Izrada 2.5D arkadne igre za više igrača u Unityu

Jakšić, Matej

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Sveučilišni preddiplomski studij računarstva

IZRADA 2.5D ARKADNE IGRE ZA VIŠE IGRAČA U UNITYU

Završni rad

Matej Jakšić

Osijek, 2018.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac Z1P - Obrazac za ocjenu završnog rada na preddiplomskom sveučilišnom studiju

Osijek, 14.09.2018.

Odboru za završne i diplomske ispite

Prijedlog ocjene završnog rada

Ime i prezime studenta:	Matej Jakšić
Studij, smjer:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	R3645, 25.09.2017.
OIB studenta:	83450410856
Mentor:	Doc.dr.sc. Časlav Livada
Sumentor:	
Sumentor iz tvrtke:	
Naslov završnog rada:	Izrada 2.5D arkadne igre za više igrača u Unityu
Znanstvena grana rada:	**Obradba informacija (zn. polje računarstvo)**
Predložena ocjena završnog rada:	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 2 razina
Datum prijedloga ocjene mentora:	14.09.2018.
Datum potvrde ocjene Odbora:	26.09.2018.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 26.09.2018.

Ime i prezime studenta:	Matej Jakšić
Studij:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	R3645, 25.09.2017.
Ephorus podudaranje [%]:	6

Ovom izjavom izjavljujem da je rad pod nazivom**: Izrada 2.5D arkadne igre za više igrača u Unityu**

izrađen pod vodstvom mentora Doc.dr.sc. Časlav Livada

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1. UVOD

Tema ovog završnog rada je 2.5D arkadna igra. 2.5 dimenzionalne igre, također zvane pseudo-3D, jest pojam korišten za opis 2D grafičkih projekcija i sličnih tehnika koje daju privid 3D-a dok su u stvarnosti 2D. Ovakav tip igre je ograničen na dvije ravnine dok se igre bez ovakvog ograničenja nazivaju 3D. Arkadna igra je jedna od podkategorija računalnih igara kojima je glavna svrha da se dođe do određene točke na mapi ili da se unište protivnici, a glavni alat za postizanje ovog cilja je kontrola vlastitog lika. Korišteni alati za izradu završnog rada su Unity Engine i Visual Studio. Cjelokupan programski kod je pisan u objektno orijentiranom programskom jeziku C#, kreiran od strane tvrtke Microsoft. Sva grafika je preuzeta s interneta [1], a igrivi likovi su kupljeni sa stranice Graphic River [2]. Sama zvukovna podloga [3] u igri je također preuzeta s interneta.

1.1. ZADATAK ZAVRŠNOG RADA

Potrebno je izraditi 2.5D arkadnu računalnu igru za više igrača koja ima sve elemente popularne igre Bomberman. Kako bismo uspjeli u ovome potrebno je biti upoznat s C# programskim jezikom i Unity Engine-om.

2. UNITY

Unity je višeplatformski *game engine* razvijen od strane tvrtke Unity Technologies [4] 2005. godine. Njegova primarna svrha je razvoj 2D i 3D igara za neku od 27 podržanih platformi, ali također se koristi i za izradu različitih simulacija za računala. Prva podržana platforma bila je Apple-ov OS X, ali su postupno dodavali podršku i za ostale platforme. Glavni programski jezik koji Unity podržava je C#. U verziji Unity 5 ukinuta je podrška za programski jezik Boo, a od verzije 2017.1 se polako ukida podrška za JavaScript. Pored osnovnih funkcija koje engine mora sadržavati, u Unity su integrirani i servisi koji omogućuju lakšu integraciju nekih mogućnosti poput oglasa, podrške za više igrača, kolaboraciju s ostalim programerima i analizu perfomansi samog projekta. Također, u Unity je integriran i Unity Asset Store koji omogućuje kupnju nekih gotovih rješenja poput teksturi, 2D slika (sprite-ova), modela i slično. Unity Asset Store [5] je zamišljen da štedi vrijeme developerima jer ne moraju utrošiti nekoliko desetaka ili stotina sati na izradu neke funkcionalnosti. Kada pokrenemo Unity, prikaže nam se prozor koji nam omogućuje izradu novog projekta ili otvaranje već postojećeg. Ukoliko odaberemo izradu novog projekta, prikazuje nam se novi prozor u kojemu je potrebno imenovati projekt i odabrati 2D/3D, kao i postojeće pakete koje želimo koristiti u našem projektu. Naravno, sve ovo moguće je podesiti u bilo kojem trenutku. Sav sadržaj igre koji nije sastavni dio Unity-a se obično stavlja u mapu *Assets.* Radi lakšeg snalaženja tijekom izrade igre, uobičajena je praksa izrada nove mape za svaki tip Asseta (zvukovi, animacije, 3D modeli, kontroleri, itd.). Nakon prilagođavanja svih postavki, otvaraju se sučelje Unity-a i scena. Scena je prostor na koji postavljamo sve elemente naše igre. Elementi igre se nazivaju objektima, te se dodavanjem skripti, zvukova, 3D modela ili slika (sprite-ova) ako se radi 2D igra i ostalih komponenti utječe na funkcionalnost samog objekta. Svakom objektu je moguće mijenjati veličinu, poziciju u sceni i rotaciju. Jedna scena je najčešće jedan nivo igre, a jedna igra može imati jednu ili više scena. Odabirom nekog od objekta u sceni prikazuje nam se prozor koji se naziva Inspektor. On nam prikazuje sve informacije o objektu, popis svih komponenti koje objekt sadrži i omogućava nam podešavanje istih. Osim scene, postoje i kartice *Animator, Project, Animation, Profiler, itd.* Kartice *Animator* i *Animation* služe za kreiranje i podešavanje animacija, kartica *Project* sadrži mapu *Assets* i omogućuje nam bržu navigaciju svim sadržajem. Kartica *Profiler* nam omogućuje detaljan uvid u svaki od aspekata igre i daje nam informacije o korištenju memorije, procesora, grafike, broju poligona, itd. Kako bi se nekim objektom u sceni upravljalo, potrebna je skripta. Svaka skripta, nakon svog stvaranja, sadrži automatski kreiranu javnu klasu s nazivom skripte, te ona nasljeđuje baznu klasu MonoBehaviour. Ova bazna klasa sadrži

unaprijed definirane funkcije, metode, poruke, operatore i članove koje možemo koristiti po potrebi. Unity prilikom kreiranja skripte automatski kreira funkcije *Start()* i *Update()*. *Start()* funkcija poziva se samo jednom i to nakon što se objekt inicijalizira, a skripta aktivira. *Update()* funkcija poziva se jednom za svaku sliku (*frame).* Osim ove dvije funkcije, također često se koristi funkcija *Awake()*. Ona se poziva nakon što se objekt inicijalizira, ali neovisno o tome da li je skripta aktivirana ili nije. Ovo nam je jako korisno kada objekt A traži da je objekt B već inicijaliziran. Pošto osvježavanje funkcije *Update()* često varira ovisno o broju zahtjeva koje treba prikazati u jednoj slici, za svu fiziku unutar igre se preporučuje korištenje funkcije *FixedUpdate()*. Često se u programiranju koriste i korutine. Korutine imaju mogućnost zaustavljanja izvođenja trenutnog koda i izvršavanje potprograma. Za pisanje svih skripti korišten je programski jezik C# i Visual Studio IDE [6]. Osim potrebnih alata, glavni izvor znanja je bila Unity dokumentacija [7].

3. REALIZACIJA ZADATKA U UNITY-U

3.1. OPIS IGRE

Potrebno je izraditi 2.5D igru s pogledom iz ptičje perspektive za više igrača. Glavni izvor ideje su popularne igre Playing with fire i Saturn Bomberman, a i želja da se iste prenesu u 2.5D okruženje te da im se dodaju određene dodatne funkcionalnosti. U igri postoje 2 igrača od kojih svaki ima svoj set kontrola kojima se kreće lijevo-desno, gore-dolje te tipka za bacanje bombi. Igrači se nalaze svaki na svojoj strani nivoa te su okruženi blokovima koji se mogu uništiti bombama koje kada se unište mogu, ali i ne moraju stvoriti neku od implementiranih vrsti pojačanja za igrača. Svaki od nivoa se razlikuje načinom igrivosti. Prvi nivo služi za privikavanje na kontrole, postoje samo dva igrača, bez uništivih blokova. Svaki igrač ima četiri bombe, brzina kretanja im je nešto veća u odnosu na ostale nivoe, a svaka bomba uništi 2 bloka u svakom smjeru. Drugi nivo sadrži blokove koji pružaju pojačanja za brzinu, broj bombi i jačinu bombe. Cilj je prikupiti što više pojačanja i time nadjačati protivnika, te ga naposljetku eliminirati. Treći nivo također sadrži blokove koji pružaju pojačanja, ali sada se u nivou nalaze i dva portala koji omogućuju teleportaciju lika u oba smjera. Svaki nivo ima prikaz života svakog igrača i brojač vremena do isteka nivoa. U slučaju eliminacije jednog igrača, pojavljuje se poruka na ekranu koja ispisuje pobjednika, a u slučaju eliminacije oba igrača ili isteka

vremena, prikazuje se poruka koja simbolizira neriješeno. U svakom trenutku je moguće pauzirati igru i vratiti se na početni zaslon kako bi se odabrao neki drugi nivo.

3.2. IGRIVI LIKOVI

Sl. 3.1. Slike igrivih likova

Kao što je ranije rečeno u potpoglavlju 3.1. Opis igre, postoje dva igriva igrača. Igrači se mogu kretati lijevo-desno, gore-dolje te imaju sposobnost bacanja bombe. Svaka kretnja je animirana te također postoje i tzv. mirujuće animacije koje se izvode kada nijedna od tipki za kretanje nije pritisnuta, ovisno o smjeru u kojemu se igrač prethodno kretao. U slučaju kada broj života jednog od igrača dođe od nule, aktivira se prigodna animacija, također ovisna o prethodnom smjeru kretanja. Svakom igraču je dodana vlastita skripta koja omogućuje kretnje i koja je naslijeđena od strane skripte Character koja upravlja animacijama igrača. Na svakom od igrača se nalazi i skripta BombSpawner koja omogućuje bacanje bombi. U nastavku slijedi programski kod i objašnjenje pojedine skripte.

```
using UnityEngine;

public class Player1 : Character
{
    private int tmpHealthHolder;
    protected override void Start()
    {
        base.Start();
        tmpHealthHolder = playerHealth;
```

```
        FindObjectOfType<GameManager>().GetP1Health(tmpHealthHolder);
    }

    protected override void Update()
    {
        GetInput();
        base.Update();
        if(tmpHealthHolder != playerHealth)
        {
            tmpHealthHolder = playerHealth;
            FindObjectOfType<GameManager>().GetP1Health(tmpHealthHolder);
        }
    }

    private void GetInput()
    {
        direction = Vector2.zero;
        if (Input.GetKey(KeyCode.W))
        {
            direction += Vector2.up;
        }
        else if (Input.GetKey(KeyCode.S))
        {
            direction += Vector2Int.down;
        }
        else if (Input.GetKey(KeyCode.A))
        {
            direction += Vector2.left;
        }
        else if (Input.GetKey(KeyCode.D))
        {
            direction += Vector2.right;
        }

    }

}
```

Programski kod 3.1. Skripta Player1

Skripta Player1 nasljeđuje skriptu Character. Ova skripta sadržava samo vektor smjera u kojemu se giba igrač, potrebne kontrole za kretanje lika i na samom početku nivoa prosljeđuje broj života skripti GameManager. Osim skripte Player1 postoji i skripta Player2. Jedina razlika u odnosu na prvu skriptu je ta što sadrži drugačije kontrole i prosljeđuje vrijednost drugoj metodi unutar skripte GameManager. U nastavku slijedi skripta Character koju obje skripte za kontrole igrača nasljeđuju, te njezino objašnjenje.

```
using System.Collections;
using UnityEngine;

[RequireComponent(typeof(Rigidbody2D))]
[RequireComponent(typeof(Animator))]
public abstract class Character : MonoBehaviour
{
    [SerializeField]
    private float playerSpeed;
    public int playerHealth;
    private Rigidbody2D charRigidbody;
    protected Vector2 direction;
    protected Animator charAnimator;
    private float waitTime = 2f;
    private bool invincible = false;
    private float timer;

    public bool IsMoving{
        get{return direction.x != 0 || direction.y != 0;}
    }

    protected virtual void Start ()
    {
        charRigidbody = GetComponent<Rigidbody2D>();
        charAnimator = GetComponent<Animator>();
       }
       protected virtual void Update ()
    {
        MovementHandler();
        timer += Time.deltaTime;
    }

    private void FixedUpdate()
    {
        if(playerHealth >= 1) { Move(); }

        else if(playerHealth <= 0)
        {
            charAnimator.SetLayerWeight(2, 1);
        }
    }
    public void Move()
    {
        charRigidbody.velocity = direction.normalized * playerSpeed;
    }
    public void MovementHandler()
    {
        if (IsMoving)
        {
            charAnimator.SetLayerWeight(1, 1);
            charAnimator.SetFloat("MovingX", direction.x);
            charAnimator.SetFloat("MovingY", direction.y);
        }
        else if (!IsMoving)
        {
            charAnimator.SetLayerWeight(1, 0);
        }
    }
```

```
    private void OnCollisionEnter2D(Collision2D objToCollide)
    {
        if(objToCollide.gameObject.tag == "movementSpeed") { playerSpeed += 0.5f; }
    }

    private void OnTriggerEnter2D(Collider2D objToCollide)
    {
        if (objToCollide.gameObject.tag == "Explosion" && invincible == false)
        {
            playerHealth--;
            StartCoroutine(Invincibility(waitTime));
        }
    }

    private IEnumerator Invincibility(float waitTime)
    {
        invincible = true;
        if (playerHealth > 0) { GetComponent<Renderer>().material.color =
Color.Lerp(Color.white, Color.grey, timer / waitTime); }
        yield return new WaitForSeconds(waitTime);
        invincible = false;
        GetComponent<Renderer>().material.color = Color.white;
    }
}
```

Programski kod 3.2. Skripta Character

Skripta Character se brine o stanju lika i potrebnim animacijama. Varijable *playerSpeed* i *playerHealth* sadrže vrijednosti o brzini i broju života igrača. Pošto želimo da varijabli *playerSpeed* vrijednost može podesiti unutar samog inspektora u Unity-u i to samo na objektu na kojemu se skripta nalazi, potrebno joj je dati atribut *SerializeField* jer Unity u inspektoru inače prikazuje samo javne varijable kao što je *playerHealth*. Nadalje varijable *charRigidbody, charAnimator* i *direction*, služe za kontrolu fizike i animacija pomoću programskog koda kao i vektor smjera u kojemu se igrač giba. Metoda *IsMoving* je tipa bool i pomoću nje doznajemo kreće li se igrač ili ne. Uobičajena praksa je da se grafika poziva u funkciji *Update()*, a fizika u *FixedUpdate()* iz razloga što se *Update()* funkcija poziva svaki frame, a *FixedUpdate()* nakon određenog vremena te je ona savršena za predikciju kretnji iz razloga što je ova metoda predvidljivija te se njome želimo osigurati da će se ova metoda svaki put isto ponašati svaki put kada pokrenemo igru na bilo kojem uređaju. Metoda *Move* se se brine o kretnji lika, a *MovementHandler* se brine o animacijama. U animatoru smo postavili vrijednosti *MovingX* i *MovingY* pomoću kojih kontroliramo koja će se animacija reproducirati ovisno o smjeru kretnje. Također u animatoru imamo 3 sloja, jedan koji se brine o animacijama dok igrač miruje, drugi koji se brine o animacijama dok se igrač kreće i treći koji se aktivira samo u slučaju kada broj života igrača

padne na nulu. *SetLayerWeight* je metoda koja se koristi za postavljanje težine sloja. Sloj *Idle* ima identifikator 0, sloj *Run* identifikator 1, a sloj *Death* identifikator 3. Tako se dok se igrač kreće, kako bi se prikazale točne animacije, sloju *Run* jačina sloja postavlja na 1, a dok igrač miruje ima vrijednost 0. *U* poruci *OnCollisionEnter2D* provjeravamo da li se igrač sudario s pojačanjem za povećanje brzine, te u slučaju sudara povećamo brzinu igraču za 0.5. Poruka *OnTriggerEnter2D* provjerava da li se dogodio okidač kada eksplozija dotakne igrača, te ako je varijabla *invincibility* postavljena na *false*, događa se smanjenje broja života igrača za jedan, te se poziva korutina *Invincibility* koja prima varijablu *waitTime* koja sadrži vrijeme trajanja izvršenja potprograma prije vraćanja na glavni program. Vrijeme je postavljeno na dvije sekunde tijekom kojega je igrač imun na eksplozije i broj života mu se ne može smanjiti. Tijekom izvršenja korutine se mijenja boja igrača u sivo kako bi se dalo do znanja da je igrač imun.

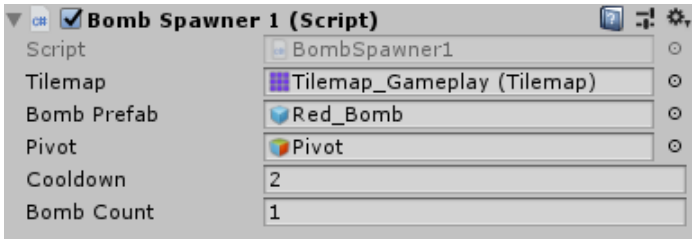
Iduća skripta koju svaki od igrača ima je skripta Bombspawner. Ova skripta se brine o mjestu stavljanja bombi i broju bombi koju svaki od igrača ima. Postoje dvije skripte BombSpawner1 i BombSpawner2, svaka je postavljena na određenog igrača. Jedina razlika među njima je u tipki za postavljanje bombi.

```
using UnityEngine;
using UnityEngine.Tilemaps;
using System.Collections;

public class BombSpawner1 : MonoBehaviour
{

    public Tilemap tilemap;
    public GameObject bombPrefab;
    public GameObject pivot;

    [SerializeField]
    private float cooldown;

    [SerializeField]
    private int bombCount;
    private int bombRemaining;
    private bool canBomb = true;

    private void Start()
    {
        bombRemaining = bombCount;
    }
```

```
    void Update()
    {
        if (Input.GetKeyDown("space") && canBomb == true)
        {
            Vector3 worldPos = pivot.transform.position;
            Vector3Int cell = tilemap.WorldToCell(worldPos);
            Vector3 cellCenterPos = tilemap.GetCellCenterWorld(cell);

            Instantiate(bombPrefab, cellCenterPos, Quaternion.identity);
            StartCoroutine(SetCooldown());
        }
    }

    IEnumerator SetCooldown()
    {
        if (bombRemaining >= 1) { bombRemaining--; }

        if (bombRemaining == 0)
        {
            canBomb = false;
            yield return new WaitForSeconds(cooldown);
            canBomb = true;
            bombRemaining = bombCount;
        }
    }

    private void OnCollisionEnter2D(Collision2D powerUp)
    {
        if (powerUp.gameObject.tag == "bombCount") { bombCount++; Start(); }
    }
}
```

Programski kod 3.3. Skripta BombSpawner

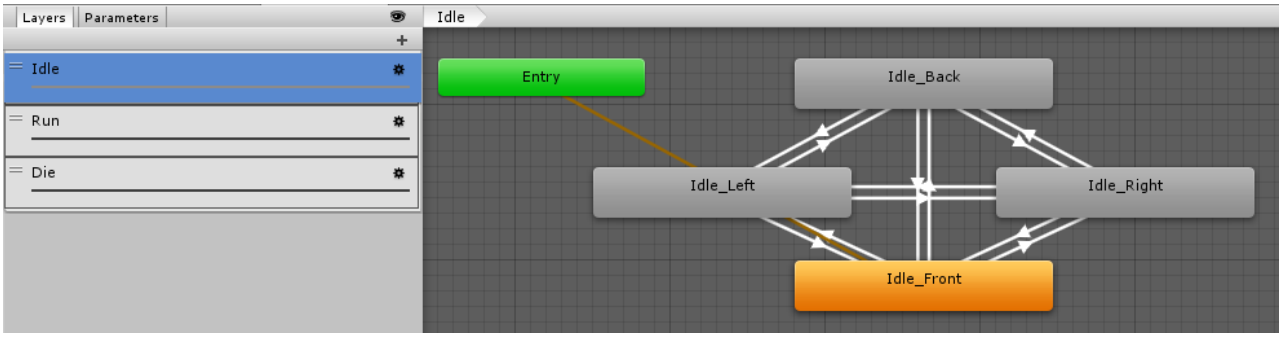
Ova skripta pritiskom na tipku *space* (ili na tipku *shift* kod drugog igrača) postavlja bombu na mjesto igrača. Kako bi skripta radila potrebno je imati *pivot* odnosno početnu točku gdje se igrač trenutno nalazi. Zatim se ta točka dodijeli vektoru *worldPos* koji tu točku proslijedi mapi nivoa Tako se dobije ćelija od koje zatim pomoću funkcije *GetCellCenterWorld* dobijemo središnju točku kako bi se bomba smjestila u samu sredinu ćelije. Bombu zatim stavimo na mjesto funkcijom *Instantiate* kojoj smo dodijelili prefab bombe, koordinate središta ćelije i *Quaternion.identity* koji govori funkciji da bombu rotira ovisno o nivou.



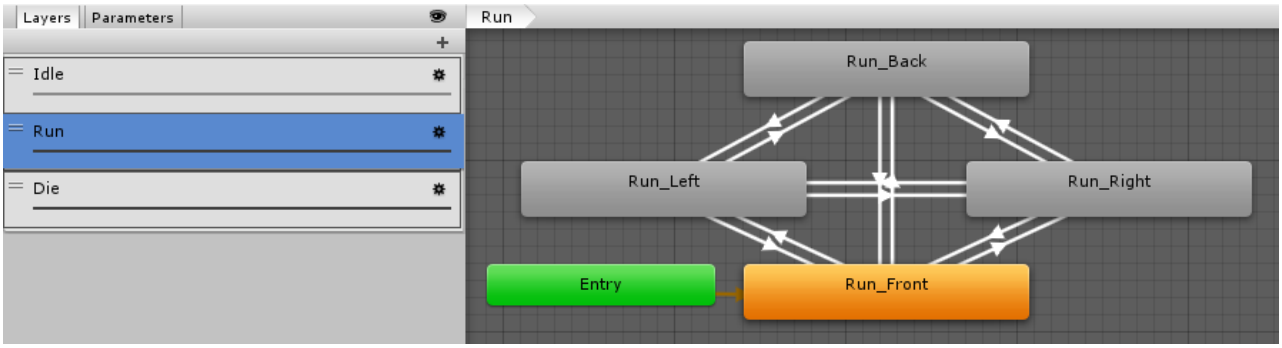
Sl. 3.2. Skripta BombSpawner u inspektoru

3.3. ANIMACIJE

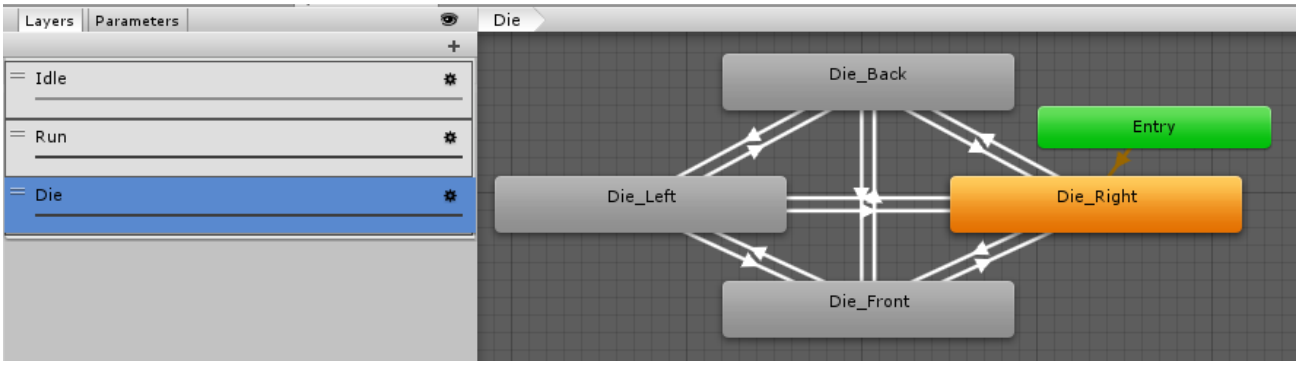
Za animacije igrača imamo tri sloja animacija koje ovisno o stanju igrača prevladavaju jedna drugu. Kada se igrač kreće prevladava sloj *Run*, a kada igrač miruje prevladava sloj *Idle*. Također postoji i sloj *Die* koji se aktivira kada broj života igrača padne na nulu.



Sl. 3.3. Idle sloj i njegovi prijelazi



Sl. 3.4. Run sloj i njegovi prijelazi



Sl. 3.5. Die sloj i njegovi prijelazi

Kako bi se vidjele promjene između animacija potrebno je napraviti prijelaze između svakog stanja lika. Također potrebno je kreirati varijable *MovingX* i *MovingY* te im za svaku promjenu postaviti uvjete koji se provjeravaju za svaku sliku (*frame)* . Jedan od prijelaza je tako iz kretanja prema dolje i kretanje ulijevo. Do ove promjene dođe kada je varijabla *MovingX* manja od nule, a varijabla *MovingY* manja od 0.1. Na slikama 3.3.1., 3.3.2. i 3.3.3. se vidi da za svaki smjer prema kojem igrač gleda postoji prijelaz na drugi smjer kada do promjene dođe.

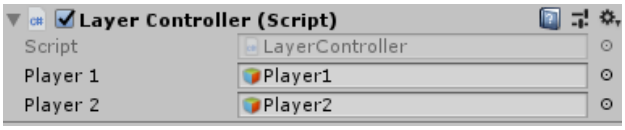
3.4. KAMERA

Kako bi se u *game* kartici, odnosno u pogledu na koji igrač ima dok igra igru išta vidjelo potrebno je imati kameru. Kamera predstavlja pogled na nivo, te joj se kao i svakom drugom objektu u igri mogu dodijeliti skripte i podesiti pozicija. Osim toga postoji i dosta opcija koje omogućavaju detaljno podešavanje vidljivosti same scene. Neke od njih su : boja pozadine, vrsta projekcije, dubina i različitih tehnika poput HDR-a i MSAA koje omogućuju veći dinamički raspon boja i kvalitetu slike. Kako je MSAA jako zahtjevna tehnika, u ovom radu je korištena tehnika FXAA [8]. Ona za razliku od MSAA ne zahtjeva puno procesorske snage, a i dalje je jako efektivna. Za implementaciju ove tehnike bilo je potrebno preuzeti s Unity Asset Store-a paket pod nazivom „FXAA Fast Approximate Anti-Aliasing“ [9] te dobivenu skriptu staviti na kameru. Nadalje, kako je ovo 2.5D igra, pojavio se određeni problem preklapanja igrača kada su igrači u međusobnom kontaktu. Ovaj problem se riješio kreiranjem skripte *LayerController* i njezinim dodavanjem na kameru. Ova skripta se može dodati na bilo koji objekt, ali najlogičniji izbor je bila kamera jer se ona bavi prikazom cjelokupnog nivoa.

```
using UnityEngine;

public class LayerController : MonoBehaviour {

    private SpriteRenderer player1renderer;
    private SpriteRenderer player2renderer;

    [SerializeField]
    private GameObject player1;
    [SerializeField]
    private GameObject player2;

       void Start () {
        player1renderer = player1.GetComponent<SpriteRenderer>();
        player2renderer = player2.GetComponent<SpriteRenderer>();
       }

       void Update () {
        double pos1 = player1.transform.position.y;
        double pos2 = player2.transform.position.y;


        if(pos1 > pos2)
        {
            player1renderer.sortingOrder = 3;
            player2renderer.sortingOrder = 4;
        }
        else if(pos1 < pos2)
        {
            player1renderer.sortingOrder = 4;
            player2renderer.sortingOrder = 3;
        }
       }
}
```

Programski kod 3.4. Skripta LayerController

U skripti se inicijaliziraju dva objekta koji predstavljaju igrače. Zatim je u *Start()* funkciji potrebno dohvatiti *SpriteRenderer* od svakog igrača kako bi se igračima mogla mijenjati pozicija u sloju unutar igre. Kako su nam je nivo igre postavljen na sloju 0 i 1, a bomba na sloj 2, igračima je bilo potrebno postavili vrijednosti više od tih. U funkciji *Update()* kontinuirano se dohvaćaju trenutne vrijednosti varijable *y* igrača, te se zatim te vrijednosti uspoređuju. Ako se prvi igrač nalazi iznad drugog igrača, sloj igrača 1 se postavlja na 3, a sloj igrača 2 na 4 i obrnuto ukoliko se prvi igrač nalazi ispod drugog igrača. Nakon napisane skripte potrebno je u inspektoru dodijeliti objekte koji predstavljaju igrače.



Sl. 3.6. Skripta LayerController u inspektoru

3.5. Bombe

Sl. 3.7. Slike bombi

Kao što je ranije rečeno, igrači postavljaju bombe pritiskom na tipku. Kako bismo mehaniku igre učinili što zanimljivijom potrebno je kreirati određene uvjete za bombe. U skripti BombSpawner korištena je metoda *Instantiate*. Ona prima već postojeći objekt od kojega želimo napraviti kopiju. U ovom slučaju smo koristili gotovi objekt bombe kojemu je prethodno dodana animacija i skripta *Bomb* kojom je omogućena funkcionalnost bombe. Kao što vidimo na slici 3.7. korištene su tri slike bombi. One imaju istu funkcionalnost, ali zbog lakšeg uočavanja bombe u odnosu na nivo, svaka od njih se koristi u drugom nivou igre.

```
using UnityEngine;

[RequireComponent(typeof(AudioClip))]

public class Bomb : MonoBehaviour
{
    [SerializeField]
    private float countdown;
    public AudioClip explosionSound;
    public Collider2D bombCollider;
    private float timer;

    private void Start()
    {
        bombCollider = GetComponent<Collider2D>();
```

```
        timer = 0f;
    }
    void Update()
    {
        timer += Time.deltaTime;
        GetComponent<Renderer>().material.color = Color.Lerp(Color.white, Color.red, timer /
countdown);

        if (timer >= countdown)
        {
            FindObjectOfType<MapDestroyer>().Explode(transform.position);
            Destroy(gameObject);
            AudioSource.PlayClipAtPoint(explosionSound, transform.position);
        }
    }
    private void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.gameObject.CompareTag("Player"))
        {
            bombCollider.isTrigger = false;
        }
    }
}
```

Programski kod 3.5. Skripta Bomb

Nakon deklaracije varijabli potrebno je u *Start()* funkciji dohvatiti komponentu *Collider2D* kako bismo ju mogli koristiti. Prilikom instanciranja bombe tajmer se postavlja na nulu, te se on zatim kako vrijeme ide povećava, dok on ne bude veći ili jednak varijabli *countdown* koja ima vrijednost da odgodi eksploziju bombe. Za vrijeme odbrojavanja, boja bombe se pomoću *Renderer* komponente mijenja iz početne u crvenu. To daje do znanja igračima koliko imaju vremena da se sklone od nadolazeće eksplozije kako ne bi primili štetu. Kada dođe do eksplozije poziva se metoda *Explode* iz skripte *MapDestroyer* (nešto više o ovoj skripti kasnije). Bomba se zatim uništava i reproducira se zvuk eksplozije na mjestu bombe. Svaka bomba ima određeni radijus oko svog tijela koji služi za detekciju kolizije i aktivaciju određenih radnji. Pošto svaki igrač na svom mjestu stajanja stvara bombu, potrebno je na komponenti *Collider2D* označiti *IsTrigger*. U skripti kao što se vidi postoji poruka *OnTriggerExit2D* koja dojavljuje kada objekt prestane dodirivati bombu. U našem slučaju to je igrač. Kada do toga dođe *isTrigger* se postavi na *false*. To znači da komponenta *Collider2D* sada postaje kao ograničenje koja blokira kretanje na mjestu gdje se bomba nalazi. Ova mehanika dodaje određenu dozu zanimljivosti igri.

3.6. Nivoi

Svaki nivo u igri predstavlja novu scenu od kojih svaka od njih koristi druge mehanike i vizualno je drugačija. Postoje ukupno tri nivoa, te se putem glavnog menija može izabrati bilo koji od njih. U svakom od nivoa postoje blokovi koji predstavljaju tlo po kojemu igrač hoda, blokovi koji su uništivi i oni koji su neuništivi. Uništivi blokovi mogu se uništiti bombama, te se zatim nasumce odabire hoće li se pojaviti neka od vrsti pojačanja za igrača ili ne. U igri postoje dvije vrste pojačanja. Pojačanja koja povećavaju brzinu igrača i pojačanja koja igraču dodjeljuju više bombi koje može baciti.

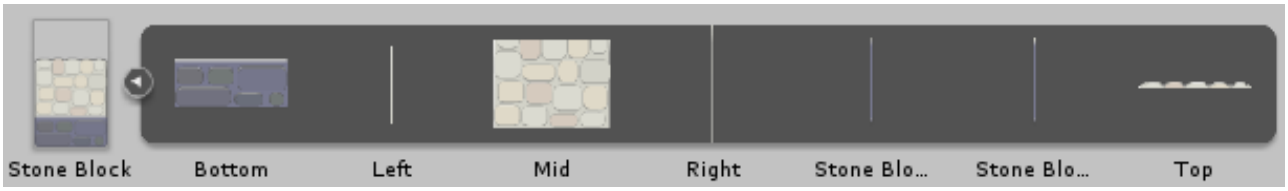
Sl. 3.8. Pojačanja unutar igre

Za lakše kreiranje nivoa korištene su „pločice“ i koordinatna mreža koja čini rešetku. Prvo je bilo potrebno razrezati svaki dio podloge na više dijelova kako bi se odvojio najoptimalniji dio koji možemo iskoristiti za pločicu. Nakon što smo to napravili u Unity-u unutar hijerarhije u kojoj se nalaze svi korišteni objekti u nivou potrebno je kreirati rešetku. Nakon početnog podešavanja veličine svake rešetke potrebno je duplicirati postojeću rešetku i dati joj sloj -1 kako se pozadinske pločice i one koje utječu na igru ne bi crtale jedne preko drugih.

Sl. 3.9. Pločice

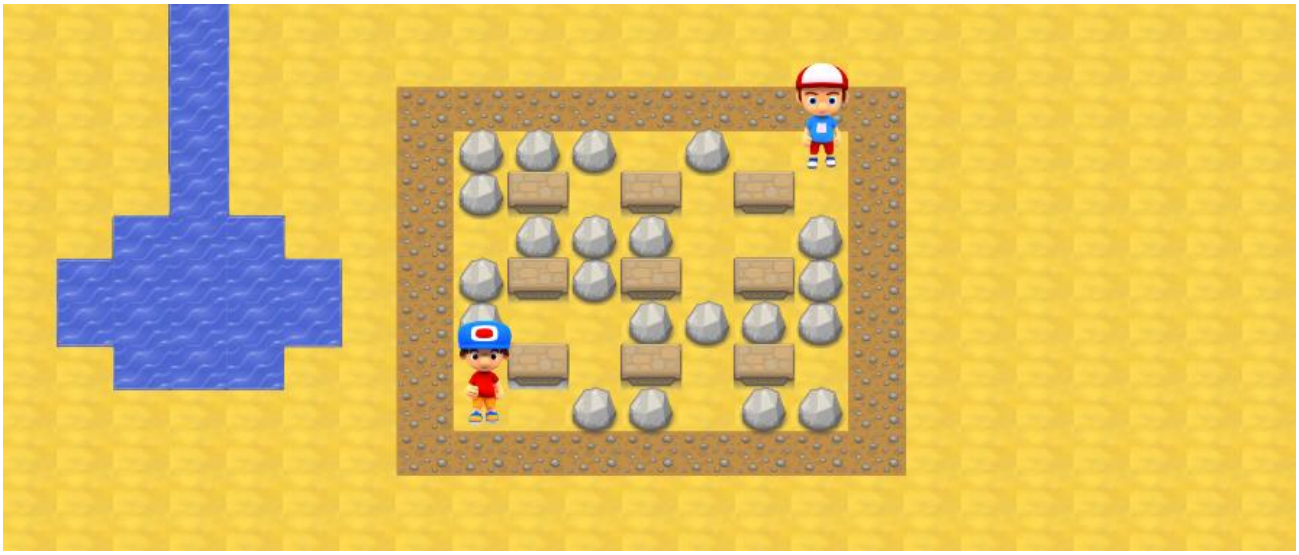
Nakon što smo kreirali pločice potrebno je kliknuti na željenu pločicu i povući mišem po sceni kako bi se one nacrtale. Ovo štedi vrijeme prilikom kreiranja nivoa. Zatim kako bi se pločicama dao njihov izvorni oblik i kako bi se izgled nivoa uljepšao potrebno im je dodati svaki od odrezanih elemenata.

Sl. 3.10. Izgled pločice prije rezanja



Sl. 3.11. Primjer pločice nakon rezanja

Svakom nivou je potrebno nakon crtanja pločica dodati obrube i elemente kako bi nivo vizualno izgledao što bolje.



Sl. 3.12. Primjer nivoa s pločicama

Sl. 3.13. Primjer nivoa nakon dodavanja svih elemenata

Osnovna mehanika ove igre je uništavanje blokova bombom. Kako bismo u tome uspjeli potrebna nam je skripta koja raspoznaje određeni tip bloka te ga „briše“ ovisno o tome je li ga moguće uništiti ili nije.

```
using UnityEngine;
using UnityEngine.Tilemaps;

public class MapDestroyer : MonoBehaviour {

    public GameObject bombCount;
    public GameObject speedUp;
    public GameObject blank;
    public GameObject explosionPrefab;

    public Tilemap tilemap;
    public Tile wallTile_1;
    public Tile wallTile_2;
    public Tile wallTile_3;
    public Tile destructibleTile;

    private GameObject prefab;
    private GameObject[] arrayPowerup;
    private GameObject powerUp;

    private void Start()
    {
        arrayPowerup = new GameObject[] { bombCount, speedUp, blank };
    }


    void Update()
    {
        powerUp = arrayPowerup[Random.Range(0, arrayPowerup.Length)];
    }
    public void Explode(Vector2 worldPos)
       {
             Vector3Int originCell = tilemap.WorldToCell(worldPos);

        ExplodeCell(originCell);

             if (ExplodeCell(originCell + new Vector3Int(1, 0, 0)))
             {
                    ExplodeCell(originCell + new Vector3Int(2, 0, 0));
        }

             if (ExplodeCell(originCell + new Vector3Int(0, 1, 0)))
             {
            ExplodeCell(originCell + new Vector3Int(0, 2, 0));
        }

        if (ExplodeCell(originCell + new Vector3Int(-1, 0, 0)))
             {
            ExplodeCell(originCell + new Vector3Int(-2, 0, 0));
        }

        if (ExplodeCell(originCell + new Vector3Int(0, -1, 0)))
             {
            ExplodeCell(originCell + new Vector3Int(0, -2, 0));
        }
    }
```

```
        bool ExplodeCell (Vector3Int cell)
        {
               Tile tile = tilemap.GetTile<Tile>(cell);

               if (tile == wallTile_1 || tile == wallTile_2 || tile == wallTile_3)
               {
                      return false;
               }

               if (tile == destructibleTile)
               {
                      tilemap.SetTile(cell, null);
                   Instantiate(powerUp, tilemap.GetCellCenterWorld(cell), Quaternion.identity);
                   if(powerUp == blank) { powerUp.SetActive(false); }
         }

               Vector3 pos = tilemap.GetCellCenterWorld(cell);
               prefab = Instantiate(explosionPrefab,pos, Quaternion.identity);
               Destroy(prefab, 0.35f);
               return true;
        }
}
```

Programski kod 3.6. Skripta MapDestroyer

Nakon deklaracije potrebnih objekata i varijabli treba instancirati polje sa svim pojačanjima u igri kako bi mogli nasumično odabrati koji od njih će se stvoriti kada se blok uništi. U funkciji *Update()* stalno se nasumično vrti polje brojeva koje prilikom uništenja blokova instanciramo. Metoda *Explode()* brine se o uništenju pojedinih blokova. Ovu metodu kao što smo ranije mogli vidjeti pozivamo iz skripte *Bomb* nakon što prođe određeno vrijeme nakon postavljanja bombe. Metoda prima mjesto bombe za koje je zatim potrebno odrediti ID ćelije. Tada se poziva metoda *ExplodeCell()* na originalno mjesto bombe, a nakon toga pomoću *if* uvjeta omogućujemo širenje eksplozije lijevo, desno, gore i dolje od originalnog mjesta eksplozije. Metoda *ExplodeCell* funkcionira tako da primi poziciju ćelije i na osnovu koordinata sazna da li je pločica uništiva ili nije. Ako nije, onda vraća *false* , a ako je, onda se instancira određena vrsta pojačanja za lika koje on zatim može pokupiti. Ako je pojačanje *blank* onda ga se deaktivira kako ono ne bi blokiralo kretanje lika. Zatim se na svako od mjesta gdje se blok nalazi, pojavljuje animacija eksplozije koja zatim oštećuje protivnika ako se nalazi u njezinom dometu. Nakon 0.35 sekundi animacija se uništava.

Nakon stvaranja određene vrste pojačanja potrebno je napraviti da se ono uništi kada dođe u dodir s igračem. To se napravi sljedećom skriptom.

```
using UnityEngine;

public class DestroyOnCollision : MonoBehaviour {

    public GameObject powerUp;

    private void OnCollisionEnter2D(Collision2D collision)
    {
        Destroy(powerUp);
    }
}
```

Programski kod 3.7. Skripta DestroyOnCollision

U trećem nivou postoje portali koji omogućuju dvosmjerno teleportiranje likova koji uđu u njih. Na svaki od portala se postavi sljedeća skripta.

```
using System.Collections;
using UnityEngine;

public class portalManager : MonoBehaviour {
    [SerializeField]
    private float waitTime;
    private bool isUsed = false;
    public portalManager2 portalScript;
    public GameObject endPoint;

    private void OnTriggerStay2D(Collider2D player)
    {
        if (isUsed == false)
        {
            player.transform.position = endPoint.transform.position;
            isActivated();
            portalScript.isActivated();
            StartCoroutine(Loop(waitTime));
        }
    }
    public IEnumerator Loop(float waitTime)
    {
        yield return new WaitForSeconds(waitTime);
        isActivated();
        portalScript.isActivated();
    }
    public void isActivated()
    {
        isUsed = !isUsed;
    }
}
```

Programski kod 3.8. Skripta portalManager

Skripta radi tako što premjesti igrača na drugi portal , zatim provjerava da li je portal nedavno korišten, te ako je, onda se varijabla *isUsed* na oba portala postavi na *true.* Zatim se poziva korutina koja nakon dvije sekunde ponovno omogući kretanje između portala.

Kako portal postoji samo kao jedna slika, radi ljepšeg izgleda skripta *portalRotation* tijekom vremena rotira portal kako bi se prikazala određena promjena iako ona ne postoji kao animacija.

```
using UnityEngine;

public class portalRotation : MonoBehaviour {

    private void FixedUpdate()
    {
        transform.Rotate(0, 0, 5);
    }

}
```

Programski kod 3.9. Skripta portalRotation

Sl. 3.14. Prvi nivo

Sl. 3.15. Drugi nivo

Sl. 3.16. Treći nivo

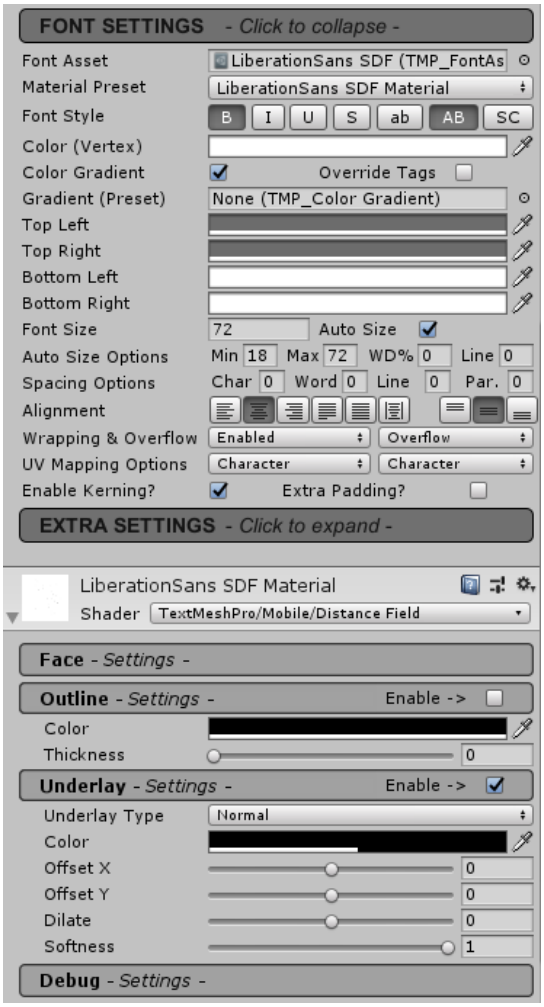
Prvi nivo ima definiranu širinu eksplozije dvije pločice, dok ostala dva nivoa imaju širinu eksplozije jedne pločice.

3.7. Korisničko sučelje

Svaka scena prikazuje podatke o preostalom vremenu i broju života za oba igrača. U slučaju kada se igrač ošteti, brojač se automatski ažurira. Kod pauziranja igre, koje je moguće u bilo kojem trenutku, brojač o preostalom vremenu se također zaustavi. Cjelokupno sučelje je prilagođeno za rad na svim rezolucijama.

Sl. 3.17. Korisničko sučelje unutar igre

Kako bi sučelje funkcioniralo potrebno je imati *canvas.* On predstavlja držač elemenata u korisničkom sučelju. On osim elemenata sa slike 3.7.1 sadrži sučelje koje je skriveno, a koje se prikazuje samo u slučaju pauze ili završetka igre. Sav tekst vidljiv korisniku je iz paketa *TextMeshPro.* Ovaj paket je vlasništvo Unity Technologies i u nadolazećim ažuriranjima Unity-a se očekuje potpuna integracija i izbacivanje postojećeg rješenja za uređivanje teksta. Pomoću njega možemo vrlo lako postaviti vrstu fonta, način sjenčanja teksta, gradijent teksta i slično, te je u svakom pogledu moćnija alternativa u odnosu na trenutno ugrađeni način obrade teksta u Unity-u.



Sl. 3.18. Neke od mogućnosti TextMeshPro-a

Tipkom ESC pauziramo igru, te nam se prikazuje sučelje koje nam omogućuje povratak u igru ili povratak na početni zaslon gdje možemo odabrati nivo, podesiti opcije ili izaći iz igre.

Sl. 3.19. Sučelje dok je igra zaustavljena

```
using UnityEngine;
using TMPro;

public class UI_Timer : MonoBehaviour {

    private TextMeshProUGUI countdown;
    private bool isPaused;

    [SerializeField]
    private float timer;

       void Start () {
        countdown = GetComponent<TextMeshProUGUI>();
        FindObjectOfType<GameManager>().GetTimer(timer);
    }
       void Update () {
        if (isPaused == false)
        {
            timer -= Time.deltaTime;
        }
        if (timer <= 0)
        {
            countdown.text = "0";

            FindObjectOfType<GameManager>().GetTimer(0);
        }
        else
        {
            FindObjectOfType<GameManager>().GetTimer(timer);
            countdown.text = Mathf.Round(timer).ToString();
        }
       }
```

```
    public void GetIsPaused(bool isPaused)
    {
        this.isPaused = isPaused;
    }
}
```

Programski kod 3.10. Skripta UI_Timer

Ova skripta odbrojava vrijeme do kraja nivoa. Nakon deklaracije varijabli potrebno je dohvatiti komponentu kako bismo mogli uređivati tekst putem skripte. Nakon toga se vrijednost brojača kontinuirano ažurira i tu vrijednost prosljeđuje skripti *GameManager* koja prati vrijeme. Metoda *GetIsPaused* govori brojaču da li je igra trenutno zaustavljena ili nije. Ovom metodom pristupamo iz skripte *GameManager*.

Skripta *GameManager* upravlja cjelokupnim radom pojedinog nivoa u igri. Ona prati vrijeme i broj života svakog igrača, te ovisno o njima odlučuje do kojeg ishoda u igri je došlo.

```
using UnityEngine;
using TMPro;
public class GameManager : MonoBehaviour {

    private float time_left;
    private bool isPaused = false;
    private int player1health;
    private int player2health;
    [SerializeField]
    private GameObject gameplaySong;
    [SerializeField]
    private GameObject pauseMenu;
    [SerializeField]
    private GameObject eventMenu;
    [SerializeField]
    public TextMeshProUGUI conclusionText;
    public GameObject player1;
    public GameObject player2;
    public TextMeshProUGUI p1Health;
    public TextMeshProUGUI p2Health;


    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            if (isPaused == false)
            {
                Time.timeScale = 0;
                isPaused = true;
                FindObjectOfType<UI_Timer>().GetIsPaused(isPaused);
                pauseMenu.SetActive(true);
            }
```

```
        else if(isPaused == true)
        {
            Time.timeScale = 1;
            isPaused = false;
            FindObjectOfType<UI_Timer>().GetIsPaused(isPaused);
            pauseMenu.SetActive(false);
        }

    }

    if(player1health > 0 && player2health <= 0 && time_left > 0)
    {
        conclusionText.text = "Player 1 WINS!";
        FindObjectOfType<UI_Timer>().GetIsPaused(isPaused);
        eventMenu.SetActive(true);
        gameplaySong.SetActive(false);
    }
    else if(player2health > 0 && player1health <= 0 && time_left > 0)
    {
        conclusionText.text = "Player 2 WINS!";
        FindObjectOfType<UI_Timer>().GetIsPaused(isPaused);
        eventMenu.SetActive(true);
        gameplaySong.SetActive(false);
    }
    else if(time_left <= 0 || player1health <= 0 && player2health <= 0)
    {
        conclusionText.text = "DRAW!";
        FindObjectOfType<UI_Timer>().GetIsPaused(isPaused);
        eventMenu.SetActive(true);
        gameplaySong.SetActive(false);

    }
}

public void GetP1Health(int health)
{
    player1health = health;
    p1Health.text = "Player 1 Health\n" + player1health.ToString();
}
public void GetP2Health(int health)
{
    player2health = health;
    p2Health.text = "Player 2 Health\n" + player2health.ToString();
}
public void GetTimer(float time)
{
    time_left = time;
}
public void Unpause()
{
    Time.timeScale = 1;
    isPaused = false;
    pauseMenu.SetActive(false);
}
public void IsPaused()
{
    Time.timeScale = 1;
    isPaused = false;
```

```
        FindObjectOfType<UI_Timer>().GetIsPaused(isPaused);
        pauseMenu.SetActive(false);
    }
}
```

Programski kod 3.11. Skripta GameManager

Nakon početnih deklaracija varijabli, skripta u funkciji *Update(),* kontinuirano prati tijek igre. Ako dođe do pritiska tipke ESC, igra se zaustavlja, skripta aktivira metodu *GetIsPaused* iz skripte *UI_Timer* kako bi se brojač zaustavio i prikazuje se meni pauze. Ukoliko brojač postane jednak nuli, ili broj života nekog od igrača padne na nulu, prikazuje se dijalog koji nam govori tko je pobijedio ili je došlo do izjednačenja. Svaki nivo igre ima jedinstvenu glazbenu podlogu koja se isključuje ukoliko dođe do kraja nivoa. Bilo koja promjena scene ima popratnu scenu koja prikazuje učitavanje.

Sl. 3.20. Scena učitavanja

```
using UnityEngine;
using System.Collections;
using TMPro;
using UnityEngine.SceneManagement;

public class SceneLoader : MonoBehaviour
{
    private bool loadScene = false;

    [SerializeField]
    private int scene;
    [SerializeField]
    private TextMeshProUGUI loadingText;

    void Update()
    {

        if (loadScene == false)
        {
            loadScene = true;
            loadingText.text = "Loading...";
            StartCoroutine(LoadNewScene());
        }
        if (loadScene == true)
        {
            loadingText.color = new Color(loadingText.color.r, loadingText.color.g,
loadingText.color.b, Mathf.PingPong(Time.time, 1));
        }

    }
    IEnumerator LoadNewScene()
    {
        yield return new WaitForSeconds(3);
        AsyncOperation async = SceneManager.LoadSceneAsync(scene);

        while (!async.isDone)
        {
            yield return null;
        }

    }

    public void SceneGetter(int scene)
    {
        this.scene = scene;
    }

}
```

Programski kod 3.12. Skripta SceneLoader

Ova skripta dohvaća novu scenu, prikazuje tekst učitavanja i prikladnu animaciju dok se nova scena učitava.

Prilikom prvog ulaska u igru prikazuje se početni ekran na kojemu možemo birati nivo, podešavati rezoluciju i kvalitetu tekstura same igre. Također je moguće podesiti glasnoću igre i pregledati kontrole za pojedinog lika.

Sl. 3.21. Glavni meni

Sl. 3.22. Birač nivoa

Sl. 3.23. Prikaz kontrola

Kako bi svaki gumb u meniju nešto radio, potrebno mu je pridružiti objekt koji sadrži skriptu s kontrolama i pripadajuću akciju na gumb. Naš objekt s kontrolama se zove *MainMenu* te mu je pridružena skripta pod istim nazivom i ona sadrži sve akcije koje se mogu dogoditi među pojedinim gumbovima. Za gumbove *Play* i *Back* ne treba napredna logika pa se zato koristi mogućnost paljenja i gašenja pojedinih dijelova canvasa koju Unity ima već ugrađenu i ne treba nikakvo dodatno programiranje.

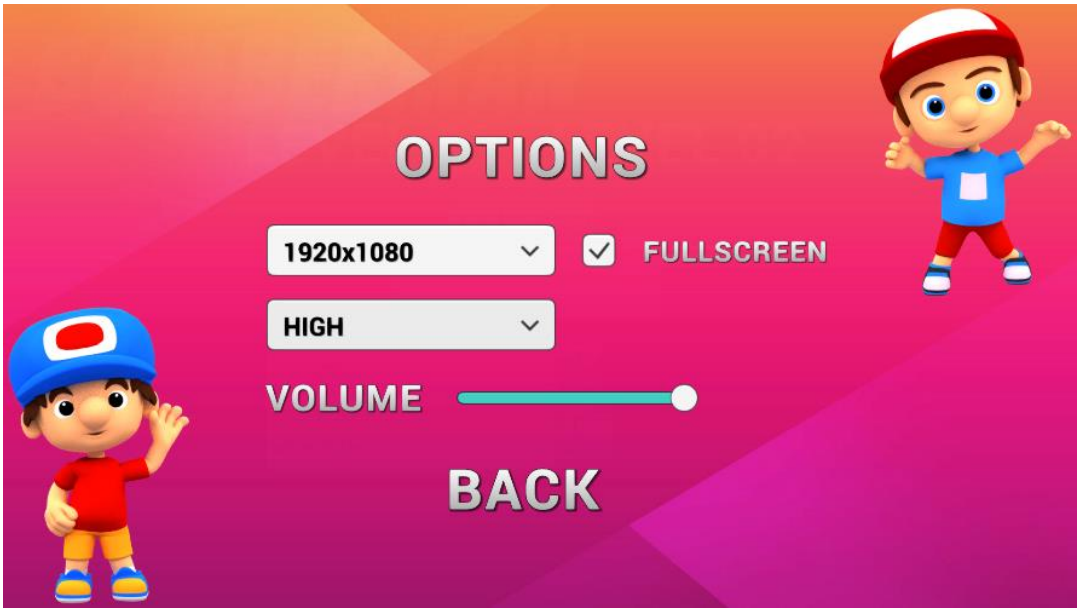
```
using UnityEngine;
using UnityEngine.SceneManagement;

public class MainMenu : MonoBehaviour {

    public void Return()
    {
        FindObjectOfType<GameManager>().IsPaused();
    }
    public void QuitGame()
    {
        Application.Quit();
    }
    public void Level01()
    {
        SceneManager.LoadScene(4);
    }
```

```
    public void Level02()
    {
        SceneManager.LoadScene(5);
    }
    public void Level03()
    {
        SceneManager.LoadScene(6);
    }
    public void LoadMenu()
    {
        SceneManager.LoadScene(0);
    }
    public void LoadingLevel01()
    {
        Time.timeScale = 1;
        SceneManager.LoadScene(4);
    }
    public void LoadingLevel02()
    {
        Time.timeScale = 1;
        SceneManager.LoadScene(5);
    }
    public void LoadingLevel03()
    {
        Time.timeScale = 1;
        SceneManager.LoadScene(6);
    }
}
```

Programski kod 3.13. Skripta MainMenu

Svaki gumb ima svoju javnu metodu kojom on pristupa kada se na njega klikne. Gumbovi unutar postavki imaju svoju vlastitu skriptu koja njima upravlja. Metoda *Return()* nastavlja s igrom, *QuitGame()* gasi igru, gumb *Level()* učitava odabrani nivo. Gumb *LoadMenu()* učitava početni zaslon igre, a *LoadingLevel()* prikazuje ekran učitavanja ovisno o odabranom nivou.

Meni postavki sastoji se od dva padajuća izbornika, jedan za postavke rezolucije, a drugi za kvalitetu teksturi, gumba *Fullscreen* koji se može uključiti i isključiti i klizača koji upravlja glasnoćom same igre. Kako su postavke za rezoluciju individualne, za svako računalo omogućeno je popunjavanje padajućeg izbornika ovisno o rezoluciji koju računalo podržava. Padajući izbornik za kvalitetu same igre ima tri moguće postavke, niska, srednja i visoka.



Sl. 3.24. Prikaz postavki

4. ZAKLJUČAK

U ovom završnom radu detaljno je opisan postupak izrade računalne igre za više igrača. Glavni alat za izradu ove igre je Unity Engine, a korišteni programski jezik je C#. Na početku smo ukratko objasnili što je to Unity te koje su mu namjene, kao i određene dodatke koje Unity sadrži kako bi olakšali posao korisnicima ovog alata. Glavni dio ovog rada objašnjava svaki korak izrade ove igre, od animacija do programskog koda pri čemu je svaki dio koda objašnjen kako radi. Za izradu ove igre potrebno je znati služiti se Unity sučeljem, programskim jezikom C#, te imati nešto naprednije znanje objektno orijentiranog programiranja. Također je potrebno imati osnovno poznavanje vektora zato što vektore koristimo za kretnje lika. Ovaj završni rad predstavlja kostur igre Bomberman te je vrlo jednostavno dodati nove funkcionalnosti poput više nivoa i igrača.

LITERATURA

[1] Lost Garden,

http://www.lostgarden.com/ [24.06.2018.]

[2] Graphic River,

https://graphicriver.net [24.06.2018.]

[3] Anttis Instrumentals,

https://www.soundclick.com/bands3/default.cfm?bandid=1277008 [30.06.2018]

[4] Unity Technologies,

https://en.wikipedia.org/wiki/Unity_Technologies [15.06.2018.]

[5] Unity Asset Store,

https://assetstore.unity.com/ [01.07.2018.]

[6] Visual Studio IDE,

https://visualstudio.microsoft.com/ [26.06.2018.]

[7] Unity Documentation,

https://docs.unity3d.com/Manual/index.html [26.06.2018.]

[8] FXAA,

https://en.wikipedia.org/wiki/Fast_approximate_anti-aliasing [01.07.2018.]

[9] Anti-Aliasing,

https://en.wikipedia.org/wiki/Spatial_anti-aliasing [01.07.2018.]

SAŽETAK

U ovom radu je detaljno opisana izrada računalne igre za više igrača u Unity Engine-u. Postoje dva igriva lika koji svaki ima svoj set kontrola. Cilj igre je skupljati pojačanja koja se stvore uništavanjem blokova i poraziti protivnika. Igrači se mogu kretati gore-dolje, lijevo-desno i imaju sposobnost bacanja bombi. Svaki igrač ima tri seta animacija i objašnjeno je kako upravljati animacijama kako bi se dobio vjerni izgled kretanja lika. Objašnjen je princip rada svake skripte i objašnjena je izrada funkcionalnog korisničko sučelja. Programski jezik korišten za izradu skripti je C#.

Ključne riječi: 2.5D računalna igra za više igrača, animator, C#, nivo, scena, Unity Engine

ABSTRACT

DEVELOPMENT OF 2.5D MULTIPLAYER ARCADE COMPUTER GAME

In this final paper, the process of making 2.5D multiplayer computer game is described in detail. The main tool used for making this game is Unity Engine, and the programming language used is C#. At the beginning we briefly explained what Unity is and what it is intended for, as well as certain additions that Unity has implemented to help ease the work for the users of this tool. The main part of this article explains every step of the game's development, from animation to program code, where each part of the code is explained how it works. To create this game, you need to know how to use the Unity interface, the C# programming language, and have advanced knowledge of object-oriented programming. It is also necessary to have a basic knowledge of the vectors because they are used for game physics and certain logic. This final work represents the skeleton of the classic Bomberman game with making it easy to add new functionalities such as more levels and players.

Key words: 2.5D multiplayer arcade computer game, animator, C#, level, scene, Unity Engine

ŽIVOTOPIS

Matej Jakšić rođen je u Osijeku, 8. studenog 1995. godine. Pohađao je osnovnu školu „Ivan Filipović“ u Osijeku. Nakon osnovne škole, 2010. godine upisuje Prirodoslovno i matematičku gimnaziju u Osijeku koju završava 2014. godine. Iste godine upisuje preddiplomski studij računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija Osijek kojeg trenutno pohađa.

Vlastoručni potpis

_____

Matej Jakšić