Izrada akcijske 2D videoigre s proceduralnim generiranjem razine u programskom alatu Unity

Rakić, Lara

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Lara Rakić

IZRADA AKCIJSKE 2D VIDEOIGRE S PROCEDURALNIM GENERIRANJEM RAZINE U PROGRAMSKOM ALATU UNITY

ZAVRŠNI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Lara Rakić

Matični broj: 0016149311

Studij: Informacijski i poslovni sustavi - Umreženi sustavi i računalne igre

IZRADA AKCIJSKE 2D VIDEOIGRE S PROCEDURALNIM GENERIRANJEM RAZINE U PROGRAMSKOM ALATU UNITY

ZAVRŠNI RAD

Mentor:

Doc. dr. sc. Mladen Konecki

Varaždin, rujan 2024.

Lara Rakić

Izjava o izvornosti

Izjavljujem da je moj završni 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.

Autorica potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Rad se bavi izradom 2D akcijske videoigre s naglaskom na proceduralno generiranje razine. Proceduralno generiranje napravljeno je pomoću metode binarne prostorne podjele (eng. *Binary Space Partitioning - BSP*). Generiranje razina uključuje stvaranje zidova, podova, platformi, ukrasnih objekata (slika), neprijatelja i izlaza. Rad se također bavi izradom osnovnih mehanika igrice, kretanja i borbe. Također se bavi izradom svih potrebnih sredstava za kreaciju igre (*tilemape* i *spriteova*). Rad će objasniti teorijsku podlogu proceduralnog generiranja razina kao i njezinu povijest kroz primjer igara baziranih na takvom načinu izrade svjetova. Objasnit će programsku podlogu iza stvaranja jedne takve videoigre, kao i sve ostale vještine potrebne za izradu. Detaljno će prikazati razvoj igre u programskom alatu Unity. Kreiranje igrice je multimedijski, slojevit proces koji zahtijeva kombiniranje znanja i vještina iz različitih područja.

Ključne riječi: platformer; videoigra; unity; 2D; proceduralna generacija; pixel art; programiranje

Sadržaj

1. Uvod

Tema ovog završnog rada je predstavljanje koncepta proceduralnog generiranja razine koristeći se programskim alatom Unity [1]. Rad objašnjava teoriju koja je u podlozi proceduralnog generiranja i prikazuje proces izrade video igre. Proceduralno generiranje temelj je ovoga rada i osnova je kreacije raznolikih i velikih svjetova. Poboljšava igrivost same videoigre čineći svako novo iskustvo igranja drugačijim i unikatnim. Ovisno o implementaciji, omogućuje stvaranje izrazito velikih svjetova, prostorno i memorijski, koristeći relativno malo resursa. Proceduralno generiranje je područje mog interesa zbog mogućnosti kreiranja unikatnih i raznolikih iskustava pri svakom ponovnom igranju. Razvoj i stvaranje video igre kreativan je i složen proces koji uključuje znanja i vještina iz raznih područja. Motivacija za odabir ove teme je želja za upotrebom i primjenom svih naučenih vještina na fakultetu. Unity je odabran kao platforma zbog svoje popularnosti i pristupačnosti kao i izrazite količine resursa koji su na raspolaganju. Osim teoretskoga dijela koji stoji iza programiranja video igre, rad opisuje programski kod, cjelokupni razvoj i stvaranje igre uključujući izradu ostalih sadržaja za igricu.

2. Metode i tehnike rada

Za realizaciju projekta korištena je platforma za razvoj videoigara Unity [1]. Unity je jedna od najpopularnijih platformi za razvoj video igara. Bazirana je na Microsoft C# programskom jeziku i podržava izradu različitih tipova video igara. Zbog velike količine dostupnih resursa kao i potpore 2D igara, Unity je odabran za izradu i ovoga rada. Visual Studio Code [2] korišten je za programski dio projekta. To je popularni editor koda koji ima vrlo dobru integraciju s platformom Unity i podržava veliki broj dodataka.

Istraživanje koje je prethodilo radu provedeno je s ciljem prikupljanja informacija i resursa potrebnih za izradu ovakve videoigre. YouTube je upotrijebljen za pomoć pri korištenju platforme Unity kao i za programska rješenja određenih dijelova projekta. Samo istraživanje o proceduralnom generiranju također je provedeno na YouTubeu. Unity dokumentacija je korištena za pomoć pri snalaženju s tom platformom kao i za rješavanje određenih tehničkih problema. Unity Discussions [3] i stackoverflow [4] korišteni su za pronalaženje ideja o određenim programskim rješenjima.

Resursi korišteni za igricu nađeni su na itch.io (*tilemap*) [5], dafont (font) [6], Unity Asset Store (*spritesheet*) [11], canva.com (generiranje pozadine glavnog izbornika) [7], craftpix (*pixel art*) [8] i Lospec (palete boja) [9] [10].

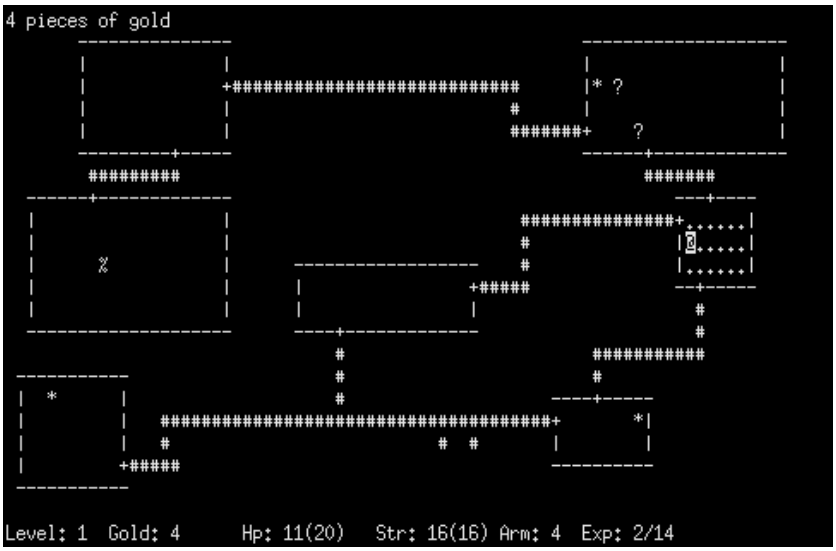
3. Razrada teme

Razrada teme je podijeljena na tri dijela. Prvi dio bavit će se proceduralnim generiranjem te će ukratko opisati njegovo korištenje u video igricama. Također će proći teorijsku podlogu poput algoritama i osnovnih principa koji su u podlozi generiranja sadržaja. Drugo poglavlje posvećeno je razvoju igre unutar Unity [1] okruženja te će prikazati proces postavljanja igre, uključujući izradu grafičkih elemenata i namještanje objekata. Bit će objašnjeni koraci za postavljanje osnovnih elemenata igre. Treći dio rada fokusira se na programski dio projekta, gdje će biti opisane skripte i algoritmi korišteni za sve funkcionalnosti igre. Pojasnit će se struktura i logika koda te će biti objašnjeno kako oni doprinose cjelokupnoj igrici.

3.1. Proceduralno generiranje i video igre

U području razvoja videoigara proceduralno generiranje metoda je kreiranja podataka pomoću algoritama i s određenim stupnjem nasumičnosti. Umjesto ručnog postavljanja svakog elementa, ovakvo generiranje omogućuje stvaranje velike količine različitog sadržaja. Ono omogućuje da je gotovo svako pokretanje igre drugačije.

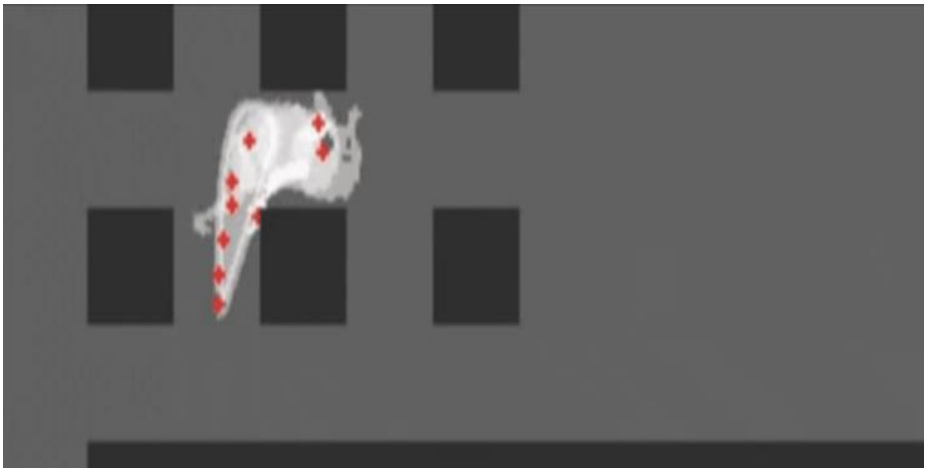
Proceduralno generiranje ima povijest korištenja u video igrama od njihovih početaka. Rogue je jedna od najpoznatijih videoigara koja je karakterizirana svojim proceduralno generiranim razinama. Prema toj igrici nastao je žanr video igara – „*rogue-like*“. Ograničenje memorije i mogućnost ponovnog igranja uvijek je bilo izrazito važno programerima videoigara, pogotovo u početnim danima kad je memorija bila smanjenog kapaciteta. Danas su računala naprednija, ali proceduralno generiranje i dalje se koristi baš zbog te raznolikosti i dinamičnog stvaranja sadržaja pri svakom novom igranju. [13]



Slika 1: Prikaz videoigre Rouge [14]

Velika inspiracija za dizajn proceduralno generiranih razine je proizašla iz želje da se rekreira društvena igra Dungeons & Dragons. Nju karakterizira izrazita mogućnost ponovnog igranja kroz beskonačan broj izbora, svjetova, neprijatelja i odluka koje igrači mogu donijeti. [13]

Osim proceduralnog generiranja razina, kod video igara se također pojavljuje proceduralna animacija. Nju karakteriziraju animacije koje nisu predefinirane napravljene već se generiraju u stvarnom vremenu. Videoigra Rain World koristi proceduralnu animaciju kako bi učinila svoj svijet realističnijim. U njoj, svi likovi unutar te igre imaju ugrađene „zglobove“ unutar svog tijela. Pomoću raznih algoritama i izračunavanja fizike, sve animacije se odvijaju u stvarnom vremenu, prirodno kao da su stvarna bića koja postoje i izvan okvira igre. [15], [16]



Slika 2: Prikaz proceduralne animacije (https://medium.com/@merxon22/recreating-rainworlds-2d-procedural-animation-part-1-4d882f947e9f)

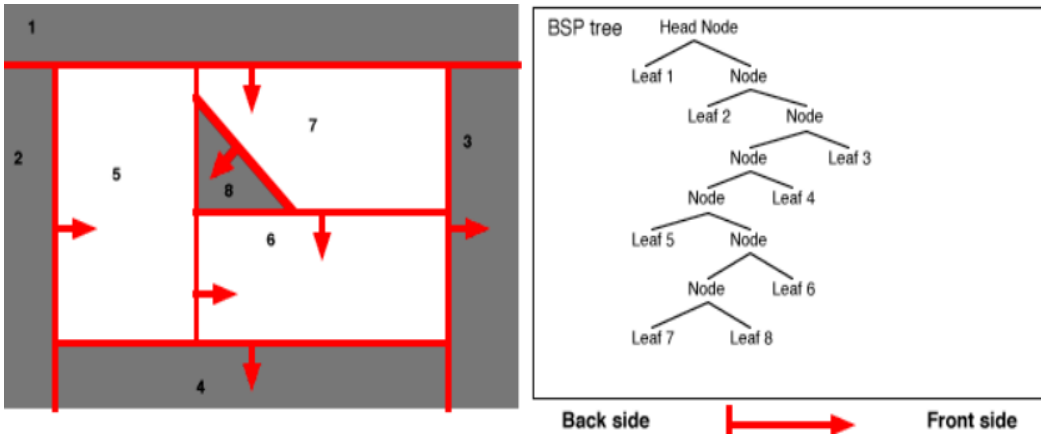
3.1.1. Binarna prostorna podjela

BSP ili binarna prostorna podjela (eng. *Binary Space Partitioning*) tehnika je podjele prostora koja se koristi za generiranje razina videoigara. Ta metoda radi na principu dijeljenja prostora (npr. razine igre) na dvije polovice pomoću jedne ravnine ili crte. Taj se postupak zatim ponavlja na svakoj od nastalih polovica sve dok se prostor ne podijeli na dovoljno male dijelove koji su jednostavni za upravljanje.

Najčešće se koristi za generiranje razine kod proceduralnog generiranja. Generira sobe podjelom veliki prostor na manje razine. Te podjele mogu biti nasumične i time postižu potrebnu razinu raznolikosti za proceduralno generiranje. Ovaj rad upotrebljava binarnu prostornu podjelu u tu svrhu. Nakon što se prostor podijeli, pomoću novonastalih pozicija mogu se generirati ostali potrebni dijelovi soba. [19]

Takva podjela se koristi i u drugim područjima poput računalne geometrije i 3D renderiranja. Osnovna ideja binarne prostorne podjele je da se kompleksni prostori ili scene podijele na jednostavnije, manje dijelove. Svaka podjela rezultira BSP stablom (eng. *BSP tree*), koje je struktura podataka korištena za organiziranje tih podjela. U BSP stablu, čvorovi predstavljaju podjele prostora, a listovi predstavljaju konačne, nepodijeljene dijelove prostora. Na taj način, svaki dio prostora može biti lako pristupačan i obrađivan tijekom igranja ili renderiranja. Ova tehnika omogućava učinkovitije upravljanje složenim scenama. [17], [18]

U ovom radu koristi se samo tehnika podjele, a rezultati se pohranjuju u listu, ne u BSP stablo.



Slika 3: Prikaz BSP stabla generiranog iz geometrijskog prostora (https://developer.valvesoftware.com/wiki/Binary_space_partitioning)

3.2. Razvoj igre

Ovo se poglavlje bavi razvojem videoigre kroz sve stadije, od ideje do postavljanja scene i korištenja programskog alata Unity. Programski dio rada bit će objašnjen u drugom poglavlju. Razvoj igre uključuje osmišljavanje početne ideje, pronalazak svih potrebnih resursa i stvaranje scene unutar Unity sučelja. Izrazito je potrebno paziti da svi resursi videoigre (igrač, pozadina, animacije, neprijatelji itd.) čine tematsku i koherentnu cjelinu te da nijedan dio ne odskače. Dio je resursa kreiran vlastoručno, a dio je pronađen na internetu te kasnije uređen tako se bolje uklapa u projekt.

3.2.1. Ideja igre

Prije početka kreiranja videoigre potrebno je utvrditi osnovnu ideju na kojoj će se ona temeljiti. Ta će ideja voditi cijeli proces razvoja, od dizajna i mehanike do vizualnog stila i grafike. Ideja videoigre iz ovog rada temelji se na klasičnom konceptu bijega iz tamnice. Ovaj rad je 2D akcijski platformer i ima kameru postavljenu sa strane. Dojam koji želi ostaviti je sličan igrama poput *Dead Cells* [20] i *Hollow Knight*. [21]

Slika 4: Prikaz videoigre Dead Cells [22]

Kroz igru, igrač će prolaziti niz povezanih soba, skakati po platformama i boriti se s neprijateljima. Cilj je igre doći do izlaza iz tamnice, a proceduralno generiranje osigurat će da svako pokretanje igre bude drugačije. Takvo stvaranje razina osigurava da igrač ne može zapamtiti izlaz iz tamnice te će osigurati bolju igrivost i angažman od strane igrača. Igra se odvija u tamnici te je potrebno grafičke elemente prilagoditi toj temi. Stil treba podsjećati na srednji vijek ili *fantasy* žanr. Glavni lik bit će vitez, a njegovi neprijatelji kosturi.

3.2.2. Grafički elementi

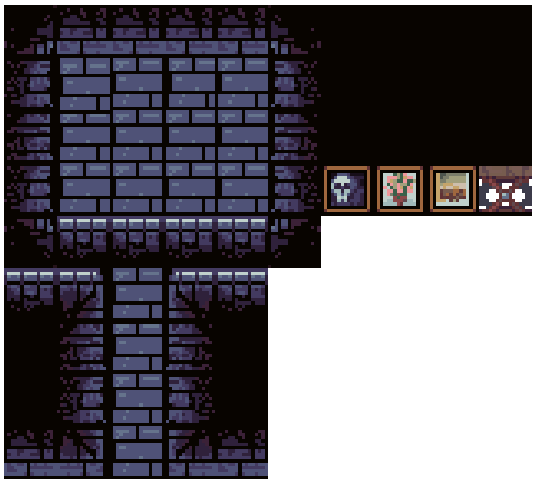
Grafički ili vizualni elementi su oni koje igrač vidi. Oni omogućuju interakciju između igrača i videoigre te stvaraju početne i trajne dojmove. Izrazito je bitno uskladiti i povezati grafičke elemente sa cijelom tematikom videoigre kao i međusobno. Prvi je korak odabir teme i ideje videoigre, u ovom slučaju to je srednji vijek. Izabrana paleta boja zato ima tamne, plave nijanse koje odaju dojam tamnice.

Slika 5: Prikaz paleta korištenih u igrici [9], [10]

Raspon boja je širok, od svijetlog do tamnog da bi se omogućio kontrast, ali i raznolikost sadržaja. Boje su odabrane sa svrhom prikaza kamenih zidova tamnice, srednjovjekovnog oklopa glavnog lika i smeđe boje platformi.

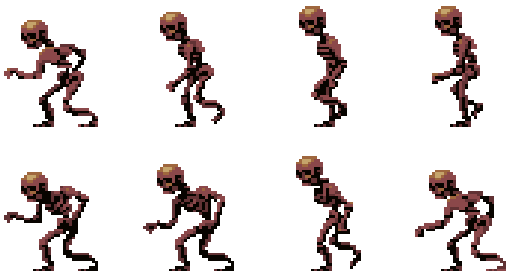
Svi su vizualni elementi igre ili ručno kreirani, ili preuzeti s platformi poput platformi itch.io [5] i Unity Asset Store. [11] Sprajtovi igrača i neprijatelja preuzeti su na internetu, a pozadina, zidovi, slike i platforme kreirane su vlastoručno. Sprajtovi su prilagođeni kako bi odgovarali paleti. Životna traka (eng. *Healthbar*) i drugi elementi sučelja stvoreni su u potpunosti pomoću već postojećih alata u Unityju. Pozadina glavnog izbornika kreirana je pomoću Canvas AI alata. [7] Svi sprajtovi su ili nacrtani ili uređeni pomoću alata Aseprite. [12]

Slika 6: Prikaz jedne od animacija glavnog lika [8]



Slika 7: Prikaz sprajtova zidova, pozadine, tla, slika i platforme

Slika 8: Prikaz vrata izlaza iz tamnice



Slika 9: Prikaz jedne animacije glavnog neprijatelja [11]

Slika 10: Prikaz životne trake igrača

Slika 11: Prikaz ekrana opcija unutar igrice

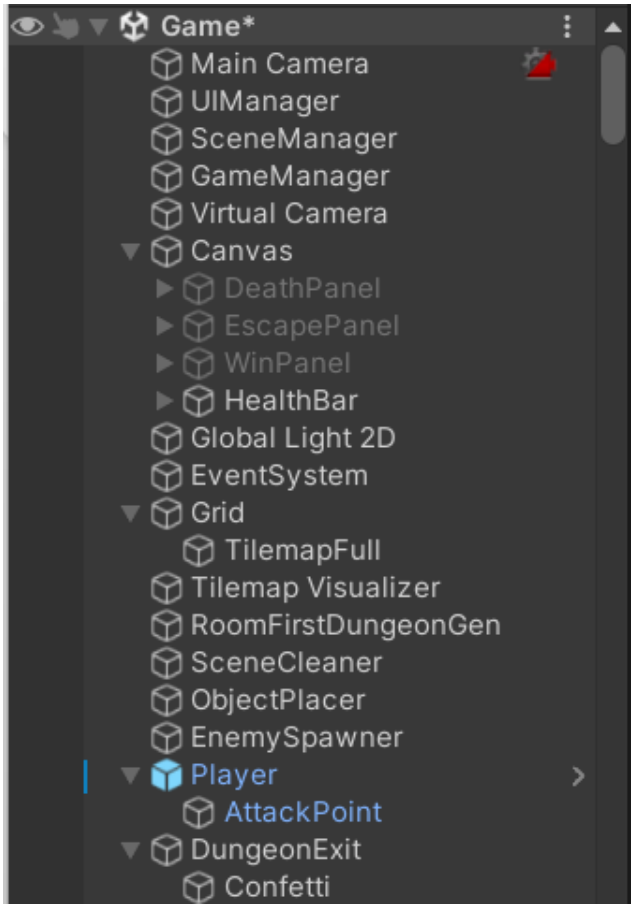
Slika 12: Prikaz glavnog izbornika igrice

U ovom projektu, sprajtovi igrača i neprijatelja su veličine 32 piksela, dok su pozadinski elementi i dekoracijski objekti veličine 16 piksela. Korišten je *no-point filter* (bez interpolacije), što znači da će sprajtovi biti prikazani s oštrim rubovima i vidljivim pikselima, zadržavajući klasični retro izgled, bez zamućenja koje bi inače moglo nastati prilikom skaliranja.

3.2.3. Objekti unutar scene

Scena unutar programskog alata Unity prikazuje sve elemente i objekte igre. U ovom radu postoje dvije scene, jedna prikazuje glavni dio videoigre gdje igrač upravlja glavnim likom, a druga je glavni izbornik.

Unutar scene „Game“ nalaze se svi potrebni objekti za funkcioniranje igre. Od objekta igrača do kamera, pomoću scene možemo njima upravljati i dodijeliti im skripte za daljnje upravljanje. Ostali objekti poput neprijatelja i platformi generiraju se tijekom stvaranja razine.

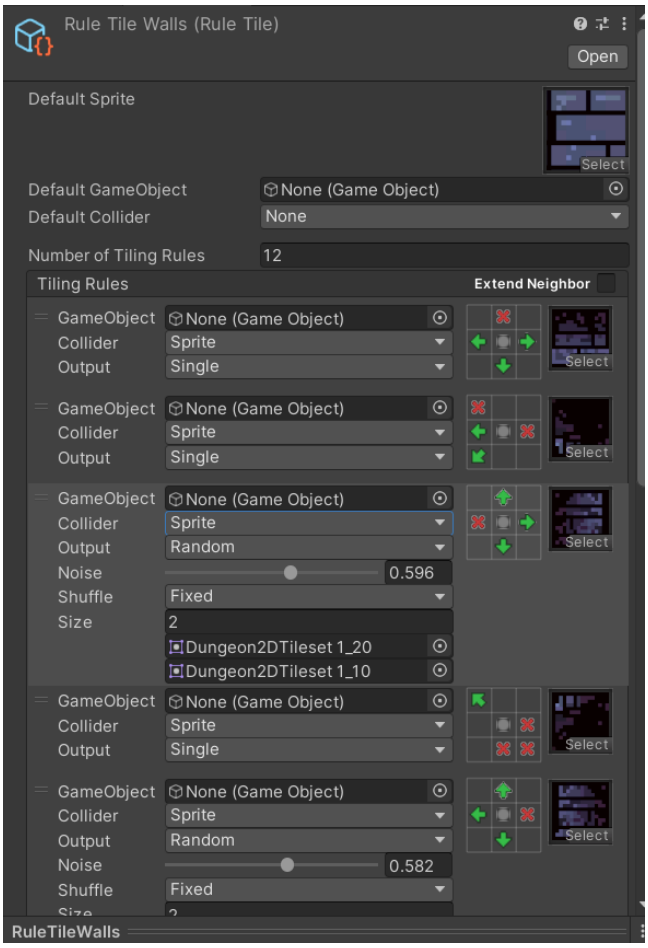


Slika 13: Prikaz scene „Game“

3.2.4.Upravljanje generiranjem razine

Skripta zaslužna za generiranje tamnice šalje sve potrebne podatke *Tilemap Visualizer* objektu i njegovoj skripti kako bi omogućila pravilno postavljanje pozadine i zidova. Takva implementacija omogućuje jednostavnije i automatizirano generiranje grafičkog prikaza tamnice. Koristeći pločice (eng. *Tiles*), taj objekt ih postavlja na *tilemap* objekt *TilemapFull* koji se nalazi unutar *grida*. On sadrži sve zidove i sve podove i koristi Unityjev *tilemap* sustav. [23]

Kako bi se olakšalo postavljanje svih pločica koristimo sustav pravila pločica (eng. *Rule Tile*). On omogućuje automatsko određivanje gdje će koja pločica biti postavljena s obzirom na jednu glavnu predefiniranu pločicu. Također omogućuje lagano mijenjanje kolizije. Glavna pločica prema kojoj su definirana pravila je pločica pozadine i ona nema koliziju. Ukupno je 12 pravila na nju definirano, te pločice su sve pločice zidova i imaju koliziju. Takva implementacija omogućuje brže postavljanje terena nego da smo napravili metodu za određivanje gdje će koja pločica biti postavljena. [24]



Slika 14: Prikaz pravila pločica

3.3. Programski kod

Sve su skripte u ovome radu napisane u programskom jeziku C#, pomoću Visual Studio razvojne okoline. Programski kod čini ključni i najzahtjevniji dio ovoga projekta. On omogućuje funkcioniranje cijele igre. Svi su nazivi funkcija, varijabli i skripti na engleskome jeziku zbog konzistentnosti i logične integracije s već postojećim metodama i skriptama.

3.3.1. Proceduralno generiranje

Proceduralno generiranje glavni je i najzahtjevniji dio ovoga rada. Podijeljeno je u više skripti, svaka je zaslužna za jedan dio. Skripta `ProceduralGen` glavna je skripta zaslužna za stvaranje tamnice koristeći BSP tehniku. Funkcija `BinarySpacePartitioning` prima tri ulazna parametra, `spaceToSplit` (pravokutan prostor kojeg će podijeliti na manje soba), `minWidth` (minimalna širina sobe) i `maxWidth` (maksimalna širina sobe). `SpaceToSplit` je `BoundsInt`, to je struktura u UnityEngine koja predstavlja pravokutnik, početno u njoj se nalazi `startPosition (0,0)` i `dungeonWidth` i `dungeonHeight` koji predstavljaju ukupnu veličinu cijele tamnice. [25]

```
public static List<BoundsInt> BinarySpacePartitioning(BoundsInt
spaceToSplit, int minWidth, int minHeight)
    {
        Queue<BoundsInt> roomsQueue = new Queue<BoundsInt>();
        List<BoundsInt> roomsList = new List<BoundsInt>();
        roomsQueue.Enqueue(spaceToSplit);
        while(roomsQueue.Count > 0)
        {
            var room = roomsQueue.Dequeue();
            if(room.size.y >= minHeight && room.size.x >= minWidth)
            {
                if(Random.value < 0.5f)
                {
                    if(room.size.y >= minHeight * 2)
                    {
                        SplitHorizontally(minHeight, roomsQueue, room);
                    }else if(room.size.x >= minWidth * 2)
                    {
                        SplitVertically(minWidth, roomsQueue, room);
                    }else if(room.size.x >= minWidth && room.size.y >=
minHeight)
                    {
```

```
                        roomsList.Add(room);
                    }
                }
                else
                {
                    if (room.size.x >= minWidth * 2)
                    {
                        SplitVertically(minWidth, roomsQueue, room);
                    }
                    else if (room.size.y >= minHeight * 2)
                    {
                        SplitHorizontally(minHeight, roomsQueue, room);
                    }
                    else if (room.size.x >= minWidth && room.size.y >=
minHeight)
                    {
                        roomsList.Add(room);
                    }
                }
            }
        }
        return roomsList;
    }
```

Koristeći `Random.Value`, skripta bira hoće li trenutnu sobu podijeliti horizontalno ili vertikalno. Takva implementacija daje dodatnu razinu nasumičnosti potrebnu za proceduralno generiranje. `xSplit` je vrijednost koja predstavlja gdje će se prostorija podijeliti duž x-osi (horizontalno), a `ySplit` duž y osi (okomito). [26]

```
private static void SplitVertically(int minWidth, Queue<BoundsInt>
roomsQueue, BoundsInt room)
        {
            var xSplit = Random.Range(1, room.size.x);
            BoundsInt room1 = new BoundsInt(room.min, new Vector3Int(xSplit,
room.size.y, room.size.z));
            BoundsInt room2 = new BoundsInt(new Vector3Int(room.min.x +
xSplit, room.min.y, room.min.z),
            new Vector3Int(room.size.x - xSplit, room.size.y, room.size.z));
            roomsQueue.Enqueue(room1);
            roomsQueue.Enqueue(room2);
        }
```

```
private static void SplitHorizontally(int minHeight, Queue<BoundsInt>
roomsQueue, BoundsInt room)
          {
             var ySplit = Random.Range(1, room.size.y);
             BoundsInt room1 = new BoundsInt(room.min, new
Vector3Int(room.size.x, ySplit, room.size.z));
             BoundsInt room2 = new BoundsInt(new Vector3Int(room.min.x,
room.min.y + ySplit, room.min.z),
             new Vector3Int(room.size.x, room.size.y - ySplit, room.size.z));
             roomsQueue.Enqueue(room1);
             roomsQueue.Enqueue(room2);
          }
      }
```

3.3.1.1. Generiranje soba

Skripta `RoomFirstDungeonGen` zaslužna je za stvaranje hodnika i soba. Ona generira njihove pozicije tako što sve proceduralno generirane sobe smanjuje, a te novonastale sobe sprema zasebno u varijablu `background`. Sobe povezuje pomoću hodnika i njihove vrijednosti sprema u `corridors`. Takva implementacija je korisna za daljnje upravljanje objektima jer odvaja sve potrebne pozicije u zasebne liste. Na vrhu klase `RoomFirstDungeonGen` deklarirane su varijable za veličinu tamnice i soba kao i varijabla `offset`. Ona pak određuje pomak između prostora koji BSP stvara i stvarnog postavljanja pozadine.

```
[Header("Room variables")]
[SerializeField] private int minRoomWidth = 4, minRoomHeight = 4;
[SerializeField] private int dungeonWidth = 20, dungeonHeight = 20;
private int offset = 2;
```

Klasa `RoomFirstDungeonGenerator` nasljeđuje klasu `AbstractDungeonGenerator` te implementira apstraktnu metodu `RunProceduralGen()`. Takva implementacija potencijalno omogućuje dodavanje više metoda za proceduralno generiranje. Ovaj se rad fokusira samo na tehniku binarne prostorne podjele. [27]

```
protected override void RunProceduralGen()
{
CreateRooms()
}
```

`CreateRooms()` poziva sve metode potrebne za generiranje cijele tamnice, uključujući stvaranje hodnika, pozadine, zidova, neprijatelja, izlaza, igrača i ostalih ukrasnih objekata. Nakon što se generiraju granice svih prostorija unutar `RoomFirstDungeonGen` skripte, poziva se metoda `CreateSimpleRooms` koja određuje pozicije pozadine i sprema te pozicije u `HashSet<Vector2Int> background`. Takav način spremanja omogućuje kasnije slanje podataka skripti odgovornoj za postavljanje pločica (eng. *Tiles*) kao i skripti odgovornoj za postavljanje objekata unutar soba. Lista `roomCenters` sadrži centre svih soba i ona je korištena za kreiranje hodnika. `HashSet<Vector2Int> corridorsOutsideRooms` sprema pozicije samo hodnika bez preklapanja sa sobama kako bi se mogli generirati ukrasni objekti slika u hodniku, ali ne i u sobama. [27]

```
private void CreateRooms()
{
    var roomsList = ProceduralGen.BinarySpacePartitioning(
    new BoundsInt((Vector3Int)startPosition, new
Vector3Int(dungeonWidth, dungeonHeight, 0)),
    minRoomWidth, minRoomHeight
    );

    HashSet<Vector2Int> background = new HashSet<Vector2Int>();

    background = CreateSimpleRooms(roomsList);

    List<Vector2Int> roomCenters = new List<Vector2Int>();
    foreach (var room in roomsList)
    {
        roomCenters.Add((Vector2Int)Vector3Int.RoundToInt(room.center));
    }

    objectPlacer.PlacePlayer(roomsList);
    objectPlacer.PlaceDoor(roomsList);
    objectPlacer.PlaceLaddersRooms(roomsList);
    HashSet<Vector2Int> corridors = ConnectRooms(roomCenters);
HashSet<Vector2Int> corridorsOutsideRooms = new
HashSet<Vector2Int>(corridors);
    foreach (var roomCenter in roomCenters)
    {
        corridorsOutsideRooms.Remove(roomCenter);
    }
```

```
    objectPlacer.LadderPlacer(corridorsOutsideRooms);

    objectPlacer.PicturePlacer(corridorsOutsideRooms);
    objectPlacer.PlaceEnemySpawners(roomsList);

    background.UnionWith(corridors);

    enemySpawner.SpawnEnemiesAtSpawners();

    tilemapVisualizer.PaintBgTiles(background);
    WallGenerator.CreateWalls(background, tilemapVisualizer);
}
```

Metoda `ConnectRooms` spaja sve centre susjednih soba. Počinje nasumičnim odabirom centra sobe iz liste svih centara, zatim poziva `FindClosestTo` da nađe najbližu sobu tom centru. To omogućuje dodatnu nasumičnost. Kada nađe najbližu sobu, spaja ju s centrom pomoću `CreateCorridor`. Taj nasumično odabran centar se zatim briše iz liste svih centara kako ne bi došlo do situacije u kojem se uspoređuje udaljenost od samog sebe. Spajanjem soba stvaraju se pozicije hodnika. [28]

```
private HashSet<Vector2Int> ConnectRooms(List<Vector2Int> roomCenters)
{
    HashSet<Vector2Int> corridors = new HashSet<Vector2Int>();
    var currentRoomCenter = roomCenters[Random.Range(0,
roomCenters.Count)];
    roomCenters.Remove(currentRoomCenter);

    while (roomCenters.Count > 0)
    {
        Vector2Int closest = FindClosestPointTo(currentRoomCenter,
roomCenters);
        roomCenters.Remove(closest);
        HashSet<Vector2Int> newCorridor = CreateCorridor(currentRoomCenter,
closest);
        currentRoomCenter = closest;
        corridors.UnionWith(newCorridor);
    }
    return corridors;
}
```

`FindClosest` metoda koristi `Vector2.Distance` da izračuna udaljenost od trenutne sredine sobe do centra najbliže susjedne sobe. Taj se proces ponavlja iterativno kroz sve preostale sredine soba dok ne nađe sobu najbližu trenutnoj i vraća njezin `Vector2Int`.

```
private Vector2Int FindClosestPointTo(Vector2Int currentRoomCenter,
List<Vector2Int> roomCenters)
{
    Vector2Int closest = Vector2Int.zero;
    float distance = float.MaxValue;
    foreach (var position in roomCenters)
    {
        float currentDistance = Vector2.Distance(position,
currentRoomCenter);
        if(currentDistance < distance)
        {
            distance = currentDistance;
            closest = position;
        }
    }
    return closest;
}
```

`CreateCorridor` stvara sve pozicije koje čine hodnik. Ona se služi `ApplyBrush` funkcijom koja omogućuje šire hodnike. Bez `ApplyBrush` hodnici bi bili široki samo jedan blok (predstavljen koordinatama, x i y). Širi hodnici bolje odgovaraju igrama koje su formata 2D s kamerom sa strane.

```
private HashSet<Vector2Int> CreateCorridor(Vector2Int currentRoomCenter,
Vector2Int destination)
{
    HashSet<Vector2Int> corridor = new HashSet<Vector2Int>();
    var position = currentRoomCenter;
    corridor.UnionWith(ApplyBrush(position));

    while (position.y != destination.y)
    {
        if (destination.y > position.y)
        {
            position += Vector2Int.up;
        }
        else if (destination.y < position.y)
        {
```

```
            position += Vector2Int.down;
        }
        corridor.UnionWith(ApplyBrush(position));
    }

    while (position.x != destination.x)
    {
        if (destination.x > position.x)
        {
            position += Vector2Int.right;
        }
        else if (destination.x < position.x)
        {
            position += Vector2Int.left;
        }
        corridor.UnionWith(ApplyBrush(position));
    }

    return corridor;
}

private HashSet<Vector2Int> ApplyBrush(Vector2Int position)
{
    HashSet<Vector2Int> brushArea = new HashSet<Vector2Int>();
    for (int x = -1; x <= 1; x++)
    {
        for (int y = -1; y <= 1; y++)
        {
            brushArea.Add(position + new Vector2Int(x, y));
        }
    }
    return brushArea;
}
```

Kako bi se još smanjile sobe i učinile estetski ljepšima, funkcija `CreateSimpleRooms` koristi razmak (eng. *Offset*) od dvije jedinice. Tu pak nastaje problem jer će kasnije skripte za generiranje objekata i postavljanje pozadina koristiti listu `roomsList` koja je šira nego sobe napravljene u `CreateSimpleRooms`. Zato će se morati ručno dodavati razmak pri postavljanju objekata. Ipak, ovakva implementacija je bolja jer `roomsList` uključuje i pozicije

zidova za razliku od rezultata `CreateSimpleRooms`. Zidovi će biti dodani okolo `background` pozicija.

```
private HashSet<Vector2Int> CreateSimpleRooms(List<BoundsInt> roomsList)
    {
        HashSet<Vector2Int> background = new HashSet<Vector2Int>();
        foreach (var room in roomsList)
        {
            for (int col = offset; col < room.size.x - offset; col++)
            {
                for (int row = offset; row < room.size.y - offset; row++)
                {
                    Vector2Int position = (Vector2Int)room.min + new
Vector2Int(col, row);
                    background.Add(position);
                }
            }
        }
        return background;
    }
```

3.3.1.2. Postavljanje objekata

Postavljanje objekata u kontekstu proceduralnog generiranja razina u video igrama predstavlja složen izazov zbog određene razine nasumičnosti koja se stvara prilikom kreiranja samog terena. Svaka pozicija u virtualnom prostoru mora biti kontrolirana kako bi se osiguralo pravilno postavljanje objekata i izbjeglo preklapanje.

Metoda `LadderPlacer` odgovorna je za postavljanje platformi unutar hodnika na temelju podataka o njihovim pozicijama. Varijable tipa `GameObject` zaslužne su za sve objekte na mapi, u ovom kontekstu varijable traže da se na njih postavi odgovarajući predložak. Varijabla `maxGap` traži razmak između platformi u hodnicima, a `ladderLenght` i `laddersPerRow` su zaslužne za postavljanje platformi u sobama. Varijabla `ladderLength` je širina platforme, a `laddersPerRow` je broj postavljenih platformi u jednom redu sobe. Svaka platforma sastavljena je od više `ladderPrefab` objekata.

```
[Header("Objects")]
[SerializeField] private GameObject ladderPrefab;
[SerializeField] private GameObject picturePrefab;
[SerializeField] private GameObject picturePrefab2;
[SerializeField] private GameObject picturePrefab3;
```

```
[SerializeField] private GameObject enemySpawnerPrefab;
[SerializeField] private GameObject player;
[SerializeField] private GameObject door;
[Header("Vertical corridor ladder placement")]
[SerializeField] private int maxGap = 5;
[Header("Ladder options")]
[SerializeField] private int ladderLength = 3;
[SerializeField] private int laddersPerRow = 1;
```

U ovoj funkciji prvo se filtriraju pozicije koje pripadaju vertikalnim hodnicima od pozicija koje pripadaju horizontalnim hodnicima, zatim se sortiraju prema y-osi (od najveće prema najmanjoj) i x-osi (od najmanje prema najvećoj). Odvajanje hodnika je potrebno jer se igrač kreće horizontalno te neće moći prolaziti visokim, vertikalnim hodnicima bez platformi. Nakon sortiranja, metoda postavlja platforme samo na pozicije koje nisu već zauzete drugim platformama i koje su smještene dovoljno daleko od drugih platformi. Sortiranje je potrebno kako bi se platforme pravilno mogle postaviti po hodniku. Funkcija prolazi kroz sortirane pozicije i na svakoj validnoj poziciji instancira (eng. *Instantiate*) objekt platforme, osiguravajući da platforme ne budu preblizu drugih već postavljenih objekata.

```
public void LadderPlacer(HashSet<Vector2Int> corridorPositions)
    {
        List<Vector2Int> sortedPositions = new List<Vector2Int>();

        foreach (var position in corridorPositions)
        {
            if (IsVerticalCorridor(position, corridorPositions))
            {
                sortedPositions.Add(position);
            }
        }

        sortedPositions.Sort((a, b) =>
        {
            int yComparison = b.y.CompareTo(a.y);
            return yComparison != 0 ? yComparison : a.x.CompareTo(b.x);
        });

        int i = 0;

        while (i < sortedPositions.Count)
```

```
        {
            int currentY = sortedPositions[i].y;
            while (i < sortedPositions.Count && sortedPositions[i].y ==
currentY)
            {
                Vector2Int currentPos = sortedPositions[i];

                if (!IsLadderNeighbourUpAndDown(currentPos))
                {
                    Vector3 ladderPosition = new Vector3(currentPos.x +
0.5f, currentPos.y + 0.5f, 0);
                    Instantiate(ladderPrefab, ladderPosition,
Quaternion.identity);
                    placedLadders.Add(currentPos);
                }

                i++;
            }

            int randomGap = Random.Range(2, maxGap + 1);
            int nextY = currentY - randomGap;

            while (i < sortedPositions.Count && sortedPositions[i].y >
nextY)
            {
                i++;
            }
        }
    }
```

Funkcija `IsVerticalCorridor` je odgovorna za filtriranje pozicija hodnika koje su prethodno definirane u skripti `RoomFirstDungeonGen`. Funkcija služi za identifikaciju i verifikaciju vertikalnih hodnika na temelju njihovih pozicija. Ona prima dva parametra: `position` (trenutnu poziciju koju provjerava) i `corridorPositions` (sve pozicije hodnika). Osigurava da svaka pozicija koju provjerava ima 2 susjeda gore i dolje, dakle da je pozicija vertikalnog hodnika. Horizontalni hodnici imaju samo jednog susjeda gore i dolje.

```
private bool IsVerticalCorridor(Vector2Int position, HashSet<Vector2Int>
corridorPositions)
{
      Vector2Int above = position + Vector2Int.up;
```

```
        Vector2Int twoAbove = position + Vector2Int.up * 2;
        Vector2Int below = position + Vector2Int.down;
        Vector2Int twoBelow = position + Vector2Int.down * 2;
        return corridorPositions.Contains(above) &&
corridorPositions.Contains(twoAbove) && corridorPositions.Contains(below)
&& corridorPositions.Contains(twoBelow);
}
```

Funkcija `IsLadderNeighbourUpAndDown` je dodatna provjera prije svakog postavljanja platforme. Ona osigurava da sve platforme u horizontalnim hodnicima nisu jedna na drugoj već da postoji razmak između njih.

```
private bool IsLadderNeighbourUpAndDown(Vector2Int position)
{
     if (placedLadders.Contains(position))
     {
          return true;
     }
     else
     {
          Vector2Int above1 = position + Vector2Int.up;
          Vector2Int above2 = position + Vector2Int.up * 2;
          Vector2Int below1 = position + Vector2Int.down;
          Vector2Int below2 = position + Vector2Int.down * 2;

          if (placedLadders.Contains(above1) ||
placedLadders.Contains(above2) || placedLadders.Contains(below1) ||
placedLadders.Contains(below2))
          {
              return true;
          }
          return false;
     }
}
```

Tehničke upute u nastavku opisuju način tehničkog oblikovanja rada i navođenja literature. Funkcija `PlaceLaddersRooms` odgovorna je za postavljanje platforme unutar soba tamnice. Ona osigurava kretanje kroz visoke i velike sobe. Prima parametar `roomsList` koji sadrži listu svih prostorija tamnice. Prolazi kroz svaki red jedne sobe (y os), ali preskače prvi i zadnji red jer su oni zidovi prostorije (prema načinu na koji radi proceduralno generiranje soba).

`Shuffle` osigurava dodatnu razinu nasumičnosti i nepredvidljivosti, jer randomizira poredak unutar liste svih mogućih pozicija. Prije instanciranja objekta platforme provjerava se postoji li platforma ili neki drugi objekt blizu trenutne pozicije. Razmak je potreban zbog preglednosti igre.

```
public void PlaceLaddersRooms(List<BoundsInt> roomsList)
{
    foreach (var room in roomsList)
    {
        for (int y = room.yMin + 1; y <= room.yMax - 2; y += 2)
        {
            List<Vector2Int> validPositions = new List<Vector2Int>();

            for (int x = room.xMin + 2; x <= room.xMax - 2; x++)
            {
                Vector2Int currentPos = new Vector2Int(x, y);

                if (IsValidLadderPosition(currentPos, room))
                {
                    validPositions.Add(currentPos);
                }
            }

            ShuffleList(validPositions);

            int laddersPlaced = 0;
            foreach (var position in validPositions)
            {
                if (!IsLadderNeighbourSixDirections(position) &&
!IsLadderNeighbourUpAndDown(position))
                {
                    PlaceLadder(position, room);
                    laddersPlaced++;

                    if (laddersPlaced >= laddersPerRow)
                    {
                        break;
                    }
                }
            }
        }
```

```
    }
}
```

`PlaceLadder` je funkcija koja instancira platforme na odgovarajuću poziciju. Dodaje mali razmak (*offset*) kako bi pozicija platforme odgovarala načinu pozicioniranja pozadine. `IsValidLadderPosition` je provjera nalazi li se mjesto unutar rubova sobe.

```
private void PlaceLadder(Vector2Int startPos, BoundsInt room)
{
    for (int i = 0; i < ladderLength; i++)
    {
        Vector2Int ladderPos = new Vector2Int(startPos.x + i, startPos.y);
        Vector3 ladderPosition = new Vector3(ladderPos.x + 0.5f,
ladderPos.y + 0.5f, 0);
        if (IsValidLadderPosition(ladderPos, room))
        {
            Instantiate(ladderPrefab, ladderPosition, Quaternion.identity);
            placedLadders.Add(ladderPos);
        }
    }
}

private bool IsValidLadderPosition(Vector2Int position, BoundsInt room)
{
    return position.x > room.xMin + 2 && position.x < room.xMax - 2 &&
position.y > room.yMin + 2 && position.y < room.yMax - 2;
}
```

Funkcije `PlacePlayer` (igrač) i funkcija `PlaceDoor` (vrata) rade na isti princip. Postavljaju objekte na prvi slobodnu x os, tri jedinice iznad minimalne y osi sobe (`min.y`) Ova je pozicija odabrana s obzirom na način na koji su zidovi generirani. Inače bi se moglo dogoditi da su objekti preblizu zidovima ili čak u njima. Validacija pozicije je provedena pomoću `IsValidLadderPosition` i `IsLadderNeighbourSixDirections`. Igrač je pozicioniran u prvoj generiranoj sobi unutar tamnice, a vrata u posljednjoj.

```
public void PlacePlayer(List<BoundsInt> roomsList)
{
    var room = roomsList[0];

    int yPosition = room.yMin + 3;
```

```
    for (int x = room.xMin + 3; x <= room.xMax - 3; x++)
    {
        Vector2Int currentPos = new Vector2Int(x, yPosition);
        if (IsValidLadderPosition(currentPos, room))
        {
            if (!IsLadderNeighbourSixDirections(currentPos))
            {
                placedLadders.Add(currentPos);
                player.transform.position = new Vector3(currentPos.x,
currentPos.y, player.transform.position.z);
                return;
            }
        }
    }
}

public void PlaceDoor(List<BoundsInt> roomsList)
{
    var room = roomsList[roomsList.Count - 1];

    int yPosition = room.yMin + 3;

    for (int x = room.xMin + 3; x <= room.xMax - 3; x++)
    {
        Vector2Int doorPosition = new Vector2Int(x, yPosition);
        if (IsValidLadderPosition(doorPosition, room))
        {
            if (!IsLadderNeighbourSixDirections(doorPosition))
            {
                placedLadders.Add(doorPosition);
                door.transform.position = new Vector3(doorPosition.x +
0.5f, doorPosition.y + 0.5f, door.transform.position.z);
                return;
            }
        }
    }
}
```

`PicturePlacer` je zadužen za postavljanje slika u hodnicima proceduralno generirane tamnice. Dekorativni elementi pridonose raznolikosti razine te zato postoje tri

različite slike koje su nasumično odabrane pomoću `UnityEngine.Random`. Slike se stavljaju na sredinu hodnika i postavljene su u razmakom od deset do petnaest jedinica.

```
public void PicturePlacer(HashSet<Vector2Int> corridorPositions)
    {
        List<Vector2Int> sortedPositions = new List<Vector2Int>();
        List<GameObject> randPicture = new List<GameObject>();
        randPicture.Add(picturePrefab);
        randPicture.Add(picturePrefab2);
        randPicture.Add(picturePrefab3);
        foreach (var position in corridorPositions)
        {
            if (IsHorizontalLine(position, corridorPositions))
            {
                sortedPositions.Add(position);
            }
        }
        int i = 0;

        while (i < sortedPositions.Count)
        {
            Vector2Int currentPos = sortedPositions[i];
            if(!IsLadderNeighbourUpAndDown(currentPos))
            {
                int prefabIndex = UnityEngine.Random.Range(0,3);
                Vector3 picturePosition = new Vector3(currentPos.x + 0.5f,
currentPos.y + 0.5f, 0);
                Instantiate(randPicture[prefabIndex], picturePosition,
Quaternion.identity);
            }
            i += Random.Range(10, 15);
        }
    }
```

Stvaranje neprijatelja radi pomoću dvije komponente: funkcija koja postavlja mjesta stvaranja (eng. *spawner*) neprijatelja i skripte koja postavlja neprijatelje ne ta mjesta. `PlaceEnemySpawner` funkcija postavlja objekte stvaranja u sve sobe tamnice osim prve (tamo se igrač generira). Koristi istu logiku kao druge funkcija generiranja unutar ove skripte.

```
public void PlaceEnemySpawners(List<BoundsInt> roomsList)
{
      for (int i = 1; i < roomsList.Count; i++)
```

```
        {
            var room = roomsList[i];
            List<Vector2Int> validPositions = new List<Vector2Int>();

            for (int x = room.xMin + 2; x <= room.xMax - 2; x++)
            {
                for (int y = room.yMin + 2; y <= room.yMax - 2; y++)
                {
                    Vector2Int currentPos = new Vector2Int(x, y);

                    if (IsValidLadderPosition(currentPos, room))
                    {
                        validPositions.Add(currentPos);
                    }
                }
            }

            ShuffleList(validPositions);

            int numberOfSpawnPoints = UnityEngine.Random.Range(2, 4);

            int placedSpawnPoints = 0;

            foreach (var position in validPositions)
            {
                if (!IsLadderNeighbourSixDirections(position))
                {
                        PlaceSpawner(position, room);
                        placedSpawnPoints++;
                        if (placedSpawnPoints >= numberOfSpawnPoints)
                        {
                            break;
                        }
                  }
            }
        }
    }

    private void PlaceSpawner(Vector2Int position, BoundsInt room)
    {
```

```
        GameObject enemySpawner = Instantiate(enemySpawnerPrefab, new
Vector3(position.x, position.y, 0), Quaternion.identity);
}
```

Skripta `EnemySpawner` generira objekte neprijatelja na mjestima stvaranja. Prolazi kroz svaki objekt mjesta stvaranja i na njegovu poziciju stvara jednoga neprijatelja.

```
public class EnemySpawner : MonoBehaviour
{
    [SerializeField] private GameObject enemyPrefab;
    public void SpawnEnemiesAtSpawners()
    {
      GameObject[]spawners =
      GameObject.FindGameObjectsWithTag("EnemySpawner”);
        foreach (GameObject spawner in spawners)
        {
         Instantiate(enemyPrefab, spawner.transform.position,
Quaternion.identity);
        }
    }
}
```

3.3.1.3. Stvaranje pozadine i zidova

Tehničke upute u nastavku opisuju način tehničkog oblikovanja rada i navođenja literature. Generiranje pozadine odvija se u skripti `TilemapVisualizer`. Ona je zaslužna za postavljanje svih pločica definiranih pomoću koda `[SerializeField] private TileBase ruleTileTile`.

Svi zidovi kao i pozadina postavljaju se na mapu pločica (eng. *Tilemap*). `PaintBgTiles` javna je metoda koja omogućava postavljanje pločica na pozicijama proslijeđenim kao `backgroundPositions` iz skripte `RoomFirstDungeonGen`. Ona poziva privatnu metodu `PaintTiles`, koja zatim iterira kroz sve pozicije i poziva `PaintSingleTile` na svaku od njih. `PaintSingleTile` uzima tu poziciju, pretvara je u odgovarajući tilemap format, te postavlja pločicu na tu poziciju koristeći `SetTile` metodu mape pločica.

`PaintSingleBasicWall` je funkcija koja omogućava postavljanje pojedinačnog zida na zadanu poziciju koristeći istu logiku kao i `PaintSingleTile`. Ona je korištena u skripti `WallGenerator`. [23]

Clear je funkcija za čišćenje svih pločica s mape. Koristi se kad je potrebno resetirati ili regenerirati razinu.

`WallGenerator` je skripta koja se bavi generiranjem zidova na osnovu već postojećih pozadinskih pozicija (`backgroundPositions`). Ova klasa služi kao alat za određivanje gdje se zidovi trebaju postaviti. `CreateWalls` je glavna funkcija unutar klase I služi za postavljanje zidova. Ona koristi `FindWallsInDirections` za pronalaženje svih pozicija okolo pozadine, a zatim koristi `TilemapVisualizer` za postavljanje pločica na te pozicije. To omogućava da su sve pozadine okužene zidovima što stvara zatvoren prostor.

`FindWallsInDirections` pretražuje sve susjedne pozicije u odnosu na pozadinu, koristeći unaprijed definirani popis smjerova (`eightDirectionsList`). Dakle pretražuje susjede u svim smjerovima. Ako neka od tih susjednih pozicija nije pozadina, dodaje se u skup pozicija zidova (`wallPositions`). To omogućava postavljanje pločica samo na pozicije zidova te odvaja tu listu od liste pozicija soba za daljnju upotrebu. [29]

```
public class TilemapVisualizer : MonoBehaviour
{
    [SerializeField] private Tilemap tilemapFull;
    [SerializeField] private TileBase ruleTileTile;

    public void PaintBgTiles(IEnumerable<Vector2Int> backgroundPositions)
    {
        PaintTiles(backgroundPositions, tilemapFull, ruleTileTile);
    }
    private void PaintTiles(IEnumerable<Vector2Int> positions, Tilemap
tilemap, TileBase tile)
    {
        foreach (var position in positions)
        {
            PaintSingleTile(tilemap, tile, position);
        }
    }
    private void PaintSingleTile(Tilemap tilemap, TileBase tile, Vector2Int
position)
    {
        var tilePosition = tilemap.WorldToCell((Vector3Int)position);
        tilemap.SetTile(tilePosition, tile);
    }
    internal void PaintSingleBasicWall(Vector2Int position)
    {
        PaintSingleTile(tilemapFull, ruleTileTile, position);
    }
```

```
    public void Clear()
    {
        tilemapFull.ClearAllTiles();
    }
}


public static class WallGenerator
{
    public static void CreateWalls(HashSet<Vector2Int> floorPositions,
TilemapVisualizer tilemapVisualizer)
    {
        var basicWallPositions = FindWallsInDirections(floorPositions,
Direction2D.eightDirectionsList);
        foreach (var position in basicWallPositions)
        {
            tilemapVisualizer.PaintSingleBasicWall(position);
        }
    }
private static HashSet<Vector2Int>
FindWallsInDirections(HashSet<Vector2Int> floorPositions, List<Vector2Int>
directionList)
    {
        HashSet<Vector2Int> wallPositions = new HashSet<Vector2Int>();
        foreach (var position in floorPositions)
        {
            foreach (var direction in directionList)
            {
                var neighbourPosition = position + direction;
                if(floorPositions.Contains(neighbourPosition) == false)
                wallPositions.Add(neighbourPosition);
            }
        }
        return wallPositions;
    }
}
```

3.3.2.Osnovne mehanike

Osnovne mehanike igre pružaju igračima alate za interakciju i napredovanje kroz igru. U kontekstu ovoga rada, ove mehanike obuhvaćaju kretanje igrača, interakciju s okolinom te ponašanje neprijatelja i drugih objekata unutar igre. Razumijevanje i implementacija ovih mehanika ključna je za stvaranje igre koja je i zabavna i funkcionalna.

3.3.2.1. Kretanje igrača

Kretanje igrača obuhvaća različite aspekte kao što su trčanje, skakanje, borba i padanje kroz platformu. Ponašanje neprijatelja pak uključuje patroliranje, napadanje i praćenje igrača.

Skripta `PlayerMovement` odgovorna je za upravljanje kretanjem igrača, uključujući hodanje, skakanje, padanje, i animaciju povezanih akcija. Osim toga, ona također detektira interakciju s određenim objektima u igri, kao što su izlazi iz tamnice.

Funkcija `Update` poziva se jednom u svakom frameu, tj. svakom prikazu slike na ekranu. Dakle, ta funkcija se poziva vrlo često, ovisno o brzini renderiranja igre (*frame rate*). `FixedUpdate` se pak poziva u fiksnim vremenskim intervalima, neovisno o brzini renderiranja. Unity osigurava da se fizičke simulacije uvijek izvršavaju u jednakim vremenskim razmacima, bez obzira na promjene te tako osigurava konzistentne rezultate kod računanja fizike, što je posebno važno za kretanje objekata i sudare. [30]

`Update` detektira pritisak tipke igrača pomoću `Input.GetAxisRaw ("Horizontal")`. Također detektira pritisak na tipku za skakanje. Funkcija je odgovorna za mijenjanje animacija. Ako igrač skače ili pada (na osnovu brzine osi y) pokreće animaciju skoka. Ako igrač stoji na mjestu, gasi tu animaciju. Prije nego što igrač može skočiti mora prvo stajati na zemlji, kako ne bi došlo do beskonačnog skakanja.

```
void Update()
{
    horizontalMove = Input.GetAxisRaw("Horizontal") * playerSpeed;

    if (Input.GetButtonDown("Jump") && rigidbody2D.velocity.y < 0.01)
    {
        jump = true;
    }
    if (rigidbody2D.velocity.y < -0.01 || rigidbody2D.velocity.y > 0.01 )
    {
        animator.SetBool("jump", true);
        animator.SetBool("run", false);
```

```
    }
    else if (Mathf.Abs(rigidbody2D.velocity.y) < 0.1)
    {
        rigidbody2D.gravityScale = 1f;
        animator.SetBool("jump", false);
    }
}
```

Funkcija `FixedUpdate` zaslužna je za poziv funkcije `Move` koja pomiče igrača. Igrač se može kretati samo ako nije u stanju smrti, dakle ako trenutna animacija nije postavljena na "*Death*". Ako igrač pritišće tipke za kretanje (što je označeno varijablom `horizontalMove` različitom od nule), animacija trčanja (*run*) se pokreće. Ako igrač ne pritišće tipke za kretanje, kretanje igrača se glatko zaustavlja i animacija za trčanje se gasi. Samo se horizontalno kretanje igrača zaustavlja kako bi još uvijek mogao vertikalno padati. Varijabla `Jump` resetira se na *false* kako igrač ne bi beskonačno skakao.

```
void FixedUpdate()
{
    if (!animator.GetCurrentAnimatorStateInfo(0).IsName("Death"))
    {
        Move(horizontalMove * Time.fixedDeltaTime, jump);
    }
    if (horizontalMove != 0)
    {
        animator.SetBool("run", true);
    }
    else
    {
        animator.SetBool("run", false);
        rigidbody2D.velocity = new Vector2(0, rigidbody2D.velocity.y);
    }
    jump = false;
}
```

Funkcija `Move` i funkcija `Flip` zajedno osiguravaju da se igrač može kretati lijevo i desno, skakati i pravilno mijenjati smjer u kojem je okrenut. Rigidbody2D.velocity upravlja horizontalnim kretanjem igrača prema već postavljenoj `horizontalMove` varijabli. Pokreće igrača lijevo ili desno ovisno o tipki koju je pritisnuo. Ako je `jump` postavljen na *true* (što znači da je igrač pritisnuo tipku za skok), igraču se dodaje vertikalna sila.

Snaga skoka (`jumpStrength`) prilagođava se kroz funkciju ovisno o trenutnoj vertikalnoj brzini kako bi se skok osjećao prirodnije. Ako igrač skače (pozitivna `velocity.y`), gravitacija se postupno povećava kako bi se smanjila visina skoka, što daje osjećaj težine. Kada igrač počne padati (negativna `velocity.y`), gravitacija se odmah postavlja na višu vrijednost varijable `fastFall`, što ubrzava padanje.

Funkcija također osigurava da se igrač okrene u smjer u kojemu se kreće. Ako se igrač kreće desno (`move > 0`), a trenutno je okrenut lijevo (`!facingRight`), poziva se funkcija `Flip`. Isto tako, ako se igrač kreće lijevo (`move < 0`), a okrenut je desno (`facingRight`), također se poziva `Flip`.

Funkcija `Flip` je zaslužna za mijenjanje grafičkog elementa koji prikazuje igrača. Ona mijenja vrijednost `facingRight` na suprotnu i obrne x-komponentu `transform.localScale`, što vizualno okreće igrača na suprotnu stranu.

```
public void Move(float move, bool jump)
{
    rigidbody2D.velocity = new Vector2(move * 10f, rigidbody2D.velocity.y);
    if (jump)
    {
        float force = jumpStrength;
        rigidbody2D.AddForce(Vector2.up * force, ForceMode2D.Impulse);
    }

    if (rigidbody2D.velocity.y > 0)
    {
        rigidbody2D.gravityScale += gravityIncreaseRate * Time.deltaTime;
        rigidbody2D.gravityScale = Mathf.Clamp(rigidbody2D.gravityScale,
jumpSpeed, maxGravityScale);
    }
    if (rigidbody2D.velocity.y < 0)
    {
        rigidbody2D.gravityScale = fastFall;
    }
    if (move > 0 && !facingRight)
    {
        Flip();
    }
    else if (move < 0 && facingRight)
    {
        Flip();
```

```
    }
}

private void Flip()
{
    facingRight = !facingRight;
    Vector3 currentScale = transform.localScale;
    currentScale.x *= -1;
    transform.localScale = currentScale;
}
```

Funkcija `OnTriggerEnter2D` u Unityju automatski se poziva kada se objekt s `Collider2D` komponentom sudari s drugim objektom koji također ima `Collider2D` komponentu i koji je označen kao "`Trigger`". Funkcija u ovoj skripti radi tako što provjerava ima li objekt s kojim se igrač sudario "`tag`" "`DungeonExit`". Ovaj "tag" je postavljen na objekt koji označava izlaz iz tamnice.

Kako igrač ne bi nekako uspio doći do izlaza dok se izvodi duga animacija smrti igrača, mora se provjeriti koja je trenutna animacija. Igrač ne može pobijediti ako je mrtav. Igrač također ne može osvojiti igru ako je ozlijeđen. Ako su svi uvjeti ispunjeni, ova linija pokreće efekt konfeta, što je vizualni prikaz pobjede i pokreće ekran pobjede i onemogućuje objekt igrača.

```
private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("DungeonExit") &&
(!animator.GetCurrentAnimatorStateInfo(0).IsName("Death")) &&
(!animator.GetCurrentAnimatorStateInfo(0).IsName("Hurt")))
        {
            confettiEffect.Play();
            uiManager.ToggleWin();
            gameObject.SetActive(false);
        }
    }
```

3.3.2.2. Sistem borbe

Sistem borbe je važan dio igre jer određuje jedan od važnijih načina interakcije igrača s videoigrom. Način na koji igrač napada je sličan načinu na koji neprijatelj napada. Veliki dio razloga zašto je igra zanimljiva igraču je upravo u savladavanju mehanike borbe. Važno da neprijatelji i način borbe svojom implementacijom odgovaraju tematici igre. U ovom radu, koji se temelji na kreiranju srednjovjekovne tamnice, implementiran je kostur kao neprijatelj.

Skripta `PlayerAttack` upravlja logikom napada igrača, uključujući prikaz napada, aktiviranje napada kada igrač klikne mišem i primjenu štete neprijateljima unutar određenog radijusa.

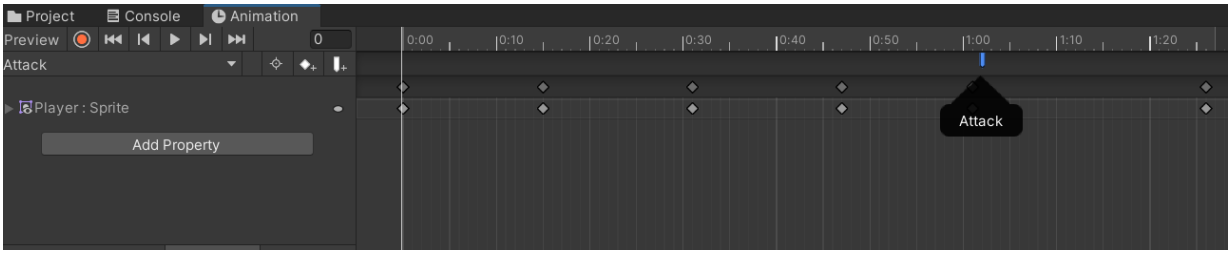
Funkcija `update` provjerava pritisak miša i aktivira animaciju napada.

```
void Update()
    {
        if(Input.GetMouseButtonDown(0))
        {
            animator.SetTrigger("attack");
        }
    }
```

Metoda `Attack` provodi taj napad. Koristeći dijete objekta igrača `AttackPoint`, pronalazi sve neprijatelje unutar radijusa napada (`attackRange`) od točke napada (`attackPoint.position`) Prolazi kroz sve pronađene neprijatelje i poziva `Damage` metodu na komponentu Health neprijatelja, time dodajući štetu. Ako neprijatelja nema, izlazi iz petlje. [31]

```
void Attack()
      {
        Collider2D[] hitEnemies =
Physics2D.OverlapCircleAll(attackPoint.position, attackRange, enemyLayers);
        foreach(Collider2D enemy in hitEnemies)
        {
            Debug.Log("Attacking for: " + damage);
            enemy.GetComponent<Health>().Damage(damage);
            if (enemy == null)
            break;
        }
      }
```

Funkcija `Attack` je pozvana kroz *Unity Animation Event* što osigurava da se smanji životni bodovi neprijatelja tek kada animacija napada stvarno udari neprijatelja.



Slika 15: Prikaz toka animacije napada igrača

3.3.2.3. Kretanje neprijatelja

Skripta `EnemyMovement` upravlja kretanjem neprijatelja kao i napadom neprijatelja. Ona omogućuje neprijatelju da patrolira po sceni, prati igrača kada je dovoljno blizu, te napada igrača kad se nađe u njegovom dometu. Također, koristi različite metode za detekciju prepreka i promjenu smjera kretanja. Koristi istu logiku napada kao kod igrača.

Funkcija `Update` osigurava da se neprijatelj može kretati samo ako nije ozlijeđen, ne umire i igrač postoji na sceni.

```
void Update()
{
if ((!animator.GetCurrentAnimatorStateInfo(0).IsName("SkeletonDeath")) &&
playerObject != null &&
(!animator.GetCurrentAnimatorStateInfo(0).IsName("SkeletonAttack2")) &&
(!animator.GetCurrentAnimatorStateInfo(0).IsName("SkeletonHurt")))
{
    Move();
}
else
{
      rigidbody2D.velocity = new Vector2(0, rigidbody2D.velocity.y);
}
}
```

Funkcija move provjerava udaljenost između neprijatelja i igrača. Ako je igrač unutar dometa playerDistance i dalje od `stopDistance`, neprijatelj počinje pratiti igrača `(ChasePlayer()`. Ako je igrač unutar `stopDistance`, neprijatelj se prestaje kretati i pokušava napasti. Ako je igrač izvan dometa, neprijatelj počinje patrolirati `(Patrol())`.

```
private void Move()
    {
        float distanceToPlayer = Vector2.Distance(transform.position,
player.position);


        if (distanceToPlayer < playerDistance &&
!animator.GetCurrentAnimatorStateInfo(0).IsName("SkeletonDeath"))
        {
            if (distanceToPlayer > stopDistance)
            {
                ChasePlayer();
            }
            else
            {
                rigidbody2D.velocity = Vector3.zero;
                if (Time.time >= nextAttackTime)
                {
                    animator.SetTrigger("Attack");
                    nextAttackTime = Time.time + 1f / attackRate;
                }
            }
        }
        else if (distanceToPlayer > playerDistance)
        {
            Patrol();
        }
    }
```

`ChasePlayer` i `Patrol` funkcije su zadužene za pomicanje neprijatelja. Ako je igrač dovoljno blizu, neprijatelj prati igrača prema njegovoj poziciji. Ako nije, neprijatelj patrolira lijevo-desno. Tu dolazi do problema jer ako se neprijatelj nalazi na platformi može pasti s nje. Zato je potreban objekt `groundCheck` koji provjerava nalazi li se ispred njega tlo. Kako bi neprijatelj prepoznao da je ispred njega zid i okrenuo se, napravljen je objekt `wallCheck`. Oba ta objekta djeca su neprijatelja. Postoje dvije provjere za tlo, jedna provjerava nalazi li se na platformi, druga provjerava pod. Takva implementacija je potrebna jer ako postoji samo provjera za platformu ona će uvijek davati *null* ako je neprijatelj na podu tamnice te bi se tada neprijatelj vječno okretao.

```
private void ChasePlayer()
    {
        transform.position = Vector2.MoveTowards(transform.position,
player.position, speed * Time.deltaTime);
        if ((player.position.x < transform.position.x && !facingLeft) ||
(player.position.x > transform.position.x && facingLeft))
        {
            Flip();
        }
    }
void Patrol()
    {
        Vector2 direction = facingLeft ? Vector2.left : Vector2.right;
        transform.position = Vector2.MoveTowards(transform.position,
(Vector2)transform.position + direction, speed * Time.deltaTime);
        RaycastHit2D groundInfo = Physics2D.Raycast(groundCheck.position,
Vector2.down, 2f, platform);
        if (groundInfo.collider == null)
        {
            groundInfo = Physics2D.Raycast(groundCheck.position,
Vector2.down, 1f, ground);
        }
        RaycastHit2D wallInfo = Physics2D.Raycast(wallCheck.position,
direction, 0.5f, ground);

        if (groundInfo.collider == null || wallInfo.collider != null)
        {
            Flip();
        }
}
```

3.3.2.4. Životni bodovi

Skripta Health ključna je komponenta svakog lika u video igri, bilo da se radi o igraču ili neprijatelju. Ova klasa upravlja količinom životnih bodova lika, načinom primanja štete, animacijama ozljede, i na kraju, upravljanjem smrti lika.

Klasa `Health` pruža osnovne funkcionalnosti poput smanjenja životnih bodova prilikom primanja štete, pokretanja odgovarajućih animacija prilikom gubitka zdravlja, te upravljanja ponašanjem objekta nakon smrti, uključujući uništavanje objekta nakon završetka animacije smrti. Također, povezana je s korisničkim sučeljem igre za prikazivanje vizualnih indikatora štete kada igrač primi udarac.

Funkcija `update` sadrži dodatnu provjeru trenutnih animacija lika i neprijatelja. Ako imaju nula životnih bodova, ne mogu se micati dok animacija smrti traje.

```
void Update()
    {
        if (animator.GetCurrentAnimatorStateInfo(0).IsName("SkeletonDeath")
|| animator.GetCurrentAnimatorStateInfo(0).IsName("Death"))
        {
            rigidbody2D.velocity = Vector3.zero;
        }
}
```

Funkcija `Damage` smanjuje životne bodove za iznos specificiran u parametru `amount`. Ako objekt nije već u animaciji smrti, dakle ako ima više od nula životnih bodova, pokreće se animacija "*Hurt*" kako bi se vizualno prikazalo da je objekt primio štetu. Ako životni podovi padnu na nulu ili ispod, pokreće se proces umiranja pozivanjem metode `Die()`. Ako je objekt označen kao "*Player*", metoda poziva `uiManager.DamageIndicator()` kako bi se ažurirao indikator životnih bodova na korisničkom sučelju.

```
public void Damage(int amount)
    {
        if (health >= 0)
        {
            this.health -= amount;
            if (!(animator.GetCurrentAnimatorStateInfo(0).IsName("Death"))
|| !(animator.GetCurrentAnimatorStateInfo(0).IsName("SkeletonDeath")))
            {
                animator.SetTrigger("Hurt");
            }
            if(health <= 0)
            {
                deathTime = Time.time;
                Die();
            }
            if (gameObject.CompareTag("Player"))
            {
                uiManager.DamageIndicator(amount);
            }
        }
    }
```

```
    private void Die()
    {
        if (!animator.GetCurrentAnimatorStateInfo(0).IsName("Death"))
        {
            animator.SetTrigger("Death");
        }
    Destroy(gameObject, animator.GetCurrentAnimatorStateInfo(0).length + deathAnim);
}
```

3.3.2.5. Platforme

Kontrola platformi ključna je za fluidno iskustvo kretanja kroz teren igre. Ovaj rad bavi se izradom 2D igre s horizontalnim pomicanjem te je zato izrazito bitno da igrač može pasti kroz platforme. Da ta funkcionalnost nije omogućena, igrač bi mogao skakati samo gore kroz razinu i nikada se ne bi mogao vratiti. Ta je funkcionalnost implementirana s jednosmjernim platformama kroz koje igrač može pasti ili skočiti.

Skripta `PlatformControls` omogućava igraču da privremeno onemogući sudar s platformama kada pritisne određenu tipku, poput S ili strelice prema dolje. To je korisno za situacije kada igrač želi pasti kroz platformu umjesto da se zadrži na njoj.

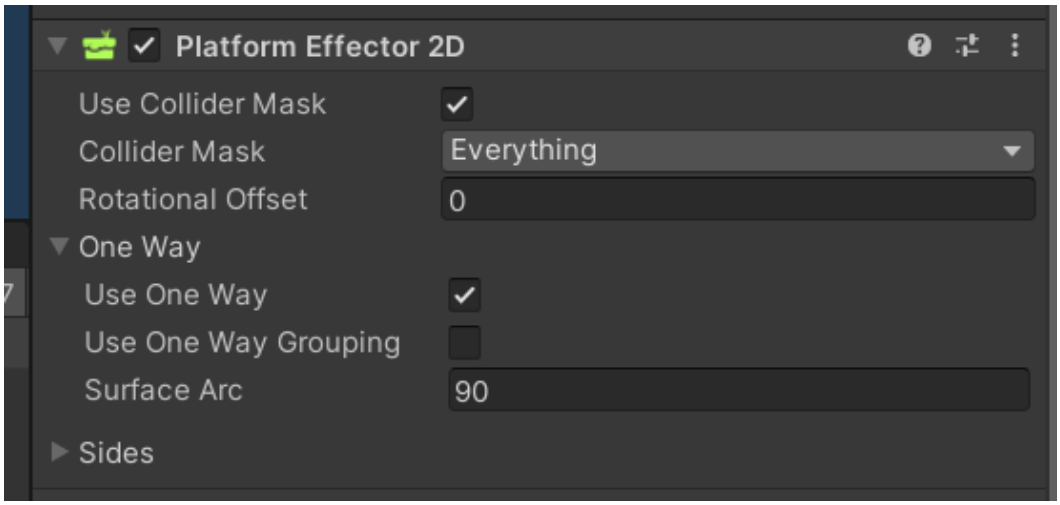
Korutina (eng. *Coroutine*) `DisableCollision` onemogućava sudar između igrača i platformi na kratko vrijeme, dopuštajući igraču da prođe kroz platformu. Nakon zadanog vremena, sudar se ponovno omogućuje, vraćajući platformu u njeno uobičajeno stanje.

```
private void Update()
    {
        if (Input.GetKeyDown(KeyCode.S) ||
Input.GetKeyDown(KeyCode.DownArrow))
        {
            StartCoroutine(DisableCollision());
        }
    }
    private IEnumerator DisableCollision()
    {
        GameObject[] platforms =
GameObject.FindGameObjectsWithTag("Platform");

        foreach (var platform in platforms)
        {
            BoxCollider2D platformCollider =
platform.GetComponent<BoxCollider2D>();
```

```
            if (platformCollider != null)
            {
                Physics2D.IgnoreCollision(playerCollider, platformCollider,
true);
            }
        }
        yield return new WaitForSeconds(0.35f);
        foreach (var platform in platforms)
        {
            BoxCollider2D platformCollider =
platform.GetComponent<BoxCollider2D>();
            if (platformCollider != null)
            {
                Physics2D.IgnoreCollision(playerCollider, platformCollider,
false);
            }
    }
 }
```

Kako bi igrač mogao skakati kroz platformu odozdo, a odozgo stati na nju, potrebno je na objekt platforme dodati *Platform Effector 2D*. Manji *Surface Arc* omogućuje da ako je razmak između dvije platforme samo jedan blok, igrač koji je visok dva bloka i dalje se može kretati. [32]



Slika 16: Prikaz inspektora objekta platforme

3.3.3. Upravljanje sučeljima

Korisnička sučelja (*UI*) u videoigrama važna su za pružanje povratnih informacija igraču i omogućavanje neke dodatne interakcije s igrom (izbornici, dijalozi…). Skripta `UIManager` upravlja prikazom različitih panela poput panela za smrt, pobjedu ili izbornika za pauzu. Ona također upravlja trakom koja predstavlja životne bodove. Ova skripta osigurava da se relevantni UI elementi prikazuju u pravim trenucima i na pravilan način reagiraju na događaje u igri, kao što su napadi na igrača, pobjeda u razini ili pauziranje igre.

`Update` provjerava je li pritisnuta tipka `Escape`. Ako je, onda aktivira panel `Options` koji ima opcije ponovnog pokretanja igre, gašenja igre ili povratka u glavni izbornik.

```
void Update()
{
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            ToggleEscape();
        }
}
```

Funkcije `ToggleDeathPanel`, `ToggleWin` i `ToggleEscape` zaslužne su za aktiviranje i deaktiviranje panela koji se prikazuju nakon smrti igrača, nakon pobjede i ako igrač pritisne tipku *escape*.[33]

```
public void ToggleDeathPanel()
{
      deathPanel.SetActive(!deathPanel.activeSelf);
}
```

Slika 17: Prikaz ekrana smrti

```
public void ToggleEscape()
{
	escapePanel.SetActive(!escapePanel.activeSelf);
}
```

Slika 18: Prikaz ekrana opcija

```
public void ToggleWin()
{
	winPanel.SetActive(!winPanel.activeSelf);
	healthBarMain.SetActive(!healthBarMain.activeSelf);
}
```

Slika 19: Prikaz ekrana pobjede

`DamageIndicator` je funkcija odgovorna za sinkroniziranje životnih bodova igrača s onima koji su prikazani na traci korisničkog sučelja. Oduzima određenu količinu bodova (*damage*) od ukupnog količine životnih bodova na traci. Ažurira traku (*healthBar*) i pokreće korutinu `DamageIndicatorRedBar()` za dodatni vizualni efekt. Takva implementacija osigurava da se zeleni dio trake odmah smanji za odgovarajuće bodove, a crveni dio trake polako spušta dok ne sustigne zeleni. Tekst je također ažuriran kako bi ispravno prikazao trenutne bodove.

```
public void DamageIndicator(int damage)
    {
        healthAmmount -= damage;
        healthBar.fillAmount = healthAmmount / 100f;
        StartCoroutine(DamageIndicatorRedBar());
        if (healthAmmount > 0)
        {
            healthAmmountText.text = $"{healthAmmount}/100";
        }
        else
        {
            healthAmmountText.text = $"0/100";
        }

    }

private IEnumerator DamageIndicatorRedBar()
    {
        float startFillAmount = redBar.fillAmount;
        float targetFillAmount = healthAmmount / 100f;
        float duration = 1f;
        float elapsed = 0f;

        while (elapsed < duration)
        {
            elapsed += Time.deltaTime;
            redBar.fillAmount = Mathf.Lerp(startFillAmount,
targetFillAmount, elapsed / duration);
            yield return null;
        }
        redBar.fillAmount = targetFillAmount;
    }
```

Klasa `GameManager` ima ključnu ulogu upravljanja događajima i stanjima tijekom igre. Ova skripta pokreće generiranje tamnice na svakom početku igre i nadgleda stanje igrača kako bi mogla uključiti panel *DeathPanel* nakon smrti igrača. Funkcija start pokreće se pri pokretanju igre i zadužena je za poziv skripta za generiranje tamnice. Funkcija `Update` čeka na smrt igrača kako bi mogla uključiti panelu korisničkog sučelja *DeathPanel*.

```
void Start()
    {
        roomFirstDungeonGenerator.GenerateDungeon();
        player = GameObject.FindGameObjectWithTag("Player");
    }
void Update()
    {
        if (player == null && !isDeathPanelOn)
        {
            uiManager.ToggleDeathPanel();
            isDeathPanelOn = true;
        }
    }
```

3.3.4.Upravljanje scenom

Kako bi mogli mijenjati scene u igri i kako bi igrač mogao ugasiti igru klikom jednog gumba, potrebno je implementirati skriptu koja se bavi upravo tim funkcionalnostima. Promjena scene je važna jer se glavni izbornik koji korisnik vidi, ne nalazi na istoj sceni kao igra. Klasa `SceneManagements` upravlja promjenama scena u igri. Omogućava ponovno učitavanje trenutne scene, prelazak na drugu scenu prema imenu te zatvaranje igre. Ova klasa zadužena je za osnovne funkcionalnosti navigacije unutar igre, poput ponovnog pokretanja razine nakon smrti igrača, prelaska na glavni meni ili zatvaranja igre kada igrač želi izaći.

```
public class SceneManagements : MonoBehaviour
{
    public void ReloadCurrentScene()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
    }
    public void ChangeSceneByName(string name)
    {
        if (name!= null)
        {
```

```
                SceneManager.LoadScene(name);
            }
        }
        public void QuitGame()
        {
            Application.Quit();
        }
}
```

4. Zaključak

U ovom završnom radu obrađena je tema proceduralnog generiranja razina u video igrama, koristeći Unity kao programski alat. Proces izrade videoigara je složen i kompliciran te uključuje razne tehnike i alate. Osim samog programiranja izrazito je važan i grafički izgled igre kao i njezina igrivost. U komponenti igrivosti najvažniju ulogu ima proceduralno generiranje koje omogućuje stvaranje dinamičkih i raznolikih razina u video igri. Automatskim stvaranjem različitog sadržaja postiže se jedinstveno iskustvo igranja pri svakom novom pokretanju.Ovaj je rad omogućio korištenje raznolikih vještina stečenih na fakultetu. Osim programiranja bilo je potrebno crtanje, osmišljavanje mehanika, dizajniranje te korištenje alata Unity.

Proceduralno generiranje provedeno je korištenjem binarne prostorne podjele (eng. *Binary Space Partitioning - BSP).* Ta metoda radi na principu podjele. Dijeli prostor na manje dijelove i ovisno o načinu implementacije omogućuje da se ta podjela izvrši drugačije svaki put. To donosi beskonačne varijacije u veličinama i podjeli soba unutar razine. Proceduralno generiranje pokazalo je svoju učinkovitost u stvaranju dinamičnih i različitih razina, a binarna prostorna podjela je pokazala svoju korisnost kao alat za podjelu prostora i generiranje soba. Ovakvi alati omogućuju kreativna rješenja s minimalnim ulaganjem resursa.

Kreirana videoigra kao rezultat ovog rada potvrđuje učinkovitost proceduralnog generiranja za poboljšanje igrivosti i stvaranje jedinstvenih iskustava u video igrama, istovremeno pružajući uvid u složenost i izazove ovakvog pristupa.

Popis literature

[1] Unity Real-Time Development Platform (Unity Hub 3.8.0.) (2024) [Na internetu]. Dostupno: https://unity.com/download [pristupano 20. srpnja, 2024.]

[2] Visual Studio Code (verzija 1.92.2) (2024) [Na internetu]. Dostupno: https://code.visualstudio.com/ [pristupano 20. srpnja, 2024.]

[3] Unity Discussions (bez dat.) [Na internetu]. Dostupno: https://discussions.unity.com/ [pristupano 20.07.2024.]

[4] stackoverflow (bez dat.) [Na internetu]. Dostupno: https://stackoverflow.com/ [pristupano 20.07.2024.]

[5] Raou, „Side-Scroller Dungeon tileset release!“, 2019. [Na internetu]. Dostupno: https://raou.itch.io/dark-dun/devlog/90492/side-scroller-dungeon-tileset-release [pristupano 20.07.2024.]

[6] Sharkshock, „Enchanted Land“, 2016. [Na internetu]. Dostupno: https://www.dafont.com/enchanted-land.font [pristupano 20.07.2024.]

[7] Canva Free AI Image Generator (bez dat.) [Na internetu]. Dostupno: https://www.canva.com/ai-image-generator/ [pristupano 20.07.2024.]

[8] „Free Knight Character Sprites Pixel Art” (10. 10. 2022.). Craftpix [Na internetu]. Dostupno: https://craftpix.net/freebies/free-knight-character-sprites-pixel-art/ [pristupano 20.07.2024.]

[9] Slynyrd, „Steam Lords Palette“, (bez dat.) [Na internetu]. Dostupno: https://lospec.com/palette-list/steam-lords [pristupano 20.07.2024.]

[10] Toby_Yasha, „TY – Disaster Girl 16 Palette“, (bez dat.) [Na internetu]. Dostupno: https://lospec.com/palette-list/ty-disaster-girl-16 [pristupano 20.07.2024.]

[11] Ansimuz, „GothicVania Cemetery”, 2018. [Na internetu]. Dostupno: https://assetstore.unity.com/packages/2d/characters/gothicvania-cemetery-120509 [pristupano 20.07.2024.]

[12] Aseprite (verzija 1.3) (2024) [Na internetu]. Dostupno: https://www.aseprite.org/ [pristupano 20. srpnja, 2024.]

[13] „Procedural generation,“ (bez dat.). u *Wikipedia, the Free Encyclopedia.* Dostupno: https://en.wikipedia.org/wiki/Procedural_generation [pristupano 20.07.2024.]

[14] *Rouge* [Slika] (bez.dat.) Dostupno: https://upload.wikimedia.org/wikipedia/commons/0/0c/Rogue_Screenshot.png [pristupano 20.07.2024.]

[15] „Procedural animation," (bez dat.) u *Wikipedia, the Free Encyclopedia.* Dostupno: https://en.wikipedia.org/wiki/Procedural_generation [pristupano 20.07.2024.]

[16] Merxon22, „Recreating RainWorld's 2D Procedural Animation— Part 1", 2023. FNa internetu]. Dostupno: https://medium.com/@merxon22/recreating-rainworlds-2d-procedural-animation-part-1-4d882f947e9f [pristupano 20.07.2024.]

[17] Valve Software (bez dat.) *Binary space partitioning* [Na internetu]. Dostupno: https://developer.valvesoftware.com/wiki/Binary_space_partitioning [pristupano 20.07.2024.]

[18] „Binary space partitioning," (bez dat.) u *Wikipedia, the Free Encyclopedia.* Dostupno: https://en.wikipedia.org/wiki/Binary_space_partitioning [pristupano 20.07.2024.]

[19] Sunny Valley Studio, (18.12.2020.) „Theory - P2 - Unity Procedural Generation of a 2D Dungeon", *Youtube* [Video datoteka]. Dostupno: https://www.youtube.com/watch?v=jlNqp_QAibo [pristupano 20.07.2024.]

[20] „Dead Cells," (bez dat.) u *Wikipedia, the Free Encyclopedia.* Dostupno: https://en.wikipedia.org/wiki/Dead_Cells [pristupano 20.07.2024.]

[21] „Hollow Knight," (bez dat.) u *Wikipedia, the Free Encyclopedia.* Dostupno: https://en.wikipedia.org/wiki/Hollow_Knight [pristupano 20.07.2024.]

[22] *Four years on, Dead Cells remains a roguelike delight* [Slika] (bez dat.) Dostupno: https://www.rockpapershotgun.com/four-years-on-dead-cells-remains-a-roguelike-delight [pristupano 20.07.2024.]

[23] Sunny Valley Studio, (18.12.2020.) „Adding Floor Tiles - P6 - Unity Procedural Generation of a 2D Dungeon", *Youtube* [Video datoteka]. Dostupno: https://youtu.be/W6cBwk0bRWE?si=-NUGSUVRwxyhLZPZ [pristupano 20.07.2024.]

[24] Unity (bez dat.) *Rule Tile* [Na internetu]. Dostupno: https://docs.unity3d.com/Packages/com.unity.2d.tilemap.extras@1.6/manual/RuleTile.html [pristupano 20.07.2024.]

[25] Sunny Valley Studio, (18.12.2020.) „Binary Space Partitioning Algorithm - P13 - Unity Procedural Generation of a 2D Dungeon", *Youtube* [Video datoteka]. Dostupno: https://youtu.be/nbi88hY9hcw?si=x4HkMs15M6GMcZMf [pristupano 20.07.2024.]

[26] Sunny Valley Studio, (18.12.2020.) „Splitting Rooms - P14 - Unity Procedural Generation of a 2D Dungeon“, *Youtube* [Video datoteka]. Dostupno: https://youtu.be/S0MNBfc0H_I?si=KmS5HPSv1hNbT0KB [pristupano 20.07.2024.]

[27] Sunny Valley Studio, (18.12.2020.) „Room First Generation - P15 - Unity Procedural Generation of a 2D Dungeon“, *Youtube* [Video datoteka]. Dostupno: https://youtu.be/pWZg1oChtnc?si=2g8lxGEhkkVKpnCd [pristupano 20.07.2024.]

[28] Sunny Valley Studio, (19.12.2020.) „Connecting rooms using corridors - P16 - Unity Procedural Generation of a 2D Dungeon“, *Youtube* [Video datoteka]. Dostupno: https://youtu.be/VpBrU4-PIUw?si=WNaakvibOq-Al4Wq [pristupano 20.07.2024.]

[29] Sunny Valley Studio, (19.12.2020.) „Placing Basic Walls - P9 - Unity Procedural Generation of a 2D Dungeon“, *Youtube* [Video datoteka]. Dostupno: https://youtu.be/LdpItlRg8OM?si=0VU6dsqmPEE-1xWh [pristupano 20.07.2024.]

[30] Unity (bez dat.) *MonoBehaviour.FixedUpdate()* [Na internetu]. Dostupno: https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html [pristupano 20.07.2024.]

[31] Brackeys, (15.12.2019.) „MELEE COMBAT in Unity“, *Youtube* [Video datoteka]. Dostupno: https://www.youtube.com/watch?v=sPiVz1k-fEs [pristupano 20.07.2024.]

[32] Unity (bez dat.) *Platform Effector 2D* [Na internetu]. Dostupno: https://docs.unity3d.com/Manual/class-PlatformEffector2D.html [pristupano 20.07.2024.]

[33] Muddy Wolf, (24.03.2021.) „Death Screen & State Manager | 2D Top Down RPG in Unity #9 | 2D Game Dev Tutorial“, *Youtube* [Video datoteka]. Dostupno: https://www.youtube.com/watch?v=HwI32elDCn0 [pristupano 20.07.2024.]

Popis slika