Izrada Unity 2D Arkadne igre s Vitezom u glavnoj ulozi

Klarić, David

MEĐIMURSKO VELEUČILIŠTE U ČAKOVCU

STRUČNI STUDIJ RAČUNARSTVO

DAVID KLARIĆ

IZRADA UNITY 2D ARKADNE IGRE S VITEZOM U GLAVNOJ ULOZI

ZAVRŠNI RAD

Čakovec, 2022.

MEĐIMURSKO VELEUČILIŠTE U ČAKOVCU

STRUČNI STUDIJ RAČUNARSTVO

DAVID KLARIĆ

IZRADA UNITY 2D ARKADNE IGRE S VITEZOM U GLAVNOJ ULOZI

CREATION OF UNITY 2D ARCADE WITH KNIGHT IN MAIN ROLE

ZAVRŠNI RAD

Mentor:

Nenad Breslauer, v. pred.

Čakovec, 2022.

MEĐIMURSKO VELEUČILIŠTE U ČAKOVCU
ODBOR ZA ZAVRŠNI RAD

Čakovec, 3. veljače 2020.

ZAVRŠNI ZADATAK br. 2019-RAČ-R-17

Pristupnik: **David Klarić (0313020155)**
Studij: redovni preddiplomski stručni studij Računarstvo
Smjer: Programsko inženjerstvo

Zadatak: **Izrada Unity 2D Arkadne igre s Vitezom u glavnoj ulozi**

Opis zadatka:

Izrada 2D računalne igre korištenjem platforme Unity. Izraditi scenu, radnja igra je smještena u prostoru tamnice. Igrač se bori protiv čudovišta i sakupljanjem zlatnika moće nadograditi snagu, štit i oružje. Koristiti platformu Unity, programski jezik C# te dodatne programske alate, Blender i ostale.
Završni rad mora sadržavati sažetak, sadržaj i uvod, nakon čega slijedi poglavlje u kojem je potrebno navesti i pojasniti osnovne ciljeve rada te očekivani rezultat. U narednom poglavlju potrebno je opisati primijenjene postupke, alate i metode. Poglavlje koje slijedi obrađivati će postignute rezultate nakon čega slijedi poglavlja u kojem se kritički raspravlja o primijenjenim metodama i postupcima te se u narednom poglavlju iznose glavni zaključci rada. Rad se završava poglavljima s popisom literature te prilozima.

Zadatak uručen pristupniku: 3. veljače 2020.
Rok za predaju rada: 20. rujna 2020.

Mentor:

Nenad Breslauer, v. pred.

Predsjednik povjerenstva za završni ispit:

ZAHVALA

Želim zahvaliti svim profesorima na prenesenom znanju, posebno mentoru čiji je kolegij uveliko doprinio mom znanju i vještinama prilikom izrade ovog rada. Također želim zahvaliti obitelji i prijateljima koji su mi bili podrška tijekom studiranja.

David Klarić

SAŽETAK

Videoigre su poznate kao bijeg od realnosti te svakodnevnog stresa kod većine populacije, dok neki igraju *videoigre* kako bi zaradili novce. Osim što postoji esport (elektronički sport) i moguća je zarada na turnirima, *vidoeigre* počinju utjecati i na filmsku industriju. Izradom filmova čija je tema nastala od postojećih *videoigara* kao što su „Uncharted“, „Detective Pikachu“ i „Sonic the Hedgehog 2“ ostvarili su zaradu na kino blagajnama veću od 400 milijuna dolara. Gotovo svakodnevno svjedočimo razvoju tehnologije koja je dostupna svima, stoga je nemoguće ne posjedovati bar jedan uređaj na kojemu se može igrati igre u slobodno vrijeme, bio to mobitel, igrača konzola, laptop ili osobno računalo.

U ovom radu prikazan je proces izrade 2D *arkadne videoigre* korištenjem razvojnog alata *Unity 2D*. *Unity* je jednostavno napravljeno korisničko sučelje koje svojim ugrađenim svojstvima olakšava rad te ne zahtijeva prethodno iskustvo novih korisnika i samih početnika. Također se koristim *programski jezik C#* u Microsoft Visual Studio programu.

U ovom radu opisani su svi koraci potrebni za samostalnu izradu 2D računalne igre, kao što su dizajniranje razina, postavljanje virtualne kamere i igrača, *animacije* i drugi. Kako bi igru otežali, dodaju se neprijatelji s kojima se igrač suočava te im se svima dodaju *animacije* da igra ne bi postala posve statična. Glavni je segment te igre trgovina koja igraču daje mogučnost nadogradnje svoje statistike. Pri tome bitno je skupiti sve zlatne novčiće koji se kasnije koriste u navedenoj trgovini. Nakon svih bitnih i funkcionalnih dijelova igre, dodaje se pozadinska glazba i zvučni efekti koji pridonose ugođaju igre. Cilj je *videoigre* eliminirati sve neprijatelje, skupiti sve zlatnike za nadogradnju te preživjeti sve razine igre. Igrač upravlja *vitezom* koji se bori protiv neprijatelja u tamnici, sakupljanjem zlatnika igrač ima mogućnost nadogradnje svoje statistike (eng. *stats*) kao što su snaga, štit i *damage*. Igrač prolazi kroz tri razine, na svakoj razini susreće se s neprijateljima te bodljikavim zamkama. Ukoliko igrač umre, ponovno polazi od prve razine i sav se postignut rezultat poništava.

Ključne riječi: *Unity 2D, arkada, programski jezik C#, videogira, vitez, animacije*.

Sadržaj

1. UVOD

Videoigre su interaktivni računalni programi namijenjeni za zabavu. Igre se igraju pomoću računala, mobitela ili konzole. U vrlo su kratkom razdoblju videoigre postale ključan dio života svakog djeteta te su postale jedan od najpopularnijih oblika zabave danas. Prva videoigra predstavljena je 1958. godine i nazivana je „Tenis za dvoje“. Danas videoigre čine globalnu industriju vrijednu više od 100 milijardi dolara, a gotovo dvije trećine američkih domova imaju članove kućanstva koji redovno igraju videoigre [1].

U današnje doba teško je ne susresti se s nekom vrstom videoigara, bilo na mobitelu, laptopu ili računalu. Nagli razvoj tehnologije i industrije videoigara omogućio je korisnicima pristup videoigrama na svim platformama, a pogotovo na mobilnim. Igre nisu samo stvorene kako bi se djeca zabavila, postoje i videoigre koje se koriste za obrazovanje djece. Osim za obrazovanje djece, postoje simulacije koje koriste odrasle osobe, kao što su profesionalni vozači koji koriste videoigre kako bi unaprijedili svoje vozačke vještine na pravim trkačim stazama. Jedan od tih simulatora je „iRacing“. U tom simulatoru nalaze se automobili i trkače staze koje profesionalni vozači koriste da bi razvili i isprobali sve potencijalne strategije te testirali aute na trkačim stazama pod raznim okolnostima [2].

2. ARKADNE IGRE

Arkadne su igre najčešće platformeri u kojima je igra podjeljena na različite razine. Platformeri su opisani kao igre u kojima se igrač kreće lijevo-desno po terenu, skačući po raznim platformama. Arkadne igre opisane su kao igre koje zahtijevaju puno vještina, savršeni su primjeri arkadnih igara: „Pac-Man“, „Donkey Kong“ i „Mortal Kombat“. Zlatno doba arkadnih igara bilo je razdoblje od kasnih 1970-ih do sredine 1980-ih. Onda su se pojavile igrače konzole, kao što su Playstation i Microsoft Xbox, koje su bile imale veliko poboljšanje grafike te nisu puno koštale [3].

Cilj je ovoga rada opis izrade 2D arkadne igre. Igra je napravljena za jednog igrača s idejom da se igrač zabavi i prilikom igranja istraži razine koje prelazi. Igrač prolazi kroz tri razine u kojima treba skupiti sve zlatnike i poraziti neprijatelje. Prikupljenim zlatnicima igrač može nadograditi svoj lik kako bi postao jači i tako si olakšao borbu s neprijateljima. Istraživanjem razina igrač nailazi na neprijatelje te zamke smještene na različitim dijelovima mape. Cilj je igre preživjeti sve razine te poraziti neprijatelje. Ukoliko igrač umre, sav prikupljen novac i nadogradnje lika poništavaju se te igrač može opet započeti igru samo od prve razine.

3. KORIŠTENI ALATI I TEHNOLOGIJE

Za izradu videogire korišten je Unity *engine* verzije 2019.3.3f1 koji koristi programski jezik C#, a zadani je program Microsoft Visual Studio.

3. 1. Unity

Unity (engl. *Cross-platform game engine)* je razvio *Unity Technologies* koji se primarno koristi za razvoj dvodimenzionalnih i trodimenzionalnih videoigara. Prvi put predstavljen je 2005. godine za OS X na *Apple's Worldwide Developers Conferenceu* i posebno je popularan za izradu iOS i Android mobilnih igara te se smatra jednostavnim za korištenje kod programera početnika. *Unity* koristi programski jezik C#, a zadani je program *Visual Studio.* Korisničko sučelje sastoji se od prozora koji služe za izradu komponenata i objekata koje se upotrebljavaju prilikom izrade igre. Neki su od elemenata sučelja projekt (eng. *Project*), scena (eng. *Scene*), inspektor (eng. *Inspector*), hijerarhija (eng. *Hierarchy*) te alatna traka. Na prozoru scena stvaraju se i postavljaju objekti koji se koriste u igri. Objekti (eng. *GameObject*) čine elemente koji se vide na sceni, a to su: igrač, neprijatelji, dizajn mape itd., koje se organiziraju u prozoru hijerarhija [4].

3. 2. Microsoft Visual Studio

Microsoft Visual Studio integrirano je razvojno okruženje (IDE) iz Microsofta. Koristi se za razvoj računalnih programa kao što su internetske stranice, internetske aplikacije te mobilne aplikacije. Unutar Visual Studia nalazi se IntelliSense komponenta koja ima mogućnost dopune koda i pronalaženja pogrešaka unutar već napisanog koda. Visual Studio podržava 36 različitih programskih jezika i može ispravljati greške za svaki programski jezik, ukoliko postoji usluga za to. Neki od programskih jezika koje podržava jesu: C, C++, C#, JavaScript, XML, HTML i CSS. Osnovna je verzija Visual Studia *Community* verzija koja je besplatna i namijenjena je studentima i samostalnim programerima. *Professional* i *Enterprise* verzije Visual Studia donose dodatne mogućnosti i njihova uloga pružanje je boljih usluga i svega što donosi Microsoftova pretplata [5].

3.3. Programski jezik C#

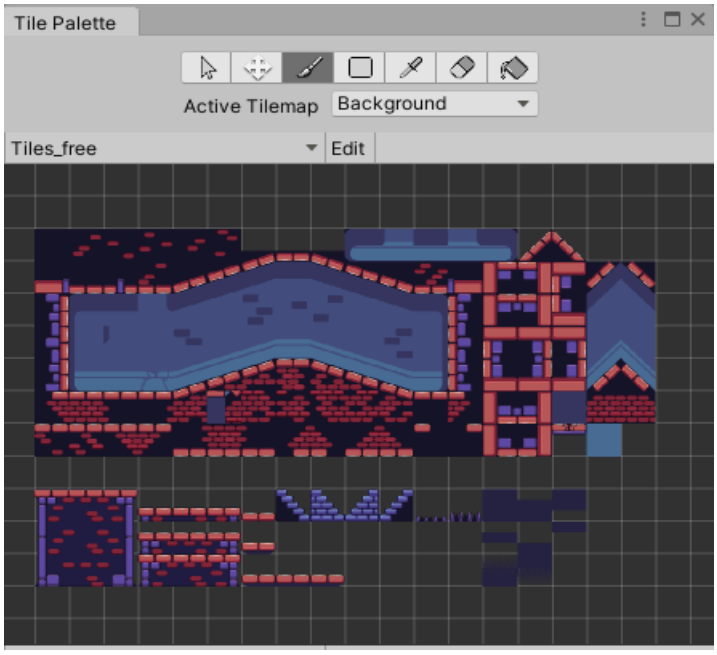
C# (C *sharp)* je moderni objektno orijentirani programski jezik. Razvio ga je Microsoft 2000. godine. Napravljen je kako bi zadovoljio sve veću potražnju za mrežnim aplikacijama koje Visual Basic i C++ nisu mogli zadovoljiti. Njegova arhitektura usvaja najbolje značajke Jave i C++. Kao rezultat toga, programeri koji dobro poznaju C i C++ mogu lako prijeći na C#. C# se koristi za izradu internetskih apilkacija, Windows aplikacija i videoigri. U svijetu videoigara programeri preferiraju C# programski jezik i Unity *game engine*, jedan od najpoznatijih alata za izradu videoigara koji je napravljen korištenjem C++ i C# programskim jezicima [6].

4. RAZVOJ VIDEOIGRE

Kroz ovo poglavlje opisuje se proces izrade videoigre. Najprije se objašnjava kreiranje razine i postavljanje kamera. Nakon toga objašnjavajaju se igrač i neprijatelji te njihove komponente. Za kraj objašnjeni su novčići te njihova uporaba u trgovini, razine igre i glavni izbornik.

4.1. Dizajniranje razine

Svakoj videoigri potrebne su razine po kojima će se igrač kretati. Razine u ovoj igri kreirane su pomoću *spriteova* koji se mogu izraditi u jednostavnim programima kao što je *Paint* jer je to zapravo *pixel art*. U ovom se projektu koriste *spriteovi* s Unity *asset storea* koje se nakon preuzimanja postavljaju u *Tile pallete* za lakši pristup pri dizajniranju razine [7].

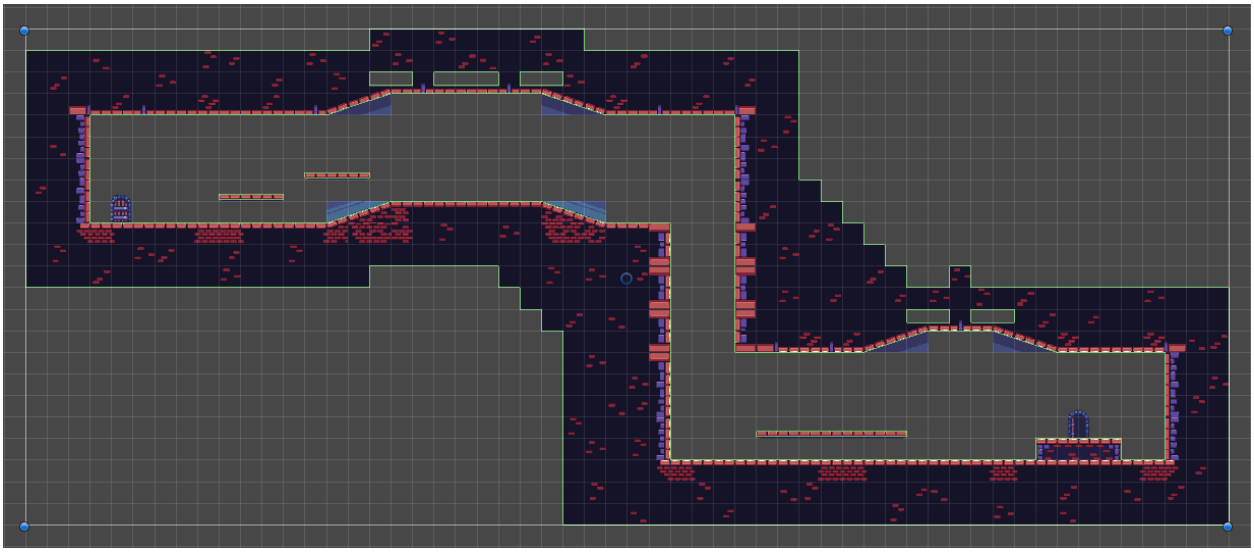


Slika 1. *Paleta spriteova za dizajn razina*

Izvor: Autor

Nakon dodavanja palete na scenu također se dodaje *Grid* koji se koristi kao platno za postavljanje *spriteova*. Klikom na određeni *sprite* dobiva se mogućnost postavljanja tog *spritea* unutar scene.

Također je važno, nakon što se dizajnira razina na *spriteove,* dodati *Tillemap Collider 2D* komponentu kako igrač ne bi propadao kroz razinu.



Slika 2. *Tilemap Collider 2D (zelena linija oko spriteova)*

Izvor: Autor

Na svakoj razini nalaze se vrata koja označavaju početak i kraj razine, vrata za početak postavljena su vizualno bez ikakvih komponenti i funkcionalnosti, dok se na vratima za kraj razine nalazi okidač (eng. *trigger*) i komponenta pod nazivom *PlayerFinishLevel*. Unutar te skripte nalazi se kôd koji prepoznaje igrača te mu daje mogućnost prelaska na drugu razinu pritiskom određene tipke.

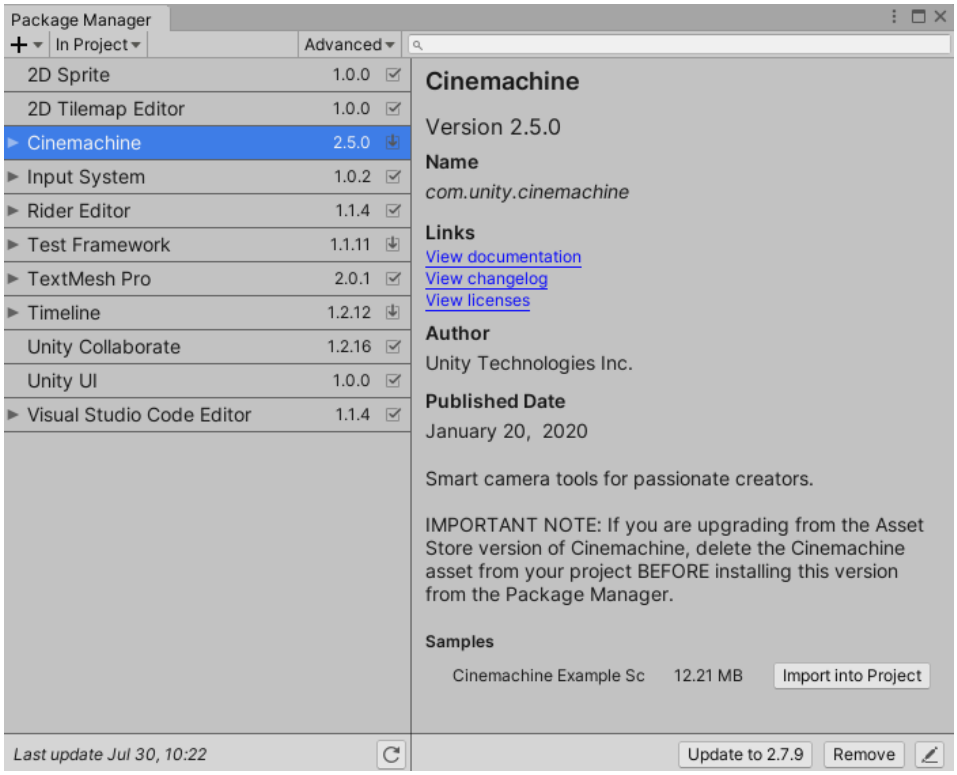
```
public class PlayerFinishLevel : MonoBehaviour
{
    public Text textpopupLevel;
    bool inRange;

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.gameObject.tag == "Player")
        {
            inRange = true;
            textpopupLevel.text = "Press Enter to finish level!";
        }
    }

    private void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.gameObject.tag == "Player")
        {
            inRange = false;
            textpopupLevel.text = "";
        }
    }

    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Return) && inRange)
        {

SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
        }
    }
}
```

Kôd 1. *Kôd za prelazak na iduću razinu*

Izvor: Autor

4. 2. Kamera

Jedan od najvažnijih elemenata svake igre jest kamera. U navedenom projektu za praćenje kretnje igrača koristi se *plugin* pod nazivom *Cinemachine. Cinemachine* je virtualna kamera koja daje mogućnost dodavanja funkcionalnosti kameri koju već koristite ili kreiranja nove kamere s raznim funkcionalnostima. Funkcionalnosti kamere koju koristimo u projektu su: „*Dead zone*“ i „*X/Y Damping*“. *Dead zone* glavnom fokusu kamere (igraču) daje mali prostor u kojemu se može kretati bez da se kamera pomiče i „*X/Y Damping*“ omogućava glatko kretanje kamere na sceni. Kako bi dodali *Cinemachine* u projekt, samo je potrebno otvoriti *Unity Package Manager* (u gornjem meniju *Window* > *Package Manager*) te ga pronaći i pritisnuti *import* tipku [8].

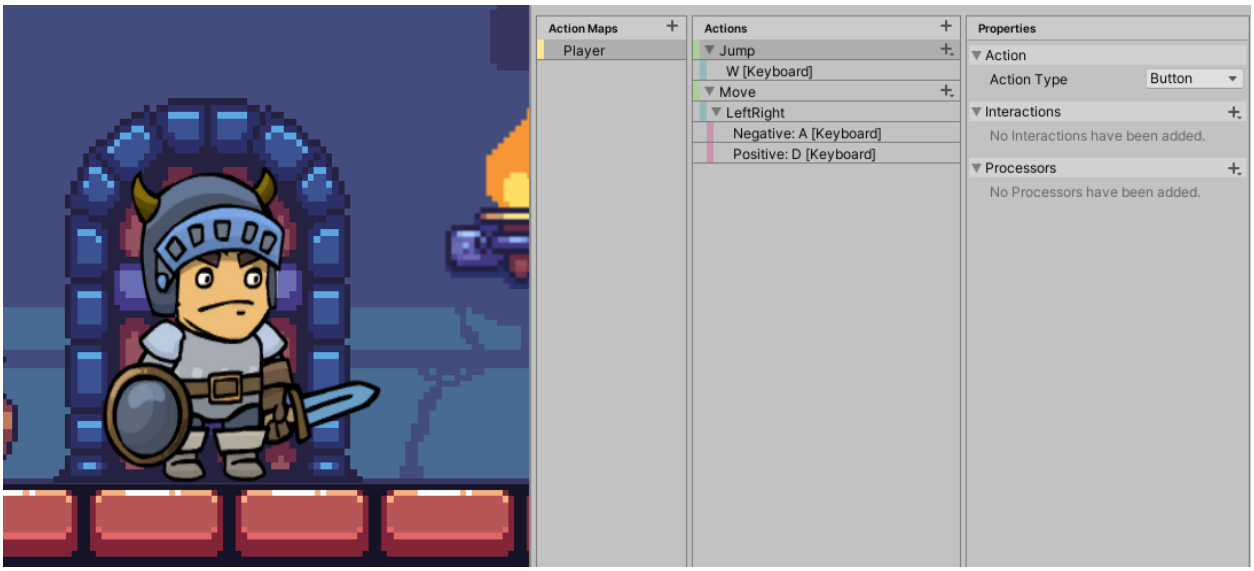


Slika 3. *Package Manager (Cinemachine)*

Izvor: Autor

5. 3. Igrač

U ovoj videoigri igrač upravlja glavnim likom koji je prikazan kao vitez. Za kretanje igrača koristi se *plugin* pod nazivom *Input System* koji se također dodaje u projekt preko *Package Managera*. Jedan od razloga zašto se u ovom radu koristi *Input System* kompatibilnost je igračevih kontrola na više platformi kao što su konzole i mobilni uređaji, koji omogućuju lakšu konverziju igre bez prevelikih promjena u kodu i postavkama *plugina*. Kako bi se igraču dodale akcije, kao što su kretanje i skakanje, kreira se *Input Actions* [9].



Slika 4. *Igrač i Input Actions*

Izvor: Autor

Input Actions dijele se na tri dijela: *Action Maps, Actions* i *Properties.*

- *Action Maps* određuje tko će koristiti kontrole, u ovom slučaju je to igrač (*Player*)
- *Actions* određuju što se želi učiniti s igračem (kretati se ili skakati)
- *Properties* pokazuje koja je tipka odabrana za aktivaciju određene akcije te ostala svojstva i postavke

Nakon toga generira se C# klasa (*Controls*) u kojoj se nalaze postavke za kontrole.

Također igraču se dodaje komponenta *GatherInput* koja prikuplja informacije iz generirane C# klase koja je prethodno napravljena, a na temelju nje kreira se *PlayerMovement* skripta u kojoj se koriste sve prethodno skupljene informacije za kretanje igrača.

```
public class PlayerMovement : MonoBehaviour
{
    public float speed;
    public float jumpForce;

    private GatherInput gi;
    private Rigidbody2D rb;
    private Animator anim;

    private int direction = 1;

    public float rayLength;
    public LayerMask groundLayer;
    public Transform leftPoint;
    public Transform rightPoint;
    private bool grounded = true;

    private bool knockBack = false;
    public bool hasControl = true;

    void Start()
    {
        gi = GetComponent<GatherInput>();
        rb = GetComponent<Rigidbody2D>();
        anim = GetComponent<Animator>();
    }

    void Update()
    {
        SetAnimatorValues();
    }

    private void FixedUpdate()
    {
        CheckStatus();
        if (knockBack || hasControl == false)
            return;
        Move();
        JumpPlayer();
    }

    private void Move()
    {
        Flip();
```

```
        rb.velocity = new Vector2(speed * gi.value, rb.velocity.y);
    }

    private void JumpPlayer()
    {
        if (gi.jumpInput)
        {
            if (grounded)
            {
                rb.velocity = new Vector2(gi.value * speed,
jumpForce);
            }
        }
        gi.jumpInput = false;
    }

    private void CheckStatus()
    {
        RaycastHit2D leftCheckHit =
Physics2D.Raycast(leftPoint.position, Vector2.down, rayLength,
groundLayer);
        RaycastHit2D rightCheckHit =
Physics2D.Raycast(rightPoint.position, Vector2.down, rayLength,
groundLayer);
        if (leftCheckHit || rightCheckHit)
        {
            grounded = true;
        }
        else
        {
            grounded = false;
        }
        SeeRays(leftCheckHit, rightCheckHit);
    }

    private void SeeRays(RaycastHit2D leftCheckHit, RaycastHit2D
rightCheckHit)
    {
        Color color1 = leftCheckHit ? Color.red : Color.green;
        Color color2 = rightCheckHit ? Color.red : Color.green;
        Debug.DrawRay(leftPoint.position, Vector2.down * rayLength,
color1);
        Debug.DrawRay(rightPoint.position, Vector2.down * rayLength,
color2);
    }

    private void Flip()
    {
        if(gi.value * direction < 0)
        {
            transform.localScale = new Vector3(-
transform.localScale.x, 0.49f, 1);
```

```
            direction *= -1;
        }
    }

    private void SetAnimatorValues()
    {
        anim.SetFloat("Speed", Mathf.Abs(rb.velocity.x));
    }

    public IEnumerator KnockBack(float forceX, float forceY, float
duration, Transform otherObject)
    {
        int knockBackDirection;
        if (transform.position.x < otherObject.position.x)
            knockBackDirection = -1;
        else
            knockBackDirection = 1;

        knockBack = true;
        rb.velocity = Vector2.zero;
        Vector2 theForce = new Vector2(knockBackDirection * forceX,
forceY);
        rb.AddForce(theForce, ForceMode2D.Impulse);
        yield return new WaitForSeconds(duration);
        knockBack = false;
        rb.velocity = Vector2.zero;
    }
}
```

Kôd 2. *Kôd za kretanje igrača*

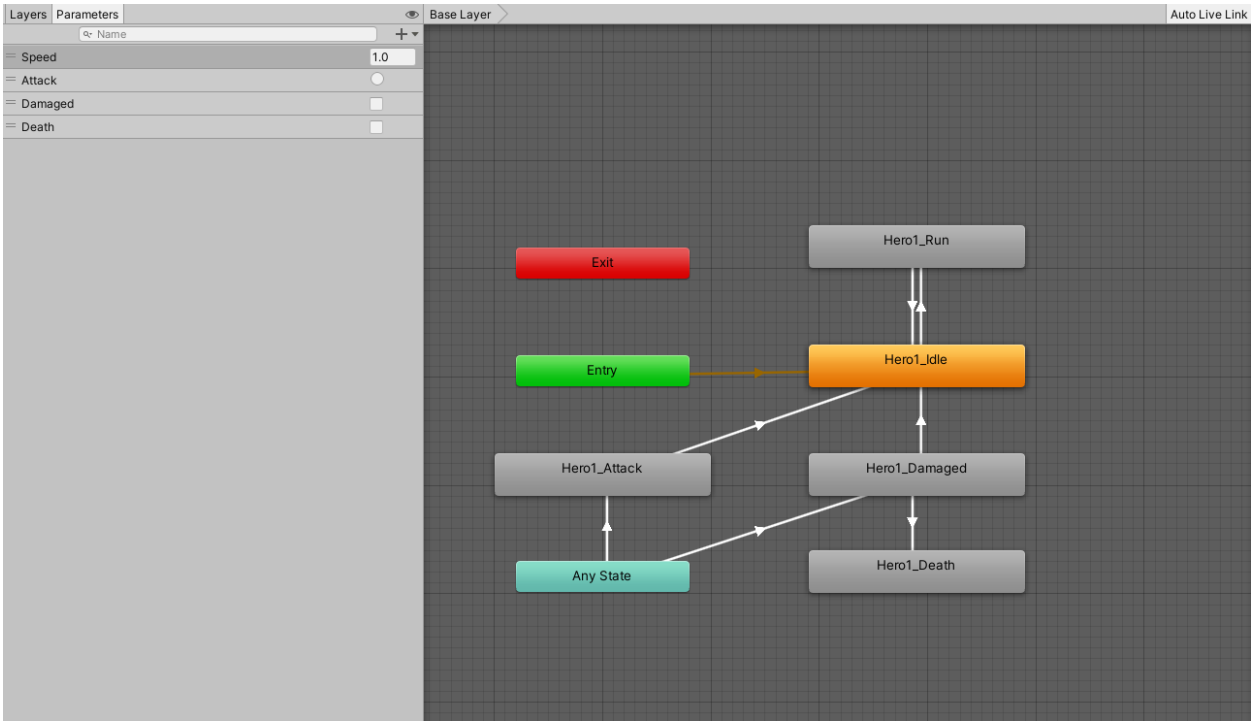
Izvor: Autor

Osim same kretnje igrača, u *PlayerMovement* skripti nalaze se neke od metoda koje su također važne za kretanje. Neke od tih metoda su *Flip, KnockBack* i *CheckStatus.*

- *Flip* metoda koristi se za okretanje igrača u smjeru njegove kretnje (ako se igrač kreće desno, vrijednost je pozitivna; ukoliko se igrač kreće lijevo, vrijednost je negativna)
- *KnockBack* metoda koristi se za odbacivanje igrača od neprijatelja ako dođe do međusobnog dodira pri kojem igrač gubi *health*
- *CheckStatus* metoda koristi se za provjeru dodiruje li igrač tlo, kako bi se onemogućila višestruka skakanja igrača i kretnje po zraku

4.3.1. Animacije

Animacije se uređuju putem Animatora, moguće ih je kreirati na više načina: stvoriti ih u programu, uvesti gotove animacije iz drugih programa ili kreirati ih pomoću skripti. Animacije za ovu igru kreirane su u Animatoru i sastoje se od nekoliko povezanih slika koje stvaraju dojam pokreta. Te slike razlikuju se u malim detaljima kao što su pokret ruku, nogu ili ostalih dijelova tijela. U Animatoru se nalaze animacije kao što su *Idle*, *Attack*, *Run*, *Damaged* i *Death*. Sve animacije prikazane su u pravokutnicima koji su povezani strelicama koje predstavljaju tranzicije iz jedne animacije u drugu te njihove parametre. Parametrima se postavljaju uvjeti kao što su *exit time* (tranzicija u određeno vrijeme) te *transition duration* (trajanje tranzicije).



Slika 5. *Animator i parametri*

Izvor: Autor

4.3.2. *Health* i *shield*

U programskom kodu za *health* i *shield* deklarirane su varijable s njihovim vrijednostima. Varijable se sastoje od maksimalne vrijednosti (*maxHealth* i *maxShield*) te trenutne vrijednosti koju igrač ima (*currentHealth* i *currentShield*). Ukoliko igrač dođe u doticaj s neprijateljem, igrač prvo gubi vrijednost *shielda* te nakon toga *healtha*. Ako igrač ostane bez *healtha* i *shielda*, dolazi do njegove smrti te se igra vraća na početak.

Metode *HandleShieldDamage* i *HandleHealthDamage* napravljene su kako bi pravilno regulirali *damage* koji igrač prima. *Damage* koji igrač prima prvo se primjenjuje na *shield*. Ako je *damage* koji igrač prima veći od vrijednosti *shielda*, onda se računa razlika koja se pohranjuje u *surplus* varijablu i razlika koja se dobije oduzima se od *healtha*.

DamagePreventation metoda služi kako bi onemogućila igraču da ga neprijatelj ili zamka neprestano ozljeđuje. Tom metodom igrač dobiva zaštitu od ozljeđivanja na određeni, kratki period.

```
public class PlayerHealth : MonoBehaviour
{
    public int maxHealth = 100;
    public int currentHealth;

    public int maxShield = 50;
    public int currentShield;

    public int surplus;

    public bool canTakeDamage = true;

    public HealthBar healthBar;
    public ShieldBar shieldBar;
    private Animator anim;
    private PlayerMovement playerMove;

    void Start()
    {
        currentHealth = maxHealth;
        healthBar.SetMaxHealth(maxHealth);

        currentShield = maxShield;
        shieldBar.SetMaxShield(maxShield);
```

```
        playerMove = GetComponentInParent<PlayerMovement>();
        anim = GetComponentInParent<Animator>();
    }

    public void TakeDamage(int damage)
    {
        if (canTakeDamage)
        {
            if (currentShield > 0 && currentHealth > 0)
                HandleShieldDamage(damage);
            else
                HandleHealthDamage(damage);

            StartCoroutine(DamagePrevention());
        }
    }

    public void HandleShieldDamage(int damage)
    {
        currentShield -= damage;
        anim.SetBool("Damaged", true);
        shieldBar.SetShield(currentShield);

        if (currentShield > 0 && currentShield < damage &&
currentHealth == maxHealth)
        {
            surplus = damage - currentShield;
        }
        if (currentShield <= 0)
        {
            currentShield = 0;
            currentHealth -= surplus;
            surplus = 0;
            healthBar.SetHealth(currentHealth);
        }
    }

    public void HandleHealthDamage(int damage)
    {
        currentHealth -= damage;
        anim.SetBool("Damaged", true);
        playerMove.hasControl = false;

        healthBar.SetHealth(currentHealth);

        if (currentHealth <= 0)
        {
            GetComponent<PolygonCollider2D>().enabled = false;
            GetComponentInParent<GatherInput>().DisableControls();
        }
    }
```

```
    private IEnumerator DamagePrevention()
    {
        canTakeDamage = false;
        yield return new WaitForSeconds(0.15f);
        if (currentHealth > 0)
        {
            canTakeDamage = true;
            anim.SetBool("Damaged", false);
            playerMove.hasControl = true;
        }
        else
        {
            anim.SetBool("Death", true);
        }
    }
}
```

Kôd 3. *Kôd za upravljanje healthom i shieldom*

Izvor: Autor

4.3.3. Borba

Za borbu igrač koristi svoj mač. Potrebno je približiti se neprijatelju dovoljno blizu da mu bude u dometu i da ga može napasti, ali treba paziti da se ne približi previše jer će ga u protivnom neprijatelj ozlijediti.

Kako bi se onemogućilo igračevo uzastopno napadanje, u *Update* metodi nalazi se *if* uvjet koji prati trenutnu vrijednost vremena, ako je to vrijeme veće ili jednako vrijednosti *NextAttackTime* varijable, onda se može napadati. Ukoliko se odluči napasti, vrijednost *NextAttackTime* varijable mijenja se. *NextAttackTime* varijabli dodaje se pola sekunde (trenutnom vremenu dodaje se jedna sekunda podijeljena s vrijednosti *AttackRate varijable,* koja iznosi dvije sekunde) te se tako dobije iduće vrijeme u kojem se može ponovno napadati.

Metoda *OnDrawGizmosSelected* služi za vizualan prikaz sfere u *editoru* i ona prikazuje domet igračevog mača prilikom napadanja neprijatelja.

```
public class PlayerCombat : MonoBehaviour
{
    public Animator animator;
    public Transform AttackPoint;
    public LayerMask EnemyLayers;

    public int AttackDamage = 40;
    public float AttackRange = 0.5f;
    public float AttackRate = 2;
    float NextAttackTIme = 0;

    void Update()
    {
        if(Time.time >=  NextAttackTIme){
            if (Input.GetKeyDown(KeyCode.K))
            {
                Attack();
                NextAttackTIme = Time.time + 1 / AttackRate;
            }
        }
    }

    void Attack()
    {
        animator.SetTrigger("Attack");
```

```
        Collider2D[] hitEnemies =
Physics2D.OverlapCircleAll(AttackPoint.position, AttackRange,
EnemyLayers);

        foreach (Collider2D enemy in hitEnemies)
        {
            enemy.GetComponent<EnemyCobra>().TakeDamage(AttackDamage);
        }
    }

    void OnDrawGizmosSelected()
    {
        if (AttackPoint == null)
            return;

        Gizmos.DrawWireSphere(AttackPoint.position, AttackRange);
    }
}
```

Kôd 4. *Kôd za borbu igrača*

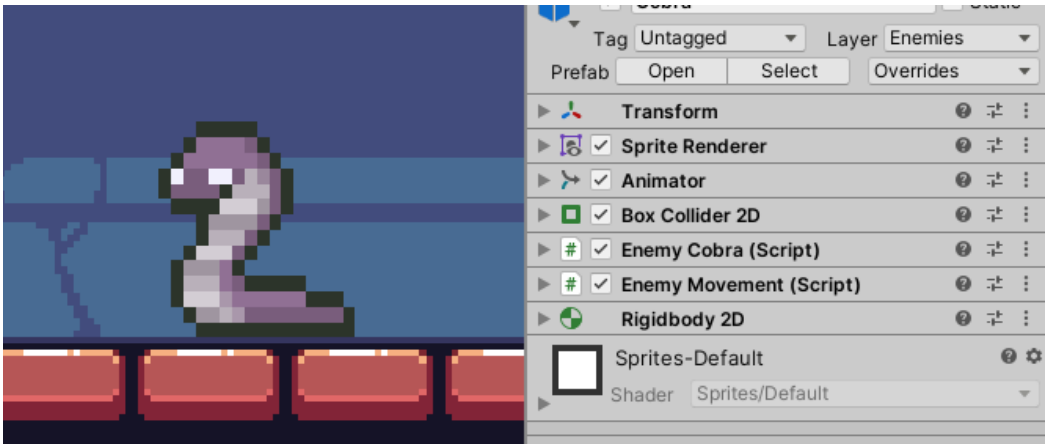
Izvor: Autor

4.4. Neprijatelji i prepreke

U ovoj igri neprijatelji su postavljeni na raznim lokacijama unutar razine. Na svakoj razini mogu se susresti glavni neprijatelji (zmije) i prepreke (šiljci) koje je potrebno preskočiti. Zmije se po razini kreću unutar određenog raspona, dok su šiljci statično pozicionirani.

4.4.1. Zmija

Zmije su postavljene na određenim lokacijama na svakoj razini. Kao i igrač, zmije koriste komponente poput *Rigidbody2D, Box Collider2D* te skripte za kretanje, borbu i *statse*.



Slika 6. *Zmija i njezine komponente*

Izvor: Autor

U sljedećoj skripti nalazi se kod za kretanje zmije. Zmija se kreće određenom brzinom u dva smjera (lijevo i desno). Kako bi zmija znala u kojem se rasponu može kretati, dodaju se dva *collidera* koji funkcioniraju tako da kada zmija dotakne *collider*, njen smjer kretnje okreće se u suprotnom smjeru.

```
public class EnemyMovement : MonoBehaviour
{
    public float Speed;
    public bool MoveRight;

    void Update()
    {
        if (MoveRight)
        {
            transform.Translate(2 * Time.deltaTime * Speed, 0, 0);
            transform.localScale = new Vector2(-4f, 5f);
        }
        else
        {
            transform.Translate(-2 * Time.deltaTime * Speed, 0, 0);
            transform.localScale = new Vector2(4f, 5f);
        }
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.gameObject.CompareTag("Wall"))
        {
            if (MoveRight)
            {
                MoveRight = false;
            }
            else
            {
                MoveRight = true;
            }
        }
    }
}
```

Kôd 5. *Kôd za kretanje zmije*

Izvor: Autor

Kako bi igrač mogao ozlijediti neprijatelja, dodaje mu se skripta u kojoj se nalaze metode za primanje ozljeda te metoda koja se aktivira prilikom smrti neprijatelja. *Take Damage* metoda oduzima *health* neprijatelju pri svakom udaru igrača, a ako neprijatelj ostane bez *healtha,* pokreće se metoda *Die. Die* metoda onemogućuje daljnje kretanje neprijatelja, pokreće animaciju za smrt te nakon određenog vremena uklanja njegov model iz igre.

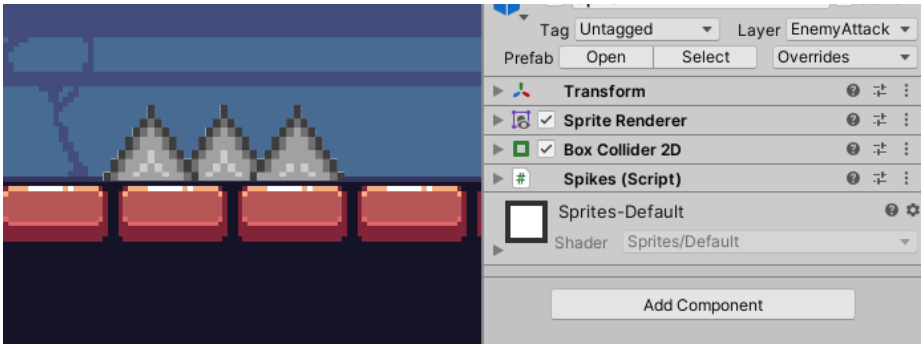
```
Public class EnemyCobra : MonoBehaviour
{
    public Animator animator;

    public int MaxHealth = 100;
    int CurrentHealt;

    void Start()
    {
        CurrentHealt = MaxHealth;
    }

    public void TakeDamage(int damage)
    {
        CurrentHealt -= damage;

        animator.SetTrigger("Hurt");

        if(CurrentHealt <= 0)
        {
            GetComponent<EnemyMovement>().enabled = false;
            Die();
        }
    }

    void Die()
    {
        Debug.Log("Enemy died!");

        animator.SetBool("IsDead", true);

        GetComponent<Collider2D>().enabled = false;
        GetComponentInChildren<CircleCollider2D>().enabled = false;
        this.enabled = false;
        GameObject.Destroy(gameObject, 2.2f);
    }
  }
```

Kôd 6. *Kôd za ozljeđivanje neprijatelja i njegovo umiranje*

Izvor: Autor

4.4.2. Šiljci

Šiljci su statične prepreke kojima je jedina svrha ozlijediti igrača ako dođe do njihove kolizije. Igrač ih samo mora preskočiti da bi ih izbjegao. Od komponenti koristi se skripta za ozljeđivanje igrača te *Box Collider2D*.



Slika 7. *Šiljci i njegove komponente*

Izvor: Autor

```
public class Spikes : MonoBehaviour
{
    public int damage;
    public float forceX;
    public float forceY;
    public float duration;

    private void OnTriggerEnter2D(Collider2D collision)
    {
        collision.GetComponent<PlayerHealth>().TakeDamage(damage);
        PlayerMovement playerMovement =
collision.GetComponentInParent<PlayerMovement>();

        StartCoroutine(playerMovement.KnockBack(forceX, forceY,
duration, transform));
    }
}
```

Kôd 7. *Kôd za šiljke*

Izvor: Autor

4.5. Novčići

Zlatni su novčići valuta koju igrač skuplja tijekom igre. Pomoću *CoinManager* skripte na ekranu mijenja se broj skupljenih novčića. Novčići su bitni za skupljanje jer se kasnije koriste u trgovini za nadogranju igračevih *statsa* (*health, shield* i *damage*).

Slika 8. *Novčići*

Izvor: Autor

4.6. Trgovina i nadogradnje

Prikupljenim novčićima igrač može nadograditi svoje *statse* u trgovini (eng. *shop*) na drugoj razini. Svaka će nadogradnja igrača koštati određeni broj novčića. Moguće je nadograditi *health, shield* i *damage*. Prilikom nadogradnje *healtha* ili *shielda*, igrač dobiva njihovu maksimalnu vrijednost ako je prethodno bio ozlijeđen. Trenutno stanje i stanje nakon nadogradnje prikazano je u donjem desnom kutu u igri.

Slika 9. *Trgovina za nadogradnje statsa*

Izvor: Autor

U skripti *UpgradeStats* nalaze se metode *UpgradeHealth*, *UpgradeShield* i *UpgradeDamage*, svaka od metoda koristi se za nadogradnju određenog *statsa* igrača. Svaka metoda sastoji se od *if* uvjeta koji prvo provjerava stanje novčića i ako igrač nema dovoljno novčića za nadogradnju koju je odabrao, dobit će tekstualnu obavijest na ekranu. Ako igrač ima dovoljno zlatnika za nadogradnju, povećava se vrijednost *statsa* te se oduzima vrijednost nadogradnje od trenutne vrijednosti novčića.

```
public class UpgradeStats : MonoBehaviour
{
    private static PlayerHealth playerHp;
    private PlayerHealth playerShld;
    private PlayerCombat playerDmg;

    private HealthBar hSlider;
    private ShieldBar sSlider;

    public CoinManager coin;
```

```
    public int upgradeHpPrice = 10;
    public int upgradeDmgPrice = 15;
    public int upgradeShldPrice = 20;

    public Text text;

    public void UpgradeHealth()
    {
        playerHp =
GameObject.Find("PlayerStats").GetComponent<PlayerHealth>();
        coin =
GameObject.Find("CoinManager").GetComponent<CoinManager>();
        hSlider =
GameObject.Find("HealthBar").GetComponent<HealthBar>();

        if (coin.score >= upgradeHpPrice)
        {
            playerHp.maxHealth += 25;
            playerHp.currentHealth += 25;
            playerHp.currentHealth = playerHp.maxHealth;
            coin.score -= upgradeHpPrice;

            hSlider.healthSlider.maxValue = playerHp.maxHealth;
            hSlider.healthSlider.value = playerHp.maxHealth;

            Debug.Log("Hp upgraded, current hp : " +
playerHp.maxHealth);
        }
        else
            text.text = "Not enough coins for upgrade!";
    }

    public void UpgradeShield()
    {
        playerShld =
GameObject.Find("PlayerStats").GetComponent<PlayerHealth>();
        coin =
GameObject.Find("CoinManager").GetComponent<CoinManager>();
        sSlider =
GameObject.Find("ShieldBar").GetComponent<ShieldBar>();

        if (coin.score >= upgradeShldPrice)
        {
            playerShld.maxShield += 25;
            playerShld.currentShield += 25;
            playerShld.currentShield = playerShld.maxShield;
            coin.score -= upgradeShldPrice;

            sSlider.shieldSlider.maxValue = playerShld.maxShield;
            sSlider.shieldSlider.value = playerShld.maxShield;
```

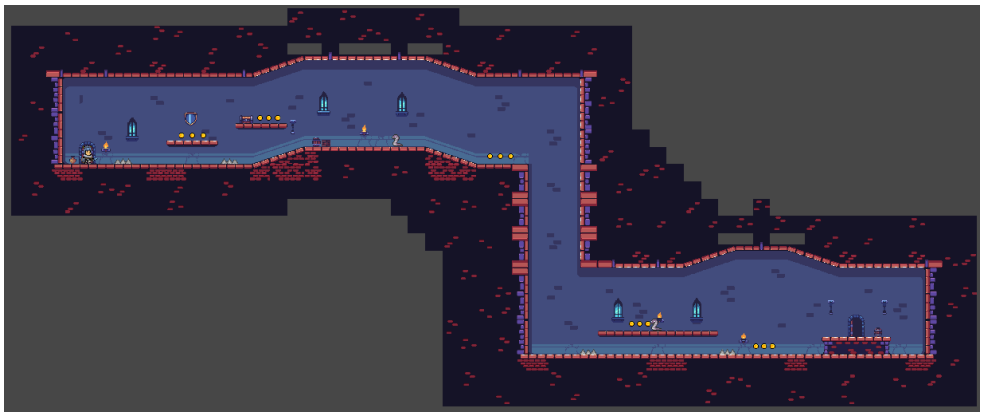
```
            Debug.Log("Shield upgraded, current max shield: " +
playerShld.maxShield);
        }
        else
            text.text = "Not enough coins for upgrade!";
    }

    public void UpgradeDamage()
    {
        playerDmg =
GameObject.Find("Hero1").GetComponent<PlayerCombat>();
        coin =
GameObject.Find("CoinManager").GetComponent<CoinManager>();

        if (coin.score >= upgradeDmgPrice)
        {
            playerDmg.AttackDamage += 20;
            coin.score -= upgradeDmgPrice;

            Debug.Log("Damage upgraded, current damage: " +
playerDmg.AttackDamage);
        }
        else
            text.text = "Not enough coins for upgrade!";
    }
}
```

Kôd 8. Kôd za nadogradnju statsa

Izvor: Autor

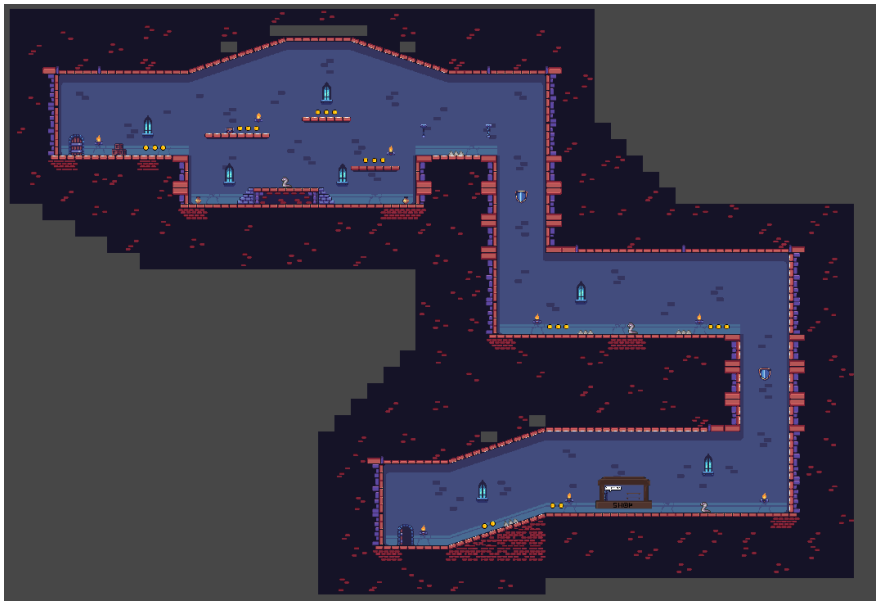
4.7. Razine

U igri postoje tri razine, svaka razina dizajnirana je kako bi testirala igračeve sposobnosti i vještine. Prva razina namijenjena je upoznavanju igrača s kontrolama, neprijateljima i preprekama koje će susretati na svakoj razini.



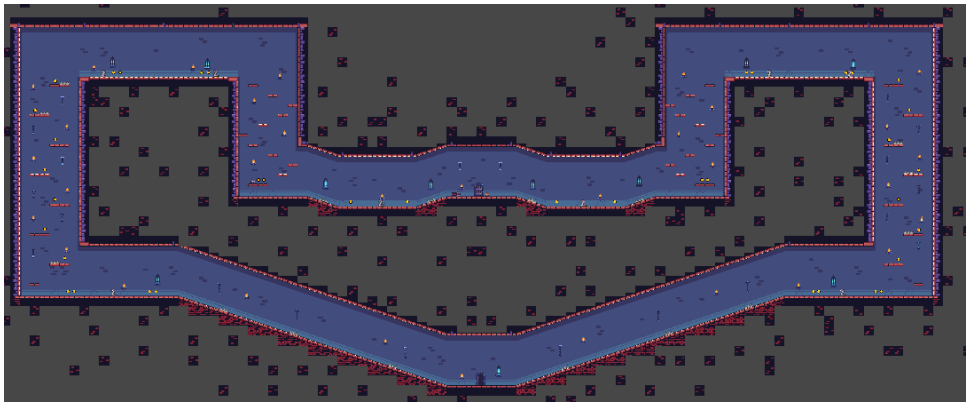
Slika 10. *Prva razina*

Izvor: Autor



Slika 11. *Druga razina*

Izvor: Autor



Slika 12. *Treća razina*

Izvor: Autor

4.8. Glavni Izbornik

Glavni izbornik se sastoji od tri gumba: *Play*, *How to play* i *Quit*. *Play* gumb pokreće igru, *How to play* vodi do uputa za igru te *Quit* kojim se gasi igra.

Slika 13. *Glavni izbornik*

Izvor: Autor

5. ZAKLJUČAK

Videoigre su jedan od najčešćih razloga zbog kojeg se ljudi zainteresiraju za programiranje te za proces izrade videoigara. Iako to nije tako lako zbog raznih segmenata izrade koje je potrebno savladati, poput dizajniranja razina, postavljanje animacija, zvuka te – najvažnije – razvijanje programskog koda potrebnog da bi igra funkcionirala, cijeli je proces zanimljiv i izazovan.

U ovom radu predstavljen je razvoj 2D arkadne igre u *Unity Engineu* uz pomoć C# programskog jezika. Cilj je rada bio naučiti proces izrade 2D igre. Svi *asseti* preuzeti su s interneta te su potom animirani na sceni. U radu su opisani procesi od dizajniranja razine do samih elemenata svake scene, kao što su: igrač, kamera, kontrole, neprijatelji, razine, trgovina i druge. Najbitniji dio bio je izrada kvalitetne igre koja se može jednostavno nadograditi i kojoj se mogu dodati nove funkcije i razine koje će ju unaprijediti.

6. LITERATURA

[1] Video Game History

https://www.history.com/topics/inventions/history-of-video-games

(27. 7. 2022.)

[2] iRacing

https://www.iracing.com/

(27. 7. 2022.)

[3] Arcade video game

https://en.wikipedia.org/wiki/Arcade_video_game

(27. 7. 2022.)

[4] Unity Game Engine Guide: How to Get Started with the Most Popular Game Engine Out There

https://www.freecodecamp.org/news/unity-game-engine-guide-how-to-get-started-with-the-most-popular-game-engine-out-there/

(27. 7. 2022.)

[5] Introduction to Visual Studio

https://www.geeksforgeeks.org/introduction-to-visual-studio/?ref=gcse

(28. 7. 2022.)

[6] What is C# used for?

https://stackify.com/what-is-c-used-for/

(28. 7. 2022.)

[7] Unity Asset Store

https://assetstore.unity.com/

(28. 7. 2022.)

[8] About Cinemachine

https://docs.unity3d.com/Packages/com.unity.cinemachine@2.1/manual/index.html

(28. 7. 2022.)

[9] Input System
https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/manual/Actions.html
(29. 7. 2022.)

POPIS SLIKA

POPIS KÔDOVA