Izrada strateške videoigre na poteze u programskom alatu Unity

Blažek, Ivan

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Ivan Blažek

Izrada strateške videoigre na poteze u programskom alatu Unity

DIPLOMSKI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Ivan Blažek

JMBAG: 0016133458

Studij: Baze podataka i baze znanja

Izrada strateške videoigre na poteze u programskom alatu Unity

DIPLOMSKI RAD

Mentor:

Doc. dr. sc. Mladen Konecki

Varaždin, srpanj 2023.

Ivan Blažek

Izjava o izvornosti

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

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

U okviru ovog diplomskog rada postavljen je cilj izrade minijaturne videoigre na poteze koristeći alat Unity. Kroz rad se izlažu teorijske definicije općenito o videoigrama te alatima koji su omogućili sveukupni proces izrade. Osim naglaska na videoigru bitan je i stil crtanja koji se koristi tokom izrade poznat pod nazivom isometric. U ovome radu objedinjuju se više industrija kao što su crtanje, dizajniranje, programiranje i mnogi drugi što u međusobnoj interakciji omogućuju izradu videoigre.

Na praktičnom primjeru prikazat će se izrada komponenti videoigre kao što su mapa, likovi, kamera, glazba, izbornik i sl. Kroz primjer sve komponente budu opisane, demonstrirane i programski implementirane. Važno je napomenuti da metodologija izrade prezentirana u radu predstavlja samo jedan od mogućih pristupa, s obzirom na postojanje različitih algoritama i strategija rješavanja problema.

Rezultat diplomskog rada je funkcionalna minijaturna videoigra koja obuhvaća jedan scenarij igre. Osim toga, naglašava se da je programsko rješenje razvijeno modularno, pružajući temelj za budući razvoj igre.

Ključne riječi: videoigra, Unity, strateška, na poteze, Aseprite, akcije, metode, isometric

SADRŽAJ:

1. UVOD

Tema ovog diplomskog rada je *Izrada strateške videoigre na poteze u programskom alatu Unity*. Svrha rada je prikazati proces izrade videoigre kojoj se radnja odvija na poteze kao što je to u primjerice šahu. Svaki sudionik ima jedan potez dok ne dođe opet red na njega. U ovom diplomskom radu pojam videoigra podrazumijeva računalni program koji kroz interaktivni pristup omogućava korisnicima zabavu na način da se sadržaj videoigre prikazuje putem monitora ili sličnog ekrana poput mobitela i televizora. U ovom radu biti će objašnjene neke osnove alata u kojem je izrađena videoigra. Alat Unity jedan je od najkorištenijih programskih alata za izradu videoigara, također besplatan i obogaćen zajednicom koja pruža ovom alatu mogućnosti koje programeri videoigara svakodnevno koriste.

Iz definirane teme proizlazi cilj ovoga rada koji podrazumijeva prikazati i obuhvatiti sve industrije koje su potrebne za izradu videoigre. Videoigra uključuje više obuhvatnih sadržaja koji daju bogatstvo jednoj videoigri kojih korisnici nisu ni sami svjesni, a čine ih ljudi poput pisaca radnje videoigre, dizajnera sadržaja, izrada i prerada video zapisa, skladanje, pisanje i izvedba glazbe pa sve do programera i programskih inženjera koji izrađuju arhitekturu i stvaraju programsku logiku u pozadini.

U ovom radu bit će prikazane samo osnove svakog dijela izrade. Cilj nije napraviti veliku, uspješnu i profitabilnu videoigru već prikazati kako industrija izgleda u minijaturnom izdanju. Cilj je prikazati kako jedan način izvedbe videoigre na poteze zahtjeva mnogo truda i vremena.

Na početku rada bit će objašnjeni neki od alata koji su korišteni tijekom izrade videoigre, različite vrste videoigara te će u ostatku rada biti naglasak na praktičan dio.

Ovaj diplomski rad uz uvod i zaključak je sastavljen od dodatnih tri poglavlja. Prvo poglavlje uvod bude imalo naglasak na upoznavanje s radom. Drugo poglavlje bude upoznavanje s alatima koji su korišteni kroz izradu praktičnog djela diplomskog rada. Treće poglavlje bude pokrilo definiciju i teorijsku podlogu videoigara. U četvrtom poglavlju budu opisani izrada videoigre i programski kodovi te koraci koji su potrebni da bi se videoigra izradila. Peto i posljednje poglavlje bude zaključak koje donosi sažetak i osvrt na sveukupni rad.

2. METODE I TEHNIKE RADA

Za izradu diplomskog rada korišteni su mnogi alati bez kojih ovaj rad ne bi bio moguć, a neki su korišteni kako bi poduprli i olakšali razvoj videoigre. Za izradu programskog dijela videoigre korišteni su primarno dva alata Unity i Microsoft Visual Studio. Za spremanje programskog koda, pregled i verzioniranje korišteni su alati Git i Github. Za izradu materijala i elemenata videoigre korišten je alat Aseprite.

2.1. Microsoft Visual Studio

Microsoft Visual Studio alat je koji služi za uređivanje i pisanje programskog koda. Videoigra koja je rađena u alata Unity koristi programski jezik C#, a izvrsno surađuje i komunicira sa Microsoft Visual Studio kojem je glavna primjena uređivanje i pisanje u programskom jeziku C#. Visual Studio dostupan je besplatno na svim većim operacijskim sustavima poput Windows i MacOS. Na Linux operacijskim sustavima dostupan je kroz drugi alat Microsoft Visual Studio Code koji je također bogat za uređivanje programskog koda, ali nije toliko specijaliziran kao Visual Studio za C#. (Microsoft Visual Studio, bez dat.)

2.2. Git i GitHub

Git je besplatni distribuirani sustav za kontrolu verzija programskog koda, što znači da prati svaku promjenu koja je napravljena nad datotekama. Služi kako bi se programska rješenja mogla istovremeno povezati na više računala, kako bi se programski kod lakše kordinirao i sinkronizirao. Dizajniran je za brzu i učinkovitu obradu od manjih do velikih projekata. Omogućuje stvaranje i odvajanje dijelova programskih kodova na više neovisnih cjelina kako bi se paralelno moglo raditi na neovisnim komponentama programskog rješenja te na kraju povezali u jednu zajedničku cjelinu tj. Verziju rješenja. Promjene koje se događaju unutar tih cjelina i sinkronizacija koja se događa je vrlo brza i omogućava za lakši rad unutar većih timova. (Git, bez dat.)

GitHub je platforma i servis temeljen na oblaku koji služi za pohranjivanje, upravljanje, praćenje i kontrolu programskog koda uz pomoć Git-a. Github omogućuje korisnicima preglednost i međusobnu komunikaciju kad su u pitanju programska rješenja koja se nalaze na platformi. Servis je besplatan te je pogodan za spremanje projekata otvorenog koda. Slična platforma GitHub-u je i GitLab, no za potrebe ovog rada koristi se GitHub u svrhu očuvanja i preglednosti programskog rješenja. (Kinsta, bez dat.)

2.3. Aseprite

Aseprite je alat koji služi za crtanje primarno u stilu 8-bit i 16-bit, koristi se za pixel-art, retro stil te omogućava i 2D animacije. Unutar ovog rada Aseprite korišten je u svrhu izrade likova i kockica koje se koriste za izgradnju mape odnosno svijeta u kojem se igra odvija. Aseprite je alat koji se plaća, ima bogatu zajednicu te jako puno edukacijskog sadržaja koji su besplatni. (Aseprite, bez dat.)

2.4. Unity

Unity je specijalizirani alat koji omogućuje izradu više platformskih videoigri (eng. *cross-platform game engine*). Unity se koristi za izradu svih vrsta videoigara poput 2D, 3D, virtualna stvarnost, mobilne igre i slično. Alat pruža jednostavno sučelje preko kojeg se videoigra izrađuje na princip da se gotovi dijelovi pozicioniraju na scenu i preko njega se igra pokreće dok se programska logika nalazi u pozadini te preko Visual Studio razvija. Unity omogućuje kreiranje objekata, skripti i jednostavnih načina ponašanja dok dodatnu logiku pišemo unutar Visual Studio programskim jezikom C#, također vrlo je jednostavno već napravljene objekte iz Aseprite prebaciti u Unity kako bi ih iskoristili. Unity je besplatan, svjetski poznat i korišten alat unutar industrije videoigara. Bogata zajednica omogućava jako puno besplatnih edukacijskih sadržaja, materijala, dodatnih alata i sadržaja koji su drugi napravili kako bi novi korisnici već postojeće elemente integrirali u svoju videoigru. (Unity, bez dat.)

3. TEORIJSKA POZADINA VIDEOIGARA

Videoigre su računalne igre s kojima korisnik ima interakciju uz pomoć nekog medija poput mobitela, tipkovnice, raznih upravljačkih uređaja i sl. Videoigre primarno se koriste za zabavu s mogućnostima profesionalnog i natjecateljskog igranja, a u slučaju natjecateljskog korištenja videoigara pojavljuje se i izraz Esport.

Na tržištu videoigre obuhvaćaju niz industrija kako bi se proizvela što bogatija videoigra kojoj je cilj korisniku pružiti što bolje iskustvo tokom igranja. Cilj je proizvesti niz pozitivnih emocija kao što su osjećaji uspjeha, sreće, znatiželje i ispunjenosti. Na platformi za distribuciju videoigara Steam je 2022. godine objavio oko 10000 videoigara te se pokazuje porast svake godine u objavljivanju više videoigara. (Statista, bez dat.)

3.1. Strateška videoigra na poteze

Strateške videoigre se baziraju na sposobnost igrača da planiraju i taktički razmišljaju kako bi došli do pobjede. Igrač treba biti u smirenom stanju, kalkulirano, logički i strateški razmišljati kako bi postigao željeni rezultat unutar videoigre. Većina strateških videoigara se odvija u stvarnom vremenu pri čemu svaki sudionik donosi odluke istovremeno dok stratešku videoigru na poteze karakterizira čekanje svog poteza kako bi sudionik napravio svoj potez. Primjeri strateških igara koji se odvijaju u stvarnom vremenu su Starcraft, Age of Empires, Command & Conquer.

Cilj strateških videoigara na poteze je omogućiti igraču da analitički donese odluku na temelju trenutne situacije u videoigri, raspodjeli resurse, pregleda situaciju i tek kad donese željenu odluku donese kraj svome potezu. Takva videoigra može rezultirati edukacijski a ne samo zabavni element kako bi pružila korisnicima spoznaju da odluke imaju znatno drugačije posljedice. Primjeri takvih videoigara su šah, civilization, XCOM, Fire Emblem.

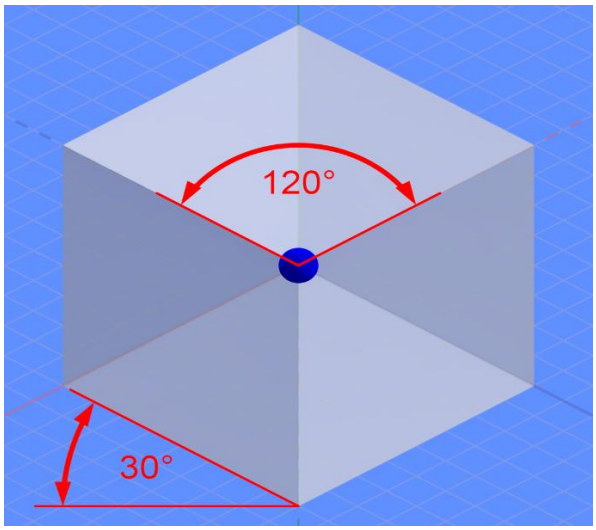
3.2. Isometric stil videoigre

Tijekom izrade videoigre korišten je isometric stil crtanja. Isometric reprezentira 3D prikaz pri čemu je svaka os jednake dužine. Riječ isometric je grčkog podrijetla te znači jednako mjerilo. Slika koje je napravljena u isometric stilu promatra se iz jednog kuta iz koje ostale osi proizlaze pod kutom od trideset stupnjeva. Ovakav stil se koristi u mnogim industrijama kao što su arhitektura, proizvodnja, vizualizacija prostora, dizajneri namještaja i sl.

Prednosti ovog stila uključuju jednostavnost mjerljivosti, preciznost, dobije se osjećaj 3D, nije potrebno gledati objekt s više strana. Neki od nedostataka su djelovanje objekta da je na poziciji iako je uistinu negdje drugdje pozicioniran, nije prikladno za zaobljene objekte, dubina objekta nije realistična.

Isometric se također u nekim slučajevima zna povezivati s 2.5D ili pseudo 3D izrazima pogotovo kada je riječ o animacijama. Razlog tome je što kod 3D stvaranja omogućena je rotacija kako bi se objekt detaljno prikazao sa svih strana, dok ovaj stil nema rotaciju da pokaže ostale strane već na ravnoj plohi simulira dubinu.

Na sljedećoj slici prikazan je isometric stil i opisani uvjeti kao što su trideset stupnjeva kut te jednakost osi.



Slika 1: Prikaz isometric objekta (SharkD 2018.)

4. IZRADA VIDEOIGRE NA POTEZE

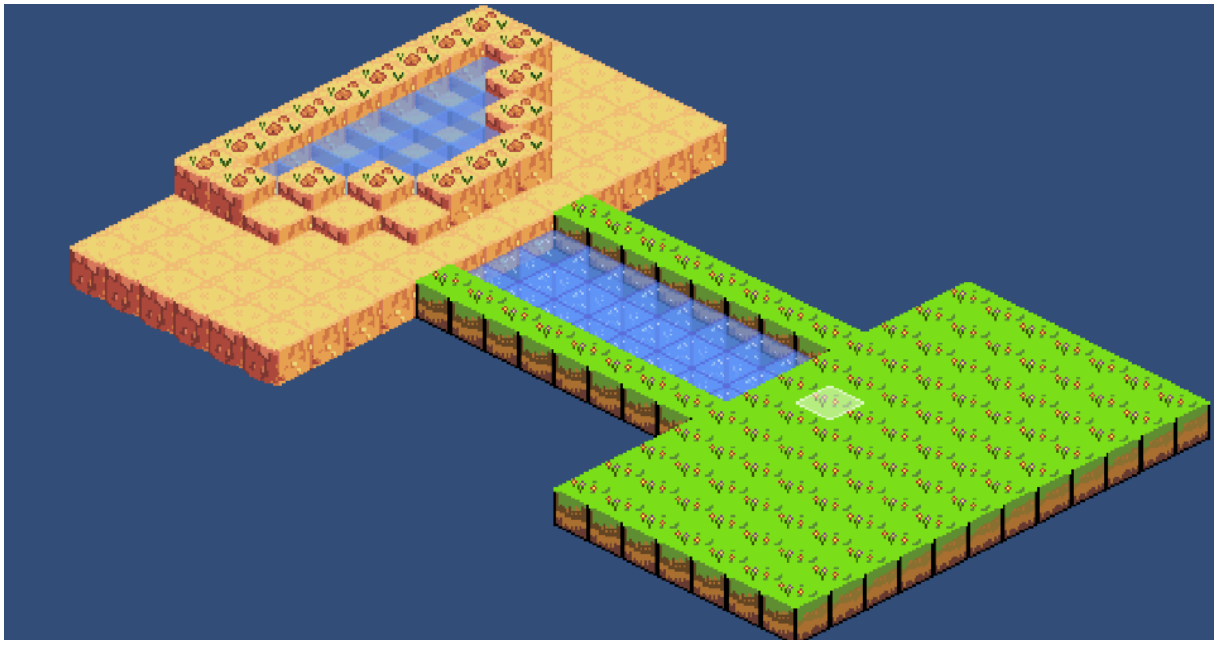
Nastavljajući se na teoriju, u ovom poglavlju se obuhvaćaju spomenuti alati i teorijska pozadina videoigra kako bi se izradila videoigra na poteze uz pomoć Unity-a. U ovom dijelu rada najprije će biti objašnjena potrebna funkcionalnost, a zatim pozadinsko programsko rješenje koje to omogućuje.

Svrha ovog praktičnog dijela i izrade jest pokazati sve potrebne korake i kombinacije industrija kako bi se na minijaturan način prikazala izrada videoigre. Cilj videoigre jest potaknuti igrača da razmišlja kako svaki potez ima utjecaj na konačan rezultat. U videoigri postoje dvije strane koje se sukobljavaju sve dok jedna strana ne ostane posljednja, nakon čega je kraj scenarija.

4.1. Izgled mape

Mapa na kojoj se odvija videoigra bazirana je na rešetkama (eng. Grid) pri čemu svaka ćelija reprezentira 32x32 piksela. Obzirom da je igra napravljena u posebnom stilu *isometric,* mapa djeluje kao da ima 3D. Sačinjena je u takozvanom 2.5D stilu gdje imamo x,y,z koordinate, ali nema realističan prikaz dubine.

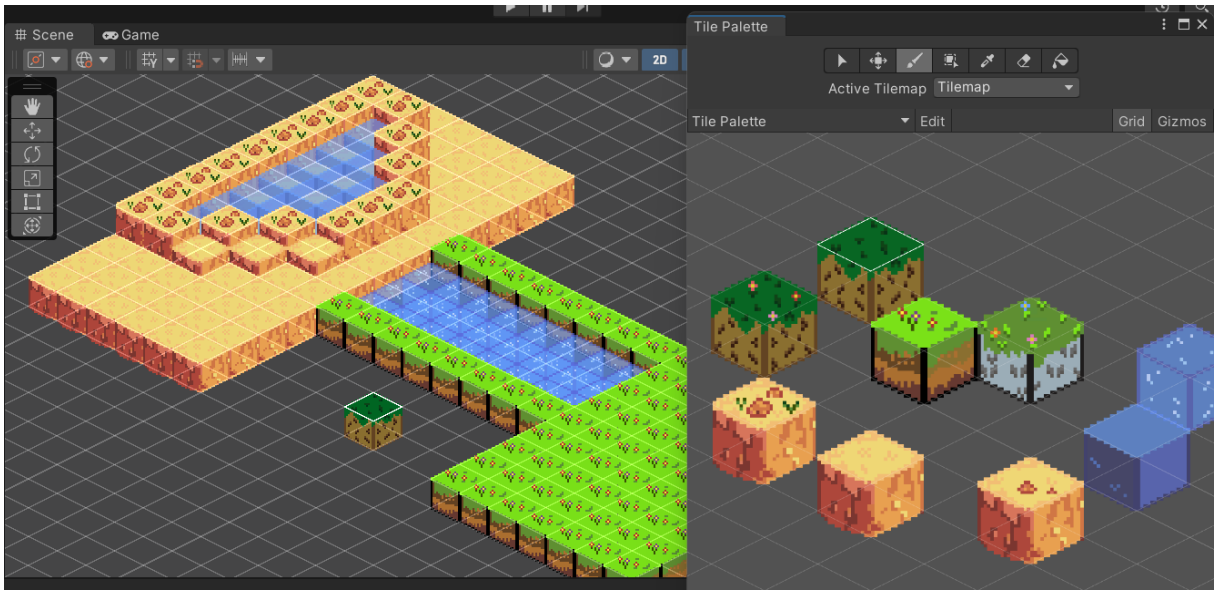
Jedan primjer mape na kojoj se videoigra odvija prikazan je na sljedećoj slici.



Slika 2: Prikaz mape na kojoj se videoigra odvija (vlastita izrada)

Na mapi se može prepoznati nekoliko različitih elemenata poput vode, pijeska, trave, zemlje i sl. Također na mapi se vidi i polje koje je osjenčano bijelom bojom i označava trenutnu poziciju pokazivača, a o tome će biti više riječi malo kasnije.

Na sljedećoj slici može se vidjeti kako se tako jedna mapa gradi, pri čemu postoji paleta blokova koji reprezentiraju elemente. Odabirom bloka unutar palete i pozicioniranjem pokazivača na scenu može se uz pomoć rešetki postaviti na ciljano mjesto odgovarajući element. Mapa je napravljena uz pomoć jednostavnih blokova koji su izrađeni u već spomenutom alatu Aseprite. Uz pomoć Z koordinate moguće je podesiti visinu blokova kako bi na mapi bili vidljivi i povišeni elementi poput pješčanih stepenica.



Slika 3: Prikaz izrade mape (vlastita izrada)

Problem nastaje jer je mapa kreirana kao objekt odnosno prazna slika s kojom korisnik ne može imati interakciju. Izrada mape kao takve nema pozadinsku logiku, kao što je prethodno rečeno, pozicioniranjem pokazivača na polje dobije se mogućnost interakcije. Tu logiku nam rješava skripta pisana u C# programskom jeziku pod nazivom *Map Manager.*

Skripta je zadužena za generiranje polja koja se nalaze na blokovima kako bi interakcija bila moguća. Unutar skripte postoje dvije varijable koje se ističu *overlayTilePrefab* i *overlayTileContainer.* Tim varijablama dodjeljuju se vrijednosti kroz Unity sučelje. Varijabla *overlayTilePrefab* označava kako će površina polja izgledati, u ovom slučaju prozirno bijela. Varijabla *overlayTileContainer* označava spremište u kojem se nalaze sva moguća polja na toj mapi. Funkcija *GenerateGrid()* zaslužna je za međusobno povezivanje komponenti kako bi se omogućila interakcija. Funkcija pregledava mapu pri čemu svakoj vrijednosti na lokaciji x,y,z gdje pronađe blok veličine 32x32 piksela te dodjeljuje površinu s kojom je moguća interakcija i sprema ga u spremište.

```
public class MapManager : MonoBehaviour
{
    private static MapManager _instance;
    public static MapManager Instance { get { return _instance; } }
    [SerializeField] private OverlayTile overlayTilePrefab;
    [SerializeField] private GameObject overlayTileContainter;
    private Dictionary<Vector2Int, OverlayTile> map;

    public Dictionary<Vector2Int, OverlayTile> GetMap() => map;
    private void Awake()
    {
        if (_instance != null && _instance != this)
        {
            Destroy(this.gameObject);
        }
        else
        {
            _instance = this;
        }
    }

    public void GenerateGrid()
    {
        //Generate Grid
        var tileMap = gameObject.GetComponentInChildren<Tilemap>();
        map = new Dictionary<Vector2Int, OverlayTile>();
        //Limit of the map
        BoundsInt bounds = tileMap.cellBounds;
        //Loop for all tiles
        for (int z = bounds.max.z; z >= bounds.min.z; z--)
        {
            for (int y = bounds.min.y; y < bounds.max.y; y++)
            {
                for (int x = bounds.min.x; x < bounds.max.x; x++)
                {
                    var tileLocation = new Vector3Int(x, y, z);
                    var tileKey = new Vector2Int(x, y);
                    //check for holes in map
```

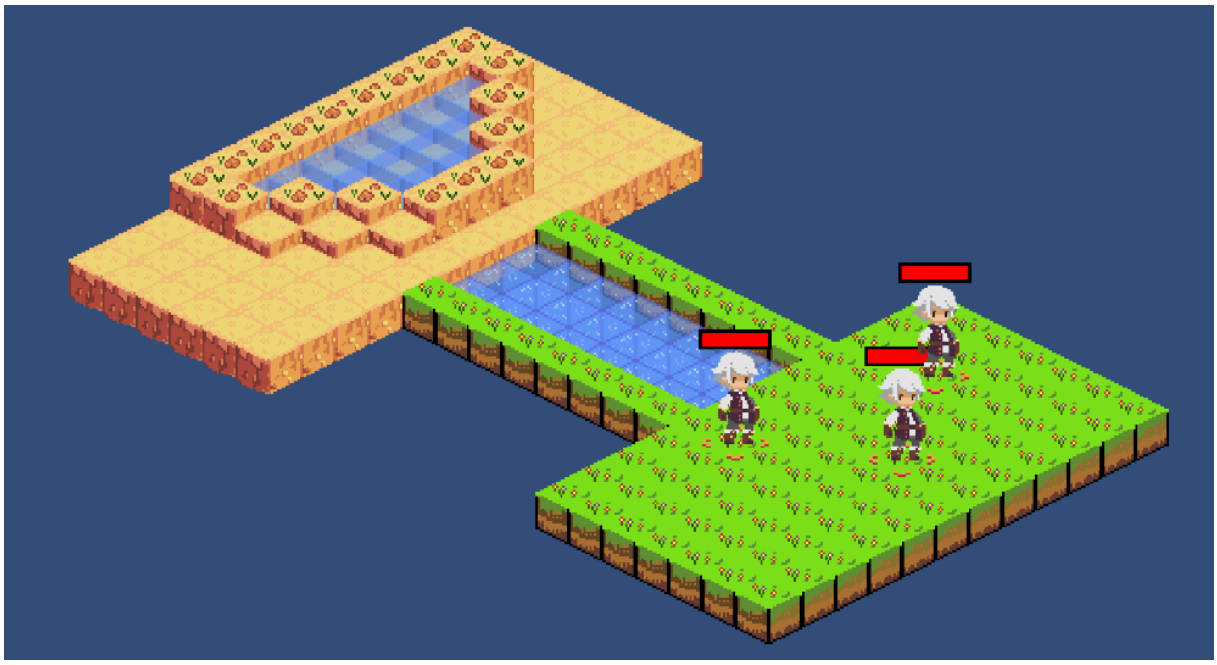
```
                    if (tileMap.HasTile(tileLocation) &&
!map.ContainsKey(tileKey))
                    {
                        var overlayTile = Instantiate(overlayTilePrefab,
overlayTileContainer.transform);
                        var cellWorldPosition =
tileMap.GetCellCenterWorld(tileLocation);


                        overlayTile.transform.position = new
Vector3(cellWorldPosition.x, cellWorldPosition.y, cellWorldPosition.z +
1);

overlayTile.GetComponent<SpriteRenderer>().sortingOrder =
tileMap.GetComponent<TilemapRenderer>().sortingOrder;
                        overlayTile.gridLocation = tileLocation;
                        map.Add(tileKey, overlayTile);
                    }
                }


            }


        }
        GameManager.Instance.UpdateGameState(GameState.SpawnUnits);
    }
}
```

Isječak programskog koda 1: Generiranje mape (vlastita izrada)

4.2. Postavljanje likova unutar scenarija

Sljedeći zadatak je kreirati programsku logiku za pozicioniranje likova igrača na postojeću interaktivnu mapu. Igraču je radi jednostavnosti dodijeljeno tri lika koja može pozicionirati bilo gdje na mapi.

Likovi se postavljaju odabirom i lijevim klikom pokazivača na polje na koje ih želimo pozicionirati kao što je prikazano na sljedećim slikama.



Slika 4: Prikaz postavljenih sva tri lika (vlastita izrada)

Za postavljanje likova prema prethodnoj slici potrebno je koristiti više međusobno povezanih skripti. Kreirana je *GameManager* skripta koja je zadužena za upravljanje toka videoigre. Sljedeća skripta *UnitManager* zadužena je za sve funkcionalnosti oko likova kao što su kretanje i više različitih vrsta napada. Treća skripta *OverlayTile* zadužena je za upravljanje poljima i njihovim vrijednostima.

Nakon što se generira interaktivna mapa, GameManager prebacuje stanje videoigre u kojem igrač pozicionira likove. Ovo stanje naziva se *SpawnUnits*, a ono određuje koliko je moguće postaviti upravljivih likova na mapu. Prvo se pokreće metoda *SpawnAllies()* sa kojom igrač postavi lika na određeno polje.

Skripta *OverlayTile* čeka dopuštenje od *GameManager* za pozicioniranje likova na polje te koristi tri metode: *OnMouseDown()* koja se poziva lijevim klikom pokazivača, *GetFocusedOnTile()* koja vraća poziciju trenutne lokacije pokazivača kako bi korisnik mogao odabrati postojeća polja te metodu *PositionCharacterOnTile(OverlayTile)* koja služi da na odabrano polje postavi lika.

Provjerava se stanje videoigre, u slučaju da nije stanje *SpawnUnits* prekida se bilo koja sljedeća provjera, nakon čega dohvaća se polje na kojem je trenutno miš pozicioniran kako bi spriječili pozicioniranje van mape ili područja koja nisu vidljiva. Nakon tih provjera ide instanciranje lika kojeg želimo postaviti. Za to imamo metodu *GetUnit()* koja se nalazi u UnitManager-u i služi samo kako bi dohvatila izgled (prefab) našeg lika. Nakon toga varijabli character se dodjeljuje vrijednost našeg prefab i uz pomoć treće funkcije

PositionCharacterOnTile() tog lika postavljamo na željeno polje pri čemu se koriste Unity metode kao što su transform kako bi se lik mogao pozicionirati na odgovarajuću x,y,z koordinatu. Metode se ponavljaju skroz dok imamo likova za postaviti, u našem slučaju skroz dok varijabla *numberOfAllies* nije manja od jedan. U trenutku kada postane manja od jedan, *GameManager* prebacuje stanje igre u sljedeću vrijednost koje bi bilo pozicioniranje protivnika.

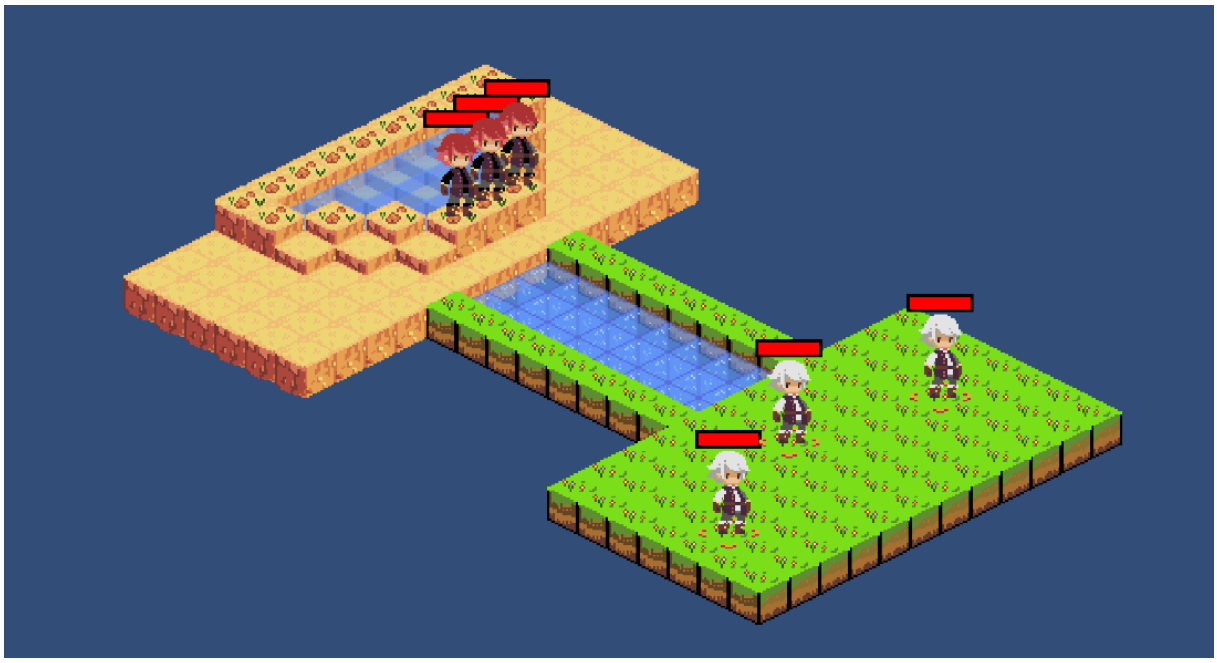
Implementacija navedenih promjena nalazi se na sljedećem isječku programskog koda.

```
//Programski dio iz UnitManager.cs
public void SpawnAllies()
    {
        numberOfAllies = 3;
    }
//Programski dio iz OverlayTile.cs
public class OverlayTile : MonoBehaviour
{
private BaseUnit character;
    void OnMouseDown()
    {
        if (GameManager.Instance.State != 
GameManager.GameState.SpawnUnits) return;
        var focusedTileHit = GetfocusedOnTile();

        if (focusedTileHit.HasValue)
        {
            OverlayTile overlayTile = 
focusedTileHit.Value.collider.gameObject.GetComponent<OverlayTile>();
            transform.position = overlayTile.transform.position;
            gameObject.GetComponent<SpriteRenderer>().sortingOrder = 
overlayTile.GetComponent<SpriteRenderer>().sortingOrder;

                if (character == null)
                {
                    character = UnitManager.Instance.GetUnit();
                    PositionCharacterOnTile(overlayTile);
                    character.SetStandingOnTile(overlayTile);
                    UnitManager.Instance.numberOfAllies--;
                }
        }
        if(UnitManager.Instance.numberOfAllies < 1)
        {
```

```
            GameManager.Instance.UpdateGameState(GameState.SpawnEnemies);
        }
    }
   public RaycastHit2D? GetfocusedOnTile()
    {
        Vector3 mousePosition =
Camera.main.ScreenToWorldPoint(Input.mousePosition);
        Vector2 mousePosition2D = new Vector2(mousePosition.x,
mousePosition.y);
        RaycastHit2D[] hits = Physics2D.RaycastAll(mousePosition2D,
Vector2.zero);
        if (hits.Length > 0)
        {
            return hits.OrderByDescending(i =>
i.collider.transform.position.z).First();
        }

        return null;
    }
    public void PositionCharacterOnTile(OverlayTile tile)
    {
        character.transform.position = new
Vector3(tile.transform.position.x, tile.transform.position.y + 0.0001f,
tile.transform.position.z + 2);
        character.GetComponent<SpriteRenderer>().sortingOrder =
tile.GetComponent<SpriteRenderer>().sortingOrder;
        character.SetStandingOnTile(tile);
        tile.allyOnTile = (Ally1)character;
    }
}
```

Isječak programskog koda 2: Pozicioniranje lika na željeno polje (vlastita izrada)

Pozicioniranje neprijatelja funkcionira na sličan način. Postoji varijabla koja definira koliko protivnika ima te skroz dok vrijednost nije manja od 1 protivnik se generira. Postavljanje protivnika na polje se izvršava automatski bez utjecaja igrača na način da se od svih mogućih polja odabere redom iz liste polja te se postavi neprijatelj ako je polje slobodno.



Slika 5: Prikaz postavljenih protivnika (vlastita izrada)

```
//Programski dio iz UnitManager.cs
public void SpawnEnemies()
    {

        numberOfEnemies = 3;
        Dictionary<Vector2Int, OverlayTile> map =
MapManager.Instance.GetMap();
        foreach (var tile in map)
        {
     if (GameManager.Instance.State !=
GameManager.GameState.SpawnEnemies) return;
            BaseUnit character = GetEnemyUnit();
            PositionCharacterOnTile(character,tile.Value);
            character.SetStandingOnTile(tile.Value);
            numberOfEnemies--;
            if (numberOfEnemies<1)
            {

GameManager.Instance.UpdateGameState(GameState.PlayerTurn);
            }
        }

    }
```

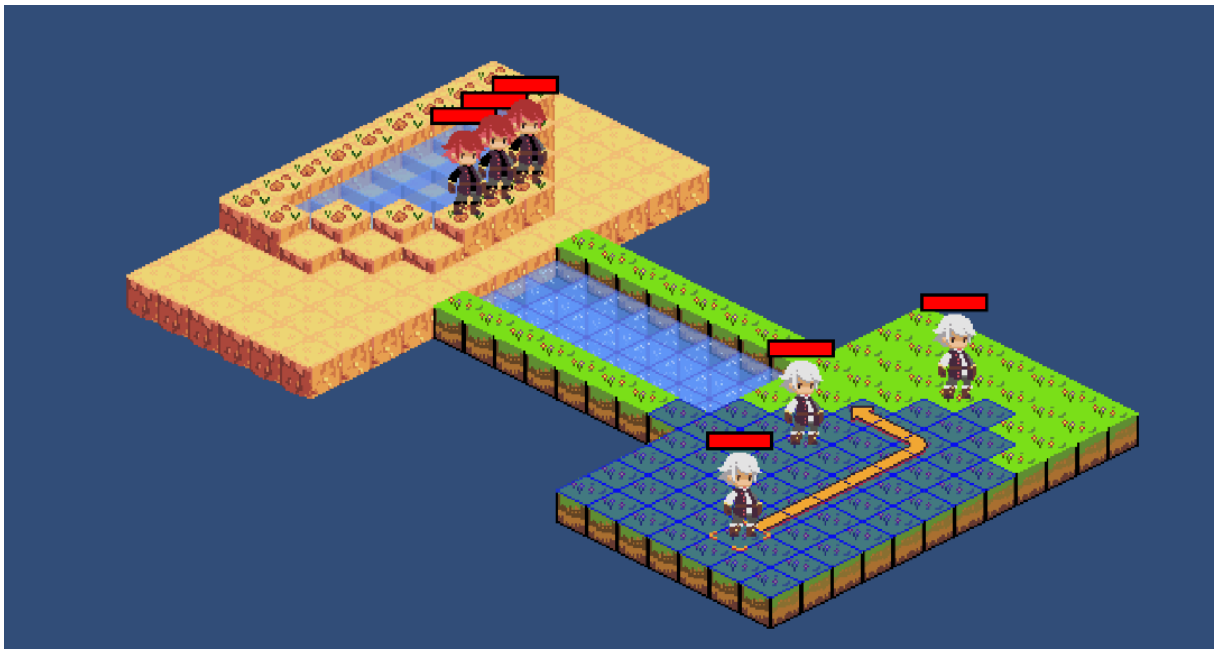
```
private void PositionCharacterOnTile(BaseUnit character, OverlayTile
tile)
    {
        character.transform.position = new
Vector3(tile.transform.position.x, tile.transform.position.y + 0.0001f,
tile.transform.position.z + 2);
        character.GetComponent<SpriteRenderer>().sortingOrder =
tile.GetComponent<SpriteRenderer>().sortingOrder;
        character.SetStandingOnTile(tile);
        tile.SetEnemyOnTile((Enemy1)character);

    }

public BaseUnit GetEnemyUnit()
    {
        var randomPrefab = GetRandomUnit<BaseEnemy>(Faction.Enemy);
        spawnedEnemy = Instantiate(randomPrefab);
        return spawnedEnemy;
    }

private T GetRandomUnit<T>(Faction faction) where T : BaseUnit
    {
        return (T)_units.Where(u => u.Faction == faction).OrderBy(o =>
Random.value).First().UnitPrefab;
    }
```

Isječak programskog koda 3: Pozicioniranje automatski neprijatelja (vlastita izrada)

4.3. Kretanje likova

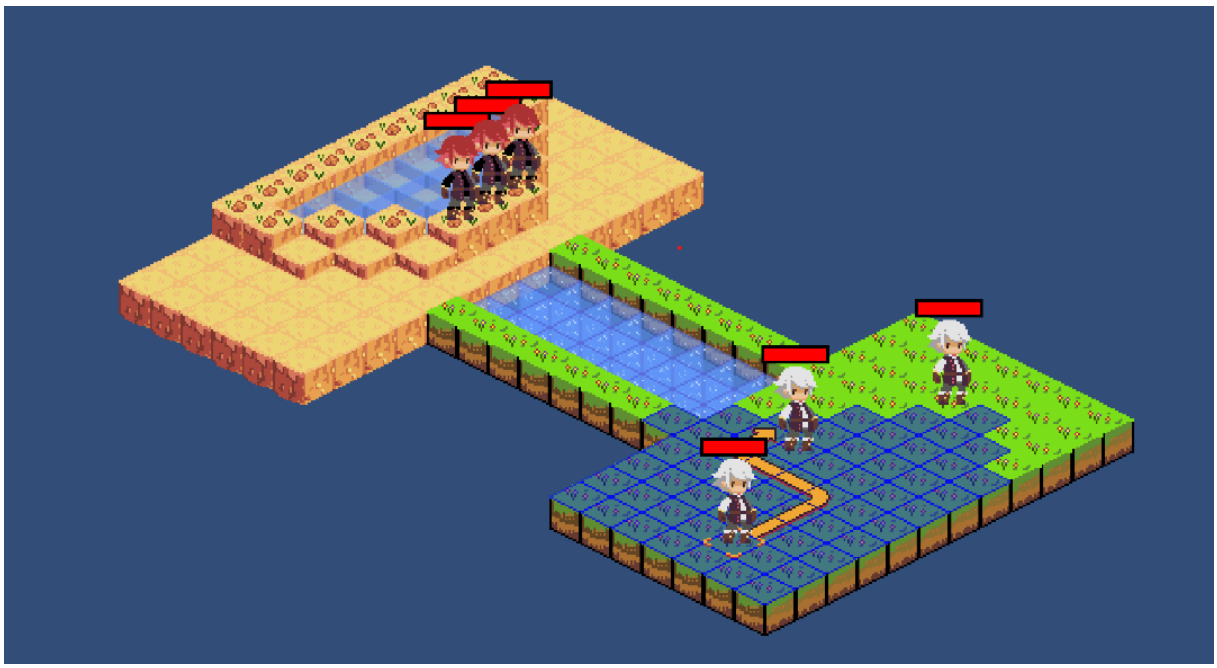
Nakon što je mapa postavljena i likovi su pozicionirani vrijeme je za odvijanje radnje. Jedna od radnji koju igrači mogu poduzeti je kretanje. Svaki lik ima sposobnost kretanja pri čemu igrač odabire željenog lika te ga premjesti na određeno polje. Kako bi to bilo moguće potrebno je niz varijabli kao što su broj koraka koje lik može napraviti, brzina kretanja te putanja do određenog cilja.

Kao što je prikazano na sljedećim slikama, lik odabire najkraći put mogući do odredišta, sa naznakom da ne može preći preko polja koje već ima lika na sebi. Igraču se prikazuje smjer kretanja pomoću strjelice.



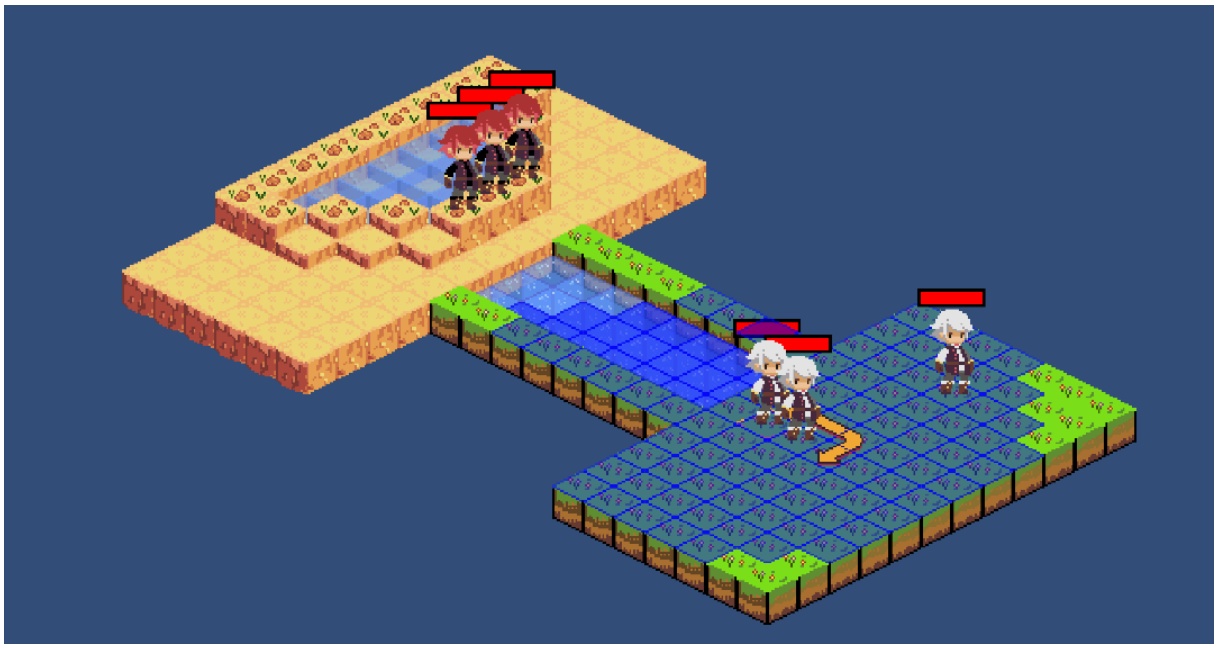
Slika 6: Prikaz mogućeg kretanja odabranog lika (vlastita izrada)

Na sljedećoj slici se prikazuje najbolja putanja kretanja lika kako bi došao iza drugog lika, te dodatni prikaz strjelice gdje mijenja smjer kretanja dva put.



Slika 7: Prikaz najkračeg puta kretanja (vlastita izrada)

Pritiskom na lijevu tipku pokazivača te odabirom polja na koje se lik može pomaknuti, pokreče se animacija kretanja koja vodi lika do odredišta. Kao što je prikazano na slici osam lik se pozicionirao na prethodno odabrano polje pri čemu je morao zauzeti dodatna polja zbog drugog lika.



Slika 8: Prikaz pozicije nakon kretanja lika (vlastita izrada)

4.3.1. A* Algoritam

A* je pretraživački algoritam koji traži najkraći put između dvije točke. U ovome slučaju korišten je ovaj algoritam jer se svaki korak odvija pojedinačno pri čemu nema dijagonalnog kretanja.

Algoritam se sastoji primarno od tri varijable:

- G označava udaljenost do ciljanog polja prateći put koji smo generirali.
- H označava procjenu koliko je potrebno do udaljenosti pri čemu se radi matematički izračun.
- F označava sumu G i H varijable.

Način na koji A* algoritam funkcionira:

- Inicijalno postoje dvije liste, otvorena i zatvorena.
- Inicijalizira se otvorena lista te se dodaje početno polje.
- Inicijalizira se zatvorena lista te se ponavljaju sljedeći koraci skroz dok otvorena lista nije prazna :
    1. Pronađe se element sa najmanjim F iz otvorene liste.
    2. Pronađeni element miče se iz otvorene liste.
    3. Pronađeni element dodaje se u zatvorenu listu.
    4. Od pronađenog elementa traže se njegovi susjedi.

I. Za svakog susjeda se računa G i H uz pomoć dodatne funkcije koja bude dodatno definirana.
II. Ako je susjed cilj prestaje se tražiti dalje.
III. Ako susjed nije u otvorenoj listi, potrebno ga je dodati.

Postupak se odvija skroz dok otvorena lista ima elemenata te se na kraju dobiva nova lista koja se sastoji od elemenata zatvorene liste. Koristi se Manhattan formula za izračun vrijednosti G i H.

$$\text{Manhattan udaljenost} = |(pocetak.x - cilj.x)| + |(pocetak.y - cilj.y)|$$

Koristi se Manhattan jer u ovome slučaju postoje samo četiriju moguća smjera kretanja - gore, desno, dolje, lijevo.

Logika koja je implementirana u metodi *FindPath(start, end, path)*, pri čemu moguća polja označavaju radijus mogućeg kretanja odabranog lika.

```
public class PathFinder
{
  public List<OverlayTile> FindPath(OverlayTile start, OverlayTile end,
List<OverlayTile> possibleTiles)
    {
        List<OverlayTile> openList = new List<OverlayTile>();
        List<OverlayTile> closedList = new List<OverlayTile>();



        openList.Add(start);
        while (openList.Count > 0)
        {
            OverlayTile currentTile = openList.OrderBy(x =>
x.F).First();
            openList.Remove(currentTile);
            closedList.Add(currentTile);
            if(currentTile == end)
            {
                return GetFinishedList(start, end);
            }
            var neighbourTiles =
MapManager.Instance.GetNeighbourTiles(currentTile, possibleTiles);
            foreach (var neighbour in neighbourTiles)
            {
                if (neighbour.isBlockedTile ||
closedList.Contains(neighbour))
```

```
                {
                    continue;
                }

                neighbour.G = GetManhattenDistance(start, neighbour);
                neighbour.H = GetManhattenDistance(end, neighbour);

                neighbour.previousTile = currentTile;

                if (!openList.Contains(neighbour))
                {
                    openList.Add(neighbour);
                }
            }
        }
        return new List<OverlayTile>();
    }

    private List<OverlayTile> GetFinishedList(OverlayTile start,
OverlayTile end)
    {
        List<OverlayTile> finishedList = new List<OverlayTile>();
        OverlayTile currentTile = end;
        while (currentTile != start)
        {
            finishedList.Add(currentTile);
            currentTile = currentTile.previousTile;
        }
        finishedList.Reverse();
        return finishedList;
    }

    private int GetManhattenDistance(OverlayTile start, OverlayTile
neighbour)
    {
        return Mathf.Abs(start.gridLocation.x -
neighbour.gridLocation.x) + Mathf.Abs(start.gridLocation.y -
neighbour.gridLocation.y);
    }

}
```

Isječak programskog koda 4: A* algoritam PathFinder (vlastita izrada)

U implementaciji se spominju moguća polja koja se dodatno dobivaju uz pomoć *RangeFinder* klase koja ima metodu *GetTilesInRange(OverlayTile, range).* Radijus označava koliko koraka lik može napraviti, u ovome slučaju osam, dok trenutno polje označava na kojem se polju trenutno lik nalazi.

```
public class RangeFinder
{
    public List<OverlayTile> GetTilesInRange(OverlayTile currentTile,
int range)
    {
        var inRangeTiles = new List<OverlayTile>();
        int stepCount = 0;
        inRangeTiles.Add(currentTile);
        var tileForPreviousStep = new List<OverlayTile>();
        tileForPreviousStep.Add(currentTile);

        while (stepCount < range)
        {
            var surroundingTiles = new List<OverlayTile>();

            foreach (var tile in tileForPreviousStep)
            {

surroundingTiles.AddRange(MapManager.Instance.GetNeighbourTiles(tile,
new List<OverlayTile>()));
            }

            inRangeTiles.AddRange(surroundingTiles);
            tileForPreviousStep = surroundingTiles.Distinct().ToList();
            stepCount++;
        }
        return inRangeTiles.Distinct().ToList();
    }

}
```

Isječak programskog koda 5: RangeFinder (vlastita izrada)

PathFinder uz pomoć *RangeFinder* omogućuje logiku kretanja pri čemu daje najkraći put do odredišta.

4.3.2. Akcija Kretanja

U prijašnjem poglavlju opisana je matematička podloga iza kretanja, no to nije dovoljno kako bi se lik pomaknuo. Kao što je vidljivo na slici X koriste se strjelice kako bi igraču bila vidljiva putanja kretanja. Za strjelice se koristi prevoditelj koji pretvara matematičke točke u slike koje igrač razumije na ekranu. Na kraju su te slike potrebne za prikaz strjelice, jer postoji mogućih dvanaest stanja od kojih se strjelica sastoji kao što su strjelica lijevo, desno, gore, dolje i sl.

```
public class DirectionalArrowsTranslator
{
  public enum ArrowDirection
    {
        None= 0, Up = 1, Down = 2, Left = 3, Right = 4, TopRight = 5,
BottomRight = 6,
        TopLeft = 7, BottomLeft = 8, UpFinished = 9, DownFinished = 10,
LeftFinished = 11,
        RightFinished = 12
    }
    public ArrowDirection TranslateDirection(OverlayTile previousTile,
OverlayTile currentTile, OverlayTile futureTile)
    {
        bool isFinal = futureTile == null;


        Vector2Int pastDirection = previousTile != null ?
currentTile.grid2DLocation - previousTile.grid2DLocation : new
Vector2Int(0,0);
        Vector2Int futureDirection = futureTile != null ?
futureTile.grid2DLocation - currentTile.grid2DLocation : new
Vector2Int(0, 0);
         Vector2Int direction = pastDirection != futureDirection ?
pastDirection + futureDirection : futureDirection;


        if (direction == new Vector2Int(0,1) && !isFinal) return
ArrowDirection.Up;
        if (direction == new Vector2Int(0, -1) && !isFinal) return
ArrowDirection.Down;
        if (direction == new Vector2Int(1, 0) && !isFinal) return
ArrowDirection.Right;
        if (direction == new Vector2Int(-1, 0) && !isFinal) return
ArrowDirection.Left;


        if (direction == new Vector2Int(1, 1) && !isFinal)
        {
```

```
            if (pastDirection.y < futureDirection.y) return
ArrowDirection.BottomLeft;
            else return ArrowDirection.TopRight;
        }
        if (direction == new Vector2Int(-1, 1) && !isFinal)
        {
            if (pastDirection.y < futureDirection.y) return
ArrowDirection.BottomRight;
            else return ArrowDirection.TopLeft;
        }
        if (direction == new Vector2Int(1, -1) && !isFinal)
        {
            if (pastDirection.y > futureDirection.y) return
ArrowDirection.TopLeft;
            else return ArrowDirection.BottomRight;
        }
        if (direction == new Vector2Int(-1, -1) && !isFinal)
        {
            if (pastDirection.y > futureDirection.y) return
ArrowDirection.TopRight;
            else return ArrowDirection.BottomLeft;
        }
        if (direction == new Vector2Int(0, 1) && isFinal) return
ArrowDirection.UpFinished;
        if (direction == new Vector2Int(0, -1) && isFinal) return
ArrowDirection.DownFinished;
        if (direction == new Vector2Int(1, 0) && isFinal) return
ArrowDirection.RightFinished;
        if (direction == new Vector2Int(-1, 0) && isFinal) return
ArrowDirection.LeftFinished;


        return ArrowDirection.None;
    }
}
```

Isječak programskog koda 6: Prevoditelj strjelica (vlastita izrada)

Nakon što se definira grafički prikaz putanje, vrijeme je da lik prati put i slijed kretanja. Inicijalna logika je da u metodi *Update()* se odvija kretanje lika pri čemu se provjerava stanje akcije.

Koristi se vektor3 sa x, y, z točkama kako bi se odredila pozicija trenutna pozicija lika te njegov sljedeći korak. *Path* varijabla je zapravo putanja dobivena sa ranije prikazanim A* algoritmom. Lik se kreće od polja do polja koja se nalaze u listi *path* te ga briše iz liste nakon njegovog prolaska.

```
    private void Update()
    {
        if (!isActive)
        {
            return;
        }

        if (isMoving && path.Count() > 0)
        {
            float stoppingDistance = .1f;

            this.targetPosition = new
Vector3(path[0].transform.position.x, path[0].transform.position.y +
0.0001f, path[0].transform.position.z);
            targetPositionTile = path[0];

            if (Vector3.Distance(transform.position, targetPosition) >
stoppingDistance)
            {
                Vector3 moveDirection = (targetPosition -
transform.position).normalized;
                float movementSpeed = 1f;
                transform.position += moveDirection * movementSpeed *
Time.deltaTime;
            }
            else
            {
                unit.GetStandingOnTile().SetAllyOnTile(null);
                PositionCharacterOnTile(targetPositionTile,
(Ally1)unit);
                path.RemoveAt(0);
            }

        }
        else if (isMoving && path.Count.Equals(0))
        {
            isMoving = false;
            foreach (var tile in inRangeTiles)
            {
                tile.HideTile();
            }
            path.Clear();
```

```
            inRangeTiles.Clear();
            ActionEnd();
        }

        if (path.Count() > 0 && isMoving)
        {
            MoveAlongPath();
        }
    }
```

Isječak programskog koda 7: Kretanje lika (vlastita izrada)

Update() se koristi samo kako bi se konstantno provjeravao i ažurirao lik. Ostala logika se nalazi u sljedeće navedenim metodama. *GetValidActionTilePositionList()* kombinira A*. *Pathfinder*, *RangeFinder* i prevoditelj strjelica kako bi se na grafičkom sučelju prikazala moguća polja i putanja koju lik može uzeti uzimajući trenutnu lokaciju miša. *GetInRangeTiles()* je jednostavna metoda koja stvara radijus kretanja koje lik može koristiti. U ovom slučaju to je osam koraka tj. osam polja na koja se lik može pomaknuti. *MoveAlongPath()* služi kako bi se konzistentnom brzinom kretao lik od polja do polja, pri čemu je moguće kombinirati logiku sa *Update()* metodom te time minimizirati količinu programskog koda. Radi lakšeg budućeg razvoja, logika je podijeljena na navedene metode.

```
public class MoveAction : BaseAction
{
    private Vector3 targetPosition;
    private OverlayTile targetPositionTile;
    private bool isMoving = false;
    private List<OverlayTile> path = new List<OverlayTile>();
    private List<OverlayTile> inRangeTiles = new List<OverlayTile>();
    private PathFinder pathFinder;
    private RangeFinder rangeFinder;
    private DirectionalArrowsTranslator directionalArrowsTranslator;

    public override string GetActionName() => "Move";
    public override ActionCategory GetActionCategory() =>
ActionCategory.Move;
    public override int GetActionPointsCost()
    {
        return 1;
    }
…
…
```

```
…
}
```

Isječak programskog koda 8: Varijable potrebne za kretanje (vlastita izrada)

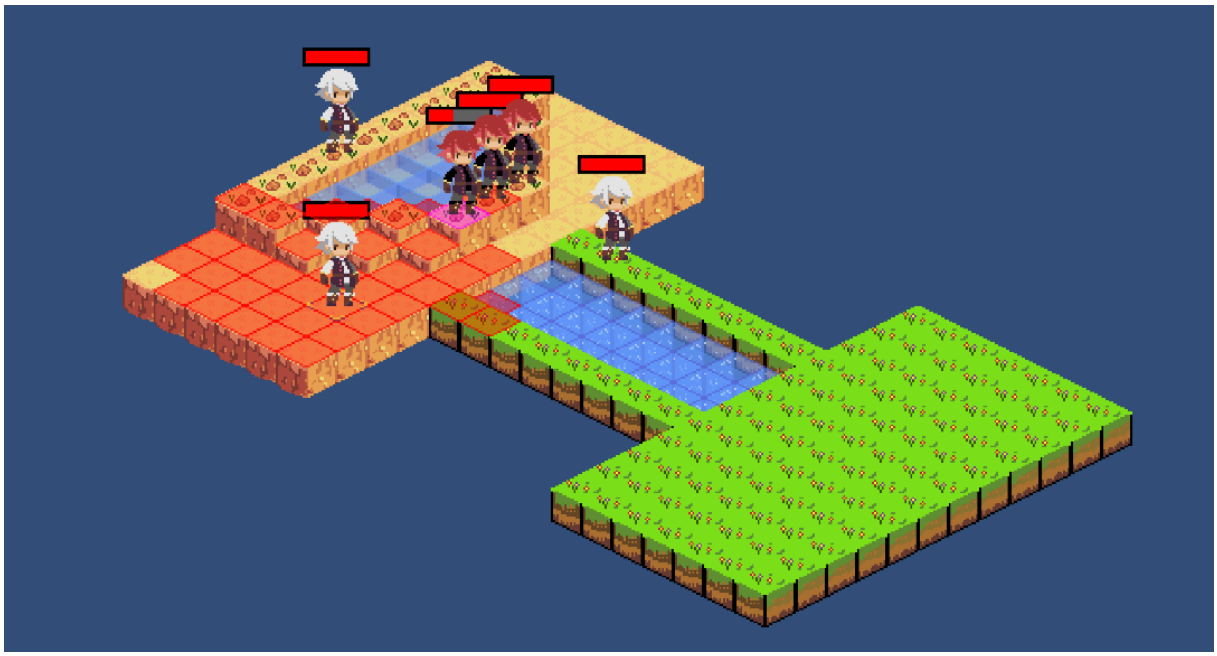
```
public override List<OverlayTile> GetValidActionTilePositionList()
    {
        pathFinder = new PathFinder();
        rangeFinder = new RangeFinder();
        directionalArrowsTranslator = new DirectionalArrowsTranslator();

        GetInRangeTiles();
        inRangeTiles.RemoveAll(tile => tile.GetEnemyOnTile() != null);

        var focusedTileHit =
MouseController.Instance.GetfocusedOnTile();
        OverlayTile overlayTile =
focusedTileHit.Value.collider.gameObject.GetComponent<OverlayTile>();
        path = pathFinder.FindPath(unit.GetStandingOnTile(),
overlayTile, inRangeTiles);
        foreach (var tile in inRangeTiles)
        {
            tile.SetArrowSprite(ArrowDirection.None);
        }
        for (int i = 0; i < path.Count; i++)
        {
            var previousTile = i > 0 ? path[i - 1] :
unit.GetStandingOnTile();
            var futureTile = i < path.Count - 1 ? path[i + 1] : null;

            var arrowDirection =
directionalArrowsTranslator.TranslateDirection(previousTile, path[i],
futureTile);
            path[i].SetArrowSprite(arrowDirection);
        }
        return inRangeTiles;
    }
    private void GetInRangeTiles()
    {

        inRangeTiles =
rangeFinder.GetTilesInRange(unit.GetStandingOnTile(), 8);
        foreach (var tile in inRangeTiles)
```

```
        {
            tile.ShowTile();
        }

    }

    public void MoveAlongPath()
    {
        float movementSpeed = 2f;
        float step = 0;
        step = movementSpeed * Time.deltaTime;
        var zIndex = path[0].transform.position.z;

        ally.transform.position =
Vector2.MoveTowards(ally.transform.position, path[0].transform.position,
step);
        ally.transform.position = new Vector3(ally.transform.position.x,
ally.transform.position.y, zIndex);
        if (Vector2.Distance(ally.transform.position,
path[0].transform.position) < 0.0001f)
        {
            PositionCharacterOnTile(path[0], ally);
            path.RemoveAt(0);
        }
    }
```

Isječak programskog koda 9: Dobivanje polja i radijusa za kretanje (vlastita izrada)

4.4. Akcija Napad

Kako je cilj igre pobijediti neprijatelje, jedini način je uništiti svakog protivnika. Akcija napad omogućuje oduzimanje životnih bodova od likova, te kada lik dosegne nula životnih bodova smatra se uništenim i briše se s mape. Mogućnost napada označeno je na mapi crvenom bojom, što označava svako polje nad kojim se može izvršiti akcija napada. Maksimalni radijus akcije ograničen je na pet polja. Prilikom odabira polja nad kojim se želi izvršiti akcija napada, provjerava se postojanost neprijateljskog lika te njegovih životnih bodova. U ovome slučaju napad oduzima šezdeset životnih bodova dok svaki lik počinje sa sto životnih bodova. Na slici X odabrani je lik nad kojim je izvršen napad te je smanjenje životnih bodova prikazano u obliku kvadratića.



Slika 9: Prikaz akcije napad (vlastita izrada)

Napad se sastoji od nekoliko stanja kao što su priprema, napad i hlađenje. Svako stanje ima svoje vremensko trajanje, a po potrebi se odvijaju različite stvari kao što su prikaz animacija, mogućnost obavijesti, promjena glazbe i sl. Najbitnije stanje je napad u kojem se poziva metoda *Attack()* koja provjerava postoji li lik te mu oduzima šezdeset životnih bodova. Logika grafičkog prikaza napada jednaka je akciji kretanja objašnjena u prijašnjem poglavlju.

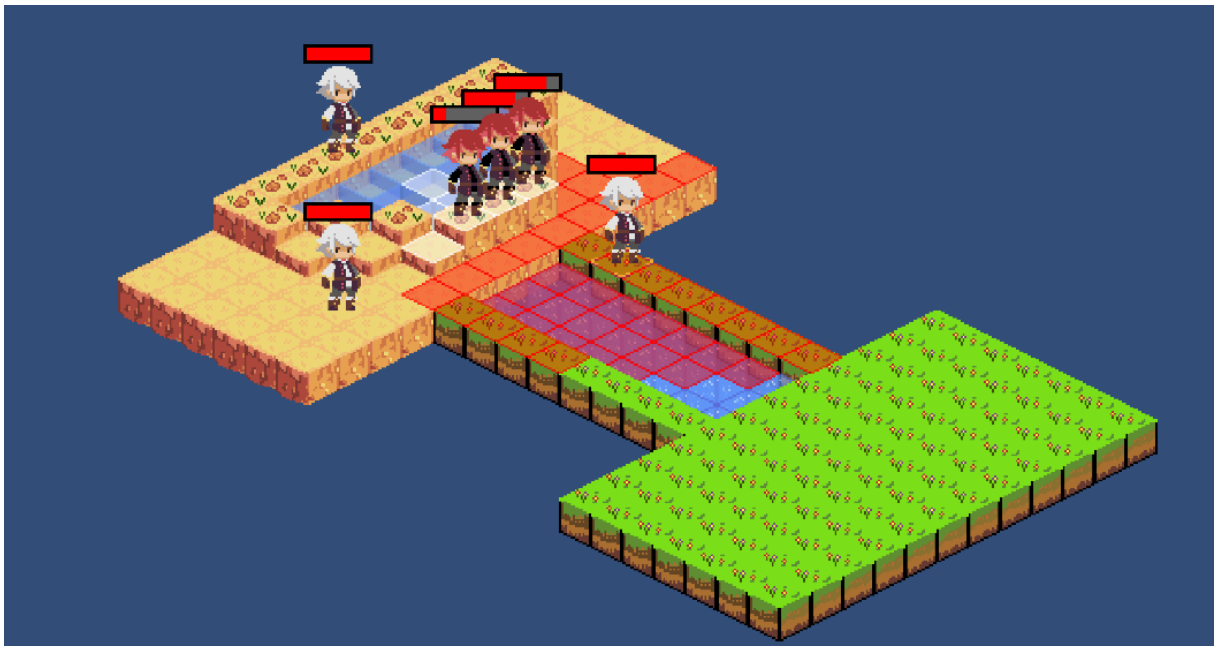
```
public class AttackAction : BaseAction
{
    private enum State
    {
        Charging,
        Attacking,
        Cooloff
    }
    private void Update()
    {
        if (!isActive)
        {
            return;
        }
        stateTimer -= Time.deltaTime;
        switch (state)
        {
            case State.Charging:
```

```
            break;
        case State.Attacking:
            if (canAttack)
            {
                Attack();
                canAttack = false;
            }
            break;
        case State.Cooloff:
            break;
    }
    if (stateTimer < 0f)
    {
        NextState();
    }
}
private void NextState()
{
    switch (state)
    {
        case State.Charging:
                state = State.Attacking;
                float attackingStateTime = 0.6f;
                stateTimer = attackingStateTime;
            break;
        case State.Attacking:
                state = State.Cooloff;
                float coolOffStateTime = 0.4f;
                stateTimer = coolOffStateTime;
            break;
        case State.Cooloff:
            ActionEnd();
            break;
    }

}
private void Attack()
{
    if(targetUnit != null)
    targetUnit.Damage(60);
```

```
        }
    }
```

Isječak programskog koda 10: Akcija Napad (vlastita izrada)

Također postoji i druga varijanta napada koja obuhvaća više polja odjednom (eng. AOE, Area of effect). Takva varijanta odjednom oduzima životne bodove većoj količini neprijatelja, pod cijenom da radi manje štete. U ovom slučaju dvadeset životnih bodova. Kao što je prikazano na slici taj napad se već jednom izvršio i svakom neprijatelju nedostaje dvadeset životnih bodova.



Slika 10: Prikaz akcije napada na područje (vlastita izrada)

Ova varijanta napada prati jednaku logiku prijašnje opisanog napada, pri čemu su definirane značajne razlike u radijusu i količini vrijednosti napada. Varijabla *aoeRadius* je postavljena na dva što označava koliko će polja obuhvatiti u svakom smjeru pri čemu se provjerava radili se o prijateljskom liku. Ukoliko se radi o prijateljskom liku ne oduzimaju mu se životni bodovi.

```
public class AOEAction : BaseAction
{
    private enum State
    {
        Charging,
        Attacking,
        Cooloff
    }
```

```
    private void Attack()
    {
        foreach (var tile in inRangeAOETiles)
        {
            if (tile.GetEnemyOnTile() != null)
            {
                tile.GetEnemyOnTile().Damage(20);
            }
        }
    }
  public override List<OverlayTile> GetValidActionTilePositionList()
    {
        GetInRangeTiles();
        var focusedTileHit =
MouseController.Instance.GetfocusedOnTile();
        OverlayTile overlayTile =
focusedTileHit.Value.collider.gameObject.GetComponent<OverlayTile>();
        overlayTile.gameObject.GetComponent<SpriteRenderer>().color =
new Color(255, 127, 51, 1);
        int aoeRadius = 2;
        inRangeAOETiles = rangeFinder.GetTilesInRange(overlayTile,
aoeRadius);
        foreach (var tile in inRangeAOETiles)
        {
            tile.gameObject.GetComponent<SpriteRenderer>().color = new
Color(255, 127, 51, 1);
        }
        if (overlayTile.GetEnemyOnTile() != null) targetUnit =
overlayTile.GetEnemyOnTile();
        return inRangeTiles;
    }
   private void GetInRangeTiles()
    {
        inRangeTiles =
rangeFinder.GetTilesInRange(unit.GetStandingOnTile(),
maxAttackDistance);
        foreach (var tile in inRangeTiles)
        {
            tile.ShowTile();
        }
    }
}
```

Isječak programskog koda 11: Akcija AOE Napad (vlastita izrada)

4.5. Logika neprijatelja

Kako bi videoigra imala interakciju s igračem potrebno je omogućiti neprijateljima da također izvršavaju naredbe. Budući da se radi o videoigri protiv računala, potrebna je minimalna logika ili umjetna inteligencija kojom bi računalo moglo smisleno donositi odluke protiv igrača. Ranije su objašnjene logike pozicioniranja neprijatelja na polje i interakcije igrača sa sučeljem. U ovom poglavlju nalazi se potrebna logika kojom računalo samostalno donosi odluke.

U trenutku kad igrač prekine potez stanje igre se prebacuje na potez protivnika pri čemu se za svakog pojedinog neprijateljskog lika donosi odluka. Kako bi to bilo moguće napravljen je upravljač za protivnika koji ide redom od lika do lika te izvršava najbolju akciju.

Logika se sastoji od stanja i vremenskog perioda koje određuje trajanje stanja kako računalo ne bi odrađivalo svu logiku u istom trenutku. U *Update()* metodi upravljač čeka svoj potez kako bi prelazio kroz stanja i obavljao naredbe za svakog lika. Ova metoda se ponavlja skroz dok neprijateljski likovi imaju mogućnost obavljanja bilo kakve naredbe.

```
public class EnemyAI : MonoBehaviour
{
    private void Awake()
    {
        state = State.WaitingForEnemyTurn;
    }
    private void Start()
    {
        TurnSystem.Instance.OnTurnChange += TurnSystem_OnTurnChange;
    }
    private void Update()
    {
        if (TurnSystem.Instance.IsPlayerTurn()) return;
        switch (state)
        {
            case State.WaitingForEnemyTurn: break;
            case State.TakingTurn:
                timer -= Time.deltaTime;
                if (timer < 0) {
                    if (TryTakeEnemeyAIAction(SetStateTakingTurn)) state
= State.Busy;
                    else TurnSystem.Instance.NextTurn();
                }
```

```
                break;
            case State.Busy: break;
        }
    }
    private void SetStateTakingTurn()
    {
        timer = 0.5f;
        state = State.TakingTurn;
    }
}
```

Isječak programskog koda 12: logika AI (vlastita izrada)

Ključna metoda kod upravljača je *TryTakeEnemyAIAction(Action)* koja odrađuje svu logiku potrebnu da neprijateljski lik odradi naredbu.

```
    private bool TryTakeEnemeyAIAction(Action onEnemyAIActionComplete)
    {
        foreach (Enemy1 enemy in
UnitManager.Instance.GetEnemyUnitList())
        {
            if (TryTakeEnemeyAIAction(enemy, onEnemyAIActionComplete))
            {
                return true;
            }
        }
        return false;
    }
    private bool TryTakeEnemeyAIAction(Enemy1 enemy, Action
onEnemyAIActionComplete)
    {
        List<OverlayTile> path = new List<OverlayTile>();

        EnemyAIAction bestEnemyAction = null;
        BaseAction bestBaseAction = null;
        foreach (BaseAction baseAction in enemy.GetBaseActionArray())
        {
            if (!enemy.CanSpendActionPoints(baseAction))
            {
                continue;
            }
            if (bestEnemyAction == null)
            {
                bestEnemyAction = baseAction.GetBestEnemyAIAction();
```

```
                bestBaseAction = baseAction;
            }
            else
            {
                EnemyAIAction testEnemyAIAction =
baseAction.GetBestEnemyAIAction();
                if(testEnemyAIAction != null &&
testEnemyAIAction.actionValue > bestEnemyAction.actionValue)
                {
                    bestEnemyAction = testEnemyAIAction;
                    bestBaseAction = baseAction;
                }
            }
        }

        if(bestEnemyAction != null &&
enemy.TrySpendActionPoints(bestBaseAction))
        {
            MapManager.Instance.HideTilePositionList();
            UnitManager.Instance.SetSelectedAction(bestBaseAction);
            bestBaseAction.GetValidActionTilePositionList();
            bestBaseAction.TakeAction(path, enemy,
onEnemyAIActionComplete);
            return true;
        }
        else
        {
            return false;
        }
    }
```

Isječak programskog koda 13: AI donošenje odluka (vlastita izrada)

Implementacija napada i kretanja računala se ne razlikuje puno od logike za igrača. Dodatno je bilo potrebno modificirati metodu za odabir željenog polja. Ovo je ostvareno uz pomoć primitivne umjetne inteligencije. Razlog zašto se logika naziva primitivna je jer računalo nema podatke od kojih uči i napreduje u donošenju odluka, već s matematičkim jednostavnim funkcijama donosi odluke.

```
    //AI difference in Move action
      public override List<OverlayTile> GetValidActionTilePositionList()
      {
          pathFinder = new PathFinder();
```

```
        rangeFinder = new RangeFinder();
        directionalArrowsTranslator = new DirectionalArrowsTranslator();
        GetInRangeTiles();
        inRangeTiles.RemoveAll(tile => tile.GetEnemyOnTile() != null);
        inRangeTiles.RemoveAll(tile => tile.GetAllyOnTile() != null);
        OverlayTile unitTilePosition =
UnitManager.Instance.GetAllyUnitList().First().GetStandingOnTile();
        int minDistance = GetManhattenDistance(inRangeTiles.First(),
unitTilePosition);
        foreach (var item in inRangeTiles)
        {
            if(GetManhattenDistance(item, unitTilePosition) <
minDistance)
            {
                minDistance = GetManhattenDistance(item,
unitTilePosition);
                overlayTile = item;
            }
        }
…
…
        return inRangeTiles;
    }
//AI difference in Attack action
    public override List<OverlayTile> GetValidActionTilePositionList()
    {
        GetInRangeTiles();
        inRangeTiles.RemoveAll(tile => tile.GetAllyOnTile() == null);
        if(inRangeTiles.Count > 0 )
        targetUnit = inRangeTiles[0].GetAllyOnTile();
        else targetUnit = null;

        //Target Unit with min HP
        foreach (var item in inRangeTiles)
        {
            if (item.GetAllyOnTile().GetHealthNormalized() <
targetUnit.GetHealthNormalized())
                targetUnit = item.GetAllyOnTile();
        }
        return inRangeTiles;
    }
```

Isječak programskog koda 14: AI odabir polja za izvršavanje akcija (vlastita izrada)

4.6. Ostalo

U ovom poglavlju opisuju se ostali dijelovi koji čine videoigru, a nisu glavni sastav same videoigre kao što su: glazba, izbornik, mogućnosti kamere, grafičko sučelje i sl. Cilj ovih dodatnih elemenata je obogatiti iskustvo igrača pri čemu elementi nemaju direktnog utjecaja na odvijanje videoigre. Također svi nabrojani elementi uključuju ranije spomenute industrije kao što su skladatelji, dizajneri, pisci i sl.

4.6.1. Glavni izbornik

Glavni izbornik predstavlja scenu koju igrač prvo uočava prilikom pokretanja videoigre. Izbornik je sačinjen od naslova videoigre te dvije tipke koje su početak igre (eng. Play) i napustiti videoigru (eng. Exit Game). Pritiskom na tipku pozivaju se odgovarajuće metode *OnPlayButton()* i *OnQuitButton()*.

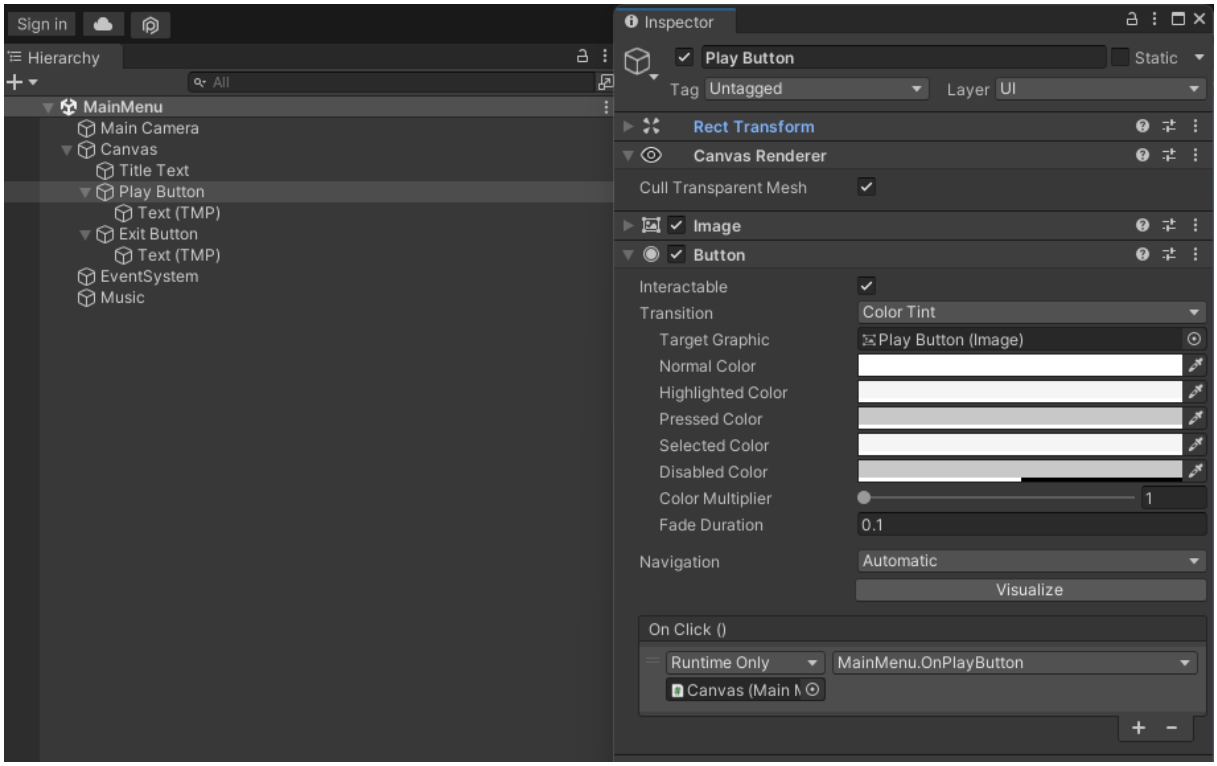
Slika 11: Glavni izbornik (vlastita izrada)

```
public class MainMenu : MonoBehaviour
{
    public void OnPlayButton()
    {
        SceneManager.LoadScene(1);
    }

    public void OnQuitButton()
    {
        Application.Quit();
```

```
        }
    }
```

Isječak programskog koda 15: Glavni izbornik (vlastita izrada)

Skriptu *MainMenu.cs* pridružimo glavnom objektu unutar Unity-a koji je u ovom slučaju Canvas. Nakon toga unutar Canvas elementa postoje dva elementa tipa tipka (eng. Button) koji imaju sami po sebi *On Click()* funkciju prikazano na slici X.



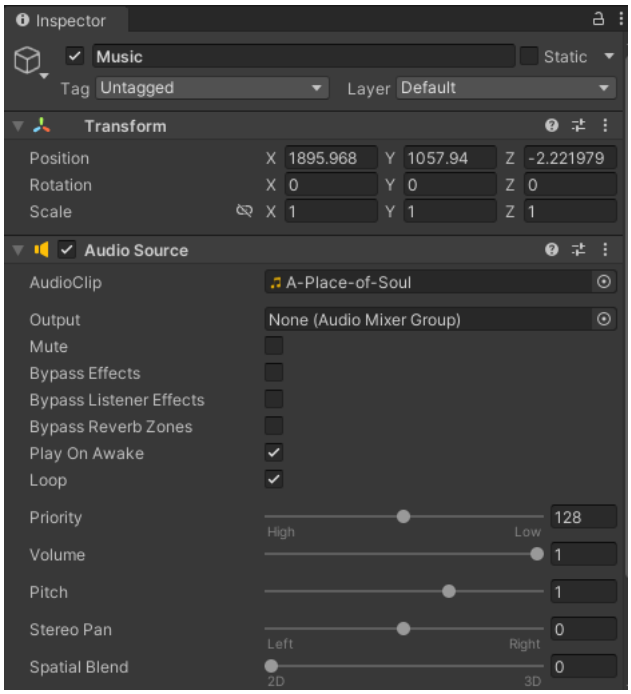
Slika 12: Glavni izbornik u Unity hijerarhiji (vlastita izrada)

4.6.2. Glazba

Glazba je bitan sastavni dio svakog čovjeka. Glazba izaziva niz emocija kod ljudi pa tako ima svrhu u videoigrama da tokom odgovarajuće scene proizlaze željene emocije na površinu. U trenucima kao što su bitke igrač treba dobiti osjećaj hrabrosti i adrenalina, dok u scenama gdje igrač treba donijeti kalkuliranu odluku potrebno je stvoriti osjećaj smirenosti. Sve to omogućava glazba U primjeru ove videoigre korištene su dvije vrste glazbe.

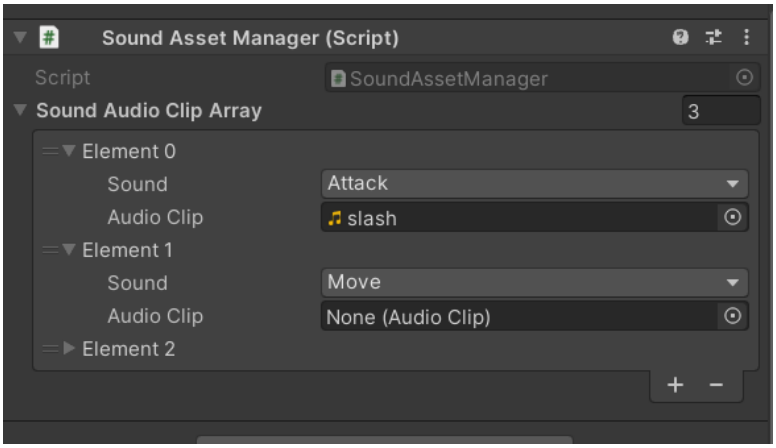
Jedna vrsta glazbe je konstantna melodija koja se ponavlja na razini scene, dok druga vrsta je zvučni efekt prilikom izvršavanja *Attack()* metode za pojedine likove.

Za dodavanje prve vrste glazbe na razini scene dovoljno je koristiti Unity Audio Source koji se doda na bilo koji objekt te odabirom AudioClip-a po želji, prikazano na slici trinaest.



Slika 13: Unity inspector Audio Source (vlastita izrada)

Drugi primjer je složeniji jer uključuje glazbene efekte koji traju po sekundu dvije i pozivaju se samo preko interakcije igrača. Primjer se sastoji od upravljača koji ima obavezu skladištenja svih melodija unutar liste te drugog upravljača koji dohvaća melodiju iz liste ovisno o kojoj metodi je riječ. Prikazano na slici četrnaest skladištu se ručno dodjeljuje glazba i poveznica za stanje kako bi upravljač *SoundManager* mogao povezati.



Slika 14: Skladište glazbe unutar Unity-a (vlastita izrada)

```
public class SoundAssetManager : MonoBehaviour
{
    public static SoundAssetManager Instance;
    private void Awake()
    {
```

```
        Instance = this;
    }

    public SoundAudioClip[] SoundAudioClipArray;

    [System.Serializable]
    public class SoundAudioClip
    {
        public SoundManager.Sound sound;
        public AudioClip audioClip;
    }
}

public static class SoundManager
{
    public enum Sound
    {
        Move,
        Attack,
        Die
    }
    private static GameObject soundGameObject;
    private static AudioSource audioSource;
    public static void PlaySound(Sound sound)
    {
        if(soundGameObject == null)
        {
            soundGameObject = new GameObject("Sound");
            audioSource = soundGameObject.AddComponent<AudioSource>();
        }
        audioSource.PlayOneShot(GetAudioClip(sound));
    }
    private static AudioClip GetAudioClip(Sound sound)
    {
        foreach (SoundAssetManager.SoundAudioClip soundAudioClip in
SoundAssetManager.Instance.SoundAudioClipArray)
        {
            if(soundAudioClip.sound == sound)
            {
                return soundAudioClip.audioClip;
            }
```

```
            }
            Debug.LogError("Sound " + sound + "not found!");
            return null;
        }
    }
    //AttackAction.cs during State.Charging
    SoundManager.PlaySound(SoundManager.Sound.Attack);
```

Isječak programskog koda 16: Upravljač glazbe (vlastita izrada)

4.6.3. Kamera

Kamera ima funkciju promijeniti fokus igrača kako bi igrač mogao dobiti više detalja o pojedinim situacijama na sceni, ovisno koliko je velika scena.

Kamera u ovom primjeru omogućuje korisniku da približi prikaz, udalji, pomakne se u stranu i sl. U primjeru je korišten vanjski paket *CinemaMachine* koji je potpuno besplatan. Jedan od mogućih položaja kamere vidljiv je na slici petnaest. *CameraController* sadržava objekt *Cinemamachine* te ovisno o tipki koju pritisnemo kamera mijenja svoj položaj.

Slika 15: Približen i pomaknut prikaz videoigre (vlastita izrada)

```
public class CameraController : MonoBehaviour
{
    [SerializeField] private CinemachineVirtualCamera
cinemachineVirtualCamera;
```

```
    private const float MIN_ZOOM_Y = 2f;
    private const float MAX_ZOOM_Y = 6f;
    static float t = 0f;

    private void Update()
    {
        Vector3 inputMoveDirection = new Vector3(0,0,0);
        if (Input.GetKey(KeyCode.W))
         inputMoveDirection.y = +1f;

        if (Input.GetKey(KeyCode.S))
         inputMoveDirection.y = -1f;

        if (Input.GetKey(KeyCode.A))
         inputMoveDirection.x = -1f;

        if (Input.GetKey(KeyCode.D))
         inputMoveDirection.x = +1f;
        float moveSpeed = 10f;
        Vector3 moveVector = transform.up * inputMoveDirection.y +
transform.right * inputMoveDirection.x;
        transform.position += moveVector * moveSpeed * Time.deltaTime;

        if (Input.mouseScrollDelta.y > 0)
        {
            StopAllCoroutines();
    StartCoroutine(Lerp(cinemachineVirtualCamera.m_Lens.OrthographicSize,
MIN_ZOOM_Y));
        }
        if (Input.mouseScrollDelta.y < 0)
        {
            StopAllCoroutines();
StartCoroutine(Lerp(cinemachineVirtualCamera.m_Lens.OrthographicSize,
MAX_ZOOM_Y));
        }
    }
    IEnumerator Lerp(float start, float end)
    {
        t = 0f;
        while (cinemachineVirtualCamera.m_Lens.OrthographicSize != end)
        {
```

```
            cinemachineVirtualCamera.m_Lens.OrthographicSize =
Mathf.Lerp(start, end, t);
            t += Time.deltaTime;
            yield return null;
        }
        yield return null;
    }
}
```

Isječak programskog koda 17: Logika kamere (vlastita izrada)

4.6.4. Grafičko sučelje

U ovom poglavlju opisuju se svi elementi na ekranu s kojima igrač ima moguću interakciju. Grafičko sučelje ima zadatak prikazati informacije korisniku kako bi mogao donijeti što bolju odluku, informacije poput mogućih akcija, životnih bodova, polja na kojem se nalazi. Jednostavan prikaz grafičkog sučelja vidljivo na slici X. Elementi od kojih se sastoji su dinamička lista tipki Action koje se postave ovisno o odabranom liku te mogućim akcijama prikazano na slici X. Sučelje uvijek zadržava jednak položaj koji kamera prati. Kako bi se igrač mogao lakše snaći, implementiran je proziran prozor koji sadržava dinamični tekst Action Points i opis Polja tj. njegovu poziciju ovisno o lokaciji pokazivača. Na slici X polje ima X,Y,Z vrijednosti od -2,-11,0, te odabrani lik ima 2 Action Points sa mogućnostima Move, Attack i AOE.

Slika 16: Grafičko sučelje s svim elementima (vlastita izrada)

Slika 17: Grafičko sučelje odabranog lika (vlastita izrada)

MenuManager upravljač zadužen je za ažuriranje podataka vezani za polje i odabranog lika. Utječe na ispis koji se nalazi u prozirnim lijevim elementima pri čemu su te informacije više informativnog i razvojnog značenja.

```
public class MenuManager : MonoBehaviour
{
    public static MenuManager Instance;
    [SerializeField] private GameObject _selectedAllyObject,
_selectedTileObject;
    void Awake()
    {
        Instance = this;
    }

    public void ShowSelectedAlly(BaseAlly ally)
    {
        if(ally == null)
        {
            _selectedAllyObject.SetActive(false);
            return;
        }
        _selectedAllyObject.GetComponentInChildren<Text>().text =
ally.name;
        _selectedAllyObject.SetActive(true);
    }
```

```
    public void ShowSelectedEnemy(BaseEnemy enemy)
    {
        if (enemy == null)
        {
            _selectedAllyObject.SetActive(false);
            return;
        }
        _selectedAllyObject.GetComponentInChildren<Text>().text =
enemy.name;
        _selectedAllyObject.SetActive(true);
    }
    public void ShowSelectedTile(OverlayTile tile)
    {
        if (tile == null)
        {
            _selectedTileObject.SetActive(false);
            return;
        }
        string tileInfo = tile.name + " grid position: " +
tile.gridLocation;
        _selectedTileObject.GetComponentInChildren<Text>().text =
tileInfo;
        _selectedTileObject.SetActive(true);

    }
}
```

Isječak programskog koda 18: Grafičko sučelje (vlastita izrada)

UnitActionUI omogućuje generiranje tipki koji pritiskom na njih daju igraću sposobnost odabira. Tipke se generiraju dinamički ovisno koliko mogućnosti odabrani lik ima, i svaki put na odabir se odašilje obavijest odabranoj akciji kako bi se koristila.

```
public class UnitActionUI : MonoBehaviour
{
    [SerializeField] private Transform actionButtonPrefab;
    [SerializeField] private Transform actionButtonContainerTransform;
    [SerializeField] private TextMeshProUGUI actionPointsText;
    private void Start()
    {
        UnitManager.Instance.OnSelectedUnitChange +=
UnitManager_OnSelectedUnitChange;
```

```
        UnitManager.Instance.OnActionStarted +=
UnitManager_OnActionStarted;

        TurnSystem.Instance.OnTurnChange += TurnSystem_OnTurnChange;

        BaseUnit.OnAnyActionPointsChanged +=
BaseUnit_OnAnyActionPointsChanged;


    }

    private void CreateUnitActionButtons()

    {

        foreach (Transform buttonTransform in
actionButtonContainerTransform)

        {

            Destroy(buttonTransform.gameObject);

        }

        Ally1 selectedUnit = UnitManager.Instance.GetSelectedUnit();

        Enemy1 selectedEnemyUnit =
UnitManager.Instance.GetSelectedEnemyUnit();


        if(selectedEnemyUnit != null)

        {

            foreach (BaseAction baseAction in
selectedEnemyUnit.GetBaseActionArray())

            {

                Transform actionButtonTransform =
Instantiate(actionButtonPrefab, actionButtonContainerTransform);

                ActionButtonUI actionButtonUI =
actionButtonTransform.GetComponent<ActionButtonUI>();

                actionButtonUI.SetBaseAction(baseAction);

            }

        }

        if(selectedUnit != null)

        {

            foreach (BaseAction baseAction in
selectedUnit.GetBaseActionArray())

            {

                Transform actionButtonTransform =
Instantiate(actionButtonPrefab, actionButtonContainerTransform);

                ActionButtonUI actionButtonUI =
actionButtonTransform.GetComponent<ActionButtonUI>();

                actionButtonUI.SetBaseAction(baseAction);

            }

        }


    }
```

```
    private void UnitManager_OnSelectedUnitChange(object sender,
EventArgs e)
    {
        CreateUnitActionButtons();
        UpdateActionPoints();
    }

}
```

Isječak programskog koda 19: Logika sučelja za akcije (vlastita izrada)

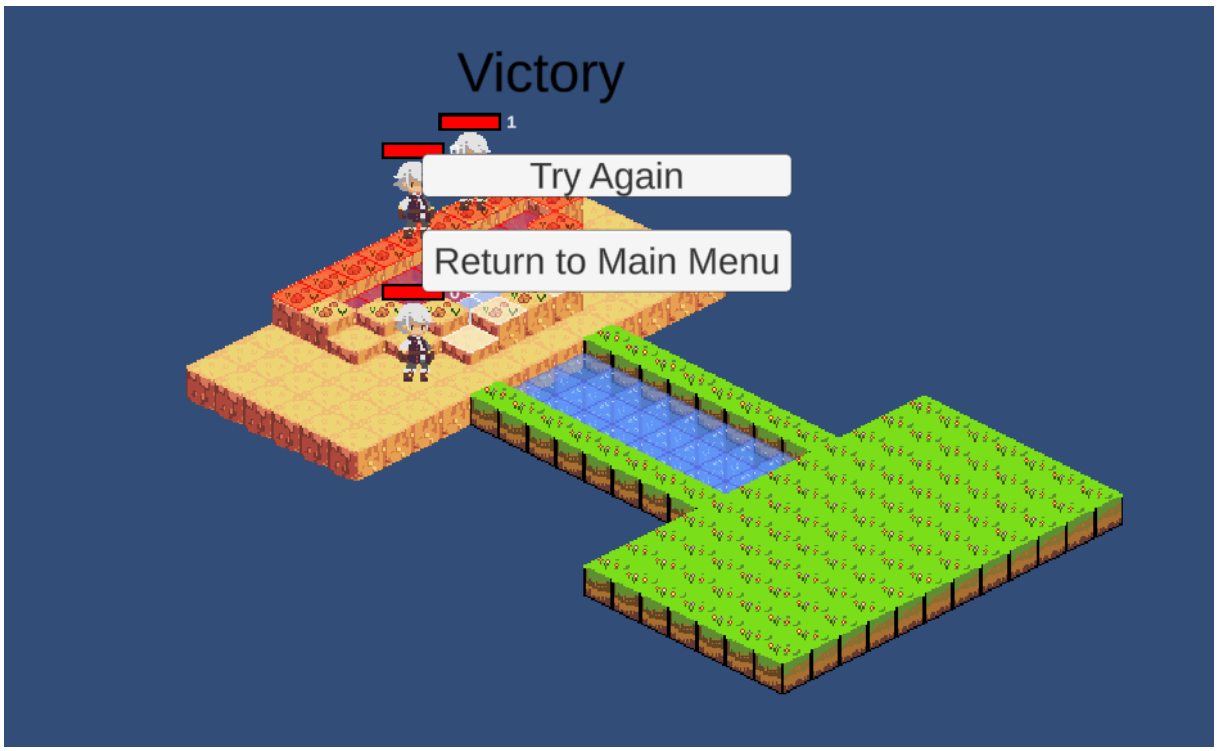
TurnSystemUI omogućuje izmjenu poteza između igraća i računala. Svaki put na pritisak kraj poteza odašilje se obavijest koji izmjeni stanje igre. Naravno izmjene tko je na potezu i koji je broj poteza u igri se također svaki put ažuriraju tokom obavijesti.

```
public class TurnSystemUI : MonoBehaviour
{
    [SerializeField] private Button endTurnBtn;
    [SerializeField] private TextMeshProUGUI turnNumberText;

    private void Start()
    {
        endTurnBtn.onClick.AddListener(() =>
        {
            TurnSystem.Instance.NextTurn();
        });

        TurnSystem.Instance.OnTurnChange += TurnSystem_OnTurnChange;
        UpdateTurnText();
        UpdateEndTurnButtonVisibility();
    }

    private void TurnSystem_OnTurnChange(object sender, EventArgs e)
    {
        UpdateTurnText();
        UpdateEndTurnButtonVisibility();
    }

    private void UpdateTurnText()
    {
        string faction = "Ally";
        if (!TurnSystem.Instance.IsPlayerTurn()) faction = "Enemy";
```

```
            turnNumberText.text = faction.ToUpper() + " TURN " +
TurnSystem.Instance.turnNumber;
        }
        private void UpdateEndTurnButtonVisibility()
        {

endTurnBtn.gameObject.SetActive(TurnSystem.Instance.IsPlayerTurn());
        }
}
```

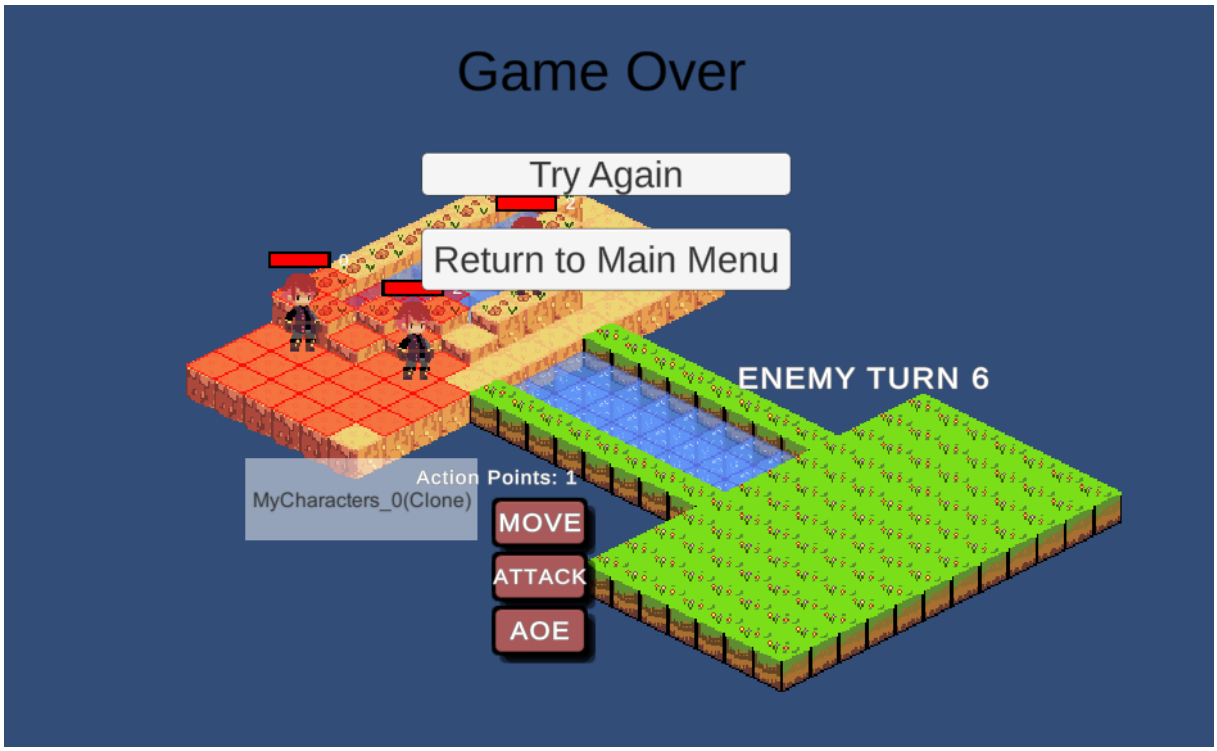
Isječak programskog koda 20: Sučelje za kraj poteza (vlastita izrada)

Kraj scenarija se događa u dva slučaja, jedan slučaj kad igrač pobjedi, drugi kad izgubi. Pobjeda se događa u trenutku kad na mapi ne postoji više ni jedan neprijateljski lik, dok gubitak kad na mapi ne postoji više ni jedan lik igrača. U oba slučaja videoigra staje, pojavljuje se prikaz s porukom Pobjeda ili gubitak te dvije tipke koje omogućuju igraču ponoviti scenarij ili ipak otići na glavni izbornik. Na sljedećoj slici može se vidjeti nedostatak neprijateljskih likova što rezultira kraj videoigre pri čemu je igrač pobijedio jer neprijatelj nema više likova.



Slika 18: Ekran pobjeda (vlastita izrada)

Na sljedećoj slici vidi se kako na mapi ne postoji niti jedan lik od igrača dok postoje tri neprijateljska lika, nedostatak likova od igrača rezultira gubitkom igre i prikazom ekrana kraj.



Slika 19: Kraj Igre (vlastita izrada)

Metode *Restart()* i *MainMenu()* se pozivaju pritiskom na tipke koje u sebi sadržavaju već postojeći okidač *On Click()* te se podešava preko Unity Button Inspector. *GameOver()* i *Victory()* metode se pozivaju u trenutku kad *UnitManager* dobije informaciju da lista protivnika ili lista likova od igraća imaju broj elemenata manje od jedan.

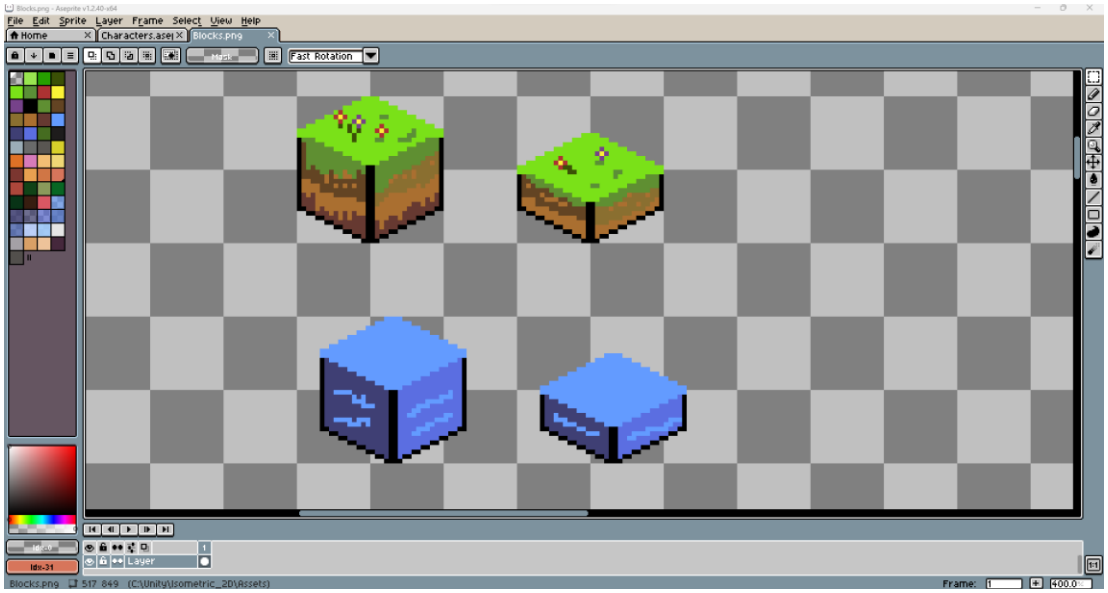
```
//UnitManager.cs
if (enemyUnitList.Count < 1)
GameManager.Instance.UpdateGameState(GameState.Victory);
if (allyUnitList.Count < 1)
GameManager.Instance.UpdateGameState(GameState.GameOver);
//GameManager.cs
    public void GameOver()
    {
        Time.timeScale = 0;
        MenuManager.Instance.gameOverScreen.SetActive(true);
    }
    public void Victory()
    {
        Time.timeScale = 0;
        MenuManager.Instance.victoryScreen.SetActive(true);
    }
    public void Restart()
    {
        Time.timeScale = 1;
```

```
SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
    }
    public void MainMenu()
    {
        Time.timeScale = 1;
        SceneManager.LoadScene(0);

    }
```

Isječak programskog koda 21: Kraj videoigre (vlastita izrada)

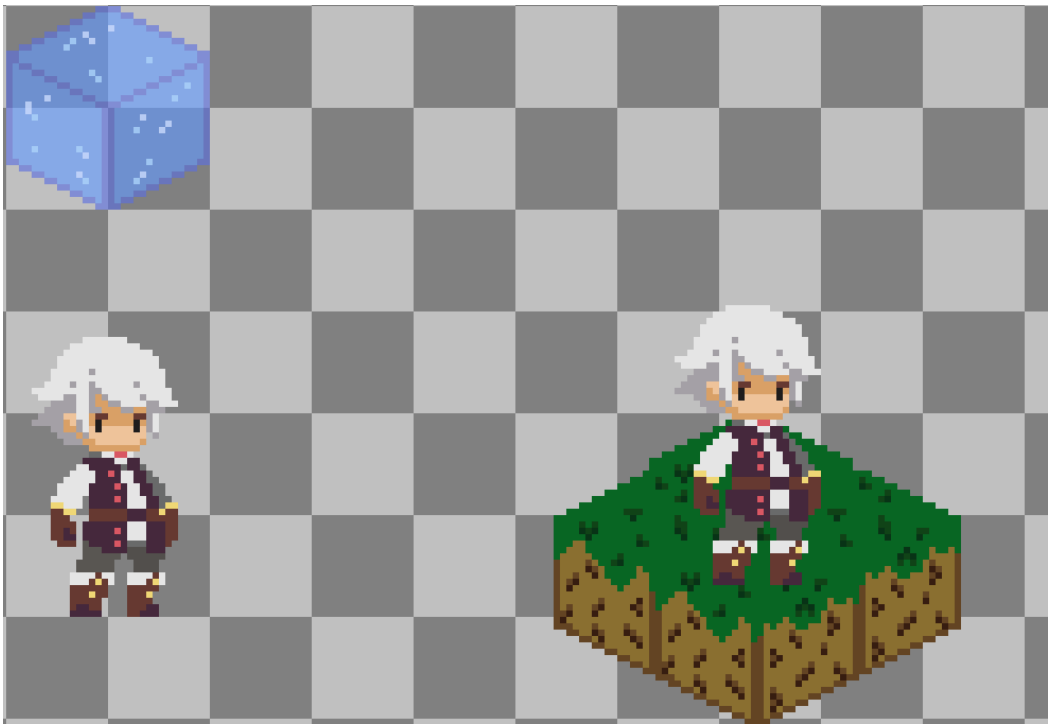
4.6.5. Aseprite izrada materijala

Kao što je opisano u ranijem poglavlju Aseprite je alat za crtanje, u ovom slučaju izrada blokova i likova. Aseprite pruža jednostavno i intuitivno korištenje pri čemu se slike vrlo jednostavno prebace u Unity kako bi se mogle koristiti za izradu videoigre. Kao što je prikazano na sljedećoj slici, alat se sastoji od paleta boja pri čemu možemo dodavati neograničen broj vlastitih boja. Sastoji se od dodatnih alata na desnoj strani kao što su debljina crte, brisanje, prepoznavanje boje, ispunjavanje oblika i sl. Na sredini se nalazi glavni prikaz na kojem se odvija crtanje, te vidljiva mreža kvadratića označeni crno sivom bojom koji imaju dimenzije 16x16 piksela. Kao što je ranije spomenuto u izradi ove videoigre korišten je isometric stil pri čemu su blokovi 32x32 piksela što je i vidljivo na sljedećoj slici jer jedan blok zauzima točno četiriju kvadratića.



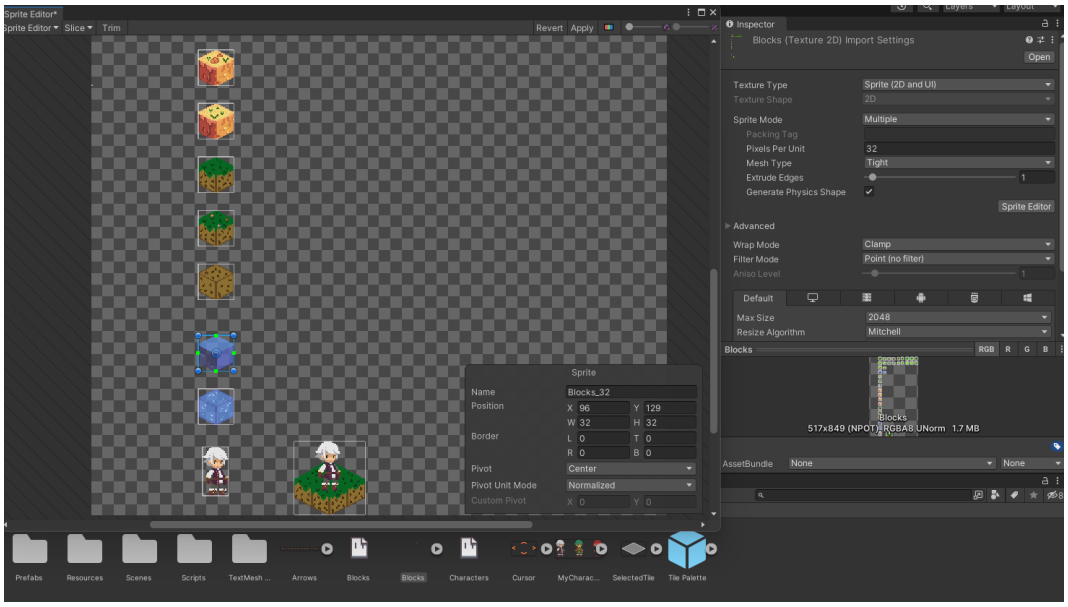
Slika 20: Sučelje alata Aseprite (vlastita izrada)

Na isti način su izrađeni i likovi pri čemu je razlika dimenzija lika. Prikazano na sljedećoj slici se vidi usporedba koliko blok zauzima piksela a koliko je lik zapravo različit.



Slika 21: Usporedba lika i bloka (vlastita izrada)

Nakon što je slika gotova te se prebaci unutar Unity-a potrebno ju je podesiti na način kao što je prikazano na sljedećoj slici. Bitne postavke su Pixels Per Unit, Sprite Mode Multiple jer je riječ o jednoj slici s više blokova, Wrap Mode Clamp te u Sprite Editor treba Slice podesiti ručno u slučaju da želimo sami odrediti pivot i veličinu bloka ili odabirom automatic kako bi Unity sam odredio za svaki blok njegov slice.



Slika 22: Prilagođavanje Asperite slike unutar Unity-a (vlastita izrada)

Nakon što su blokovi podešeni treba ih dodati u paletu blokova koja je prikazano u ranijem poglavlju. Paleta se može kreirati pritiskom Create->2D->Tile Palette->Isometric.

5. ZAKLJUČAK

Područje ovog diplomskog rada je proces izrade videoigre, konkretno minijaturna verzija sveobuhvatnog procesa koji uključuje od izrade likova, dodavanje glazbe pa do samog programiranja. Na početku je objašnjena tematike videoigre kako je riječ o strateškoj videoigri na poteze, te je prikazano i objašnjen stil crtanja što u ovom slučaju predstavlja isometric stil.

Upotrebom alata kao što su Aseprite i Unity, u kombinaciji s Microsoft Visual Studiom, demonstrirana je sposobnost samostalnog razvijanja funkcionalne videoigre. U radu su istraženi i objašnjeni alati koji su korišteni tijekom procesa izrade te je naglasak stavljen na njihovu funkcionalnost i doprinos ukupnom procesu razvoja videoigre.

Kroz praktičan primjer korišteni su svi nabrojani alati, opisane funkcionalnosti te prikazane kako djeluju. Prikazano je kako izrada videoigre na poteze uključuje niz koraka kako bi se videoigra mogla igrati, posjedovati početak i kraj. Koraci su uključivali izradu mape na kojoj se videoigra odvija, postavljanje likova, akcije koje likovi mogu poduzeti, matematičke algoritme za efikasnost, jednostavna umjetna inteligencija pa sve do grafičkog sučelja i glazbe.

Kao rezultat praktičnog primjera vidljivo je da izrada videoigre obuhvaća niz širokih područja, uključuje jako puno vremena i truda kako bi se temelji izgradili. Prikazuje kako je potrebna interakcija između svih komponenti, kako se vrijednosti prosljeđuju i prikazuju korisnicima. Naglasak je na interakciji među svim komponentama i svakoj industriji koja to omogućuje kako bi nastao jedan zajednički proizvod videoigre.

Svijet videoigara konstantno raste, jedna takva pojava omogućuje rast svim ostalim industrijama koje su usko povezane što nakraju omogućuje veću kvalitetu videoigre pa tako i bolje iskustvo korisnicima. Videoigre danas ne samo da pružaju zabavu, već imaju potencijal za edukaciju i simulaciju stvarnog svijeta unutar virtualnog okruženja.

6. LITERATURA

[1.] Microsoft Visual Studio (bez dat.) Preuzeto 23.5.2023. s https://visualstudio.microsoft.com/

[2.] Git (bez dat.) Preuzeto 27.5.2023. s https://git-scm.com

[3.]Kinsta (bez dat.) What is Github? Preuzeto 23.5.2023. s https://kinsta.com/knowledgebase/what-is-github

[4.] Statista (bez dat.) Number of games released on Steam worldwide from 2004 to 2022. Preuzeto 23.7.2023. s https://www.statista.com/statistics/552623/number-games-released-steam/

[5.] Emma Taggart (27.5.2022). Understanding Isometric Illustration. Preuzeto 13.6.2023. s https://www.linearity.io/blog/isometric-illustration/

[6.] Aseprite (bez dat.) Preuzeto 25.5.2023. s https://www.aseprite.org/

[7.] Ben Barnhart (20.7.2022) Isometric design: A designer's guide. Preuzeto 27.5.2023. s https://www.linearity.io/blog/isometric-design/

[8.] What is a strategy game? (bez dat.) Preuzeto 18.6.2023. s https://www.kathleenmercury.com/what-is-a-strategy-game.html

[9.] What makes a strategy Game Successful (bez dat.) Preuzeto 23.6.2023. https://www.gamedesigning.org/learn/strategy-game/

[10.] Turn Based Strategy (bez dat.) Preuzeto 23.6.2023. https://tvtropes.org/pmwiki/pmwiki.php/Main/TurnBasedStrategy

[11.] Strategic games (bez dat.) Preuzeto 23.6.2023. https://cramton.umd.edu/econ414/ch02.pdf

[12.] Dan Norton (20.4.2017) real-time vs. turn-based mechanics in learning games Preuzeto 24.6.2023. https://www.filamentgames.com/blog/real-time-vs-turn-based-mechanics-learning-games/

[13.] Unity Documentation (bez dat.). Preuzeto 28.5.2023. s https://docs.unity3d.com/ScriptReference/

[14.] John French (13.2.2020). The right way to pause a game in unity. Preuzeto 28.6.2023 s https://gamedevbeginner.com/the-right-way-to-pause-the-game-in-unity/

[15.] Matej Marek (1.5.2021). How to Build and Test Your Game in Unity. Preuzeto 29.6.2023. s https://medium.com/geekculture/how-to-build-and-test-your-game-in-unity-1eeab1b7937e

[16.] Udemy. (bez dat). Unity Turn-Based Strategy Game: Intermediate C# Coding. Preuzeto 12.4.2023. s https://www.udemy.com/course/unity-turn-based-strategy/

[17.] AdamCYounis youtube (bez datuma). Pixel Art Class. Preuzeto 12.4.2023. s https://www.youtube.com/playlist?list=PLLdxW--S_0h4dlWUpl-TzBp-ulqK3NiM_

[18.] Tarodev youtube (bez datuma). Unity Tutorials. Preuzeto 12.4.2023. https://www.youtube.com/playlist?list=PLKeKudbESdcyRmaa30z-vDBWVV4iz29LB

[19.] John French (22.9.2022). Scriptable Objects in Unity(how and when to use them). Preuzeto 14.4.2023. s https://gamedevbeginner.com/scriptable-objects-in-unity/

[20.] Garegin Tadevosyan (bez dat.) Unity AI Development: A Finite-state Machine Tutorial Preuzeto 23.5.2023 s https://www.toptal.com/unity-unity3d/unity-ai-development-finite-state-machine-tutorial

[21.] Thomas Kesler (9.4.2021). Game Manager.. One Manager to Rule them All. Preuzeto 24.4.2023. s https://foxxthom.medium.com/game-manager-one-manager-to-rule-them-all-1c06afa72b23

[22.] Unity Learn (bez dat.) Preuzeto 5.4.2023. s https://learn.unity.com/

7. POPIS KORIŠTENIH BIBLIOTEKA

[1.] Cinemachine 2.8.9. Preuzeta 5.6.2023. s Unity Package Manager

[2.] TextMeshPro 3.0.6. Preuzeta 23.5.2023. s Unity Package Manager

[3.] Universal RP 12.1.10. Preuzeta 15.4.2023. s Unity Package Manager

8. POPIS SLIKA

9. POPIS PRILOGA

1. Glazba Preuzeta s https://noonsol.net/download/#TRPLINKPROCESSED

10. POPIS ISJEČAKA PROGRAMSKOG KODA