Razvoj 2D akcijske avanture u Unity okruženju

Krnjić, Domagoj

Sveučilište Jurja Dobrile u Puli
Tehnički fakultet u Puli

DOMAGOJ KRNJIĆ

RAZVOJ 2D AKCIJSKE AVANTURE U UNITY OKRUŽENJU

Završni rad

Pula, rujan 2024. godine

Sveučilište Jurja Dobrile u Puli
Tehnički fakultet u Puli

DOMAGOJ KRNJIĆ

RAZVOJ 2D AKCIJSKE AVANTURE U UNITY OKRUŽENJU

Završni rad

JMBAG: 0303096721, redoviti student/ići
Studijski smjer: Sveučilišni prijediplomski studij Računarstva

Kolegij: Programiranje
Znanstveno područje: Tehničke znanosti
Znanstveno polje: Računarstvo
Znanstvena grana: Programsko inženjerstvo
Mentor: izv. prof. dr. sc. Tihomir Orehovački

Pula, rujan 2024. godine

Sažetak

Za ovaj završni rad cilj je bio razviti 2D akcijsku avanturu koristeći Unity razvojno okruženje i programski jezik C#. Zbog svoje jednostavnosti i opsežne dokumentacije, Unity se pokazao kao fleksibilan i učinkovit alat za izradu indie igara. Za realizaciju projekta korišteni su vanjski alati, poput softvera za izradu pixel arta i OpenGameArt repozitorija za vizualne elemente. U igri je naglasak stavljen na istraživanje, preživljavanje i borbu protiv neprijatelja, koji su kontrolirani umjetnom inteligencijom. Cilj igrača je prikupiti zvijezde, komunicirati sa seljanima te na kraju poraziti glavnog neprijatelja. Iako vremenska ograničenja nisu omogućila potpunu implementaciju svih planiranih funkcionalnosti, osnovni cilj izrade indie igre je postignut.

Ključne riječi: Unity okruženje, animacije, programiranje, C#, Akcijska avantura.

Abstract

For this thesis, the goal was to develop a 2D action adventure using the Unity development environment and the C# programming language. Due to its simplicity and extensive documentation, Unity has proven to be a flexible and efficient tool for creating indie games. For the realization of the project, external tools were used, such as software for creating pixel art and the OpenGameArt repository for visual elements. The game's emphasis is on exploration, survival and combat against enemies, which are controlled by artificial intelligence. The goal of the player is to collect stars, communicate with the villagers and finally defeat the main enemy. Although time constraints did not allow the full implementation of all planned functionalities, the basic goal of creating an indie game was achieved.

Keywords: Unity environment, animations, programming, C#, Action adventure.

Sadržaj

1. Uvod

Tema ovog rada bila je izrada 2D akcijske avanture unutar Unity razvojnog okruženja, koja obuhvaća sve ključne korake u procesu razvoja videoigre. Unity koristi C# kao programski jezik unutar svog editora te je dizajniran za izradu igara za različite platforme, uključujući računala, konzole i mobilne uređaje. Cilj ovog rada bio je tehnička realizacija igre, uz primjenu potrebnih znanja u razvoju igara te rješavanje izazova koji su se javljali tijekom procesa izrade. Igra se fokusira na elemente istraživanja i preživljavanja, a inspiraciju crpi iz igara poput „The Legend of Zelda“, „Stardew Valley“ i „Evoland“.

U ovom radu prikazan je cjelokupni proces izrade igre „2D akcijska avantura“ kako bi se demonstrirala mogućnost samostalnog kreiranja indie igre u suvremenim uvjetima. Igra počinje tako što se glavni lik pojavljuje na sredini mape, s ciljem prikupljanja deset zvijezda i poraza glavnog neprijatelja. Radnja započinje istraživanjem šume, gdje je igraču dodijeljen zadatak prikupljanja zvijezda. Tijekom igre, glavni lik može interagirati s raznim objektima na mapi, uključujući skrivena blaga u škrinjama koje igrač može otkriti. Pored toga, igraču je omogućeno komunicirati sa stanovnicima sela u blizini i preuzimati zadatke od njih. Zbog opasnosti koje vrebaju u ovoj avanturi, igraču je potrebna odgovarajuća oprema za borbu protiv neprijatelja. Unutar sela se nalazi trgovac koji prodaje oružje, opremu i sadnice za vrt, koje igrač može kupiti za određenu svotu novca. Interakcije sa seljanima odvijaju se kroz dijalog koji se mijenja s napretkom u zadacima. Borbeni sustav igre temelji se na umjetnoj inteligenciji koja detektira prisutnost igrača kada se nalazi u blizini neprijatelja. Postoji više vrsta neprijatelja, svaki s vlastitom borbenom mehanikom. Krajnji cilj igre je poraziti glavnog neprijatelja, kojeg igrač mora pronaći i analizirati njegovu taktiku borbe. U miroljubivim zonama igre igrač može spremiti svoj napredak i kasnije nastaviti igru od te točke.

Rad je podijeljen u pet poglavlja, uključujući uvod i zaključak. Igra je razvijena u Unity verziji 2022.3.5f1, a igriva je isključivo na operativnom sustavu Windows. U daljnjem tekstu opisuje se detaljna razrada projekta, uključujući korištenje vanjskih alata za obradu pixel umjetnosti i IDE Visual Studio za programiranje tijekom izrade igre.

2. Slične igre

Inspiraciju za razvoj igre pružile su igre poput Evoland 2 i The Legend of Zelda: A Link to the Past, te većina mehanika dolazi iz igara istog žanra.

2.1. Evoland 2

Evoland 2: A Slight Case of Spacetime Continuum Disorder je role-playing video igra iz 2015. godine koju je razvila i objavila Shiro Games. Evoland 2 je prvi put objavljen za Microsoft Windows u kolovozu 2015., a podrška za ostale platforme, poput kućnih konzola i Linuxa, tek izlazi u veljači 2019. Evoland 2 je nastavak originalnog Evolanda, s grafičkim stilom koji se mijenja prilikom putovanja glavnog lika kroz vrijeme [1]. Slika 1 prikazuje izgled igre Evoland 2.

Slika 1. Snimka zaslona iz igre Evoland 2 [1]

2.2. The Legend of Zelda: A Link to the Past

The Legend of Zelda: A Link to the Past je akcijsko-pustolovna igra koju je razvio i objavio Nintendo za Super NES. Treća je igra u serijalu The Legend of Zelda, objavljena 1991. u Japanu te 1992. u Sjevernoj Americi i Europi.

Radnja je smještena u vrijeme puno prije prve dvije igre Zelda. Igrač igra kao heroj po imenu Link koji mora spasiti Hyrule, poraziti kralja Ganona i spasiti potomke sedam mudraca. A Link to the Past općenito se smatra izvrsnom igrom koja je hvaljena zbog svoje prezentacije i inovativnosti igranja, često se nalazi na listama najboljih igara svih vremena [2]. Slika 2 prikazuje izgled igre The Legend of Zelda A Link to the Past.

Slika 2. Snimka zaslona iz igre The Legend of Zelda: A Link to the Past [3]

3. Razvojno okruženje Unity

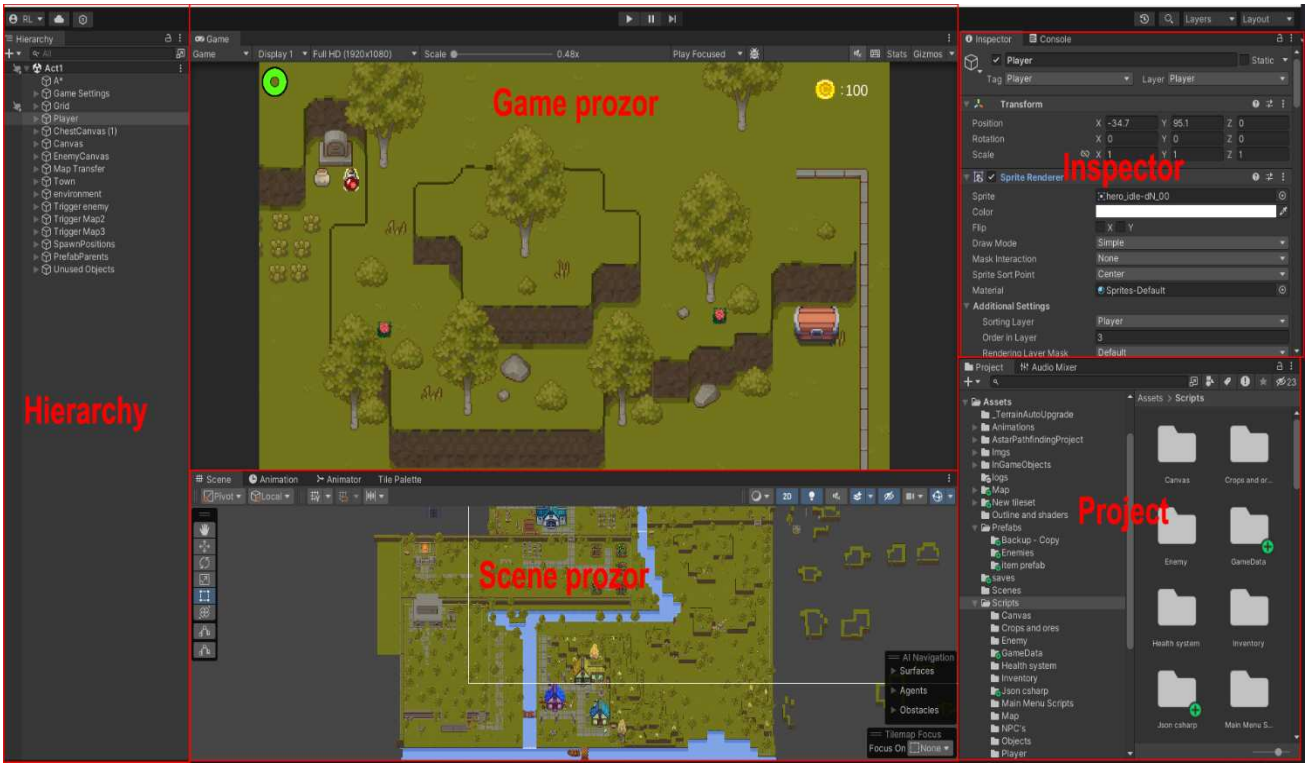
U razvoju igre za ovaj projekt korišteno je razvojno okruženje Unity Game Engine. Unity je alat za razvoj igara koji je dizajnirao i izradio Unity Technologies 2005. s ciljem da programeri ne moraju izrađivati novo razvojno okruženje ako žele kreirati igru, nego mogu koristiti unaprijed napravljeno okruženje [4]. Među ostalim softverima za izradu igara, Unity se ističe svojom sposobnošću da omogućava igranje u prvom licu te u dvodimenzionalnom i trodimenzionalnom grafičkom okruženju, uključujući VR. Unity je izvorno bio dostupan samo za Mac, ali sada podržava više platformi. Multiplatformska podrška znači da se u Unity-u mogu razvijati igre za Windows, Mac, Linux, pametne telefone, tablete, konzole za videoigre itd. [5]. Unity je prilagođen početnicima, što omogućava široj publici da uđe u svijet kreiranja videoigara, neovisno o prijašnjem iskustvu. Unity, uz bogatu ponudu materijala, često se koristi kao alat za početnike u razvoju igara zbog jednostavnosti korištenja. Unity Asset Store omogućuje pristup velikoj količini 3D modela, 2D sprite-ova, zvučnih efekata, dodatnih alata i vizualnih efekata. Ovaj pristup omogućava efikasniju raspodjelu vremena, s većim fokusom na programiranje i dizajn igre. Razvojno okruženje Unity je napravljeno u C++, a za kreiranje igara koristi se jezik visoke razine, C#, koji je također objektno orijentiran. Najpopularnije igre izrađene u Unity-u su: Among Us, Fall Guys, Cuphead, Pokemon Go, Genshin Impact, Hollow Knight, Hearthstone [5].

2.1. Unity editor sučelje

Unity editor sučelje prema zadanim postavkama (eng. default) je podijeljeno u određene sekcije, a svaka sekcija ima svoju ulogu i to su: Project, Hierarchy, Scene, Game, Console, Inspector. [6]) Slika 3 prikazuje izgled Unity editor sučelja sa označenim sekcijama.

- **Project:** sekcija prikazuje sve datoteke koje su relevantne za igru, poput spriteova, skripti, zvukova, fontova itd. Za ubacivanje novih stvari u projekt sekciju, samo se povuče izvan Unity-a datoteka i pusti na mjesto gdje je projekt prozor.
- **Hierarchy:** Sadrži sve game objekte koji su trenutno u sceni, oni su sortirani po redu kada su dodani u scenu, a mogu se naknadno ručno premještati. Hierarchy sekcija navodi naše objekte u grupe, zvane „Parents“ i „Children“.

- **Inspector:** Služi za prilagodbu postavki odabranog objekta, omogućavajući izmjene imena, oznaka (eng. Tag) i slojeva (eng. Layer).
- **Scene view:** Prikazuje vizualni prikaz igre i omogućuje interakciju s objektima u sceni.
- **Game view:** Prikazuje kako će igra izgledati kada se pokrene.
- **Console window:**. Koristi se za prikazivanje grešaka i upozorenja tijekom razvoja igre [6].



Slika 3. Prikaz grafičkog sučelja Unity.

2.2. Komponente

Game objekt je osnovna klasa za sve entitete u Unity-u [7]. Predstavlja nevidljivi spremnik koji zauzima određenu poziciju u prostoru, ima rotaciju i veličinu. Na game objekt se mogu dodavati komponente s različitim značajkama.

Komponenta (eng. Component) je osnovna klasa za sve što je dodijeljeno game objektu. [8]. Svaki game objekt može imati neograničen broj komponenata. Unity nudi niz unaprijed definiranih komponenata, dok programer može kreirati nove pomoću C# skripti.

Kako bi skripta mogla biti komponenta, mora naslijediti klasu MonoBehaviour [9] Kao što je prikazano na Slika 4.

```
1   using UnityEngine;
    Unity Script | 0 references
2   public class newComponent : MonoBehaviour
3   {
4       // Start is called before the first frame update
        Unity Message | 0 references
5       void Start()
6       {
7       }
8       // Update is called once per frame
        Unity Message | 0 references
9       void Update()
10      {
11      }
12  }
```

Slika 4. Novo kreirana komponenta

MonoBehaviour klasa sadrži osnovne funkcionalnosti komponente u Unity-u . Svaka skripta koja nasljeđuje ovu klasu sadrži zadane funkcije poput:

- **Start:** Ova funkcija pokreće se jednom, prije funkcije Update, dok je objekt skripte aktivan u sceni.
- **Update:** Ova funkcija se pokreće svaki okvir (eng. Frame) dok je objekt skripta aktivna u sceni.
- **Awake** funkcija koja se može dodati u skriptu i izvršava se prije funkcije Start. Za razliku od Start funkcije, game objekt skripta ne treba biti aktivna u sceni kako bi se Awake funkcija pozvala.
- **OnTriggerEnter2D:** funkcija koja se poziva kada drugi objekt s komponentom Collider2D napravi koliziju s objektom koji ima isto Collider2D i ovu funkciju.
- **OnTriggerExit2D:** funkcija koja se poziva kada drugi objekt s komponentom Collider2D izađe iz kolizije s objektom koji ima isto Collider2D i ovu funkciju [9].

Prikazivanje varijabli u editoru omogućuje se tako što se varijabli u skripti doda public. Jedna od izazovnih strana korištenja public modifikatora pristupa je to što, kada Unity serijalizira programerske skripte, editor samo serijalizira public modifikator pristupa, pa

tako svaka komponenta s instancom te skripte može pristupiti toj varijabli. Također, može se koristiti SerializeField što forsira Unity da prikaže privatni modifikator pristupa u editoru [10].

Zadane komponente za potrebe projekta:

- **Transform**: Kontrolira poziciju, rotaciju i mjerilo GameObjecta. Ova komponenta je zadana na svakom GameObjectu te se bez nje ne može kreirati objekt.
- **Rigidbody2D**: Komponenta se dodaje kako bi kontrolirali fiziku GameObjecta. Rigidbody2D nadjačava Transform komponentu, te ona sama ažurira poziciju.
- **Collider 2D:** Collider je komponenta koja upravlja sudarima koji se dogode između GameObjecta. 2D Collider omogućuje da određujemo veličinu oblika. Za 2D igre postoje: Box Collider 2D, Circle Collider 2D i Polygon Collider 2D.
- **Animator:** Animator kontrolira animacije na GameObjectu koristeći Animator Controller, pomoću kojeg se definiraju isječci animacije.
- **Camera**: Prikazuje scenu iz određene perspektive, te je komponenta s kojom igrač ima pogled na svijet.
- **Audio Source**: Reproducira audio zapise.
- **Audio Listener**: Prima audio signale sa svih Audio Sourceova u sceni. Svaka scena može imati samo jedan Audio Listener, te je uvijek dodan glavnoj kameri.
- **Canvas**: Komponenta koja prikazuje sve UI elemente u Unityju. Potrebna je za prikaz bilo kakvih UI elemenata i kontrolira njihov redoslijed prikaza. Svaki UI element mora biti dijete Canvasa kako bi se mogao prikazati u sceni.
- **Image**: Prikazuje UI sliku, koja se koristi za pozadine, gumbe i druge UI vizualne elemente [8].

4. Proces razvoja igre

4.1. Glavni izbornik

Glavni izbornik u igrama obično služi kao prvi dojam koji igrač dobiva o igrici, nakon najave (eng. Trailer). Izbornik s prikladnim izgledom i glazbom može zadržati pažnju korisnika i poboljšati ukupno iskustvo u igri, uključujući prilikom čekanja suigrača ili tijekom pauziranja igre. Uz to, važan je dio i dizajn opcija jer daje predodžbu o tome je li igra dobro ili loše razvijena te koliko će lako ili teško biti kretanje kroz izbornike/igru. Nakon pokretanja igre, igrač se susreće s glavnim izbornikom prikazanim na Slika 5 gdje ima 5 opcija, a to su Nova igra (eng. New Game), Nastavi prijašnju igru (eng. Continue previous game), Zasluge (eng. Credits), Opcije (eng. Options) i Napusti igru (eng. Quit). Pritiskom na bilo koju opciju otvara se novi UI prozor.

Slika 5. Prikaz glavnog izbornika

Slika 6 prikazuje New Game prozor koji omogućuje igraču da izabere ime na koje će se igra spremati. Ime rekorda filtrira se da nema razmaka kako bi se sigurno spremilo kao

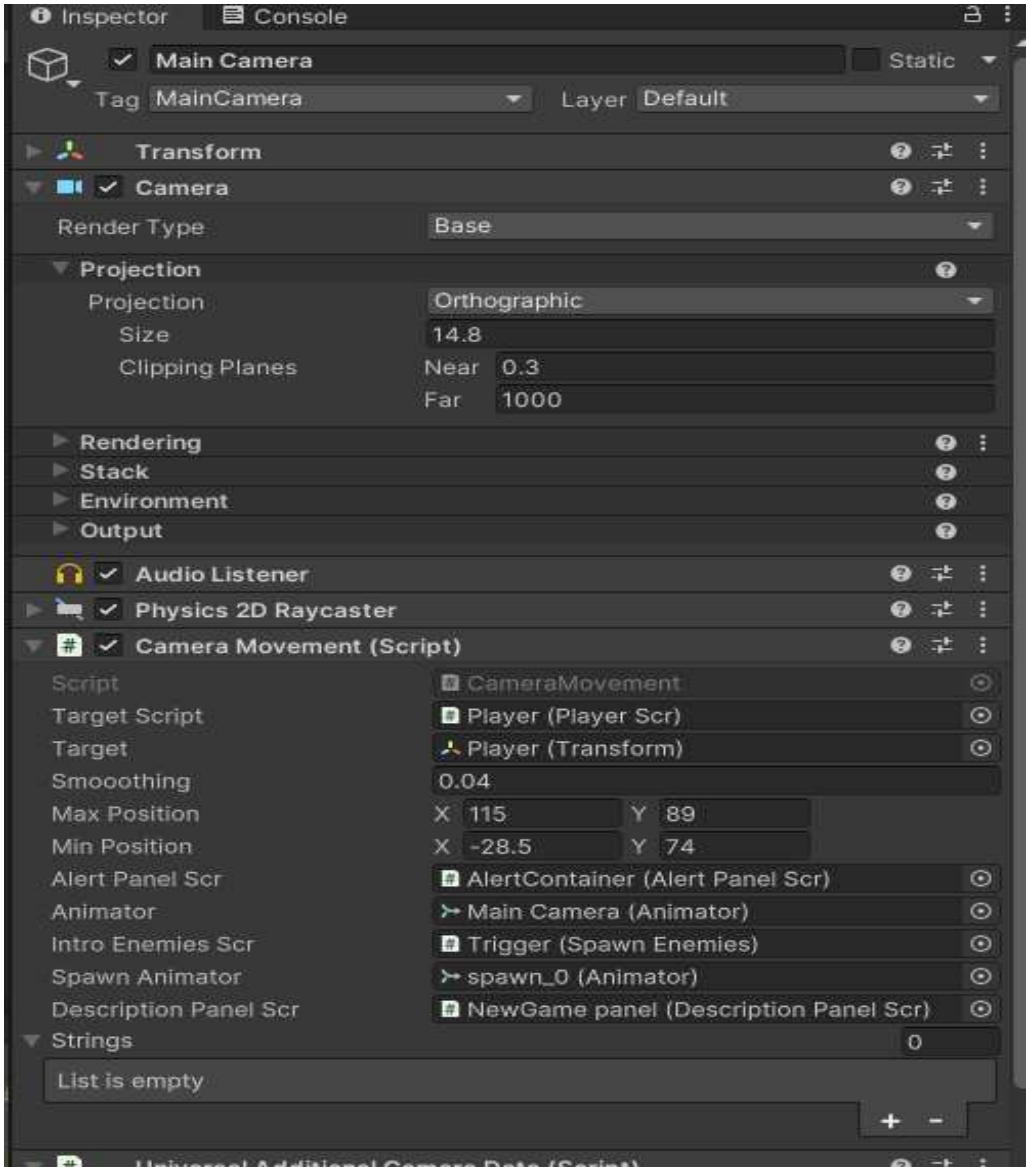
ime datoteke json. Ako se ime ne izabere, igra uzima trenutni timestamp. Prozor New Game također nudi igraču da odabere razinu težine.

Slika 6. Prikaz prozora za novu igru

Vrsta načina na koji Unity sprema unesene podatke te ih dijeli kroz sve scene naziva se ScriptableObject. To je spremnik podataka koji omogućuje pohranu podataka u Unityu, neovisno o scenama. Jedna od primjena je optimizacija memorije, odnosno kako bi se spriječilo dupliciranje podataka. Najviše je korisno za „Prefabe“ GameObjecta s MonoBehaviour skriptama koje sadrže statične podatke. Korištenjem ScriptableObjecta, pravimo nacrt i navodimo koje podatke može pohraniti, te kreiramo objekt iz tog šablona. Svaki put kada kreiramo „Prefab“, imat će svoju kopiju tih podataka. Na ovaj način postoji samo jedna kopija podataka u memoriji, što smanjuje korištenje memorije [11]. Primjer jednog ScriptableObjecta iz igre: unutar glavnog izbornika igraču je igrač omogućeno izmjenjivati ScriptableObject koji se prenosi u novu scenu kada igrač odabere „Start Game“ i sprema ih kao Assete unutar projekta. Podaci spremljeni u ScriptableObject Assete zapisuju se na disk, što ih čini uvijek pristupačnima između scena.

4.2. Kamera

Kamere su GameObjecti koji prikazuju svijet igračima. Svaka Unity scena dolazi s GameObjectom Main Camera, koja hvata (eng. Capture) objekte u sceni i prikazuje ih u Game prozoru. Kako bi se prikazala bilo koja scena u Unityu, mora postojati kamera. U jednoj sceni može se koristiti neograničen broj kamera, a svaka kamera može prikazivati scenu iz drugačije perspektive [12].



Slika 7. Inspector glavne kamere iz igre

Slika 7 prikazuje glavnu kameru koja sadrži komponentu „Camera“. Projekcija (eng. Projection) mijenja kako kamera simulira perspektivu, između „perspective“ ili „orthographic“ stajališta. Razlika između njih jest što „orthographic“ stajalište ne prikazuje „sense of depth“ ili osjećaj dubine. Ova projekcija je najviše korisna kod kreiranja 2D igara. Zbog toga se u ovoj igri koristi ortografska kamera, kako bi svi objekti izgledali kao da su isto udaljeni [12].

```
public class CameraMovement : MonoBehaviour
{
    public PlayerScr targetScript;
    public Transform target;
    public float smoothing;
    public Vector2 maxPosition;
    public Vector2 minPosition;
    Vector3 targetPosition;
    public AlertPanelScr alertPanelScr;
    public Animator animator;
    [SerializeField] SpawnEnemies introEnemiesScr;
    [SerializeField] Animator spawnAnimator;
    [SerializeField] descriptionPanelScr descriptionPanelScr;
    [SerializeField] List<string> strings = new List<string>();
    public bool cameraAnimDone = true;
    private string str1 = "Controls \n Movement \"W A S D\"\r\nRunning \"left Shift\"\r\nInteraction \"E\"\r\nInventory \"I\
    private string str2 = "Welcome to Mystic Quest: Arachne's Lair!\r\n\r\nAs dawn breaks, you find yourself mysteriously su
    @ Unity Message | 0 references
    private void Start()
    {

        strings.Add(str1);
        strings.Add(str2);
    }
    @ Unity Message | 0 references
    void LateUpdate()
    {
            if (transform.position != target.position)
            {
                targetPosition = new Vector3(target.position.x, target.position.y, transform.position.z);
                targetPosition.x = Mathf.Clamp(targetPosition.x, minPosition.x, maxPosition.x);
                targetPosition.y = Mathf.Clamp(targetPosition.y, minPosition.y, maxPosition.y);
                transform.position = Vector3.Lerp(transform.position, targetPosition, smoothing);
            }
    }

    1 reference
    public void animationDone()
    {
        animator.enabled = false;
        descriptionPanelScr.showDescriptionPanel(strings);
        cameraAnimDone = true;
    }
    0 references
    public void spawnAnimation()
    {
        spawnAnimator.enabled = true;
    }
    1 reference
    public void spawnPlayer()...
    1 reference
    public void newgame()
    {
        //Debug.Log("New game");
        cameraAnimDone = false;
        PlayerScr.playerCanMove = false;
        introEnemiesScr.newGameIntroSpawn();
        animator.Play("cameraanimator");
    }
    2 references
    public void MapTransfer(Vector2 valueMin, Vector2 valueMax, string name)
    {
        minPosition = valueMin;
        maxPosition = valueMax;
        alertPanelScr.showAlertPanel(name);
    }
```

Slika 8. Skripta glavne kamere iz igre

Kameri je dodijeljena C# skripta koja će joj kontrolirati ponašanje kroz cijelu igru. Slika 8 prikazuje skriptu „Camera Movement“ s funkcijom LateUpdate, koja se poziva automatski svaki frame poslije Update funkcije. Razlog korištenja LateUpdate-a je taj što je potrebno da se prije izvedu kretnje igrača od kamere koja bi ga trebala pratiti. Funkcija provjerava uvjet ima li komponenta Transform kamere istu poziciju kao i „target“, što je u ovom slučaju igrač. Ako je uvjet istinit, pozicija igrača se sprema u Vector3 varijablu, te se provjerava sa statičkom metodom „Mathf.Clamp“ je li pozicija igrača veća ili manja od granica kamere. Ako je prvi argument u rasponu između min i max, vraća prvi argument. Ako je prvi argument niži od min, vraća min, a ako je prvi argument veći od max, vraća max [13]. Zadnja linija u bloku postavlja poziciju kamere kao vraćeno stanje metode „Vector3.Lerp“. Metoda Lerp vraća interpolirani float između prva dva parametra, dok zadnji parametar float predstavlja koliko daleko treba interpolirati [13]. Ova metoda se koristi kako bi kamera imala malo kašnjenje iza igrača. U CameraMovement skripti postoje par funkcija ovisno o lokaciji i željenoj mehanici. MapTransfer je funkcija koja se izvodi kada igrač dođe na mjesto gdje se mijenja mapa, uglavnom to bude nevidljivi objekt sa OnTriggerEnter funkcijom. Ona poziva funkciju od kamere kako bi se promijenile granice do kojih kamera može doći, te kako bi se prikazalo ime mape kao tekst. NewGame funkcija se izvodi kada igrač odabere novu igru u izborniku, tada kamera preko Unity Animator komponente izvodi uvod igrice. Kada intro animacija završi, kamera iz instance igračeve skripte poziva funkciju spawnPlayer.

4.3. Zvuk

Zvučni efekti doprinose kvaliteti igre i omogućavaju poboljšano iskustvo igranja. Kako bi Unity uopće mogao reproducirati zvukove, potrebna je komponenta audio listener, koja je kao zadana već postavljena na glavnu kameru u sceni [14]. U ovoj igri audio listener je prebačen na odvojeni GameObject nazvan „AudioManager“, koji će imati sve potrebno za ostvarivanje zvuka. Osim audio listener-a, GameObject „AudioManager“ će imati i komponentu Audio Source koja će služiti kao izvor zvuka. Kod kompleksnijih game objekata poput igrača, koji imaju više različitih zvukova, treba pozivati na AudioManager zvukove kroz skriptu.

```
public class AudioManager : MonoBehaviour
{
    public static AudioManager instance;
    public AudioMixerGroup mixerGroup;
    public Sound[] sounds;
    @ Unity Message | 0 references
    void Awake()
    {
        if (instance != null)
        {
            Destroy(gameObject);
        }
        else
        {
            instance = this;
        }
        foreach (Sound s in sounds)
        {
            s.source = gameObject.AddComponent<AudioSource>();
            s.source.clip = s.clip;
            s.source.loop = s.loop;
            s.source.outputAudioMixerGroup = mixerGroup;
        }
    }
    42 references
    public void Play(string sound)
    {
        Sound s = Array.Find(sounds, item => item.name == sound);
        if (s == null)
        {
            Debug.LogWarning("Sound: " + name + " not found!");
            return;
        }
        s.source.volume = s.volume * (1f + UnityEngine.Random.Range(-s.volumeVariance / 2f, s.volumeVariance / 2f));
        s.source.pitch = s.pitch * (1f + UnityEngine.Random.Range(-s.pitchVariance / 2f, s.pitchVariance / 2f));
        s.source.Play();
    }
}
```

Slika 9. Prikaz AudioManager skripte za kontroliranje zvuka

Sounds polje prikazano na Slika 9 sadrži veći broj zvukova koje GameObject AudioManager može reproducirati. Na Awake funkcijskom pozivu dodjeljuje svakom zvuku u polju AudioSource komponentu te ostale konfiguracije za zvuk, poput bool tipa podatka koji označava ako se zvuk ponavlja. Funkcija također provjerava postoji li već AudioManager objekt u sceni; ako postoji, uništava se. Funkcija Play se poziva iz drugih skripti i ima samo jedan parametar tipa string koji je ime zvuka. Tijekom poziva funkcije provjerava se je li string argument, odnosno postoji li ime zvuka u polju „sounds“. Ako ime ne postoji u polju zvukova, funkcija se završava pomoću naredbe 'return'. Ukoliko zvuk postoji, isti se pokreće.

4.4. Animacije i Animator

Animacija stvara iluziju pokreta manipulirajući redoslijedom i brzinom prikazivanja nepokretnih slika[15]. Za razliku od 3D animacije, 2D animacija omogućuje kretanje samo

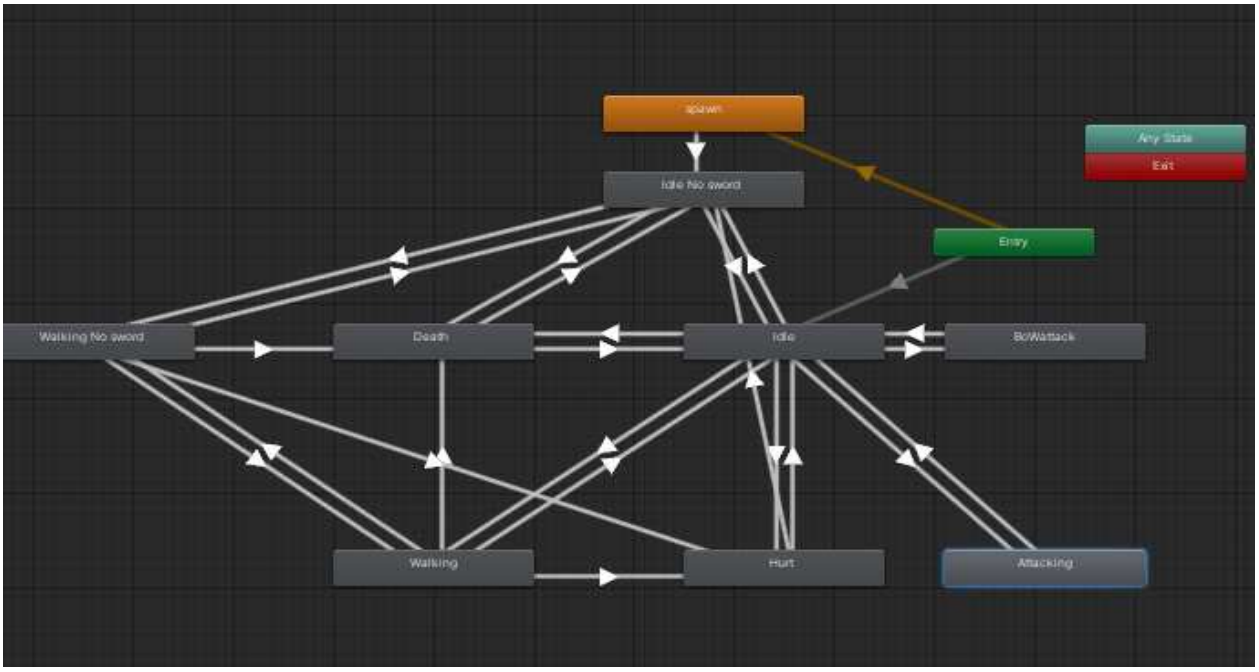
u četiri smjera . U prošlosti je 2D animacija bila crtana rukom, ali sada je moguće napraviti 2D animaciju putem softvera na računalu [16].

Slika 10. Prikaz animacije hodanja ulijevo i Unity animator

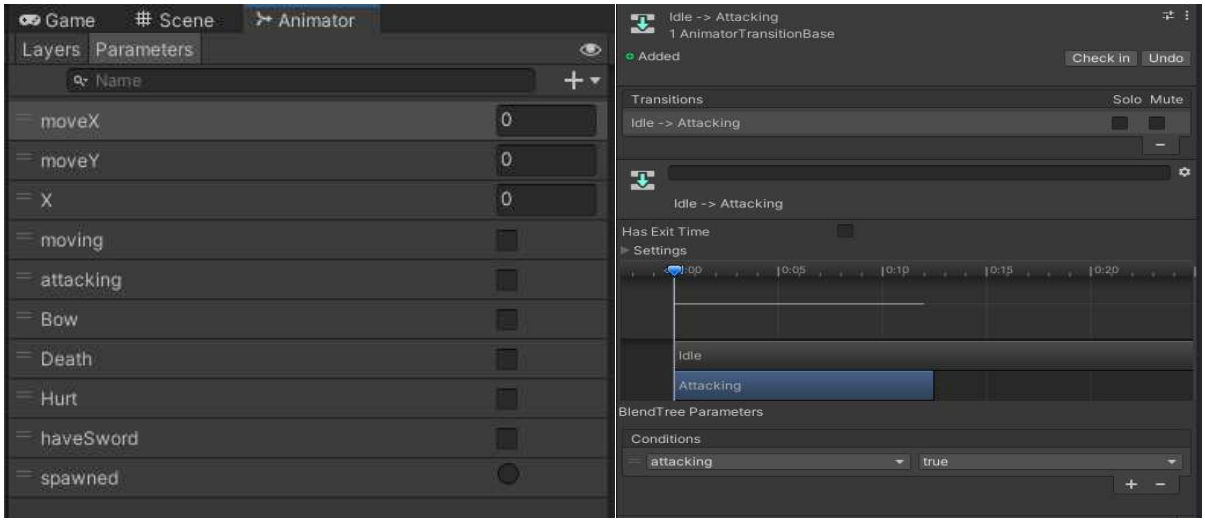
Slika 10 prikazuje četiri pixel art slike koje stvaraju iluziju kretanja lika ulijevo. Unity-eva sekcija za animacije određuje redoslijed kojim će se slika prikazati u određenom vremenskom periodu. U odabranom periodu animacije također se mogu dodavati eventi, što je korisno kada želimo pozvati funkciju, a animacija još nije završila. Za kompliciranije objekte poput igrača, potreban je Unity Animator controller.

Animator controller kontrolira logiku animiranog GameObject-a [17]. U Animatoru je moguće dodati parametre kao što su okidač (eng. Trigger), bool, int i float. Ovi parametri se zatim mogu pozivati ili mijenjati kroz skripte koje imaju tu animator instancu. Također postoje „States“ koji služe za kontrolu i tranzicije slijeda isječaka animacije . States predstavljaju pojedinačne Animation isječke u Animatoru. Tranzicije određuju kojim putem se animacija kreće od jednog „state-a“ do drugog, a parametri kontroliraju kada se ti prijelazi dogode [17]. Slika 11 prikazuje Animator igrača igre, te je postavljen na Player

game objekt. MoveX i MoveY se ažuriraju i mijenjaju svaki frame kada igrač hoda ili trči. Ovisno o varijablama MoveX i MoveY, animacija s najbližim zadovoljenim uvjetom je ona koja će se prikazati. Slika 11 prikazuje entry state sa strjelicom prema „spawn“ animaciji, što znači da je „spawn“ animacija zadana kao prva u redu izvođenja, dok ostale ovise o parametrima ili kraju izvođenja prethodne animacije ako nema uvjeta.



Slika 11. Animator komponenta igrača

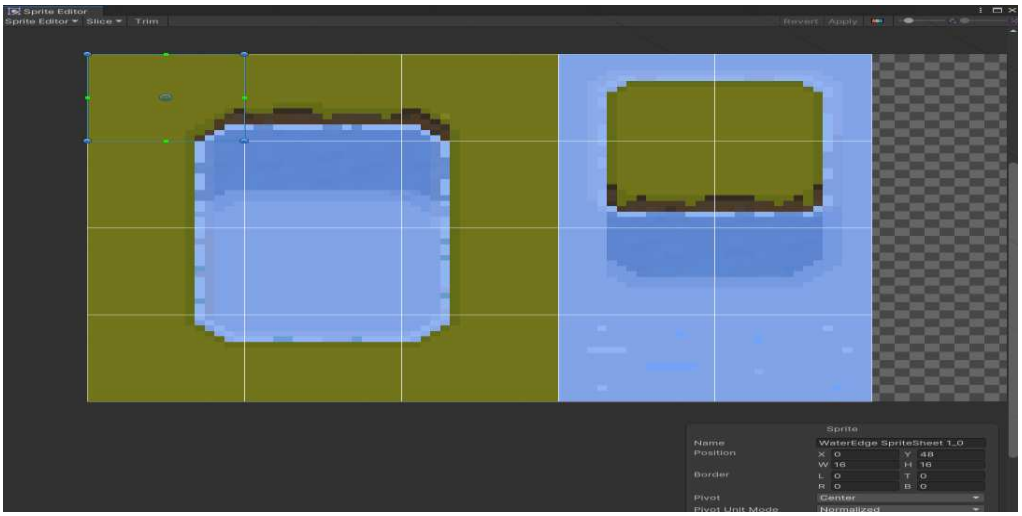


Slika 12. Animator komponenta igrača

Slika 12 prikazuje parametre tipa bool za Player GameObject: Moving, attacking, Bow, Death Hurt i haveSword. Kako se ne bi prekidala animacija tijekom izvođenja, u Unity Animatoru postoji varijabla Has Exit Time. Ta varijabla omogućuje da se animacije ne prekinu sve dok nisu u potpunosti završene, iako je parametar već promijenjen [17]. U nekim uvjetima ta varijabla se uključuje, poput u igri "Napad mača". Kako bi se spriječilo nepredviđeno iskorištavanje mehanika igre, igraču se onemogućuje kretanje dok je animacija napada u tijeku. Nakon završetka animacije napada izvesti će se sljedeća animacija.

4.5. Izgradnja okruženja u igri

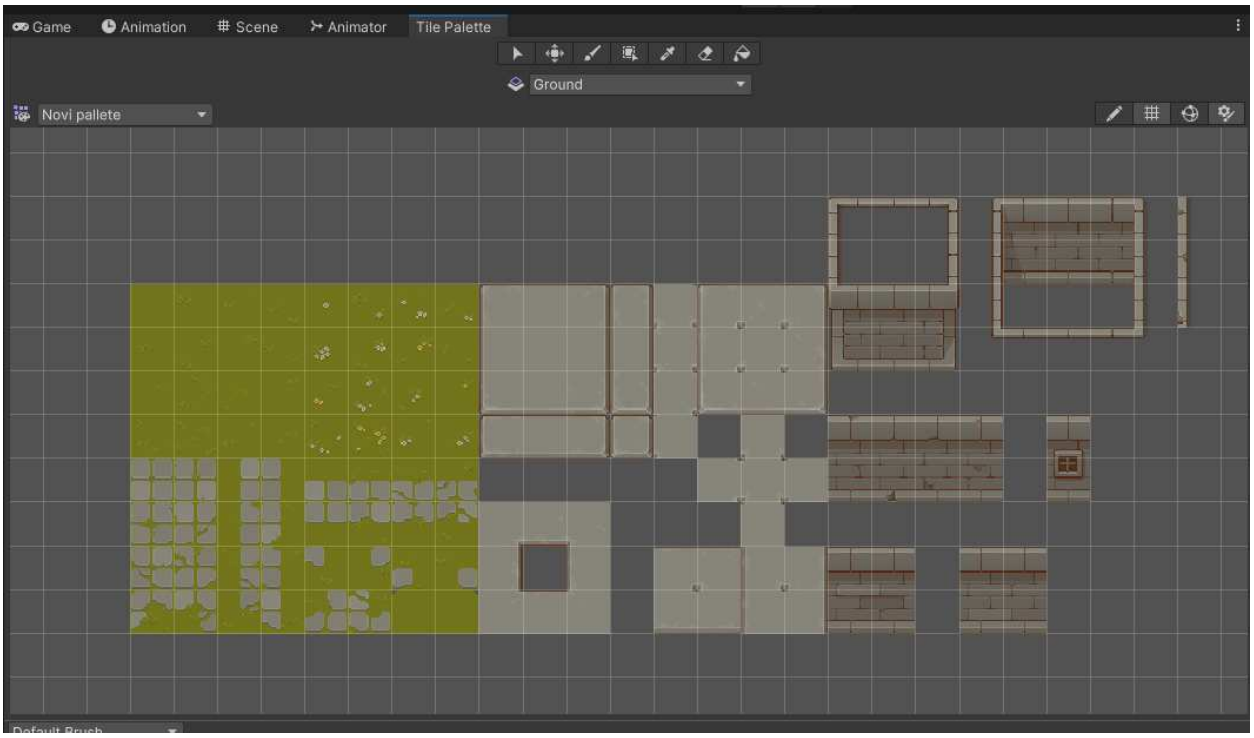
Za dizajn platforme ili mape korišteni su Tilesets i Tile Palette. Tileset je skup kvadratnih ili pravokutnih slika (pločica) koje se koriste za izradu 2D razina. Pločice mogu poprimiti različite izglede, kao što su izgled zemlje, trave, kamena i zida. Pločice su obično jednakih dimenzija kako bi se pravilno postavile na platformi, to jest da nema praznina između njih. Kada su postavljene, pločice poprimaju oblik rešetke (eng. Grid). Slike koje se ubacuju u rešetku potrebno je prvo ubaciti u Unity i postaviti postavke kao što prikazuje Slika 13 „Texture Type“ kao Sprite. Ako slika sadrži više pločica, tada se koristi Unity-ev Sprite Editor, koji se zatim koristi za dijeljenje slike u različite pločice [18]. Sljedeći korak u stvaranju okruženja 2D igre u Unityju je rad s Tile Palette i Tilemaps.



Slika 13. Prikaz kreiranja tileset-a u sprite editoru

Za kreiranje Tile Palette kao prikazano na Slika 14 potrebno je otvoriti Tile pallete prozor unutar Unity programa sljedećim koracima:

- Prozor Tile Palette: Window > 2D > Tile Palette u Unityjevom izborniku
- Stvaranje nove palete: pritiskom na gumb "Stvori novu paletu" u prozoru palete pločica. Nakon toga, potrebno je imenovati paletu, odabrati vrstu rešetke (eng. Grid) te na kraju odabrati mjesto spremanja unutar projekta.
- Dodavanje pločica u paletu: Dodavanje pločica se radi tako da se povuku željene pločice unutar prozora palete [19].



Slika 14. Tilemap Palette iz projekta

Nakon dodane palete koristi se Tilemap. Ako Tilemap ne postoji u popisu komponenti, potrebno ga je uvesti iz „Unity Package Manager“. Tilemap pomaže u pojednostavljenju dizajna i omogućava brzo slaganje rešetke za izgled karte. Za kreiranje Tilemap-a potrebno je odabrati kreiranje GameObject-a u sceni, pa onda 2D Object > Tilemap. Unity će stvoriti novi Grid objekt, a ispod njega će biti komponenta Tilemap. Unity Tilemap sastoji se od dva glavna elementa, a to su: rešetke i samog Tilemapa [20]. U ovom sustavu može postojati samo jedan grid u sceni, dok grid može sadržavati više tilemapa.

4.6. Korisničko sučelje (User interface, UI)

Unity UI predstavlja sve vizualne komponente s kojima korisnik vrši interakciju, uključujući gumbe za navigaciju, prikaz zdravlja igrača, trenutni novac, prikaz inventara i okvir dijaloga [21]. Prikaz korisničkog sučelja iz igre prikazan je na Slika 15. Dizajniranje korisničkog sučelja u Unityu zahtijeva da se prije svega postavi Canvas. Canvas sadrži tri komponente: Canvas, Canvas Scaler i Graphic Raycaster. Canvas komponenta mijenja gdje i kako se isti prikazuje na zaslonu. Unity UI komponente raspoređuju se unutar GameObject-a Canvasa s komponentom Rect Transform. To omogućuje kontrolu položaja, veličine i poravnanja elemenata. UI komponente u Canvasu ovise o hijerarhiji, što znači da se prvi UI element u redu prikazuje prvi, te ide redom do N. Ako UI elementi dijele isti prostor, posljednji u hijerarhiji bit će nacrtan iznad ostalih. Skripte se mogu priložiti bilo kojem elementu kako bi se omogućila interaktivnost korisničkog sučelja i korisnika. Unutar Unity scene može postojati više Canvasa [22]. Unity Canvas ažurira UI elemente svaki put kad se dogodi izmjena na njima, što bi značilo da korištenjem jednog Canvasa može doći do problema optimizacije [23]. Unutar UI izbornika postoje četiri glavna GameObject-a, a to su: Images, Buttons, Text i Panels.

Slika 15. Grafičko sučelje iz projekta

4.7. Fizika i kolizije

Fizika i sustavi kolizije u igri bitni su kako bi se likovi ponašali po nekim pravilima fizike. Upravljanje fizikom omogućuje komponenta „Rigidbody2D“, a određivanje fizičkog oblika omogućuje „Collider“. Kako bi svaki objekt igre u Unityju mogao poštovati fizičke zakone, mora imati komponentu Rigidbody2D stavljenu na njega [24]. U 2D igri, igrač i neprijatelji će imati pričvršćenu komponentu Rigidbody2D, a statični objekti moraju imati samo „Collider“ kako bi mogli vršiti interakcije sa svijetom kao što je prikazano na Slika 16. Svakom tijelu sa Rigidbody2D potrebne su:

• Gravitacija: Ovo je često isključeno, budući da likovi neće padati zbog gravitacije.

• Rotacija oko Z-osi je zaključana tako da se likovi ne okreću na neprirodan način tijekom sudara ili kretanja [24].

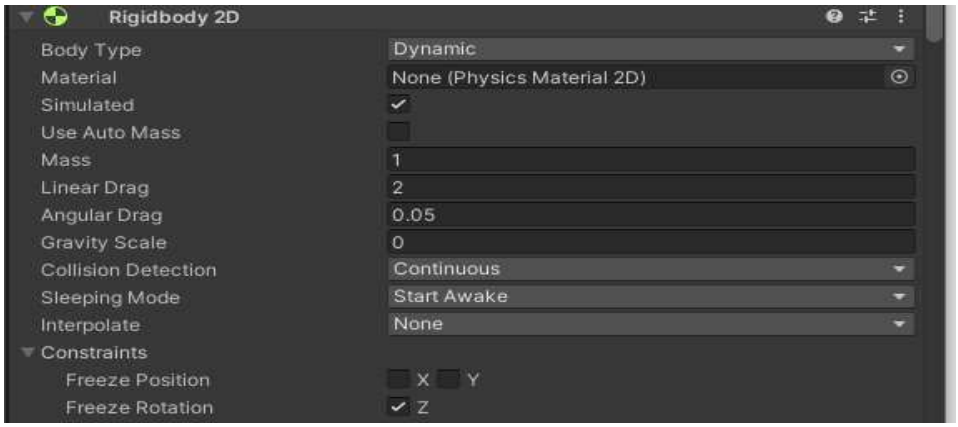
Postoji više vrsta komponente „Collider“ koje se koriste za različite primjene, a one su: „Circle Collider“, „Box Collider“, „Polygon Collider“, „Edge Collider“, „Capsule Collider“, „Composite Collider“ i „Custom Collider“. Box Collideri definiraju fizičke granice oko lika ili bilo kojeg objekta [25]. Većinu vremena bit će dva Box Collidera povezana s likom igrača, tako i za neprijatelje, a to su:

• Collision Collider: Rješava fizičke sudare lika igrača s okolinom. Sprječava igrača da se kreće kroz zidove ili bilo koje druge prepreke, osiguravajući da je kretanje ograničeno na dopuštene dijelove mape.

• Hitbox Collider: Ovaj Collider je za interakcije u borbi. Prepoznaje preklapanje oružja igrača, mača, s neprijateljem. Hitbox collider se postavlja kao okidač koji detektira preklapanja ostalih „Collider-a“ sa istim okidačem.

Drugi interaktivni objekti, poput vrata ili škrinja s blagom, također koriste okidače za provjeru prisutnosti igrača i pokreću događaje poput znakova kada je lik dovoljno blizu, ili interakciju s likom koji se ne može igrati (eng. NPC) kada igrač uđe u NPC-ev interakcijski prostor. RigidBody2D sadrži tri moguća tipa fizike, a to su „Dynamic“, „Kinematic“ i „Static“. Statički tip se koristi na objektima koji se ne mogu micati, niti imati okidač interakciju, što bi bili zidovi i prepreke. Kinematički tip se koristi na objektima koji će biti potpuno kontrolirani kroz C# skripte, kao što su pozicija, rotacija i svi ostali fizički atributi

tog objekta, koji se moraju ručno odrediti [26]. Primjer korištenja bio bi udarac mačem neprijatelja, koji će se odbaciti po želji programera. Dinamički tip kontrolira Unity Engine, a sve fizičke atribute, kao i sudare, igra sama kontrolira [27].



Slika 16. Komponenta Rigidbody2D igrača

4.8. Mehanike igre

4.8.1 Sustavi kretanja

U 2D igri, a posebno u RPG-u, sustav kretanja čini ključnu komponentu koja određuje kako igrač kontrolira svog lika i kako se lik kreće svijetom igre.

```
void FixedUpdate()
{
    if (FlashActive)
    {
        flash();
    }
    // Debug.Log(startTime);
    change = Vector3.zero;
    change.x = Input.GetAxisRaw("Horizontal");
    change.y = Input.GetAxisRaw("Vertical");
    if(Input.GetKey(KeyCode.Space) && currentState != PlayerState.attack && currentState != PlayerState.stagger && moving == false
    {
        cooldownBool = false;
        StartCoroutine(AttackCo());
    }
    else if( currentState == PlayerState.walk || currentState == PlayerState.idle)
    {
        if(playerCanMove)
            UpdateAnimationAndMove();
    }
```

Slika 17. Playerscr programski kod – 1. dio

Slika 17 prikazuje funkciju FixedUpdate koja postavlja vektor promjene na nulu ili Vector3(0, 0, 0), kako bi promjena bila glatkija te da se objekt što kraće zaustavlja ako se igrač više ne kreće. „change.x = Input.GetAxisRaw("Horizontal");“ i „change.y = Input.GetAxisRaw("Vertical");“ linije čitaju vodoravni i okomiti unos s tipkovnice. Vraćena vrijednost tih funkcija je u rasponu od -1 do 1, a unos s tipkovnice uvijek će biti -1, 0 ili 1. Rezultati dobiveni od pritisnutih tipki stavljaju se u vektor promjene (change). Horizontalna os upravlja se tipkama „a“ i „d“, dok se vertikalna os upravlja tipkama „w“ i „s“. Ovaj sustav omogućava igraču kretanje lijevo/desno (vodoravno) i gore/dolje (vertikalno). „if(currentState == PlayerState.walk || currentState== PlayerState.idle)“ ovaj uvjet dozvoljava kretanje samo ako je enum („walk“) ili („idle“) stanje, kako bi filtrirali stanja poput stanja smrti, ako je slučajem igrač izgubio sve živote. Nakon provjere može li se „player“ kretati, poziva se funkcija „UpdateAnimationAndMove“.

```
private void UpdateAnimationAndMove()
{
    if(attackActive== false)
    {
        if(change != Vector3.zero && canMove)
        {
                moving = true;
                MoveCharacter();
            animator.SetFloat("moveX", change.x);
            animator.SetFloat("moveY", change.y);
            animator.SetBool("moving", true);
        }
        else
        {
            animator.SetBool("moving", false);
                moving = false;
        }
    }
}
```

Slika 18. Playerscr programski kod – 2. dio

Slika 18 prikazuje funkciju UpdateAnimationAndMove koja provjerava uvjet: „if (change != Vector3.zero && canMove)“, odnosno dolazi li do promjene položaja igrača. Funkcija ažurira animacije, postavlja stanje parametra „moving“ na vrijednost „true“ te na kraju poziva funkciju MoveCharacter.

```
1 reference
public void MoveCharacter()
{
    change.Normalize();
    //audioManager.Play("PlayerSteps");
    myRididbody.MovePosition(transform.position + change * speed * Time.fixedDeltaTime);
}
```

Slika 19. Playerscr programski kod – 3. dio

Normaliziranje vektora promjene sa Slika 19 osigurava da se lik kreće istom brzinom bez obzira na smjer [28]. U slučaju dijagonalnog kretanja, brzina neće biti veća od kretanja udesno. Zadnja linija koristi komponentu Rigidbody, kako bi se omogućilo kretanje temeljeno na fizici. Ovo se primjenjuje dodavanjem normaliziranog vektora promjene, pomnoženog brzinom i Time.fixedDeltaTime, za kretanje koje je neovisno o vremenu.

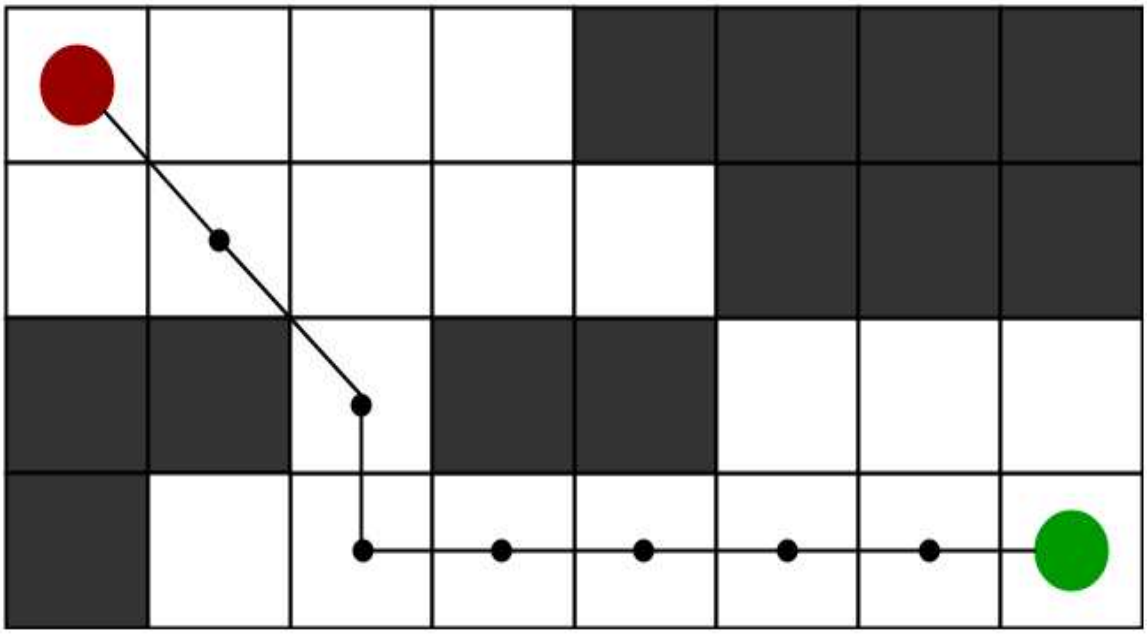
```
private void Update()
{
    if (CanPass)
    {
        anim.SetBool("Moving", true);
        npcScr.dialogTekst = newText;
        if (Vector3.Distance(movespots.position, transform.position) <= 1) {

            //CanPass = false;
            anim.SetBool("Moving", false);
            gameObject.GetComponent<AllowPassage>().enabled = false;
        }
        Wall.SetActive(false);
        transform.position = Vector2.MoveTowards(transform.position, movespots.position, speed * Time.deltaTime);
        anim.SetFloat("x1", (movespots.position.x - transform.position.x));
        anim.SetFloat("y1", (movespots.position.y - transform.position.y));
    }
}
```

Slika 20. NPC programski kod

Slika 20 prikazuje Update funkciju koja izmjenjuje položaj NPC-a s njegove trenutne lokacije na cilj određenom brzinom. Približavanje cilju: linija transform.position= Vector2.MoveTowards(transform.position, movespots.position, speed * Time.deltaTime); omogućava NPC-u se da kreće u smjeru pozicije zadane u polju movespots. Ažuriranja animacije: kako bi odražavao smjer kretanja, kod ažurira parametre animacije: anim.SetFloat("x1", (movespots.position.x - transform.position.x)); anim.SetFloat("y1", (movespots.position.y - transform.position.y)); Ove linije osiguravaju da se animacija NPC-a mijenja ovisno o smjeru u kojem se kreće, kako bi animacija bila usklađena s promjenom pozicije NPC-a. Kretanje neprijateljskog NPC-a uključuje kombinaciju više

funkcija uz korištenje A* algoritma. Navedeni algoritam puno je bolji od korištenja Unityjevog MoveTowards za pronalaženje igračeve putanje budući da izračunava kako doći do cilja na najbolji mogući način, uzimajući u obzir sve vrste prepreka i opći položaj. Dok MoveTowards jednostavno miče neprijatelja prema meti u ravnoj liniji, A* pronalazi najkraći put bez prepreka ili slijepih ulica kao što prikazuje Slika 21.

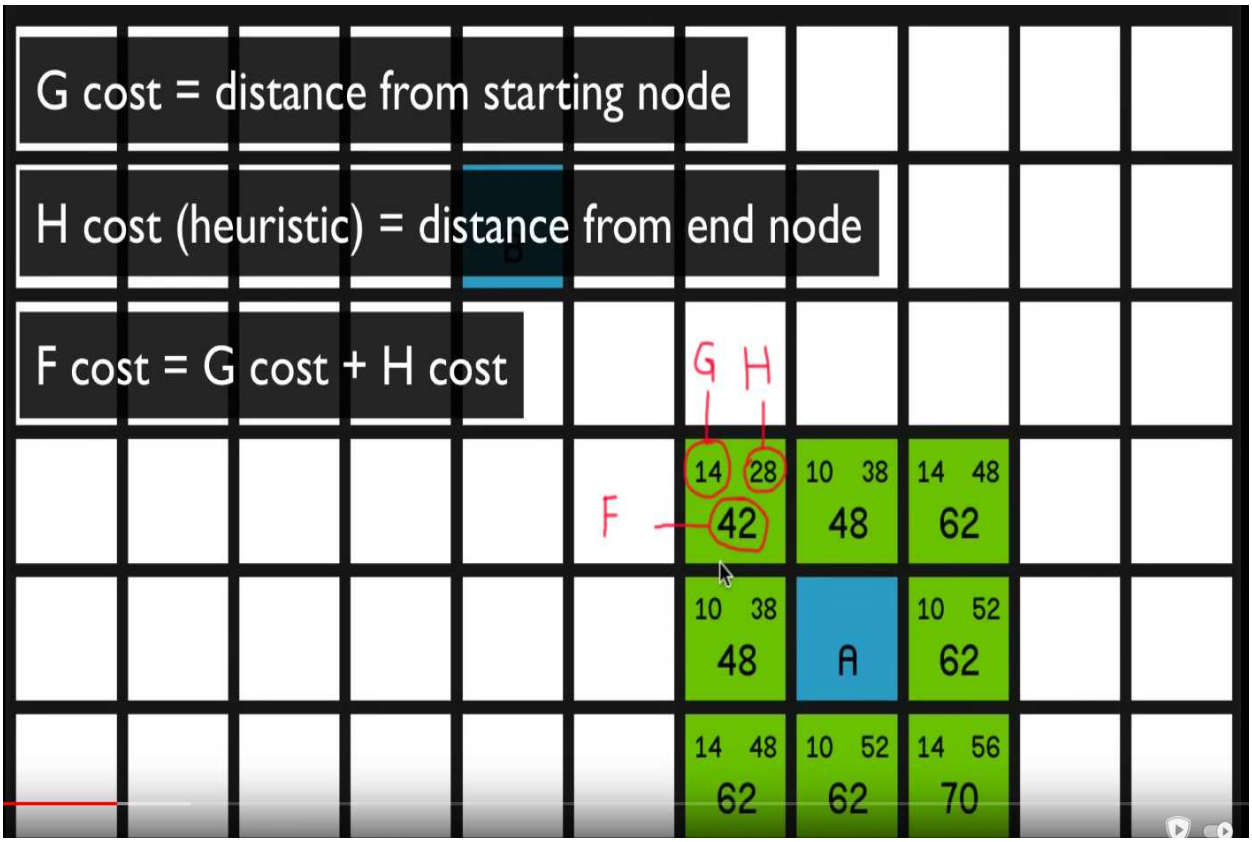


Slika 21. Prikaz generiranja puta A algoritma [29]*

A* je jedan od najučinkovitijih algoritama za pronalaženje puta i obilazak grafa. Vrlo je poznat u razvoju igara koje sadrže neku vrstu pronalaženja puta, budući da brzo izračunava najkraći put s velikom točnošću [29].

Ključne komponente:

- g-cijena: Označava cijenu koja je korištena za pomak s početne pozicije na trenutnu ćeliju. Cijena se nastavlja povećavati kako se algoritam pomiče iz jedne ćelije u drugu.
- h-cijena: Procijenjeni ukupni trošak koliko će koštati put od početne ćelije do ciljne ćelije.
- f-trošak: To predstavlja zbroj g-troška i h-troška. Najmanji f-trošak od ćelija koje su istražene postat će poželjan put, te će u sljedećoj iteraciji ta ćelija biti istražena [30].



Slika 22. Prikaz kako funkcionira A algoritam [30]*

Prva ćelija trenutna pozicija koja je otvorena i smještena na popis otvorenih ćelija. Zatim se izračunava f-trošak za susjedne ćelije i odabire ćelija s najnižom f-cijenom za daljnje širenje. Odabrana ćelija postaje zatvorena i postupak se ponavlja, izračunavajući f-trošak susjednih ćelija, osim onih koje su već zatvorene. Kako se približava cilju, g-cijena se povećava, a h-cijena se smanjuje [30] kao što prikazuje Slika 22. Kako bi se A* algoritam koristio unutar Unity okruženja, treba se uvesti iz Unity Asset Store-a. Nakon toga se dodaje Pathfinder skripta praznom GameObject-u. Svi objekti na layeru maske „Collider“ će A* algoritam smatrati preprekama, te će ih izbjegavati u pronalaženju najbržeg puta [31]. Svi NPC-evi koji će imati ovaj algoritam trebaju imati i skriptu „Seeker“. Skripta Seeker će generirati put, a kontroliranje brzine NPC-eva će se vršiti kroz „EnemyFollowPlayer“ komponentu.

```
public class EnemyFollowPlayer : EnemyR
{
    [Header("AI Behaviour")]
    public EnemyStateMachine currentStateAi;
    //[SerializeField] private float speed = 500f;
    [SerializeField] private float sightRange = 25;
    private bool hasLineOfSight = false;
    private int layerMask = ~(1 << 8 | 1 << 6);

    // Seeker
    public float nextWaypointDistance = 3f;
    private Path path;
    private int currentWaypoint = 0;
    private bool reachedEndOfPath = false;
    private Seeker seeker;
    private float distance;
    private bool isUpdatingPath = false;

    // Temp variables
    private Vector2 direction;
    private Vector2 force;
```

Slika 23. EnemyFollowPlayer programski kod – 1. dio

Slika 23 prikazuje klasu EnemyFollowPlayer koja nasljeđuje od osnovne klase EnemyR i upravlja ponašanjem umjetne inteligencije da prati igrača ili luta uokolo. Enum „EnemyStateMachine“ mijenja stanje NPC-a ovisno o tome je li u potrazi za igračem ili patrolira. Detekcija i linija vidljivosti:

- sightRange: Definira koliko daleko neprijatelj može vidjeti.
- hasLineOfSight: Prati može li neprijatelj vidjeti igrača.
- layerMask: Isključuje određene slojeve (npr. drveće) iz provjera vidljivosti.

Pronalaženje puta:

- Seeker: komponenta odgovorna za generiranje putanje.
- Path: trenutni put koji se prati.

Patrola (eng. Patrol): Neprijatelji po zadanom stanju patroliraju, odnosno kreću se nasumično unutar predefiniranog područja sve dok ne vide igrača. Potjera (eng. Chase): Stanje potjere se pokreće kada neprijatelj otkrije igrača, izmjenjuje način kretanja na A* i meta je igrač. Raycast otkrivanje za promjenu stanja: Prijelaz iz lutanja u potjeru upravlja

se putem Raycasta. Raycast vraća istinu ako se zraka (eng. Ray) križa s Colliderom, inače vraća lažno.

```
1 reference
void RayCastShot()
{
    ray = Physics2D.Raycast(transform.position, target.position - transform.position, sightRange, layerMask);
    if (ray.collider != null)
    {
        // Debug.Log(ray.collider.gameObject.name);
        hasLineOfSight = ray.collider.CompareTag("Player");
        if (hasLineOfSight)
        {
            Debug.DrawRay(transform.position, target.position - transform.position, Color.green);
            if (!isUpdatingPath)
            {
                canMove = true;
                anim.SetBool("StartWalking", true);
                StopCoroutine(WanderCooldown());
                currentStateAi = EnemyStateMachine.Chase;
                isUpdatingPath = true;
                if (!hasTarget)
                    EmoteShow();

                hasTarget = true;
                InvokeRepeating("UpdatePathPlayer", 0f, .5f);
            }
        }
        else
        {
            Debug.DrawRay(transform.position, target.position - transform.position, Color.red);
            if (isUpdatingPath)
            {
                CancelInvoke("UpdatePathPlayer");
                isUpdatingPath = false;
            }
        }
    }
}
```

Slika 24. EnemyFollowPlayer programski kod – 2. dio

Slika 24 prikazuje funkciju Raycast koja je zaslužna za provjeru postoji li jasna linija vidljivosti prema igraču. Prima 4 argumenta, a to su: trenutna pozicija neprijatelja, smjer u kojem se nalazi igrač, domet do kojeg neprijatelj može opaziti igrača i layerMask ili sloj koji će ignorirati. Ako je igrač otkriven (hasLineOfSight je true), neprijatelj se prebacuje u stanje Chase, te se počinje kretati prema igraču i ažurira svoju putanju pomoću algoritma A*. InvokeRepeating poziva metodu UpdatePathPlayer. U vremenu od pola sekunde, ta ista će se ponovo pozvati. Korištenjem InvokeRepeating optimizira se igra, pošto je generiranje i traženje puta skup proces. Putanja NPC-a ne mora se ažurirati u svakom okviru, stoga je interval postavljen na pola sekunde radi optimizacije.

```
if (currentStateAi == EnemyStateMachine.Patrol && canMove)
{
    // Debug.Log("patrol logic");

    if (Vector2.Distance(transform.position, initialPosition) > wanderingRadius)
    {
        direction = ((Vector2)path.vectorPath[currentWaypoint] - myRigidbody.position).normalized;
        force = direction * enemyScribtableObject.speed * Time.deltaTime;
        myRigidbody.AddForce(force);
        distance = Vector2.Distance(myRigidbody.position, path.vectorPath[currentWaypoint]);
        if (distance < nextWaypointDistance)
            currentWaypoint++;
        return;
    }
    if (!coroutineStarted)
    {
        direction = ((Vector2)path.vectorPath[currentWaypoint] - myRigidbody.position).normalized;
        force = direction * enemyScribtableObject.speed * Time.deltaTime;
        myRigidbody.AddForce(force);
        distance = Vector2.Distance(myRigidbody.position, path.vectorPath[currentWaypoint]);
        if (distance < nextWaypointDistance)
            currentWaypoint++;
    }
    if (tempPosition == transform.position)
    {
        stuckCounter++;
        if (stuckCounter > 5)
        {
            StopCoroutine(WanderCooldown());
            StartCoroutine(WanderCooldown());
            stuckCounter = 0;
        }
    }
    else
    {
        tempPosition = transform.position;
    }
```

Slika 25. EnemyFollowPlayer programski kod – 3. dio

Slika 25 prikazuje isječak skripte neprijatelja tijekom stanja patroliranja, dok još nije Raycast pronašao igrača, linija currentStateAi == EnemyStateMachine.Patrol: provjerava je li neprijatelj još uvijek u stanju patroliranja. Dok canMove boolean provjerava je li neprijatelju dopušteno kretanje. Ako su oba uvjeta istinita, patrolna logika se nastavlja. Provjera granice: uvjet provjerava je li neprijatelj otišao predaleko od svoje početne pozicije pomoću formule udaljenosti između trenutne pozicije neprijatelja (transform.position) i početne pozicije (initialPosition). Ako je ta udaljenost veća od radijusa patrole, neprijatelj mijenja smjer kako bi se vratio u patrolu unutar radijusa. Nakon istinitog uvjeta izračunava se smjer i sila. Linija myRigidbody.AddForce(force) primjenjuje silu na GameObject neprijatelja, što pomiče Rect Transform poziciju objekta prema

sljedećoj točki u radijusu. Linija if (distance < nextWaypointDistance) provjerava je li neprijatelj unutar dometa trenutne točke, nastavlja do sljedeće u patrolnom putu, povećavajući vrijednost varijable currentWaypoint. Uvjet provjerava ako !coroutineStarted nije krenuo, nastavlja se kretati duž putanje patrole na isti način kao što je prijašnje opisano sa smjerom. Ako je objekt zastao na nekom dijelu tijekom njegove putanje, izvodi se blok koda koji provjerava je li se pozicija nije promijenila, tada se stuckCounter povećava. Uvjet if (stuckCounter > 5) provjerava je li se neprijatelj zaglavio previše puta tijekom 5 okvira, zaustavlja trenutnu rutinu i započinje novu funkciju WanderCooldown().

Slika 26 odgovorna je za upravljanje ponašanjem neprijateljske umjetne inteligencije kada je u stanju potjere, a to provjerava prvim uvjetom. Prvi uvjet unutar bloka provjerava je li neprijatelj još uvijek izvan svog dometa napada, koji je definiran kao radijus napada. Izračun smjera ((Vector2)path.vectorPath[currentWaypoint] - myRigidbody.position). normalized; izračunava normalizirani vektor smjera od trenutnog položaja neprijatelja do sljedeće točke na putanji koju generira algoritam A*. Tijekom kretanja, animacija se ažurira ovisno o varijabli hasLineOfSight, koja prima vrijednost od Raycast zrake. Ako neprijatelj ima jasnu liniju prema meti (line of sight), smjer kretanja animacije postavlja se prema meti. Dok, ako je stanje varijable hasLineOfSight „false“, neprijatelj ne može vidjeti metu, tada se smjer animacije postavlja na putanju izračunatu algoritmom A*. Primjer: Ako se igrač sakrije iza objekta i prikrije „line of sight“, NPC će ići do zadnje pozicije gdje je A* zabilježio igrača. Kad dođe do te pozicije, tada dolazi do uvjeta ako ga vidi ili ne. Nakon animacije objektu se dodaje sila koja je produkt smjera pomnoženog s brzinom i vremenom. Komponenta Rigidbody2D sa izračunatom silom pomiče objekt prema cilju. U slučaju neistinitosti uvjeta, je li neprijatelj još uvijek izvan svog dometa, provjerava se „else if“ blok koji ga vraća iz logike potjere. Ako uvjet unutar ovog bloka nije zadovoljen, neprijatelj započinje funkciju napada.

```
else if (currentStateAi == EnemyStateMachine.Chase && canMove)
{
   // Debug.Log("chase logic");
    if (Vector2.Distance(target.position, transform.position) > enemyScribtableObject.attackRadius)
    {
        anim.SetBool("StartWalking", true);
        direction = ((Vector2)path.vectorPath[currentWaypoint] - myRigidbody.position).normalized;
        if (hasLineOfSight)
        {
            anim.SetFloat("MoveX", (target.position.x - transform.position.x));
            anim.SetFloat("MoveY", (target.position.y - transform.position.y));
        }
        else
        {
            anim.SetFloat("MoveX",  path.vectorPath[currentWaypoint].x - myRigidbody.position.x);
            anim.SetFloat("MoveY", path.vectorPath[currentWaypoint].y - myRigidbody.position.y);
        }

        force = direction * enemyScribtableObject.speed * Time.deltaTime;
        myRigidbody.AddForce(force);
        distance = Vector2.Distance(myRigidbody.position, path.vectorPath[currentWaypoint]);
        if (distance < nextWaypointDistance)
            currentWaypoint++;
    }
    else if (Vector3.Distance(target.position, transform.position) > enemyScribtableObject.chaseRadius)
    {
        Debug.Log(">chaseRadius");
       // anim.SetBool("StartWalking", false);
        return;
    }
    else if (hasLineOfSight && Vector3.Distance(target.position, transform.position) <= enemyScribtableObject.attackRadius)
    {
        if (!coroutineStarted2)
            StartCoroutine(AttackCo());
    }
}
```

Slika 26. EnemyFollowPlayer programski kod – 4. dio

4.8.2 Sustav borbe

Kada je riječ o stvaranju 2D RPG-a, borba je jedan od najbitnijih sustava. Popularan izbor je korištenje jednostavnog sustava napada u kojem igrač izvoditi osnovne napade jednostavnim pritiskom na gumb. Igrač se sukobljava s neprijateljima u stvarnom vremenu, bez naizmjeničnih mehanika. Kada igrač udari neprijatelja ili obrnuto, može dobiti kratki bljesak, zajedno sa zvučnim efektom i animacijom koja prikazuje gubitak srca na mjestu gdje je nastala šteta. Prikaz sustava borbe vidljiv je na Slika 27

Slika 27. Prikaz vidljivih kolizija i zraka iz igre

Knockback skripta prikazana na Slika 28 se nalazi na GameObject-u igrača, odnosno na „polygon collider2D“ mača sa svih četiri strane, također na neprijateljima i projektilima koje neprijatelji ispaljuju tijekom borbe. Sadrži varijable:

- thrust: Varijabla vraća količinu guranja koja je dana objektu tijekom sudara.
- knockTime: Trajanje dok objekt ostaje u "knocked" stanju, odnosno dok se ne može kretati.
- Damage: Označava količinu pretrpljene štete nakon udarca s neprijateljem ili igračem.
- damageBoost: Statička varijabla koja se koristi za dinamičko povećanje štete tijekom igranja. Oprema na igraču može utjecati na ovu varijablu.
- Hit: Referenca na komponentu Rigidbody2D kojom je objekt pogođen.

```
public class Knockback : MonoBehaviour
{
    public float thrust;
    public float knockTime;
    public float damage;
    public static float damageBoost = 0;
    Rigidbody2D hit;
```

Slika 28. Knockback programski kod – 1. dio

```
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.CompareTag("breakable") && this.gameObject.CompareTag("PlayerHitBox"))
        other.GetComponent<pot>().Smash();

    if (other.gameObject.CompareTag("Enemy") || other.gameObject.CompareTag("PlayerHurtBox"))
    {
        if (other.gameObject.CompareTag("PlayerHurtBox"))
            hit = other.GetComponentInParent<Rigidbody2D>();

        else
            hit = other.GetComponent<Rigidbody2D>();

        if (hit != null)
        {
            if (other.gameObject.CompareTag("Enemy") && gameObject.CompareTag("Enemy"))
                return;
            Vector2 difference = hit.transform.position - transform.position;
            difference = difference.normalized * thrust;
            hit.AddForce(difference, ForceMode2D.Impulse);
            if (other.gameObject.CompareTag("Enemy") &&  other.isTrigger)
            {
                hit.GetComponent<EnemyR>().currentState = EnemyStateR.stagger;
                if (PlayerScr.IsDashing)
                    other.GetComponent<EnemyR>().Knock(hit, knockTime, damage + damageBoost);
                else
                {
                    other.GetComponent<EnemyR>().Knock(hit, knockTime, damage + damageBoost);
                    AdrenalinScr.value += 0.1f;
                }
            }
            if(other.gameObject.CompareTag("PlayerHurtBox"))
            {
                if(other.GetComponentInParent<PlayerScr>().currentState != PlayerState.stagger)
                {
                    hit.GetComponent<PlayerScr>().currentState = PlayerState.stagger;
                    other.GetComponentInParent<PlayerScr>().Knock(knockTime, damage);
                }
            }
        }
    }
}
```

Slika 29. Knockback programski kod – 2. dio

Slika 29 prikazuje funkciju OnTriggerEnter2D, koja prima parametar „Collider2D“, te se poziva ako objekt s komponentom „Collider2D“ sa uključenom varijablom „isTrigger“ dođe u doticaj s komponentom istim na GameObject-u s Knockback skriptom.

Prvi uvjetni blok provjerava je li sudarna oznaka "breakable" i također provjerava je li "Collider" koji je bio uključen u doticaj bio igračev hitbox ili projektil. U slučaju istine, kod dohvaća "pot" komponentu, skriptu koja kontrolira objekte koji se mogu polomiti, i poziva javnu funkciju Smash() da razbije objekt.

Drugi uvjetni blok provjerava objekt s kojim ima koliziju, ako ima oznaku „Enemy“ ili „PlayerHurtBox“. Ovisno o oznaci, u slučaju istinitog uvjeta sprema se referenca na komponentu Rigidbody2D od vlasnika oznake. Ako je to igračeva hurtbox, ide gore u hijerarhiji kako bi se dobila igračeva Rigidbody2D komponenta. Inače se izvodi izravno dobivanje neprijateljskog Rigidbody2D.

Sljedeći uvjet provjerava ako neprijatelj udari drugog neprijatelja, tada se ništa ne vraća i izlazi se iz funkcije. Ako taj uvjet nije istinit, tada će se izračunati smjer i veličina povratne sile koja će se primijeniti na pogođeni objekt. Linija difference.normalize * thrust osigurat će da udarac djeluje u pravom smjeru i potrebnog impulsa. Uspješni udarac igrača prema neprijatelju mijenja stanje neprijatelja na „stagger“, što ga sprječava da se kreće ili napada. Funkcija Knock se zatim poziva na neprijatelja koji je zadobio udarac, dodajući silu prema smjeru od igrača i štetu (eng. Knockback, damage). Ako je igrač pogođen, ali nije već u stanju („stagger“), bit će udaren i pretrpjet će štetu.

```
2 references
public virtual void Knock(Rigidbody2D myRigidbody, float knockTime, float damage)
{
    if (isTargetable)
    {
        isTargetable = false;

        if (Health - damage <= 0)
        {
            heartsToSpawn = (int)(Health / 2);
            Debug.Log(heartsToSpawn);
        }

        else
            heartsToSpawn = (int)(damage / 2);
        Health -= damage;
        UpdateHearts();
        for (int i = 0; i < heartsToSpawn; i++)
        {
            Instantiate(knockHeart, transform.position + new Vector3(0, 1, 0) * (i + 1), Quaternion.identity);
        }
        if (damage % 2 == 1 && Health>=0)
            Instantiate(knockHeart2, transform.position + new Vector3(0, 1, 0) * (heartsToSpawn + 1), Quaternion.identity);

        if (Health <= 0)
            Death();
        else
        {
            FlashActive = true;
            flashCounter = flashLenght;
            FindObjectOfType<AudioManager>().Play("LogPain");
            StartCoroutine(KnockCo(myRigidbody, knockTime));
        }
    }
}
```

Slika 30. EnemyFollowPlayer programski kod – 5. dio

Neprijatelj i igrač dijele funkciju Knock slične implementacije, a Slika 30 predstavlja onu prvu. Prvim uvjetom provjerava se može li lik biti udaren, kako se ne bi dogodilo više poziva funkcije dok prva još nije završila. Istinitim slučajem izračunava se ukupna srca koja se trebaju maknuti animacijom iz korisničkog sučelja, te se zdravlje neprijatelja smanjuje za vrijednost iz argumenta „damage“. Petlja „for“ naredbom instancira ukupna srca koja se trebaju maknuti animacijom. Uvjet provjerava je li zdravlje manje ili jednako nuli; ako jest, poziva se funkcija „Death“ (smrt), u suprotnom, pokrenut će knockback Coroutine kako bi simulirao udarac. Nakon oštećenja, vizualni efekt treperenja se aktivira na neprijatelju, a animacija je nadopunjena zvučnim efektom „LogPain“ na udarac.

```
public void Knock(float knockTime, float damage)
{
    if(isTargetable)
    {
        if (!GodMode)
        {
            if (armorManager.activeProtection && armorManager.hasProtection)
            {
                FlashActive = true;
                flashCounter = flashLenght;
                AdrenalinScr.value -= 0.1f;
                StartCoroutine(KnockCo(knockTime));
                armorManager.ArmorHit();
            }
            else if (armorManager.activeProtection == false)
            {
                HeartManager.playerCurrentHealth -= damage;
                FlashActive = true;
                flashCounter = flashLenght;
                PlayerHealthSignal.Raise();
                AdrenalinScr.value -= 0.1f;
                if (HeartManager.playerCurrentHealth > 0)
                {
                    audioManager.Play("PlayerPain");
                    //FindObjectOfType<AudioManager>().Play("PlayerPain");
                    StartCoroutine(KnockCo(knockTime));
                }
                else
                    StartCoroutine(DeathCo());
            }
        }
        else
        {
            myRididbody.velocity = Vector2.zero;
            currentState = PlayerState.idle;
        }
    }
    else
        currentState = PlayerState.idle;
}
```

Slika 31. Playerscr programski kod – 4 dio

Slika 31 prikazuje igračevu metodu Knock. Ova funkcija upravlja reakcijom igrača na pretrpljenu štetu, provjeravajući može li igrač biti ozlijeđen, pri čemu u God Mode-u igrač neće pretrpjeti nikakvu štetu niti će biti „knocked“. Primjena zaštitne opreme, poput zaštitnog prstena, kod igrača s aktivnom zaštitom ne dovodi do smanjenja zdravlja. Dok, ako nema zaštitu, igrač šalje signal drugim objektima na koje utječe, a zatim se njegov „health bar“ ažurira. Ako njegovo zdravlje opadne na manje ili jednako nuli, poziva funkciju DeathCo unutar Unity Coroutine.

```
public virtual void Death()
{
    DeSelect();
    currentSelected = false;
    FindObjectOfType<AudioManager>().Play("DieLog");
    InitHearts2();
    Redirect.Killed(enemyScribtableObject.enemyName);
    SpawnEnemiesArea.currentMinionCount--;
    //remove restriction
    tempVectorSpawnCoin = new Vector3(Random.Range(-1, 1), 0, Random.Range(-1, 1));
    for (counterI = 0; counterI < enemyScribtableObject.gold; counterI++)
    {
        Instantiate(itemInside, transform.position + tempVectorSpawnCoin, Quaternion.identity);
    }
    counterI = Random.Range(1, 10);
    if(counterI <= 2)
    {
        Instantiate(heartSpawn, transform.position + tempVectorSpawnCoin, Quaternion.identity);
    }
    Instantiate(soul, transform.position + new Vector3(0, 0, 0), Quaternion.identity);
    Destroy(this.gameObject);
}
```

Slika 32. EnemyFollowPlayer programski kod – 6. dio

Slika 32 prikazuje Funkciju Death() koja se poziva nakon smrti neprijatelja. Ova funkcija izmjenjuje odabir ovog neprijatelja kao glavnu metu ako je u to vrijeme bio označen i pušta zvuk smrti. „Redirect Killed“ šalje informaciju ako je NPC-eva smrt potrebna za neki zadatak igrača. Petlja instancira novac, te poslije postoji 20% šansa da stvori srce, koje služi igraču za obnavljanje zdravlja. Prije uništavanja objekta, neprijatelj instancira „soul“, odnosno dušu GameObject-a, kao vizualnu animaciju smrti.

```
370     private IEnumerator DeathCo()
371     {
372         GameManager.gameOver = true;
373         FlashActive = false;
374         canMove = false;
375         PlayerSprite.color = new Color(255, 255, 255);
376         myRididbody.velocity = Vector2.zero;
377         animator.SetBool("Death", true);
378         yield return new WaitForSeconds(1.5f);
379         alertPanelScr.showAlertPanel("Game over!");
380         yield return new WaitForSeconds(1f);
381         Time.timeScale = 0;
382     }
```

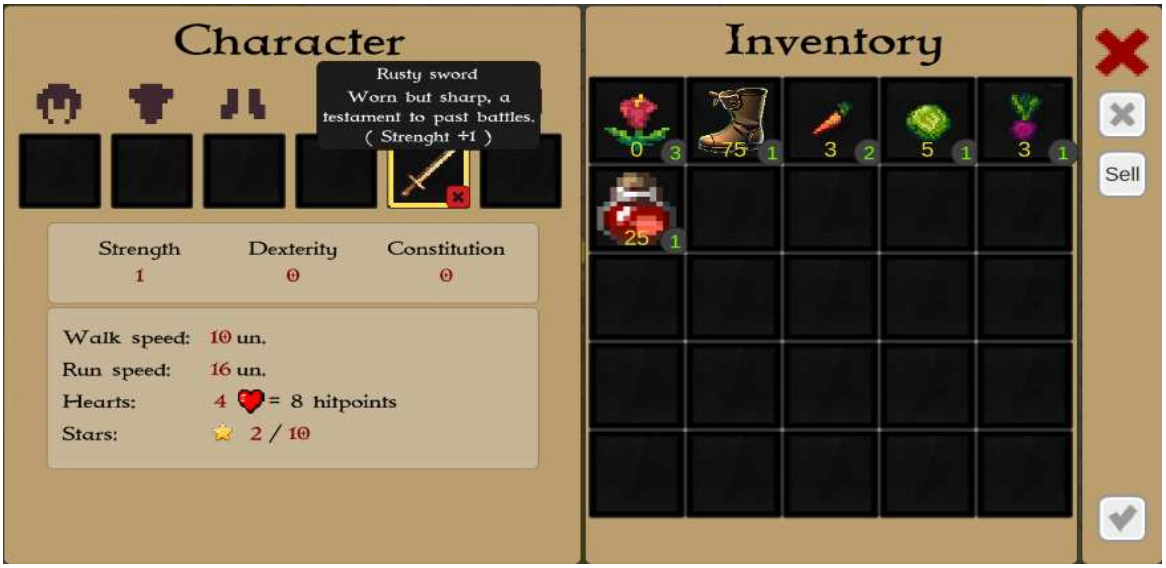
Slika 33. Playerscr programski kod – 5 dio

Slika 33 prikazuje IEnumerator DeathCo koji se poziva kada je trenutno zdravlje manje ili jednako nuli. Statičkoj varijabli *gameOver* postavlja se vrijednost 'true', nakon čega igrač gubi mogućnost kretanja, njegova boja od udara postaje kao zadana, brzina od udarca

se postavlja na nulu, te se pokreće animacija smrti igrača pomoću komponente animator. Nakon 1.5 sekunde, tj. kada animacija smrti igrača u potpunosti završi, prikazuje se panel s tekstom „Game over!“. Na kraju, vrijeme se zaustavlja, što stopira sve objekte i njihove komponente, osim objekta Canvas, koji može igraču ponuditi da krene iznova.

4.8.3 Sustav inventara

Inventar u RPG igrama jedan je od najbitnijih dijelova igre, omogućuje igraču da upravlja svojim predmetima i opremom koje je zadobio tijekom igre. Slika 34 prikazuje Inventar koji se sastoji od ograničenog broja utora koji mogu pohraniti više vrsta predmeta. U projektu postoje predmeti koji se mogu složiti po pet istih na jedan utor i oni koji zauzimaju jedan utor po jednom predmetu. Svaki zauzet utor sadrži sliku pripadajućeg predmeta, broj koliko ih ima složenih i prodajnu cijenu. Isto tako, ako je utor zauzet, „hover“ mišem na taj utor će prikazati opisni tekst tog predmeta. U inventaru igrač ima opciju uništiti predmet iz utora pritiskom na gumb „X“, također postoji i opcija da igrač proda predmet s gumbom „Sell“, ali samo kada je kod trgovca.



Slika 34. Prikaz invertara iz igre

Inventar sadrži maksimalno 25 utora. Ako želi osloboditi mjesto, treba ili uništiti pojedini predmet ili ga prodati. Pristup inventaru je moguć pritiskom slova „i“ na tipkovnici, koje može koristiti u bilo kojem dijelu igre. Važno je napomenuti da otvaranjem inventara

korisnik ne pauzira igru, pa otvaranje inventara tijekom borbe može biti loša ideja. Stavljanje predmeta u inventar moguće je na sljedeće načine: kupnjom predmeta od trgovca, prikupljanjem predmeta s mape i opljačkavanjem škrinje. Prikupljeni predmet stavlja se na prvo slobodno mjesto unutar liste, ako nije predmet koji se slaže ili ako su predmeti koji su složeni došli do limita slaganja.

```
Unity Script | 30 references
public class CreateItem : ScriptableObject
{
    public TypeOfItem Type;
    public TypeOfEquipment TypeOfEquipment;
    public int ID = 0;
    new public string name = "New item";
    new public string description = "Description";
    new public bool pickable;
    new public bool isStackable;
    new public int Price;
    //new public int damageBoost;
    new public int HealthBoost;
    new public int StrenghtBoost;
    new public int ConstitutionBoost;
    new public int DexterityBoost;
    new public float MonsterGoldBonus;
    new public bool nullify = false;
    public Sprite icon = null;
    public bool isDefaultItem = false;
}

19 references
public enum TypeOfItem
{
    HealingPotion,
    Equipment,
    Quest,
    Arrows,
    Plantable,
    Star
}
9 references
public enum TypeOfEquipment
{
    None,
    Weapon,
    Helmet,
    Chest,
    Boots,
    Ring,
    Bow
}
```

Slika 35. Skripta CreateItem zaslužna za kreiranje predmeta

Prikaz „ScriptableObject“-a predmeta prikazuje Slika 35. Svaki predmet sadrži varijable tipa, koji je jedinstveni broj, ime, opis, je li se može pokupiti, složiti, cijena predmeta, ikona i ostale. Kreiranjem novog predmeta preko Unity „ScriptableObject“-a omogućava se da

pomoću Unity sučelja izmijenimo informacije za sve predmete istog imena. Korištenjem ovog sustava olakšavamo upravljanje predmetima, jer jednom izmjenom na ScriptableObjectu ažuriramo sve predmete unutar igre.

```
1 reference
public void Use()
{
    if (CantUse == false)
    {
        if (Type == TypeOfItem.HealingPotion)
        {
            if (HeartManager.playerCurrentHealth < HeartManager.playerMaxHealth)
            {
                if (HeartManager.playerCurrentHealth == HeartManager.playerMaxHealth - 1)
                    HeartManager.playerCurrentHealth += 1;
                else
                    HeartManager.playerCurrentHealth += 2;

                FindObjectOfType<AudioManager>().Play("HealHeart");
                plyScr.PlayerHealthSignal.Raise();
                if (counter1 == 1)
                {
                    Tooltip.SetActive(false);
                    name = null;
                    description = null;
                    img = null;
                    //  sprite.enabled = false;
                    sprite.sprite = null;
                    haveItem = false;
                    if (PlayerInv)
                        invScr.checkSpaceInInventory(1);
                }
                else if (counter1 > 1 && counter1 < 6)
                    counter1--;

            }
        }
        else if (Type == TypeOfItem.Equipment)
            Equip();
    }
}
```

Slika 36. Isječak koda skripte Item

Slika 36 prikazuje isječak skripte Item. Pritiskom na utor s predmetom lijevim klikom miša pozivamo funkciju „Use“. Ako je predmet onaj koji služi kao oprema, taj se predmet oblači, dok predmeti poput napitaka zdravlja nestaju zauvijek. Iskorišteni predmeti oslobađaju mjesto u inventaru.

```
4 references
public void Destroy()
{
    if (haveItem != false)
    {
        if (thisItem.isStackable && counter1 > 1 && thisItem.Type != TypeOfItem.Quest)
        {
            name = null;
            description = null;
            img = null;
            haveItem = false;
            DeleteButton.SetActive(false);
            counter1 = 1;
        }
        else if (thisItem.isStackable && counter1 >= 1 && thisItem.Type == TypeOfItem.Quest)
        {
            name = null;
            description = null;
            img = null;
            haveItem = false;
            DeleteButton.SetActive(false);
            counter1 = 1;
        }
        else
        {
            name = null;
            description = null;
            img = null;
            if (PlayerInv)
                invScr.checkSpaceInInventory(1);
            haveItem = false;
            DeleteButton.SetActive(false);
        }
    }
}
```

Slika 37. Isječak koda skripte Item – 2 dio

Slika 37 prikazuje isječak koda koji stavlja sve vrijednosti utora na null i varijablu „haveitem“ na lažno stanje, što omogućava da inventar skripta prepozna prazan utor.

```
public void AddItem(CreateItem item,    bool loading )
    {
    if (item.Type == TypeOfItem.Star)
    {
        ++starCount;
        starsCountText.text = starCount.ToString();
        Redirect.Gathering(item.name, num, loading);
        if(starCount >= 10)
        {
            descriptionPanelScr.showDescriptionPanel();
          //  alertPanelScr.showAlertPanel("Game finished, demo completed!");
        }
        return;
    }

    if (isFull)
    {
        alertPanelScr.showAlertPanel("No space in inventory");
        return;
    }
    //Ako inv nije pun...
```

Slika 38. Programski kod skripte Inventory – dio 1

```
    for (int  i = 0; i < slot.Length; i++)
    {
        itemScr = slot[i].GetComponentInChildren<item>();
        if (itemScr.haveItem && item.isStackable == false)
        {
            continue;
        }
        else if(itemScr.haveItem && item.isStackable && itemScr.counter1 < 5 && item.name == itemScr.name)
        {
            Debug.Log("gather");
            itemScr.counter1++;
            if (item.Type == TypeOfItem.Quest)
            {
                Debug.Log("salje");
                num++;
                Redirect.Gathering(item.name, num, loading);
            }
            return;
        }
        else if (itemScr.haveItem == false)
        {
            itemScr.thisItem = item;
            itemScr.CurrentNum = i;
            itemScr.name = item.name;
            itemScr.description = item.description;
            itemScr.img = item.icon;
            itemScr.Type = item.Type;
            itemScr.haveItem = true;
            Pun++;
            if (item.Type == TypeOfItem.Quest)
            {
                num++;
                Redirect.Gathering(item.name,num, loading);
            }
            if (Pun == spaceInInventory)
            {
                isFull = true;
                Debug.Log("pun");
            }
            return ;
        }
    }
}
```

Slika 39. Programski kod skripte Inventory – dio 2

Slika 38 i Slika 39 prikazuju funkciju „AddItem“ koja filtrira što učiniti s predmetom. Drugi dio inventara čini praznih 6 utora, namijenjenih za igračevu opremu, koji imaju utjecaj na mehaniku igre. Utori za opremu imaju poseban tip predmeta koji mogu primiti, u jednom trenutku na jednom utoru moguće je imati samo jedan predmet odgovarajućeg tipa. U slučaju da igrač želi staviti drugi predmet kao trenutnu opremu, igraču se uklanjaju koristi predmeta te se taj predmet dodaje u inventar, dok drugi predmet zauzima taj utor.

4.8.4 Napredak

Primarni cilj igre je prikupiti svih 10 zvijezda koje su skrivene širom mape kao što je prikazano na Slika 40. Osim toga, određeni broj zvijezda može se dobiti porazom moćnog neprijatelja.

Slika 40. Prikaz prikupljanja predmeta za napredak

4.8.5 Interakcija s NPC-evima, dijalog i zadaci

Dijalog je važan element u svakoj igri, jer pomaže u razvoju priče i stvaranju veze između likova i igrača. U 2D akcijskoj avanturi, dijalog se koristi za davanje uputa, predstavljanje likova i razvoj priče, što je prikazano na Slika 41.

Slika 41. Prikaz dijaloga iz igre

```
Unity Message | 1 reference
public virtual void OnTriggerEnter2D(Collider2D other)
{
    if (other.CompareTag("Player") && other.isTrigger)
    {
        playerInRange = true;
        Vector2 directionToPlayer = ((Vector2)other.transform.position - (Vector2)transform.position).normalized;
        anim.SetFloat("x", directionToPlayer.x);
        anim.SetFloat("y", directionToPlayer.y);
    }
}
```

Slika 42. Programski kod skripte NpcScr– 1 dio

Prilaskom NPC liku, igrač ulazi u zonu interakcije, koju prikazuje Slika 42. NPC se okreće prema igraču i stavlja varijablu „playerInRange“ na istinu.

```
public virtual void Interact()
{
    if (playerInRange == true)

        if (Input.GetKeyDown(KeyCode.E))
        {
            dialogBoxScr.showDialog(dialogTekst, name);
        }
}
```

Slika 43. Programski kod skripte NpcScr– 2 dio

Pritiskom tipke „E“ sa tipkovnice, pokreće se dijalog između igrača i tog NPC-a kao što prikazuje isječak Slika 43.

```
2 references
public void showDialog(string text, string name)
{
    talking = true;
    rectTransform.anchoredPosition = new Vector3(0, 0, 0);
    Shop = false;
    acceptButton.SetActive(false);
    rejectButton.SetActive(false);
    placeHolder = text.ToString();
    tmProNameText.text = name;
}
```

Slika 44. Programski kod skripte NpcScr– 3 dio

Slika 44 prikazuje isječak koda koji postavlja varijablu „talking“ na istinu, kako se ne bi moglo pokrenuti više dijaloga u isto vrijeme. RectTransform postavlja korisničko sučelje dijaloga na početnu poziciju gdje je vidljivo igraču. Također prikazuje tekst koji je funkcija prima od NPC-a. Kao i većina RPG-ova, ova igra također sadrži zadatke koje igra omogućuje rješavati. Korisničko sučelje tijekom dijaloga prikazuje dva gumba koja služe za prihvaćanje ponude ili ne. Ako ako je zadatak prihvaćen, taj zadatak će se prikazati unutar sučelja „Quests“.

Slika 45. Prikaz UI sučelja za upravljanje zadacima iz igre

Sučelje za zadatke prikazano na Slika 45 otvara se pritiskom slova „L“ na tipkovnici. Prihvaćeni zadaci imaju svoj panel gdje se može vidjeti ime zadatka, njegov kratak opis i progresija tog zadatka. U slučaju završetka zadatka, sučelje mijenja boju te obavještava igrača da je zadatak završen. Također, tijekom ažuriranja zadataka igrač je kratko upoznat s pop-up panelom. Kao i predmeti, svaki zadatak je kreiran pomoću

„ScriptableObject“. NPC prima referencu na dodijeljen zadatak; u slučaju prihvaćenog zadatka, isti se dodjeljuje igraču uz AddQuest funkciju.

```
2 references
public void AddQuest(Create_Quest quest, NpcQuestScr npcScr, int currentCount)
{
    QuestObject questObjectTemp1 = new QuestObject();
    Debug.Log("Dodaj quest: " + quest.name);
    GameObject panel = Instantiate(panelPrefab, new Vector3(0, 0, 0), Quaternion.identity) as GameObject;
    panel.transform.SetParent(parentGameObject.transform);
    panel.transform.localScale = new Vector3(0.75f, 0.75f, 1);
    questObjectTemp1.quest = quest;
    questObjectTemp1.Scr = npcScr;
    questObjectTemp1.completedQuest = false;
    questObjectTemp1.activeQuest = true;
    QuestController questController = panel.gameObject.GetComponent<QuestController>();
    questObjectTemp1.panelScr = questController;
    Debug.Log(currentCount);
    questObjectTemp1.counter = currentCount;
    questObjects.Add(questObjectTemp1);
    //panel texts
    panel.SetActive(true);
    questController.nameOfQuest.text = quest.name;
    questController.description.text = quest.description.ToString();
    questController.progression.text = questObjectTemp1.counter.ToString();
    questController.activeQuest = true;
    questController.progressionFull.text = quest.count.ToString();
    if (questObjects.Count > 5)
    {
        sizeOfContainerBottom = rectTransform.offsetMin.y - 70f;
        rectTransform.offsetMin = new Vector2(rectTransform.offsetMin.x, sizeOfContainerBottom);
    }
    if (questObjectTemp1.counter >= questObjectTemp1.quest.count)
    {
        Debug.Log("finished quest");
        questObjectTemp1.quest.Finished = true;
        questObjectTemp1.Scr.NpcQuest.Finished = true;
        questObjectTemp1.completedQuest = true;
        questObjectTemp1.panelScr.QuestCompleted();
    }
}
```

Slika 46. Programski kod skripte RedirectQuest– 1 dio

Slika 46 prikazuje funkciju koja stvara novi panel u sučelju za zadatke te ga pozicionira na zadnje mjesto u listi. Taj panel ima iste informacije kao i dobiveni zadatak te prati njegov napredak. Ako postoji više od pet panela, odnosno pet zadataka već u listi, uvjetom ga sakrivamo ispod zadnjeg panela, a korisnik bi trebao listati s mišem da se panel otkrije. Sljedeći uvjet provjerava je li trenutni brojač (counter) ovog zadatka već isti kao što zadatak traži, ako je istina, onda se postavlja da je zadatak završen.

```
public void Killed(string name)
{
    Debug.Log("killed " + name);
    questObjectTemp = questObjects.Find(p => p.quest.Target == name);
    if( questObjectTemp!=null)
    {
        if (questObjectTemp.quest.count > questObjectTemp.counter)
        {
            questObjectTemp.counter++;
            questObjectTemp.panelScr.progression.text = questObjectTemp.counter.ToString();
            tempText = questObjectTemp.counter + " / " +  questObjectTemp.quest.count;
            ProgressUpdate(questObjectTemp.quest.name, tempText,false);
            if (questObjectTemp.counter >= questObjectTemp.quest.count )
            {
                Debug.Log("finished quest");
                questObjectTemp.quest.Finished = true;
                questObjectTemp.Scr.NpcQuest.Finished = true;
                questObjectTemp.completedQuest = true;
                questObjectTemp.panelScr.QuestCompleted();
                ProgressUpdate(questObjectTemp.quest.name,"Completed!", false);
            }
        }
    }
}
```

Slika 47. Programski kod skripte RedirectQuest– 2 dio

Slika 47 prikazuje funkciju komponente predmeta, koja provjerava je li neprijatelj tražen za neki zadatak, ako jest, povećava mu brojač. U slučaju da je tada riješen zadatak, obavještava NPC zaduženog lika i korisničko sučelje.

```
public void QuestConversation()
{
    if (!questTaken)
    {
        dialogBoxScr.currentQuestGiver = this.gameObject;
        dialogBoxScr.placeHolder = this.NpcQuest.description.ToString();
        dialogBoxScr.acceptButton.SetActive(true);
        dialogBoxScr.rejectButton.SetActive(true);
    }
    else if (questTaken && !NpcQuest.Finished && questEnded == false)
        dialogBoxScr.placeHolder = this.NpcQuest.alreadyTakenQuest.ToString();

    else if (questTaken && this.NpcQuest.Finished && !questEnded)
    {
        dialogBoxScr.placeHolder = NpcQuest.completedQuest.ToString();
        PlayerScr.Gold += NpcQuest.money;
        questEnded = true;
        emoteSprite.enabled = false;
        redirectScr.DeleteQuest(NpcQuest.name);
        gameManager.addInQuestList(assignedId, questTaken, questEnded);
        if (NpcQuest.Type == TypeOfQuest.Gathering)
            invScr.DeleteItemAfterQuestEnded(NpcQuest);
    }
}
```

Slika 48. Programski kod skripte NpcQuest

Slika 48 prikazuje funkciju koja se poziva svakom interakcijom s NPC-em koji ima zadatak. Prvi uvjet provjerava je li zadatak već prihvaćen. Ako nije, dijalog se mijenja na temelju stanja NPC-a, te se prikazuju gumbi za interakciju. Drugi uvjet provjerava je li zadatak već prihvaćen, ali ne dovršen. U slučaju istine, Dijalog se mijenja kako bi došlo do provjere o napretku zadatka. Zadnji uvjet ispunjen je ako je zadatak riješen i igrač već nije bio u ovom uvjetu. U ovom slučaju, NPC označava da je zadatak riješen, nagrađuje igrača novcem, te šalje informaciju sučelju da se obriše trenutni zadatak iz liste. Ako je tip zadatka bilo sakupljanje nečega, NPC poziva funkciju iz invertera da briše predmete koji su bili potrebni za zadatak.

4.8.6 Sustav ekonomije

Sustav ekonomije uveden je u igru kako bi se poboljšala složenost igre u skladu s pravilima RPG žanra. To znači da igrač može sakupljati novac, kupovati, prodavati s trgovcima i zarađivati na zadacima. Prikupljanje novca prikazano je na Slika 49

Slika 49. Prikaz skupljanja novca u igri

```
public class ShopNPC : NpcScr
{
    [Header("Shop")]
    public CreateItem[] itemsInStock = new CreateItem[5];
    public shopInventory ShopInvScr;
    public static int id;
    private int CounterOfItems;
   // private bool talking = false;
```

Slika 50. Programski kod skripte ShopNPC– 1 dio

Isječak koda sa Slika 50 prikazuje ShopNPC skriptu koja nasljeđuje od originalne „NpcScr“. Skripta predstavlja trgovca koji ima ponudu predmeta koje posjeduje, a ti predmeti imaju svoje cijene. Predmeti se dodaju unutar Unity sučelja i statični su tijekom cijele igre.

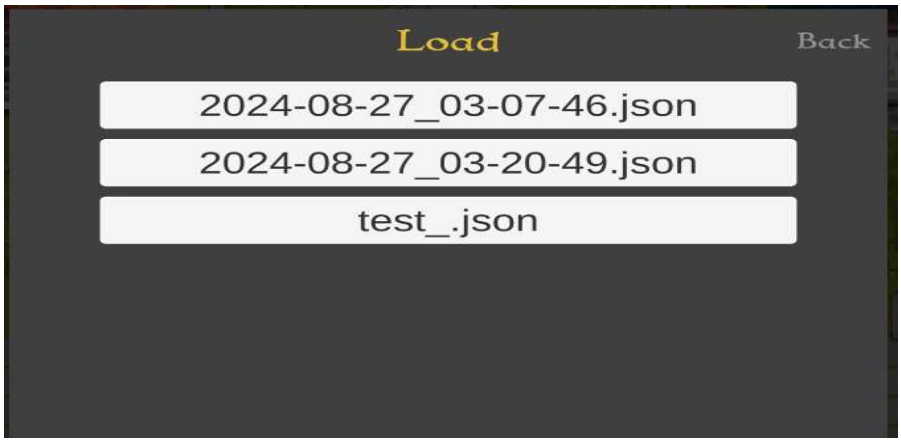
```
public override void Interact()
{
        if (Input.GetKeyDown(KeyCode.E))
        {
            //talking = true;
            dialogBoxScr.currentQuestGiver = this.gameObject;
            dialogBoxScr.showDialogShop(dialogTekst, name);
            ShopInvScr.DeleteItems();
            for (int i = 0; i < CounterOfItems; i++)
            {
                ShopInvScr.AddItem1(itemsInStock[i], i);
            }
        }
}
```

Slika 51. Programski kod skripte ShopNPC– 2 dio

Slika 51 prikazuje funkciju interakcije s igračem. Poziva se modificirana nadjačana funkcija „Interact“ koja, umjesto dijaloga, prikazuje gumbe. Ti gumbi provjeravaju želi li korisnik otvoriti UI sučelje za trgovinu, a ako je to slučaj, funkcija obnavlja zalihu predmeta.

4.8.7 Sustav spremanja i učitavanja igre

Značajke spremanja i učitavanja u igrama bitne su za očuvanje napretka, nudeći fleksibilnost i sprječavajući frustraciju ponavljanja ili gubitka.



Slika 52. Prikaz ponuđenih JSON datoteka za učitavanje napretka iz igre

```
public void SavetoJson(Vector3 playerPosition, bool godmode, float health, float gold, int arrows, int stars,
     List<ChestObject> chestList, List<PotObject> potlist, bool passage, List<ItemsOnGroundObject> pickUpItem
{
    try
    {
        GameData data = new GameData();
        data.spawnPosition = playerPosition;
        data.godMode = godmode;
        data.difficulty = diff;
        // Add to adjust to diff
        data.currentHealth = health;
        data.gold = gold;
        data.arrows = arrows;
        data.stars = stars;
        data.items = items;
        data.equipment = equipment;
        data.camMaxPosition = mapScrObject.maxPosition;
        data.camMinPosition = mapScrObject.minPosition;
        data.chests = chestList;
        data.pots = potlist;
        data.canPass = passage;
        data.pickUpItems = pickUpItemList;
        data.quests = questObject;
        data.bossDefeated = BossAi.bossDefeated;

        string dateTimeFormat = "dd/MM/yyyy HH:mm:ss";
        timestamp = DateTime.Now.ToString(dateTimeFormat, CultureInfo.InvariantCulture);
        timestamp = ConvertToFileName(timestamp);

        if (!string.IsNullOrEmpty(recordName))
        {
            data.recordName = recordName;

            string json = JsonUtility.ToJson(data, true);
            currentGameRecord = Path.Combine(Application.dataPath, "saves", data.recordName + ".json");
            File.WriteAllText(currentGameRecord, json);
        }
        else
        {
            timestamp = DateTime.Now.ToString(dateTimeFormat, CultureInfo.InvariantCulture);
            timestamp = ConvertToFileName(timestamp);
            data.recordName = timestamp;
            string json = JsonUtility.ToJson(data, true);
            currentGameRecord = Path.Combine(Application.dataPath, "saves", timestamp + ".json");
            File.WriteAllText(currentGameRecord, json);
        }

    }
    catch (Exception e)
    {
        alertPanelScr.showAlertPanel("Failed to start a new game!");
        BugLogger(e);
    }
}
```

Slika 53. Programski kod skripte SaveorLoad – 1 dio

Za spremanje napretka igre zaslužna je funkcija SaveToJson, koja se poziva samo na određenim dijelovima mape, koje su podalje od borbe. Ako se pokrene nova igra, spremanje se automatski poziva pri učitavanju nove scene. Slika 53 prikazuje isječak koda u kojem je „GameData“ objekt sa svim podacima koji su relevantni za spremanje u json datoteku. Svi argumenti tijekom poziva dodjeljuju se objektu „GameData“. Ako

korisnik nije sam nazvao ime ove sesije, igra sama dodjeljuje trenutnu vremensku oznaku kao ime ove sesije. Zapisivanje u json datoteku omogućava naredba File.WriteAllText koja prepisuje trenutne podatke na podatke koji već postoje unutar te datoteke. Ukoliko se dogodi greška bilo gdje tijekom ove funkcije, igra obavještava korisnika o grešci te zapisuje grešku u tekstualnu datoteku.

```
283     private IEnumerator Loading(string name)
284     {
285         loading = true;
286         player.triggerBox.enabled = false;
287         yield return Frames(2);
288         player.triggerBox.enabled = true;
289         string json;
290         alertPanelScr.showAlertPanel("Loaded");
291         json = File.ReadAllText(Application.dataPath + "/saves/" + name);
292         filePath = Application.dataPath + "/saves/" + name;
293         currentGameRecord = filePath;
294         Debug.Log("load-ao "+ currentGameRecord);
295 
296         if (File.Exists(filePath))
297         {
298             GameData data = JsonUtility.FromJson<GameData>(json);
299             player.transform.position = data.spawnPosition;
300             PlayerScr.GodMode = data.godMode;
301             //record name
302             // Diff
303             equipment.LoadEquipment(data.equipment);
304             HeartManager.playerCurrentHealth = data.currentHealth;
305             PlayerScr.Gold = data.gold;
306             PlayerScr.Arrows = data.arrows;
307             Inventory.starCount = data.stars;
308             inventory.LoadInventory(data.items);
309 
310             cam.MapTransfer(data.camMinPosition, data.camMaxPosition);
311             manager.loadChests(data.chests);
312             //manager.loadPlant(data.plantList);
313             manager.loadPots(data.pots);
314             manager.Passage(data.canPass);
315             manager.loadPickUpItems(data.pickUpItems);
316             manager.loadQuests(data.quests);
317             player.loadPlayer();
318             Boss.Load(data.bossDefeated);
319             SpawnEnemies.defaultDifficulty = data.difficulty;
320             RunningScr.value = 1;
321 
322         }
323         else
324         {
325             Debug.LogError("There are no save files");
326         }
327     }
```

Slika 54. Programski kod skripte SaveorLoad – 2 dio

Funkcija „loading“ prikazana na Slika 54 poziva se kada igrač odabere napredak koji želi očitati. Funkcija osigurava se da se samo jedanput u jednom trenutku može očitavati napredak. Nakon provjere postoji li datoteka, dohvaća se json format iz datoteke i zapisuje unutar objekta GameData. Sve varijable se onda ažuriraju s ovim spremljenim podacima.

```
public List<ItemObject> SaveInventory()
{
    List<ItemObject> items = new List<ItemObject>();
    for ( i = 0; i < slot.Length; i++)
    {
        itemScr = slot[i].GetComponentInChildren<item>();
        if (itemScr.haveItem)
        {
            ItemObject itemObject = new ItemObject();
            itemObject.assign(itemScr.thisItem.ID, itemScr.counter1);
            items.Add(itemObject);
        }
    }
    return items;
}
```

Slika 55. Programski kod skripte Inventory – dio 3

Isječak koda sa Slika 55 prikazuje funkciju za spremanje inventara. Funkcija SaveInventory kreira listu objekata predmeta i dodaje svaki predmet koji trenutno postoji toj listi.

```
1 reference
public void LoadInventory(List<ItemObject> items)
{
    starsCountText.text = starCount.ToString();
    wipeInvenory();
    foreach (ItemObject item1 in items)
    {
        for (i = 0; i < allItems.Count; i++)
        {
            //Debug.Log(item1.id + " , " + item1.count);
                if (item1.id == allItems[i].ID)
                {
                    for(int j=0;j< item1.count; j++)
                    {
                        AddItem(allItems[i], true);
                    }
                    break;
                }
        }
```

Slika 56. Programski kod skripte Inventory – dio 4

Primjer jednog ažuriranja je iz skripte inventara prikazana na Slika 56. Svi predmeti koji već postoje u inventaru brišu se pozivom funkcije „wipeInventory“, a zatim petlja prolazi kroz listu objekata predmeta iz argumenta te svaki predmet dodaje u inventar.

5. Zaključak

Cilj ovog rada opisuje proces razvoja 2D akcijske avanture u Unity okruženju, obuhvaćajući sve ključne korake u razvoju videoigre, od početne ideje do konačne realizacije. Unity je fleksibilan, svestran i intuitivan alat za razvoj igara, koji se pokazao prikladnim za kreiranje indie igara, omogućujući implementaciju raznih tehnika i vizualnih elemenata. Unity okruženje dolazi s opsežnom dokumentacijom koja pokriva sve aspekte korištenja programa, a postoji i značajan broj kvalitetnih vodiča na platformama poput YouTubea, koje pružaju indie kreatori. Korištenje programskog jezika C# u Unity editoru olakšalo je proces programiranja igre zbog njegove objektno orijentirane prirode, dok je uključivanje vanjskih alata poput softvera za izradu pikselizirane umjetnosti poboljšalo vizualni aspekt igre. Dodatno, korištenje IDE Visual Studio doprinijelo je učinkovitosti i kvaliteti programiranja tijekom izrade igre. Iako je moguće samostalno kreirati pojedine dizajne, većina pikselizirane umjetnosti preuzeta je s repozitorija kao što je OpenGameArt, koji je namijenjen za projekte besplatnih i otvorenih videoigara.

Iako Unity nudi širok raspon alata i mogućnosti, izrada kvalitetne pikselne umjetnosti, animacija i zvučnih elemenata zahtijeva znatno više vremena i stručnih vještina nego što je bilo raspoloživo u okviru ovog projekta. Zbog ograničenog vremena dostupnog za razvoj, neke naprednije funkcionalnosti i detaljno dotjerivanje igre nisu bile u potpunosti realizirane kako je prvobitno planirano, te razina kvalitete igre nije jednaka komercijalnim proizvodima. Unatoč tome, projekt je pokazao da je izrada indie igre izvediva s dostupnim alatima i znanjem, te je planirani ciljevi razvoja su u većini ispunjeni. Budući razvoj igre trebao bi biti usmjeren na proširenje mape, složenije AI sustave, unapređenje borbenih mehanika te poboljšanje vizualnog i zvučnog dizajna. Nadalje, kroz proces izrade igre prepoznate su metode koje nisu optimalne zbog loše optimizacije i potencijalnih grešaka, a koje bi u budućnosti trebalo preurediti.

Ovaj rad postavlja temelj za daljnji razvoj i proširenje igre, usmjeren na maksimalno iskorištavanje Unity alata u razvoju kreativnog i privlačnog iskustva igranja. Brzi tehnološki napredak i rastuće mogućnosti industrije videoigara otvaraju prostor za buduće inovacije i poboljšanja.

2D akcijska avantura Github: https://github.com/RikiVu/2D-akcijska-avantura

6. Literatura

[1] Evoland, pristup: rujan 2024,
https://evoland.fandom.com/wiki/Evoland_Wiki

[2] Wikipedia, pristup: rujan 2024,
https://en.wikipedia.org/wiki/The_Legend_of_Zelda:_A_Link_to_the_Past

[3] Zelda.fandom, pristup: rujan 2024,
https://zelda.fandom.com/wiki/The_Legend_of_Zelda:_A_Link_to_the_Past

[4] Zenva (2023.), pristup: rujan 2024,
https://gamedevacademy.org/what-is-unity/

[5] Wikipedia, pristup: rujan 2024,
https://en.wikipedia.org/wiki/Unity_(game_engine)

[6] Unity Technologies, pristup: rujan 2024,
https://docs.unity3d.com/Manual/UsingTheEditor.html

[7] Unity Technologies, pristup: rujan 2024,
https://docs.unity3d.com/Manual/class-GameObject.html

[8] Unity Technologies, pristup: rujan 2024,
https://docs.unity3d.com/ScriptReference/Component.html

[9] Unity Technologies, pristup: rujan 2024,
https://docs.unity3d.com/Manual/class-MonoBehaviour.html

[10] Unity Technologies, pristup: rujan 2024,
https://docs.unity3d.com/ScriptReference/SerializeField.html

[11] Unity Technologies, pristup: rujan 2024,
https://docs.unity3d.com/Manual/class-ScriptableObject.html

[12] Unity Technologies, pristup: rujan 2024,
https://docs.unity3d.com/ScriptReference/Camera.html

[13] Unity Technologies, pristup: rujan 2024,
https://docs.unity3d.com/Manual/class-Mathf.html

[14] Unity Technologies, pristup: rujan 2024,
https://docs.unity3d.com/Manual/AudioOverview.html

[15] Unity Technologies, pristup: rujan 2024,
https://unity.com/topics/what-is-2d-animation

[16] Wikipedia, pristup: rujan 2024,
https://en.wikipedia.org/wiki/Animation
[17] Unity Technologies, pristup: rujan 2024,
https://docs.unity3d.com/Manual/class-AnimatorController.html
[18] Maglione, Sandro , pristup: rujan 2024,
https://www.sandromaglione.com/articles/how-to-create-a-pixel-art-tileset-complete-guide
[19] Unity Technologies, pristup: rujan 2024,
https://docs.unity3d.com/Manual/Tilemap-Palette.html
[20] Unity Technologies, pristup: rujan 2024,
https://docs.unity3d.com/Manual/Tilemap.html
[21] Churchville, Fred, pristup: rujan 2024,
https://www.techtarget.com/searchapparchitecture/definition/user-interface-UI
[22] Technologies, Unity , pristup: rujan 2024,
https://docs.unity3d.com/2020.1/Documentation/Manual/UICanvas.html
[23] Technologies, Unity , pristup: rujan 2024,
https://unity.com/how-to/unity-ui-optimization-tips
[24] Technologies, Unity , pristup: rujan 2024,
https://docs.unity3d.com/ScriptReference/Rigidbody2D.html
[25] Technologies, Unity , pristup: rujan 2024,
https://docs.unity3d.com/ScriptReference/Collider2D.html
[26] Technologies, Unity , pristup: rujan 2024,
https://docs.unity3d.com/2022.2/Documentation/Manual/rigidbody2D-body-types.html
[28] Technologies, Unity , pristup: rujan 2024,
https://docs.unity3d.com/ScriptReference/Vector3.Normalize.html
[29] geekforgeeks, pristup: rujan 2024,
https://www.geeksforgeeks.org/a-search-algorithm/
[30] Sebastian Lague, Youtube, pristup: rujan 2024,
https://www.youtube.com/watch?v=-L-WgKMFuhE&list=PLFt_AvWsXl0cq5Umv3pMC9SPnKjfp9eGW
[31] Granberg, Aron, pristup: rujan 2024, https://arongranberg.com/astar/docs/

Popis slika