S V E U Č I L I Š T E U S P L I T U
FAKULTET ELEKTROTEHNIKE, STROJARSTVA I BRODOGRADNJE

Računalna grafika

SEMINAR

IZRADA VIDEO IGRE U GODOT ENGINEU

Ivan Akrapović

Dominik Mišadin

Split, siječanj 2025.

SADRŽAJ

1. UVOD

Cilj ovog seminara je primijeniti teorijsko i praktično znanje iz kolegija Računalna grafika. Znanju koje uključuje osvjetljenje, teksturiranje i animaciju scene te sjenčenje, pridodat ćemo novo iskustvo izrade 2D video igre u platformerskom žanru. Iako danas postoji mnogo alata u kojima se to može izvesti, mi smo odabrali Godot Engine (ver. 4.3) u kombinaciji sa C# programskim jezikom i .NET frameworkom. Izvorni programski kod projekta dostupan je na github.com [7].

2. TEKSTURIRANJE

Za umjetnički smjer video igre odredili smo koristiti tzv. *pixel art* kako bi mogli iskoristiti već dostupne teksture u zajednici *indie* razvoja video igara. Naša scena će biti prikazana u tamnom okruženju dvorca ili tornja. Cilj igre će biti jednostavan – pronaći ključ te otključati škrinju na vrhu scene.

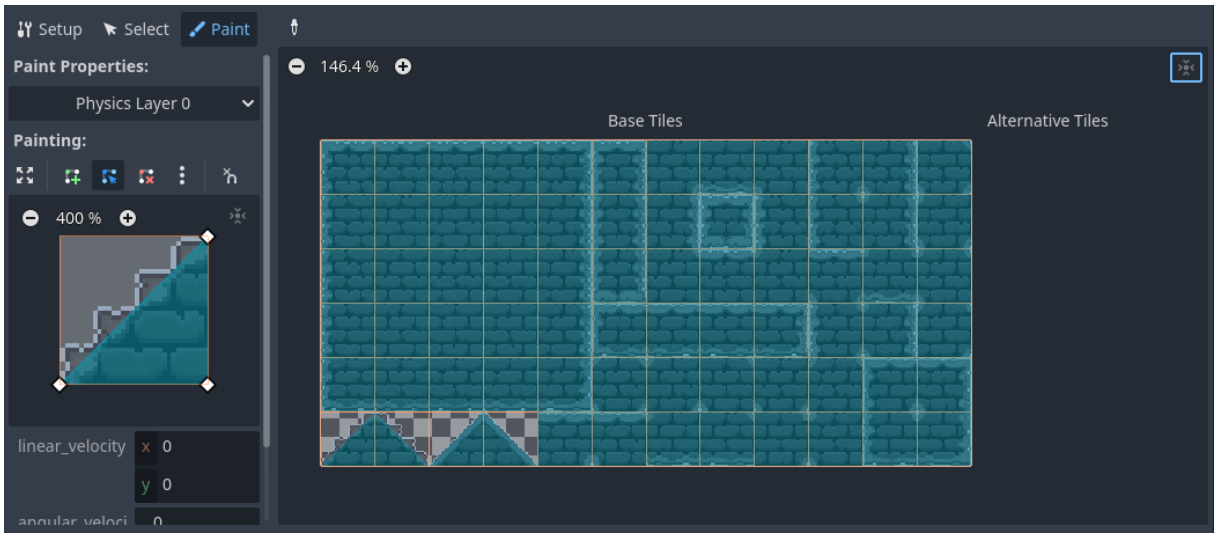
2.1. Definiranje *TileSeta* i kolizija

Jednom kad smo odabrali i preuzeli željeni *tileset* [1] s kojim ćemo graditi većinu scene, krećemo s unošenjem tekstura u Godot Engine. Na slici 2.1 vidimo odabranu teksturu koju ćemo podijeliti na *grid* s veličinom polja 32x32px, prema savjetu tvorca. Drugim riječima, kada budemo postavljali polja tekstura u scenu, koristi ćemo na sceni također *grid* s poljima 32x32px, a isti se vodi pod nazivom *tilemap*.

Slika 2.1. Prvi korak unosa teksture kroz Tileset Editor

Međutim, prije nego što pređemo na slaganje scene, unaprijed ćemo odraditi jedan korak koji će biti potreban, a to je definiranje kolizija našeg tileseta. Na slici 2.2 primijetimo da se nalazimo u sličnom Tilemap editoru, s naglaskom na uređivanje *Physics Layera*.

Naš primjer je relativno jednostavan i nije bilo nužno namještati zadane blokove, što se vidi poluprozirnom plavom bojom. Kasnije u projektu se može definirati koji čvorovi imaju interakciju i koliziju s tim površinama.



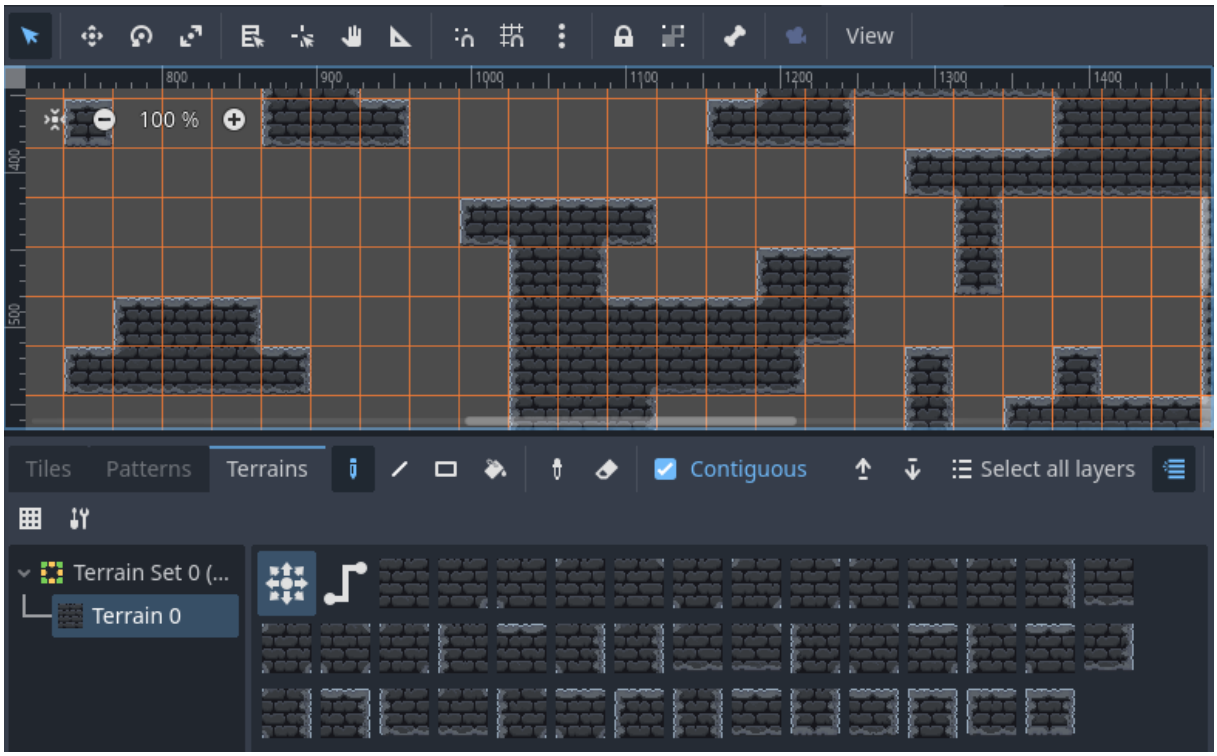
Slika 2.2. Definiranje kolizija za tileset teksturu

Postoji još jedan neobavezni dodatni korak s kojim si možemo omogućiti rad s moćnim alatom u Godot Engineu u idućem koraku teksturiranja. Na slici 2.3 zelenom bojom označene su putanje koje su povezane, što znači da će biti moguće s univerzalnim alatom crtati proizvoljno po zaslonu i teksture će se automatski odabirati. Više o tome kasnije.

Slika 2.3. Definiranje Terrain seta za tileset teksturu

2.2. Dizajniranje i stvaranje levela

Na red je stigao korak koji je usredotočen na kreativni proces izgradnje scene. Na scenu dodamo novi čvor **TileMapLayer** i definiramo *grid* s poljima veličine 32x32px. Pronađemo *Terrain* podizbornik te odaberemo željenu teksturu iz *tileseta*. Mišem zatim odabiremo polja gdje će se odabrani blok teksture pojaviti, kao na slici 2.4.



Slika 2.4. Korištenje TileMape za dizajn scene

Štoviše, možemo koristiti *connect mode* ili *path mode* što je prikazano s prve dvije ikone u našem *Terrain* izborniku. Iznimno su korisni alati, posebice za grubo skiciranje *tilemape* jer se automatski odabire tekstura koja se nadovezuje na prethodno postavljenu ili cijelu susjednu grupu tekstura. Ovo je bio razlog zašto želimo imati *terrain set* definiran u našem *tilesetu*.

Nakon zadnja tri koraka, može se zaključiti da dizajn *levela* i teksturiranje u ovom slučaju je spojeno u zajednički proces. Može se započeti teksturirati ako želimo to, no još bolja ideja je pričekati s detaljima za kasnije, nakon što odredimo kako će se naš lik moći kretati. To će doći do izražaja kada shvatimo koliko visoko i daleko lik može skočiti.

3. UPRAVLJAČKE KONTROLE

Već smo ustanovili da se radi o video igri u platformerskom žanru, za što su nam potrebne platforme za skakanje i lik s kojim ćemo skakati po *levelu*. Godot Engine nudi mnoštvo raznih elemenata koji su raspodijeljeni po scenama i čvorovima (engl. *node*). Možemo stvoriti novi čvor u izvornoj sceni (engl. *root scene*) s imenom **PlayerController** koji nasljeđuje Godot klasu **CharecterBody2D** u našem C# programu. Takvo nasljeđivanje nam omogućuje izravno korištenje metoda kao što su *_Ready()* i *_PysicsProcess(double delta)*, gdje se prva izvrši jednom pri ulasku čvora u scenu, a druga se izvrši u svakom *frameu* što se generira. Primijetimo kako se u potonjoj nalazi i parametar *delta* koji ukazuje koliko vremena je prošlo od prethodno generiranog *framea*. Upravo u *_PysicsProcess* metodi ćemo obrađivati kretnje našeg lika unutar scene, što je vidljivo u isječku koda 3.1.

Isječak koda 3.1 *Jednostavni način za promjenu brzine i smjera lika s kojim upravljamo*

```
private Vector2 newVelocity = new Vector2();

public override void _PhysicsProcess(double delta)
{
    newVelocity = Velocity;

    if (!IsOnFloor())
        newVelocity += GetGravity() * (float)delta;

    HandleJump();
    HandleMovement();
    HandleDash();

    Velocity = newVelocity;
    MoveAndSlide();
}
```

Važno je imati na umu da bi brzina generiranja *frameova* (engl. *frame rate*) trebala biti konzistentna kroz vrijeme korištenja programa, odnosno, *delta* bi trebala biti uvijek ista. Korisno je imati na umu da zbog ovog ponašanja može doći do neželjenih nuspojava i promjena brzine kretnji. Kako se ne radi o igri s više igrača, za naše potrebe ovo će biti više nego dovoljno.

Razmotrimo prvo metodu *HandleMovement* iz isječka koda 3.2 koja je zadužena za promjene smjera lijevo i desno. Prije svega, dohvatimo trenutačno pritisnute kontrole u obliku vektora. Zanimaju nas ulazne vrijednosti s prefiksom „ui_“ koje su zadane vrijednosti u Godotu, kao

pseudonimi koji se mogu dodavati i mijenjati u postavkama projekta. Nakon provjere ulaznih vrijednosti uzimamo u obzir dva slučaja.

Ako postoji nekakva promjena na ulaznim kontrolama, primijenimo novu brzinu uz pomoć definiranih konstanti za brzinu kretnje **Speed** te **Acceleration**. Kroz metodu **Mathf.Lerp** dobijemo interpolaciju između novog i starog stanja kretnje, kako bi prijelaz bio glađi. Kako igrač može u bilo kojem trenutku pustiti kontrole, želimo pokriti još slučaj da se lik nastavi lagano kliziti po završetku kretnje i tu dolazi u obzir konstanta trenja **Friction** sve dok se ne zaustavi, odnosno brzina postane nula.

Isječak koda 3.2 *Primjer dohvaćanja smjera kretanja s ulaza*

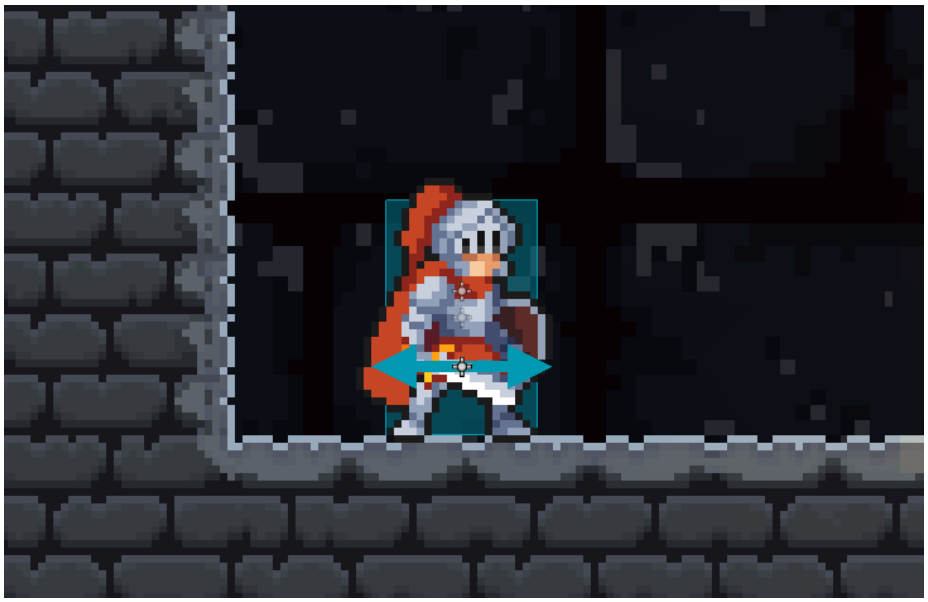
```
public const float Speed = 200.0f;
private float Friction = 0.1f;
private float Acceleration = .25f;

private void HandleMovement ()
{
    Vector2 direction = Input.GetVector("ui_left","ui_right","ui_up","ui_down");

    if (direction != Vector2.Zero)
    {
        velocity.X = Mathf.Lerp(velocity.X, direction.X * Speed, Acceleration);
    }
    else
    {
        velocity.X = Mathf.Lerp(velocity.X, 0, Friction);
    }
}
```

Prije nego što analiziramo *HandleJump* metodu, koja je naravno, vezana za mogućnost skakanja lika, želimo razumjeti kako naš lik izgleda. Kada pogledamo sliku 3.1 vidimo našeg lika viteza u oklopu i još tri definirane stavke. Prva stavka je pravokutnik u pozadini koji definira granice sudara (engl. *collision*) koja služi za ograničavanje kretnji od zida do zida te sprječava propadanje kroz pod ili platforme.

Druge dvije stavke su prikazane s usmjerenim strelicama prema lijevo i desno. Radi se o čvorovima **RayCast2D** za detekciju sudara na vrhovima strelica, odnosno, nama će služiti za mehaniku kretnje odbijanja o zid (engl. *wall jumping*).



Slika 3.1 Prikaz lika kojim upravljamo i njegov okvir za sudare

Ako je naš lik u blizini zida možemo odrediti s koje strane mu prilazi te omogućiti da još jednom skoči, ali uzeti u obzir da ne smije beskonačno skakati bez da između skokova ni jednom nije dotaknuo pod. Upravo to smo izveli s isječkom koda 3.3, mjenjajući iznos trenutačnoj brzini.

Isječak koda 3.3 *Procesiranje pritiska Space tipke za skok lika*

```
private void HandleJump()
{
    if (!Input.IsActionJustPressed("ui_accept"))
         return;
    if (GetNode<RayCast2D>("LeftSide").IsColliding() && !IsWallJumping) {
        velocity.X = -JumpVelocity;
        velocity.Y = JumpVelocity;
        IsWallJumping = true;
    }
    else if (GetNode<RayCast2D>("RightSide").IsColliding() && !IsWallJumping) {
        velocity.X = JumpVelocity;
        velocity.Y = JumpVelocity;
        IsWallJumping = true;
    }
    else if (IsOnFloor()) {
        velocity.Y = JumpVelocity;
        IsWallJumping = false;
    }
}
```

4. ANIMACIJA

U dosadašnjem radu, naša scena je bila statična. Kako bi unijeli živosti u okruženje u kojem se igrač nalazi, uvest ćemo animacije likova i objekata. S obzirom na to da radimo s 2D video igrom i *pixel art* likovima, koristi ćemo metodu s animiranim *spriteovima*. Krenimo s našim likom s kojim igrač upravlja.

4.1. Animacija lika viteza

Pronalaskom željene teksture koje ćemo koristiti za našeg glavnog lika možemo proučiti tzv. *sprite sheet* koji nam nudi nekoliko opcija za likove u raznim radnjama. Počevši od stajanja u mjestu i trčanja, do skakanja i ozljeđivanja. Na slici 4.1 imaju tri primjera, gdje svaki ima različiti broj slika.

Slika 4.1. Sprite sheet viteza dok stoji u mjestu (na vrhu), trči (u sredini) i skače (na dnu) [2]

Kako bi se mogli prilagoditi tim slučajevima, u grafičkom sučelju Godot Enginea može se promijeniti brzina izvođenja animacije (engl. *frame rate* ili *FPS – frames per second*). Jasno je iz svega navedenog da možemo definirati više različitih animacija za jedan **AnimatedSprite** čvor, no kako određujemo kada se koji aktivira? U pomoć nam dolaze skripte i programska rješenja dinamičnih animacija. U prethodnom poglavlju smo razradili kako se izvode kontrole koje igrač šalje s tipkovnice, a zasad ćemo taj dio ignorirati u primjerima isječaka koda za promjenu animacije. Započnimo s isječkom koda 4.1, s kojim želimo omogućiti postavljanje

animacije za različite kretnje s različitim prioritetima. Primjerice, ako je vitez u skoku, ali igrač pritišće tipku za kretnju lijevo, želimo da se animacija za skok izvodi dok je vitez u zraku umjesto trčanja na lijevo. Time animacija skoka ima viši prioritet nad animacijom trčanja.

Isječak koda 4.1. *Metoda za postavljanje animacije po prioritetu (PlayerController.cs)*

```
private void SetAnimation(string animation, int priority = 0)
{
    if (priority >= CurrentAnimationPriority)
    {
        CurrentAnimation = animation;
        CurrentAnimationPriority = priority;
        GetNode<AnimatedSprite2D>("PlayerSprite").Play(animation);
    }
}
```

Prvi primjer dinamičnog mijenjanja animacije viteza prikazan je pojednostavljenim isječkom koda 4.2. Zasad je dovoljna već zadana (engl. *default*) vrijednost parametra za prioritet izvođenja animacije.

Isječak koda 4.2. *Pokretanje animacije skoka na pritisak Space tipke*

```
private void HandleJump()
{
    if (!Input.IsActionJustPressed("ui_accept"))
        return;

    /* ... */
    SetAnimation("Jump");
}
```

Slijedeći primjer iz isječka koda 4.3 prikazuje pokretanje animacije za trčanje i stajanje u mjestu. Za razliku od dosadašnjih primjera, ovdje je važno uzeti u obzir da se radi o potencijalnoj promjeni smjera kretanja. Umjesto kreiranja zasebnog *sprite sheeta*, za potrebe 2D video igre, dovoljno je zrcaliti *sprite* viteza, koju god animaciju trenutno ima pokrenutu.

Dodatno se uvjerimo da vitez nije u zraku s naslijeđenom metodom iz Godota **IsOnFloor**. Ovim primjerima vidimo koliko čvrsto su isprepletene kontrole lika i animacije koje se pritom moraju izvesti.

Isječak koda 4.3. *Pokretanje animacije trčanja lijevo ili desno, odnosno stajanja u mjestu*

```
private void HandleMovement()
{
    Vector2 direction = Input.GetVector("ui_left", "ui_right", ...)
    if (direction != Vector2.Zero)
    {
        GetNode<AnimatedSprite2D>("PlayerSprite").FlipH = direction.X < 0;

        if (IsOnFloor())
            SetAnimation("Run");

        /* ... */
    }
    else if (IsOnFloor())
        SetAnimation("Idle");
        velocity.X = Mathf.Lerp(velocity.X, 0, Friction);
    }
}
```

Za kraj predstavit ćemo još dvije jednostavne animacije, no za razliku od prethodnih, one se ne pokreću prilikom kontroliranja viteza, već ovisno o događanjima nad vitezom. U trenutku gubitka jednog od tri života ili svih života, pokrenut će se jedna od animacija za ozljedu ili smrt. Ponajviše zbog ovog slučaja, potreban nam je prioritet prilikom postavljanja animacije, prikazano isječkom koda 4.4.

Isječak koda 4.4*. Pokretanje animacije prilikom ozljeđivanja ili gubitka svih života*

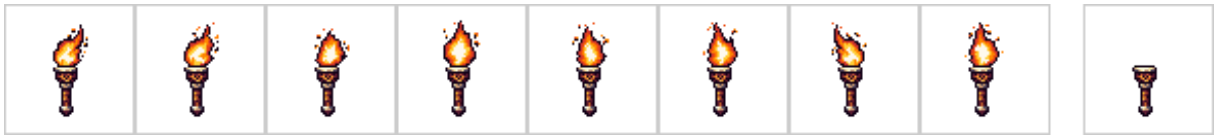
```
private void OnHealthChanged(int newHealth)
{
    if (newHealth > 0)
        SetAnimation("Hurt", 4);
    else
        SetAnimation("Death", 5);
}
```

5. OSVJETLJENJE

Već smo spomenuli da okruženje u kojem se nalazi vitez je toranj dvorca kroz koji se penje. Kako bi naglasili doživljaj napuštenog dvorca, možemo postaviti većinu scene u tamu. Time će se znatno smanjiti vidljivost, ali u okruženje će se dobro uklopiti baklje na zidovima koje će u ograničenom radijusu emitirati svjetlost.

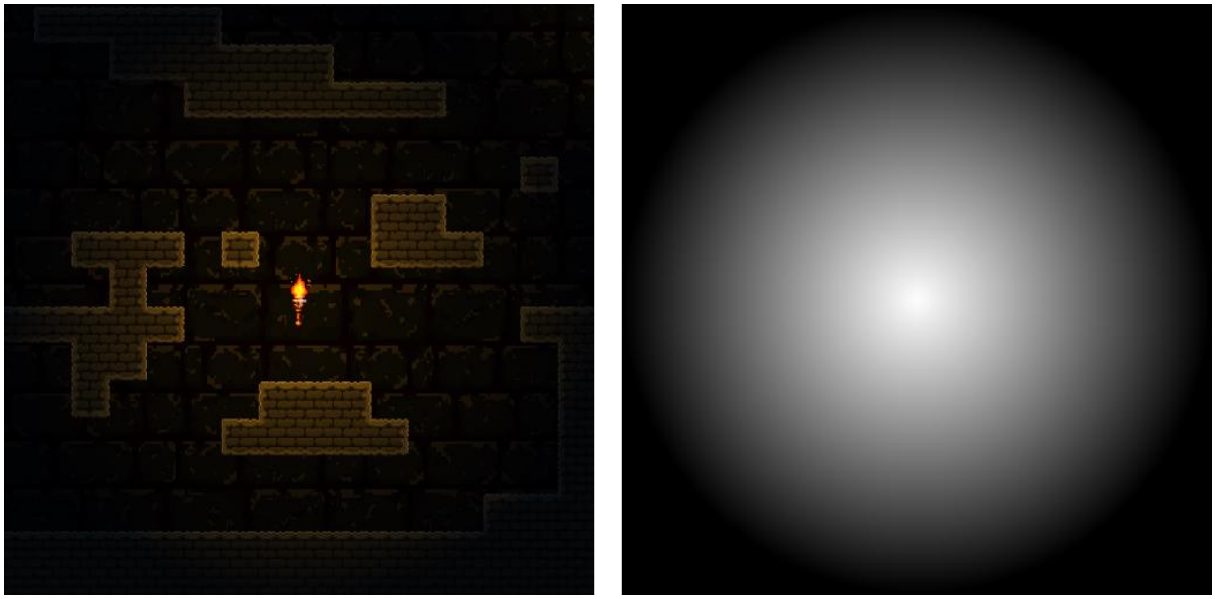
5.1. Izvori svjetla

Scenu ćemo osvijetliti s dva tipa izvora svjetlosti, s ciljem da igraču omogućimo lakše navigiranje kroz izazovne uvijete. Prvi tip sa slike 5.1 jest već spomenuta baklja, za koju ćemo pripremiti dva stanja.



Slika 5.1. Dva stanja baklje – sprite sheet dok gori (lijevo) i ugašena (desno) [3]

Dodavanjem novog čvora tipa **PointLight2D** započinjemo s izvorom svjetla u Godot Engineu koji se iz jedne točke širi u svim smjerovima. Ipak, potrebno je imat teksturu s kojom će biti definiran intenzitet svjetla, prikazano primjerom na slici 5.2.



Slika 5.2. Baklja kao izvor svjetlosti (lijevo) i tekstura s gradijentom za oblik svjetla (desno)

Primjenjivanjem teksture s crno-bijelom *bitmapom*, Godot Engine mapira normalizirane razine svjetlosti piksela s intenzitetom izvora svjetlosti na istom mjestu od točke izvora svjetlosti. U našem primjeru odabrana je topla narančasta boja za izvor svjetlosti, s prilagođenim intenzitetom.

Kako se radi o plamenu baklje, koji može titrati nasumičnim intenzitetom, obogatit ćemo doživljaj s implementacijom istog. Prilikom dodavanja nasumične vrijednosti, kroz kod 5.1 možemo omogućiti uređivanje raspona intenziteta i brzine promjene u samom Godot Engine IDE, prikazano u prve tri linije koda. Zatim kalkulacije provodimo u **Process** metodi u svakom frameu izvođenja.

Isječak koda 5.1. *Nasumična promjena intenziteta svjetlosti (Torch.cs)*

```
[Export(PropertyHint.Range, "0.0,1.0")] public float IntensityMin = 0.7f;
[Export(PropertyHint.Range, "0.0,1.0")] public float IntensityMax = 1.0f;
[Export(PropertyHint.Range, "0.0,0.5")] public float FlickerSpeed = 0.1f;

private float timer = 0.0f;
private Random random = new Random();

public override void _Process(double delta)
{
    timer += (float)delta;

    if (timer < FlickerSpeed)
        return;

    timer = 0.0f;
    float intensityRange = IntensityMax - IntensityMin;
    float randomIntensity = random.NextDouble() * intensityRange + IntensityMin;
    Energy = randomIntensity;
}
```

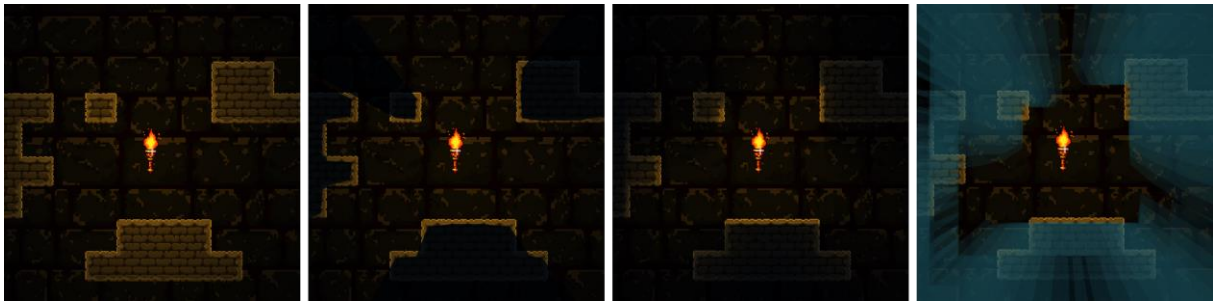
5.2. Statične sjene

Ako se vratimo na sliku 5.2, primijetimo da naše točkasto svjetlo širi se u svim smjerovima, odnosno prema definiranoj teksturi. Međutim, intuitivno bi očekivali da se na preprekama zaustavi te stvori sjene iza neprozirnih objekata. Takav slučaj nije nužno očit u 2D sceni, točnije, u obzir se moraju uzeti položaji slojeva na sceni. Odnosno, eksplicitno moramo odrediti koji izvori svjetla utječu na koje neprozirne objekte. Za početak uključimo sjene (engl. *shadows*) na točkastim izvorima svjetla, no brzo se vidi da se ništa nije promijenilo.

Problemu sjena ćemo pristupiti na nekoliko načina, ovisno o vrsti objekta s kojim želimo blokirati svjetlost. Prvo namještamo *tileset* još jednom, ovaj put označavamo površine tekstura za sloj *Occlusion Layer* sa slike 5.3 koji će biti zadužen za blokiranje svjetlosti.

Slika 5.3. Označavanje površina koje blokiraju svjetlost na tilesetu

Kroz iteriranje označavanja površina možemo doći do dobrih rezultata, ali Godot nam nudi još nekoliko opcija. Kako bi sve zajedno bolje izgledalo, mogu se dodatno prilagoditi postavke točkastog svjetla filtera (PCF5) i zaglađivanja sjena, što rezultira blagim prijelazima uz cijenu naglašenih traka u gradijentu sjene. Također, ako želimo stvoriti suprotni dojam, možemo promjeniti boju sjene u svijetliju prema primjeru sa slike 5.4.



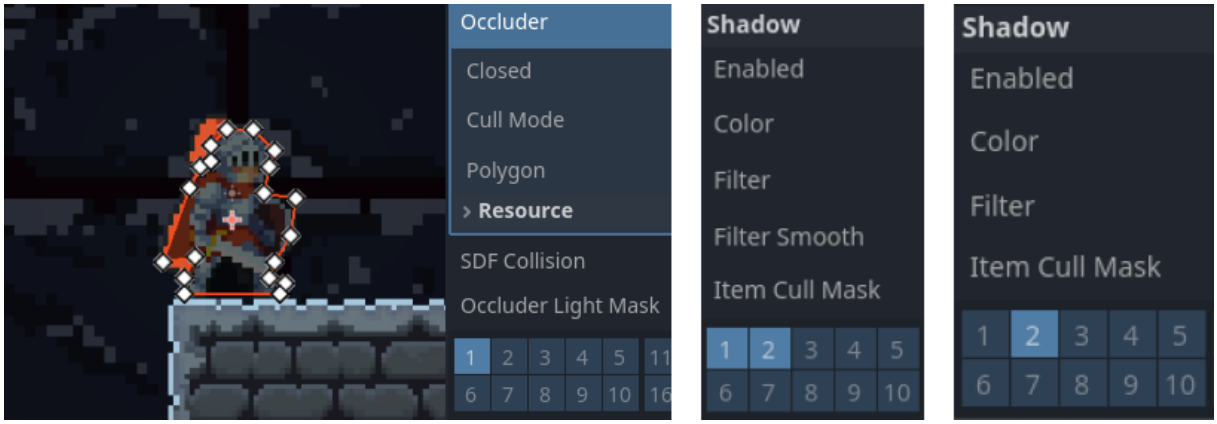
Slika 5.4. S lijeva na desno: bez sjena, sjene bez zaglađivanja, sjene sa zaglađivanjem, sjena s drugačijom bojom

Iako je scena dobila novu dubinu atmosfere, primjećujemo da je pretamna za našeg viteza s kojim igrač ne bi lako došao do sljedećeg koraka u prolasku kroz level. Zato ćemo dodati još jedno pasivno točkasto svjetlo, koje će se emitirati iza igrača. To će omogućiti, čak i u dijelovima gdje nema dostatne svjetlosti, da igrač može vidjeti blisku okolinu kao na slici 5.5.

Slika 5.5. Svjetlo slabog intenziteta na igraču

5.3. Dinamične sjene

Nastavljajući sa sjenama, možemo nadograditi našeg lika viteza, da i on ima interakciju sa svijetom, započevši s najintuitivnijem izborom, a to su sjene oko izvora svjetlosti poput baklji. To ćemo postići dodavanjem čvora na viteza tipa **LightOccluder2D** kojem pridijelimo poligon oblika kojim želimo blokirati odabrani sloj svjetla. Upravo to je prikazano na slici 5.6, s naglaskom na pažljivo odabrane slojeve u kvadratićima, kako maska igrača ne bi blokirala njegovo vlastito svjetlo, već samo okolna svjetla na sloju 1.



Slika 5.6. S lijeva na desno: Light Occluder za viteza i njegova maska, sloj za svjetlo baklje, sloj za svjetlo na igraču

Za demonstraciju blokiranja svjetlosti i stvaranje sjena, pogledajmo sliku 5.7 gdje dva objekta blokiraju svjetlo koje dolazi od baklje u sredini, pri čemu je korišten filter za zaglađivanje *PCF5*.

Slika 5.7. Demonstracija blokiranja svjetlosti

6. INTERAKTIVNOST SA SVIJETOM

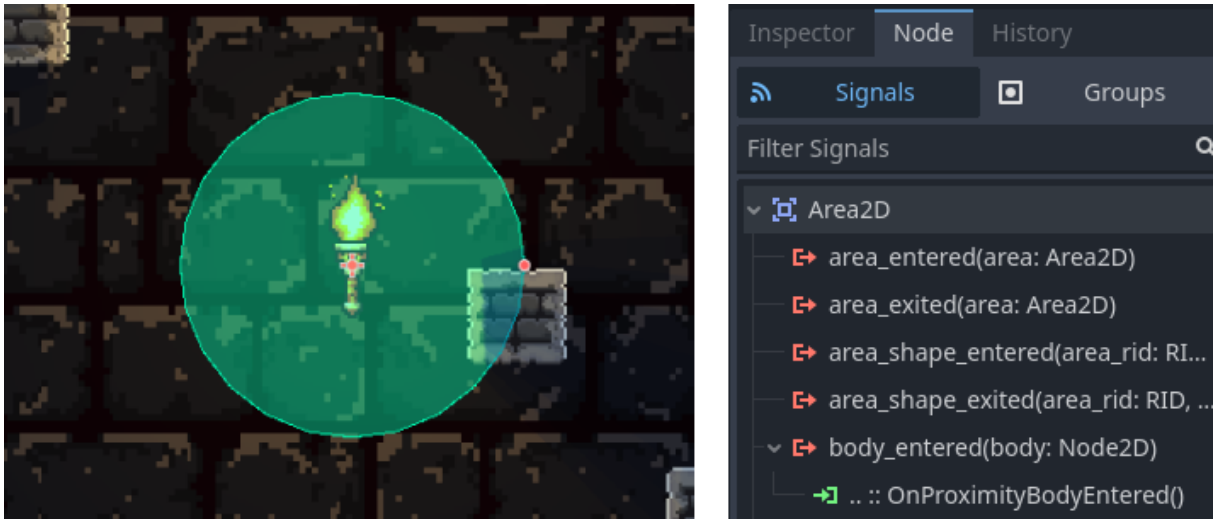
Kako bi naša igrica bila zabavna potrebna je interaktivnost igrača s okolinom (objektima). Neki od objekata koji su zahtijevali tu funkcionalnost su: baklje, *spawneri* šišmiša i sami šišmiši.

Za implementaciju svake od ovih funkcionalnosti potrebno je znanje o čvorovima i skriptama u Godot Engineu te tako povezati različite elemente igre u koherentnu cjelinu. S obzirom na to da nas zanima kada je vitez u blizini nekog objekta s kojim uspostavlja interakciju, potrebna nam je detekcija takvog događaja. Za to se koriste **Area2D** čvorovi koji opisuju prostor koji objekti zauzimaju i **CollisionShape2D** čvorovi koji definiraju veličinu prostora objekta te emitira signal kada se dogodi takav događaj.

Signali su mehanizam za komunikaciju između čvorova u Godotu. Omogućuju skripti jednog čvora da obavijesti druge čvorove o određenim događajima bez direktnog povezivanja koda, čime se poboljšava modularnost i čitljivost projekta. U C# to se postiže s delegatima i događajima (engl. *events*).

6.1. Gašenje baklji

Započnimo s jednostavnim primjerom, u kojem želimo da neke baklje se ugase ako se vitez pojavi unutar predodređene udaljenosti od baklje. Nakon definiranja površine unutar koje se detektira prolaz, potrebno je na taj čvor povezati C# skriptu sa signalom iz Godota sa slike 6.1.



Slika 6.1. Izgled površine za detekciju prolaza igrača (lijevo) i definiran signal body_entered i callback metoda OnProximityBodyEntered (desno)

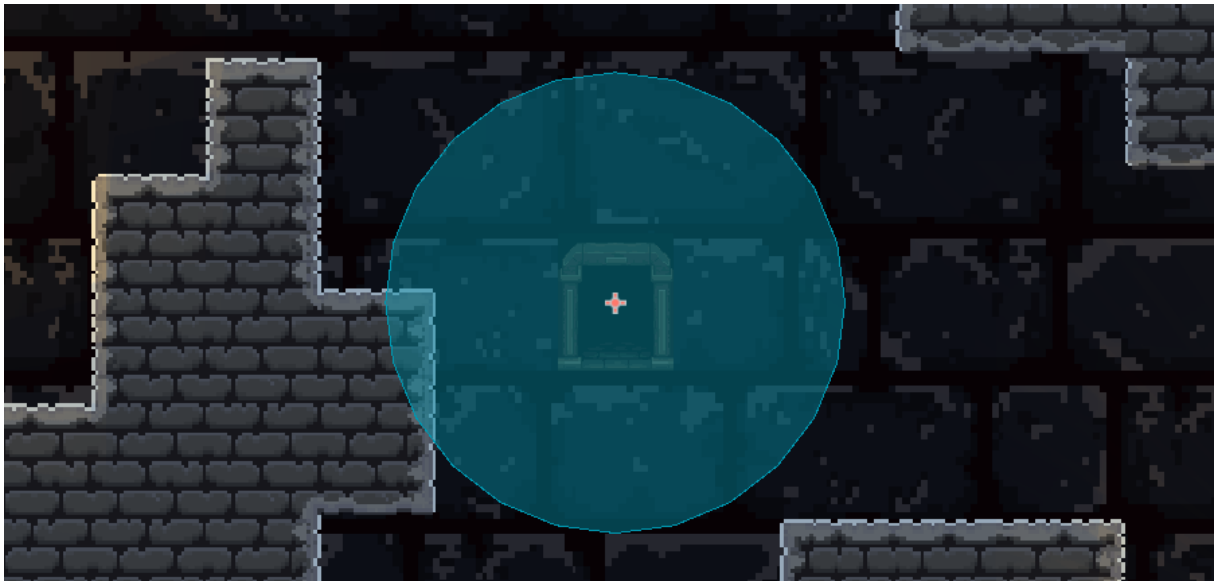
To znači da će se izvesti metoda **OnProximityBodyEntered** iz koda 6.1 kada Godot emitira događaj **body_entered**.

Isječak koda 6.1. *Gašenje svjetla i postavljanje sprite animacije na "Off" (Torch.cs)*

```
private void OnProximityBodyEntered(Node body)
{
    if (body is PlayerController && _isOn)
    {
        TurnOff();
        this.Energy = 0;
        _isOn = false;
        _sprite.Play("Off");
    }
}
```

6.2. Stvaranje i napad neprijatelja

S ciljem da se potakne igrača na brzo razmišljanje i ubrzan tempo igre, stvaramo neprijatelje u intervalima ili ako prođe blizu označenih mjesta, tzv. *spawnera.* Izvedba je vrlo slična kao i kod gašenja baklje, gdje detektiramo je li igrač prošao blizu označenog čvora s teksturom na slici 6.2.



Slika 6.2. Spawner šišmiša sa svojom površinom za detekciju prolaska

Prilikom stvaranja novog šišmiša iz *spawnera* potrebno je obratiti pažnju na to da postoji maksimalni broj aktivnih neprijatelja te vremenski interval koji ne dopušta stvaranje novog

šišmiša. U trenutku stvaranja namjestimo koordinate novog šišmiša relativno na globalni koordinantni sustav, što je prikazano u kodu 6.2.

Isječak koda 6.2. *Stvaranje novog šišmiša kod prolaska igrača pored (BatSpawner.cs)*

```
private async void HandlePlayerEntered()
{
    if (currentBatCount < MaxBats && canSpawn)
    {
        SpawnBat();
        canSpawn = false;

        await ToSignal(GetTree().CreateTimer(SpawnCooldown), "timeout");
        canSpawn = true;
    }
}

private void SpawnBat()
{
    if (Bat == null)
        return;
    CallDeferred("DeferredSpawnBat");
}

private void DeferredSpawnBat()
{
    Node2D bat = (Node2D)Bat.Instantiate();
    GetParent().AddChild(bat);
    bat.GlobalPosition = GlobalPosition;
    currentBatCount++;
}
```

Šišmiši su glavni neprijatelji u ovoj video igri. Konstantno prate igrača i mogu letjeti preko svih prepreka. Igrač biva ozlijeđen u dodiru sa šišmišem, a prema kodu 6.3 oduzima mu se život.

Isječak koda 6.3. *Prilaz igrača u blizinu šišmiša uništava ga i oduzima život vitezu (Bat.cs)*

```
private void OnProximityBodyEntered(Node body)
{
    if (body is not PlayerController) return;
    playerData.DecreaseHealth(1);
    QueueFree();  // Ukloni sismisa
}
```

Zatim se emitira signal da se smanjio broj života igrača te se pokrene prigodna animacija lika.

S obzirom na to da šišmiš nema kolizije s platformama, može se znatno lakše kretati po dvorcu, što bi značilo da igrač ne može pobjeći. Stoga, brzina šišmiša je smanjena u odnosu na igrača, te ga prati prema algoritmu prikazanom u isječku koda 6.4.

Isječak koda 6.4. *Algoritam šišmiša za praćenje igrača (Bat.cs)*

```
public override void _PhysicsProcess(double delta)
{
    if (player == null) return;
    var playerCollisionShape = player.GetNode<CollisionShape2D>("Proximity");

    if (playerCollisionShape == null) return;
    Vector2 targetPosition = playerCollisionShape.GlobalPosition;
    Vector2 direction = targetPosition - GlobalPosition;

    if (direction.Length() > 0)
    {
        targetDirection = direction.Normalized();
        Position += targetDirection * speed * delta;
        UpdateSpriteFlip(direction);
    }
}
```

Šišmiš također ima svoj animirani *sprite* prikazan slikom 6.3. Na njega također možemo staviti **LightOccluder** čvor za bacanje sjene priliko prolaska u blizini baklje.

Slika 6.3. Sprite sheet šišmiša [4]

6.3. Otključavanje škrinje

Za završetak igre potrebno je pronaći skriveni ključ te skupiti ga, a zatim ponaći škrinju koja se otvara s istim ključem. Kako su skripte vezane za same čvorove, potrebno je novo rješenje za spremanje stanja igre i podataka o igraču (engl. *game state*). Za to možemo registrirati *singleton* kontoler u postavkama Godot projekta, prikazano kodom 6.5. Time nam je omogućeno pristupanje tim podacima iz bilo koje skripte, odnosno kontrolera nekog čvora.

Isječak koda 6.5. *PlayerData singleton klasa*

```
public partial class PlayerData : Node
{
    [Signal]
    public delegate void HealthChangedEventHandler(int newHealth);
    private int health = 3; // Player's health (default is 3 hearts)
    private HashSet<string> inventory = new HashSet<string>();

    public int Health => health;

    public void AddItem(string item)
    {
        inventory.Add(item);
    }

    public bool HasItem(string item)
    {
        return inventory.Contains(item);
    }

    public void RemoveItem(string item)
    {
        inventory.Remove(item);
    }

    public void DecreaseHealth(int amount = 1)
    {
        health = Mathf.Max(0, health - amount);
        EmitSignal(SignalName.HealthChanged, health);
    }

    public void IncreaseHealth(int amount = 1)
    {
        health = Mathf.Min(3, health + amount); // Max hearts = 3
        EmitSignal(SignalName.HealthChanged, health);
    }

    public void ResetData()
    {
        health = 3;
        inventory.Clear();
    }
}
```

Sada vidimo da se u ovom *singletonu* poziva **EmitSignal**, čime se osigura pokretanje animacije viteza za ozljedu ili smrt.

Dosad smo već više puta spominjali signale, pogotovo kada čvor uđe u predodređenu površinu. Isti princip primjenjujemo i na sljedeća dva primjera, a prvo u kodu 6.6 dodajemo novo stanje u inventar igrača preko javne metode **AddItem**. Time smo dodali skupljeni ključ u inventar igrača, a zatim uklonimo animirani *sprite* ključa iz scene.

Isječak koda 6.6. *Dodavanje ključa u popis stvari u PlayerData singletonu (Key.cs)*

```
private void OnPlayerPickedUp(Node body)
{
    if (body is PlayerController)
    {
        GetNode<PlayerData>("/root/PlayerData").AddItem("key");
        QueueFree();  // Ukloni čvor ključa
    }
}
```

Potom, u trenutku kada igrač priđe škrinji provjerimo ima li igrač ključ u svom inventaru preko metode **HasItem**. Ako igrač ima ključ, jednom pokrenemo animaciju za otvaranje škrinje te završimo igru.

Isječak koda 6.7. *Otvaranje škrinje ako igrač ima ključ (Chest.cs)*

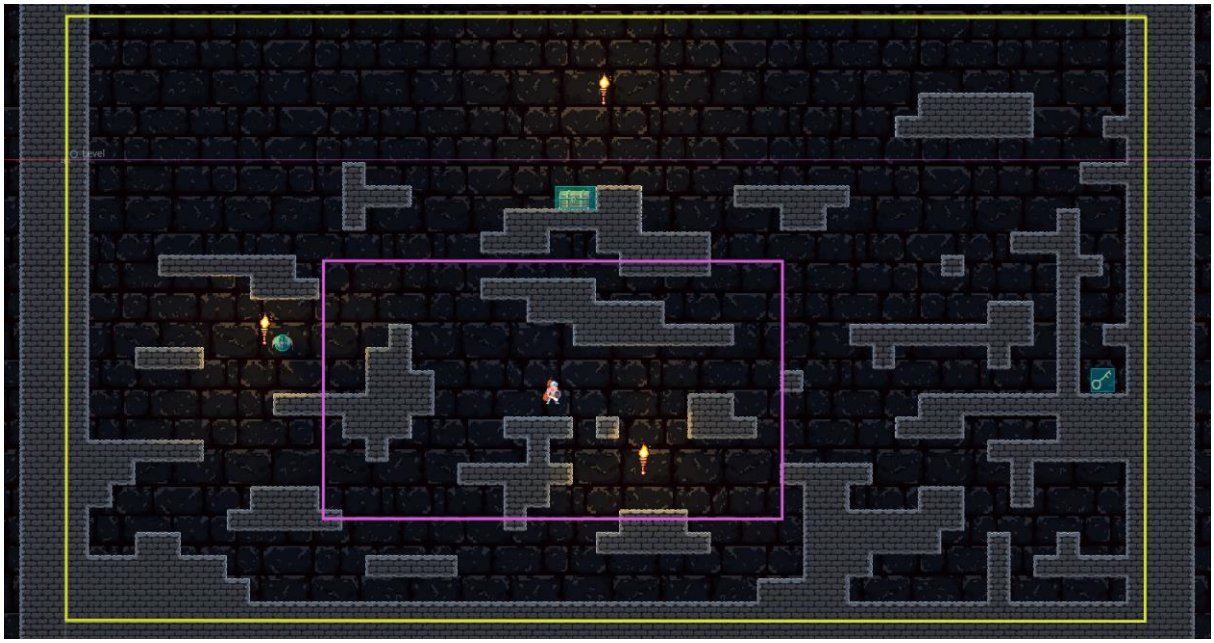
```
public void OnPlayerTryToUnlock(Node body)
{
    if (body is PlayerController)
    {
        if (GetNode<PlayerData>("/root/PlayerData").HasItem("key"))
        {
            GetNode<AnimatedSprite2D>("ChestSprite").Play("open");
        }
    }
}
```

Izgled ključa i škrinje prilikom otvaranja možemo vidjeti na slici 6.4.

Slika 6.4. Animirani spriteovi za ključ (gore)[5] i škrinju (dolje)[6]

7. KAMERA

Godot Engine nudi čvor s imenom **Camera2D** s kojim se jednostavno može konfigurirati kako će se kamera ponašati. Prvo isti čvor dodamo čvoru igrača te odredimo količinu **Zooma** u postavkama kamere, što je na slici 7.1 označeno ružičastim pravokutnikom oko viteza. S obzirom na to da je igrač čvor-roditelj čvoru kamere, kamera će se zajedno s igračem translatirati po 2D površini. Postavkom **Smoothed** uključujemo interpolaciju pozicije kamere dok igrač mijenja smjer kretanja.



Slika 7.1. Izgled kamere na sceni

Zadnji problem koji je ostao jest scenarij u kojem je vitez blizu ruba ili poda, ne želimo da je kamera centrirana i dalje na igrača, već da ima granice dokle se smije translatirati. Na slici poviše to je prikazano žutim okvirom, a definira se postavkama kamere **Limit** za svaku od četiri stranice.

8. ZAKLJUČAK

Izrada 2D pixel art video igre u Godot Engineu pokazala se kao zanimljiv i edukativan proces koji je spojio tehničko znanje i kreativnost. Kroz rad na jednostavnom 2D platformeru naučili smo koristiti ključne alate i funkcionalnosti ovog moćnog alata za razvoj igara.

Korištenjem *tilesetova* i *tilemapa* uspješno smo stvorili raznolike i modularne levele, dok su animirani *spriteovi* oživjeli likove i okoliš. Implementacija svjetla i sjena, poput baklji s dinamičkim osvjetljenjem, dodala je dodatni sloj atmosfere i realizma. Godotov intuitivan sustav skriptiranja omogućio je jednostavnu integraciju mehanika igre, poput kretanja igrača i interakcije s objektima, što je značajno ubrzalo razvojni proces.

Ovaj projekt omogućio nam je bolje razumijevanje razvoja igara te nam je pokazao koliko detalja i pažnje zahtijeva svaki aspekt izrade video igre, od dizajna grafike do optimizacije koda. U budućnosti se ova igra može proširiti dodavanjem novih razina, neprijatelja i izazova kako bi dodatno unaprijedili stečene vještine i omogućili igračima bogatije iskustvo.

Razvoj ove igre potvrđuje kako je Godot Engine idealan alat za entuzijaste i studente koji žele zakoračiti u svijet razvoja video igara, pružajući fleksibilnost i prilagodljivost čak i za složenije projekte.

LITERATURA

[1] Jordon Games: „Castle Brick Tileset“, https://jordon-games.itch.io/castle-stone-tileset

[2] Mattz Art.: „Knight 2D Pixel Art“, https://xzany.itch.io/free-knight-2d-pixel-art

[3] Rone3190: „Torch Animated“, https://rone3190.itch.io/torch-32x32-animated

[4] PixelSkey: „Bat Pixel Art Pack“, https://pixelskeys.itch.io/bat-pixel-art-pack-free

[5] Frakasset.: „Rotating Key - Animated Pixel Art“ , https://frakassets.itch.io/free-rotating-key

[6] Karsiori: „Pixel Art Chest Pack - Animated“, https://karsiori.itch.io/pixel-art-chest-pack-animated

[7] Mišadin, D.; Akrapović, I.: „Izvorni programski kod video igre u Godot Engineu CaveClimber“, https://github.com/dmisadin/CaveClimber