Izrada 2D platformera u programskom alatu Unity

Hraščanec, Dino

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Dino Hraščanec

IZRADA 2D PLATFORMERA U PROGRAMSKOM ALATU UNITY

ZAVRŠNI RAD

Varaždin, 2020.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Dino Hraščanec

Matični broj: 45897/17-R

Studij: Informacijski sustavi

IZRADA 2D PLATFORMERA U PROGRAMSKOM ALATU UNITY

ZAVRŠNI RAD

Mentor/Mentorica:

Doc. dr. sc. Mladen Konecki

Varaždin, lipanj 2020.

Dino Hraščanec

Izjava o izvornosti

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

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

---

Sažetak

U današnje vrijeme, računalne igre globalizirale su cijeli svijet. Računalne igre veoma su popularne već 30 godina. Svakim danom, industrija računalnih igara se sve više i više proširuje, zbog čega se golemi tok novaca pojavljuje oko istog.

Osobno sam veliki ljubitelj računalnih igara od samog djetinjstva. Upravo iz prethodno navedenih razloga, odlučio sam raditi završni rad o „Izradi 2D Platformera u programskom alatu Unity“. U početku završnog rada reći ću nešto općenito o povijesti računalnih igara, povijesti programskog alata Unity te današnjem stanju računalnih igara i programskog alata Unity.

Potom ću krenuti s opisom tematike svog završnog rada, značenju paketa u programskom alatu Unity, pozadinom te kamerom u igri. Potom ću objasniti kontrole (osnovno kretanje, jednostruki skok, dvostruki skok, pucanje i sl.). Također će biti objašnjen i način izrade i prikaza animacija u programskom alatu Unity. Slijedi opis prepreka (šiljci odozdo i kretajuće platforme), opis neprijatelja (hodajući dinosaur, ovješeni bodljikavac, crveni trkač) te način njihove implementacije. U završnom radu također su opisane „stvari za sakupljanje“ poput dodatnih života i kovanica koje doprinose bodovima u igri. Nakon toga slijedi opis zvuka i efekata u igri te način njihove implementacije. Svaki prethodno navedeni dio sastoji se od dva djela: teorijski dio i primjena teorijskog djela u igri.

Kao zasebna točka u ovom završnom radu javlja se grafičko sučelje igre tj. „Game UI“, gdje se nalazi opis početnog zaslona igre, korisničkog sučelja tijekom pokrenute igre te izgleda menija pauze. Kao završna točka ovog završnog rada odabrana je tematika „Poboljšanje performansi igre“ u kojoj se poboljšavaju performanse igre te sprječava punjenje radne memorije nepotrebnim objektima koji u konačnici usporavaju rad igre.

Na kraju završnog rada nalazi se zaključak o programskom alatu Unity, svijetu igara te osobna perspektiva budućnosti programskom alata Unity.

Sadržaj

1. Uvod

Računalne igre (engl. Computer Games) postoje već pedeset godina te su kroz svoju povijest postojanja doživjele velike promjene. Nekada su računalne igre imale tek poneku kontrolu, poneki zvuk i bile su isključivo u 2D obliku. Danas računalne igre imaju veoma velik broj kontrola, pozadinskih zvukova i zvučnih efekata te su pretežito u 3D obliku. Kroz pedeset godina razvoj računalnih igara doživio je ogromni razvoj te se u bližoj, ali i daljnjoj budućnosti očekuje nastavak navedenog trenda.

Računalne igre predstavljaju pretežito način zabave za ljude, iako postoje i načini zarada na računalnim igrama poput streaminga ili izrade i prodaje računalnih igara. Kao što je već rečeno, u današnjem svijetu prevladavaju 3D igre, no nije sporna pojava novih 2D igara. Računalne igre izrađuju se u različitim razvojnim okruženjima za izradu računalnih igara poput OpenGL-a, Unitya, Unreal Engine, Godota u određenim programskim jezicima.

U ovom ću završnom radu izraditi temeljne funkcionalnosti igre 2D platformera poput korisničkog sučelja, prelaska između scena, zvuka te kontroli kretanja i pucanja. Također ću prikazati različite tipove prepreka, neprijatelja, efekata i sl. Izrada temeljnih funkcionalnosti igre 2D platformera bit će izrađena u programskom alatu Unity u programskom jeziku C# (.NET).

2. Povijest računalnih igara

Povijest računalnih igara javlja se sredinom dvadesetog stoljeća i traje sve do dana današnjeg. U povijesti računalnih igara poznato nam je 7 razvojnih faza: „rana povijest“, „novo industrijsko razdoblje“, „zlatno doba“, „1980te“, „1990te“, „2000te“ i „2010te“. U nastavku slijedi kratki opis i temeljni predstavnici svake faze.

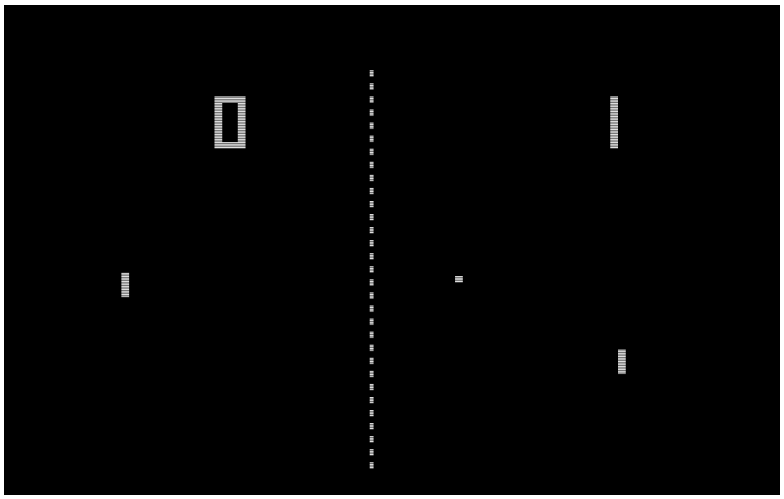
2.1. Rana povijest (1948 – 1972)

Sredinom dvadesetog stoljeća, u svijetu se počinje javljati ideja računalnih igara. U M.I.T.-u, profesori i studenti igrali su igrice poput Tic-Tac-Toe na računalima tipa IBM 1560. 1962. godine, Steve Russell u kolaboraciji s Graetzom, Wiitanenom, Saundersom i Pinerom izradio je prvu javno dostupnu igricu „Spacewar“. „Spacewar“ uskoro postaje popularna igrica te se javlja ideja o računalnim igrama diljem svijeta. [1]

Slika 1: Spacewar

2.2. Novo industrijsko razdoblje (1972 – 1978)

1972. godine, izrađena je prva uspješna širokodostupna igra, „Pong“ od strane videoindustrije „Atari“. Računalna igra „Pong“ je 2D računalna igra koja se temeljila na simulaciji stolnog tenisa. U razdoblju nove industrije došlo je do zasićenja tržišta igrama koje su bile izrađivane na isti princip što je rezultiralo padom interesa te u konačnici padom prodaje. Upravo iz tog razloga kompanija „Atari“ okreće se razvoju FPS igara. Jedna od takvih FPS igara je „Maze War“. [1]



Slika 2: Pong

2.3. Zlatno razdoblje (1978 – 1982)

Naziv „zlatno razdoblje“ dolazi isključivo zbog uspješne eksplozije arkadnih video igara. Veoma poznata arkadna igra iz tog razdoblja je „Space Invaders“. 1980. godine, javlja se i prva 3D igra, „Battlezone“ od strane industrije videoigara „Atari“. 1980. tih godina, dolazi era „igara u boji“. 1980. japanska tvrtka „Namco“ izrađuje jednu od veoma popularnih arkadnih igara, koja se igra još i dan danas, „Pacman“. Nadalje, u zlatnom razdoblju javlja se i igra „Donkey Kong“ od strane tvrtke „Nintendo“. [1]

Slika 3: Space Invaders

2.4. 1980te (1982 – 1990)

1983. godine od strane „Nintenda“ javlja se igra „Mario Bros“, koja se također igra još i dan danas. 1984. godine od strane A. Pajitnova javlja se igra „Tetris“. „Tetris“ je igra u kojoj se u današnje vrijeme organiziraju različita natjecanja u kojima se mogu osvojiti velike svote novaca. U razdoblju 1980tih, nastaju igre poput „Super Mario Bros“, „The Legend of Zelda“, „Castlevania“ itd… Možemo reći da je razdoblje između 1980tih do 1990tih godina bilo ključno za razvoj računalnih igara. [2]

Slika 4: Super Mario Bros

2.5. 1990te (1991 – 2000)

1990te godine često se u industriji video igara naziva glavnim inovacijskim desetljećem. U tom razdoblju prelazi se na 3D igre poput igara pucanja u prvom licu, strategija i MMO igara. 1991. godine nastaje izrazito popularna igrica, „Sonic the Hedgedog“, od strane tvrtke „Sonic Team“. 1990tih javljaju se igre poput „Wolfenstein 3D“, „Doom“, „Tekken“, „Super Mario 64“. „Resident Evil“, „Quake“, „Grand Turismo“, „Grand Theft Auto“, „Starcraft“ i sl. [1]

Slika 5: Starcraft

2.6. 2000te (2001 – 2010)

Razdoblje 2000tih godina je razdoblje koje karakterizira kreiranje inovativnih modova za igre. Također, u ovom razdoblju pojavila se i Sony-eva gaming konzola „PlayStation 2“ koja je uvelike doprinijela industriji video igara. Na prijelazu iz 20. u 21. stoljeće, nastaju svima dobro poznate popularne igre poput „Unreal Tournament“, „Counter Strike“, „The Sims“, „Call Of Duty“, „World of Warcraft“ (kolijevka MMORPG igara). [2]

Slika 6: World Of Warcraft Classic

Upravo zahvaljujući razdoblju 2000tih godina, današnja gaming industrija preslikava mnoge mehanike iz tog razdoblja u sadašnje razdoblje.

2.7. 2010te (2011 – 2020) - Računalne igre danas

U 21. stoljeću, računalne igre svakodnevno sve više i više globaliziraju svijet. Milijarde i milijarde ljudi svakodnevno provode slobodno vrijeme igrajući računalne igre. Naravno, postoji mogućnost zarade od računalnih igara putem izrade i prodaje računalnih igara ili streamanja. U današnje vrijeme igraju se igre koje su nastale 1980tih godina (npr. „Pacman“, razne inačice Super Maria), igre nastale 1990tih godina (razne inačice Dooma, Tekkena, Grand Turisma, Grand Theft Autoa i sl). Također, u današnje vrijeme igraju se i igre nastale u 21vom stoljeću: razne inačice The Simsa, World of Warcrafta, Counter Strikea i sl. Budućnost računalnih igara nije nimalo upitna, već je u potpunosti jasno da će se i dalje širiti i globalizirati svijet.

3. Programski alat Unity

Unity je programski alat za razvoj računalnih igara (engl. *game engine*) koji je razvijen od strane Unity Technologies-a, a predstavljen na Apple-ovoj konferenciji 2005. godine. Programski alat Unity nastao je 2005. godine s ciljem da ujedini razvoj računalnih igara i omogući izradu aplikacije za sve vrste platformi (stolna računala, laptopi, mobilne, konzole i sl). [3]

Slika 7: Logo programskog alata Unity

Programski alat Unity je *cross-platform* alat, što znači da ga je moguće koristiti na raznim platformama: Windows, MacOS, UNIX sistemi. Unity omogućuje izradu izvršne datoteke za različite platforme: iOS, Android, Tizen, Windows, Linux, Mac, PlayStation, Playstation VR, Nintendo itd. U programskom alatu Unity moguće je izraditi 2D igre, ali i 3D igre. Primarni i najzastupljeniji jezik Unitya je C#. U prijašnjim verzijama Unitya, bio je moguć razvoj u platformama Boo i JavaScript („*UnityScript*“), ali su one izbačene iz razloga što je gotovo 99% korisnika koristilo .NET (C#).

Iz prethodno navedenih razloga, prije izrade igre u programskom alatu Unity, potrebno je poznavati programiranja u programskom jeziku C#. Programski alat Unity pruža nam intuitivni editor koji nam olakšava izradu igre u usporedbi s drugim programskim alatima za razvoj računalnih igara poput OpenGLa. Postoji više načina odabira koju razinu programskog alata Unity možemo koristiti („Free“, „Plus“, „Pro“, „Enterprise“), a razlika je u cijeni i mogućnostima koje možemo koristiti.

Neke od popularnih igara izrađenih u Unityu: „Temple Run“, „Angry Birds“, „Plague Inc“, „Slender“, „Surgeon Simulator“, „Crossy Road“, „Pokemon Go“.

Slika 8: Temple Run

Slika 9: Plague inc

4. Izrada računalne igre

U nastavku će biti prikazan temelj ovog završnog rada vezan isključivo za razvoj računalne igre 2D Platformera. U početku ću opisati dvije igre koje su bile izvor moje motivacije za izradu ovog završnog rada, a potom ću opisati implementaciju igre u programskom alatu Unity.

4.1. Motivacija 1: igra „Super Mario Bros“ i igra „Meatboy“

Motivacija za izradu završnog rada 2D platformera bile su igre „Super Mario Bros“ i „Meatboy“. Kad sam bio dječak obožavao sam igrati „Super Mario Bros“ igru te prisustvovati određenim lokalnim natjecanjima u navedenoj igri. Igra „Meatboy“ bila mi je također jedna od omiljenih zbog njezinih fora vizualnih i zvučnih efekata.

4.1.1. Sličnosti s igrom „Super Mario Bros“

U ovom završnom radu uočit će se određene sličnosti s igrom „Super Mario Bros“. Prvenstvena stvar je mehanika kretanja. U obje igre možemo se kretati lijevo i desno te skakati. U igri ovog završnog rada imamo dodatnu mogućnost dvostrukog skoka. Nadalje, Super Mario može ispucavati određene projektile kako bi ubijao neprijatelje. Također se u igri u ovom završnom radu mogu ispucavati projektili kako bi se ubijali neprijatelji. Super Mario ima mogućnost ubijanja neprijatelja pomoću skoka na njegovu glavu. Također se u igri u ovom završnom radu može ubiti neprijatelj skokom na njegovu glavu. Ono što je nedostatak igre „Super Mario Bros“ je manjak vizualnih efekata koji u ovom završnom radu dominiraju.

4.1.2. Sličnosti s igrom „Meatboy“

Kao što je već rečeno, jedna od igara koje su me potaknule na izradu 2D platformera je i igra „Meatboy“. Igra „Meatboy“ očarala me svojim jednostavnim, no opet veoma uočljivim vizualnim efektima. Zbog toga je u ovom završnom radu prikazano nekoliko tipova vizualnih efekata (animacije kretanja, efekt stvaranja, efekt smrti, efekt sudara projektila i sl.). U igri „Meatboy“ postoji *timer* koji se povećava te je cilj završiti određenu razinu u što kraćem vremenu. U ovom završnom radu također postoji *timer*, no on se smanjuje te je igru potrebno završiti u što kraćem vremenu.

4.2. Tematika

Tematika igre temelji se na izlasku glavnog lika iz podzemlja (tame) na zemljinu površinu (svijetlost) zbog ponestajanja kisika. Glavni lik, tražeći blago, imao je nesretni slučaj urušavanja rudnika te igrač mora pomoći glavnom liku pronaći put natrag do zemljine površine (svijetlosti) kako se ne bi ugušio. Igrač ima određen broj sekundi da iziđe na zemljinu površinu. Ukoliko navedeno vrijeme istekne, igrač se onesviješta i umire (game-over). Igrača na putu do izlaska iz podzemlja očekuje nekolicina krivih puteva, velik broj neprijatelja iz podzemlja, prepreka i zagonetka.

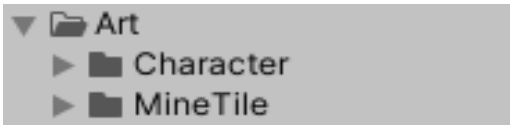
4.3. Paketi (engl. packages)

Programski alat Unity sadrži svoj vlastiti upravitelj paketima koji se naziva „Unity Package Manager“. Unity Package Manager omogućava instalaciju raznih biblioteka, ubacivanje različitih paketa, tekstura ili animacija te ubacivanje određenih predložaka projekata (engl. template projects) u naš vlastiti projekt.

4.3.1. Primjer paketa (engl. packages) u igri

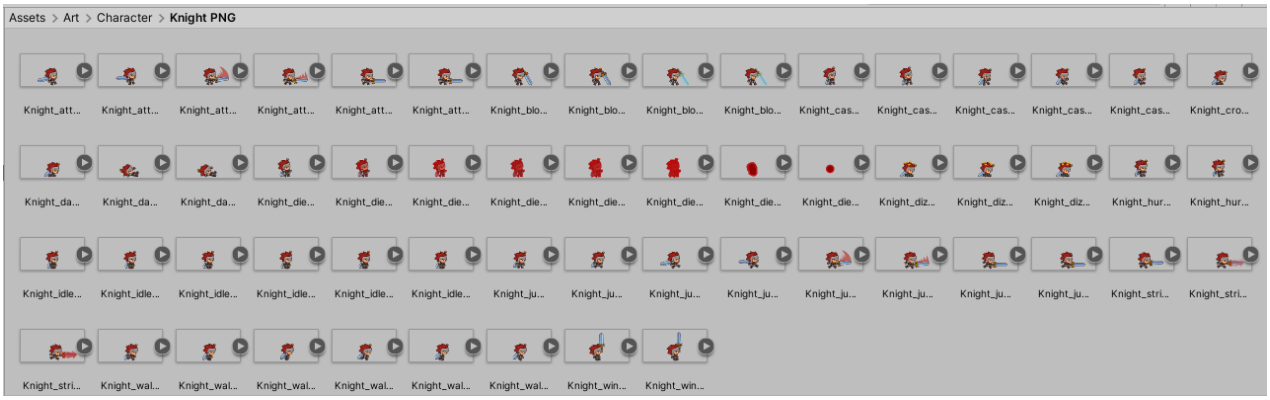
Primjer korištenja Unity Package Managera u ovom završnom radu jest npr. ubacivanje prethodno napravljenih tekstura za pozadinu, prethodno napravljenih tekstura za ambijent (engl. environment) te prethodno napravljenih tekstura za igrača i protivnike. Prethodno napravljene teksture korištene su zbog fokusa završnog rada na funkcionalnosti i algoritme u igri, a ne na dizajn. Ubacivanjem prethodno napravljenih tekstura pomoću Unity Package Managera skraćujemo vrijeme izrade dizajna igre te samim time dobivamo više vremena za izradu funkcionalnosti igre.

U projektu se u folderu Art nalazi više ubačenih paketa, npr: „Character“ – sadrži stvari za prikaz charactera i „MineTile“ – sadrži stvari za prikaz pozadine i ambijenta u igri.



Slika 10: Prikaz nekih paketa u igri

U paketu „Character“, možemo vidjeti različite sličice koje nam prikazuju karakter, odnosno glavni lik igre. Svaku od tih sličica možemo iskoristiti u našoj igri kako bi se igraču pružila što veće zadovoljstvo i uživanje u igranju igre.



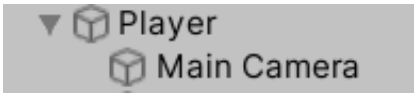
Slika 11: Prikaz sličica paketa Character

4.4. Pozadina (engl. Background)

Pozadina, odnosno ambijent u igri, izrazito je bitan element u formiranju bilo kojeg žanra igre. Pozadina uveliko pridonosi kontekstu i boljem razumijevanju igre. Pozadina se u Unity-u može formirati preko elemenata određenih paketa na način da se rasporede po onim djelovima mape koje će kamera tijekom igre obuhvaćati. U primjeru ovog završnog rada, koristili smo element iz paketa „MineTile“ te smo ga rasporedili po cijeloj mapi koju će prilikom igre obuhvaćati kamera.

4.5. Kamera (engl. Camera)

Objekt kamere je objekt koji je dostupan u projektu nakon što se kreira projekt, bez potrebe posebnog ubacivanja istog. Kamera nam omogućuje postavljanje vidnog polja u igri. Svaka kamera ima svoju poziciju u igri te se može namjestiti tako da prikazuje željene kadrove. U našem primjeru 2D platformera, kamera mora pratiti igrača. Način na koji možemo realizirati da kamera prati igrača jest postavljanje kamere kao dijete objekta glavnog lika.



Slika 12: Postavljanje prateće kamere na igrača

Prethodno navedeni način implementacije je najjednostavniji način implementacije prateće kamere. U ovom završnom radu prateću kameru smo implementirali pomoću Lerp tj.

Linear Interpolation funkcije. Pomoću Lerp funkcija dobivamo glatkoću prilikom kretanja kamere. Objektu kamere pridružili smo skriptu *„CameraFollow.cs“*:

```
1 public class CameraFollow : MonoBehaviour
2 {
3     public Transform target;
4     public float smoothSpeed;
5     public Vector3 offset;
6
7     void FixedUpdate()
8     {
9         Vector3 currentPosition = transform.position;
10        Vector3 desiredPosition = target.position + offset;
11        Vector3 smoothedPosition = Vector3.Lerp(currentPosition,
12             desiredPosition, smoothSpeed*Time.deltaTime);
13        transform.position = smoothedPosition;
14    }
15 }
```

Za implementaciju prateće kamere pomoću Lerp funkcije koristili smo tri svojstva: *target* tipa Transform – objekt koji će kamera pratiti, *smoothSpeed* tipa float – brzina kojom će se kamera kretati, *offset* tipa Vector3 koji definira pomak kamere u odnosu na objekt koji se prati. Potrebna nam je Unity metoda *FixedUpdate()* koja je prikazana u linijama 7-14. U liniji 9 bilježimo trenutnu poziciju kamere. U liniji 10 bilježimo željenu poziciju kamere kojoj pridodajemo *offset*. U liniji 11 i 12 kreiramo Lerp funkciju koja se kreće od početne pozicije prema konačnoj poziciji u vremenu *smoothSpeed*Time.deltaTime*. Na poslijedku se pozicija kamere u liniji 13 postavlja na prethodno definiranu varijablu *smoothedPosition* čime se prikazuje efekt glatkog kretanja kamere.

4.6. Kontrole (engl. Controls)

Za svaku izrađenu igru, izrazito je bitan segment kontrola. Jedan od segmenta kontrola je kretanje (*engl. movement*). Kretanje određuje način na koji će se likovi u igri pomicati. U 2D igrama, moguće je kretanje po X i Y osima, dok je u 3D igrama moguće kretanje po X, Y i Z osima. Izrada kretanja u 2D igrama je jednostavnija zbog toga što je potrebno usklađivanje samo 2 osi, za razliku od 3D igre gdje je potrebno usklađivanje 3 osi. U 2D igri možemo se kretati po x osi ulijevo (-x) ili udesno (+x). Nadalje, moguće je kretanje po y osi prema gore (+y) ili prema dolje (-y). Nadalje, važna kontrola 2D i 3D akcijskih igara jesu mogućnosti pucanja i udaranja. Pomoću kontrole pucanja možemo primjerice ubiti neprijatelja te lakše savladati određene prepreke u igri, a samim time i stići do cilja.

4.6.1. Primjer kretanja (engl. Movement) u igri

U ovom završnom radu radi se o 2D igri, zbog čega je moguće kretanje po X i Y osi. Glavni lik može se kretati lijevo i desno te ima mogućnost jednostrukog i dvostrukog skoka. Implementacija kretanja nalazi se u klasi *„PlayerMovement“* koja je pridružena objektu igrača. Objektu igrača je potrebno također dodati Unity komponente „RigidBody2D“ te „2DCircleCollider“. Za kretanje igrača potrebno je definirati više svojstva različitih tipova podataka koje su definirane u klasi *„PlayerController“*:

```
1 public class PlayerController : MonoBehaviour
2 {
3    // Properties - Movement
4    public float moveSpeed;
5    public float jumpHeight;
6    // Properties - Jump
7    public Transform groundCheck;
8    public float groundCheckRadius;
9    public LayerMask whatIsGround;
10   private bool grounded;
11   private bool doubleJumped;
12
13   // U nastavku se nalaze metode…
14 }
```

Varijabla *moveSpeed* određuje nam brzinu kojom će se kretati igrač u igri, dok nam varijabla *jumpHeight* određuje visinu skoka igrača. Varijable definirane u liniji 7-11 potrebne su za skok. Varijabla *grounded* određuje nam ukoliko je igrač „prizemljen“, dok nam varijabla *doubleJumped* određuje ukoliko je igrač iskoristio mogućnost dvostrukog skoka. U nastavku su prikazane implementacije metoda za kretanje igrača:

```
1 void FixedUpdate()
2 {
3     grounded = Physics2D.OverlapCircle(groundCheck.position,
4                 groundCheckRadius, whatIsGround);
5 }
6
7 void Update()
8 {
9     //Jump
10    if (Input.GetKeyDown(KeyCode.Space))
11    {
12        if (grounded)
13        {
14            Jump();
15            doubleJumped = false;
16        }
17        else
18        {
```

```
19              if (!doubleJumped)
20              {
21                  Jump();
22                  doubleJumped = true;
23              }
24          }
25      }
26
27      // Go right
28      if (Input.GetKey(KeyCode.D))
29      {
30          GetComponent<Rigidbody2D>().velocity = new Vector2(moveSpeed,
31              GetComponent<Rigidbody2D>().velocity.y);
32          transform.localScale = new Vector3(1f, 1f, 1f);
33      }
34
35      // Go left
36      if (Input.GetKey(KeyCode.A))
37      {
38          GetComponent<Rigidbody2D>().velocity = new Vector2
39              (-moveSpeed, GetComponent<Rigidbody2D>().velocity.y);
40          transform.localScale = new Vector3(-1f, 1f, 1f);
41      }
42  }
```

Metode koje su potrebne za kretanje igrača su predefinirane metode u Unityu. Jedna od njih je metoda *FixedUpdate()* koja je izrazito važna za fiziku igre (engl. *Physics2D*). Navedena metoda pokreće se jedan ili više puta po sličici (engl. *frame*-u). U predefiniranoj metodi *FixedUpdate()* koristimo metodu *OverlapCircle()* u kojoj provjeravamo ukoliko je korisnik „prizemljen“ te dobivenu vrijednost pridružujemo varijabli *grounded*. U predefiniranoj metodi *Update()* koja se pokreće jednom u jednom *frameu*, definirano je kretanje igrača udesno ukoliko se pritisne tipka D, odnosno ulijevo ukoliko se pritisne tipka A. U linijama 10-25 definirana je mogućnost skoka (12-16) ukoliko je igrač prizemljen, odnosno dvostrukog skoka (17-24) ukoliko igrač nije prizemljen te prethodno nije iskoristio dvostruki skok.

```
1  private void Jump()
2  {
3      GetComponent<Rigidbody2D>().velocity = new Vector2
4          (GetComponent<Rigidbody2D>().velocity.x, jumpHeight);
5  }
```

Metoda *Jump()* pristupa Unity komponenti „RigidBody2D“, odnosno njezinom svojstvu *velocity* te joj pridružuje objekt tipa Vector2 s trenutnom x pozicijom na x-osi te *jumpHeight* vrijednošću na y-osi.

Igrač također ima mogućnost kretanja po ljestvama. Kretanje po ljestvama omogućeno je pritiskom na tipku W kada se nalazimo u okolini ljestvi.

Slika 13: Ljestve

Najvažnije stvari vezane za ljestve definirane su u klasi *„LadderZone“,* dok se u klasi *„PlayerController“* nalaze dodatne stvari vezane za kretanje igrača po ljestvama. U nastavku slijedi dio prikaza implementacije metode *Update()* u klasi *„PlayerController“*.

```
1  void Update()
2  {
3      if (onLadder)
4      {
5          GetComponent<Rigidbody2D>().gravityScale = 0f;
6          if (Input.GetKey(KeyCode.W))
7          {
8              GetComponent<Rigidbody2D>().velocity = new Vector2(0,
9                  climbSpeed);
10         }
11         else if (Input.GetKey(KeyCode.S))
12         {
13             GetComponent<Rigidbody2D>().velocity = new Vector2(0,
14                 -climbSpeed);
15         }
16         else
17         {
18             GetComponent<Rigidbody2D>().velocity = new Vector2(0, 0);
19         }
20      }
21
```

```
22      if (!onLadder)
23      {
24          GetComponent<Rigidbody2D>().gravityScale = gravityStore;
25      }
26  }
```

Blok naredni u linijama 3-20 izvršava se ukoliko je svojstvo igrača *onLadder* istinito. Gravitacija igrača postavlja se na iznos 0. U slučaju pritiska tipke W, igrač se penje po ljestva. Za spuštanje po ljestvama koristi se tipka S. Naredba iz linije 24 izvršava se ukoliko igrač nije u okolini ljestava. Tada se njegova gravitacija postavlja na početnu vrijednost.

U prikazu koda ispod prikazana je implementacija potrebna za penjanje po ljestvama u klasi *„LadderZone“.* Ljestvama je potrebno pridružiti Unity svojstvo „BoxCollider2D“ zbog metoda *OnTriggerEnter2D()* i *OnTriggerExit2D()*. U metodi *OnTriggerEnter2D()* u linijama 10-16 provjeravamo ukoliko je objekt s imenom „Player“ u dodiru s ljestvama. Ukoliko jest, tada se svojstvo igrača *onLadder* postavlja na vrijednost true. U metodi *OnTriggerExit2D()* u linijama 18-24 provjeravamo ukoliko objekt s imenom „Player“ nije u dodiru s ljestvama. Tada se svojstvo igrača *onLadder* postavlja na vrijednost false.

```
1  public class LadderZone : MonoBehaviour
2  {
3      private PlayerController player;
4
5      void Start()
6      {
7          player = FindObjectOfType<PlayerController>();
8      }
9
10      void OnTriggerEnter2D(Collider2D other)
11      {
12          if (other.name == "Player")
13          {
14              player.onLadder = true;
15          }
16      }
17
18      void OnTriggerExit2D(Collider2D other)
19      {
20          if (other.name == "Player")
21          {
22              player.onLadder = false;
23          }
24      }
25  }
```

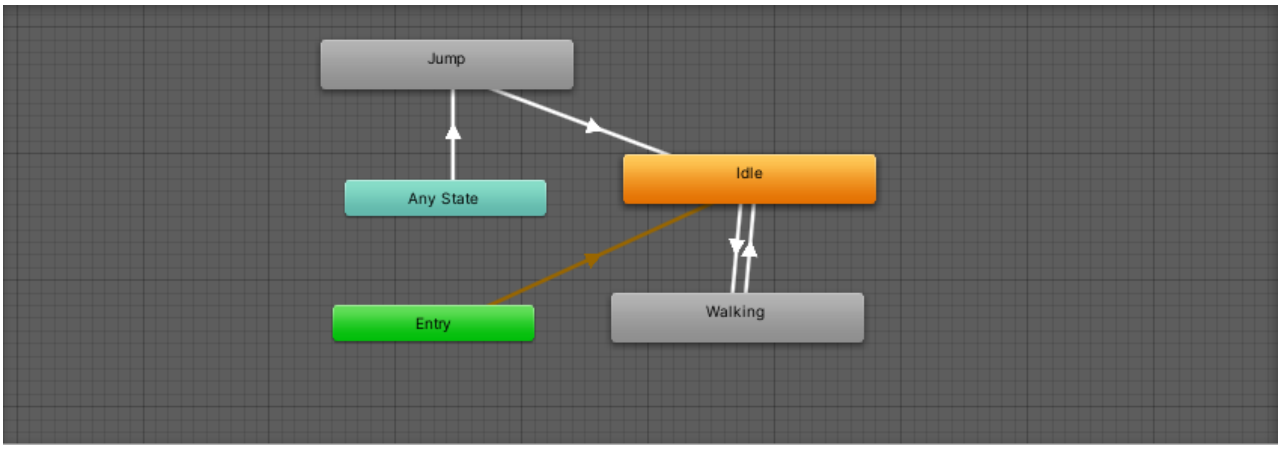
4.6.2. Animacija kretanja u igri

Prethodno opisana funkcionalnost omogućava kretanje igrača, ali uz nedostatak dinamičke slike. Upravo iz navedenog razloga potrebno je izraditi animacije. Za izradu animacije u Unity alatu za izradu igara potrebni su nam dodatni prozori „Animator“ i „Animation“. Također, potrebna nam je sitna preinaka u prethodno navedenom programskom kodu.

U novo stvorenom prozoru „Animation“ kreirali smo tri tipa animacija: *„Idle“* – kada igrač stoji na mjestu, *„Walking“* – kada se igrač kreće u određenom smjeru, *„Jump“* – kada je igrač pritisnuo tipku za skok. U prozoru „Animation“, potrebno je više sličica postaviti u određenom redoslijedu te odrediti vremenski interval između istih kako bi se postigao efekt „animacije“. Što je više sličica u kraćem vremenskom periodu, to je animacija više fluidna i „čišća“.

Slika 14: Sličice korištene za animaciju "Walking"

U novo stvorenom prozoru „Animator“ uočili novo kreirana stanja te dodali logiku kojom će se dešavati prijelazi stanja.



Slika 15: Prozor "Animator" za animacije kretanja igrača u igri

Možemo uočiti da je početno stanje navedeno stanje *„Idle“*. Igrač prelazi u stanje *„Walking“* kada pritisne tipku A ili tipku D na tipkovnici, odnosno kada je vektor brzine po osi x > 0. Igrač se iz stanja *„Walking“* vraća u stanje *„Idle“* kada prestane držati tipku A ili tipku D, odnosno kada je vektor brzine po osi x = 0. Igrač također u bilo koje vrijeme može preći u stanje *„Jump“* pritiskom na tipku „Space“. Igrač se iz stanja *„Jump“* vraća u stanje *„Idle“* kada dotakne objekt s slojem „Ground“.

4.6.3. Primjer kontrole pucanja u igri

U ovom završnom radu, igrač ima mogućnost napada stvaranjem određenih projektila pritiskom na tipku 1 na Num Pad djelu tipkovnice. Kada igrač pritisne tipku 1, ispaljuje se projektil koji se kreće u smjeru u kojem je igrač bio okrenut u trenutku pritiska tipke. Svaki projektil prilikom kolizije s drugim objektom uzrokuje nastanak eksplozije. Ukoliko je objekt koji je u koliziji s projektilom neprijatelj, isti se uništava.

Slika 16: Primjer ispaljivanja projektila u igri

Implementacija projektila je veoma jednostavna. Potrebno je pronaći / napraviti animaciju projektila koji će se ispaljivati kada korisnik pritisne određenu tipku te napisati nekoliko linija koda prikazanih u nastavku:

```
1 if (Input.GetKey(KeyCode.Keypad1) && canShoot)
2 {
3     canShoot = false;
4     Instantiate(projectile, firePoint.position,
5         firePoint.rotation);
6 }
```

U linijama 1-6 prikazan je način ispaljivanja projektila u igri. Korisnik mora pritisnuti tipku 1 na Num Pad djelu tipkovnice te igraču mora biti dopušteno ispuštanje projektila (varijabla *canShoot* treba biti postavljena u true). Ukoliko su prethodni uvjeti zadovoljeni, projektil se ispaljuje te se varijabla *canShoot* postavlja u vrijednost false. U nastavku slijedi opis metoda iz klase *„FireBallController“:*

```
1  void Start()
2  {
3      player = FindObjectOfType<PlayerController>();
4
5      if (player.transform.localScale.x < 0)
6      {
7          moveSpeed = -moveSpeed;
8          gameObject.transform.rotation = new Quaternion(0f,0f,180f,0f);
9      }
10 }
11
12 void Update()
13 {
14     GetComponent<Rigidbody2D>().velocity = new Vector2(moveSpeed, 16
15     GetComponent<Rigidbody2D>().velocity.y);
16 }
17
18 void OnTriggerEnter2D(Collider2D other)
19 {
20     if (other.tag == "Enemy")
21     {
22         other.GetComponent<EnemyHealthController>()
23             .giveDamage(damageToGive);
24     }
25
26     Instantiate(projectileExplosionEffect, other.transform.position,
27         other.transform.rotation);
28     Destroy(gameObject);
29 }
```

Kao što sama riječ govori, odnosi se na implementaciju ispaljivanja projektila u igri. U metodi *Start()* u liniji 1-10 pronalazimo objekt igrača, a potom provjeravamo ukoliko je igrač okrenut u lijevu stranu (linija 5) te u skladu s rotacijom igrača stvaramo i ispaljujemo projektil u određenom smjeru prema definiranoj brzini čija je vrijednost spremljena u varijabli *moveSpeed*. Metoda *Update()* u linijama 12-16 osigurava kretanje projektila u određenom smjeru. Metoda *OnTriggerEnter2D()* koja se prožima kroz linije 18-29 okida se kada projektil dotakne neki drugi objekt. Potom se provjerava ukoliko je drugi objekt tipa „Enemy“, te ukoliko jest, umanjuje preostalu snagu protivnika. U linijama 26-28 prikazuje se efekt eksplozije te uništavanja projektila.

4.6.4.Primjer kontrole skoka na neprijateljevu glavu u igri

U ovom završnom radu također je implementirana mogućnost uništavanja neprijatelja pomoću skoka na njegovu glavu. Navedena tehnika inspirirana je igrama poput Super Maria. Implementacija kontrole skoka na neprijateljevu glavu implementirana je u klasi

„HurtEnemyOnContact“. U klasi postoje 3 svojstva: *damageToGive* – koji određuje koliku štetu nanosimo neprijatelju prilikom skoka na njegovu glavu, *bounceOnEnemy* – koji definira koliki je odskok nakon skoka na neprijateljevu glavu i *rigidBody2D* koji se u metodi *Start()* inicijalizira na igrača.

```
1  public class HurtEnemyOnContact : MonoBehaviour
2  {
3      public int damageToGive;
4      public float bounceOnEnemy;
5      private Rigidbody2D rigidBody2D;
6      AudioSource audioSource;
7      public AudioClip audioClip;
8
9      void Start()
10     {
11         audioSource = GetComponent<AudioSource>();
12         rigidBody2D = transform.parent.GetComponent<Rigidbody2D>();
13     }
14
15     void OnTriggerEnter2D(Collider2D other)
16     {
17         if (other.tag == "Enemy")
18         {
19             other.GetComponent<EnemyHealthController>().
                   giveDamage(damageToGive);
20             rigidBody2D.velocity = new Vector2(rigidBody2D.velocity.x,
21                 bounceOnEnemy);
22             audioSource.PlayOneShot(audioClip);
23        }
24     }
25 }
```

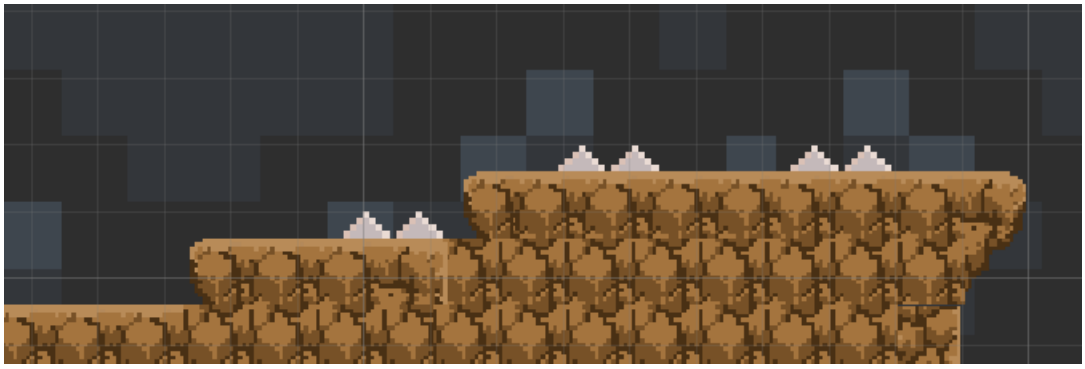
U linijama 15-24 nalazi se metoda *OnTriggerEnter2D()* koja nam omogućuje provjeru ukoliko se radi o koliziji između igračevih nogu i neprijatelja. Ukoliko jest, tada se neprijatelju nanosi šteta količine *damageToGive*, dok igrač odskače u zrak u vrijednosti svojstva *bounceOnEnemy*. Također, prilikom skoka na neprijateljevu glavu pokreće se zvuk *audioClip*.

4.7. Prepreke

Neopisivo veliku vrijednost i uzbuđenje za razno-razne igre pruža koncept prepreka. Prepreke su odličan način za prikaz maštovitosti kreatora igre te su jedan od ključnih elemenata koji igru čine posebnom. Postoje različite vrste prepreka: šiljci odozgo, šiljci odozdo, kretajuće platforme i sl.

4.7.1. Šiljci odozdo (engl. Spikes) u igri

Kao što je već prethodno spomenuto, jedna od mogućnih prepreka u igri su šiljci odozdo. U ovom završnom radu, navedena prepreka služi za ubijanje igrača te gubitak jednog života i ponovno stvaranje na posljednje zauzetoj kontrolnoj točki.



Slika 17: Primjer šiljaka odozdo u igri

Svaki objekt šiljka odozdo ima pridruženu Unity komponentu „BoxCollider2D“ koji je postavljen kao okidač (engl. trigger). Nadalje, svakom objektu šiljka pridružena je skripta *„KillPlayer.cs“* čija će funkcionalnost biti opisana u nastavku:

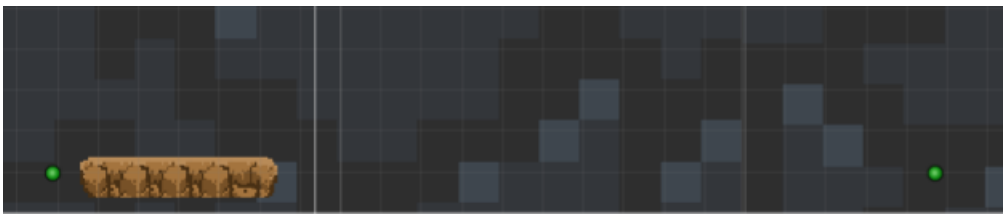
```
1 void OnTriggerEnter2D(Collider2D other)
2 {
3     if (other.name == "Player" && LevelManager.isDead==false)
4     {
5         LevelManager.isDead = true;
6         levelManager.RespawnPlayer();
7         lifeManager.TakeLife();
8     }
9 }
```

Kao što je već spomenuto, svaki šiljak odozdo ima pridruženu skriptu *„KillPlayer.cs“.* U navedenoj skripti postoji događaj (engl. event) *OnTriggerEnter2D* koji se okida kada se dva objekta međusobno dodiruju. U liniji 3, provjerava se ukoliko je objekt igrača u koliziji s objektom šiljaka odozdo. Ukoliko je prethodno navedena tvrdnja istinita, poziva se funkcija *RespawnPlayer()* iz klase *„LevelManager“* pomoću instanciranog objekta *levelManager*. Funkcija *RespawnPlayer()* stvara igrača ponovno na posljednjoj kontrolnoj točki koju je dodirnuo (zauzeo):

```
1 void RespawnPlayer()
2 {
3    player.transform.position = currentCheckpoint.transform.position;
4 }
```

4.7.2. Kretajuće platforme

Kretajuće platforme predstavljaju još jednu od prepreka koju igrač mora savladati kako bi uspješno završio igru. Kretajuće platforme kreću se od točke A prema točki B te predstavljaju poveznicu između dva djela trenutne razine. Igrač mora doći s jednog djela trenutne razine na drugi dio trenutne razine.



Slika 18: Prikaz kretajuće platforme u igri

Kretajuća platforma kreće se od lijeve zelene točkice prema desnoj zelenoj točkici prikazanoj na slici 18. – „Prikaz kretajuće platforme u igri“. Način implementacije kretajuće platforme nalazi se u klasi *„MovingPlatform“*:

```
1  public class MovingPlatform : MonoBehaviour
2  {
3      public GameObject platform;
4      public float moveSpeed;
5      public Transform currentPoint;
6      public Transform[] points;
7      public int pointSelection;
8
9      void Start()
10     {
11         currentPoint = points[pointSelection];
12     }
13
14     void Update()
15     {
16         platform.transform.position =
17             Vector3.MoveTowards(platform.transform.position,
18             currentPoint.position, Time.deltaTime * moveSpeed);
19
20         if(platform.transform.position == currentPoint.position)
21         {
22             pointSelection++;
23
```

```
24              if (pointSelection == points.Length)
25                  pointSelection = 0;
26
27              currentPoint = points[pointSelection];
28          }
29      }
30 }
```

Prilikom implementacije kretajuće platforme koristili smo 5 svojstva: *platform* tipa GameObject, *moveSpeed* tipa float, *currentPoint* tipa Transform, *points* tipa polje Transformova i *pointSelection* tipa int. Svojstvo *platform* označava platformu koja se kreće od točke A do točke B (polje Transformova – svojstvo *points*) određenom brzinom *moveSpeed*. Svojstvo *currentPoint* označava koja je trenutna točka prema kojoj se platforma kreće. U metodi *Start()* svojstvo *currentPoint* postavlja se na prvi Transform u polju Transformova *points[]* zbog toga što *pointSelection* pri inicijalizaciji iznosi 0. U metodi *Update()* u 16-18 upravljamo kretanjem platforme s jedne točke prema drugoj točki. Linije 20-28 izvršavaju se samo ako se pozicija platforme nalazi na ciljnoj poziciji. Upravo tada se ciljna pozicija postavlja na novu ciljnu poziciju te se kretajuća platforma počinje kretati prema njoj.

4.8. Neprijatelji

Svijet igara obiluje mnoštvom neprijateljskih čudovišta. Upravo iz tog razloga, u ovom završnom radu implementirani su različiti tipovi neprijatelja. Važno je napomenuti da svaki neprijatelj iznad svoje glave ima snagu tj. „health bar“. Neprijateljev „health bar“ smanjuje se sukladno s štetom tj. „damage-om“ koji mu napravi igrač.

4.8.1. Neprijatelj tip 1 – Hodajući dinosaur

Neprijatelj tip 1 – „Hodajući dinosaur“ najjednostavnija je vrsta neprijatelja za savladavanje u ovom završnom radu. Jedini način na koji hodajući dinosaur može igraču oduzeti život je ukoliko ga se dotakne. Hodajući dinosaur nema mogućnost pucanja, letenja, brzog trčanja ili sl., već se kreće konstantnom brzinom istim putanjama.

Slika 19: Animacija hodajućeg dinosaura

Slika 20: Prikaz neprijatelja "Hodajući dinosaur" u igri

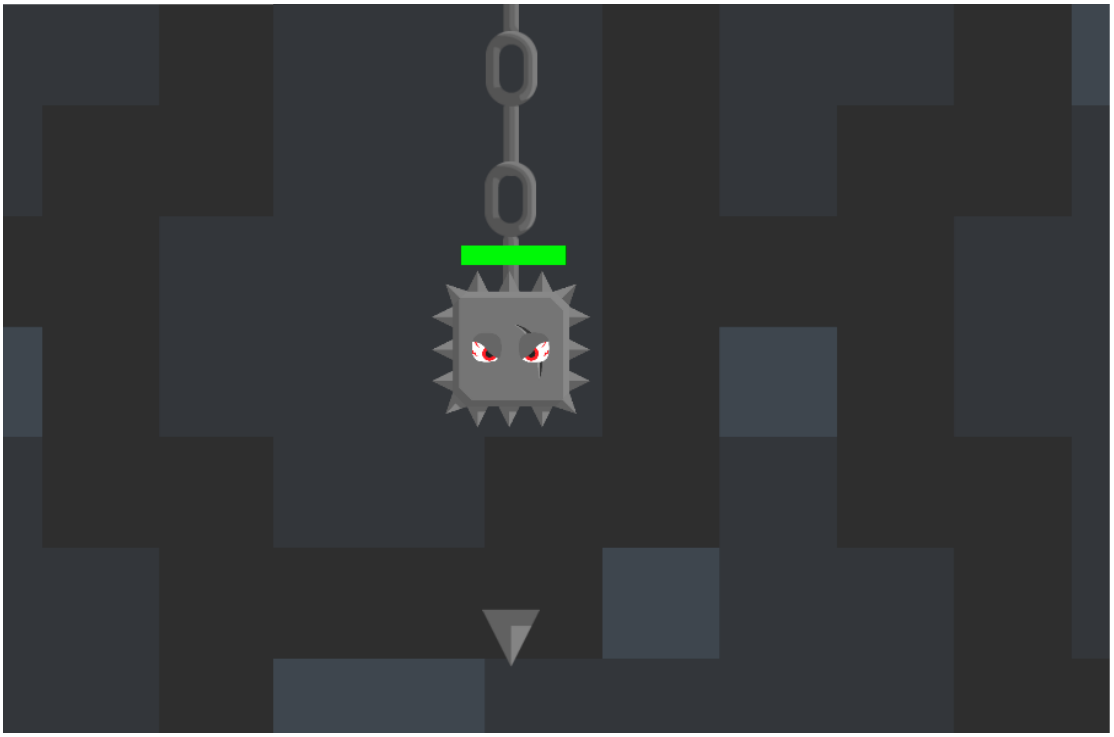
Igrač može neprijatelja tipa 1 – „Hodajući dinosaur“ jednostavno zaobići laganim preskokom ili dvostrukim skokom i pobjeći mu. Također, igrač može ubiti neprijatelja što će rezultirati nestajanjem hodajućeg dinosaura s mape. Neprijatelj se može ubiti pomoću vatrenih loptica ili pomoću skoka na neprijateljevu glavu.

```
1  void Start()
2  {
3      moveRight = false;
4  }
5
6
7  void Update()
8  {
9      atEdge = Physics2D.OverlapCircle(edgeCheck.position,
10     wallCheckRadius, whatIsWall);
11
12     if (!atEdge)
13         moveRight = !moveRight;
14
15     if (moveRight)
16     {
17         transform.localScale = new Vector3(1f, 1f, 1f);
18         GetComponent<Rigidbody2D>().velocity = new Vector2
19             (moveSpeed, GetComponent<Rigidbody2D>().velocity.y);
20     }
21     else
22     {
23         transform.localScale = new Vector3(-1f, 1f, 1f);
24         GetComponent<Rigidbody2D>().velocity = new Vector2
25             (-moveSpeed, GetComponent<Rigidbody2D>().velocity.y);
26      }
27 }
```

Implementacija hodajućeg dinosaura izrazito je lagana i trivijalna. U metodi *Start()* u koja je prikazana na liniji 2-4 postoji samo jedna linija koda koja mijenja vrijednost varijable *moveRight* u false. Navedena promjena vrijednosti varijable *moveRight* govori nam da će se neprijatelj u početku kretati u lijevu stranu. Ukoliko bi se neprijatelj neprestano kretao u lijevu stranu naišli bismo na problem neprestanog sudaranja o zid. Upravo iz prethodno navedenog razloga, u linijama 8-27 omogućavamo kretanje neprijatelja lijevo-desno tj. promjenu smjera kretanja. Metoda *Update()* izvršava se više puta u jednoj sekundi. Svakim izvršavanjem metode *Update()* provjeravamo ukoliko je neprijatelj došao do ruba platforme za hodanje. Ukoliko jest, tada se smjer kretanja mijenja iz lijeva u desno ili iz desna u lijevu (linija 13,14) te se sukladno tome podešavaju veličine neprijatelja, odnosno brzine neprijatelja (izvršavaju se linije 16-20 ili 22-26).

4.8.2.Neprijatelj tip 2 – Ovješeni bodljikavac

U igri postoji i drugi tip neprijatelja koji se zove „Ovješeni bodljikavac“. Naziv ovješeni dolazi od asocijacije lanca na koji je neprijatelj ovješen. Naziv bodljikavac dolazi od asocijacije oblika lika koji je pun bodlji. Ovješeni bodljikavac nema mogućnost kretanja kao neprijatelj tipa 1 – Hodajući dinosaur, no ima mogućnost ispuštanja određenih projektila u obliku bodlji koje se repetitivno izbacuju svakih X sekundi.



Slika 21: Izgled neprijatelja tipa 2 - Ovješeni bodljikavac

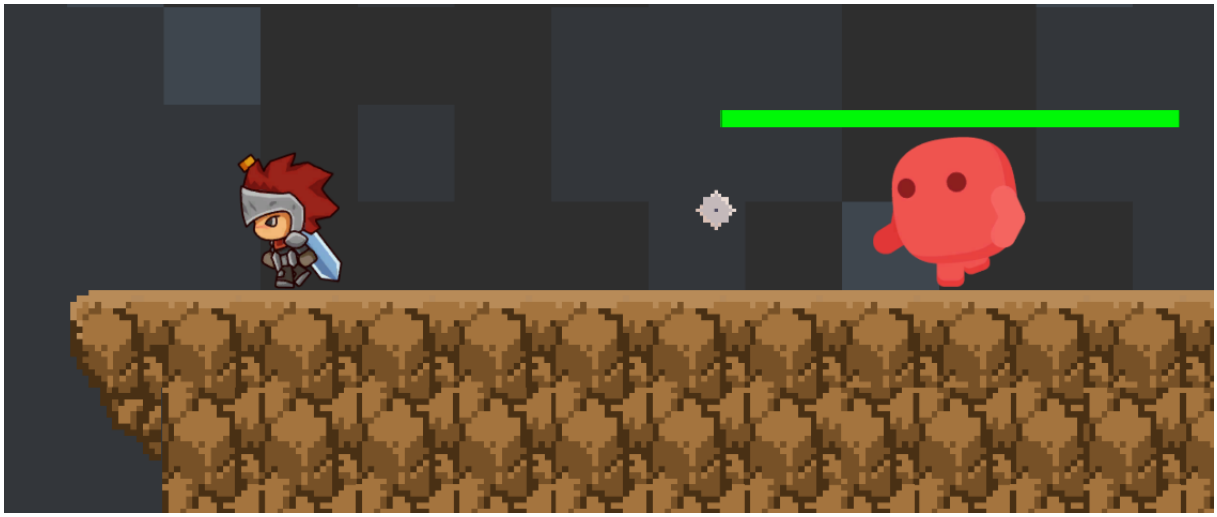
Implementacija ovješenog bodljikavca slična je implementaciji neprijatelja hodajućeg dinosaura uz malu preinaku. Navedena preinaka je izuzetak kretanja, uz dodatnu mogućnost ispaljivanja bodlje koja je definirana u klasi *„EnemyTwoInsantiateProjectile“:*

```
1  public class EnemyTwoInsantiateProjectile : MonoBehaviour
2  {
3      private float countdownTimer;
4      public float cooldownTime;
5      public GameObject projectile;
6
7      void Start()
8      {
9          countdownTimer = cooldownTime;
10     }
11
12     void Update()
13     {
14         countdownTimer -= Time.deltaTime;
15
16         if (countdownTimer < 0)
17         {
18             Instantiate(projectile,
19                 transform.position,transform.rotation);
20             countdownTimer = cooldownTime;
21         }
22     }
23 }
```

Za implementaciju ispaljivanja projektila kod neprijatelja korištena su tri svojstva: *countdownTimer* - koji nam bilježi koliko je vremena preostalo do ispaljivanja sljedećeg projektila, *cooldownTime* – koliko je vremena potrebno da se projektil ispusti, *projectile* – svojstvo tipa GameObject koje definira projektil koji se ispušta. U metodi *Start()* svojstvo *countdownTimer* postavlja se na vrijednost svojstva *cooldownTime*. U metodi *Update()* *cooldownTimer* se neprestano umanjuje. Kada je vrijednost svojstva *cooldownTimer* manja od 0, instancira se objekt projektila te se vrijednost svojstva *countdownTimer* postavlja na vrijednost svojstva *cooldownTime*.

4.8.3. Neprijatelj tip 3 – Crveni trkač

U igri postoji još jedan tip neprijatelja – „Crveni trkač“ čija glavna karakteristika je brzo trčanje te mogućnost ispaljivanja dva projektila svake dvije sekunde. Igrač umire ukoliko ga pogodi projektil koji je bačen od strane crvenog trkača ili se sudari s crvenim trkačem. Način implementacije trčanja jednak je implementaciji prikazanoj kod neprijatelja tipa 1 – „Hodajući dinosaur“, dok je način ispaljivanja projektila jednak implementaciji ispaljivanja projektila kod igrača.



Slika 22: Neprijatelj tip 3 - Crveni trkač

4.9. Stvari za sakupljanje (engl. Pickups)

U 2D igrama postoje različiti tipovi pozitivnih efekata koje se mogu pokupiti (npr. dodatni život, ubrzanje kretanja, povećanje skoka, jače oružje i sl). Stvari koje možemo pokupiti u igrama mogu biti i negativnih efekata (npr. izgubljeni život, usporavanje kretanja, smanjenje skoka, slabije oružje i sl). Sve prethodno navedene stvari dočaravaju svijet igre te ju čine zanimljivijom.

4.9.1. Dodatni život

U ovom završnom radu postoji mogućnost sakupljanja dodatnog života. Na određenim lokacijama na mapi nalazi se ikonica čovječuljka igrača koju igrač mora pokupiti kako bi dobio dodatni život. Implementacija dodatnog života u potpunosti je jednostavna te je implementirana u klasi *„LifePickup“*:

```
1  public class LifePickup : MonoBehaviour
2  {
3      public LifeManager lifeManager;
4      public AudioSource lifePickupSound;
5
6      void Start()
7      {
8          lifeManager = FindObjectOfType<LifeManager>();
9      }
10
11      void OnTriggerEnter2D(Collider2D other)
12      {
13          if (other.name == "Player")
```

```
14          {
15              lifeManager.GiveLife();
16              lifePickupSound.Play();
17              Destroy(gameObject);
18          }
19      }
20 }
```

Implementacija dodavanja dodatnog života implementirana je u klasi *„LifePickup“* koja je pridružena objektu (slici) koja označava dodatni život. Za ovakvu jednostavnu inačicu potrebna nam je jedna varijabla *lifeManager* tipa *LifeManager* koju u metodi *Start()* pronalazimo pomoću funkcije *FindObjectOfType<>().* Navedeni objekt ima osluškivač događaja *OnTriggerEnter2D()* koji provjerava ukoliko su objekt i igrač dotaknuti. Ukoliko jesu, igraču se dodaje život te se objekt dodatnog života uništava.

4.9.2. Kovanice (engl. Coins)

U ovom završnom radu implementirana je mogućnost sakupljanja novčića kojim sakupljamo bodove. Bodovi će u kasnijim implementacijama verzije igre služiti za kupovanje novih stvari koje će nam olakšavati i uljepšavati tijek igre. Također, u budućnosti će postojati određene igre na sreću u kojima će biti moguće sakupiti dodatni broj novčića ili ih izgubiti.

Implementacija novčića je u potpunosti jednostavna. Potrebno je pronaći sliku novčića te ju po želji animirati, potom joj dodati Unity komponentu „CircleCollider2D“ i namjestiti radijus te u konačnici dodijeliti skriptu „*CoinPickup.cs“*:

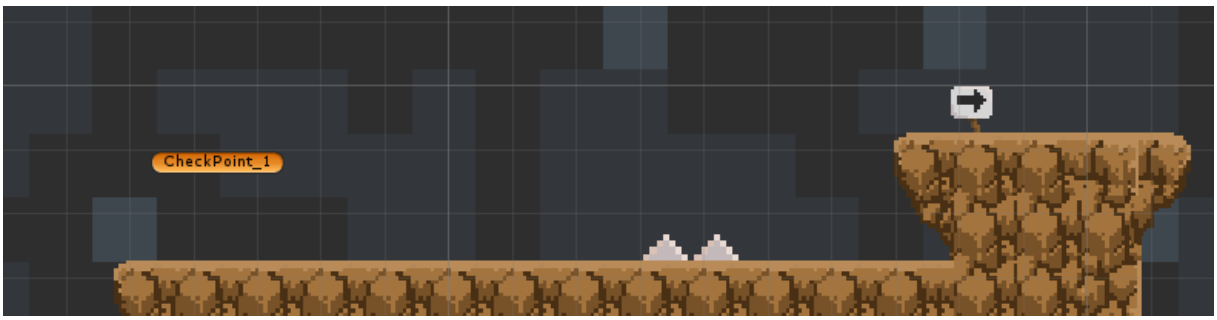
```
1  public class CoinPickup : MonoBehaviour
2  {
3    public int pointsToAdd;
4    public AudioSource coinPickupSound;
5
6    void OnTriggerEnter2D(Collider2D other)
7    {
8        if (other.GetComponent<PlayerController>() == null)
9            return;
10
11       ScoreManager.AddPoints(pointsToAdd);
12       coinPickupSound.Play();
13
14       Destroy(gameObject);
15    }
16 }
```

Sve što nam je potrebno za programsku stranu implementacije sakupljanja novčića jest varijabla tipa integer *pointsToAdd* koja označava koliko bodova ćemo dodati prilikom sakupljanja novčića. Način detekcije kolizije igrača i novčića ostvaren je pomoću događaja koja

se nalazi u linijama 7-15. Kada je bilo koji objekt u koliziji s novčićem, slijedi provjera ukoliko navedeni objekt ima pridruženu komponentu *PlayerController*. Ukoliko je prethodno navedena tvrdnja istinita, dodaju se bodovi u vrijednosti prethodno definirane varijable *pointsToAdd* te objekt novčića nestaje, odnosno uništava se.

4.9.3. Kontrolne točke (engl. Checkpoints) u igri

Kontrolne točke su veoma važan segment igre. Kontrolnu točku možemo opisati kao točku na kojoj će se igrač ponovno stvoriti ukoliko izgubi život te je broj preostalih života veći od nule. Kontrolne točke olakšavaju igraču prelazak na sljedeću razinu i stvaraju dodatnu motivaciju za bržim završetkom igre. Kontrolna točka prikazana je na slici 23 – „Primjer kontrolne točke u igri“ narančastom bojom.



Slika 23: Primjer kontrolne točke u igri

Kontrolnu točku u ovom završnom radu implementirali smo na način da je stvoren prazan objekt tipa GameObject te je na njega pridodana Unity komponenta „BoxCollider2D“ koja je postavljena kao okidač (engl. trigger). Navedeni GameObject ima također pridruženu skriptu *„Checkpoint.cs“*. Prikaz glavnog djela implementacije kontrolnih točaka sastoji se od sljedećeg koda:

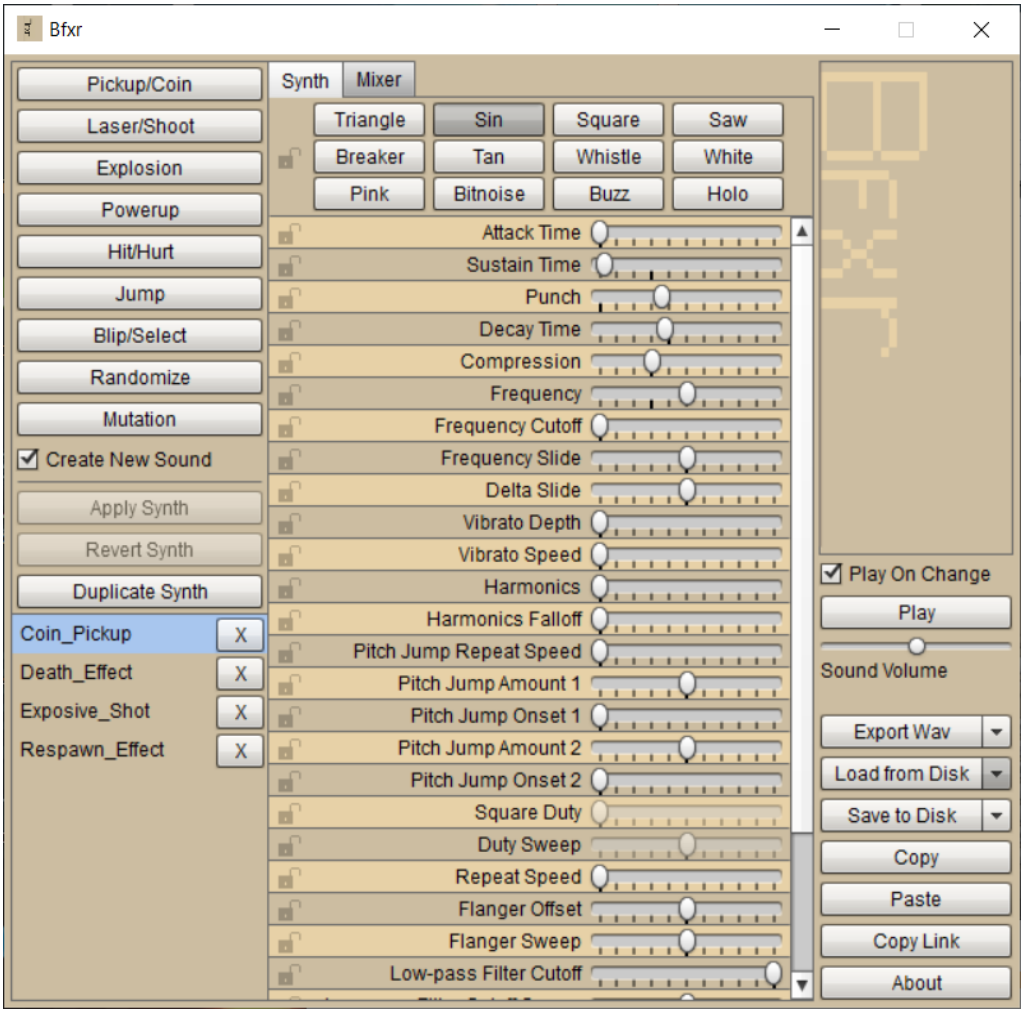
```
1  void OnTriggerEnter2D(Collider2D other)
2  {
3      (other.name == "Player")
4      {
5          levelManager.currentCheckpoint = gameObject;
6      }
7  }
```

Svaka kontrolna točka ima pridružen okidač događaja *OnTriggerEnter2D()* koji provjerava ukoliko došlo do kolizije između dva ili više objekata koji imaju definirano svojstvo tipa *Collider*. U liniji 3 vrši se provjera ukoliko je kontrolna točka u koliziji s objektom koji ima

naziv igrač. Ukoliko je prethodna tvrdnja istinita, kontrolna točka se u skripti „*LevelManager.cs*“ postavlja u posljednje dodirnutu kontrolnu točku.

4.10. Zvuk (engl. Sound)

Mnogi ljudi smatraju zvuk najvažnijom komponentom igre. Upravo iz tog razloga u ovoj igri korišteni su zvučni efekti generirani pomoću besplatnog programa „*Bfxr*“ koji je napravljen isključivo za zvučne efekte. Nadalje, u svakoj igri mora postojati i pozadinska muzika koja će doprinositi boljem ugođaju u samoj igri. Pozadinska muzika treba biti u skladu s razinom na kojoj se igrač u tom trenutku nalazi. [7]

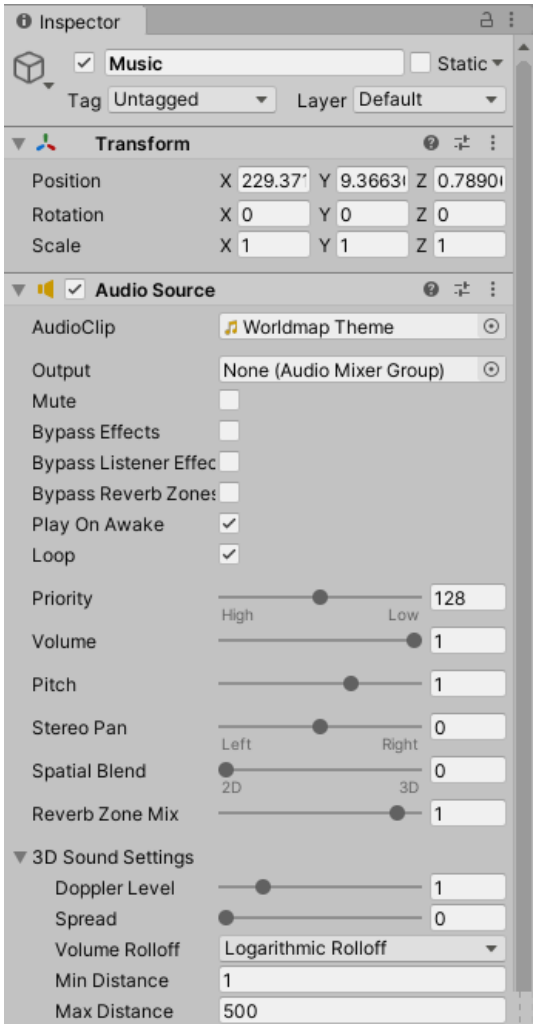


Slika 24: Program "Bfxr" za kreiranje zvučnih efekata

4.10.1. Primjer zvuka u igri

U našoj igri postoji pozadinski zvuk (engl. background sound) te zvučni efekti. Neki od postojećih zvučnih efekata su sljedeći: „coin_pickup“, „death_effect“, „explosion_sound“,

„hit_effect“, „respawn_effect“. Svaki od navedenih zvučnih efekata se pridružuje određenom objektu na način dodavanja specifične Unity komponente „Audio Source“. „Audio Source“ komponenta ima različite mogućnosti konfiguriranja, no najbitnija je „Play On Awake“, koja definira ukoliko će se zvuk pokrenuti kada se „Audio Source“ uspješno učita na objekt ili ne. „Play On Awake“ je uključen kod npr. pridruživanja „death_effect“ zvuka na „death particle“ zbog toga što se zvuk treba pokrenuti kada neprijatelj / igrač umru. S druge strane, svojstvo „Play On Awake“ je isključeno kod npr. pridruživanja „coin_pickup“ zvučnog efekta na objekt novčića, jer se zvuk mora pokrenuti tek kada je igrač u koliziji s novčićem.



Slika 25: Način implementacije muzike u programskom alatu Unity

U igri je zbog ugođaja implementirana i pozadinska muzika koja je preuzeta s stranice koja nudi besplatne zvukove za 2D igre [8]. Dobro odabrana pozadinska muzika može uveliko doprinijeti svijetu i dočaravanju igre. Na svakoj sceni u igri nalazi se određena muzika. Implementacija muzike u Unity programskom alatu za izradu igara u potpunosti je jednostavna.

Potrebno je napraviti obični prazni GameObject (na slici preimenovan u „Music“), dodati Audio Source komponentu te na svojstvo AudioClip pridružiti određeni zvuk.

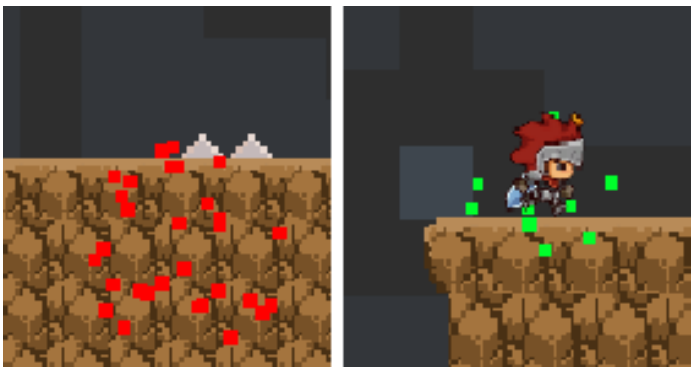
4.11. Efekti (engl. Particles)

Efekti u igrama su upravo „ono nešto“ što svaku igru čini izrazito posebnom i primamljivom. Prilikom izrade računalne igre, UI / UX dizajneri ulažu velike napore u izradu efekata igre. Postoje različiti efekti koji se javljaju u igrama: efekt prilikom stvaranja igrača, efekt prilikom smrti igrača, efekt pucanja, efekt skoka i sl. Iako naglasak ovog završnog rada nije na efektima u igri, neizostavno je bilo uraditi neke od njih te poboljšati užitak igre.

4.11.1. Primjer efekata u igri

U igri su napravljeni efekti koji se javljaju prilikom stvaranja i smrti igrača. Način implementacije efekata u programskom alatu Unity se izrađuje pomoću tzv. „Particleova“. Potrebno je stvoriti efekt tj. „Particle“ te ga urediti po želji. Prilikom uređivanja efekta postoje različiti načini za konfiguraciju istog: trajanje, ponavljanje, veličina, rotacija, boja, gravitacija, oblik, animiranje, kolizije i slično. U ovom završnom radu, korištene su samo neke od prethodno navedenih konfiguracija za izradu efekata.

Prikaz izgleda efekata nalazi se na slici 26 – „Prikaz efekata smrti i ponovnog stvaranja u igri“.



Slika 26: Prikaz efekta smrti i ponovnog stvaranja u igri

Važno je napomenuti način implementacije efekata u igri. Javlja se dosad nespomenuta mogućnost Unity programskog alata, odnosno formiranje paralelnih rutina tzv. „Coroutinea“.

```
1  public IEnumerator RespawnPlayerCo()
2  {
3      Instantiate(deathParticle, player.transform.position,
4      player.transform.rotation);
5      player.enabled = false;
6      player.GetComponent<Renderer>().enabled = false;
7
8      yield return new WaitForSeconds(respawnDelay);
9
10     player.transform.position = currentCheckpoint.transform.position;
11     player.enabled = true;
12     player.GetComponent<Renderer>().enabled = true;
13     Instantiate(respawnParticle, currentCheckpoint.transform.position,
14         currentCheckpoint.transform.rotation);
15 }
```

U funkciji *RespawnPlayerCo()* koja se nalazi u linijama 1-15, prikazana je cijela logika vezana za efekte i ponovno stvaranje igrača nakon smrti. U liniji 3-4 instanciramo efekt smrti na poziciji igrača gdje je izgubio život. U linijama 5 i 6 onemogućujemo kretanje igrača te onemogućujemo vidljivost igrača. Potom se u liniji 8 javlja čekanje trajanja *respawnDelay* kako bi se dodatno dobio efekt smrti te kako bi igrač shvatio na koji način je izgubio život. Potom u liniji 10 namještamo poziciju igrača na poziciju posljednje kontrolne točku. U linijama 10 i 12 omogućujemo kretanje igrača i vidljivost igrača te u liniji 13-14 prikazujemo efekt ponovnog stvaranja igrača na poziciji posljednje kontrolne točke. Formiranje paralelnih rutina za efekte izrazito je važno kako se ne bi glavna UI rutina zaustavila, odnosno kako bi se slijed igre nastavio odvijati normalno bez zamrzavanja ekrana.

4.12. Završetak igre

Završetak igre inspiriran je samom tematikom igre. Kao što svi znamo, u podzemlju nema zelenog drveća, niti zelenog grmlja. Što je igrač bliže izlasku na zemljinu površinu, time se pojavljuje sve više i više grmlja te u konačnici drveća. Završnica igre implementirana je na način da igrač dolazi do drveta kao simbola života (zbog toga što drveća proizvode kisik koji ljudi udišu) te pritiskom na tipku „G“ ulazi u svijet drveća. Ulaskom u svijet drveća tj. „EndGameScene“, igračev kisik u potpunosti se puni. Prilikom implementacije završetka igre koristili smo skriptu *„LevelLoader.cs“* koju smo pridružili objektu drveta:

```
1  public class LevelLoader : MonoBehaviour
2  {
3      private bool playerInZone;
4      public GameObject theText;
5
6      void Start()
7      {
8          playerInZone = false;
```

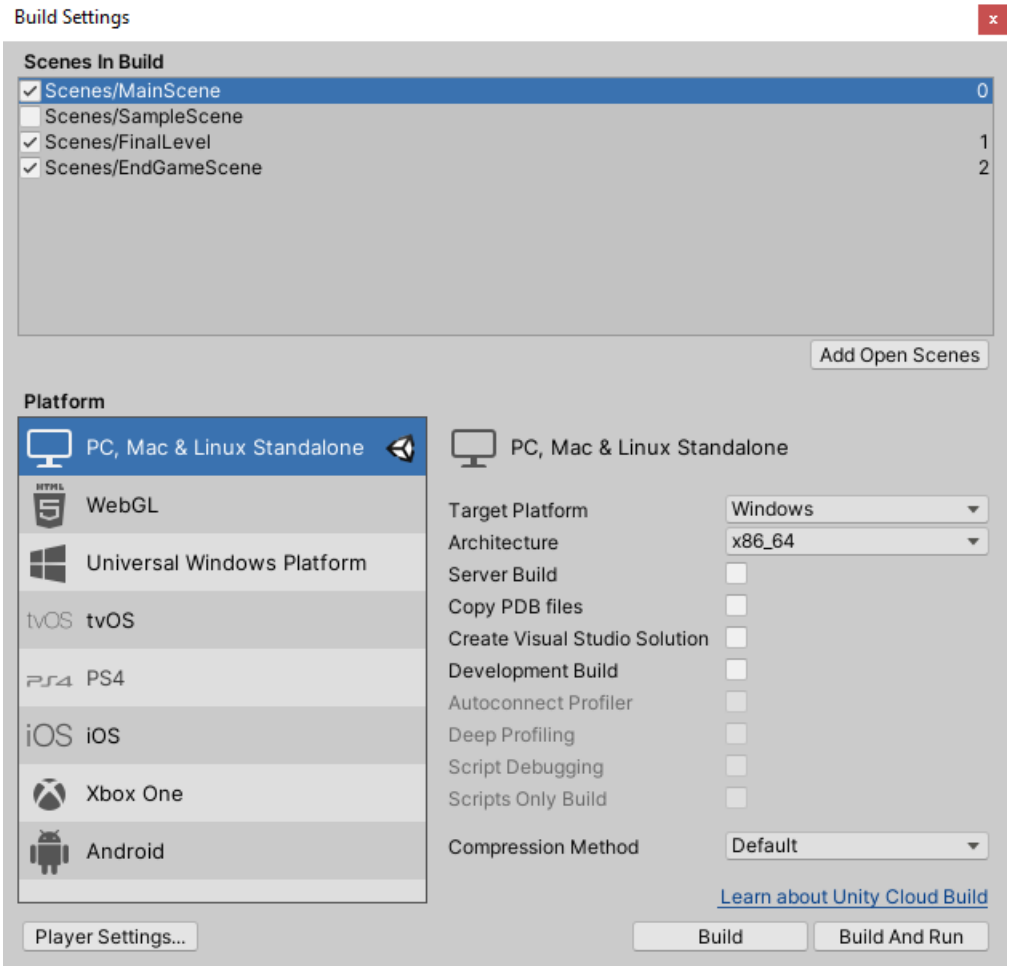
```
9       }
10
11      void Update()
12      {
13          if(Input.GetKeyDown(KeyCode.G) && playerInZone)
14          {
15          SceneManager.LoadScene("EndGameScene");
16          }
17      }
18
19      void OnTriggerEnter2D(Collider2D other)
20      {
21          if (other.name == "Player")
22          {
23              playerInZone = true;
24              theText.SetActive(true);
25          }
26      }
27
28      void OnTriggerExit2D(Collider2D other)
29      {
30          if (other.name == "Player")
31          {
32              playerInZone = false;
33              theText.SetActive(false);
34          }
35      }
36  }
```

Potrebno nam je samo jedno svojstvo *playerInZone* koje označava ukoliko je igrač na minimalnoj udaljenosti od drveta da bi se teleportirao u svijet drveća. U početku je svojstvo *playerInZone* postavljeno na vrijednost laži (engl. false). Postoje dva upravitelja događaja: *OnTriggerEnter2D()* koji postavlja svojstvo *playerInZone* u istinu ukoliko je igrač na minimalnoj udaljenosti do drveta, odnosno *OnTriggerExit2D()* koji postavlja svojstvo *playerInZone* u neistinu ukoliko igrač nije na minimalnoj udaljenosti do drveta. U metodi *Update()* provjeravamo ukoliko je igrač u zoni te ukoliko je pritisnuta tipka enter. U tom slučaju učitava se novi level „EndGameScene“.

4.12.1. Izrada izvršne datoteke

Izrada izvršne datoteke u Unityu je veoma pristupačna i jednostavna. Nije potrebno izdvajati puno vremena za izradu izvršne datoteke kao kod većine aplikacija u računalnoj industriji. Unity pruža grafičko sučelje za izradu izvršnih datoteka za različite uređaje. Izvršna datoteka može biti namjenjena za PC, Web, iOS, Android, PS4, XBox One itd. U ovom završnom radu će biti izrađena izvršna datoteka za operacijski sustav Windows. Sve što je potrebno za izradu izvršne datoteke je odabrati scene koje želimo da izvršna datoteka sadrži.

Zatim je potrebno odabrati platformu i arhitekturu za koju želimo izraditi izvršnu datoteku i pritisnuti gumbić „Build“.



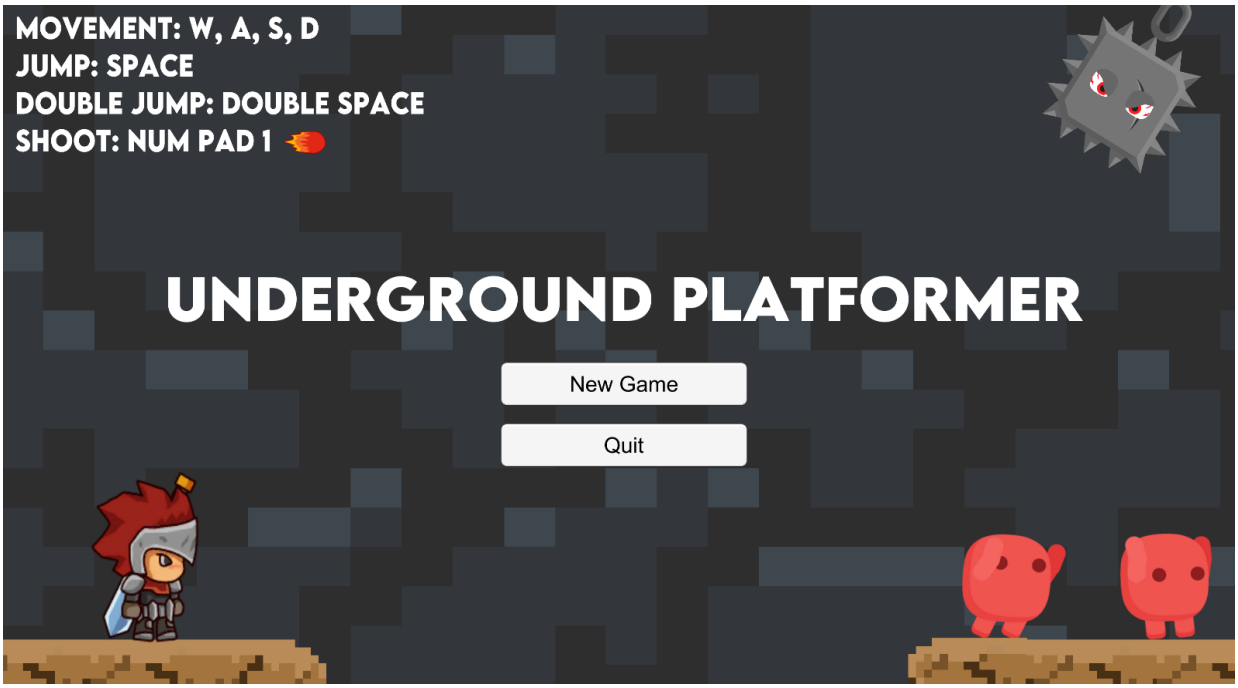
Slika 27: Način izrade izvršne datoteke

5. Grafičko sučelje igre (engl. Game UI)

Grafičko sučelje igre (Game UI) predstavlja način komunikacije korisnika s našom aplikacijom. Kao grafičko sučelje možemo zamisliti glavni meni aplikacije u kojem igrač može odabrati određene opcije. Odabir opcija obično se izvršava kroz interaktivne gumbiće pomoću kojih se izvršavaju određene metode i sl. Grafičkom sučelju igre pridonosi i intuitivnost i logičnost određenih interakcija tj. „User Experience“ (UX).

5.1. Početni meni

Početni meni svake igrice je prva stvar koju igrač uočava prilikom pokretanja iste. Iz navedenog razloga, početni meni mora izgledati primamljivo i intuitivno. Samim time je veća šansa da ćemo igrača zainteresirati za našu igru te da će ju pokrenuti. U ovom završnom radu odrađen je jednostavni meni koji prikazuje funkcionalnosti pokretanja igre i izlaska iz igre zbog toga što naglasak ovog završnog rada nije na dizajnu.



Slika 28: Prikaz početnog menija

Način implementacije početnog zaslona je u potpunosti jednostavan. Potrebno je napraviti dizajn te osnovne funkcionalnosti za pokretanje određenih scena. U našem završnom radu prikazujemo samo 2 funkcionalnosti: „New Game“ i „Quit“. One su prikazane u klasi „MainMenu“:

```
1  public class MainMenu : MonoBehaviour
2  {
3      public string startLevel;
4      public int playerLives;
5
6      public void NewGame()
7      {
8          PlayerPrefs.SetInt("PlayerCurrentLives", playerLives);
9          PlayerPrefs.SetInt("PlayerCurrentScore", 0);
10         SceneManager.LoadScene(startLevel);
11     }
12
13     public void QuitGame()
14     {
15         Application.Quit();
16     }
17 }
```

U prethodno priloženom kodu vidimo samo dva svojstva: *startLevel* - tipa string koje definira naziv početne tj. prve razine i *playerLives* – tipa int koji definira broj života igrača. U linijama 7-11 nalazi se funkcija NewGame() koja postavlja igračeve živote na vrijednost varijable playerLives te igračeve bodove na 0. U liniji 10 koristi se statička klasa „SceneManager“ i njezina metodu LoadScene() koja služi za učitavanje određene scene. U linijama 14-16 prikazana je funkcija QuitGame() koja služi za izlazak iz igre.

5.2. UI prilikom pokrenute igre

Prikaz korisničkog sučelja tokom igre u potpunosti je pojednostavljen i razumljiv. Postoje tri svojstva: *points* – koji broji količinu bodova koju smo sakupili tokom igre, *lives* – koji broji preostale živote u igri, *oxygen* – količina preostalog kisika u sekundama.

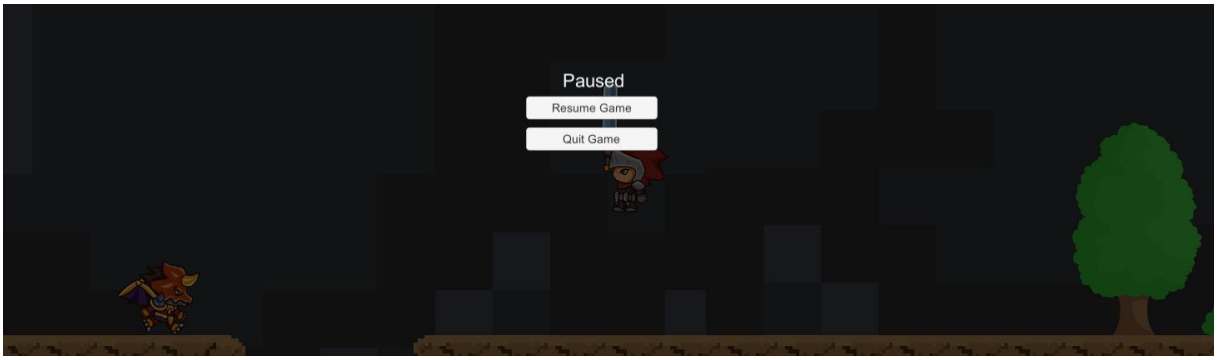
Slika 29: Prikaz korisničkog sučelja tokom igre

Bodovi tj. „points“ se u igri skupljaju ubijanjem protivnika te sakupljanjem određenih kovanica koje su razbacene kroz čitavu mapu. Igrač može izgubiti život te se tada komponenta korisničkog sučelja „Lives“ smanjuje. Igrač također može pokupiti dodatni život te se tada

komponenta korisničkog sučelja „Lives“ povećava. Komponenta korisničkog sučelja kisik tj. „Oxygen“ neprestano se smanjuje.

5.3. Meni pauze

U ovom završnom radu implementiran je meni pauze u koji igrač ulazi pritiskom na tipku ESC. Izlazak iz menija pauze moguć je pritiskom na tipku „Resume Game“ ili pomoću ponovnog pritiska tipke ESC. Način implementacije menija pauze je u potpunosti trivijalan za izradu te je implementiran u klasi *„PauseMenu“.*



Slika 30: Izgled menija pauze u igri

Potrebne su nam sljedeća svojstva: *mainMenu* (string) koji označava naziv scenarija glavnog menija, *isPaused* (bool) koji označava ukoliko je igra pauzirana ili nije te *pauseMenuCanvas* (GameObject) koji će označavati platno menija pauze. Implementacija menija pauze nalazi se u klasi *„MenuPause“*:

```
1  public class PauseMenu : MonoBehaviour
2  {
3      public string mainMenu;
4      public bool isPaused;
5      public GameObject pauseMenuCanvas;
6
7      void Start()
8      {
9          isPaused = false;
10     }
11
12     void Update()
13     {
14         if (isPaused)
15         {
16             pauseMenuCanvas.SetActive(true);
17             Time.timeScale = 0f;
18         }
```

```
19          else
20          {
21              pauseMenuCanvas.SetActive(false);
22              Time.timeScale = 1f;
23          }
24
25          if (Input.GetKeyDown(KeyCode.Escape))
26          {
27              isPaused = !isPaused;
28          }
29      }
30
31      public void Resume()
32      {
33          isPaused = false;
34      }
35
36      public void QuitToMainMenu()
37      {
38          SceneManager.LoadScene(mainMenu);
39      }
40  }
```

U metodi *Start()* postavljamo vrijednost svojstva *isPaused* na vrijednost false, čime označavamo da igra prilikom njezina pokretanja nije pauzirana. U metodi *Update()* koja se izvršava više puta u sekundi provjeravamo ukoliko je igra pauzirana ili nije (linije 14-23) te u linijama 26-28 provjeravamo ukoliko je korisnik pritisnuo gumb ESC. Ukoliko je igra pauzirana, izvršava se blok koda između linija 15-18 prilikom čega se prikazuje platno *pauseMenuCanvas* te se brzina toka igre postavlja na 0 (tj. igra se zaustavlja, odnosno zamrzava). Ukoliko igra nije pauzirana, izvršava se blok koda između linija 20-23 prilikom čega se sakriva platno *pauseMenuCanvas* te se brzina toga igre postavlja na 1 (tj. igra ima normalni vremenski tok i nije više zamrznuta).

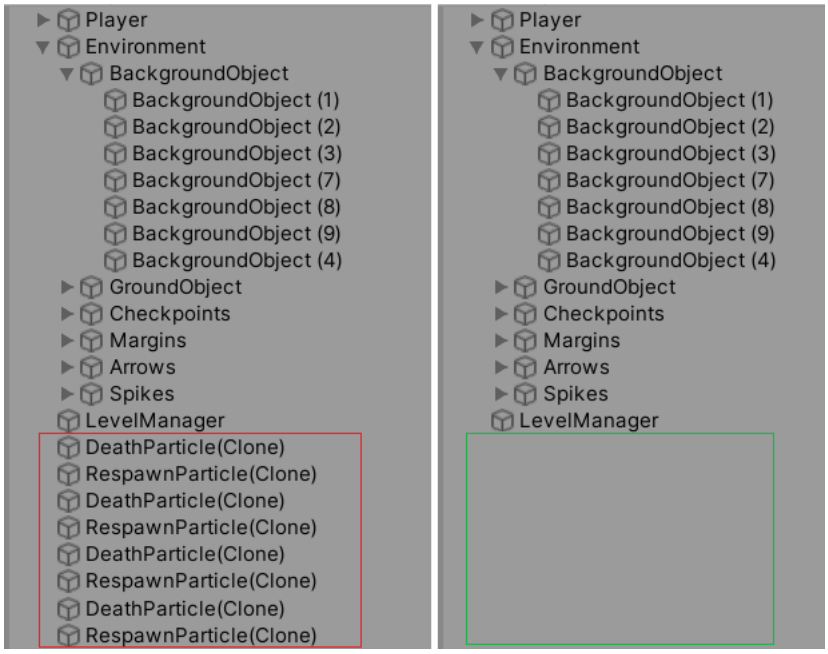
U linijama 32-34 nalazi se metode *Resume()* čiju funkcionalnost pridružujemo gumbiću „Resume Game“. Pritiskom na gumbić „Resume Game“ igra više nije pauzirana. U 36-39 prikazana je metoda *QuitToMainMenu()* u kojoj koristimo klasu *„SceneManager“* te njezinu statičku metodu *LoadScene()* pomoću koje učitavamo određenu scenu.

6. Poboljšavanje performansi igre

Svakom ljubitelju računalnih igara poznati su problemi zamrzivanja ekrana, sporijeg rada igre, gubljenja određenih sličica u igri i sl. Ponekad se navedeni problemi javljaju iz razloga slabe konfiguracije računala, a ponekad i iz loše implementacije igre od strane programera. U ovom završnom radu, bit će opisana neka od mogućih poboljšanja igre te sprječavanja prethodno navedenih problema.

6.1. Ručno izrađeni sakupljač iskorištenih efekata

Kod kreiranja određenih vizualnih efekata (engl. Particles), postoji problem automatskog ne-brisanja. Ukoliko se određeni vizualni efekti često pojavljuju, a pritom ne brišu, dolazimo do zaključka da će radna memorija računala biti uvelike upotpunjena nepotrebno iskorištenim podacima koji više ne služe ni za kakvu svrhu... Upravo iz prethodno navedenog razloga moguća je pojava zamrzavanja ekrana, sporijeg rada igre i sl.



Slika 31: Problem punjenja RAMa iskorištenim efektima

Prikaz punjenja memorije nepotrebno iskorištenim podacima nalazi se na slici 31 - „Problem punjenja RAMa iskorištenim efektima“. Na lijevoj slici nalazi se prikaz već iskorištenih efekata koji se više ne pojavljuju na zaslonu te bez ikakve koristi nepotrebno zauzimaju mjesto u radnoj memoriji. Možemo zamisliti situaciju kada bi postojalo preko 1000 takvih objekata. Tada bi RAM bio gotovo prepun nepotrebno iskorištenih efekata.

Način na koji rješavamo prethodni problem jest kreiranjem skripte koja će provjeravati ukoliko je određeni efekt završio. U slučaju da je efekt završio, potrebno ga je ukloniti iz igre.

Uklanjanjem efekta iz igre, oslobodit će se i mjesto u radnoj memoriji čime postižemo veću fluidnost igre. Prikaz poboljšanja nalazi se u klasi *„DestroyFinishedParticle“:*

```
1  public class DestroyFinishedParticle : MonoBehaviour
2  {
3      private ParticleSystem thisParticleSystem;
4
5      void Start()
6      {
7          thisParticleSystem = GetComponent<ParticleSystem>();
8      }
9
10     void Update()
11     {
12         if (thisParticleSystem.isPlaying)
13             return;
14
15         Destroy(gameObject);
16     }
17 }
```

Navedenu skriptu potrebno je staviti na određeni efekt (engl. Particle) kako bi ispravno radila. Način rada sakupljača jest sljedeći: preuzima se svojstvo „ParticlySystem“ te se više puta u sekundi provjerava ukoliko je efekt završen. Ukoliko je efekt završen, on se briše iz igre, a samim time i iz radne memorije. Upravo iz tog razloga, manja je mogućnost sporijeg rada igre, zamrzavanja ekrana i sl.

6.2. Uništavanje projektila nakon određenog vremena

Jedno od poboljšanja u ovoj igri jest uništavanje projektila ispaljenih od strane igrača nakon određenog vremena. Prethodno navedena rečenica omogućava nam da se objekti ne kreću kroz prostor sve dok ne dodirnu drugi objekt, već da nakon nekog vremena nestanu iz igre. Funkcionalnost uništavanja projektila nakon određenog vremena implementirana je u klasi *„DestroyObjectOverTime“:*

```
1  public class DestroyObjectOverTime : MonoBehaviour
2  {
3      public float lifetime;
4
5      void Update()
6      {
7          lifetime -= Time.deltaTime;
8
9          if (lifetime < 0)
10             Destroy(gameObject);
11     }
12 }
```

Postoji jedno svojstvo *lifetime* koje određuje nakon koliko sekundi će se objekt uništiti. Također postoji i već svima poznata funkcija *Update()* u kojoj se lifetime svaki okvir (engl. frame) smanjuje za vrijednost Time.deltaTime. Kada je svojstvo *lifetime* poprimilo vrijednost manju od 0, objekt se uništava.

7. Zaključak

Završni rad u potpunosti je ispunio moja očekivanja te osnovne funkcionalnosti 2D igre. Svaka 2D igra mora imati osnovne kontrole, pozadinski zvuk, zvučne efekte, vizualne efekte, kontrolne točke itd. Ukoliko igraču želimo zakomplicirati određenu razinu, mogu se ubaciti različite vrste neprijatelja i prepreka (npr. šiljci odozdo, šiljci odozgo, kretajuće platforme i sl).

Programski alat za razvoj računalnih igara „Unity“ u kombinaciji s programskim jezikom C# (.NET) čine odličnu kombinaciju za početak karijere u industriji video igara. Unity je veoma jednostavan alat koji pruža grafičko sučelje kao pomoć programeru kod razvoja računalnih igara. U potpunosti je intuitivan za korištenje.

Budućnost programskom alata Unity nije upitna. Unity će nastaviti trend svog širenja i globaliziranja industrije videoigara zbog njegove prethodno spomenute jednostavnosti i intuitivnosti. Računalne igre, kao ni programski alat Unity, također nemaju upitnu budućnost. Sasvim je sigurno da će računalne igre u bližoj, ali i daljnjoj budućnosti ostati veoma popularne.

Ovaj završni rad bio je početak moje buduće karijere u industriji računalnih igara. Naučio sam jako puno novih stvari uz razvoj računalnih igara te već sada samostalno mogu razumjeti načine implementacije određenih igrica. Veoma sam zadovoljan konačnim ishodom ovog završnog rada.

8. Popis literature

[1] *History of Video Games* (rujan 2020.), U Wikipedia. Preuzeto 25. svibnja 2020. s https://en.wikipedia.org/wiki/History_of_video_games

[2] Ahoy (16. rujan 2013.) *A Brief History of Video Games* [Video File]. Preuzeto 25. svibnja 2020. s https://www.youtube.com/watch?v=GoyGlyrYb9c

[3] *Unity (*2020.*)*, U Wikipedia. Preuzeto 27. svibnja 2020. s https://en.wikipedia.org/wiki/Unity_(game_engine)

[4] Unity (2019.) *Unity's Package Manager*. Preuzeto 04. lipnja 2020. s https://docs.unity3d.com/Manual/Packages.html

[5] Unity (2019.) *2D Game Resources*. Preuzeto 05. lipnja 2020. s https://unity.com/how-to/beginner-2D-game-resources

[5] Ethan Scully (2019.) *How To Make A Simple 2D Game in Unity*. Preuzeto 03. lipnja 2020. s https://careerkarma.com/blog/make-a-game-in-unity/

[6] Tim Borzon (2019.) *How to Build a Complete 2D Platformer in Unity*. Preuzeto 03. lipnja 2020. s https://gamedevacademy.org/how-to-build-a-complete-2d-platformer-in-unity/

[7] Bfxr (bez dat.) *What is Bfxr*. Preuzeto 04. srpnja 2020. s: https://www.bfxr.net/

[8] CodeManu (2015.) *Platformer Game Music Pack*. Preuzeto 15. srpnja 2020. s https://opengameart.org/content/platformer-game-music-pack

[9] Brackeys (2018.) *How to make a 2D Game in Unity. Preuzeto 03. lipnja 2020. s* https://www.youtube.com/watch?v=on9nwbZngyw&list=PLPV2KyIb3jR6TFcFuzI2bB7TMNIIBpKMQ

[10] gamesplusjames (2015.) *Unity 2D Platformer Tutorial – Learn The Basic Of Making a Game* [Video file]. Preuzeto 10. lipnja 2020. s https://www.youtube.com/watch?v=-ixk2uxEc94&list=PLiyfvmtjWC_Up8XNvM3OSqgbJoMQgHkVz

[11] Unity (2020.) *Unity Core Platform*. Preuzeto 25. svibnja 2020. s https://unity.com/products/core-platform

[12] Brackeys (2017.) *Smooth Camera Follow In Unity – Tutorial*. [Video file] Preuzeto 23. srpnja 2020. s https://www.youtube.com/watch?v=MFQhpwc6cKE

8.1. Popis slika

8.2. Izvori slika

- Slika 1: https://en.wikipedia.org/wiki/Spacewar!#/media/File:Spacewar_gameplay.jpg
- Slika 2: https://en.wikipedia.org/wiki/File:Pong.png
- Slika 3: https://en.wikipedia.org/wiki/Space_Invaders#/media/File:SpaceInvaders-Gameplay.gif
- Slika 4: https://www.google.com/search?q=super+mario+bros&sxsrf=ALeKk01vWNAZSxeYmWvea5wxR0z14B1u7g:1591347786663&source=lnms&tbm=isch&sa=X&ved=2ahUKEwi2zqa8qOrpAhW1AxAIHQFtD-EQ_AUoAXoECBYQAw&biw=1745&bih=881#imgrc=tWGXf_4HzecQ_M
- Slika 5: https://www.google.com/search?q=first+starcraft+game&tbm=isch&ved=2ahUKEwjmte2Li-TgAhUJkaQKHXaAA1gQ2-cCegQIABAA&oq=first+starcraft+game&gs_lcp=CgNpbWcQAzIECAAQEzoECCMQJzoFCAAQsQM6AggAOgQIABBDOgQIABAeOggIABAIEB4QEzoGCAAQHhATULFJWMJfYJ5gaABwAHgAgAF8iAH6DpIBBDExLjiYAQCgAQGqAQtnd3Mtd2l6LWltZ8ABAQ&sclient=img&ei=DOIZX-
- Slika 6: https://www.google.com/search?q=world+of+warcraft+2004&tbm=isch&ved=2ahUKEwim54KquurpAhWKuKQKHXTEBJUQ2-cCegQIABAA&oq=world+of+warcraft+2004&gs_lcp=CgNpbWcQAzIECCMQJzIECAAQEzIECAAQEzIECAAQEzIECAAQEzIECAAQEzIECAAQEzIECAAQE1DTJljaLGCHLWgAcAB4AIABa4gBrAKSAQMyLjGYAQCgAQGqAQtnd3Mtd2l6LWltZw&sclient=img&ei=BB3aXqbVCYrxkgX0iJOoCQ&bih=881&biw=1745#imgrc=AnU7IR8rcrxVwM
- Slika 7: https://en.wikipedia.org/wiki/Unity_(game_engine)#/media/File:Unity_Technologies_logo.svg
- Slika 8: https://www.google.com/search?q=temple+run&sxsrf=ALeKk01WFj4CWeKGjISzuxmoT8-GI4-xlQ:1591355259965&source=lnms&tbm=isch&sa=X&ved=2ahUKEwiA6OynxOrpAhWIE5oKHYfwBhEQ_AUoAXoECBcQAw&biw=1745&bih=881#imgrc=dsR1CgU-0BXX5M
- Slika 9: https://www.google.com/search?q=plague+inc&tbm=isch&ved=2ahUKEwjzmPqoxOrpAh

VGtqQKHRxoCmwQ2-cCegQIABAA&oq=plague+inc&gs_lcp=CgNpbWcQAzIECCMQJzIECAAQHjIECAAQHjIECAAQHjIECAAQHjIECAAQHjIECAAQHjIECAAQHjIECAAQHjoFCAAQsQM6BAgAEEM6AggAUMx3WKd8YNx8aABwAHgAgAFuiAGaB5IBAzIuN5gBAKABAaoBC2d3cy13aXotaW1n&sclient=img&ei=fifaXvPhEMbskgWc0KngBg&bih=881&biw=1745#imgrc=0Z3xtgMrCt8o3M