2D platformer računalna igra sa samoučećim agentom

Čalogović, Mihaela

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
Sveučilišni prijediplomski studij računarstva

Završni rad

2D platformer računalna igra sa samoučećim agentom

Rijeka, srpanj 2023.

Mihaela Čalogović
0069085725

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
Sveučilišni prijediplomski studij računarstva

Završni rad

2D platformer računalna igra sa samoučećim agentom

Mentor: prof. dr. sc. Ivan Štajduhar

Rijeka, srpanj 2023.

Mihaela Čalogović
0069085725

Umjesto ove stranice umetnuti zadatak za završni ili diplomski rad

Izjava o samostalnoj izradi rada

Izjavljujem da sam samostalno izradila ovaj rad.

Rijeka, srpanj 2023.

_____
Mihaela Čalogović

Sadržaj

Popis slika

Uvod

Umjetna inteligencija (UI) predstavlja jedno od najdinamičnijih područja suvremene tehnologije i istraživanja. Primjenjuje se u različitim domenama, kao što su na primjer autonomna vozila, robotska manipulacija, analiza podataka, medicinska dijagnostika i, naravno, razvoj računalnih igara. U ovom završnom radu istražuje se primjena umjetne inteligencije u kontekstu razvoja računalne igre s naglaskom na modeliranje i animaciju glavnog lika, izradu same računalne igre te razvoj inteligencije koja optimizira rješavanje igre.

Glavni cilj ovog istraživanja je kombinacija naprednih tehnika umjetne inteligencije s tehnikama modeliranja i animacije te tehnikama izrade računalne igre kako bi se postiglo rješavanje razina umjetnom inteligencijom. U tu svrhu, u ovom radu je izrađen model glavnog lika i njegove animacije koristeći specijalizirani alat za piksel art grafiku, *Aseprite*, dok je za razvoj same igre korišten *Unity engine* - popularan alat koji omogućuje izradu igara za različite platforme. Osim toga, izrađeni su kompleksni samoučeći agenti koji optimiziraju rješavanje različitih razina igre. Za razvoj inteligencije glavnog lika u računalnoj igri korišten je *Unity ML-Agents*, alat koji pruža okvir za razvoj naprednih algoritama učenja, donošenja odluka i simulacije inteligentnog ponašanja likova.

Istraživanje se također usredotočuje na integraciju različitih aspekata umjetne inteligencije u računalnu igru, kao što su strojno učenje i donošenje odluka, kako bi se ostvarila visoka razina autonomnosti i adaptivnosti u ponašanju glavnog lika unutar računalne igre. Kroz ovaj rad će se detaljno analizirati postupci razvoja modela i animacija, izrada računalne igre u *Unity engineu*, kao i primjena samoučećih agenata koristeći *Unity ML-Agents*.

Poglavlje 1

Izrada modela i animacija za glavnog lika

1.1 Izmjena programa za izradu modela i animacija

U tekstu zadatka navedeno je da će se izrada modela i animacija za glavnog lika raditi u programu *Blender*[1]. *Aseprite*[2] i *Blender* su softverski alati koji se često koriste u industriji razvoja igara, ali imaju različite namjene i funkcionalnosti. *Aseprite* je specijaliziran za izradu i animaciju 2D piksel art grafike, dok je *Blender* višenamjenski alat koji se koristi za 3D modeliranje, animaciju i renderiranje.

Korištenje *Asepritea* umjesto *Blendera* u ovom slučaju ima nekoliko prednosti u kontekstu izrade modela i animacije glavnog lika u 2D računalnoj igri. *Aseprite* pruža alate koji su posebno prilagođeni za izradu piksel art grafike, što može rezultirati visokokvalitetnim i detaljnim dizajnima u 2D stilu. Osim toga, *Aseprite* nudi podršku za animaciju piksel arta, što omogućuje kretanje i transformaciju likova na jednostavan način.

S druge strane, *Blender*, iako snažan alat za 3D, može biti kompleksan za korištenje samo u svrhu 2D animacije. Njegova snaga leži u kompleksnom 3D modeliranju, simulacijama i renderiranju, što je suvišno ako je potrebna samo 2D animacija.

Dakle, odabir *Asepritea* umjesto *Blendera* za izradu modela i animacije glavnog lika u 2D igrici pruža specijalizirane alate, jednostavnost upotrebe i fokus na piksel

art stil, što dovodi do bržeg i efikasnijeg procesa izrade i boljih rezultata u konačnom proizvodu.

1.2 Izrada modela glavnog lika

Izrada modela za glavnog lika u računalnoj igri predstavlja važan korak u postizanju vizualne privlačnosti i prepoznatljivosti. *Aseprite* u tom segmentu nudi intuitivno sučelje i funkcionalnosti koje olakšavaju proces stvaranja likova.

Za potrebe ovog rada korišten je tutorial[3] za *Aseprite* da se bolje razumije rad u *Asepriteu* i korištenje samog *Asepritea* te dizajniranje modela glavnog lika.

Prvi korak u izradi modela je bio skiciranje dizajna lika psa. Ovaj korak je uključivao skiciranje i planiranje izgleda lika, boja, proporcija i drugih karakteristika kako bi se postigao željeni izgled i osobnost psa.

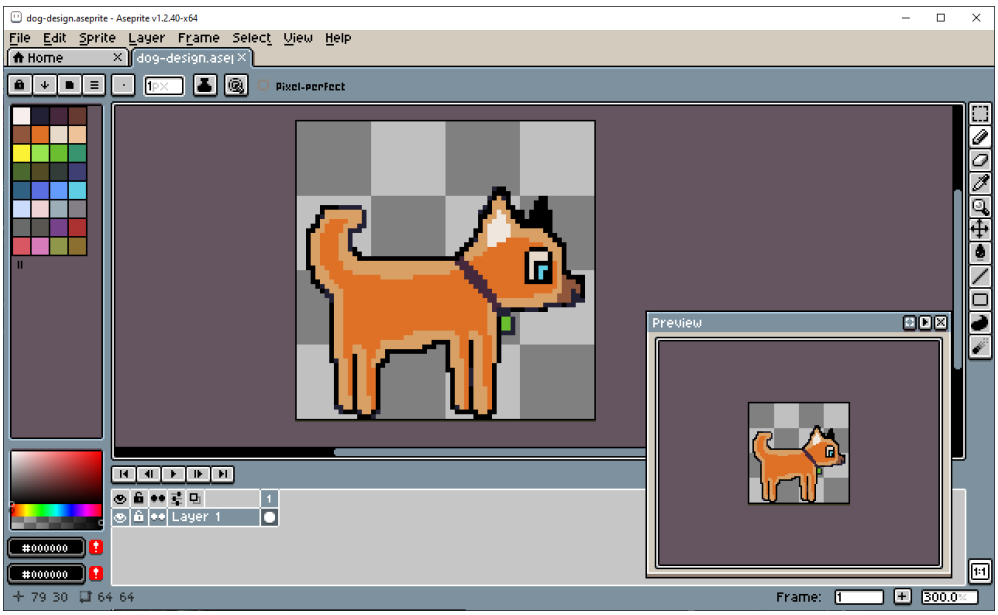
Nakon što je skica dizajna definirana, započela je izrada osnovne siluete lika u *Asepriteu*. Korišteni su alati za crtanje, kao što su linije i oblici, kako bi se definirale osnovne konture tijela, glave, ušiju, repa i drugih karakterističnih dijelova psa.

Sljedeći korak bio je dodavanje detalja kako bi se liku dalo više karakteristika i prepoznatljivosti. Korišteni su alati za crtanje i oblikovanje kako bi se stvorili elementi poput očiju, njuške, dlake i drugih karakterističnih detalja psa.

Kada su oblik i detaljni lika bili zadovoljavajući, slijedio je korak bojanja. *Aseprite* je omogućio odabir palete boja i precizno bojanje svakog piksela lika, što je omogućilo dodjeljivanje odgovarajućih boja različitim dijelovima psa.

Proces izrade modela za glavnog lika u *Asepriteu* bio je kreativan i prilagodljiv. Kroz eksperimentiranje s različitim tehnikama, stilovima i detaljima, postignut je željeni izgled i prepoznatljivost lika psa u skladu s vizijom.

Na slici 1.1 prikazan je prvi dizajn modela glavnog lika u *Asepriteu*.



Slika 1.1 Model glavnog lika

1.3 Izrada animacija glavnog lika

Pri izradi animacije prvo se definira animacijski okvir, to jest određuje se broj okvira koji će činiti animaciju. Svaki okvir predstavlja pojedini trenutak u animaciji i sadrži promjene u položaju, izgledu ili izražavanju lika.

Nakon toga slijedi kreiranje animacijskih okvira u animaciji. Postavljanje animacijskih okvira pomaže u stvaranju osnovnih pokreta i animacijskog ritma.

Kada je napravljena osnovna animacija dolazi dodavanje detalja i efekata. Tijekom procesa izrade animacije moguće je dodati detalje, izraze lica, efekte i druge elemente koji poboljšavaju vizualni dojam animacije. *Aseprite* nudi alate za crtanje i uređivanje koji olakšavaju dodavanje ovih detalja i efekata.

Kada je animacija dovršena, proveden je pregled kako bi se provjerila kvaliteta pokreta. Potom je animacija izvezena u format sprite liste.

Za potrebe ovog rada također je korišten jedan već postojeći model i animacije za glavnog lika psa tako da bi se dobila inspiracija za izradu različitih okvira

animacije[4].

1.3.1 Izrada neaktivne animacije

Neaktivna animacija se fokusira na prikazivanje lika u statičnom stanju, obično dok miruje ili čeka neku radnju. Proces izrade neaktivne animacije započinje s definiranjem položaja i izgleda lika tijekom neaktivnosti. To uključuje postavljanje lika u prirodan i uravnotežen položaj, prilagođavanje izraza lica i tijela.

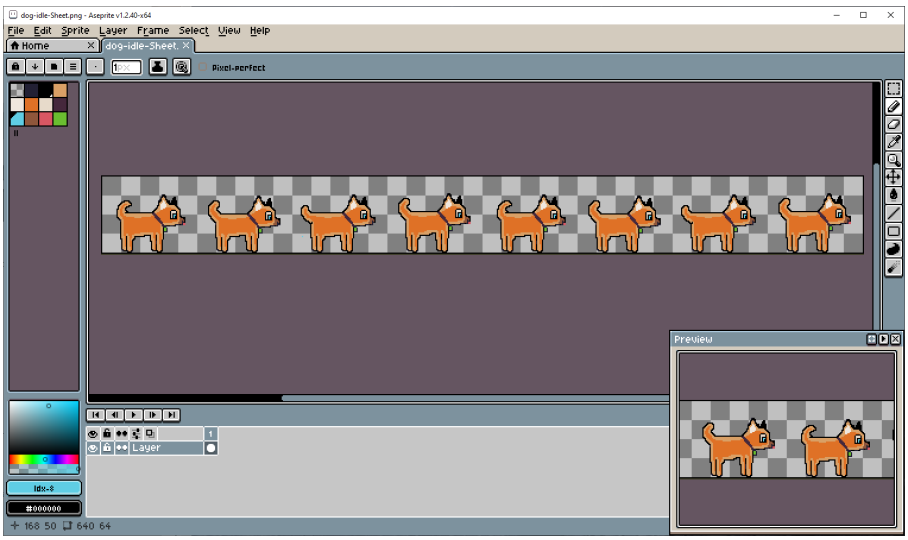
Nakon što je lik postavljen u željeni položaj, koriste se animacijski okviri kako bi se stvorio osjećaj suptilnih pokreta ili varijacija u izgledu lika. To može uključivati lagano pomicanje dijelova tijela, kao što su mahanje repa, plaženje jezika te pomicanje tijela prema gore i prema dole tako da se dobije dojam disanja glavnog lika.

Važno je pridodati pažnju detaljima i kvaliteti izvedbe u neaktivnoj animaciji kako bi se postigla vjerodostojnost i privlačnost lika. To može uključivati fino podešavanje detalja poput sjene, svjetla, teksture ili drugih vizualnih elemenata kako bi se poboljšala kvaliteta animacije.

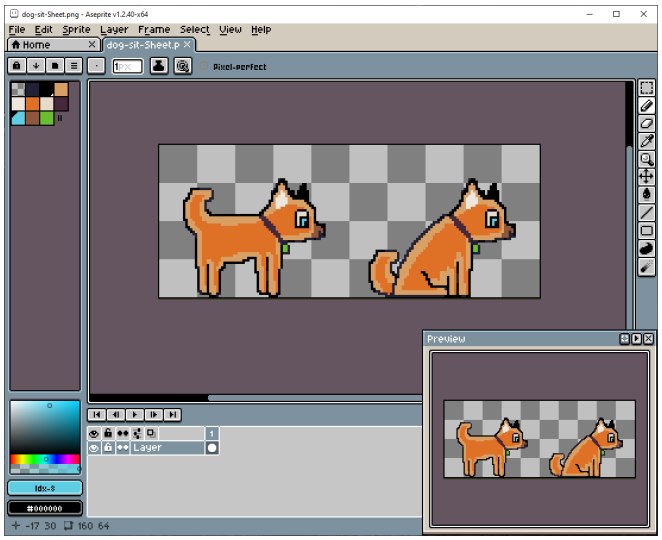
Konačno, nakon završetka izrade neaktivne animacije, proveden je pregled kako bi se osigurala dosljednost, prepoznatljivost i estetska privlačnost. Pregled je omogućio identifikaciju potencijalnih poboljšanja ili ispravaka prije nego što se animacija integrirala u računalnu igru.

Za prikaz neaktivnosti glavnog lika određeno je da će se koristiti 8 animacijskih okvira koji prikazuju pomicanje tijela. Na slici 1.2 prikazano je tih 8 animacijskih okvira.

Također je na kraju u neaktivnoj animaciji dodan i vremenski nešto duži animacijski okvir za prikaz psa dok sjedi tako da se još bolje prikaže opuštenost glavnog lika. To je samo jedan animacijski okvir koji je dodan na kraju animacije, koja se onda vraća natrag na prvi animacijski okvir neaktivne animacije. Na slici 1.3 prikazan je taj animacijski okvir.



Slika 1.2 Neaktivna animacija



Slika 1.3 Animacija sjedenja

1.3.2 Izrada animacije skakanja

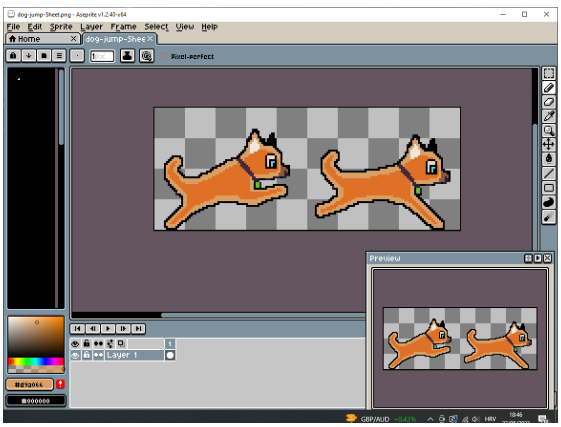
Animacija skakanja igra ključnu ulogu u računalnim igrama jer liku pruža dinamičnost, pokretljivost i interaktivnost s okolinom.

Animacija skakanja započinje sa naglim odbijanjem glavnog lika od podloge, čime se pokreće vertikalno kretanje prema gore. Tijekom tog pokreta, stražnje noge se pružaju prema dolje, a prednje noge se koriste za isticanje energije skoka.

Nakon toga, lik počinje silaziti prema dolje, prateći zakone gravitacije. Prednje noge su u tom trenutku ispružene prema dole kako bi apsorbirale udarac pri slijetanju.

U ovoj animaciji također je bilo važno pridodati pažnju detaljima poput sjene, svjetla i teksture kako bi se animacija skakanja dodatno poboljšala. Pregled animacije omogućio je identifikaciju mogućih poboljšanja ili ispravaka kako bi se postigla kvalitetna i uvjerljiva animacija skakanja.

Na slici 1.4 prikazana su ta dva animacijska okvira koja prikazuju animaciju za skakanje.



Slika 1.4 Animacija skakanja

1.3.3 Izrada animacije hodanja

Animacija hodanja je ključni element u računalnim igrama jer pruža osjećaj da se lik kreće i da je povezan s okolinom.

Prvi korak u izradi animacije hodanja je određivanje koraka i ciklusa. Hodanje se sastoji od ponavljanja sličnih koraka kako bi se stvorio dojam neprekidnog kretanja. Definiranje duljine koraka, brzine hodanja i ostalih karakteristika pomaže u

postizanju prirodnog izgleda animacije.

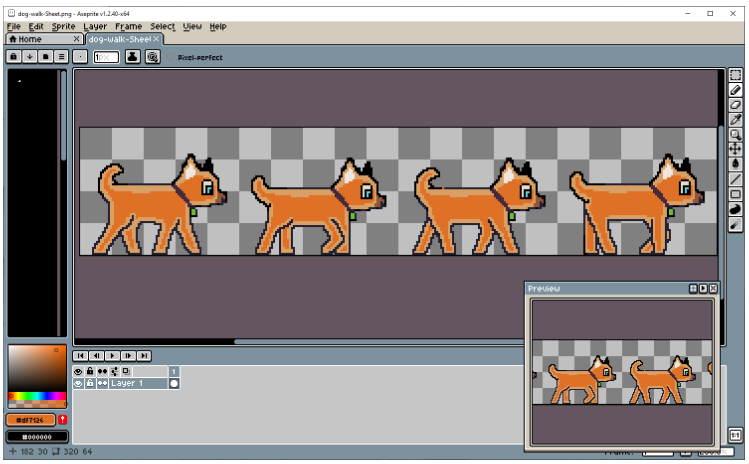
Animacijski okviri predstavljaju ključne trenutke u hodanju, kao što su podizanje noge, prijenos težine i spuštanje noge. Postavljanje animacijskih okvira pomaže u stvaranju osnovnih pokreta i ritma hodanja.

Tijekom animacije hodanja, posebna pažnja se posvetila pokretu stražnjih nogu, kukova i prednjih nogu. To uključuje podizanje noge, prenošenje težine, savijanje koljena i prirodan pokret prednjih noga kako bi se postigao dojam stvarnog hodanja.

Dodavanje detalja kao što su sjena, svjetlo, tekstura i efekti pomaže u poboljšanju animacije hodanja.

Nakon što je animacija hodanja dovršena, proveden je pregled kako bi se osigurala dosljednost, glatkoća i prirodan izgled pokreta.

Na slici 1.5 prikazana su 4 animacijska okvira koja su napravljena za prikaz animacije hodanja.



Slika 1.5 Animacija hodanja

1.3.4 Izrada animacije ozljeđivanja

Animacija ozljeđivanja igra važnu ulogu u računalnim igrama jer pruža likovima osjećaj ranjivosti, fizičkog stresa i interakcije s neprijateljima ili okolinom.

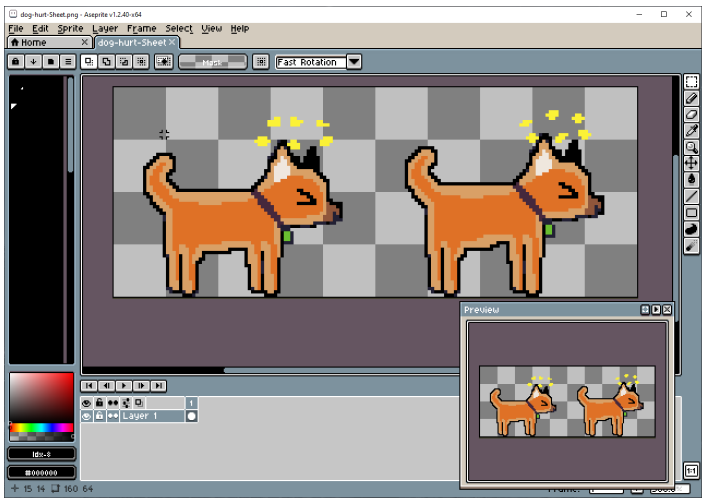
Prije izrade animacije ozljeđivanja, potrebno je definirati kako će lik reagirati na ozljedu. To uključuje razmišljanje o tome kako će izgledati izrazi lica, držanje tijela i pokreti lika kada je ozlijeđen.

Animacijski okviri predstavljaju ključne trenutke u animaciji ozljeđivanja, poput udarca. Postavljanje animacijskih okvira omogućuje stvaranje osnovnih pokreta i prikazuje reakciju lika na ozljedu.

U animaciji ozljeđivanja, važno je prikazati fizički stres koji lik doživljava. To može uključivati izraze lica koji odražavaju bol ili šok te prikaz vrtoglavice koju je lik dobio.

Kada je animacija ozljeđivanja dovršena, proveden je pregled kako bi se osigurala dosljednost, prepoznatljivost i uvjerljivost animacije.

Na slici 1.6 prikazana su 2 animacijska okvira koja prikazuju animaciju ozljeđivanja glavnog lika.



Slika 1.6 Animacija ozljeđivanja

Poglavlje 2

Izrada računalne igre

Za izradu računalne igre korišten je program *Unity*[5]. *Unity* je jedna od najpopularnijih i najmoćnijih razvojnih okolina za izradu računalnih igara.

Program ima intuitivno korisničko sučelje i veliki broj alata koji olakšavaju izradu računalne igre. Vizualni editor za scenu i animacije te podrška za programski jezik C# čine *Unity* pristupačnim čak i za početnike u razvoju igara.

Unity pruža podršku za fiziku, animaciju, zvuk, umjetnu inteligenciju i još mnogo toga, što omogućuje brži razvoj igre bez potrebe za izgradnjom funkcionalnosti od nule.

Unity omogućuje brzo iteriranje tijekom razvojnog procesa. Može se lako testirati igru u stvarnom vremenu, raditi izmjene i pratiti rezultate. To omogućuje brže otkrivanje i ispravljanje problema te brži napredak u razvoju.

Unity također koristi koncept komponenata kako bi omogućio fleksibilnost i modularnost u razvoju igara. Svaki objekt u *Unityju* može imati različite komponente koje definiraju njegovo ponašanje, izgled i interakcije s okolinom. Komponente su modularni dijelovi koji se mogu dodati, ukloniti ili prilagođavati kako bi se postigao željeni rezultat. Komponente mogu biti *Unityjeve* već napravljene ili skripte koje je programer napisao i koje kontroliraju ponašanje određenog objekta u računalnoj igri.

Za izradu računalne igre također je korišten tečaj na *Udemyju*[6] koji korak po korak objašnjava te pojednostavljuje izradu 2D računalne igre.

2.1 Dodavanje tilemap-a i pozadine

Jedan od najvažnijih elemenata u izradi 2D računalne igre je *tilemap* od kojeg se sastoji svaka razina u računalnoj igri. *Tilemap* je koncept u razvoju računalnih igara koji omogućuje organizaciju i prikazivanje svijeta računalne igre kroz raspored kvadratnih ili pravokutnih elemenata poznatih kao "pločice" (engl. *tiles*). Pomoću *tilemapa* se izrađuju različite scene, razine ili pozadine u igri kao kombinacija različitih pločica.

Kada se u scenu u računanlnoj igri doda *tilemap*, treba imati sljedeće komponente: `Tilemap Collider 2D`, `Composite Collider 2D` te `Rigidbody 2D`.

`Tilemap Collider 2D` komponenta generira za svaku pločicu na sceni sudarač tako da objekti na sceni koji imaju gravitaciju ne padnu kroz *tilemap*. Kada se doda ta komponenta, treba se postaviti vrijednost boolean varijable "koristi kompozit" (engl. *used by composite*) na "istinito" tako da svaka pločica nema svoj zasebni sudarač već sve spojene pločice imaju zajednički sudarač. Kada se ne bi postavilo na "istinito", glavni lik bi mogao zapinjati za svaku pločicu, što narušava iskustvo igranja.

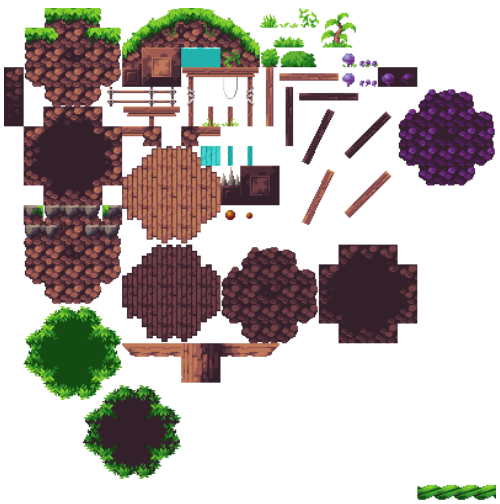
`Composite Collider 2D` komponenta je sudarač koji međusobno djeluje sa 2D sustavom fizike. On treba biti postavljen na *tilemap* kako bi boolean varijabla kod `Tilemap Collider 2D` funkcionirala. Na `Composite Collider 2D` komponenti se također postavlja geometrijski tip na poligone, za glađe pomake glavnog lika na razini.

Kada u *Unityju* dodamo `Composite Collider 2D` komponentu, *Unity* automatski s njom doda i `Rigidbody 2D`. `Rigidbody 2D` komponenta omogućuje ponašanje temeljeno na fizici, kao što su reakcije na gravitaciju, masu, otpor i moment. Na `Rigidbody 2D` komponenti trebamo postaviti tip tijela na kinetički tako da cijela razina nema gravitaciju, kako ne bi padala neprestano.

Tilemap treba dodati u *tile palette* dijelu *Unity* korisničkog sučelja tako da se može koristiti za bilo koju razinu u računalnoj igri.

Pozadina se dodaje u *Unity* povlačenjem iz preglednika datoteka i ispuštanjem na scenu.

Tilemap i pozadina koji su korišteni za potrebe ovog rada preuzeti su sa web poveznice `https://anokolisa.itch.io/basic-140-tiles-grassland-and-mines`. Na slici 2.1 prikazan je *tilemap* korišten u izradi igre. Na slikama 2.2 i 2.3 prikazane su pozadine korištene u izradi igre.



Slika 2.1 Tilemap korišten za izradu razina

Slika 2.2 Prednja pozadina korištena u razinama

Slika 2.3 Stražnja pozadina korištena u razinama

2.2 Izrada logike za glavnog lika

Dodavanje glavnog lika radi se tako da se iz preglednika datoteka povuče prvi animacijski okvir neaktivne animacije i stavi na scenu. Kada se doda objekt glavnog

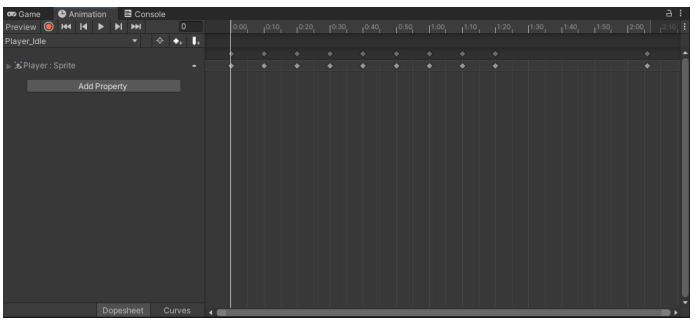
lika na scenu, treba imati sljedeće komponente: `Sprite Renderer`, `Rigidbody 2D`, `Capsule Collider 2D` i `Animator`.

`Sprite Renderer` komponenta služi za okretanje glavnog lika u smjeru u kojem se kreće. Kod `Sprite Renderer` komponente bitno je da objekt glavnog lika bude na *Player* sloju za sortiranje tako da se može vidjeti ispred pozadine i ispred *tilemapa*.

`Rigidbody 2D` komponenta treba biti dodana na objekt glavnog lika kako bi njegovo ponašanje bilo temeljeno na fizici, odnosno kako bi imao prikladne reakcije na gravitaciju, masu, otpor i moment. Na toj komponenti također treba biti postavljena boolean varijabla "Zamrzni rotaciju" (engl. *Freeze rotation*) na "istinito" jer inače kad bi se glavni lik udario u rub razine i slično, rotirao bi se i pao, što nije ugodno iskustvo za igrača koji igra računalnu igru.

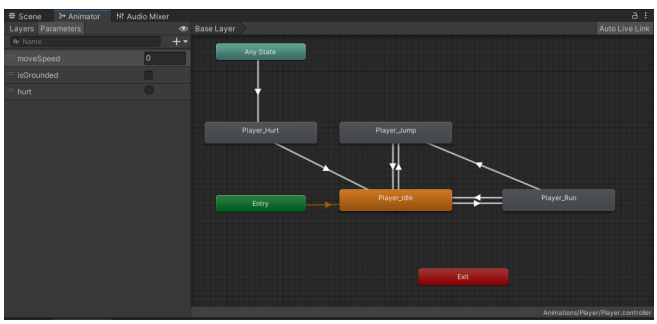
`Capsule Collider 2D` komponenta, isto kao i `Composite Collider 2D`, je sudarač koji međusobno djeluje sa 2D sustavom fizike. Ta komponenta se postavi na glavnog lika tako da se dobiju glađi pomaci glavnog lika po razini. Kada bi se postavila `Box Collider 2D` komponenta umjesto `Capsule Collider 2D` komponente onda bi `Box Collider 2D` sudarač zapinjao za rubove razine.

`Animator` komponenta služi za dodavanje animacija koje su izrađene u prošlom poglavlju ovog rada u računalnu igru. Kada se dodaju animacije na objekte u računalnoj igri koristi se prozor za animiranje unutar *Unityja*. U njemu postavljamo svaki animacijski okvir koji je izrađen za određenu animaciju. Ovisno o tome koliko različitih animacija trebamo, toliko različitih datoteka animacija imamo u pregledniku datoteka. Na slici 2.4 prikazan je primjer dodavanja neaktivne animacije u *Unity*.



Slika 2.4 Dodavanje animacijskih okvira za određenu animaciju

Kada se postave sve animacije koje se žele imati za određeni objekt u računalnoj igri onda se u prozoru za animator postavlja kako će se raditi izmjene između tih animacija. Te izmjene se rade kroz uvjete, u ovom slučaju float varijabla `moveSpeed`, boolean varijabla `isGrounded` i trigger varijabla `hurt`. Te varijable se kontroliraju kroz C# skriptu koja se zove `PlayerController`. Na slici 2.4 prikazan je prozor za animator te postavljene izmjene između animacija.



Slika 2.5 Postavljanje izmjena između različitih animacija

2.2.1 PlayerController

U `PlayerController` C# skripti potrebne su sljedeće varijable za izradu logike kretanja, skakanja i slično za glavnog lika:

```
public static PlayerController instance;

public float moveSpeed, jumpForce;
public Rigidbody2D theRB;
private SpriteRenderer theSR;
private bool isGrounded, canDoubleJump;
public Transform groundCheckPoint;
public LayerMask whatIsGround;
private Animator anim;
public float knockBackLength, knockBackForce;
private float knockBackCounter;
```

```
public float bounceForce;
public bool stopInput;
```

`Public static PlayerController instance` varijabla potrebna je tako da se iz drugih C# skripti može pristupiti određenim metodama i varijablama preko jedne instance objekta glavnog lika. `Instance` varijabla se postavlja u *Unityjevoj* `Awake` metodi koja se koristi za inicijalizaciju objekta prije nego što računalna igra započne ili prije nego što se objekt aktivira.

```
// Awake is used to initialize something before the game starts
private void Awake() {
    instance = this;
}
```

Unity ima još dvije metode koje svaka skripta ima, a to su `Start` i `Update` metoda. `Start` metoda funkcionira slično `Awake` metodi, samo umjesto da se pozove prije nego računalna igra započne, ona se pozove taman prije početka prvog okvira računalne igre. `Update` metoda se poziva svaki okvir. Ako je računalna igra napravljena da bude 30 *fps* (engl. *frames per second*) onda bi se `Update` metoda pozvala 30 puta u sekundi.

U `Start` metodi `PlayerController` skripte postavljamo samo `Animator` i `Sprite Renderer` varijable na vrijednosti komponenta koje objekt ima.

```
// Start is called before the first frame update
void Start() {
    anim = GetComponent<Animator>();
    theSR = GetComponent<SpriteRenderer>();
}
```

U `Update` metodi napravljena je logika za kretanje glavnog lika, za skakanje glavnog lika, kontroliranje u koju stranu je glavni lik okrenut, kontroliranje animatora te kontroliranje koliko dugo će biti ozljeđen i koliko dugo će biti nepobjediv nakon što je udaren.

```
// Update is called once per frame
void Update() {
    // If the pause menu is not open do things as the player
    if (!PauseMenu.instance.isPaused && !stopInput) {
        // If the player didn't get hit the player can walk and jump
        if (knockBackCounter <= 0) {
            // Moving the player
            theRB.velocity = new Vector2(moveSpeed * Input.GetAxis("Horizontal")
                theRB.velocity.y);

            // Checking if the player is touching the ground to know if the
                player can jump
            isGrounded = Physics2D.OverlapCircle(groundCheckPoint.position,
                .2f, whatIsGround);
            if (isGrounded) {
                canDoubleJump = true;
            }

            // Player jumping
            if (Input.GetButtonDown("Jump")) {
                if (isGrounded) {
                    theRB.velocity = new Vector2(theRB.velocity.x, jumpForce);
                    AudioManager.instance.PlaySFX(10);
                } else if (canDoubleJump) {
                    theRB.velocity = new Vector2(theRB.velocity.x, jumpForce);
                    canDoubleJump = false;
                    AudioManager.instance.PlaySFX(10);
                }
            }

            // Controlling which way the player is turned
            if (theRB.velocity.x < 0) {
                theSR.flipX = true;
```

```
            } else if (theRB.velocity.x > 0) {
                theSR.flipX = false;
            }
        // If the player got hit than decrease the knockBackCounter and move
            him away from the danger
        } else {
            knockBackCounter -= Time.deltaTime;
            if (!theSR.flipX) {
                theRB.velocity = new Vector2(-knockBackForce, theRB.velocity.y);
            } else {
                theRB.velocity = new Vector2(knockBackForce, theRB.velocity.y);
            }
        }
    }

    // Controlling the animations on the player
    anim.SetFloat("moveSpeed", Mathf.Abs(theRB.velocity.x));
    anim.SetBool("isGrounded", isGrounded);
}
```

2.3 Izrada šiljaka

Za izradu šiljaka u računalnoj igri potrebno je da šiljci imaju `DamagePlayer` C# skriptu te `Box Collider 2D` komponentu.

`Box Collider 2D` komponenta je potrebna zato da bi se, kada glavni lik skoči ili se na neki drugi način udari na šiljke, aktivirala još jedna *Unityjeva* metoda koja se zove `OnTriggerEnter2D`. Na `Box Collider 2D` komponenti treba biti postavljena boolean varijabla "je okidač" (engl. *is trigger*) na "istinito" tako da, kada se u programskom kodu pozove *Unityjeva* metoda `OnTriggerEnter2D`, *Unity* zna što treba napraviti i provjeriti.

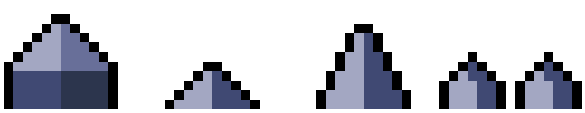
`OnTriggerEnter2D` metoda uzima kao ulazni parametar Collider2D varijablu te se

onda u toj metodi provjerava je li taj drugi *collider* koji je ušao u `Box Collider 2D` označen kao *Player*. Ako je, onda se smanjuje zdravlje glavnog lika, inače se ignorira.

U `DamagePlayer` C# skripti se nalazi ta metoda, koja poziva metodu u `PlayerHealth Controller` C# skripti koja ozljeđuje glavnog lika, tj. smanjuje zdravlje glavnog lika.

```
// Function in Unity for triggering an event
// Deals damage to the player
void OnTriggerEnter2D(Collider2D other) {
    if (other.CompareTag("Player")) {
        PlayerHealthController.instance.DealDamage();
    }
}
```

Na slici 2.6 prikazan je model korišten za šiljke koji je preuzet sa web poveznice `https://totuslotus.itch.io/small-platformer-pack`.



Slika 2.6 Šiljci korišteni u izradi računalne igre

2.4 Izrada sustava za zdravlje glavnog lika

Jedan od važnijih dijelova glavnoga lika je sustav za zdravlje koji kontrolira koliko puta glavni lik treba biti udaren da umre te povećanje njegovog zdravlja.

Potrebne varijable za kontroliranje sustava za zdravlje su:

```
public static PlayerHealthController instance;
public int currentHealth, maxHealth;
public float invincibleLength;
```

```
private float invincibleCounter;
private SpriteRenderer theSR;
public GameObject deathEffect;
```

U `Start` metodi postavlja se trenutno zdravlje glavnog lika na maksimum koji može imati te se postavlja `Sprite Renderer` tako da se može promijeniti boja glavnog lika u prozirnu, kako bi se igraču dao dojam da je glavni lik na kratko nepobjediv.

```
// Start is called before the first frame update
void Start() {
    currentHealth = maxHealth;
    theSR = GetComponent<SpriteRenderer>();
}
```

U `Update` metodi se smanjuje brojač za nepobjedivost dok ne dođe do nule, nakon čega se glavni lik opet može ozlijediti.

```
// Update is called once per frame
void Update() {
    // Controlling the invincibility of the player after he gets hit
    if (invincibleCounter > 0) {
        invincibleCounter -= Time.deltaTime;
        if (invincibleCounter <= 0) {
            theSR.color = new Color(theSR.color.r, theSR.color.g,
                theSR.color.b, 1f);
        }
    }
}
```

U `DealDamage` metodi smanjuje se zdravlje glavnog lika, a ako mu je zdravlje palo ispod nule, onda ga se ponovno postavlja na mjesto na kojem je *checkpoint* ili na početak razine. Ako glavnom liku zdravlje nije palo ispod nule, onda mu se postavlja nepobjedivost i smanjuje jačina boje.

```
// Deal Damage is used for decreasing the players health
public void DealDamage() {
    if (invincibleCounter <= 0) {
        currentHealth--;

        // If the player has no more health make it disappear
        if (currentHealth <= 0) {
            currentHealth = 0;
            Instantiate(deathEffect, transform.position,
                transform.rotation);
            LevelManager.instance.RespawnPlayer();
        // If the player got hit make it invincible for some time
            and call the knock back function
        } else {
            invincibleCounter = invincibleLength;
            theSR.color = new Color(theSR.color.r, theSR.color.g,
                theSR.color.b, .5f);

            PlayerController.instance.KnockBack();
            AudioManager.instance.PlaySFX(9);
        }

        UIController.instance.UpdateHealthDisplay();
        GameObject.FindGameObjectWithTag("Player").
            GetComponent<AgentLevel5>().AddRewardIfEnemyHitPlayer();
    }
}
```

U `HealPlayer` metodi povećava se zdravlje glavnog lika za jedan te se provjerava je li zdravlje prešlo maksimum. Ako jest, zdravlje se vraća na maksimalnu vrijednost koju može imati.

```
// Heal Player is used for increasing the players health
```

```
public void HealPlayer() {
    currentHealth++;
    if (currentHealth > maxHealth) {
        currentHealth = maxHealth;
    }
    UIController.instance.UpdateHealthDisplay();
}
```

2.5 Izrada plohe za ubijanje

Ako glavni lik padne sa razine, potrebna je ploha za ubijanje da ga ubije (engl. *kill plane*). Ta ploha je prazan objekt koji ima samo `Box Collider 2D` koji zauzima cijeli prostor ispod razine. Na tom `Box Collider 2D` bit će označena boolean varijabla "*is trigger*" na "istinito" tako da se unutar C# skripte može okinuti `OnTriggerEnter2D` *Unity* metoda koja će ponovno postaviti glavnog lika na *checkpoint* ili na početak razine.

```
// Function in Unity for triggering an event
// If the player falls of the level respawn him
private void OnTriggerEnter2D(Collider2D other) {
    if (other.CompareTag("Player")) {
        LevelManager.instance.RespawnPlayer();
    }
}
```

2.6 Izrada predmeta za sakupljanje

Glavni lik može dobivati bodove tako da sakuplja novčiće, a zdravlje si može povećati trešnjama. Novčići su preuzeti sa web poveznice `https://totuslotus.itch.io/small-platformer-pack`, a trešnje su preuzete iz *Udemy* tečaja.

Za sakupljanje novčića ili trešanja potrebna je C# skripta u kojoj kontroliramo

njihovo sakupljanje. I novčići i trešnje imaju na sebi `Circle Collider 2D` komponentu koja također na sebi ima označenu boolean varijablu "*is trigger*" na "istinito".

U C# skripti postoji `OnTriggerEnter2D` metoda u kojoj se povećava broj sakupljenih novčića ili zdravlje glavnog lika.

```
// Function in Unity for triggering an event
// Logic for the player picking up a pickup
private void OnTriggerEnter2D(Collider2D other) {
    if (other.CompareTag("Player") && !isCollected) {
        // Logic for picking up a coin collectable
        if (isCoin) {
            LevelManager.instance.coinsCollected++;
            isCollected = true;
            AudioManager.instance.PlaySFX(6);
            Destroy(gameObject);
            Instantiate(pickupEffect, transform.position,
            transform.rotation);
            UIController.instance.UpdateCoinCount();
        }

        // Logic for picking up a health pickup
        if (isHeal) {
            if (PlayerHealthController.instance.currentHealth !=
            PlayerHealthController.instance.maxHealth) {
                PlayerHealthController.instance.HealPlayer();
                isCollected = true;
                AudioManager.instance.PlaySFX(7);
                Destroy(gameObject);
                Instantiate(pickupEffect, transform.position,
                transform.rotation);
            }
        }
    }
```

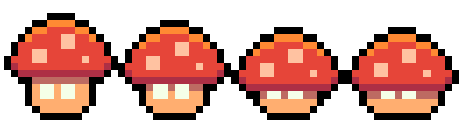
Na slici 2.7 prikazani su model i animacija za novčić, a na slici 2.8 prikazani su model i animacija za trešnju.

Slika 2.7 Novčići korišteni u računalnoj igri

Slika 2.8 Trešnje korištene u računalnoj igri

2.7 Izrada gljiva-neprijatelja

Postavljanje modela i animacija za gljiva-neprijatelja radi se slično kao i postavljanje animacija za glavnog lika. Na slici 2.9 prikazani su ti modeli i animacije koje su korištene u izradi računalne igre. Navedeno je preuzeto sa web poveznice `https://totuslotus.itch.io/small-platformer-pack`.



Slika 2.9 Model i neaktivna animacija gljiva neprijatelja

Za kontroliranje kretanja i stajanja gljiva-neprijatelja potrebna je C# skripta. Potrebne varijable da se postigne navedeno su:

```
public float moveSpeed;
public Transform leftPoint, rightPoint;
```

```
private bool movingRight;
private Rigidbody2D theRB;
public SpriteRenderer theSR;
public float moveTime, waitTime;
private float moveCount, waitCount;
private Animator anim;
```

Gljiva-neprijatelj će se kretati između dvije točke, a nakon što se kreće neko vrijeme, stat će na par sekundi pa se nastaviti kretati. Logika za kretanje je sljedeća:

```
// Update is called once per frame
void Update() {
    // Moving the enemy for a certain period of time
    if (moveCount > 0) {
        moveCount -= Time.deltaTime;

        // Moving right
        if (movingRight) {
            theRB.velocity = new Vector2(moveSpeed, theRB.velocity.y);
            theSR.flipX = true;
            if (transform.position.x > rightPoint.position.x){
                movingRight = false;
            }
        // Moving left
        } else {
            theRB.velocity = new Vector2(-moveSpeed, theRB.velocity.y);
            theSR.flipX = false;
            if (transform.position.x < leftPoint.position.x) {
                movingRight = true;
            }
        }

        // If the time for moving has passed set up the time for waiting
```

```
            if (moveCount <= 0) {
                waitCount = Random.Range(waitTime * .75f, waitTime * 1.25f);
            }

            anim.SetBool("isMoving", true);
        // Enemy waiting and standing still for a certain period of time
        } else if (waitCount > 0) {
            waitCount -= Time.deltaTime;
            theRB.velocity = new Vector2(0f, theRB.velocity.y);

            // If the time for waiting has passed set up the time for moving
            if (waitCount <= 0) {
                moveCount = Random.Range(moveTime * .75f, moveTime * 1.25f);
            }

            anim.SetBool("isMoving", false);
        }
    }
```

2.8 Izrada kutije za ubijanje neprijatelja

Za ubijanje neprijatelja potreban je `Box Collider 2D` koji će se postaviti na objekt koji je dijete glavnog lika. Taj objekt će biti odmah ispod nogu glavnog lika te će također biti postavljena vrijednost boolean varijable "*is trigger*" na "istinito" tako da, kada glavni lik hoda po tlu, noge ne budu iznad tla za veličinu `Box Collidera 2D`. Na tom objektu također je bitno da postoji C# skripta koja će kontrolirati da glavni lik, kada skoči na neprijatelja, ubije tog neprijatelja. Logika koja se nalazi u toj C# skripti je sljedeća:

```
// Function in Unity for triggering an event
// If the player hits the enemy deactivate that enemy and show
death effect
```

```
private void OnTriggerEnter2D(Collider2D other) {
    if (other.CompareTag("Enemy")) {
        other.transform.parent.gameObject.SetActive(false);
        Instantiate(deathEffect, other.transform.position,
        other.transform.rotation);
        PlayerController.instance.Bounce();
        GameObject.FindGameObjectWithTag("Player").
        GetComponent<AgentLevel5>().AddRewardForKillingEnemy();

        // Chance of dropping a collectible and instantiating it
        float dropSelect = Random.Range(0f, 100f);
        if (dropSelect <= chanceToDrop) {
            Instantiate(collectible, other.transform.position,
            other.transform.rotation);
        }

        AudioManager.instance.PlaySFX(3);
    }
}
```

Da neprijatelj može ozljeđivati glavnog lika, postavi se prazan objekt na neprijatelja te se doda `Box Collider 2D` komponenta i `DamagePlayer` C# skripta.

2.9 Postavljanje završetka razine

Da bi igrač znao kada je došao do kraja određene razine korištena je zastavica kroz koju se treba proći da bi se pokrenula iduća razina. Zastavica je preuzeta iz *Udemy* tečaja. Na slici 2.10 prikazana je ta zastavica.

Ta zastavica treba na sebi imati `Box Collider 2D` koji ima boolean varijablu "*is trigger*" postavljenu na "istinito" tako da se okine `OnTriggerEnter2D` metoda kada glavni lik prođe kroz tu zastavicu. U toj metodi se nalazi logika za završavanje trenutne i pokretanje iduće razine.

Slika 2.10 Zastavica korištena u izradi računalne igre

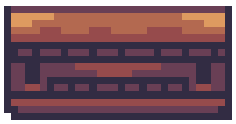
```
// Function in Unity for triggering an event
// Function for ending a level when the player passes the flag pole
private void OnTriggerEnter2D(Collider2D other){
    if (other.CompareTag("Player")){
        LevelManager.instance.EndLevel();
    }
}
```

2.10 Izrada platforme

Platforma u računalnoj igri služi kao još jedno mjesto na koje glavni lik može skočiti. Pri dodavanju platforme u računalnu igru potrebno je da platforma ima `Edge Collider 2D` komponentu. Ta komponenta služi da glavni lik može skočiti na platformu.

Da glavni lik na platformu može skočiti odozdo, potrebna je i `Platform Effector 2D` komponenta. Ta komponenta primjenjuje različita ponašanja platformi kao što su jednosmjerne kolizije, uklanjanje bočnog trenja i slično. Kako bi navedena komponenta djelovala na objekt, treba se na `Edge Collider 2D` komponenti postaviti boolean varijabla "koristi efektor" (engl. *used by effector*) na "istinito".

Na slici 2.11 prikazana je platforma preuzeta iz *Udemy* tečaja.



Slika 2.11 Platforma korištena u izradi računalne igre

2.11 Izrada letećeg šišmiš neprijatelja

Postavljanje modela i animacija za letećeg šišmiš-neprijatelja radi se slično kao i postavljanje animacija za glavnog lika i gljiva-neprijatelja. Na slici 2.12 prikazani su ti modeli i animacije korištene u izradi računalne igre. Navedeno je preuzeto sa web poveznice `https://rentro-ghost.itch.io/bat-sprites`.

Slika 2.12 Leteći šišmiš neprijatelj korišten u izradi računalne igre

Za kontroliranje kretanja i napadanja letećeg šišmiš-neprijatelja potrebna je C# skripta. Potrebne varijable da se postigne navedeno su:

```
public Transform[] points;
public float moveSpeed;
private int currentPoint;

public SpriteRenderer theSR;

public float distanceToAttackPlayer, chaseSpeed, waitAfterAttack;
private Vector3 attackTarget;
private float attackCounter;
private bool stopAttacking;
```

U `Update` metodi šišmiš se normalno kreće sve dok glavni lik nije dovoljno blizu

da ga napadne. Kada je glavni lik dovoljno blizu, onda šišmiš napadne, stoji pored glavnog lika nekoliko sekundi te se nakon toga vrati na desnu točku do koje može ići. Programski kod koji radi navedeno je sljedeći:

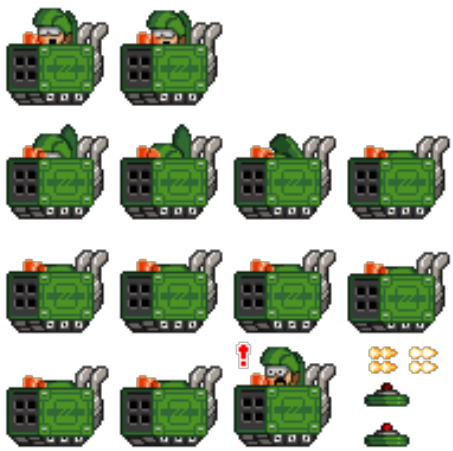
```
// Update is called once per frame
void Update() {
    if (attackCounter > 0) {
        attackCounter -= Time.deltaTime;
        stopAttacking = true;
    } else {
        // If the player isn't close enough for the enemy
            to attack him
        if (Vector3.Distance(transform.position,
        PlayerController.instance.transform.position) >
        distanceToAttackPlayer && !stopAttacking) {
            attackTarget = Vector3.zero;

            // Logic for moving the flying enemy
            transform.position = Vector3.MoveTowards(transform.position,
            points[currentPoint].position, moveSpeed * Time.deltaTime);
            if (Vector3.Distance(transform.position,
            points[currentPoint].position) < .05f) {
                currentPoint++;
                if (currentPoint >= points.Length) {
                    currentPoint = 0;
                }
            }

            if (transform.position.x < points[currentPoint].position.x) {
                theSR.flipX = true;
            } else if (transform.position.x >
            points[currentPoint].position.x) {
                theSR.flipX = false;
```

```
            }
        // If the enemy attacked the player, stop attacking
            after some time
        } else if (stopAttacking) {
            transform.position = Vector3.MoveTowards(transform.position,
            points[1].transform.position, chaseSpeed * Time.deltaTime);

            if (Vector3.Distance(transform.position,
            points[1].transform.position) < .1f) {
                stopAttacking = false;
            }
        // If the player is close enough for the enemy to attack him
        } else {
            if (attackTarget == Vector3.zero) {
                attackTarget =
                PlayerController.instance.transform.position;
            }

            transform.position = Vector3.MoveTowards(transform.position,
            attackTarget, chaseSpeed * Time.deltaTime);

            if (Vector3.Distance(transform.position, attackTarget)
            <= .1f) {
                attackCounter = waitAfterAttack;
                attackTarget = Vector3.zero;
            }
        }
    }
}
```

2.12 Izrada glavnog neprijatelja

Glavni neprijatelj je izrađen za svoju zasebnu razinu u računalnoj igri. Nakon što igrač pobjedi glavnog neprijatelja završava računalna igra.

Glavni neprijatelj je preuzet iz *Udemy* tečaja. Na slici 2.13 prikazan je model korišten za glavnog neprijatelja u računalnoj igri.



Slika 2.13 Modeli i animacije glavnog neprijatelja korištene u računalnoj igri

Dodavanje animacija za glavnog neprijatelja radi se na sličan način kao i ostale animacije u računalnoj igri.

Glavni neprijatelj u stanju "stajanje" može izbacivati projektile iz svojeg tenka. Kada ga glavni lik udari onda se glavni neprijatelj kreće prema drugoj točki te tokom kretanja ispušta mine koje mogu udariti glavnog lika. Glavni neprijatelj može biti udaren pet puta, a kada ga glavni lik udari pet puta, onda umire. Logika za kontroliranje navedenog se nalazi u C# skripti `BossController`. Potrebne varijable za kontroliranje navedenog su:

```
public enum bossStates { shooting, hurt, moving, ended };
public bossStates currentState;

public Transform theBoss;
```

```
public Animator anim;

[Header("Movement")]
public float moveSpeed;
public Transform leftPoint, rightPoint;
public bool moveRight;
public GameObject mine;
public Transform minePoint;
public float timeBetweenMines;
private float mineCounter;

[Header("Shooting")]
public GameObject bullet;
public Transform firePoint;
public float timeBetweenShots;
private float shotCounter;

[Header("Hurt")]
public float hurtTime;
private float hurtCounter;
public GameObject hitbox;

[Header("Health")]
public int health = 5;
public GameObject explosion, winPlatform;
public bool isDefeated;
public float shotSpeedUp, mineSpeedUp;
```

Logika za izbacivanje projektila je sljedeća:

```
// Logic for boss shooting
        case bossStates.shooting :
            shotCounter -= Time.deltaTime;
```

```
            if (shotCounter <= 0) {
                shotCounter = timeBetweenShots;
                var newBullet = Instantiate(bullet, firePoint.position,
                    firePoint.rotation);
                newBullet.transform.localScale = theBoss.localScale;
            }

            break;
```

Logika za ispuštanje mina i kretanje prema drugoj točki je sljedeća:

```
 // Logic for boss moving
        case bossStates.moving :
            if (moveRight) {
                theBoss.position += new Vector3(moveSpeed *
                Time.deltaTime, 0f, 0f);
                if (theBoss.position.x > rightPoint.position.x) {
                    theBoss.localScale = new Vector3(1f, 1f, 1f);
                    moveRight = false;
                    EndMovement();
                }
            } else {
                theBoss.position -= new Vector3(moveSpeed *
                Time.deltaTime, 0f, 0f);
                if (theBoss.position.x < leftPoint.position.x) {
                    theBoss.localScale = new Vector3(-1f, 1f, 1f);
                    moveRight = true;
                    EndMovement();
                }
            }

            mineCounter -= Time.deltaTime;
            if (mineCounter <= 0) {
```

```
                mineCounter = timeBetweenMines;
                Instantiate(mine, minePoint.position, minePoint.rotation);
            }

            break;
```

Logika za kada je glavni neprijatelj ozlijeđen je sljedeća:

```
// Logic fot boss getting hurt
        case bossStates.hurt :
            if (hurtCounter > 0) {
                hurtCounter -= Time.deltaTime;
                if (hurtCounter <= 0) {
                    currentState = bossStates.moving;
                    mineCounter = timeBetweenMines;
                    if (isDefeated) {
                        theBoss.gameObject.SetActive(false);
                        Instantiate(explosion, theBoss.position,
                        theBoss.rotation);
                        winPlatform.SetActive(true);
                        AudioManager.instance.StopBossMusic();
                        currentState = bossStates.ended;
                    }
                }
            }

            break;
```

Za potrebe ozljeđivanja glavnog neprijatelja, glavni neprijatelj treba imati objekt za svoju "kutiju za ozljeđivanje" (engl. *hitbox*). To je dodatni objekt koji na sebi ima `Box Collider 2D` koji ima postavljenu boolean varijablu "*is trigger*" na "istinito". U C# skripti se provjerava je li drugi objekt, koji je udario objekt koji sadrži navedenu skriptu, objekt glavnog lika, u kojem slučaju se smanjuje zdravlje glavnog neprijatelja. Logika koja se nalazi u spomenutoj skripti je sljedeća:

```
// Function in Unity for triggering an event
// If the player is above the boss and hits him boss takes a hit
private void OnTriggerEnter2D(Collider2D other) {
    if (other.CompareTag("Player") &&
    PlayerController.instance.transform.position.y >
    transform.position.y) {
        bossCont.TakeHit();
        PlayerController.instance.Bounce();
        gameObject.SetActive(false);
    }
}
```

Ova metoda poziva metodu u `BossController` C# skripti koja ozljeđuje glavnog neprijatelja te mijenja stanje glavnog neprijatelja u ozlijeđeno stanje. Logika za navedeno je sljedeća:

```
// Function for boss taking a hit
public void TakeHit() {
    currentState = bossStates.hurt;
    hurtCounter = hurtTime;
    anim.SetTrigger("Hit");
    AudioManager.instance.PlaySFX(0);
    GameObject.FindGameObjectWithTag("Player").
    GetComponent<AgentLevel5>().AddRewardForKillingEnemy();

    // Removing all the mines before placing new ones
    BossMine[] mines = FindObjectsOfType<BossMine>();
    if (mines.Length > 0) {
        foreach (BossMine mine in mines) {
            mine.Explode();
        }
    }
```

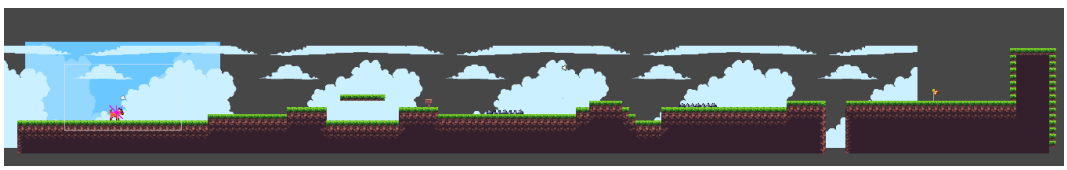
```
        health--;
        if (health <= 0) {
            isDefeated = true;
        } else {
            timeBetweenShots /= shotSpeedUp;
            timeBetweenMines /= mineSpeedUp;
        }
    }
```

Kada glavni lik pobijedi neprijatelja, pojave se platforme na koje glavni lik može skočiti i završiti razinu.

2.13 Razine izrađene u računalnoj igri

2.13.1 Prva razina

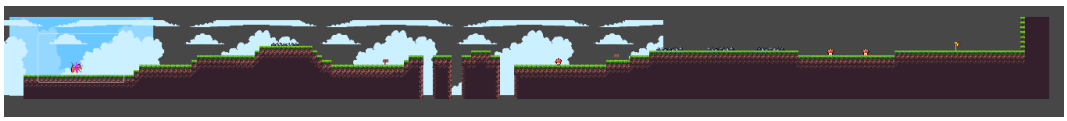
Na prvoj razini postavljeno je par šiljaka na koje se glavni lik može udariti. Postavljeni su načini dobivanja bodova, novčići, te načini za povećanje zdravlja, trešnje. Na slici 2.14 prikazana je prva razina.



Slika 2.14 Prva razina u računalnoj igri

2.13.2 Druga razina

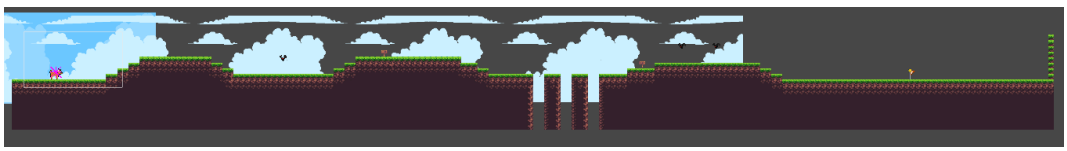
Na drugoj razini dodane su tri gljiva-neprijatelja koje glavni lik mora ubiti. Na slici 2.15 prikazana je druga razina.



Slika 2.15 Druga razina u računalnoj igri

2.13.3 Treća razina

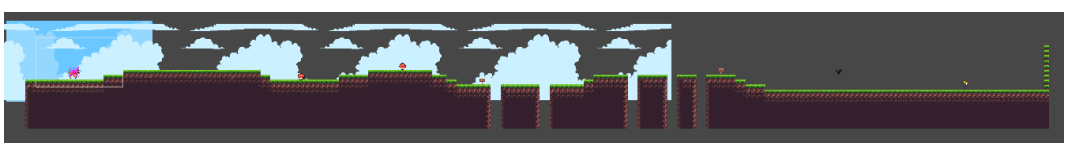
Na trećoj razini dodani su tri leteća šišmiš-neprijatelja koje glavni lik mora ubiti. Na slici 2.16 prikazana je treća razina.



Slika 2.16 Treća razina u računalnoj igri

2.13.4 Četvrta razina

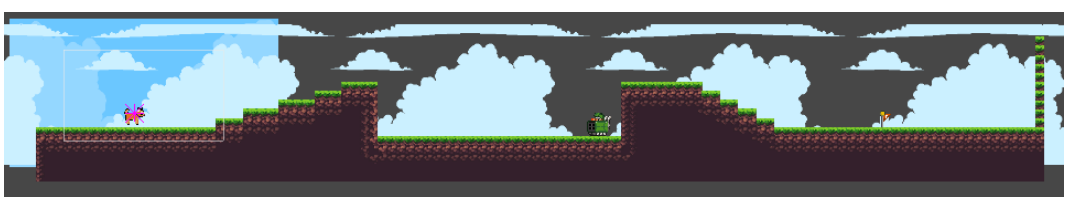
Na četvrtoj razini dodana je kombinacija gljiva-neprijatelja i letećih šišmiš-neprijatelja koje glavni lik mora ubiti. Na slici 2.17 prikazana je četvrta razina.



Slika 2.17 Četvrta razina u računalnoj igri

2.13.5 Peta razina

Na petoj razini dodan je glavni neprijatelj kojeg glavni lik mora ubiti. Na slici 2.18 prikazana je peta razina.



Slika 2.18 Peta razina u računalnoj igri

Poglavlje 3

Izrada samoučećih agenata

Unityjevi agenti za strojno učenje[7] (engl. *Unity machine learning agents*) su snažan alat za izradu računalnih igara i primjenu umjetne inteligencije u računalnim igrama. Oni programerima računalnih igara pružaju algoritme temeljene na *PyTorchu*, kojima jednostavno treniraju inteligentne agente za 2D i 3D računalne igre. Pomoću navedenih *Python* algoritama mogu se provoditi ojačano učenje, učenje imitacijom i razne druge metode korištene u umjetnoj inteligenciji. Navedeni trenirani agenti mogu se koristiti u svakakve svrhe, kao što su kontroliranje likova koje se ne može igrati, automatizirano testiranje verzija računalne igre i evaluacija različitih odluka o dizajnu računalne igre.

3.1 Učenje imitacijom

Ojačano učenje koristi kombinaciju nagrada i kazni da se trenira samoučeći agent, dok učenje imitacijom koristi sistem temeljen na interakciji između agenta učitelja, koji izvodi određeni zadatak, i agenta studenta, koji imitira učitelja. Ovaj način treniranja samoučećih agenata je koristan kada se treniraju kompleksniji zadaci jer će ubrzati učenje. Za potrebe ovog rada korištena je kombinacija učenja imitacijom te ojačanog učenja. Učenje imitacijom je korišteno na početku treniranja da agent nauči kako doći do kraja razine, a kasnije se koristi ojačano učenje da se agent poboljša.

3.2 Instalacija *Unity ML-agents*

Instalacija *Unity ML-agents* radila se preko tutoriala[8]. Prvo je potrebno provjeriti preporučenu verziju *Pythona* za onu verziju *ML-agentsa* koju će se instalirati. Nakon toga se s *Python* web stranice preuzme i instalira potrebna verzija *Pythona*.

Kada je *Python* instaliran, onda se u komandnoj liniji uđe u direktorij gdje se nalazi projekt te se tamo napravi virtualno okruženje za umjetnu inteligenciju. Naredba za izradu virtualnog okruženja je sljedeća:

```
python -m venv venv
```

Za provjeru da je izrada virtualnog okruženja prošla uspješno, aktivira se virtualno okruženje naredbom:

```
venv\Scripts\activate
```

Nakon što je napravljeno i aktivirano virtualno okruženje provjerava se da li je potrebno nadograditi upravitelj paketa za *Python*. Ako je, automatski ga i nadogradi. Naredba korištena za provođenje navedenog je sljedeća:

```
python -m pip install --upgrade pip
```

Za prikaz kvalitete samoučećeg agenta potreban je *PyTorch*. *PyTorch* je knjižnica korištena pri izradi umjetne inteligencije. *PyTorch* se instalira naredbom:

```
pip3 install torch torchvision torchaudio --index-url
    https://download.pytorch.org/whl/cu117
```

Nakon što je sve od navedenog instalirano, treba se još u virtualno okruženje instalirati *ML-agents*. To se radi naredbom:

```
pip install mlagents
```

Može se provjeriti je li se *ML-agents* uspješno instalirao sljedećom naredbom.

```
mlagents-learn --help
```

Ako ova naredba ispiše sve moguće naredbe koje se mogu raditi sa `mlagents-learn`, onda je instalacija prošla.

U *Unity* još treba dodati paket za *ML-agents* tako da se u `window -> package manageru` otvori instalirane pakete i doda *ML-agents* paket.

3.3 Samoučeći agent za prvu razinu

Za svakog agenta korištenog u izradi ovog rada bilo je potrebno izraditi C# skriptu, demo po kojem će agent raditi učenje imitacijom, te konfiguracijsku datoteku.

3.3.1 Izrada AgentLevel1 C# skripte

Potrebna klasa i knjižnice

Svaka skripta treba nasljeđivati klasu `Agent` tako da se može pristupiti metodama koje su napravljene u paketu *ML-agents*. Knjižnice potrebne za navedenu klasu su sljedeće:

```
using Unity.MLAgents;
using Unity.MLAgents.Sensors;
using Unity.MLAgents.Actuators;
```

Knjižnica `Sensors` potrebna je da se s agentom mogu raditi određena zapažanja po kojima onda agent radi određene odluke. Knjižnica `Actuators` potrebna je da na temelju uočenih zapažanja agent napravi određene akcije.

Potrebne varijable

Za prvu razinu potrebne su sljedeće varijable:

```
[SerializeField] private LevelExit levelExit;
[SerializeField] private GameObject[] spikes;
[SerializeField] private Checkpoint[] checkpoints;
```

```
private Rigidbody2D theRB;
private SpriteRenderer theSR;
private Animator anim;
private bool isGrounded, canDoubleJump;
private Vector3 positionOfPlayer;
public LayerMask whatIsGround, whatIsPickup, whatIsEnemy, whatIsSpike,
    whatIsCheckpoint, whatIsLevelEnd, whatIsBullets, whatIsMine;
public GameObject coin, health, hole;
public List<GameObject> spawnedPickups = new List<GameObject>();
private bool passedSpikeOne, passedSpikeTwo, passedHole;
```

Start metoda

Kao i ostale C# skripte izrađene do sada, skripta za agenta također ima `Start` metodu koja postavlja `Rigidbody 2D`, `Sprite Renderer` te `Animator` za kontroliranje tijela i animacije glavnog lika. Također, u `Start` metodi se postavlja vrijednost pozicije na koju se, kada završi epizoda treniranja, postavlja glavni lik, tj. u ovom slučaju samoučeći agent.

```
private void Start() {
    theRB = GetComponent<Rigidbody2D>();
    theSR = GetComponent<SpriteRenderer>();
    anim = GetComponent<Animator>();
    positionOfPlayer = this.transform.position;
}
```

OnEpisodeBegin metoda

Svaka agent C# skripta ima u sebi metodu `OnEpisodeBegin` gdje se na početku svake epizode postavljaju određene vrijednosti. U ovom slučaju na početku svake epizode pozicija glavnog lika se postavlja na početak razine, zdravlje glavnog lika se postavlja na maksimalnu vrijednost koju može imati, broj novčića koje je sakupio glavni lik se postavlja na nulu, deaktiviraju se svi *checkpointovi*, ažuriraju se vrijednosti zdravlja i

novčića sakupljenih na korisničkom sučelju računalne igre te se postavljaju vrijednosti boolean varijabli `passedSpikeOne`, `passedSpikeTwo` i `passedHole` na "neistinito".

```
Debug.Log("On episode begin");
this.transform.position = positionOfPlayer;
PlayerHealthController.instance.currentHealth =
    PlayerHealthController.instance.maxHealth;
LevelManager.instance.coinsCollected = 0;
CheckpointController.instance.DeactivateCheckpoints();
UIController.instance.UpdateCoinCount();
UIController.instance.UpdateHealthDisplay();
passedSpikeOne = false;
passedSpikeTwo = false;
passedHole = false;
```

Za postavljanje novčića koje agent treba sakupiti logika se zakomplicirala. Premda se novčići, kada se sakupe, uklanjaju iz pojedine razine s `Destroy` metodom, onda se na početku svake epizode, čak i ako umre agent ili iz nekog drugog razloga ne dođe do kraja razine, svi novčići trenutno u sceni se moraju ukloniti te ponovno instancirati. To se radi sa sljedećim programskim kodom:

```
// Destroy all the created pickups from the last episode
foreach (GameObject pickupToDestroy in spawnedPickups) {
    Destroy(pickupToDestroy);
}

GameObject pickup;

// Clear the list of pickups created
spawnedPickups.Clear();
Debug.Log("Destroyed all pickups");

// Instantiate pickups
pickup = Instantiate(coin, new Vector3(34.5f, 2f, 0f),
```

```
        new Quaternion(0f, 0f, 0f, 0f));
    spawnedPickups.Add(pickup);
    pickup = Instantiate(coin, new Vector3(36.5f, 2f, 0f),
        new Quaternion(0f, 0f, 0f, 0f));
    spawnedPickups.Add(pickup);
    pickup = Instantiate(coin, new Vector3(38.5f, 2f, 0f),
        new Quaternion(0f, 0f, 0f, 0f));
    spawnedPickups.Add(pickup);
    pickup = Instantiate(coin, new Vector3(50f, -1f, 0f),
        new Quaternion(0f, 0f, 0f, 0f));
    spawnedPickups.Add(pickup);
    pickup = Instantiate(coin, new Vector3(54f, -1f, 0f),
        new Quaternion(0f, 0f, 0f, 0f));
    spawnedPickups.Add(pickup);
    pickup = Instantiate(coin, new Vector3(62.5f, -1f, 0f),
        new Quaternion(0f, 0f, 0f, 0f));
    spawnedPickups.Add(pickup);
    pickup = Instantiate(coin, new Vector3(66.5f, -1f, 0f),
        new Quaternion(0f, 0f, 0f, 0f));
    spawnedPickups.Add(pickup);
    pickup = Instantiate(coin, new Vector3(92f, 0f, 0f),
        new Quaternion(0f, 0f, 0f, 0f));
    spawnedPickups.Add(pickup);
    pickup = Instantiate(coin, new Vector3(94.5f, 0f, 0f),
        new Quaternion(0f, 0f, 0f, 0f));
    spawnedPickups.Add(pickup);
    pickup = Instantiate(coin, new Vector3(97f, 0f, 0f),
        new Quaternion(0f, 0f, 0f, 0f));
    spawnedPickups.Add(pickup);
    pickup = Instantiate(coin, new Vector3(99.5f, 0f, 0f),
        new Quaternion(0f, 0f, 0f, 0f));
    spawnedPickups.Add(pickup);
```

```
pickup = Instantiate(health, new Vector3(73.6f, 1.2f, 0f),
    new Quaternion(0f, 0f, 0f, 0f));
spawnedPickups.Add(pickup);
```

Ova metoda je slična u svim agent C# skriptama. Izmjene su se uglavnom odnosile na promjenu pozicije instanciranja novčića i trešanja. Ostale promjene će biti navedene u poglavljima vezana za određenu razinu.

CollectObservations metoda

Druga metoda koju svaka agent C# skripta ima u sebi je `CollectObservations` metoda. Ova metoda služi tome da agent napravi određena zapažanja po kojima će odlučivati koje akcije će napraviti u određenom trenutku. Zapažanja koja su određena u prvoj razini su ista zapažanja koja će agent raditi i na ostalim razinama u računalnoj igri.

Najvažnija zapažanja koja agent radi su zapažanja što se nalazi u određenim smjerovima oko njega. To se radi tako da se postave `layerMask` varijable kao što su `whatIsGround`, koji agentu određuje što je oko njega tlo po kojem može hodati, `whatIsPickup`, koji agentu određuje što je oko njega predmet za sakupljanje, `whatIsEnemy`, koji agentu određuje što je oko njega neprijatelj kojeg treba ubiti i slično.

Također mu se za zapažanja postavljaju njegova pozicija te pozicija zastavice za završetak razine, količina zdravlja kojeg trenutno ima, postotak sakupljenih novčića, njegova udaljenost od zastavice te brzina kojom ide.

```
// On each frame collect observations before making a decision of an action
public override void CollectObservations(VectorSensor sensor) {
    // Making observations using raycasting for how close some things are
    MakeObservations(whatIsGround, sensor);
    MakeObservations(whatIsCheckpoint, sensor);
    MakeObservations(whatIsLevelEnd, sensor);
    MakeObservations(whatIsPickup, sensor);
    MakeObservations(whatIsSpike, sensor);
```

```
        MakeObservations(whatIsEnemy, sensor);
        MakeObservations(whatIsBullets, sensor);
        MakeObservations(whatIsMine, sensor);

        // Add observations for players location and level exit location
        sensor.AddObservation(transform.position);
        sensor.AddObservation(levelExit.transform.position);

        // Add observation for how much health the player has
        sensor.AddObservation(PlayerHealthController.instance.currentHealth);
        // Add observation for how many coins the player has collected
        sensor.AddObservation(LevelManager.instance.coinsCollected / 11);

        sensor.AddObservation(Vector3.Distance(transform.position,
            levelExit.transform.position));

        sensor.AddObservation(theRB.velocity);
    }

    // Make observations with raycasting
    private void MakeObservations(LayerMask layerMask, VectorSensor sensor) {
        sensor.AddObservation(Physics2D.Raycast(new Vector2(transform.position.x
            transform.position.y), Vector2.down, 40f, layerMask));
        sensor.AddObservation(Physics2D.Raycast(new Vector2(transform.position.x
            transform.position.y), Vector2.up, 40f, layerMask));
        sensor.AddObservation(Physics2D.Raycast(new Vector2(transform.position.x
            transform.position.y), Vector2.left, 40f, layerMask));
        sensor.AddObservation(Physics2D.Raycast(new Vector2(transform.position.x
            transform.position.y), Vector2.right, 40f, layerMask));
        sensor.AddObservation(Physics2D.Raycast(new Vector2(transform.position.x
            transform.position.y), Vector2.down + Vector2.left, 40f, layerMask))
        sensor.AddObservation(Physics2D.Raycast(new Vector2(transform.position.x
            transform.position.y), Vector2.down + Vector2.right, 40f, layerMask)
```

```
        sensor.AddObservation(Physics2D.Raycast(new Vector2(transform.position.x
            transform.position.y), Vector2.up + Vector2.left, 40f, layerMask));
        sensor.AddObservation(Physics2D.Raycast(new Vector2(transform.position.x
            transform.position.y), Vector2.up + Vector2.right, 40f, layerMask));
    }
```

OnActionReceived metoda

ML-agents daje pristup dva tipa akcija koje agent može raditi. To su diskretne (engl. *discrete*) i stalne (engl. *continuous*) akcije. Diskretne akcije su cijeli brojevi od 0 na dalje, dok su stalne akcije decimalni brojevi u rasponu od -1 do 1. U izradi ovog rada korištena su oba tipa akcija. Diskretne akcije su korištene za skakanje agenta: 0 je da ne skoči, dok je 1 da skoči. Stalne akcije su korištene za kretanje agenta lijevo i desno.

U metodi `OnActionReceived` agent radi akcije prema određenim pravilima. Programski kod koji se nalazi u toj metodi sličan je programskom kodu korištenom za kretanje glavnog lika kada igrač igra računalnu igru. Programski kod je sljedeći:

```
    // On action received make an action like move or jump
    public override void OnActionReceived(ActionBuffers actions){
        float moveX = actions.ContinuousActions[0];
        int jump = actions.DiscreteActions[0];

        float moveSpeed = PlayerController.instance.moveSpeed;
        theRB.velocity = new Vector2(moveX * moveSpeed, theRB.velocity.y);

        isGrounded = Physics2D.OverlapCircle(PlayerController.instance.
            groundCheckPoint.position, .2f, whatIsGround);
        if (isGrounded) {
            canDoubleJump = true;
        }

        if (jump == 1) {
```

```
            if (isGrounded) {
                theRB.velocity = new Vector2(theRB.velocity.x,
                    PlayerController.instance.jumpForce);
            } else if (canDoubleJump) {
                theRB.velocity = new Vector2(theRB.velocity.x,
                    PlayerController.instance.jumpForce);
                canDoubleJump = false;
            }
        }

        if (theRB.velocity.x < 0) {
            theSR.flipX = true;
        } else if (theRB.velocity.x > 0) {
            theSR.flipX = false;
        }

        anim.SetFloat("moveSpeed", Mathf.Abs(theRB.velocity.x));
        anim.SetBool("isGrounded", isGrounded);

        AddReward(-1f / MaxStep);

        if (transform.position.x > spikes[0].transform.position.x &&
            !passedSpikeOne) {
            AddReward(+2f);
            passedSpikeOne = true;
        }
        if (transform.position.x > spikes[1].transform.position.x &&
            !passedSpikeTwo) {
            AddReward(+2f);
            passedSpikeTwo = true;
        }
        if (transform.position.x > hole.transform.position.x && !passedHole) {
            AddReward(+2f);
```

```
            passedHole = true;
        }
    }
```

U ovoj metodi, premda ona funkcionira slično kao `Update` metoda u drugim C# skriptama, gleda se da agent, kada prođe šiljke ili preskoči rupu, dobije određenu nagradu kako bi znao da je napravio željenu akciju. Također se u ovoj metodi nalazi i negativna nagrada za broj koraka tako da agent zna da treba u što manjem broju koraka doći do kraja razine.

Ova metoda je slična u svim C# skriptama za agenta, jedine potrebne izmjene su izmjene za broj šiljaka koje treba preskočiti te broj rupa na razini.

Heuristic metoda

Kod učenja imitacijom, metoda potrebna za izradu primjera kako završiti određenu razinu je `Heuristic` metoda, koja po određenim ulaznim podacima određuje kakve akcije su korištene, a agent po tome uči. Programski kod za navedeno je sljedeći:

```
// Heuristic actions for recording demonstrations for imitation learning
public override void Heuristic(in ActionBuffers actionsOut) {
    ActionSegment<float> continuousActions = actionsOut.ContinuousActions;
    ActionSegment<int> discreteActions = actionsOut.DiscreteActions;

    continuousActions[0] = Input.GetAxis("Horizontal");
    discreteActions[0] = Input.GetButtonDown("Jump") ? 1 : 0;
}
```

Ova metoda je identična u svim agent C# skriptama.

OnTriggerEnter2D metoda

U `OnTriggerEnter2D` metodi provjeravamo ako je agent udario nešto od sljedećeg: plohu za ubijanje, zastavicu za završetak razine, novčić, trešnju ili šiljke. Ovisno o tome što je udario dobiti će pozitivnu ili negativnu nagradu.

Kada agent udari plohu za ubijanje agentu se daje -10 bodova te završava epizoda i započinje nova.

```
// Player fell off the level
if (other.gameObject.name.Equals("Kill Plane")) {
    Debug.Log("Player died");
    AddReward(-10f);
    EndEpisode();
```

Kada agent udari zastavicu za završetak razine, agent dobiva +50 bodova, a ovisno o tome je li sakupio sve novčiće ili nije, dodatno dobiva ili gubi određenu količinu bodova. Također, slična logika se primjenjuje i ako je agent završio razinu sa maksimalnom količinom zdravlja ili nije. Kada agent prođe kroz zastavicu završava epizoda i započinje nova.

```
// Player got to the end of the level
} else if (other.gameObject.name.Equals("Level End")) {
    Debug.Log("Got to the end");
    AddReward(+50f);

    // If the player collected all the coins or not add a positive or
        negative reward
    if (LevelManager.instance.coinsCollected < 11) {
        AddReward(-10f + (LevelManager.instance.coinsCollected / 11)
            * 10);
    } else if (LevelManager.instance.coinsCollected >= 11) {
        AddReward(+10f);
    }

    // If the player has max health or not add a positive or negative
        reward
    if (PlayerHealthController.instance.currentHealth !=
        PlayerHealthController.instance.maxHealth) {
        switch (PlayerHealthController.instance.currentHealth) {
```

```
                case 5: AddReward(-1f); break;
                case 4: AddReward(-2f); break;
                case 3: AddReward(-3f); break;
                case 2: AddReward(-4f); break;
                case 1: AddReward(-5f); break;
            }
        } else {
            AddReward(+10f);
        }

        EndEpisode();
```

Kada agent sakupi novčić, agent dobiva +5 bodova.

```
    // If the player collected a coin add a reward
    } else if (other.gameObject.name.Contains("Coin")) {
        spawnedPickups.Remove(other.gameObject);
        Debug.Log("Removed a pickup from list");
        Debug.Log("Got a coin");
        AddReward(+5f);
```

Kada agent sakupi trešnju, agent dobiva +1 bod jer se želi izbjeći da agent svaki put želi skupiti trešnju radi dolaženja do kraja razine sa maksimalnim zdravljem.

```
    // If the player collected a health pickup add a reward
    } else if (other.gameObject.name.Contains("Health Pickup")) {
        if (PlayerHealthController.instance.currentHealth <
            PlayerHealthController.instance.maxHealth) {
            spawnedPickups.Remove(other.gameObject);
            Debug.Log("Removed a pickup from list");
            Debug.Log("Got healed");
            AddReward(+1f);
        }
```

Ako se agent udari na šiljke, agent dobiva -5 bodova.

```
// If the player hit spikes add a negative reward
} else if (other.gameObject.name.Contains("Spikes")) {
    Debug.Log("Ran into spikes");
    AddReward(-5f);
}
```

Također na kraju ove metode se radi provjera ako agent više nema zdravlja, u kojem slučaju ga se ubije, epizoda završava i započinje nova.

```
// If the player has no more health end the episode and add a negative
    reward
if (PlayerHealthController.instance.currentHealth <= 0) {
    Debug.Log("Player died because no more health");
    AddReward(-10f);
    EndEpisode();
}
```

3.3.2 Dodavanje samoučećeg agenta na objekt glavnog lika

Kada se na objekt glavnog lika doda skripta koja nasljeđuje `Agent` klasu, *Unity* automatski doda i komponentu "Parametri ponašanja" (engl. *behaviour parameters*). Na toj komponenti trebaju se postaviti sljedeće vrijednosti: `Behaviour name`, `Vector Observations - space size`, `Actions - Continuous branches`, `Actions - Discrete branches`, `Actions - Discrete branches - Branch 0 size` i `Behaviour type`.

- `Behaviour name` je ime trenutnog ponašanja koje samoučeći agent uči te će pod tim imenom biti spremljen taj model u direktoriju. Za prvu razinu ova vrijednost je postavljena na "Level 1 Imitation".
- `Vector Observations - space size` je broj zapažanja koje će samoučeći agent raditi. Za prvu i svaku drugu razinu ova vrijednost je postavljena na "75".
- `Actions - Continuous branches` je broj stalnih akcija koje samoučeći agent može raditi. Za prvu i svaku drugu razinu ova vrijednost je postavljena na "1".

- `Actions - Discrete branches` je broj diskretnih akcija koje samoučeći agent može raditi. Za prvu i svaku drugu razinu ova vrijednost je postavljena na "1".
- `Action - Discrete branches - Branch 0 size` je broj mogućih vrijednosti koje će imati diskretna akcija. Za prvu i svaku drugu razinu ova vrijednost je postavljena na "2" jer samoučeći agent može odlučiti da će ili skočiti ili ne.
- `Behaviour type` je tip ponašanja koje se trenutno želi koristiti. Kada se radi demonstracija što agent treba raditi na određenoj razini, ova vrijednost je postavljena na "Heuristic", a kada agent uči imitirati demonstraciju, onda je ova vrijednost postavljena na "Default".

Na objekt glavnog lika također se treba postaviti komponenta "Traženje odluka" (engl. *Decision requester*). Bez navedene komponente agent ne bi mogao raditi odluke te ne bi mogao završiti određenu razinu. Tokom treniranja samoučećeg agenta vrijednost "perioda odluke" (engl. *Decision period*) se može povećati da se ubrza treniranje.

Zadnja komponenta koju se treba dodati je "demonstracijski snimač" (engl. *Demonstration recorder*) koji služi da se snima demonstracija po kojoj će samoučeći agent učiti. Kada se želi snimati demonstracija, vrijednost boolean varijable `Record` je postavljena na "istinito", a kada se trenira samoučeći agent, onda je postavljena na "neistinito".

3.3.3 Snimanje demonstracije i početak treniranja

Prije nego se krene snimati demonstracija, trebaju biti postavljene vrijednosti za još nekoliko varijabli u komponenti `Demonstration recorder`. Za prvu razinu vrijednost varijable `Demonstration name` postavljena je na "Level1Demo". Demonstracija će biti spremljena pod navedenim imenom u direktorij koji se odredi u drugoj varijabli. Ta varijabla se zove `Demonstration Directory`, a vrijednost te varijable je postavljena na "AIDemo".

U demonstraciji je napravljena trideset i jedna epizoda gdje je igrač došao do kraja razine sa najboljim mogućim rezultatima.

Kada je demonstracija snimljena može se početi treniranje metodom učenja imi-

tacijom. Da bi učenje imitacijom bilo omogućeno treba se u konfiguracijsku datoteku postaviti određene vrijednosti. Za pomoć pri postavljanju konfiguracijske datoteke koristio se članak[9] koji navedeno opisuje.

Signali nagrađivanja (engl. Reward signals)

Dvije vrste signala nagrađivanja korištene su za potrebe ovog rada: *extrinsic* i *GAIL*.

Extrinsic nagrade temelje se na nagradama dobivenima u okolini. Pod tim nagradama se misli na sve nagrade definirane u C# skripti za samoučećeg agenta.

GAIL (engl. *Generative Adversarial Imitation Learning*) nagrade temelje se na nagradama dobivenima prema tome koliko dobro samoučeći agent oponaša demonstraciju koju je dobio.

U konfiguracijskoj datoteci postavljaju se vrijednosti za *extrinsic* nagrade koje su `strength` i `gamma` vrijednosti. `Strength` vrijednost određuje koliku težinu se želi pridodati *extrinsic* nagradama. Na početku treniranja ova vrijednost je "0.60". Tijekom treniranja ova vrijednost će se povećavati sve dok ne dođe do vrijednosti "1.0". `Gamma` vrijednost je vrijednost faktora popusta za nagrade koje će doći iz okoline u budućnosti. Tokom treniranja ova vrijednost je "0.99".

Za *GAIL* nagrade, u konfiguracijskoj datoteci, trebaju se postaviti vrijednosti za `strength` i `demo_path`. `Strength` vrijednost, slično kao i kod *extrinsic* nagrada, određuje koliku težinu se želi pridodati *GAIL* nagradama. Na početku treniranja ova vrijednost je "0.50". Tijekom treniranja ova vrijednost je smanjivana sve dok nije došla do vrijednosti "0.0". `Demo_path` vrijednost je vrijednost gdje se nalazi demonstracija po kojoj će se samoučeći agent učiti.

Kloniranje ponašanja (engl. *Behavioural cloning*)

Kloniranje ponašanja služi tome da samoučeći agent točno oponaša sve akcije pružene u demonstraciji. Vrijednosti koje su postavljene za korištenje kloniranja ponašanja su `strength` i `demo_path`. Sa `strength` vrijednosti može se kontrolirati koliko će samoučeći agent oponašati akcije iz demonstracije. Na početku treniranja ova vrijednost je postavljena na "0.50". Tijekom treniranja ova vrijednost je smanjena sve dok nije

došla do "0.0". `Demo_path` vrijednost je vrijednost gdje se nalazi demonstracija po kojoj će se samoučeći agent učiti.

Početak treniranja

Treniranje se pokreće naredbom:

```
mlagents-learn --run-id AIImitation Configuration\imitation1.yaml
```

gdje je `--run-id AIImitation` ime direktorija u kojem će se, kada se zaustavi treniranje, spremiti model, a `Configuration\imitation1.yaml` konfiguracijska datoteka za učenje imitacijom.

Kada se zaustavi treniranje i želi iznova započeti s treniranjem od točke na kojoj je stalo, koristi se sljedeća naredba:

```
mlagents-learn --run-id AIImitation results\AIImitation\configuration
.yaml --resume
```

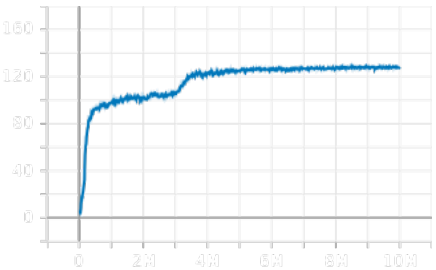
Koja konfiguracijska datoteka će se koristiti je i dalje specificirana zato što se postepeno mijenjaju vrijednosti za jačinu određenih signala nagrađivanja te kloniranja ponašanja.

3.3.4 Rezultati treniranja prve razine

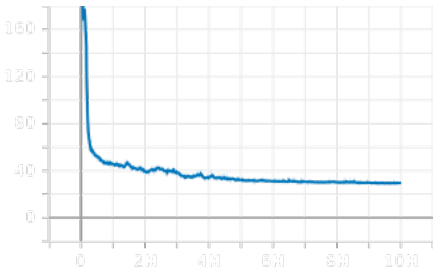
Samoučeći agent je uspješno naučio kako doći do kraja prve razine. Problemi koje još ima su da pokušava sakupiti trešnju čak i ako mu je zdravlje na maksimalnoj vrijednosti te se nekad "boji" preskočiti rupu, no i dalje uspije i dođe do zastavice za završetak razine.

U `tensorboardu` mogu se vidjeti grafovi koji prikazuju različita poboljšanja samoučećeg agenta. Dva najvažnija, gledana tijekom treniranja, su kumulativna nagrada i dužina epizoda. Graf kumulativne nagrade bi se tijekom treniranja trebao povećavati, sve dok se ne stabilizira na određenoj vrijednosti, kada je završeno treniranje.

Graf dužine epizode bi se tijekom treniranja trebao smanjivati, sve dok se ne stabilizira na određenoj vrijednosti, kada je završeno treniranje. Navedeni grafovi prikazani su na slikama 3.1 i 3.2.



Slika 3.1 Graf kumulativne nagrade za samoučećeg agenta na prvoj razini



Slika 3.2 Graf dužine epizode za samoučećeg agenta na prvoj razini

Kada bi se treniranje pustilo da traje još duže, kada bi se poboljšala zapažanja, kada bi se poboljšao sistem nagrade i kada bi se promijenila konfiguracijska datoteka, moguće je da bi se dobilo još bolje rezultate samoučećeg agenta.

3.4 Samoučeći agent za drugu razinu

3.4.1 Izrada AgentLevel2 C# skripte

Promjene za adaptiranje skripte za agenta za drugu razinu su napravljene u: `OnEpisodeBegin` metodi i `OnActionReceived` metodi te su dodane metode za davanje pozitivnih nagrada samoučećem agentu ako ubije neprijatelja i davanje negativnih nagrada ako neprijatelj udari agenta.

OnEpisodeBegin metoda

U `OnEpisodeBegin` metodi promijenio se broj boolean varijabli za agentovo preskakanje šiljaka ili rupa jer njih ima više na drugoj razini. Promijenjeno je gdje i koliko novčića i trešanja će se instancirati na početku svake epizode. Također dodano je da se na početku svake epizode svi objekti neprijatelja postave na aktivne.

```
foreach (GameObject enemy in enemies) {
    enemy.SetActive(true);
}
```

OnActionReceived metoda

U `OnActionReceived` metodi promijenila se samo količina *if* grananja za kada samoučeći agent preskoči rupe ili šiljke.

AddRewardForKillingEnemy metoda

`AddRewardForKillingEnemy` metoda poziva se iz `Stompbox` C# skripte, kada agent ubije neprijatelja, te ta metoda dodaje pozitivnu nagradu agentu.

```
public void AddRewardForKillingEnemy() {
    Debug.Log("Killed an enemy");
    AddReward(+10f);
}
```

AddRewardIfEnemyHitPlayer metoda

`AddRewardIfEnemyHitPlayer` metoda poziva se iz `PlayerHealthController` C# skripte, kada neprijatelj udari agenta, te ta metoda dodaje negativnu nagradu agentu.

```
public void AddRewardIfEnemyHitPlayer() {
    Debug.Log("Got hit by enemy");
    AddReward(-5f);
}
```

3.4.2 Dodavanje samoučećeg agenta na objekt glavnog lika

Na isti način na koji se dodala C# skripta za samoučećeg agenta na prvu razinu, tako se dodaje i na drugu razinu.

Izmijeniti treba vrijednost varijable `Behaviour name` u `Behaviour parameters` na "Level 2 Imitation", te vrijednost varijable `Demonstration name` u `Demonstration recorder` na "Level2Demo".

Snimanje demonstracije, konfiguracijska datoteka i početak treniranja rade se na isti način kao i za prvu razinu.

3.4.3 Rezultati treniranja druge razine

Samoučeći agent je uspješno naučio kako doći do kraja druge razine. Problemi koje još ima su da nije naučio da treba dvaput skočiti da preskoči šiljke te se i dalje nekada udari na neprijatelja, no uspješno dolazi do kraja razine.

Graf kumulativne nagrade i graf dužine epizode prikazani su na slikama 3.3 i 3.4.

Slika 3.3 Graf kumulativne nagrade za samoučećeg agenta na drugoj razini

Slika 3.4 Graf dužine epizode za samoučećeg agenta na drugoj razini

U grafu dužine epizode na kraju naglo poraste dužina epizode jer je vrijednost `Decision Perioda` slučajno ostavljena na vrijednosti ispod najbrže.

Isto kao i za samoučećeg agenta na prvoj razini, kada bi se poboljšala zapažanja, kada bi se produžilo treniranje, kada bi se poboljšao sistem nagrade i kada bi se promijenila konfiguracijska datoteka, moguće je da bi se dobilo još bolje rezultate samoučećeg agenta.

3.5 Samoučeći agent za treću razinu

3.5.1 Izrada AgentLevel3 C# skripte

Promjene koje su bile napravljene da se adaptira skripta za agenta za treću razinu se nalaze u `OnEpisodeBegin` metodi i u `OnActionReceived` metodi.

OnEpisodeBegin metoda

U `OnEpisodeBegin` metodi promjene su slične kao i promjene za drugu razinu. To su promjene: broj boolean varijabli koje se trebaju postaviti na "neistinito" i gdje i koliko se novčića i trešanja instancira na početku svake epizode.

OnActionReceived metoda

U `OnActionReceived` metodi napravljena je ista izmjena kao i za drugu razinu, a to je količina *if* grananja za kada samoučeći agent preskoči rupe ili šiljke.

3.5.2 Dodavanje samoučećeg agenta na objekt glavnog lika

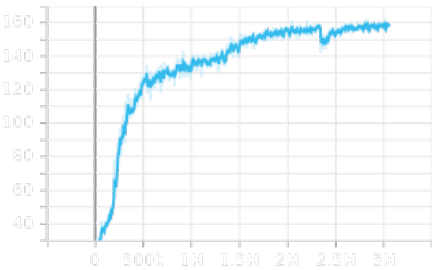
Isto kao i u prethodnom potpoglavlju, izmjene koje se trebaju napraviti su imenovanje ponašanja i imenovanje demonstracije. Vrijednost za ponašanje je "Level 3 Imitation", dok je vrijednost za demontraciju "Level3Demo".

Snimanje demostracije, konfiguracijska datoteka i početak treniranja rade se na isti način kao i za prve dvije razine.

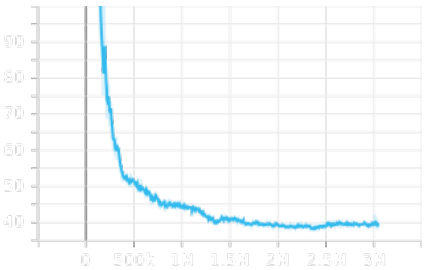
3.5.3 Rezultati treniranja treće razine

Samoučeći agent je uspješno naučio kako doći do kraja treće razine. Problemi koje još ima je da nekada ne ubije sve neprijatelje postavljene na razini, već samo prođe pored njih i završi razinu prolaskom kroz zastavicu.

Graf kumulativne nagrade i graf dužine epizode prikazani su na slikama 3.5 i 3.6.



Slika 3.5 Graf kumulativne nagrade za samoučećeg agenta na trećoj razini



Slika 3.6 Graf dužine epizode za samoučećeg agenta na trećoj razini

Moguće da bi se dobilo bolje rezultate kada bi se duže trenirao samoučeći agent, kada bi se poboljšao sistem nagrade, kada bi se poboljšala zapažanja te kada bi se promijenila konfiguracijska datoteka.

3.6 Samoučeći agent za četvrtu razinu

3.6.1 Izrada AgentLevel4 C# skripte

Promjene koje su bile napravljene da se adaptira skripta za agenta za četvrtu razinu su na istim mjestima kao i promjene koje su se radile za drugu i treću razinu.

U `OnEpisodeBegin` metodi potrebne promjene su izmjena broja boolean varijabli koje se trebaju postaviti na "neistinito" te izmjena gdje i koliko se novčića i trešanja treba instancirati na početku svake epizode.

U `OnActionReceived` metodi izmijenjena je količina *if* grananja za kada samoučeći agent preskoči rupe ili šiljke.

3.6.2 Dodavanje samoučećeg agenta na objekt glavnog lika

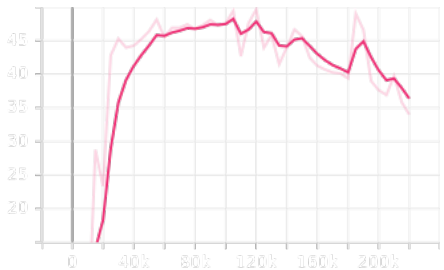
Isto kao i u prethodna dva potpoglavlja izmjene koje se trebaju napraviti su imenovanje ponašanja te imenovanje demonstracije. Vrijednost za ponašanje je "Level 4 Imitation", dok je vrijednost za demonstraciju "Level4Demo".

Snimanje demonstracije, konfiguracijska datoteka i početak treniranja rade se na isti način kao i za prve tri razine.

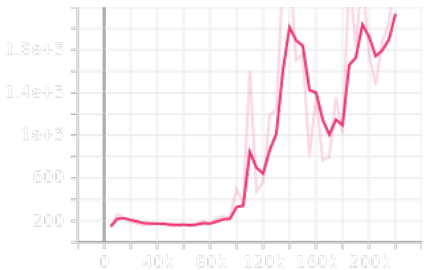
3.6.3 Rezultati treniranja četvrte razine

Samoučeći agent nije uspješno naučio kako doći do kraja četvrte razine. Problemi koje ima su da ne može "shvatiti" kako da ubije neprijatelja te kako da ne bude pogođen od strane neprijatelja. Moguće da kada bi se promijenio sistem nagrade, količina zapažanja i/ili konfiguracijska datoteka samoučeći agent bi bolje razumio što treba napraviti. Trenutno samoučeći agent dobro uči kada radi kloniranje ponašanja iz demonstracije, no čim se počne prebacivati sa učenja imitacijom na ojačano učenje, samoučeći agent se počne "bojati" dobivanja negativnih nagrada pa ne pokušava više dobiti ni pozitivne.

Graf kumulativne nagrade i graf dužine epizode prikazani su na slikama 3.7 i 3.8.



Slika 3.7 Graf kumulativne nagrade za samoučećeg agenta na četvrtoj razini



Slika 3.8 Graf dužine epizode za samoučećeg agenta na četvrtoj razini

3.7 Samoučeći agent za petu razinu

3.7.1 Izrada AgentLevel5 C# skripte

Promjene koje su napravljene da se adaptira skripta za agenta za petu razinu su u sljedećim metodama: `Start`, `OnEpisodeBegin` i `OnActionReceived`.

Start metoda

U `Start` metodi potrebno je zapamtiti početnu poziciju glavnog neprijatelja tako da se u `OnEpisodeBegin` metodi može glavnog neprijatelja svaki put postaviti na isto mjesto.

```
positionOfTank = theTank.transform.position;
```

OnEpisodeBegin metoda

U `OnEpisodeBegin` metodi aktivira se borba protiv glavnog neprijatelja te se ponovno postavljaju vrijednosti njegovog zdravlja, njegovo trenutno stanje te razne druge potrebne vrijednosti kako bi u idućoj epizodi ponovno započela borba protiv glavnog neprijatelja.

Također se maknu iz scene svi projektili i mine koje trenutno postoje.

```
// On episode begin do this
public override void OnEpisodeBegin() {
    Debug.Log("On episode begin");
    this.transform.position = positionOfPlayer;
    PlayerHealthController.instance.currentHealth =
    PlayerHealthController.instance.maxHealth;
    LevelManager.instance.coinsCollected = 0;
    CheckpointController.instance.DeactivateCheckpoints();
    UIController.instance.UpdateCoinCount();
    UIController.instance.UpdateHealthDisplay();

    theBossBattle.gameObject.SetActive(false);
    theBossActivator.SetActive(true);
    theBossBattle.health = 5;
    theBossBattle.currentState = BossController.bossStates.shooting;
    theTank.SetActive(true);
    winPlatforms.SetActive(false);
```

```
        theBossBattle.isDefeated = false;
        theBossBattle.timeBetweenMines = 1.25f;
        theBossBattle.timeBetweenShots = 2f;
        theBossBattle.hitbox.SetActive(true);
        theTank.transform.position = positionOfTank;

        theTank.transform.localScale = new Vector3(1f, 1f, 1f);
        theBossBattle.moveRight = false;

        GameObject[] mines = GameObject.FindGameObjectsWithTag("Mines");
        if (mines.Length > 0) {
            Debug.Log("exists mines");
            foreach (GameObject mine in mines) {
                Destroy(mine);
            }
        }

        GameObject[] bullets = GameObject.FindGameObjectsWithTag("Bullets");
        if (bullets.Length > 0) {
            Debug.Log("exists bullets");
            foreach (GameObject bullet in bullets) {
                Destroy(bullet);
            }
        }
    }
```

OnActionReceived metoda

U `OnActionReceived` provjerava se ako je glavnom neprijatelju zdravlje došlo do nule, u kojem slučaju agent dobiva pozitivnu nagradu.

```
    if (theBossBattle.health <= 0) {
        Debug.Log("Killed the boss");
```

```
        AddReward(+30f);
    }
```

3.7.2 Dodavanje samoučećeg agenta na objekt glavnog lika

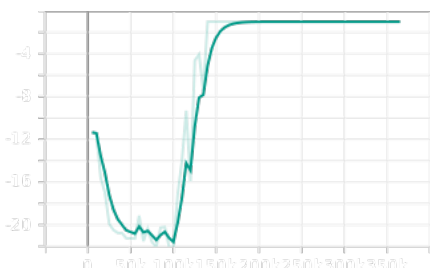
Isto kao i u prethodnim potpoglavljima izmjene koje se trebaju napraviti su imenovanje ponašanja te imenovanje demonstracije. Vrijednost za ponašanje je "Level 5 Imitation", dok je vrijednost za demonstraciju "Level5Demo".

Snimanje demonstracije, konfiguracijska datoteka i početak treniranja rade se na isti način kao i za prve četiri razine.

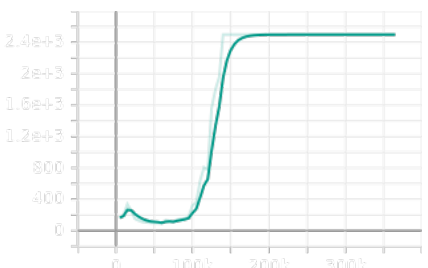
3.7.3 Rezultati treniranja pete razine

Samoučeći agent nije uspješno naučio kako doći do kraja pete razine. Problemi koje ima su da ne "razumije" što i koliko puta treba napraviti da pobijedi glavnog neprijatelja. Tijekom treniranja metodom učenja imitacijom uspijeva udariti glavnog neprijatelja, no nakon što ga jednom udari, ne razumije što treba napraviti da ga udari ponovno. Nakon nekog vremena treniranja samoučeći agent prestane uopće ulaziti u prostor gdje se nalazi glavni neprijatelj jer ga počne biti "strah" dobivati negativne nagrade i počne trčati i skakati sve dok ne dosegne maksimalni broj koraka koji može napraviti prije početka nove epizode.

Graf kumulativne nagrade i graf dužine epizode prikazani su na slikama 3.9 i 3.10.



Slika 3.9 Graf kumulativne nagrade za samoučećeg agenta na petoj razini



Slika 3.10 Graf dužine epizode za samoučećeg agenta na petoj razini

Kada bi se promijenio sistem nagrade, količina zapažanja i/ili konfiguracijska datoteka moguće je da bi se dobilo bolje rezultate i postiglo da samoučeći agent završi razinu, no sa trenutnim vrijednostima postavljenim to se ne može postići.

Zaključak

U ovom radu su razvijeni i primijenjeni samoučeći agenti za rješavanje, odnosno prelaženje različitih razina u računalnoj igri. Cilj je bio istražiti te analizirati uspješnost samoučećih agenata na različitim razinama računalne igre.

Rezultati istraživanja pokazuju da je samoučeći agent postigao zadovoljavajuće rezultate na prve tri razine igre. Agent je uspješno naučio optimalne strategije i donosio odluke temeljene na svojim prethodnim iskustvima. Međutim, na zadnje dvije razine računalne igre rezultati su pokazali slabiju uspješnost samoučećeg agenta. Agent se suočavao s težim izazovima i složenijim situacijama koje nije znao kako riješiti. To ukazuje na potrebu za daljnjim unapređenjem algoritma učenja kako bi se mogao nositi s kompleksnijim scenarijima.

Moguće je da bi se bolji rezultati dobili kada bi se, umjesto pokretanja novog treniranja za svaku razinu zasebno, koristio model treniran na ranijim razinama da se treniraju kasnije razine. Također, bolji rezultati bi se dobili i kada bi se broj zapažanja u okolini povećao te kada bi se sistem nagrade poboljšao.

Unatoč tim izazovima, ovo istraživanje pruža uvid u primjenu umjetne inteligencije u automatiziranju rješavanja računalne igre. Otkriveno je da je samoučeći agent sposoban naučiti i primijeniti strategije na određenim razinama računalne igre, ali daljnji rad je potreban kako bi se unaprijedila uspješnost na složenijim razinama.

U budućnosti, moguće je proširiti ovu studiju na druge računalne igre, primijeniti naprednije algoritme i eksperimentirati s različitim vrstama umjetne inteligencije kako bi se poboljšala uspješnost agenta. To će omogućiti daljnje istraživanje i razumijevanje sposobnosti umjetne inteligencije u rješavanju složenih problema u računalnim igrama.

Bibliografija

[1] Blender. , s Interneta, https://www.blender.org/ , Travanj 2023.

[2] Aseprite. , s Interneta, https://www.aseprite.org/ , Travanj 2023.

[3] AdamCYounis: "An Aseprite Crash Course In 30 Minutes" , s Interneta, https://www.youtube.com/watch?v=59Y6OTzNrhk&t=626s&ab_channel=AdamCYounis , Travanj 2023.

[4] Itch.io. , s Interneta, https://nvph-studio.itch.io/dog-animation-4-different-dogs , Travanj 2023.

[5] Unity. , s Interneta , https://unity.com/ , Svibanj 2023.

[6] Udemy. , s Interneta , https://www.udemy.com/course/unityplatformer/ , Svibanj 2023.

[7] Unity ML-Agents Toolkit. , s Interneta , https://unity-technologies.github.io/ml-agents/ , Lipanj 2023.

[8] Enemy AI in Unity Games with ML-Agents Toolkit. , s Interneta , https://pavcreations.com/enemy-ai-in-unity-games-with-ml-agents-toolkit/ , Lipanj 2023.

[9] Imitation Learning for 2D platformer with Unity ML-Agents. , s Interneta , https://pavcreations.com/imitation-learning-for-2d-platformer-with-unity-ml-agents/ , Lipanj 2023.

Sažetak

U ovom završnom radu opisano je kako izraditi model i animacije za glavnog lika psa, kako izraditi jednostavnu 2D platformer računalnu igru te kako u nju implementirati samoučećeg agenta. Pojašnjeno je kako izraditi model i animacije u programu *Aseprite*, konkretno kako izraditi model i animacije psa. Objašnjena je osnovna izrada računalne igre u *Unity engineu* te kako, koristeći alat *Unity ML-agents*, implementirati i napraviti logiku za samoučeće agente kojima je cilj završiti sve razine u računalnoj igri.

Ključne riječi **— model i animacije, računalna igra, samoučeći agenti**

Abstract

In this thesis, it is described how to design a model and animations for the main character of a dog, how to make a simple 2D platformer computer game and how to implement self-learning agents in the game. It is explained how to design a model and animations using the program *Aseprite*, specifically how to design the model and animations for a dog. It is also explained how to make a simple computer game using *Unity engine* and how to, using *Unity ML-agents* toolkit, implement and create the logic for self-learning agents whose goal is to finish all the levels in a computer game.

Keywords **— model and animations, computer game, self-learning agents**