrednjavanje crnih i bijelih piksela odnosno zamućenja piksela na slici korištenjem Gaussove funkcije (eng. *Gaussian blur*).

2.3.1 Algoritam za neutralizaciju osvjetljenja na slici

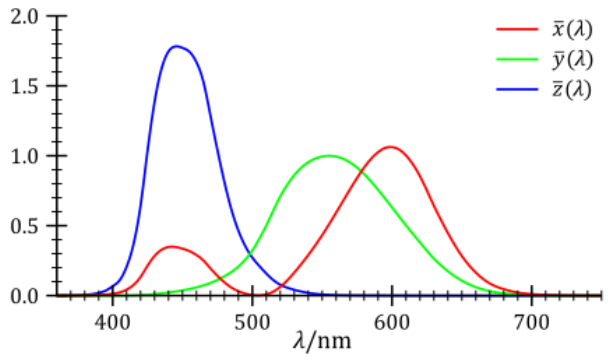
Prije pretvorbe slike u crno bijelu ili korištenja Gaussovog zamagljenja potrebno je na slici neutralizirati svijetlom naglašene dijelove i sjene na slici kako teren ne bude previše deformiran već da donekle zadrži konture objekata na slici. Neutralizacija osvjetljenja će se izvesti putem korekcije zasićenja (eng. *saturation*), svjetlosti (eng. *brightness*) i kontrasta na unesenoj slici. Potrebno je unijeti željenu jačinu pojedinog parametra u rasponu od 0 do 1. Svjetlost na slici se neutralizira tako da se računa svjetlost na svakom pikselu. Da bi se izračunala jačina svjetlosti pojedinog piksela potreban je skalarni umnožak vektora svjetlosti piksela i koeficijenta svjetlosti za svjetlost slike. Zatim izračunata jačina osvjetljenja piksela se koristi pri računanju zasićenja piksela, ali i za računanje kontrasta slike. Nakon toga korigirana slika se vraća natrag u program kako bi se pretvorila u crno bijelu sliku.

2.3.2 Pretvorba slike u crno bijelu sliku

Svjetlost slike ili Luma je ponderiran zbroj gama korigiranih (eng. *gamma correction* ili samo *gamma*) RGB komponenti boja na slici ili videu. Algoritam radi tako da se za svaki piksel izračuna svjetlina tog piksela i zapiše u drugu sliku generirajući sliku napravljenu samo od bijelih i crnih piksela. Formula za izračun svjetlosti piksela je:

$$Y = c_1 \cdot R + c_2 \cdot G + c_3 \cdot B \quad (2\text{-}1)$$

Gdje su: c_1, c_2, c_3 koeficijenti pojedinog algoritma, a R,G,B gama korigirane komponente boja na slici. Koeficijenti se računaju prema CIE funkcijama za podudaranje boja i relevantnih standardnih kromatičnosti komponenata boja. Funkcije za podudaranje boja su numerički opis tristimulusne vrijednosti spektralnih boja. Graf funkcija za podudaranje boja je prikaza na slici Sl. 2.8. Koeficijenti se koriste kako bi se što vjernije prikazao spektar primarnih boja.



Sl. 2.8. Grafički prikaz tristimulusne vrijednosti spektralnih boja.

Za računanje osvjetljenja pojedinog piksela na zadnoj slici kako bi se generirala crno-bijela slika koristi se formula naziva Rec. 601 luma[7]:

$$Y = 0.299 \cdot \mathrm{R} + 0.587 \cdot \mathrm{G} + 0.114 \cdot \mathrm{B} \qquad (2\text{-}2)$$

Nakon što se generira crno-bijela slika ona se koristi kao mapa za visine terena unutar igrice, na temelju čega se generira teren.

2.3.3 Algoritam za zamagljenje slike

Najčešće korišten algoritam za zamagljenje tadane slike je baziran na Gaussovoj funkciji. Osim zamućenja piksela na slici algoritam se koristi kako bi se uklonili nepravilnosti ili neki očiti detalji na slici, ali i da se otkriju rubovi objekta na slici (eng. *edge detection*) nakon usrednjavanja.

Gaussova funkcija za jednu dimenziju je:

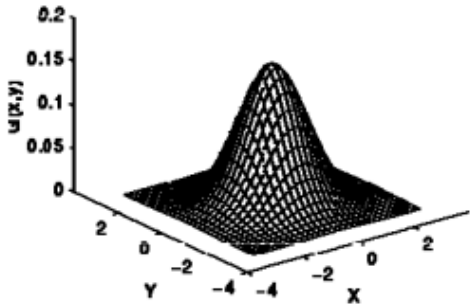
$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}} \qquad (2\text{-}3)$$

Dok Gaussova funkcija za dvije dimenzije je produkt dvije Gaussove funkcije po jedne u svakoj dimenziji:

$$G(x,y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x^2+y^2)}{2\sigma^2}} \qquad (2\text{-}4)$$

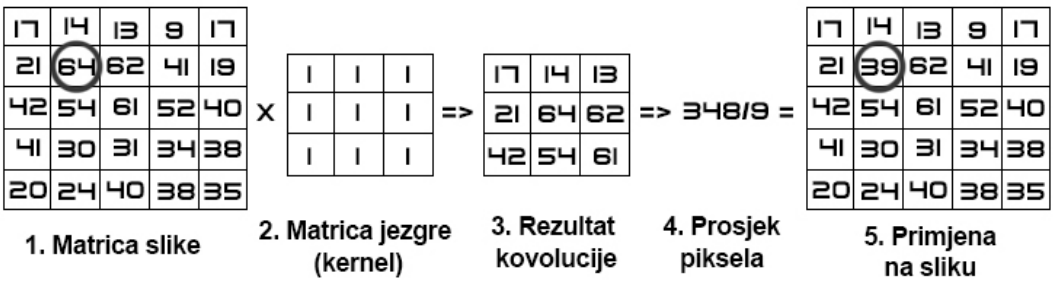
Gdje je su x i y udaljenosti od ishodišta po vertikalnoj ili horizontalnoj osi, σ je standardna devijacija Gaussove radiobe, a izračun $\frac{1}{\sqrt{2\pi\sigma^2}}$ je zapravo faktor za skaliranje Gaussove razdiobe. Kada radimo sa slikama potrebno je koristiti Gaussovu funkciju za dvije dimenzije

gdje je sredina razdiobe jednaka nuli, a standardna devijacija razdiobe jednaka jedinici. Graf takve funkcije je prikazan na slici 2.9.



Sl. 2.9. Grafički prikaz dvodimenzionalne Gaussove funkcije sa sredinom jednakoj 0, a σ = 1.

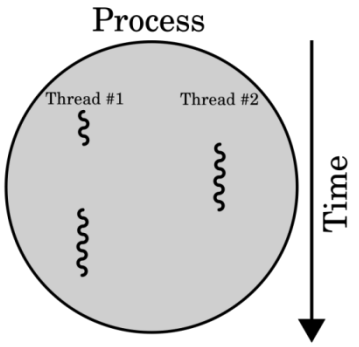
Gaussova zamagljenost (eng. *Gaussian blur*) kreće iz centra slike, te se koristi ponderiranom konvolucijskom jezgrom (eng. *kernel*) . Konvolucija je matematička funkcija nastala integriranjem umnoška dviju funkcija po intervalu njihove definicije gdje su te fukcije ravnopravne tako da svaka infinitezimalna promjena jedne funkcije utječe na drugu funkciju u cijelome intervalu definicije[8]. Brzina ovog algoritma ovisi o veličini jezgre koja se koristi, što je jezgra veća to će proces biti sporiji, no kvaliteta zamagljene slike će biti veća. Svaka jezgra mora imati vrijednosti unutar sebe one se dobijaju tako da se izračuna Gaussova funkcija sa zadanom standardnom devijacijom. Najčešće se koriste već prethodno generirane jezgre. Algoritam radi tako da se svaki piksel stavi u sredinu jezgre. Tada se svaki piksel, koji se nalazi unutar jezgre, pomnoži sa vrijednosti u jezgri. Sve se vrijednosti zbroje i podjele sa ukupnom veličinom jezgre kako bi se dobio prosjek piksela. Taj prosjek se zapisuje u piksel na slici koji se nalazi u sredini jezgre. Na slici 2.10. je grafički prikazan proces računanja prosjeka piksela na slici[9].



Sl. 2.10. Grafički prikaz računanja prosjeka piksela putem Gaussovog algoritma.

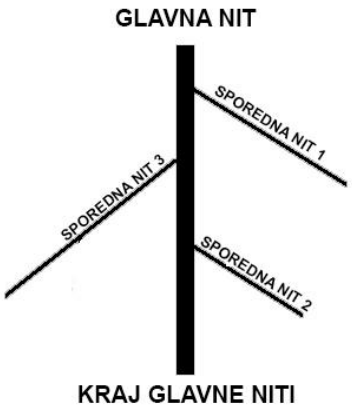
2.4. Višenitno programiranje

Višenitno programiranje (eng. *multithreaded programming*) je istovremeno izvršavanje više manjih zadataka kroz sve jezgre procesora. To skraćuje vrijeme koje je potrebno da se obrade podatci. Proces je program koji se trenutačno izvodi, a on se može podjeliti u više nezavisnih jedinica zvanim nitima (eng. *threads*). Niti su jako lagani procesi unutar samog procesa. Na slici 2.11. je prikazan izgled niti i procesa.



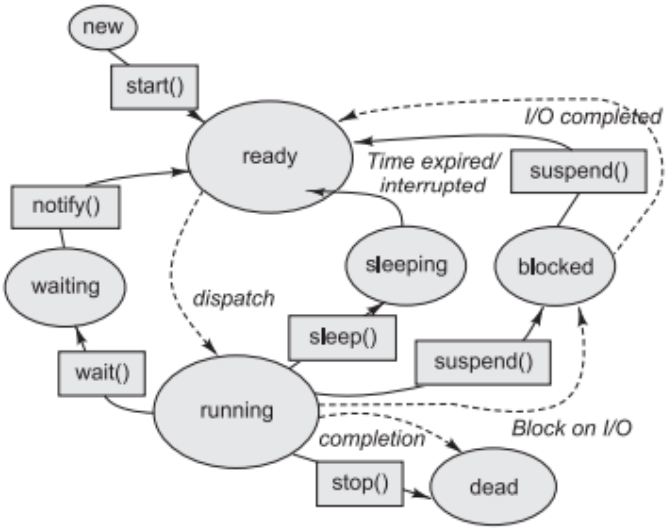
Sl. 2.11. Grafički prikaz procesa i niti unutar programa.

Postoje dva načina multitaskinga kod programa: procesno orijentiran multitasking (eng. *process-based multitasking*) koji je kontroliran od strane operacijskog sustava i nitno orijentiran multitasking (eng. *thread-based multitasking*) koji je kontroliran od strane programera koji piše program. Vrijeme izvođenja programa u niti (eng. *timeslice*) je kontroliran od strane sklopovskih brojača. Oni procesoru javljaju kada je kraj izvođenja pojedine niti i signalizira procesoru da treba spremiti rezultate izvođenja programa u svoj programski stog (eng. *stack*). Nakon čega procesor podatke sprema sa programskog stoga u predefinirana mjesta u memoriji. Isto tako procesor može zatražiti natrag te podatke iz memorije te ih ponovno staviti u programski stog. Taj proces se naziva kontekstno prebacivanje (eng. *context switching*). Sve niti unutar pojedinog procesa koriste istu memoriju i imaju ista stanja, te mogu komunicirati između sebe jer koriste iste varijable. Svi procesi imaju glavnu nit i ostale sporedne niti. Glavna nit je nit koja se stvara odmah čim se program pokrene i postoji sve dok program radi. Sporedne niti se stvaraju prema potrebi, te se gase kada prođe vrijeme izvođenja niti.



Sl. 2.12. Grafički prikaz izvođenja glavne i sporednih niti.

Unutar životnog vijeka nit unutar procesa može biti samo u jednom stanju. Stanje se mijenja ovisno o operaciji koja se trenutno izvodi. Stanja niti unutar procesa nisu povezana sa stanjima niti u operacijskog sustava. Kada se nit tek kreira onda će njeno stanje će biti NEW, a kada se njenog izvođenje proslijedi planeru koji stanje niti prebacuje u RUNNABLE. Ovisno o korištenju operacija unutar procesa stanja niti se može mijenjati u WAINTING, SLEEPING i BLOCKED. Kada nit prestaje sa radom onda se njeno stanje prebacuje u stanje TERMINATED[10].



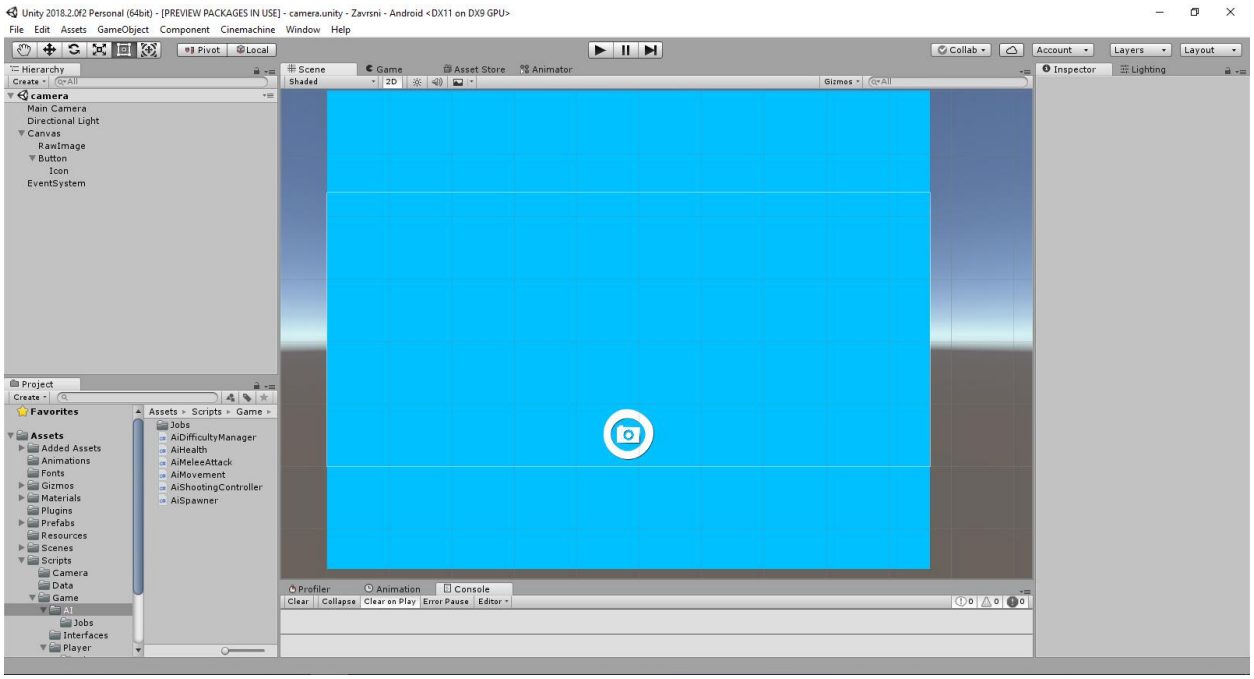
Sl. 2.13. Grafički prikaz životnog vijeka niti.

3. IZRADA VIDEO IGRE

U prošlom poglavlju objašnjenja je teorija iza izrade video igre u bilo kojem pokretaču igre. U ovom poglavlju će biti prikazana izrada video igre na praktičnom prikazu uz programski kod. Prilikom izrade video igre pratile su se upute proizvođača samog pokretača igre, te su se uveliko koristile klase koje su serijalizirane (eng. *ScriptableObjects*), a za višenitno programiranje koristio se Unity Jobs sistem.

3.1. Uzimanje slike sa kamere mobilnog uređaja

Prije pisanja programskog koda važno je pripremiti scenu gdje se dodaje pozadinska slika koja se kasnije mijenja sa okvirima (eng. *Frames*) od kamere uređaja i gumb koji će zaustaviti uzimanje okvira s kamere uređaja i zadnji okvir prije pritiska gumba spremiti u varijablu. Na slici 3.1 je prikazana scena za uzimanje slike s kamere mobilnog uređaja.



Sl. 3.1. Prikaz scene za prikupljanje slika sa kamere uređaja u Unity pokretaču igre.

Prilikom pokretanja scene potrebno je provjeriti ima li uređaj kameru i može li se ona koristiti. Pošto se u većini slučajeva koristi stražnja kamera uređaja programski kod traži samo stražnje kamere. Nakon toga je potrebno pozadinsku sliku predstaviti kao teksturu koja se može tijekom vremena mijenjati. Za to se koristi ugrađena klasa WebCamTexture koja omogućuje da se na teksturu prikaže video uživo s bilo koje kamere (web kamere ili kamere na mobilnom uređaju). Nakon što se pripremi kamera i tekstura tada se započinje s prikazivanjem videa uživo

na pripremljenu teksturu. Ako uređaj nema kameru tada se javlja pogreška i vraća se igrica na glavni izbornik. U programskom kodu 3.1 je prikazan dio DeviceCamera klase koji je zadužen za prikazivanje okvira na teksturu.

```
public class DeviceCamera : MonoBehaviour
{
    public RawImage background;

    [NonSerialized]
    public static Texture2D takenPhoto;

    private WebCamTexture deviceCamera;

    #region UnityFunctions
    private void Awake ()
    {
        SetupDeviceCamera();
    }
    #endregion

    #region HelperFunctions
    private void SetupDeviceCamera()
    {
        WebCamDevice[] devices = WebCamTexture.devices;
        if(devices.Length == 0)
        {
            Debug.LogError("No camera detected!");
            SceneManager.LoadScene(0, LoadSceneMode.Single); // Loads main menu
            return;
        }

        for (int i = 0; i < devices.Length; i++)
        {
            if(!devices[i].isFrontFacing)
            {
                deviceCamera = new WebCamTexture(
                                devices[i].name,
                                Screen.width,
                                Screen.height
                            );
                break;
            }
        }

        if(deviceCamera == null)
        {
            return;
        }

        deviceCamera.Play();
        background.color = Color.white;
        background.texture = deviceCamera;
    }
```

Programski kod 3.1. Prikaz početka tijela klase DeviceCamera.

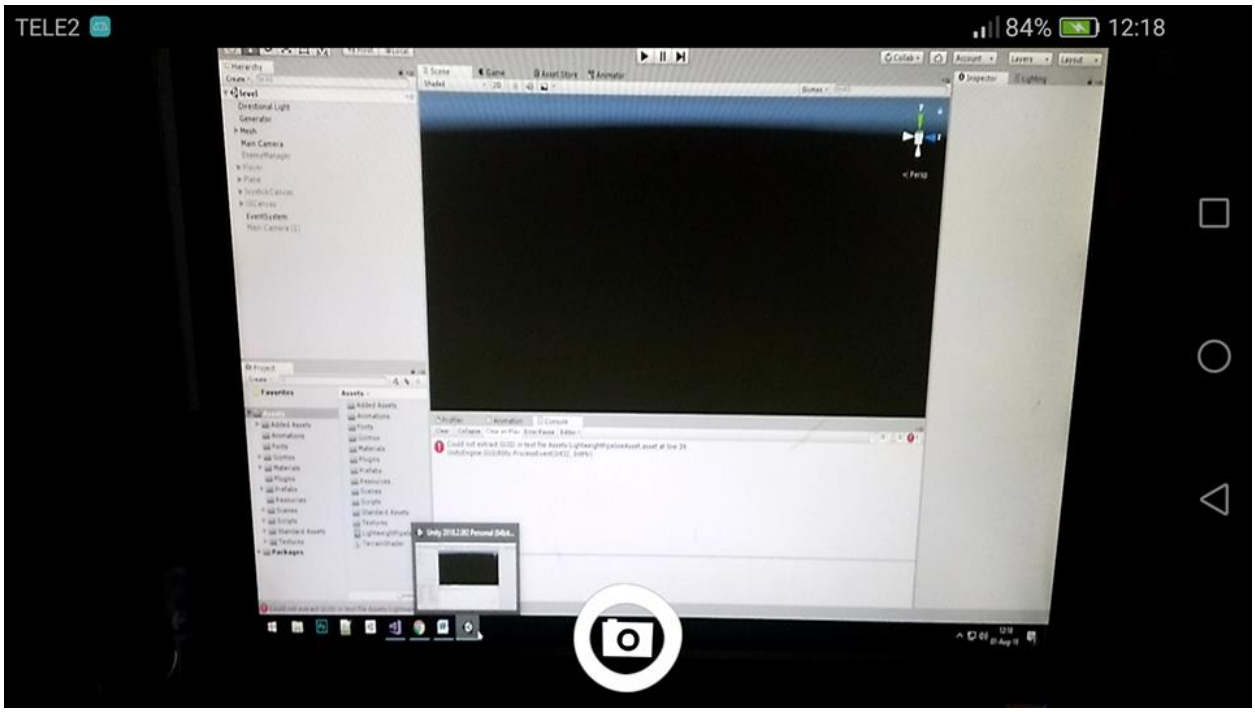
WebCamTexture tekstura se učitava prilikom svakog okvira u igrici. Da bi se uslikala slika kada korisnik pritisne gumb za slikanje potrebno je pričekati kraj trenutnog okvira u igri i uzeti sliku iz njega. Da bi se to izvelo ne može se koristiti normalna funkcija već korutina (eng. Coroutine). Korutina je tip funkcije koji može zaustaviti svoje izvođenje i prepustiti Unityu da nastavi daljnje izvođenje, te se kasnije može vratiti i nastaviti svoje izvođenje. Svaka korutina je funkcija tipa IEnumerator koji omogućuje te radnje. Na programskom kodu 3.2 su prikazane funkcije za uzimanje slike s kamere uređaja.

```
public void BackButton()
{
    SceneManager.LoadScene("main_menu", LoadSceneMode.Single);
}

public void TakeCameraPhoto()
{
    StartCoroutine("TakePhotoCoroutine");
}

private IEnumerator TakePhotoCoroutine()
{
    yield return new WaitForEndOfFrame();

    takenPhoto = new Texture2D(deviceCamera.width, deviceCamera.height);
    takenPhoto.SetPixels(deviceCamera.GetPixels());
    TextureScale.Bilinear(takenPhoto, 128, 128);
    takenPhoto.Apply();

    deviceCamera.Stop();
    background.texture = takenPhoto;
    SceneManager.LoadScene("loading", LoadSceneMode.Single);
}
```

Programski kod 3.2. Prikaz funkcija za uzimanje slike sa kamere uređaja.

Nakon što korisnik klikne gumb za slikanje tada se poziva funkcija TakeCameraPhoto. Unity nema ugrađen način za direktno pozivanje korutine kada se pritisne neki UI (eng. user interface) element. Stoga se mora pozvati javna funkcija koja tada poziva korutinu. Nakon što se započne korutina čeka se kraj izvođenja trenutnog okvira, te se nastavlja s uzimanjem slike s teksture. Prvo se pravi varijabla teksture, te se kopiraju svi pikseli sa WebCamTexture teksture. Nakon toga se uslikanoj slici mijenja veličina na 128 x 128 piksela, kako bi se ubrzao proces generiranja terena. Čitav proces mijenja veličine slike je isprogramiran koristeći višenitno programiranje što omogućuje brzu pretvorbu slike. Nakon toga se uslikana slika sprema u globalnu varijablu i učitava se scena za učitavanje levela. Scena loading služi kako bi se prikazao napredak učitavanja levela. Bez učitavanja posebne scene onda bi korisniku aplikacija bila zamrznuta bez

ikakve prethodne najave, te bi on mogao misliti da nešto nije u redu s aplikacijom. Na slici 3.2. je prikazana scena za uzimanje slike s kamere uređaja kada je aplikacija pokrenuta.



Sl. 3.2. Prikaz uzimanja slike sa kamere uređaja kada je igra pokrenuta.

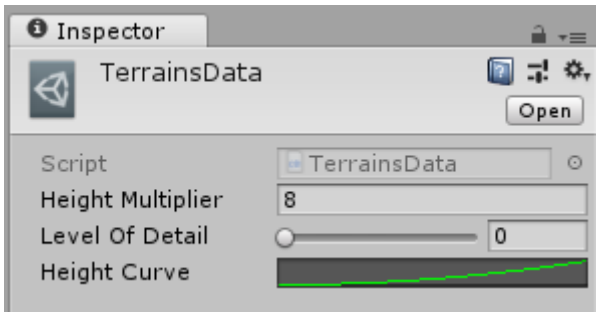
3.2. Priprema podataka za generiranje terena

Prije no što se mapa generira potrebni su podatci koji će se koristiti prilikom generiranja. Potrebno je napraviti klasu naziva TerrainsData koja će se serijalizirati sa podatcima potrebnim za generiranje terena. Serijalizacija je postupak pretvorbe objekta u bajtove koji se tada mogu spremiti kao datoteka. U programskom kodu 3.3 je prikazano tijelo TerrainsData klase.

```
[CreateAssetMenu()]
public class TerrainsData : ScriptableObject
{
    public float heightMultiplier = 8f;
    [Range(0, 6)]
    public int levelOfDetail = 0;
    public AnimationCurve heightCurve;

    public float minTerrainHeight
    {
        get
        {
            return heightMultiplier * heightCurve.Evaluate(0);
        }
    }

    public float maxTerrainHeight
    {
        get
        {
            return heightMultiplier * heightCurve.Evaluate(1);
        }
    }
}
```

Programski kod 3.3. Prikaz programskog koda za klasu TerrainsData.

Varijabla heightMultiplier se množi s maksimalnom visinom terena dobivenog generiranjem koja je uvijek jedan, čime varijabla heightMultiplier određuje maksimalnu visinu terena. Varijabla levelOfDetail određuje koliko teren treba imati detalja odnosno s kojom količinom detalja će se prikazati visinska mapa (eng. heightmap). Ostale varijable služe za bojanje terena putem programa za grafičku karticu (eng. shader). Na slici 3.3 je prikazana serijalizirana klasa.



Sl. 3.3. Prikaz serijalizirane klase TerrainsData.

3.3. Obrada slike u programskom jeziku C#

U poglavlju 2 objašnjena je teorija iza obrade slike. Za potrebe video igre koristiti će se algoritmi za uravnoteženje boja na slici, algoritam za zamagljenje slike i algoritam za pretvordbu slike u crno bijelu. U klasi ImageProcessing napisani su algoritmi za zamagljenje slike i algoritam za uravnoteženje boja na slici. Za pretvorbu slike iz obojane u crno bijelu koristiti će se ugrađena funkcija od strane pokretača igre. Sve funkcije unutar klase su statične i imaju svoje javne statične pozivatelje. Svaki pozivatelj tada poziva jednu funkciju koja tada poziva ostale pojedinačne funkcije prema potrebi. Ovo omogućuje kasnije lakšu pretvordbu programskog koda iz sinkronog u asinkroni. U programskom kodu 3.4 je prikazan početak klase ImageProcessing.

```
public class ImageProcessing
{
    public enum ImageProcessingTypes
    {
        IMG_PROCESS_GAUSSIAN_BLUR,
        IMG_PROCESS_COLOR_BALANCE
    }
    public class ProcessingData
    {
        public int start;
        public int end;

        public ProcessingData(int s, int e)
        {
            start = s;
            end = e;
        }
    }
    public static readonly float[][] GAUSSIAN_KERNEL_3x3 = new float[3][]
    {
        new float[3]{ 0.07511361f, 0.1238414f, 0.07511361f },
        new float[3]{ 0.1238414f,  0.20418f,   0.1238414f  },
        new float[3]{ 0.07511361f, 0.1238414f, 0.07511361f }
    };

    private static Color[] texColors;
    private static Color[] newColors;

    private static int width;
    private static int height;

    private static object[] parameters;

    public static Texture2D GaussianBlur(Texture2D texture, params object[] parameters)
    {
        return FilterMethod(texture, ImageProcessingTypes.IMG_PROCESS_GAUSSIAN_BLUR,
parameters);
    }
    public static Texture2D ColorBalance(Texture2D texture, params object[] parameters)
    {
        return FilterMethod(texture, ImageProcessingTypes.IMG_PROCESS_COLOR_BALANCE,
parameters);
    }
```

Programski kod 3.4. Prikaz početka klase ImageProcessing.

U programskom kodu 3.5 je prikazana funkcija koja obrađuje unesene parametre i poziva algoritme po potrebi. Parametar types pomaže pri odabiru algoritma koji se treba pozvati. Kada se obradi slika tada se obrađeni pikseli spremaju u globalnu varijablu newColors. Na kraju funkcije FilterMethod pravi se nova slika sa istom veličinom kao unesena slika, ali sa novim pikselima.

```
private static Texture2D FilterMethod(Texture2D inputTexture, ImageProcessingTypes
types, params object[] filterParams)
    {
        texColors = null;
        newColors = null;

        width = inputTexture.width;
        height = inputTexture.height;

        try
        {
            texColors = inputTexture.GetPixels();
        }
        catch (UnityException e)
        {
            Debug.LogError(e);
            return inputTexture;
        }
        parameters = filterParams;
        newColors = new Color[texColors.Length];
        ProcessingData threadData = new ProcessingData(0, texColors.Length);

        switch (types)
        {
            case ImageProcessingTypes.IMG_PROCESS_GAUSSIAN_BLUR:
                {
                    GaussianBlurFilter(threadData);
                    break;
                }
            case ImageProcessingTypes.IMG_PROCESS_COLOR_BALANCE:
                {
                    ColorBalanceFilter(threadData);
                    break;
                }
        }

        Texture2D outputTexture = new Texture2D(inputTexture.width,
inputTexture.height);
        outputTexture.SetPixels(newColors);
        outputTexture.Apply();
        return outputTexture;
    }
```

Programski kod 3.5. Prikaz tijela FilterMethod funkcije.

U programskom kodu 3.6 je prikazana funkcija za zamagljenja slike koristeći Gaussovu formulu (eng. *Gaussian blur*).

```
private static void GaussianBlurFilter(System.Object obj)
{
    ProcessingData processingData = (ProcessingData)obj;
    float[][] kernel = (float[][])parameters[0];
    int kernelSize = kernel.Length;
    int iteration = (int)parameters[1];

    for (int i = 0; i < iteration; i++)
    {
        for (int pixel = processingData.start; pixel < processingData.end; pixel++)
        {
            int pixelX = pixel % width;
            int pixelY = pixel / width;

            Color convolutedPixel = new Color(0f, 0f, 0f, 0f);
            for (int y = 0; y < kernelSize; y++)
            {
                 int kernelY = y - kernelSize / 2;
                 for (int x = 0; x < kernelSize; x++)
                 {
                    int kernelX = x - kernelSize / 2;
                    try
                    {
                        convolutedPixel += texColors[GetIndex(pixelX + kernelX, pixelY
+ kernelY, width, height)] * kernel[x][y];
                    }
                    catch (Exception e)
                    {
                        Debug.LogError(e);
                    }
                }
            }
            newColors[pixel] = convolutedPixel;
        }
        if (i < iteration - 1)
        {
            for (int colorIndex = 0; colorIndex < texColors.Length; colorIndex++)
                texColors[colorIndex] = newColors[colorIndex];
        }
    }
}
```

Programski kod 3.6. Prikaz tijela GaussianBlurFilter funkcije.

U funkciji GaussianBlurFilter je potrebno definirati jezgru i broj iteracija filtera. Što je veći broj iteracija to je zamagljenje slike veće, ali je izvođenje funkcije duže. Veličina jezgre (eng. *kernel*) određuje brzinu izvođenja funkcije. Jezgra koja se koristi dobivena je proceduralno putem uvrštavanja varijabli u Gaussovu formulu, a koristi se 3x3 jezgra. Filter radi tako da prođe kroz sve piksele na slici, te na svakom pikselu stavi sredinu jezgre i pomnoži sve susjedne piksele s jezgrom i spremi zbroj u piksel koji se u datom trenutku obrađuje. U programskom kodu 3.7 je prikazano tijelo funkcije ColorBalanceFilter.

```
private static void ColorBalanceFilter(System.Object obj)
{
    ProcessingData processingData = (ProcessingData)obj;
    float saturation = (float)parameters[0];
    float brightness = (float)parameters[1];
    float contrast = (float)parameters[2];

    for (int pixel = processingData.start; pixel < processingData.end; pixel++)
    {
        Vector3 brightnessColor = new Vector3(texColors[pixel].r, texColors[pixel].g,
texColors[pixel].b) * brightness;
        Vector3 luminaceCoeff = new Vector3(0.2125f, 0.7154f, 0.0721f);
        float brightnessIntensity = Vector3.Dot(brightnessColor, luminaceCoeff);

        Vector3 saturationColor = Vector3.Lerp(new Vector3(brightnessIntensity,
brightnessIntensity, brightnessIntensity), brightnessColor, saturation);

        Vector3 contrastColor = Vector3.Lerp(new Vector3(.5f, .5f, .5f),
saturationColor, contrast);
        newColors[pixel] = new Color(contrastColor.x, contrastColor.y,
contrastColor.z, texColors[pixel].a);
    }
}
```

Programski kod 3.7. Prikaz tijela ColorBalanceFilter funkcije.

ColorBalanceFilter funkcija za svaki piksel na slici mijenja zasićenje na slici, svjetlinu i kontrast slike. U prvom dijelu funkcije računa se intenzivitet svjetlost na slici. Zatim mijenja svjetlost na slici po unosu u parametre. U drugom dijelu funkcije računa se boja zasićenja slike na temelju jačine svjetlosti na slici, boji svjetlosti i unesenog zasićenja u parametre funkcije. U posljednjem dijelu funkcije računa se kontrast na slici na temelju zasićenja i parametra funkcije koji je prethodno unesen. Svi obrađeni pikseli zatim se spremaju u globalnu varijablu newColors i vraćaju korisniku kao nova slika. Prilikom obrade slike ne koristi se višenitno programiranje jer nije potrebno. Prije obrade slika se smanjuje na 128x128 piksela što omogućuje bržu obradu kroz nju.

3.4. Generiranje terena na temelju slike

Da bi se generirao teren na temelju slike potrebno je imati visinsku mapu na temelju koje će biti generiran teren. Kako program unaprijed ne zna kakva će se slika koristiti teren se mora generirati proceduralno. Visinska mapa u ovoj video igri se dobije nakon što se obradi slika dobivena iz kamere na mobitelu. Za generiranje mape brine se MapGenerator klasa koja povezuje klase za generiranje 3D objekta terena i klase za obradu slike. Nakon što se obradi digitalna slika funkcijom GenerateHeightmapFloats se digitalna slika pretvara u visinsku mapu. U programskom 3.8 je prikazana funkcija GenerateHeightmapFloats.

```
private float[,] GenerateHeightmapFloats(Texture2D texture)
{
        float[,] heightmap = new float[texture.width, texture.height];

        for (int y = 0; y < texture.height; y++)
        {
                for (int x = 0; x < texture.width; x++)
                {
                        heightmap[x, y] = texture.GetPixel(x, y).grayscale;
                }
        }
        return heightmap;
}
```

Programski kod 3.8. Prikaz GenerateHeightmapFloats funkcije.

Kada se obradi slika onda se poziva funkcija GenerateHeightmapFloats. Ona svaki piksel slike pretvara u crno bijeli kako bi se generirala visinska mapa koja se zatim sprema u dvodimenzionalno polje realnih brojeva. Zatim se poziva funkcija GenerateTerrainMeshFloats koja proceduralno generira 3D objekt terena. Proceduralno generiranje terena se radi tako da se prvo prođe kroz sve piksele na visinskoj mapi, te se na temelju svjetlosti piksela na slici odabire visina terena. Nakon toga slijedi generiranje vrhova, UV točaka i trokuti. Nakon što su se generirali vrhovi koristi se stil bojanja pod nazivom Vertex Colors gdje se bojaju samo vrhovi, te se te boje prikazuju na samom 3D objektu. Na kraju sve informacije se predaju ugrađenom objektu pod nazivom Mesh koji tada prikazuje 3D objekt terena na korisnikovom ekranu. Da bi se obojani vrhovi prikazali na korisnikovom ekranu potrebno je isprogramirati poseban program za grafičke kartice koji isključivo služi za prikaz obojanih vrhova. U programskom kodu 3.9 prikazano je tijelo funkcije GenerateMeshFloats koja generira teren na temelju realnih vrijednosti (eng. *Floats*) visinske mape.

```
public static MeshData GenerateTerrainMeshFloats(float[,] heightmap, int levelOfDetail, float
heightMultiplier, ColorData colorData)
{
        int width = heightmap.GetLength(0);
        int height = heightmap.GetLength(0);

        int vertexIndex = 0;
        float topLeftX = (width - 1) / -2f;
        float topLeftZ = (height - 1) / 2f;

        int simplificationIncrement = (levelOfDetail == 0) ? 1 : levelOfDetail * 2;
        int verticesPerLine = (width - 1) / simplificationIncrement + 1;

        MeshData meshData = new MeshData(width, height);

        for (int z = 0; z < height; z += simplificationIncrement)
        {
                for (int x = 0; x < width; x += simplificationIncrement)
                {
                        float y = heightmap[x, z] * heightMultiplier;
                        Color32 tempColor = Color.white;

                        meshData.vertices[vertexIndex] = new Vector3(topLeftX + x, y, topLeftZ - z);
                        meshData.uvs[vertexIndex] = new Vector2(x / (float)width, z / (float)height);

                        if (heightmap[x, z] >= colorData.colorHeights[0] && heightmap[x, z] <=
colorData.colorHeights[1])
                                tempColor = colorData.baseColors[1];
                        else if (heightmap[x, z] >= colorData.colorHeights[1] && heightmap[x, z] <=
colorData.colorHeights[2])
                                tempColor = colorData.baseColors[2];
                        else
                                tempColor = colorData.baseColors[0];

                        meshData.colors32[vertexIndex] = tempColor;

                        if (x < width - simplificationIncrement && z < height -
simplificationIncrement)
                        {
                                meshData.CreateTriangle(vertexIndex, vertexIndex + verticesPerLine +
1, vertexIndex + verticesPerLine);
                                meshData.CreateTriangle(vertexIndex + verticesPerLine + 1,
vertexIndex, vertexIndex + 1);
                        }
                        vertexIndex++;
                }
        }
        return meshData;
}
```

Programski kod 3.9. Prikaz tijela GenerateTerrainMeshFloats funkcije.

GenerateTerrainMeshFloats funkcija generira teren na temelju realnih vrijednosti visinske mape. Ona generiranje obavi 10 puta brže od generiranja terena na temelju visinske mape kao slike. Razlog tome je jer procesor računala radi najbrže s vrijednostima nego s referencama. Sve slike unutar Unity pokretača igre su referentni tipovi. Jer procesor računala ima ugrađene algoritme za optimizaciju i bržu obradu osnovnih tipova podataka nego referentne podatake. Reference su izvedeni tipovi podataka, dok realni tip podataka je osnovni tip.

Nakon što se generira mapa tada se okida događaj (eng. *event*) OnMapGenerated. Događaji u C# jeziku se rade tako da se prvo napravi javna globalna varijabla tipa delegate.

Delegati su tipovi koji predstavljaju reference varijable na neku funkciju. Tada se sve ostale klase koje trebaju informaciju o događaju se pozivaju na isti. Kada se događaj dogodi tada se okidaju funkcije koje su priključene na taj događaj. U programskom kodu 3.10 su prikazani primjeri deklaracije, pozivanja i okidanja događaja.

```
MAP GENERATOR SKRIPTA:
public delegate void MapReadyDelegate();
public event MapReadyDelegate OnMapGenerated;

private void Start()
{
	if (OnMapGenerated != null)
	{
		OnMapGenerated();
	}
}

DECORATIONS SPAWNER SKRIPTA:
generator = FindObjectOfType<MapGenerator>();
generator.OnMapGenerated += SpawnDecorations;

private void SpawnDecorations()
{
	for (int i = 0; i < spawnAreas.Length; i++)
	{
		for (int spawnPoints = 0; spawnPoints < 50; spawnPoints++)
		{
			Vector3 origin = new Vector3(Random.Range(spawnAreas[i].bounds.min.x,
spawnAreas[i].bounds.max.x),
				generator.terrainData.heightMultiplier + 3f,
				Random.Range(spawnAreas[i].bounds.min.z,
spawnAreas[i].bounds.max.z));
			Ray spawnerRay = new Ray(origin, Vector3.down);
			RaycastHit spawnHit;

			if (Physics.Raycast(spawnerRay, out spawnHit))
			{
				int index = Random.Range(0, decorationPrefabs.Length);
				GameObject go = Instantiate(decorationPrefabs[index],
spawnHit.point, Random.rotation, decorParent);
				float scale = Random.Range(0.05f, 0.6f);
				go.transform.localScale = new Vector3(scale, scale, scale);
				decorObjects.Add(go);
			}
		}
	}

	if(OnDecorationsSpawned != null)
	{
		OnDecorationsSpawned();
	}
}
```

Programski kod 3.10. Prikaz deklaracije, okidanja i pozivanja događaja u C# programskom jeziku.

Nakon što se mapa generira postavljaju se dekorativni objekti kako bi ugođaj u igrici bio bolji. Postavljanje dekorativnih objekata se odvija tek kada se okine događaj OnMapGenerated. Za postavljanje dekorativnih objekata zadužena je klasa DecorationsSpawner. U programskom kodu 3.11 je prikazano tijelo SpawnDecoratioins funkcije.

```
public class DecorationsSpawner : MonoBehaviour
{
    public GameObject[] decorationPrefabs;
    public GameObject spawnAreasGO;
    public int maxSpawnDecorations = 50;
    public Transform decorParent;
    private MapGenerator generator;
    private BoxCollider[] spawnAreas;
    private List<GameObject> decorObjects;

    public delegate void DecorationsSpawnedDelegate();
    public event DecorationsSpawnedDelegate OnDecorationsSpawned;

    private void Awake()
    {
        spawnAreas = spawnAreasGO.GetComponents<BoxCollider>();
        decorObjects = new List<GameObject>();

        generator = FindObjectOfType<MapGenerator>();
        generator.OnMapGenerated += SpawnDecorations;

        SpawnController spawnController = FindObjectOfType<SpawnController>();
        spawnController.OnPlayerSpawned += DelayTurnLOD;
    }
    private void SpawnDecorations()
    {
        for (int i = 0; i < spawnAreas.Length; i++)
        {
            for (int spawnPoints = 0; spawnPoints < 50; spawnPoints++)
            {
                Vector3 origin = new Vector3(Random.Range(spawnAreas[i].bounds.min.x,
spawnAreas[i].bounds.max.x),
                    generator.terrainData.heightMultiplier + 3f,
                    Random.Range(spawnAreas[i].bounds.min.z, spawnAreas[i].bounds.max.z));
                Ray spawnerRay = new Ray(origin, Vector3.down);
                RaycastHit spawnHit;

                if (Physics.Raycast(spawnerRay, out spawnHit))
                {
                    int index = Random.Range(0, decorationPrefabs.Length);
                    GameObject go = Instantiate(decorationPrefabs[index], spawnHit.point,
Random.rotation, decorParent);
                    float scale = Random.Range(0.05f, 0.6f);
                    go.transform.localScale = new Vector3(scale, scale, scale);
                    decorObjects.Add(go);
                }
            }
        }
        if(OnDecorationsSpawned != null)
        {
            OnDecorationsSpawned();
        }
    }

    private void DelayTurnLOD()
    {
        StartCoroutine(TurnLODGroups());
    }

    private IEnumerator TurnLODGroups()
    {
        yield return new WaitForSeconds(5f);
        for (int i = 0; i < decorObjects.Count; i++)
        {
            decorObjects[i].GetComponent<LODGroup>().enabled = true;
        }
    }
}
```

Programski kod 3.11. Prikaz tijela klase DecorationsSpawner.

Na mjestu gdje 3D objekt terena treba biti su napravljeni prostori za postavljanje na teren. Prostori za postavljanje dekorativnih objekata na teren su spremljeni u javnu globalnu varijablu spawnAreasGO. Tada se prilikom pokretanja scene svi prostori spremaju u polje ugrađenog tipa BoxCollider. Kada se pokrene postavljanje dekorativnih objekata na svaki prostor se postavlja oko 50 objekata. Za svaki objekt se uzima nasumična vrijednost unutar prostora za postavljanje dok se visina za postavljanje uzima najviši vrh terena plus tri metra kako bi se osiguralo da se svi dekorativni objekti postave. Nakon toga se pucaju zrake (eng. *raycasting*) koje zatim vraćaju informaciju doticanja zrake s terenom. Na točki doticanja zrake s terenom se postavlja dekorativni objekt. Važno je napomenuti da se zrake mogu jedino dotaknuti s terenom jer se unutar koda definira slojna maska (eng. *layer mask*). Slojne maske definiraju s kojim slojem se mogu zrake dotaknuti. Radi performansi igre sve dekoracije imaju level detalja (eng. level od details (LOD)) što omogućuje da objekti mijenjaju razinu detalja kako se kamera udaljava od njih. No ovdje se koriste kako bi se nepotrebni objekti obrisali iz scene ako igrač nije kod njih. Kako igrač prilazi objektima tako se oni ponovno stvaraju u sceni. Ovo ne zahtijeva dodatno procesiranje podataka pošto su objekti stvoreni sustav samo uključuje i isključuje objekte. Problem se javlja kada se level tek učita i kada se snima čitav teren iz zraka. Tada je potrebno da se svi dekorativni objekti pojave. Taj problem je riješen tako da je na svakom gotovom objektu (eng. prefab) ugašen LOD sustav koji se uključuje koristeći korutine, tek nakon što se igrač stvori na mapi. Tada se kamera spušta do igrača i nema potrebe imati uključene dekorativne objekte koji su jako udaljeni od samog igrača jer oni uzrokuju padove okvira po sekundi (eng. *frames per second*) procesor i grafička kartica moraju procesirati nepotrebne objekte.

3.5. Igrač i njegova mehanika

Sve stvari vezane za igrača u video igri je višenitno programirano. Pošto se sve te stvari odvijaju vrlo često u igri kako korisnik nebi brzo istrošio bateriju program je programiran koristeći ugrađeni sustav za višenitno programiranje nazvan Jobs sustav. Unity Jobs sistem omogućava korisniku jednostavno pisanje višenitnog programa koji dobro radi sa ugrađenim stvarima u Unity pokretač igrica[11]. Igračeva mehanika se sastoji od četiri manje mehanike, a to su: zdravlje, kretnje, pucanje i stvaranje igrača. Čitava video igra je vrlo jednostavna kada igrač umre onda je igra završena, a igrač može započeti igru ispočetka ili izabrati novu mapu.

3.5.1 Mehanika stvaranja igrača u svijet

Za stvaranje igrača u svijet zadužena je klasa SpawnController. Igrač se nasumično postavlja na 3D objekt terena. Objekt igrača je prethodno napravljen u Unity pokretaču igrica kako se nebi trošilo vrijeme na njegovo kreiranje. U programskom kodu 3.12 je prikazano tijelo SpawnController klase.

```
public class SpawnController : MonoBehaviour
{
    [Range(-18f, 13f)]
    public float maxSpawnPointX = 13f;
    [Range(-18f, 13f)]
    public float minSpawnPointX = -18f;
    [Range(-10f, 19f)]
    public float maxSpawnPointZ = 19f;
    [Range(-10f, 19f)]
    public float minSpawnPointZ = -10f;
 public GameObject playerGO;

    private MapGenerator generator;

    public delegate void PlayerSpawnedDelegate();
    public event PlayerSpawnedDelegate OnPlayerSpawned;

    private void Awake()
    {
        generator = FindObjectOfType<MapGenerator>();
        generator.OnMapGenerated += SpawnPlayer;
    }
    private void SpawnPlayer()
    {
        float spawnPointX = Random.Range(minSpawnPointX, maxSpawnPointX);
        float spawnPointY = generator.terrainData.maxTerrainHeight;
        float spawnPointZ = Random.Range(minSpawnPointZ, maxSpawnPointZ);
        Vector3 spawnPosition = new Vector3(spawnPointX, spawnPointY, spawnPointZ);

        RaycastHit hit;
        if (Physics.Raycast(spawnPosition, Vector3.down, out hit))
        {
            spawnPosition.y = hit.point.y + playerGO.transform.localScale.y;
        }

        playerGO.transform.position = spawnPosition;
        playerGO.transform.rotation = Quaternion.identity;
        playerGO.SetActive(true);
        GamePause.isGamePaused = false;
        if (OnPlayerSpawned != null)
        {
            OnPlayerSpawned();
        }
    }
}
```

Programski kod 3.12. Prikaz tijela klase SpawnController.

Nakon što se 3D objekt terena generira okida se događaj OnMapGenerated koji pokreče funkciju SpawnPlayer u SpawnController klasi. X i Z pozicija stvaranja igrača je nasumično određena dok se Y pozicija dobije tako da se uzme najveći vrh terena. Nakon toga se ispaljuje zraka od te pozicije na sam teren. Ako zraka pogodi teren ta točka se uzima kao pozicija stvaranja igrača. Pozicija objekta igrača se tada postavlja na poziciju stvaranja, te se objekt igrača prikazuje na korisnikovom ekranu. No problem je u tome što je izvorišna točka objekta igrača u sredini te će se igrač stvoriti na pola izvan objekta terena. Zato je potrebno uvećati visinu pozicije stvaranja za visinu objekta igrača.

3.5.2 Mehanika kretanja igrača

Igrač se kreće na temelju virtualne komandne palice (eng. *virtual joystick*) i korisnikovog povlaćenja prsta lijevo desno po ekranu mobilnog uređaja. Za igračevo kretanje zadužena je PlayerMovement klasa. U toj klasi se objekt igrača okreće i kreće koristeći Unity Job sistem, odnosno pomoću višenitnog programiranja. Dizajner igrice može odrediti brzinu kretanja i jačinu skakanja igrača također može postaviti kontrolore kretnji. Kada se pokrene level igrač se stavlja u posebnu vrstu polja koja je ugrađena u pokretač igrica pod nazivom TransformAccessArray. Takva polja se nazivaju izvorna polja jer ih procesor može brzo obraditi i bliže su strojnom jeziku nego obična C# polja. Nakon toga se na svakom ažuriranju sličice provjerava korisnikov unos sa virtualnih komandnih palica. Ukoliko je korisnik pomaknio virtualnu komandnu palicu tada se u drugoj programskoj niti obrađuju podatci na temelju čijih se igrač pomiče. Ukoliko se igrač pomaknuo varijabla isPlayerMoving se postavlja u istinito stanje, u suprutnome je ona u lažnom stanju. Zatim se u drugoj programskoj niti provjerava da li je korisnik povukao prst preko ekrana sa lijeva na desno ili sa desno na lijevo. Na temelju obrađenih podataka iz te programske niti igrač se okreće lijevo ili desno. Sva obrada podataka iz druge programske niti i kretanje igrača se odvija u ugrađenoj FixedUpdate funkciji. FixedUpdate funkcija se poziva svakih nekoliko milisekundi nakon što pokretač video igrica obradi podatke dobivene iz pokretača zaduženog za fiziku. U programskom kodu 3.13 je prikazana FixedUpdate funkcija.

```
private void FixedUpdate()
{
        // Movement
        movementHandle.Complete();
        MovementJob movementJob = new MovementJob()
        {
                movementSpeed = movementSpeed,
                horizontalAxis = -moveJoystick.virtualAxis.x,
                jumpAxis = Input.GetAxisRaw("Jump"),
                jumpPower = jumpPower,
                verticalAxis = -moveJoystick.virtualAxis.y,
                isPlayerAiming = animator.GetBool("Aiming"),
                deltaTime = Time.deltaTime
        };
        movementHandle = movementJob.Schedule(playerTransform);
        JobHandle.ScheduleBatchedJobs();

        if (moveJoystick.virtualAxis != Vector2.zero)
        {
                isPlayerMoving = true;
        }
        else
        {
                isPlayerMoving = false;
        }

        animator.SetBool("Walking", isPlayerMoving);
}
```

Programski kod 3.13. Prikaz tijela FixedUpdate funkcije.

U programskom kodu 3.14 je prikazan višenitni programski kod za igračeve kretnje koji na temelju informacija sa komandne palice ostvaruje igračevu sa jedne pozicije na drugu.

```
public struct MovementJob : IJobParallelForTransform
{
    [ReadOnly] public float movementSpeed;
    [ReadOnly] public float horizontalAxis;
    [ReadOnly] public float jumpAxis;
    [ReadOnly] public float jumpPower;
    [ReadOnly] public float verticalAxis;
    [ReadOnly] public bool isPlayerAiming;
    [ReadOnly] public float deltaTime;

    public void Execute(int index, TransformAccess transform)
    {
        Vector3 movement = new Vector3(horizontalAxis, jumpAxis * jumpPower, verticalAxis);
        transform.position += movement.normalized * movementSpeed * deltaTime;

        if (!isPlayerAiming && movement != Vector3.zero)
        {
            movement.y = 0f;
            transform.rotation = Quaternion.LookRotation(movement);
        }
    }
}
```

Programski kod 3.14. Prikaz višenitnog programskog koda koji pomiče igrača.

Za obradu podataka s ekrana za potrebe virtualne komandne palice se brine VirtualJoystick klasa. Klasa se mora postaviti na sliku koja se nalazi na korisnikovom sučelju

koja prikazuje virtualnu komandnu palicu. VirtualJoystick klasa je proširena s tri sučelja. IDragHandler, IPointerUpHandler i IPointerDownHandler. Kada korisnik dodirne kružnicu na slici i krene ju povlačiti u stranu do kraja kružnice koja simbolizira pozadinu komadne palice tada se javlja događaj koji pokreće OnDrag metodu. Kada korisnik pusti komadnu palicu tada se virtualAxis varijabla resetira na nulu, a držač komadne palice se vraća na sredinu pozadinske slike. U programskom kodu 3.15 je prikazano tijelo VirtualJoystick klase.

```
public class VirtualJoystick : MonoBehaviour, IDragHandler, IPointerUpHandler, IPointerDownHandler
{
    public Vector2 virtualAxis;

    private Image joystickBcg;
    private Image joystickHandler;

    private void Start()
    {
        joystickBcg = GetComponent<Image>();
        joystickHandler = transform.GetChild(0).GetComponent<Image>();
    }

    public virtual void OnDrag(PointerEventData eventData)
    {
        Vector2 pos;
        if (RectTransformUtility.ScreenPointToLocalPointInRectangle(joystickBcg.rectTransform,
eventData.position, eventData.pressEventCamera, out pos))
        {
            pos.x = (pos.x / joystickBcg.rectTransform.sizeDelta.x);
            pos.y = (pos.y / joystickBcg.rectTransform.sizeDelta.y);

            virtualAxis = new Vector2(pos.x * 2 + 1f, pos.y * 2 - 1f);

            if(virtualAxis.magnitude > 1f)
            {
                virtualAxis = virtualAxis.normalized;
            }

            if (joystickHandler != null)
            {
                joystickHandler.rectTransform.anchoredPosition = new Vector2(virtualAxis.x *
(joystickBcg.rectTransform.sizeDelta.x / 3f), virtualAxis.y * (joystickBcg.rectTransform.sizeDelta.y /
3f));
            }
        }
    }

    public virtual void OnPointerDown(PointerEventData eventData)
    {
        OnDrag(eventData);
    }

    public virtual void OnPointerUp(PointerEventData eventData)
    {
        virtualAxis = Vector3.zero;
        joystickHandler.rectTransform.anchoredPosition = Vector3.zero;
    }
}
```

Programski kod 3.15. Prikaz tijela klase VirtualJoystick.

3.5.3 Mehanika igračeva život

Za igračev život zadužena je klasa PlayerHealth koja je produžena sa sučeljem nazvanim IEntityHealth. Sučelja naime sadrže listu metoda i varijabli koje klasa ili struktura trebaju implementirati kako bi mogli biti u potpunosti prošireni. U našem slučaju sučelje IEntityHealth ima dvije funkcije OnEntityTakeDamage i OnEntityDies koje se pozivaju ako igrač ili neprijatelj (eng. AI) prime štetu ili umru. U programskom kodu 3.16 je prikazano tijelo sučelja IEntityHealth.

```
public interface IEntityHealth
{
    void OnEntityTakeDamage(float amount, Vector3 hitPosition);
    void OnEntityDies();
}
```

Programski kod 3.16. Prikaz tijela sučelja IEntityHealth.

Igračev život je prikazan putem klizača na dnu ekrana. Kada igrač primi štetu ekran brzo pocrveni i odmah se makne crvena boja kako bi se igraču dočaralo da je primio štetu. Također se njegov život na klizaču smanjuje. Kada igračev život bude manji od 0 tada on umire, te se na ekranu pokazuje tekst o gubitku života i mogućnosti da se ponovi nivo ili generira novi. U programskom kodu 3.17 je prikazana tijela OnEntityTakeDamage i OnEntityDies funkcija.

```
public void OnEntityTakeDamage(float amount, Vector3 hitPosition)
{
        currentHealth -= amount;
        healthSlider.value = currentHealth;
        isDamaged = true;

        damageAudio.Play();

        if (currentHealth <= 0 && !isDead)
        {
            OnEntityDies();
        }
}

public void OnEntityDies()
{
        GetComponentInChildren<PlayerShooting>().DisableEffects();
        isDead = true;
}
```

Programski kod 3.17. Prikaz tijela OnEntityTakeDamage i OnEntityDies funkcija.

3.5.4 Mehanika pucanja iz puške

Igrač u igri posjeduje pušku iz koje ispaljuje lasere prema vanzemaljcima. Pucanje se odvija kada igrač klikne gumb za pucanje na svome ekranu. Dizajner video igrica može u Unityu namiještati koliko pojedino oružje može oduzimati život neprijateljima, te na koju daljinu može

pojedino oružje pucati. Igračevo pucanje je isprogramirano uz pomoć višenitnog programiranja jer se ono može više puta u jednoj sekundi dogoditi pa je bolje odraditi odmah u više jezgri nego opteretiti samo jednu jezgru korisnikovog uređaja. U programskom kodu 3.18 je prikazano tijelo WeaponShootJob funkcije.

```
private void WeaponShootJob()
{
        // Effects
        gunLight.enabled = true;
        gunLineRenderer.SetPosition(0, transform.position);
        gunLineRenderer.enabled = true;
        gunParticles.Play();
        audioSource.Play();

        raycastCommands = new NativeArray<RaycastCommand>(1, Allocator.TempJob);
        raycastHits = new NativeArray<RaycastHit>(1, Allocator.TempJob);

        var raycastCmdJob = new RaycastCommandJob()
        {
                origin = transform.position,
                direction = transform.forward,
                range = shootingRange,
                layerMask = shootableMask,
                rayCommands = raycastCommands
        };
        raycastOriginHandler = raycastCmdJob.Schedule(1, 64);
        raycastCmdHandler = RaycastCommand.ScheduleBatch(raycastCommands,
                raycastHits,
                64,
                raycastOriginHandler);
        raycastCmdHandler.Complete();

        if(raycastHits[0].point != Vector3.zero)
        {
                AiHealth entityHealth = raycastHits[0].collider.GetComponent<AiHealth>();
                if (entityHealth != null)
                {
                        entityHealth.OnEntityTakeDamage(shootDamage, raycastHits[0].point);
                        if (entityHealth.currentHealth <= 0f)
                        {
                                if (OnEnemyDies != null)
                                {
                                        OnEnemyDies(raycastHits[0].collider.gameObject);
                                }
                        }
                }
                gunLineRenderer.SetPosition(1, raycastHits[0].point);
        }
        else
        {
                gunLineRenderer.SetPosition(1, transform.position + transform.forward *
shootingRange);
        }

        StartCoroutine(DisableEffectsCoroutine());
        raycastCommands.Dispose();
        raycastHits.Dispose();
}
```

Programski kod 3.18. Prikaz WeaponShootJob funkcije.

Kada igrač pritisne tipku poziva se WeaponShootJob metoda koja zatim prikazuje vizualne efekte pucanja iz puške, te započinje pripreme za Job sustav. Prvo se pokreće prvi

posao koji sprema zrake hitca u RaycastCommand tip varijable. Tu se spremaju početne informacije zrake kao što je početak, smjer, veličina i slojna maska zrake. Zatim nakon što posao završi vraća informacije u globalnu varijablu. Nakon toga se započinje drugi posao koji ispaljuje zrake prema informacijama zadanim u prethodnom poslu gdje se informacije o pogotku zrake sprema u globalnu varijablu raycastHits. Ako je zraka nešto pogodila onda će se zraka lasera zaustaviti do točke pogotka i po mogućnosti nanijeti štetu neprijatelju koji posjeduje AiHealth komponentu. U suprotnom linija lasera se zaustavlja na maksimalnom dometu oružja. Na kraju funkcije se sve varijable čiste kako ne bi došlo do curenja memorije (eng. memory leak). U programskom kodu 3.19 je prikazano tijelo poslova potrebnih za igračevo pucanje.

```
private struct RaycastCommandJob : IJobParallelFor
{
     [ReadOnly] public Vector3 origin;
     [ReadOnly] public Vector3 direction;
     [ReadOnly] public float range;
     [ReadOnly] public int layerMask
     [WriteOnly] public NativeArray<RaycastCommand> rayCommands;

     public void Execute(int index)
     {
          rayCommands[index] = new RaycastCommand(origin, direction, range, layerMask);
     }
}
```

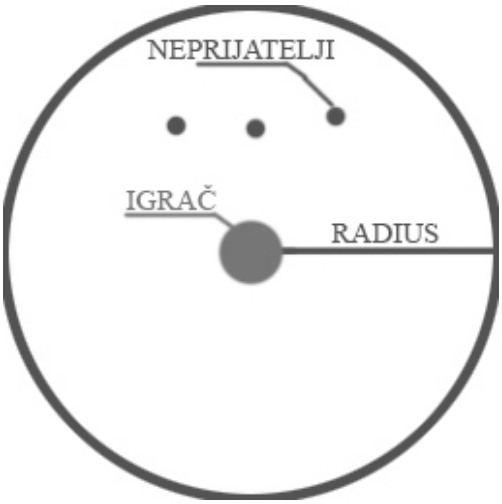
Programski kod 3.19. Prikaz tijela poslova u PlayerShooting klasi.

3.6. Neprijatelji i njihova mehanika

Neprijateljska umjetna inteligencija je napravljena koristeći ugrađene klase u Unityu za umjetnu inteligenciju, ali i koristeći višenitno programiranje u Unity pokretaču igrica.

3.6.1 Mehanika stvaranja neprijatelja

Neprijatelji se stvaraju deset sekundi nakon što se on stvorio na 3D objektu terena. Oni se nasumično stvaraju oko samog igrača na unaprijed definiranom radijusu kružnice. Njihovo stvaranje je napravljeno u višenitnom programiranju kako ne bi utjecalo na padove sličica po sekundi u igrici. Na slici 3.25 je prikazan način stvaranja neprijatelja oko igrača.



Sl. 3.4. Prikaz načina stvaranja neprijatelja oko igrača.

Kako neprijateljima ne bi dugo vremena trebalo da dođu do samog igrača centar kružnice za stvaranje neprijatelja je uvijek igrač. Neprijatelji se stvaraju u valovima i svakim valom neprijatelja je sve više i teže ih je ubiti. Neprijatelji se stvaraju svakih 3 sekunde, a njihov maksimalan broj je 50. Kada igrač ubije sve neprijatelje unutar jednog vala onda se pokreće novi val napada koji povećava težinu i brojnost neprijatelja. Za stvaranje neprijatelja se neprestano svakih 3 sekunde poziva metoda SpawnEnemy. Ta metoda koristi metodu RandomPoint kako bi dobila nasumičnu poziciju oko igrača. Kada odluči gdje se treba objekt neprijatelja stvoriti tada se stvara objekt neprijatelja. Nakon što se stvori objekt neprijatelja algoritam na temelju broju vala napada treba li se promijeniti težina neprijatelja. Metoda RandomPoint je slična metodama korištenim za stvaranje igrača na 3D objekt terena samo što se ovdje koriste ugrađene funkcije za ispucavanje zraka samo na teren po kojem se može kretati neprijatelj. Zraka za određivanje pozicije stvaranja se ispucava iznad najviše točke terena. Ako zraka pogodi teren tada se ta točka vraća u SpawnEnemy metodu. U programskom kodu 3.20 je prikazane funkcije SpawnEnemy i RandomPoint.

```
private void SpawnEnemy()
{
        if (currentSpawnedEntities >= numberOfEnemies)
        {
                CancelInvoke();
                return;
        }
        Vector3 spawnPos;
        RandomPoint(playerTransform.position, 6.0f, out spawnPos);
        spawnPos.y += 0.8f;
        GameObject enemy = PoolManager.instance.ReuseObject(enemyObject, spawnPos,
Quaternion.identity);
        NavMeshAgent enemyAgent = enemy.GetComponent<NavMeshAgent>();
        AiDifficultyManager difficulty = enemy.GetComponent<AiDifficultyManager>();
        enemyAgent.enabled = true;
        enemyAgent.Warp(spawnPos);
        difficulty.entityDifficulty = currentDifficulty;

        if (waveManager.currentWave % waveEnemyDifficultyUpdate == 0 &&
                currentDifficulty != EntityDifficulty.DIFFICULTY_ACE_PLAYER &&
                currentSpawnedEntities == 0)
        {
                difficulty.entityDifficulty += 1;
                currentDifficulty = difficulty.entityDifficulty;

                if (currentDifficulty >= EntityDifficulty.DIFFICULTY_ACE_PLAYER)
                        currentDifficulty = EntityDifficulty.DIFFICULTY_ACE_PLAYER;
        }
        currentSpawnedEntities++;
}

private struct PrepareRandomPoint : IJob
{
        [ReadOnly] public Vector3 center;
        [ReadOnly] public float radius;
        [ReadOnly] public Vector3 randomValue;
        [WriteOnly] public NativeArray<Vector3> randomPoint;

        public void Execute()
        {
                randomPoint[0] = center + randomValue * radius;
        }
}
```

Programski kod 3.20. Prikaz programskog koda za stvaranja neprijatelja oko igrača.

Valovi napada su prisutni u svim video igrama koje su akcijskog žanra. U najpopularnijim igrača napadaju neprijatelji sve dok vi ne ostvari napredak u nivou. Ovdje je sustav valova napada nešto drukčiji. Za svaki val napada postoje prethodno određeni podatci kao što su težina neprijatelja i koliko je stvoreno neprijatelja na levelu. Kada igrač ubije sve neprijatelje stvorene u trenutnom valu napada povećava se val napada. Za valove napada zadužena je WaveManager klasa. U programskom kodu 3.21 je prikazan programski kod za upravljanje valovima napada.

```
private void UpdateAliveEnemies(int count)
{
        currentAliveEnemies = count;
        Debug.Log("Current alive enemies: " + currentAliveEnemies);
}
private void UpdateAliveEnemiesDown(GameObject enemy)
{
        currentAliveEnemies -= 1;

        if(currentAliveEnemies <= 0)
        {
                currentAliveEnemies = 0;
                StartCoroutine(SpawnNewEnemies());
        }
}
IEnumerator SpawnNewEnemies()
{
        yield return new WaitForSeconds(newWaveWaitTime);

        currentWave += 1;
        waveText.SetText("Wave: {0}", currentWave);

        if (OnNewWaveStarts != null)
        {
                OnNewWaveStarts();
        }
}
```

Programski kod 3.21. Prikaz programskog koda važnog za upravljanje valovima napada.

3.6.2 Mehanika kretanja neprijatelja

Ugrađena umjetna inteligencija kod neprijatelja koristi ugrađenu klasu NavMeshAgent koja omogućava lako kretanje neprijatelja bez dodatnog programiranja algoritma za pronalaženje puta. Da bi se neprijatelj mogao kretati po terenu potrebno je postaviti NavMeshSurface komponentu na teren i generirati ga kada se generira 3D objekt terena. Kada se generira NavMeshSurface tada umjetna inteligencija zna koje objekte mora zaobići ili kuda se ne može proći. U programskom kodu 3.22 je prikazan programski kod za generiranje NavMeshSurface terena nakon što se generira 3D objekt terena.

```
public class NavMeshBuilder : MonoBehaviour
{
    public delegate void NavMeshBuildDelegate();
    public event NavMeshBuildDelegate OnNavMeshBuilt;

    private void Awake()
    {
        DecorationsSpawner decorations = FindObjectOfType<DecorationsSpawner>();
        decorations.OnDecorationsSpawned += BuildNavMesh;
    }

    private void BuildNavMesh()
    {
        StartCoroutine(DelayedBuildNavMesh());
    }

    private IEnumerator DelayedBuildNavMesh()
    {
        yield return new WaitForSeconds(1f);

        NavMeshSurface surface = GetComponent<NavMeshSurface>();
        surface.BuildNavMesh();

        if(OnNavMeshBuilt != null)
        {
            OnNavMeshBuilt();
        }
        yield return null;
    }
}
```

Programski kod 3.22. Prikaz programskog koda za generiranje NavMeshSurface terena.

Bez generiranog NavMeshSurface terena umjetna inteligencija se ne može ni stvoriti jer Unity pokazuje grešku da se pokušava stvoriti NavMeshAgent na teren bez NavMeshSurface komponente i generiranog NavMesh terena. Za kretanje neprijatelja zadužena je AiMovement klasa koja na svakom ažuriranju sličica u igrici provjerava jeli igrač živ. Ako je igrač živ onda ga neprijatelj prati i pokušava ubiti, a ako je mrtav onda se gasi NavMeshAgent komponenta koja omogućava praćenje igrača. NavMeshAgent klasa ima pogrešku jer unatoč varijabli za zaustavljanje neprijatelji se ne zaustavljaju kada igrač stoji na mjestu i stalno pokušavaju doći na poziciju igrača što rezultira iritantnim trzanjem objekta neprijatelja koji se čini kao da pokušava gurati igrača, a ne napasti ga. Taj problem je riješen tako da se provjeri dali igrač stoji i koja je udaljenost igrača od neprijatelja. Ako je igrač stoji na mjestu i ako je udaljenost igrača i neprijatelja manja od 0.256 onda se neprijatelj zaustavlja i napada igrača. Ako se igrač udalji od neprijatelja tada neprijatelj nastavlja s potjerom. U programskom kodu 3.23 je prikazan programski kod za kretanje neprijatelja.

```
private void Update ()
{
	if (GamePause.isGamePaused || playerHealth.currentHealth <= 0f)
	{
		if (!navAgent.isOnNavMesh)
			return;

		navAgent.isStopped = true;
		aiRigidbody.velocity = Vector3.zero;
		animator.SetBool("Moving", false);
		return;
	}

   if (Vector3.Distance(target.position, transform.position) <= 0.45f &&
!playerMovement.isPlayerMoving)
   {
		navAgent.isStopped = true;
		aiRigidbody.velocity = Vector3.zero;
		animator.SetBool("Moving", false);
   }
   else
   {
		animator.SetBool("Moving", true);
		navAgent.isStopped = false;
		navAgent.SetDestination(target.position);
   }
}
```

Programski kod 3.23. Prikaz programskog koda za kretanje neprijatelja po terenu.

3.6.3 Mehanika napadanje igrača

Mehanika napadanja igrača je vrlo jednostavna. Ako je igrač blizu neprijatelja tada ga neprijatelj napada svake 0.3 sekunde. Dizajner igrice može unutar Unitya uređivati štetu koju neprijatelj može nanijeti igraču i interval napada. Oko 3D objekta neprijatelja je napravljen sudarač (eng. collider) koji je namješten da bude okidač (eng. trigger). Kada igrač uđe unutar tog sudarača onda se poziva metoda OnTriggerEnter koja postavlja globalnu varijablu isEnemyInRange u istinu i okida događaj OnPlayerInRange sa parametrom u istini. Ako se detektira da je igrač unutar okidača i ako je on živ tada neprijatelj započinje napadanje igrača svakih 0.3 sekunde. Kada igrač izađe izvan raspona okidača tada se poziva OnTriggerExit metoda koja postavlja globalnu varijablu isEnemyInRange na laž i okida događaj OnPlayerInRange sa parametrom u laži. U programskom kodu 3.24 je prikazan progamski kod zadužen za mehaniku napadanja igrača.

```
private void OnTriggerEnter(Collider other)
{
	if(!isEnemyInRange && other.gameObject == target)
	{
		isEnemyInRange = true;
	}
}
private void OnTriggerExit(Collider other)
{
	if(isEnemyInRange && other.gameObject == target)
	{
		StartCoroutine(EnemyNotInRangeDelayed());
	}
}
private void Update()
{
	attackTimer += Time.deltaTime;

	if(attackTimer >= attackCoolTime && isEnemyInRange && playerHealth.currentHealth >
0f)
	{
		if (!GamePause.isGamePaused)
			AttackPlayer();
	}
}
private void AttackPlayer()
{
	if (playerHealth.currentHealth > 0f)
	{
		playerHealth.OnEntityTakeDamage(damage, Vector3.zero);
	}
	attackTimer = 0f;
	animator.SetTrigger("Attacking");
}
```

Programski kod 3.24. Prikaz programskog koda za napadanje igrača.

3.6.4 Mehanika života

Kao i kod igrača neprijatelj svoju AiHealth klasu proširuje sa IEntityHealth sučeljem. Da bi se klasa upotpunosti proširila potrebno je implementirati OnEntityTakeDamage i OnEntityDies metode. Kada igrač puca neprijatelja tada se poziva OnEntityTakeDamage metoda koja tada nanosi štetu nerijatelju i stvara efekte štete. Ako je neprijateljev život manji od nule tada se poziva OnEntityDies metoda. U toj metodi se tada gase sve komponete važne za kretanje neprijatelja kako bi se izbjegli pogreške prilikom smrti. Nakon dvije sekunde se gasi 3D objekt neprijatelja kako bi se mogao kasnije ponovno iskoristiti. U programskom kodu 3.25 je prikazan programski kod mehanike života kod neprijatelja.

```
public void OnEntityDies()
{
	if (capsuleCollider != null)
	{
		capsuleCollider.isTrigger = true;
	}

	GetComponent<AiMovement>().enabled = false;
	GetComponent<NavMeshAgent>().enabled = false;
	GetComponent<Rigidbody>().isKinematic = true;
	animator.SetTrigger("Died");

	isDead = true;
	StartCoroutine(DestroyEntity());
}
public void OnEntityTakeDamage(float amount, Vector3 hitPosition)
{
	if (isDead)
		return;

	currentHealth -= amount;
	damageAudio.Play();

	if (hitParticles != null)
	{
		hitParticles.transform.position = hitPosition;
		hitParticles.Play();
	}

	if(currentHealth <= 0f)
	{
		OnEntityDies();
	}
}
private IEnumerator DestroyEntity()
{
	yield return new WaitForSeconds(2.0f);
	gameObject.SetActive(false);
	StopCoroutine(DestroyEntity());
}
```

Programski kod 3.25. Prikaz programskog koda mehanike neprijateljevog života.

3.7. Kamera i vizualni efekti

Za što bolji doživljaj prilikom igranja video igrica potrebni su vizualni efekti i kamere. Radi optimizacije igre za Android uređaje ona radi na posebnom sustavu za prikaz stvari na ekranu uređaja pod nazivom Lightweight Render Pipeline skračenog naziva LWRP. Lightweight Render Pipeline omogućuje lijepi prikaz 3D objekata i vizualnih efekata uz malo padanje sličica po sekundi. Iz toga sustava za prikaz objekata na ekran uređaja su izbačene stvari koje nisu potrebne slabim uređajima kao što su posebne sjene i načini prikaza svjetlosti na 3D objektima. Unity pokretač igrica također ima ugrađen paket za vizaulne efekte pod nazivom Post Processing Stack. Najveći problem vizualnih efekata su veliki padovi sličica po sekundi jer oni zahtjevaju veliku procesorsku snagu. Stoga u ovome projektu video igra koristi samo sjene u kutevima

ekrana kako scena nebi bila previše svijetla jer je radnja igre ipak u svemiru. Ostali efekti kao što je dodavanje boja ili dodatne svjetline 3D objekata su previše zahtjevni i uzrokuju pad sličica po sekundi ispod 10.

Video igra najčešće ima više od jedne kamere jer su posebne kamere za kretanje igrača, a posebne za scene koje dočaravaju priču igre. Ovdje igra koristi ugrađeni sustav za kamere pod nazivom Cinemachine i ugrađeni sustav za upravljanje objektima u pokretaču igrica pod nazivom Timeline. Cinemachine je skup programskih kodova koji omogućuju da se s jednom kamerom u sceni upravlja putem više virtualnih kamera. Svaka virtualna kamera posjeduje informacije koje tada prenosi glavnoj kameri u sceni kao što su: pozicija kamere, koji objekt kamera treba pratiti ili udaljenost kamere od objekta koji prati. Putem Timeline sustava se upravlja kada je koja virtualna kamera uključena, ali dodaje se i efekt prijelaza s jedne virtualne kamere na drugu. Prva virtualna kamera gleda u tek generirani teren dok se igrač ne stvori. Kada se igrač stvori tada se prelazi na drugu kameru koja se nalazi iznad samog igrača i koja prati njegove kretnje. Prilikom prijelaza s prve virtualne kamere na drugu uključuje se grafičko sučelje što označava početak igre. Kada igrač pogine tada se pokreče animacija koja gasi grafičko sučelje za igračeve kretnje, a animirano uključuje grafičko sučelje za kraj igre.

3.8. Zvučni efekti i glazba

Uz vizualne efekte također su jako važni i zvučni efekti. Da bi igrač imao što bolji doživljaj igre efekti moraju dočaravati trenutno stanje igre. Projekt trenutno ima tri zvučna efekta: zvuk laserskog pištolja, zvuk kada igrač prima štetu i zvuk kada neprijatelj prima štetu. Svaki zvučni efekt se automatski pušta putem ugrađene Unity metode pod nazivom Play(). Svi zvučni efekti su unutar 3D područja i malo su glasniji od glazbe unutar igre što omogućava veći doživljaj igrača.

Sve pjesme u projektu su posebnog tipa pod nazivom ScriptableObjects. Unutar tih objekata se dodaje naziv pjesme i njen autor, jačina zvuka kojom će se pjesma pustiti, njen maksimalan ton i njen audio klip. ScriptableObjects omogućavaju lakše pravljenje i grupiranje pjesama jer se ne mora baviti s jako velikom klasom što otežava kasnije uređivanje koda. U programskom kodu 3.26 je prikazana MusicData klasa.

```
[CreateAssetMenu()]
public class MusicData : ScriptableObject
{
    public AudioClip songClip;
    public string songInfo;

    [Range(0, 1)]
    public float songVolume;

    [Range(0.1f, 1)]
    public float songPitch;
}
```

Programski kod 3.26. Prikaz tijela MusicData klase.

U projektu postoji više pjesama koje se nasumično puštaju. Kada jedna pjesma završi tada se druga kreće puštati. Jako je važno paziti na dozvole od strane autora glazbe jer se neka glazba ne smije koristiti u komercijalne ili ikakve druge svrhe osim slušanja. Zbog dozvole od autora da se smije glazba puštati unutar projekta uz njegovo navođenje dodana je animacija koja prikazuje autora i naziv pjesme ispod linije koja prikazuje igračev život. Pjesme unutar projekta su smanjenje na minimum jer nisu ovdje da budu u središtu igračeve pažnje već da poboljšaju doživljaj. U programskom kodu 3.27 je prikazana klasa MusicManager koja nasumično pušta pjesme.

```
public class MusicManager : MonoBehaviour
{
    public static MusicManager instance;

    public GameObject songInfoText;
    public MusicData[] musicData;

    private AudioSource audioSource;

    private void Start()
    {
        audioSource = GetComponent<AudioSource>();
    }
    private void Update()
    {
        if(!audioSource.isPlaying && SoundManager.isSoundOn)
        {
            PlayRandomSong();
        }
    }
    private void PlayRandomSong()
    {
        int index = Random.Range(0, musicData.Length);
        audioSource.clip = musicData[index].songClip;
        audioSource.volume = musicData[index].songVolume;
        audioSource.pitch = musicData[index].songPitch;
        audioSource.Play();

songInfoText.GetComponentInChildren<TextMeshProUGUI>().SetText(musicData[index].songInfo);
        songInfoText.GetComponent<Animation>().Play();
    }
}
```

Programski kod 3.27. Prikaz tijela MusicManager klase.

Na samom učitavanju scene uzima se referenca za ugrađenu komponentu AudioSource koja omogućava puštanje zvukova u Unity pokretaču igrica. Na svakom ažuriranju sličica u video igri provjerava se dali se pušta pjesma iz izvora zvuka, ako ne onda se pušta nasumično odabrana pjesma. Za puštanje nasumično odabrane pjesme zadužena je PlayRandomSong metoda. U prvom dijelu metode se uzima indeks u polju zvukova koji može biti od 0 do maksimalnog broja elemenata polja. Tada se na izvor zvuka stavlja audio klip, jačina zvuka i visina tona. Nakon što se to postavi zvuk se pušta, i pokreće se animacija za prikaz informacija o trenutno puštenoj pjesmi.

Svi zvukovi unutar igre se mogu ugasiti na glavnom izborniku ili izborniku unutar igre. Ako su zvukovi ugašeni tada se pjesme u pozadini ne puštaju niti igrač čuje zvučne efekte. U programskom kodu 3.28 je prikazano tijelo SoundManager klase.

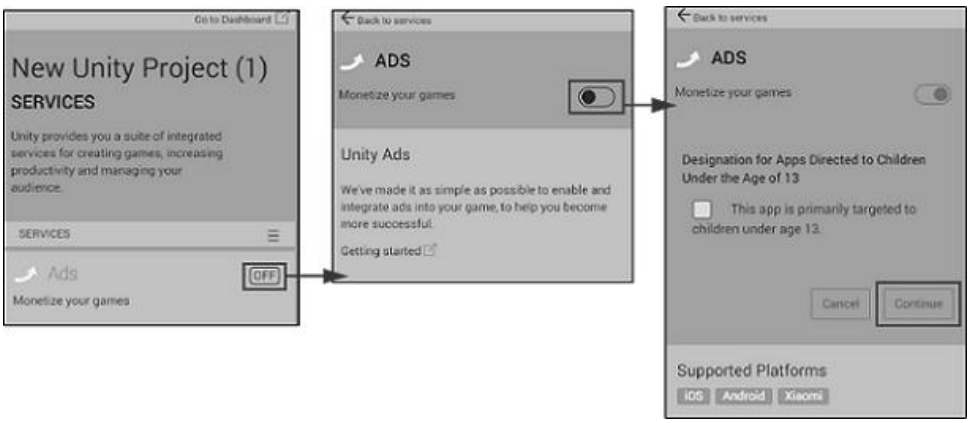
```
public class SoundManager : MonoBehaviour
{
    public static bool isSoundOn = true;

    #region UnityFunctions
    private void Start()
    {
        if (!isSoundOn)
        {
            AudioListener.volume = 0f;
        }
        Else.
        {
            AudioListener.volume = 1f;
        }
    }
    #endregion

    #region ButtonFunctions
    public static void ToggleGameSounds()
    {
        if (!isSoundOn)
        {
            isSoundOn = true;
            AudioListener.volume = 1f;
        }
        else
        {
            isSoundOn = false;
            AudioListener.volume = 0f;
        }
    }
    #endregion
}
```

Programski kod 3.28. Prikaz tijela SoundManager klase.

Trenutno stanje uključenosti zvuka u video igri se sprema u statičnoj varijabli isSoundOn. Statične varijable su varijable čija vrijednost se ne mijenja dok se program izvodi ili dok se vrijednost varijable ne promjeni unutar programskog koda. SoundManager klasa se nalazi u glavnom izborniku i kada se on pokrene automatski se isSoundOn varijabla se stavlja u stanje istine. Na svakom pokretanju scena, nakon postavljanja stanja isSoundOn varijable, se provjerava jeli zvuk uključen ili isključen. Ako je zvuk isključen tada se on isključuje unutar scene pomoću AudioListener klase. AudioListener je ugrađena klasa koja omogućuje da se zvukovi unutar scene čuju bez nje to ne bi bilo moguće. Ako je zvuk uključen jačina zvuka se automatski stavlja na 100 posto. U klasi također postoji statična metoda pod nazivom ToggleGameSounds koja isključuje ili uključuje zvuk unutar igre. Statične metode su metode koje se mogu pozvati u bilo kojem dijelu programskog koda bez dodatne inicijalizacije SoundManager klase.

3.9. Zarada u video igrama

Dvije najčešće strategije zarade kod video igrica koje ciljaju tržište mobilnih uređaja su kupovina unutar aplikacije i video oglasi. Kupovina unutar aplikacije (eng. *in app purchase*) se koristi kod video igrica koje imaju virtualnu valutu kojom se može nadograditi igrivi lik ili kupiti prolaz nivoa. Najunosniji način zarade u video igrama je nagrađivanje igrača za pogledani video. U projektu za ovaj završni rad zarađivat će se putem video oglasa. Unity pokretač video igrica ima ugrađeni sustav video oglasa koji se može vrlo lako aktivirati. Potrebno je samo aktivirati Unity Ads servis putem Unity urednika. U projektu će igrač moći nastaviti tamo gdje je stao ako pogleda oglas. No kako ne bi bilo fer prema ostalima da se može cijelo vrijeme tako raditi igrač će samo jednom imati priliku nastaviti tamo gdje je stao. Na slici 3.5 je prikazan način aktivacije Unity Ads servisa.



Sl. 3.5. Prikaz aktivacije Unity Ads servisa unutar Unity uređivača.

Nakon što se implementiraju oglasi unutar projekta tada Unity generira prihod na temelju broja pregleda plaćenih videa. Što više ljudi koristi video igru više će se novaca zaraditi. Unity automatski bira marketinšku kampanju odnosno na koji način će prikazati video oglas. Od krajnjeg korisnika se može tražiti pregled videa do kraja ili klikove. Veličina prihoda ovisi o puno faktora kao što je platforma i regija odakle dolazi igrač. No najveći faktor koji se najviše prilikom generiranja prihoda je broj igrača koji koriste igru. Za generiranje prihoda se koristi CPM odnosno cijena koju reklamer mora platiti na tisuću korisnika (eng. *cost per mille*)[12].

4. PRIMJERI IZ IGRE

Video igra je žanra pucanja s kamerom iz ptičje perspektive (eng. *top down shooter*), a glavni lik je astronaut kojeg napadaju vanzemaljci. Posebnost ove video igre od ostalih video igrica na tržištu toga žanra je ta što se nivo proceduralno generira putem slike koju je korisnik uslikao neposredno prije samog generiranja nivoa. Ona je prilagođena svim uzrastima jer sadrži vizualne efekte i likove kao da se radi o igri za niže uzraste. Na slici 4.1 su prikazani glavni likovi ove video igre.

Sl. 4.1. Prikaz glavnih likova u video igri.

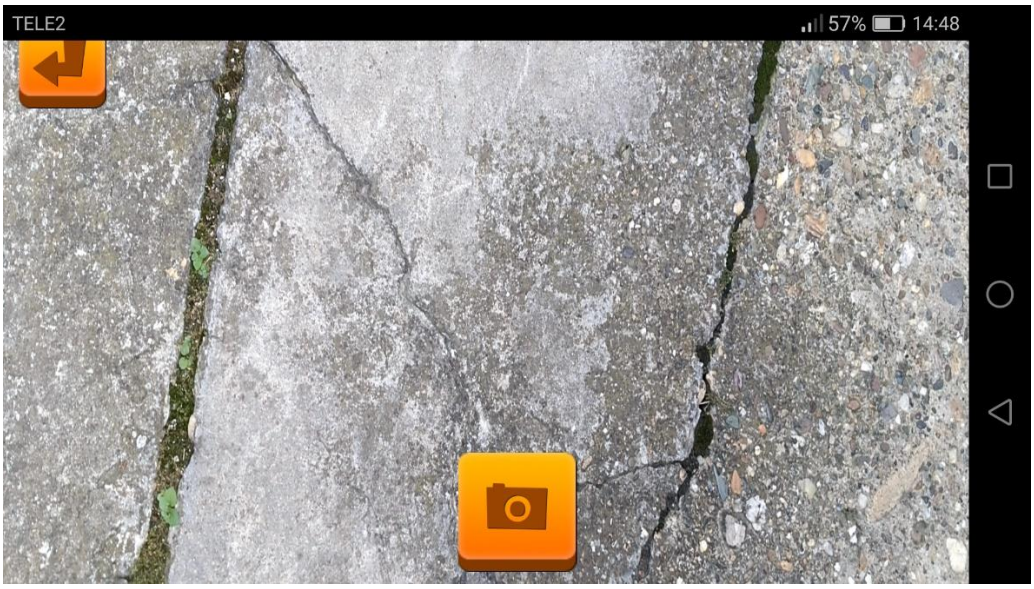
No također sadrži natjecateljski dio igre koji je prilagođen za veće uzraste. Naime cilj u ovoj video igri je ubiti neprijatelje i doći do što višeg vala napada. Svakim valom napada neprijatelji su brži, ali ih se i više stvori nego u prošlom valu što otežava igraču prolazak vala napada. Ako igrač prvi puta umre unutar trenutne igračke sesije tada može nastaviti tamo gdje je umro ukoliko pogleda plaćeni video oglas. No, ako nakon toga umre tada može ponoviti nivo sa istom slikom ili generirati novi nivo sa novom slikom.

4.1. Početak igre

Kada korisnik pokrene video igru prikazuje mu se glavni izbornik dok se u pozadini čuje tematska melodija igre. Glavni izbornik je jednostavan i samo objašnjiv jer koristi tekst i ikone kao i sve ostale mobilne aplikacije na koje su korisnici odavno naučili. Na izborniku imate tri gumba za: pokretanje igre, izlazak iz igre i gašenje zvuka u igri. Na slici 4.2 je prikazan izgled glavnog izbornika.

Sl. 4.2. Prikaz glavnog izbornika nakon ulaska u igru.

Ako korisnik klikne gumb za gašenje zvuka unutar video igre, tada se sva pozadinska glazba i zvučni efekti gase sve dok ih korisnik ponovno ne uključi u izborniku za vrijeme igre. Klikne li korisnik na gumb za pokretanje igre tada se scena s glavnim izbornikom isključuje i uključuje se scena za slikanje slike. Na slici 4.3 je prikazana scena za slikanje slike.



Sl. 4.3. Prikaz scene za slikanje slike mobilnim uređajem.

Preko cijeloga ekrana se prikazuje prikaz sa stražnje kamere korisnikova mobilnog uređaja. Uz to su prikazana dva gumba za: slikanje slike i vraćanje na glavni izbornik. Kada korisnik klikne gumb za slikanje tada se slika sprema u memoriju programa i pokreće se generiranje nivoa.

4.2. Stvaranje igrača i tijek igre

Nakon što je generiran teren 3D objekt igrača se nasumično stvara na terenu, te jedna kamera snima cijeli teren sa svim dekoracijama iz ptičje perspektive. Na slici 4.4 je prikazan generirani nivo iz ptičje perspektive.

Sl. 4.4. Prikaz generiranog nivoa iz ptičije perspektive.

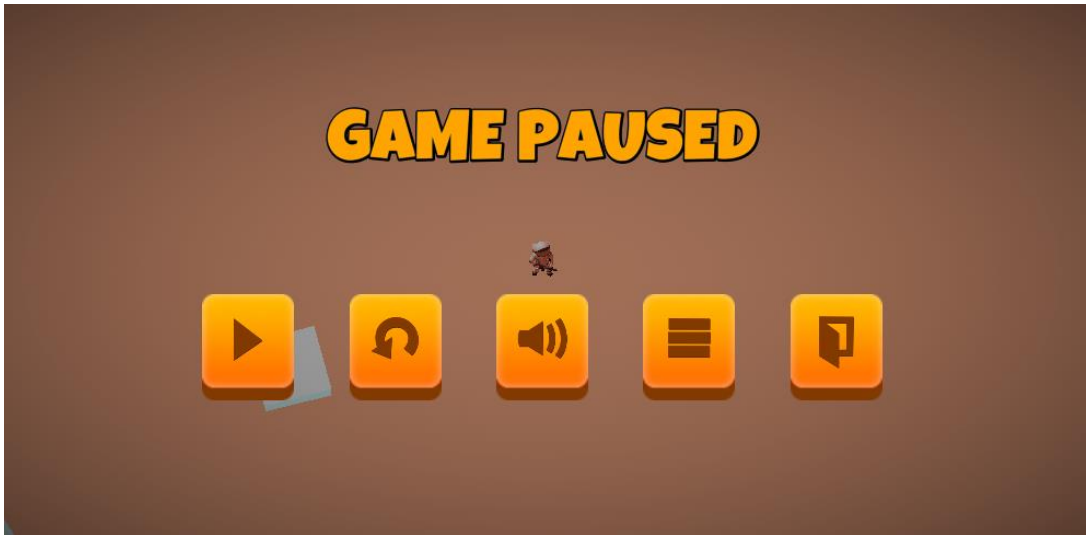
Nakon nekoliko sekundi počinje se izvršavati sekvenca koja prebacuje kameru iznad samog igrača kao u ostalim video igrama toga žanra, te se uključuje korisničko sučelje.

4.2.1 Korisničko sučelje u igre

Korisničko sučelje se sastoji od dva dijela: gornjeg i donjeg. U donjem dijelu prikazano je trenutno stanje igračeva života, te komadna palica koja kontrolira igračeve kretnje i gumb za pucanje iz laserske puške. U gornjem dijelu je prikazan trenutni val napada i gumb za pauziranje video igre koji također uključuje izbornik. Na slici 4.5 je prikazano korisničko sučelje (eng. *user interface*).

Sl. 4.5. Prikaz korisničkog sučelja.

Kada korisnik pritisne ikonu u gornjem desnom uglu tada se video igra pauzira i prikazuje se izbornik. Na slici 4.6 je prikazan izbornik unutar igre.



Sl. 4.6. Prikaz izbornika tijekom pauzirane igre.

Korisnik unutar izbornika može birati želi li nastaviti gdje je stao ili pak ponovno pokrenuti trenutni nivo. Također može birati između odlaska u glavni izbornik ili gašenja video igre. Prije izlaska iz video igre ili odlaska u glavni izbornik korisnika se pita za potvrdu radnje.

4.2.2 Valovi napada

Nakon što se igrač stvorio na teren i nakon što je odrađena sekvenca za prikaz igrača tada započinju valovi napada. Valovi napada se također mijenjaju kada igrač ubije sve neprijatelje

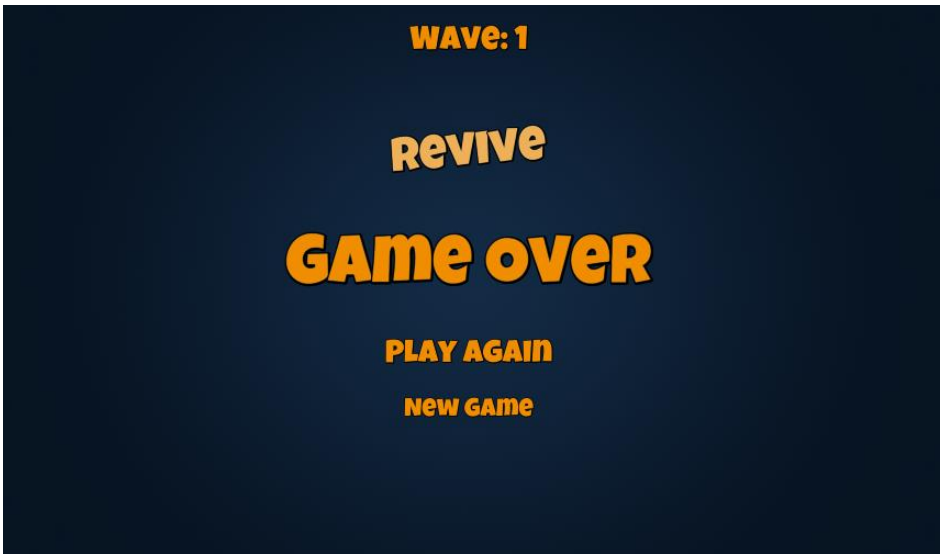
predviđene za trenutni val napada. Između svakog vala napada ima kašnjenje od nekoliko sekundi kako bi igrač mogao predahnuti nakon posljednjeg napada. Svaki val ima predefiniran broj vanzemaljaca za stvaranje oko igrača. Tako se za svaki novi val broj neprijatelja povećava za dva. Maksimalan broj stvorenih neprijatelja na terenu je 50, a to je ograničeno jer se zbog performansi koristi stvaranje objekata u grupe (eng. *object pooling*) odnosno stvaranje 50 objekata prije prikaza scene na korisnikovom ekranu koji su onda ugašeni, te se prema potrebi uključuju ili isključuju. Stvaranje neprijatelja na korisnikov ekran je zapravo uključivanje objekta koji je prethodno isključen, a kada neprijatelj umre tada se pozicija objekta resetira na nulu i objekt se isključuje. Na slici 4.7 je prikazan jedan od vala napada.

Sl. 4.7. Prikaz vala napada.

Kada se vanzemaljci stvore odmah dobivaju naredbu da prate igrača i pokušaju mu nanijeti štetu sve dok je on živ.

4.3. Igračeva smrt i kraj trenutne sjednice

Kada igrač umre odmah se pokreće sekvenca koja pauzira cijelu igru i prikazuje izbornik koji korisniku prikazuje: val napada na kojem je korisnik umro, gumb za nastavljanje igre, gumb za ponovno pokretanje nivoa i gumb za ponovno generiranje nivoa s novom slikom. Na slici 4.8 je prikazan izbornik nakon igračeve smrti.



Sl. 4.8. Prikaz izbornika nakon igračeve smrti.

Gumb za ponovno nastavljanje nakon korisnikove smrti se pojavljuje samo jednom unutar trenutne igračke sesije. Nakon što igrač klikne na taj gumb tada se prikazuje plaćeni oglas od Unity-a nakon kojega igrač nastavlja gdje je stao udaljen od neprijatelja. Kada igrač ponovno umre unutar iste igračke sesije taj mu se gumb više ne prikazuje. Ako korisnik odabere gumb za ponovno pokretanje trenutnog nivoa tada se pokreće nova igračka sesija što znači da se brišu postignuća koja je korisnik postigao tijekom prošle igračke sesije, ali s istom slikom i istim nivoom samo se igrač stvori na drugoj poziciji. To omogućava da korisniku ne bude igra monotona ma koliko god puta ju on ponovno pokrenuo. Posljednji gumb je gumb koji završava trenutnu igračku sesiju i pokreće scenu za slikanje slike s korisnikova mobilnog uređaja. Tada se prošla slika iz memorije briše i slika se nova slika prema kojoj se generira novi nivo.

5. ZAKLJUČAK

Projektni zadatak završnog rada je bio napraviti 3D video igru za Android mobilne uređaje u Unity pokretaču igrica. Za izradu te video igre je bilo potrebno razumjeti tehnologije iza obrade digitalnih slika i proceduralnog generiranja nivoa kao i naučiti višenitno programiranje. Što je oduzelo više vremena nego izrada same video igre jer se tehnologije za obradu digitalne slike nisu česte u programskom jeziku C# već u jezicima kao što je C i C++. Osim toga tehnologije su morale biti primjenjive u Android operacijskom sustavu što je dodatno otežalo stvar jer se nisu gotove biblioteke napravljene za Windows operacijski sustav. Osim toga pravo višenitno programiranje unutar Unity pokretača igrica je novo i još nije u potpunosti implementirano u sam pokretač. Zbog toga se nisu potpuno sve stvari programirale s višenitnim programiranjem.

Za sve tehnologije koje su se iskoristile za izradu projektnog zadatka dane su detaljne teorijske podloge gdje je objašnjeno kako izraditi 3D model i prikazati ga na korisnikovom ekranu, te kako obraditi digitalnu sliku. Osim toga unutar završnog rada je detaljno opisan način izrade video igre od obrade digitalne slike do igračevih mehanika i specijalnih efekata. Također je obrađena tema o monetizaciji video igre gdje je objašnjen način zarade putem Unity Ads servisa koji omogućuje zaradu putem video oglasa. Samu video igru je moguće pokretati na većini mobilnih uređaja od onih s novim verzijama Android-a do onih sa starijim verzijama. Samo će performanse igre ovisiti o procesorskoj snazi centralne i grafičke jedinice. Dan je veliki značaj optimiziranju video igre jer je poželjno da se igra vrti 30 sličica po sekundi na svim uređajima. Zbog toga je i uveden pojam višenitnog programiranja u ovaj projektni zadatak jer ono povećava performanse video igre, osim toga se pazilo da 3D modeli nemaju previše mnogokuta (eng. polygons) jer i to rezultira padu performansi.

U budućnosti će pomoći stečena znanja tijekom izrade ovog završnog rada. No također je plan ovaj projektni zadatak u skorije vrijeme objaviti na Google Play servisu uz dodatne optimizacije i poboljšanja. Jedno od poboljšanja će svakako biti implementacija rangiranja korisnika po lokaciji i korisnikovim prijateljima što će povećati nadmetanje između korisnika.

LITERATURA

[1] Margaret Rouse, 3D modeling, TechTarget, 2016., dostupno na: https://whatis.techtarget.com/definition/3D-modeling [Lipanj 2018.],

[2] Adam Watkins, Creating Games with Unity and Maya: How to Develop Fun and Marketable 3D Games, Focal Press, 2011.,

[3] Matt Schell, Rendering In Unity, Unity Technologies, 2017., dostupno na: https://unity3d.com/learn/tutorials/topics/graphics/rendering-unity [Lipanj 2018.],

[4] Eric Chadwick, Ambient Occlusion, Polycount, 2010., dostupno na: http://wiki.polycount.com/wiki/Ambient_occlusion_map [Lipanj 2018.],

[5] M. Bailey, S. Cunningham, Graphics Shaders: Theory and practice, CRC Press, SAD, 2011.,

[6] W.Y. Lo, S.M. Puchalski, Veterinary Radiology & Ultrasound, Department of Surgical and Radiological Sciences, School of Veterinary Medicine, University of California, 2008.,

[7] Charles Poynton, Video engineering, 2004., dostupno na: http://poynton.ca/notes/video/Constant_luminance.html [Srpanj 2018.],

[8] Hrvatsko strukovno nazivlje, dostupno na http://struna.ihjj.hr/naziv/konvolucija/19228/ [Srpanj 2018.],

[9] L. Shapiro, G. Stockman, Computer Vision, Prentice Hall, 2001.,

[10] R. Buyya, S. T. Selvi, X. Chu, Object-Oriented Programming with Java, Tata McGraw Hill, 2009.,

[11] Unity Tehnologies, Job System Overview, 2018., dostupno na https://docs.unity3d.com/Manual/JobSystemOverview.html [Srpanj 2018.],

[12] Unity Tehnologies, Unity Ads Revenue, 2018., dostupno na https://support.unity3d.com/hc/en-us/articles/205263819-How-do-I-earn-revenue-from-Unity-Ads [Kolovoz 2018.]

SAŽETAK

Zadatak završnog rada je bio napraviti 3D video igru u Unity pokretaču video igrica koja će se moći igrati na Android mobilnim uređajima. Glavna značajka video igre je ta što se nivo proceduralno generira pomoću informacija dobivenih iz obrađenih digitalnih slika. Glavni cilj u video igri je ubiti što više neprijatelja i doći do što većeg vala napada. Svakim novim valom stvara se više neprijatelja koji su brži i spretniji od prošlih. Igrač se kreće pomoću virtualne komadne palice programirane u C# programskom jeziku, a puca pomoću gumba s druge strane ekrana. Rad je podijeljen u tri poglavlja ne brojeći uvod i zaključak. U prvom poglavlju je teorijski opisano pravljenje 3D modela i njihovog prikaza na korisnikove ekrane, također je opisana tehnologija obrade digitalne slike odnosno dana je teorijska podloga za sve algoritme korištene u projektnom zadatku. Osim toga u teorijskom dijelu je napisan dio o višenitnom programiranju koje je korišteno u većini slučajeva tijekom izrade projektnog zadatka. U trećem poglavlju je detaljno opisan tijek izrade video igre od pisanja programskog koda za mehanike igre do specijalnih efekata i kamere. U posljednjem poglavlju je opisana video igra iz korisničke perspektive sve od početka do završetka igre.

Ključne riječi:

Android igra, Space Adventures, Unity Engine, obrada digitalne slike, višenitno programiranje

ABSTRACT

Creating a 3D video game using Unity Game Engine

Goal in this paper is to create 3D Android game in the Unity Game Engine. Main feature of this video game is procedural generation of terrain based on processing a digital image. Goal in video game is to kill as many enemies as possible and reach the biggest wave. With each new wave there are more enemies spawning both faster and stronger than in the previous wave. Player moves via virtual joysticks and fires from the weapon with click on fire button. This paper is divided into three chapters without counting the introduction and conclusion. In first chapter there is a theoretical description of 3D modeling, digital image processing and multithreaded programming. In the second chapter it is described how this video game was made from describing the program code to special effects. In the last chapter the gameplay of this video game from start to finish is described.

Keywords:

Android game, Space Adventures, Unity Engine, digital image processing, multithreaded programming

ŽIVOTOPIS

Matej Arlović rođen je 5.svibnja 1996. godine u Osijeku. U Osijeku završava Osnovnu školu Ljudevita Gaja nakon čega se upisuje u Elektrotehničku i prometnu školu Osijek zanimanja Mehatroničar. Po završetku srednje škole upisuje prediplomski stručni studij Elektrotehnike, smjer Automatika kao redovan student na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija Osijek. Na drugoj godini studija pohađa PHP Akademiju koju su pokrenuli tvrtka Inchoo i FERIT. Tijekom ljeta iste godine pohađa praksu u tvrtci Inchoo kao backend programer.

U Osijeku, rujan 2018. Matej Arlović

_____

(Vlastoručni potpis)