Izrada 2D igre u Unity engine-u

Bolarić, Kristian

VELEUČILIŠTE SUVREMENIH INFORMACIJSKIH TEHNOLOGIJA

STRUČNI PRIJEDIPLOMSKI STUDIJ INFORMACIJSKIH TEHNOLOGIJA

Kristian Bolarić

ZAVRŠNI RAD

IZRADA 2D IGRE U UNITY ENGINE-U

Zagreb, listopada 2024.

Veleučilište suvremenih informacijskih tehnologija
10000 Zagreb, Ulica Vjekoslava Klaića 7

Studij:	Stručni prijediplomski studij informacijskih tehnologija smjer programiranje
Student:	**Kristian Bolarić**
Matični broj:	2021008

Zadatak završnog rada

Predmet:	Programiranje u C#
Naslov:	**Izrada 2D igre u Unity engine-u**
Zadatak:	Opisati izradu igara u okruženju Unity. U okviru rada napraviti oglednu igru s dinamičkom promjenom terena.
Mentor:	mr. sc. Julijan Šribar, v. pred.
Zadatak uručen kandidatu:	24.5.2024.
Rok za predaju rada:	17.10.2024.
Rad predan:	_____

Povjerenstvo:

Jurica Đurić, v. pred.	član predsjednik	_____
mr. sc. Julijan Šribar, v. pred.	mentor	_____
Mariza Maini, pred.	član	_____

SADRŽAJ

POPIS SLIKA

POPIS KODOVA

1. UVOD

Razvoj videoigara i interaktivnih sadržaja doživio je značajan napredak zahvaljujući razvoju novih tehnologija i alata koji omogućavaju stvaranje složenih i vizualno impresivnih projekata. Jedan od najpopularnijih alata za razvoj videoigara i drugih interaktivnih aplikacija je Unity, multiplatformski game engine koji se koristi širom svijeta.

Industrija razvoja videoigara je grana informacijskih tehnologija koja ne oprašta pogreške u njihovoj implementaciji. Za izradu uspješne videoigre potrebne su brojne vještine koje uključuju: poznavanje enginea u kojem se planira razviti igra, poznavanje jezika koji su uključeni u engine (u slučaju Unityja to su C# i JavaScript), mogućnost izrade 2D spriteova ili 3D modela, koji su dostupni i preko Unity Asset Storea, mogućnost izrade zvučnih efekata i glazbe za igru. Zbog svih ovih potrebnih vještina, videoigre se razvijaju u timovima čiji su članovi iskusni u pojedinim područjima nabrojenih vještina.

Ovaj rad opisuje razvoj videoigara samostalnog programera (engl. *indie developer*, od engl. *independent* – samostalno), od razvoja kôda i njegove optimizacije, do vizualnog dizajna videoigre. Spomenut će se i važnost izbora stila izvedbe videoigre i kako odabir stila utječe na cijeli razvoj. Opisat će se Unity i njegova radna okolina. Proći će se kroz njegove komponente koje je nužno razumjeti za uspješan razvoj videoigre. Osim prozora koje Unity prikaže pri prvom pokretanju, to uključuje i temeljne klase i metode koje Unity implementira svojim Unity Scripting API-em [2].

Iako je cilj ovog rada proći kroz Unity radnu okolinu, spomenut će se alternative koje se mogu koristiti umjesto njega. Svaka od tih alternativa ima svoje prednosti i nedostatke koje je važno razumjeti prije odabira alata za razvoj.

U praktičnom dijelu ovog rada proći će se kroz sve korake i temeljne sustave potrebne da videoigra bude potpuna i spremna za korištenje. Prikazat će se logika iza objekata u igri i objasniti njihova implementacija. Također će se proći kroz važne Unity komponente kao animator i kako povezati njegove animacije s kôdom. U videoigri su korišteni resursi napravljeni isključivo za nju. Ti resursi uključuju 2D modele (spriteove), zvučne efekte, slike i glazbu.

Sama igra odvija se u 2D okolini rađenoj PixelArt tehnikom. Cilj igrača je doći do kraja nivoa korištenjem dinamičke promjene terena. Igrač nema mogućnost borbe stoga mu je jedina opcija pomno zaobilaziti svaku prepreku i neprijatelja na kojeg naiđe. Na pojedinom nivou su skriveni

kristali koje igrač može sakupiti kako bi poboljšao svojstva svojeg avatara te se pripremiti za veliku borbu na kraju.

Ovaj rad pruža detaljan uvid u proces razvoja videoigre koristeći Unity, nudeći praktične smjernice i primjere, korak po korak, koji mogu poslužiti kao osnova za daljnji rad i usavršavanje.

2. RAZVOJ VIDEOIGARA

Ovo poglavlje opisuje najčešće probleme u razvoju videoigara.

2.1. Alati

Za izradu videoigre potrebni su brojni alati. Osim enginea, koji je glavna karika u izradi, potrebni su i alati za:

- pisanje kôda, to jest skripti (npr. Visual Studio)
- 3D modeliranje u slučaju izrade 3D videoigre (npr. Blender)
- izradu slika ili 2D spriteova u slučaju izrade 2D igre (npr. Photoshop)
- izradu glazbe i oblikovanje zvučnih efekata (npr. FL Studio).

2.2. Stil

Razvoj igre ovisi o odabiru vizualnog stila, npr. radi li se o 2D ili 3D igri. Za svaki stil potrebne su posebne vještine i mogućnost rada u alatima neophodnima za izvođenje tog stila. Na primjer, ako je odabrani stil igre 3D, potrebna je vještina rada u programu za 3D modeliranje, kao što je Blender.

2.3. Inovativnost

Svaka uspješna videoigra mora biti inovativna. Od vizualnog izgleda aplikacije, kao što su okoliš u koji će se igrač smjestiti ili animacije, do izazova u kojima će se igrač naći. Zato pri planiranju izrade videoigre treba voditi računa postoji li interes za takvu igru i po čemu bi se ona razlikovala od već postojećih igara na tržištu.

2.4. Game engine

Proces izrade videoigre ovisi o game engineu koji se odabere. Svaki game engine ima svoja posebna svojstva koja ga čine privlačnim. Veće kompanije često imaju svoje game engine razvijene unutar kompanije. Tri najpopularnija enginea koji se globalno koriste su:

- Unreal Engine
- Godot
- Unity.

2.5. Unreal Engine

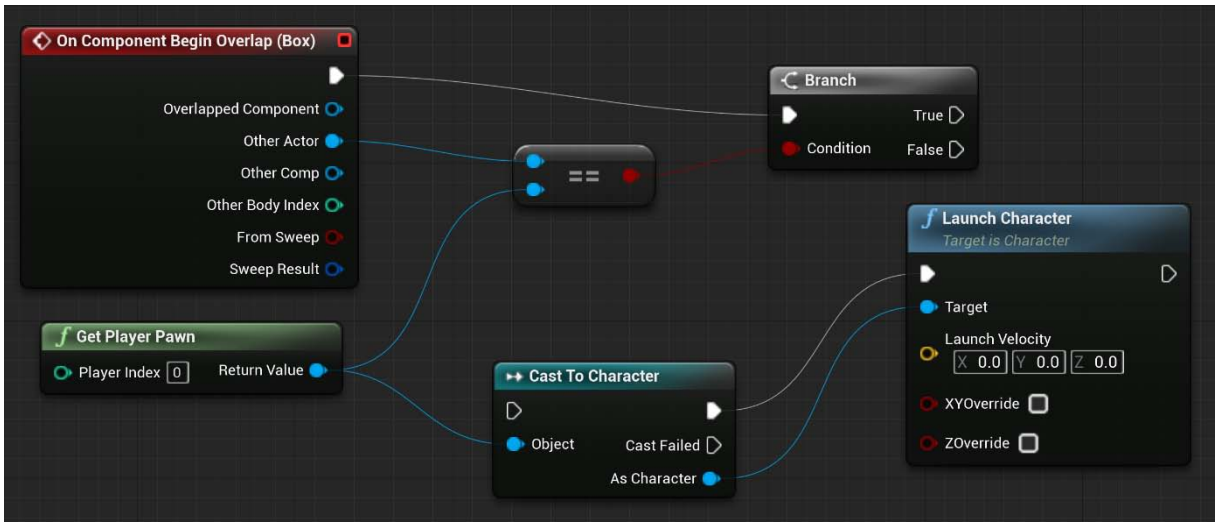
Nakon svoje zadnje velike verzije (Unreal Engine 5), Unreal Engineu je narasla popularnost i poprimio je reputaciju enginea s najboljim vizualnim mogućnostima. Njegov sustav

postavljanja osvjetljenja i veliki izbor mogućnosti pri izradi 3D terena čine ga najboljim izborom za izradu 3D igara.

Logika u Unreal Engineu se odrađuje na dva načina:

- Kôdom pisanim u jeziku C++
- Korištenjem *blueprintsa.*

Blueprints je alat koji omogućava vizualno programiranje: umjesto pisanja kôda, postavljaju se čvorovi (engl. *node*s) koji se onda povezuju (Slika 1).



Slika 1. Unreal Blueprint (Epic Games, 2024)

2.6. Godot

Godot je open source engine koji privlači publiku činjenicom da je kompletno besplatan. Za razliku od Unityja i Unreal Enginea, videoigre napravljene u Godotu mogu se komercijalno koristiti bez potrebe plaćanja, neovisno o iznosu koji programer zaradi. Osim toga, Godot ima podršku za C++, C# i svoj vlastiti GDScript koji je sličan Pythonu i lagan je za naučiti. Godot se najviše koristi za izradu 2D igara.

2.7. Unity

Unity je široko dostupan engine. Njegovo jednostavno sučelje čini ga odličnim izborom za početnike. Početnici također mogu zarađivati od svojih radova koristeći besplatnu osobnu licencu. Unity je popularan alat, pa je dostupna velika količina literature u obliku dokumentacije, objava na društvenim mrežama te YouTube videozapisa [3]. Unity je detaljno opisan u sljedećem poglavlju.

3. STRUKTURA UNITYJA

U ovom poglavlju opisat će se mogućnosti Unityja te kako izgleda njegova razvojna okolina.

3.1. Uvod u Unity

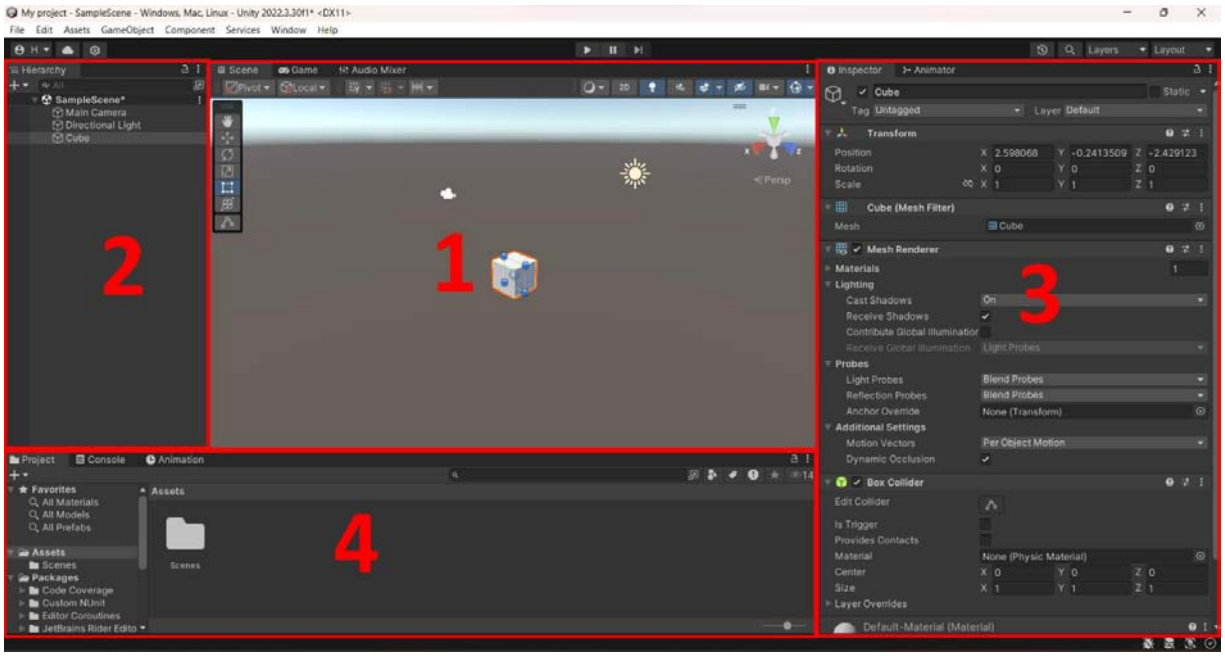
Unity [4] je besplatan višeplatformski engine za izradu videoigara koji je objavila tvrtka Unity Technologies u lipnju 2005. godine. Inicijalno je služio kao game engine za Mac OS, a kasnije je proširen na ostale platforme. Pomoću Unityja moguće je raditi videoigre za računala, konzole, mobilne te uređaje virtualne stvarnosti (engl. *virtual reality, VR*).

Unity je popularan među samostalnim programerima zbog svoje strukture koju mogu i početnici lako shvatiti. Unity omogućuje besplatni razvoj i zaradu od videoigara za osobno korištenje ili za manje kompanije koje imaju godišnji prihod manji od 200.000$ korištenjem osobne licence (engl. *Personal License*).

Za pisanje kôda u Unityju koristi se C#. Ranije je bilo moguće i u JavaScriptu ali je 2017. godine ukinuta njegova podrška.

3.2. Početni izgled Unityja

Pri prvom pokretanju, razvojna okolina Unityja je podijeljena na četiri dijela (Slika 2).

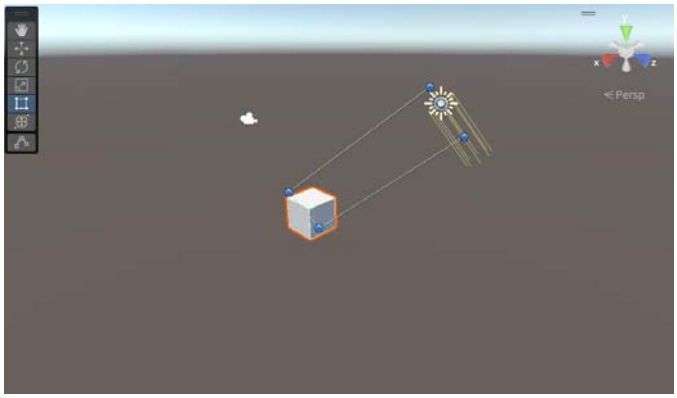


Slika 2. Početni zaslon Unityja

Okvir označen brojem 1 prikazuje Scene i Game poglede te Audio Mixer. Okvir označen brojem 2 prikazuje hijerarhiju objekata. Okvir označen brojem 3 prikazuje Inspector i Animator prozore. Okvir označen brojem 4 prikazuje Project, Console i Animation prozore.

3.2.1. Scene pogled

Scene pogled (engl. *view*) pruža pogled scene (Slika 3).

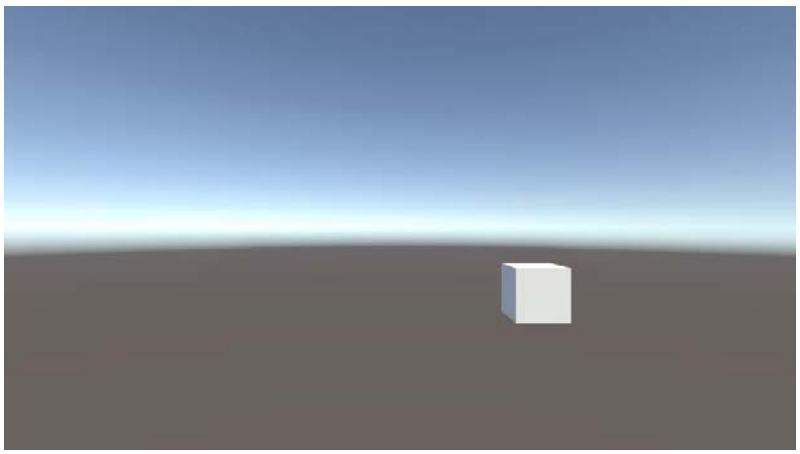


Slika 3. Unity Scene pogled

U njoj se nalaze svi postavljeni objekti čijom se pozicijom, oblikom i veličinom jednostavno može manipulirati. Pomicanjem kamere može se promijeniti kut pogleda na scenu.

3.2.2. Game pogled

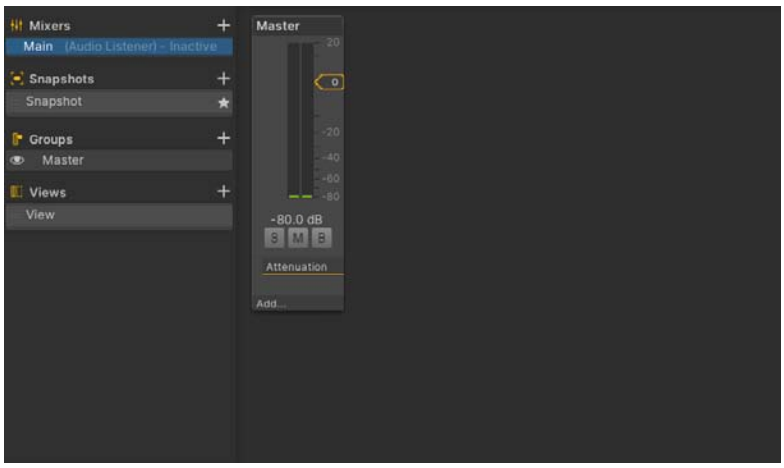
Game pogled (Slika 4) je sličan Scene pogledu, a razlikuje se po tome da pogled na scenu daje iz igračeve perspektive. On ne dozvoljava odabir i manipulaciju objektima već služi isključivo za provjeru igračeve kamere.



Slika 4. Unity Game pogled

3.2.3. Audio Mixer

Audio Mixer služi za rad sa zvukom videoigre (Slika 5).

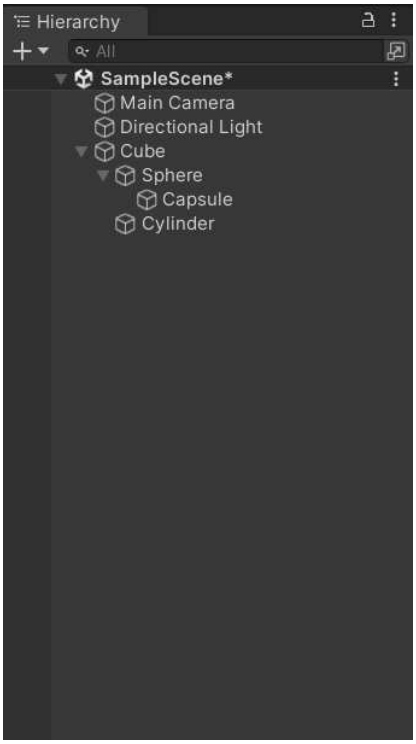


Slika 5. Unity Audio Mixer

U njemu je moguće grupirati zvučne izlaze u različite kanale, te dodavati efekte na njih.

3.2.4. Hierarchy

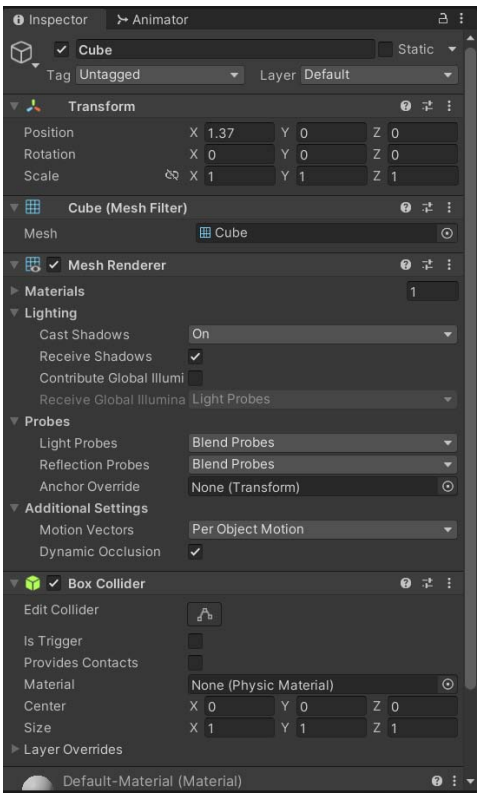
Prozor Hierarchy (Slika 6) služi za prikaz svih objekata u sceni u hijerarhijskom obliku te omogućava lagani pristup pojedinim objektima i njihovo grupiranje. Objekti se mogu vezati u strukturu *parent – child* tako da se jedan objekt jednostavno povuče na drugi.



Slika 6. Prozor Hierarchy

3.2.5. Inspector

Prozor Inspector (Slika 7) služi za pregled svojstava odabranog objekta.



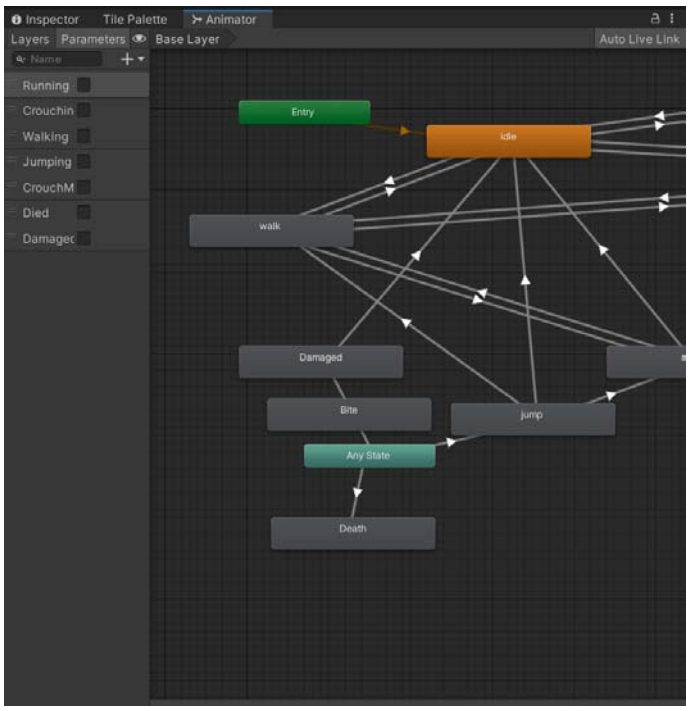
Slika 7. Prozor Inspector

Na odabrani objekt moguće je dodavati razna svojstva kao što su collideri, efekti ili skripte koje će diktirati ponašanje objekta.

Svaki objekt na sebi ima komponentu Transform koja sadrži koordinate, rotaciju i veličinu objekta. Pomicanje objekta je također moguće promjenom komponente Transform.

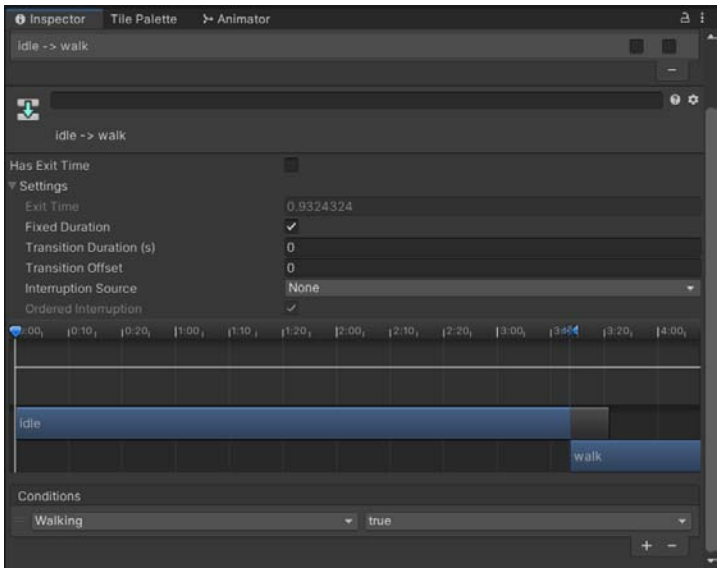
3.2.6. Animator

Animator (Slika 8) je prozor u kojem moguć rad s animacijama odabranog objekta.



Slika 8. Prozor Animator

Pomoću njega se upravlja tranzicijama između animacija kreiranih u prozoru Animation. Tranzicijama se upravlja pomoću parametara koji se definiraju u Animatoru, usporedbom njihovih vrijednosti s onima u listi uvjeta za tranziciju (Slika 9).



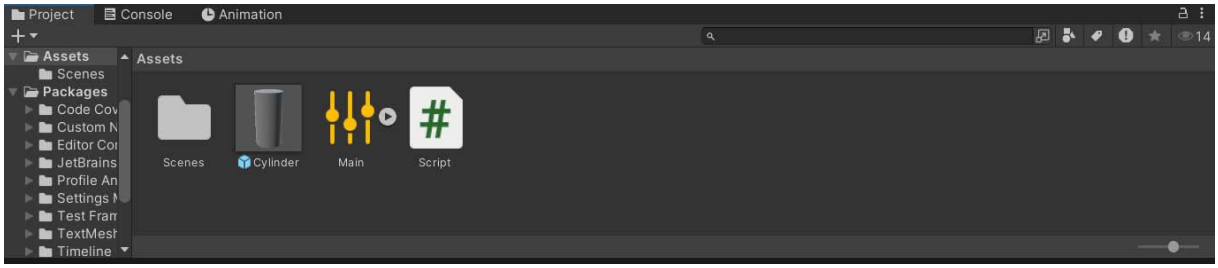
Slika 9. Stanja za prijelaz animacija

Njihova vrijednost može se promijeniti u skriptama kreiranjem reference na komponentu Animator te pristupom parametru na sljedeći način.

```
>>> Animator.SetBool("imeParametra", true);
```

3.2.7. Project

U prozoru Project (Slika 10) moguće je pristupiti svim resursima uvezenim u projekt.



Slika 10. Prozor Project

Neki od resursa uključuju scene, 3D modele, 2D spriteove, zvučne komponente, skripte.

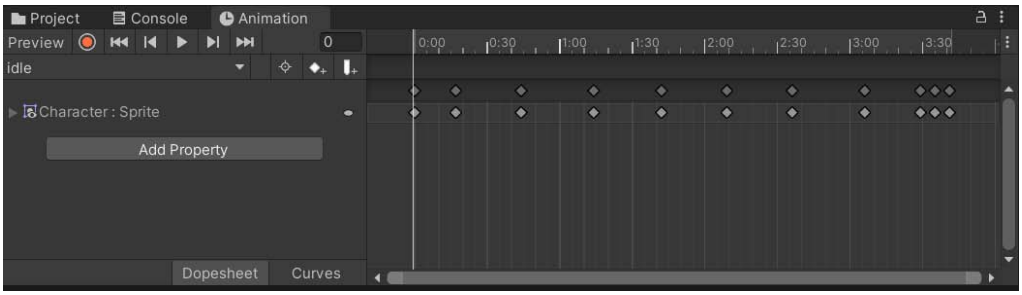
3.2.8. Console

Prozor Console prikazuje jednostavnu konzolu. U njoj se prikazuju pogreške koje prijavljuju compiler i engine te poruke poslane sljedećom komandom.

```
>>> Debug.Log("Poruka");
```

3.2.9. Animation

U prozoru Animation (Slika 11) postavljaju se animacije za odabrani objekt.



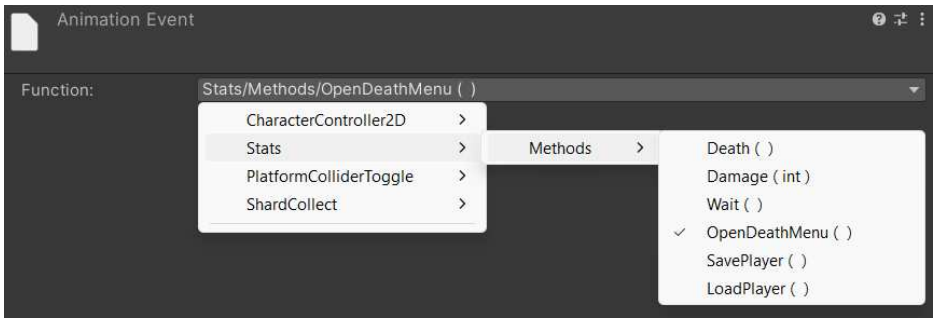
Slika 11. Prozor Animation

Bijeli rombovi na vremenskoj traci nazivaju se *Keys*. Oni predstavljaju ključne trenutke u animaciji, kao što su promjena slike ili pomicanje kosti za animiranje. Mogu se odnositi i na promjenu lokacije cijelog objekta ili uključivanje i isključivanje neke komponente. To se radi u *keyframe recording* modu, koji se aktivira crvenom tipkom *record*.

Na animacije je također moguće dodati događaj (engl. *event*), koji je simboliziran bijelom crticom na vremenskoj liniji (Slika 12).

Slika 12. Animation event

Taj događaj poziva metodu iz bilo koje skripte vezane za objekt koji se animira (Slika 13).



Slika 13. Pozivanje metode animation eventom

3.2.10. Skripte u Unityju

Unity skripte [2] sadržavaju svu logiku iza objekata u sceni. Kôd se piše u jeziku C#. Pri kreiranju C# skripte, kreira se klasa s dvije ugrađene metode: `Start` i `Update`. Inicijalni kôd prikazan je u primjeru (Kôd 1).

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class Skripta : MonoBehaviour
{
    // Start is called before the first frame update
    void Start() {  }
    // Update is called once per frame
    void Update() {  }
}
```

Kôd 1. Unity script

Prilikom stvaranja, svaka klasa nasljeđuje klasu `MonoBehaviour`, koja omogućuje da se instance klase pridruži objektu i dijele njegov životni ciklus.

Kôd unutar metode `Start` se izvodi prije prve izvedbe metode `Update`, dok se metoda `Update` izvodi za svaki frame.

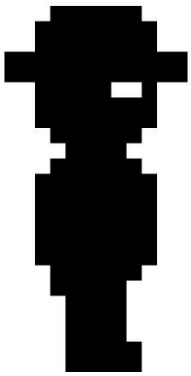
Postoje još dvije varijante metode `Update`:

- `FixedUpdate` – izvodi se neovisno o frameovima, određenom promjenjivom brzinom
- `LateUpdate` – izvodi se nakon metode `Update`, na samom kraju framea.

4. PRAKTIČNI RAD – IZRADA 2D IGRE U UNITY ENGINEU

Cilj praktičnog rada bio je napraviti 2D igru u Unityju s funkcionalnim temeljnim sustavima: glavni izbornik, postavke, mogućnost spremanja i nastavka igre. Igra se fokusira na napredak kroz nivo naizmjeničnim promjenom iz jedne u grugu dim enziju, gdje svaka im a svoj e prepreke.

Za vizualni prikaz korištene su crna, bijela te nijanse sive boje. Grafika je realizirana pomoću PixelArt umjetničkog stila (Slika 14), u kojem se slike stvaraju u niskoj rezoluciji, pomoću piksela, te se kasnije skaliraju na veće rezolucije.



Slika 14. Čovjek napravljen pomoću PixelArt tehnike

4.1. Kreiranje glavnog lika

Za početak, postavljen je glavni lik kojim će igrač upravljati. To se odrađuje u tri faze:

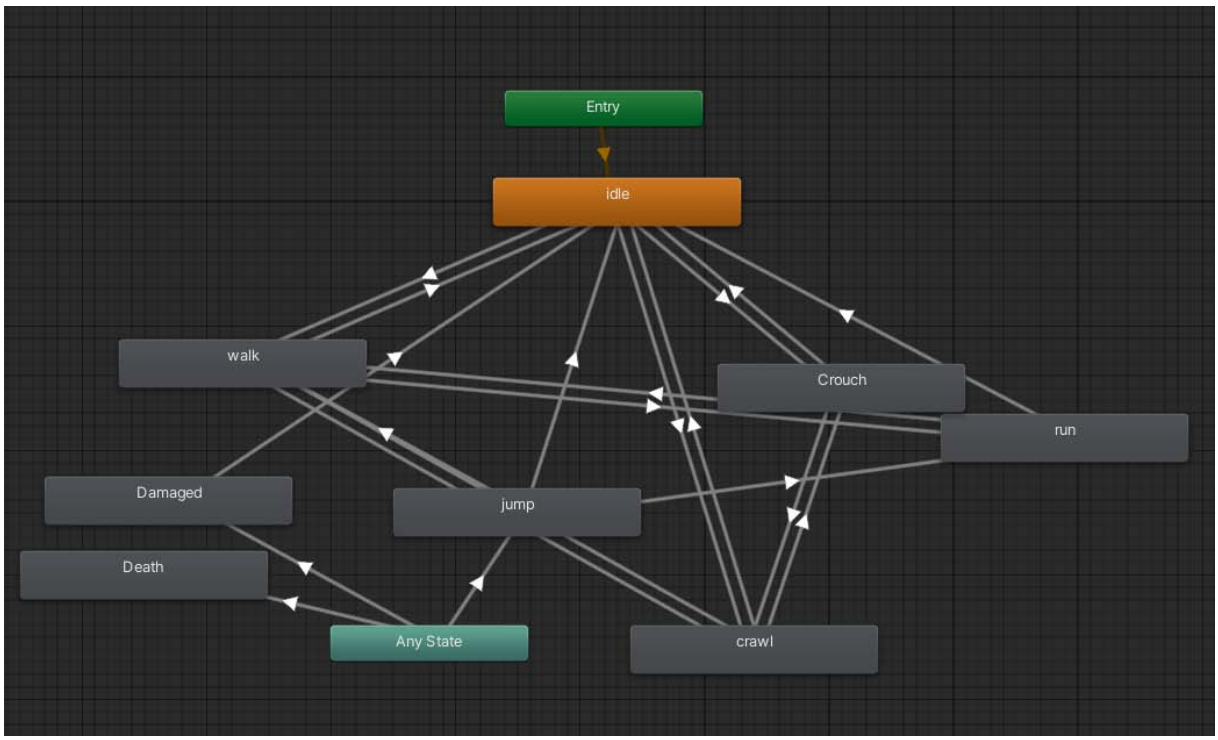
- Crtanje objekta i njegovih animacija
- Pisanje logike iza objekta
- Povezivanje objekta, animacija i logike.

Nakon vizualnog dizajna glavnog lika prelazi se na pisanje logike kojom će se njime upravljati. Prvo se u metodi `Update` provjerava unos s tipkovnice (Kôd 2).

```
if (Input.GetButtonDown("Jump") && !isCarrying)
{
  jump = true;
  animator.SetBool("Jumping", true);
}
if (Input.GetButtonDown("Crouch") && !run && !isJumping &&
    !isCarrying)
{
  crouch = true;
}
else if (Input.GetButtonUp("Crouch"))
{
  crouch = false;
}
if (Input.GetKeyDown(KeyCode.LeftShift) && !crouch &&
                     !isCarrying && stats.stamina > 0)
{
  Run();
}
else if (Input.GetKeyUp(KeyCode.LeftShift) && !crouch && run)
{
  StopRunning();
}
```

Kôd 2. Provjera pritiska tipke na tipkovnici

Nakon provjere postavljaju se parametri animatora kako bi se na glavnom liku prikazivale odgovarajuće animacije (Slika 15).



Slika 15. Animacije glavnog lika povezane u prozoru Animator

Također se postavljaju odgovarajuće logičke (engl. *bool*) vrijednosti koje se šalju metodi `Move`, koja pomoću tih vrijednosti upravlja kretanjem igrača. Metoda `Move` se poziva u metodi `FixedUpdate` i radi provjere prikazane u Kôd 3 i u Kôd 4.

```
if (!crouch)
{
  if (Physics2D.OverlapCircle(ceilingCheck.position,
                              ceilingRadius, crouchCheck))
  {
    crouch = true;
  }
}
```

Kôd 3. Postavljanje čučnja ako igrač ne može ustati

```
if (crouch && grounded)
{
  //...
  move *= stats.crouchSpeed;
  crouchDisableCollider.enabled = false;
  standDisableCollider.enabled = true;
}
else
{
  crouchDisableCollider.enabled = true;
  standDisableCollider.enabled = false;
     //...
}
```

Kôd 4. Ulazak u stanje čučnja

Igrač se pokreće pristupom Rigidbody komponenti igrača. Komponenta Rigidbody je odgovorna za sve sile koje djeluju na objekt. To uključuje gravitaciju, direktan pristup i manipulaciju brzinom objekta (Kôd 5) te korištenje metode `AddForce` (Kôd 6).

```
targetVelocity = new Vector2(move * 10f, rb.velocity.y);
rb.velocity = Vector3.SmoothDamp(rb.velocity, targetVelocity,
                                 ref velocity, movementSmoothing);
```

Kôd 5. Promjena svojstva *velocity* komponente Rigidbody

```
if (grounded && jump)
{
  grounded = false;
  rb.AddForce(new Vector2(0f, stats.jumpForce));
}
```

Kôd 6. Skakanje pomoću metode `AddForce`

Igrač se pokreće direktnim postavljanjem brzine njegovog objekta, a za glađi osjećaj kretanja se koristi varijabla `movementSmoothing` koja se postavlja metodom `SmoothDamp`.

Brzina igrača određena je u klasi `Stats` (Kôd 7). Ona sadržava svojstva glavnog lika.

```
public class Stats : MonoBehaviour
{
  public int maxHealth = 3;
  public int health;
  public float stamina;
  public float maxStamina = 100f;
  public int shardsCollected = 0;
  public float staminaRecovery = 2f;
  public float staminaDrain = 5f;
  public float speed = 100;
  public float runSpeed = 50f;
  public float jumpForce = 2500;
//...
}
```

Kôd 7. Klasa `Stats`

4.2. Promjena dimenzije

Prelazak u drugu dimenziju radi izbjegavanja prepreka jest osnovni način za rješavanje izazova u igri. Pritiskom tipke ALT igrač mijenja dimenziju, što odmiče određene prepreke i stvara nove.

Promjena dimenzije implementira se razvrstavanjem terena u tri grupe:

- DimensionObjects – objekti koji se uključe pri ulasku u dimenziju
- RemoveOnDimension – objekti koji se isključe pri ulasku u dimenziju
- Static – objekti koji su uvijek prisutni.

Za kontrolu ulaska u dimenziju koristi se prazan objekt s uključenom skriptom (Kôd 8).

```
private void FixedUpdate()
{
if (Input.GetKey(KeyCode.LeftAlt) &&
    Time.time > lastUsedTime + dimensionCooldown &&
    !dimensionLocked)
  {
    if (!isDimension && !PauseMenu.isPaused &&
        !CheckpointMenu.inMenu)
    {
      dimensionObject.SetActive(true);
      removeOnDimension.SetActive(false);
      dimensionEffect.SetActive(true);
      isDimension = true;
      enterRiftAudio.Play();
    }
    else if (!PauseMenu.isPaused && !CheckpointMenu.inMenu)
    {
      dimensionObject.SetActive(false);
      removeOnDimension.SetActive(true);
      dimensionEffect.SetActive(false);
      isDimension = false;
      exitRiftAudio.Play();
    }
    lastUsedTime = Time.time;
  }
```

Kôd 8. Promjena dimenzije

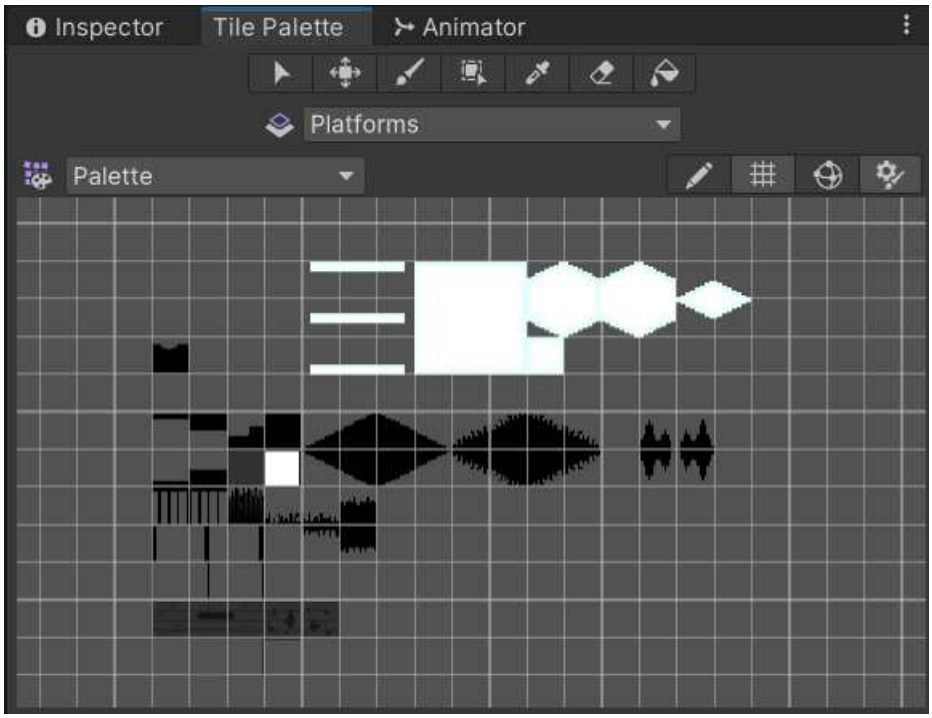
Metoda `FixedUpdate` provjerava pritisak tipke ALT za prelazak u drugu dimenziju. Uz pomoć Unity klase `Time` koja je odgovorna za praćenje vremena, implementirana je kratka pauza između promjene dimenzije, kako bi se onemogućila trenutačna promjena.

Napravljene su reference na tri grupe terena, koje se uključuju i isključuju metodom `SetActive`. Ako je ulazna vrijednost *true* objekt se uključi, a u protivnom isključi.

Pomoću logičke vrijednosti `isDimension` prati se je li igrač u dimenziji ili nije, kako bi se uključile i isključile željene grupe terena.

4.3. Teren

Za 2D videoigre, Unity nudi Tilemaps koji omogućuju da se po njima teren igre crta. 2D spriteovi se umeću u Tile Palette, koji funkcionira kao paleta boja u nekom programu za crtanje, a svaki tile predstavlja jednu boju. Njemu je moguće pristupiti preko prozora Tile Palette (Slika 16).



Slika 16. Unity prozor Tile Palette

Tilemap se u scenu dodaje kao objekt, a pri kreiranju Tilemapa automatski će se kreirati mreža (engl. *Grid*) koja određuje veličinu svakog Tilea. Na objekt se dodaje Tilemap Collider, koji dodaje svakom Tileu njegov vlastiti Collider. Nakon pripreme Tilemapa i Tile Palette, može se krenuti s kreiranjem terena pomoću alata na prozoru Tile Palette.

4.4. Prepreke

Igraču je potrebno je postaviti prepreke i neprijatelje koji će mu predstavljati izazov. Prepreke mogu biti stacionarne ili mogu aktivno napadati igrača.

4.4.1. Glitch

Glitch je stacionarna prepreka koja se nalazi samo u dimenziji i kada ju igrač dotakne, gubi jedan životni bod. To se implementira korištenjem metode `Damage` u klasi `Stats` koja će

smanjiti vrijednost varijable odgovorne za životne bodove igrača (Kôd 9). Metoda je javna kako bi joj sve ostale prepreke u budućnosti mogle slobodno pristupiti. Kao ulazni parametar prima cijeli broj koji određuje za koliko treba životne bodove umanjiti.

```
private void OnCollisionEnter2D(Collision2D collision)
{
  if (collision.gameObject.tag == "Player")
  {
    // ...
    collision.gameObject.GetComponent<Stats>().Damage(1);
    // ...
  }
}
```

Kôd 9. Pozivanje metode `Damage` pri koliziji s preprekom

Za provjeru sudara s *Glitchom*, koristi se ugrađena Unity metoda `OnCollisionEnter2D`, koja se poziva kada se dva 2D Collidera dotaknu. Tada se provjerava je li tag objekta koji se sudario „Player“. Ako je, pristupa se metodi `Damage` te smanjuju životni bodovi igrača.

4.4.2. Tracker Glitch

Slično kao *Glitch*, *Tracker Glitch* je inicijalno statička prepreka u dimenziji. Razlika je u tome da kada igrač dođe na određenu udaljenost i ako nije sagnut, *Tracker Glitch* će ga početi pratiti (Kôd 10).

```
private void OnTriggerEnter2D(Collider2D other)
{
  if(other.gameObject.CompareTag("Player"))
  {
    //...
    player.GetComponent<CharacterController2D>();
    if (!characterController.crouch)
    {
      isAwake = true;
      StartCoroutine(ExplosionTimer());
    }
}
```

Kôd 10. Pokretanje pomoću metode `OnTriggerEnter2D`

Ugrađena metoda `OnTriggerEnter2D` se poziva kada neki objekt uđe u okidač (engl. *trigger collider*) objekta koji nosi tu skriptu. Okidač je prostor u kojem neki objekt može detektirati ostale objekte.

Pri ulasku igrača u okidač Tracker Glitch se pokreće. Postavljanjem logičke vrijednosti `isAwake` na *true* omogućuje se izvođenje kôda za praćenje igrača (Kôd 11).

```
if (isAwake)
{
  if (player.transform.position.x < transform.position.x)
  {
    transform.position = position.position –
                         1 * xSpeed * Time.deltaTime;
  }
  else
  {
    //ako je igračeva X koordinata veća, koristi se zbrajanje
  }
  if (player.transform.position.y < transform.position.y)
  {
    transform.position = position.position
                        - 1 * ySpeed * Time.deltaTime;
  }
  else
  {
    //ako je igračeva Y koordinata veća, koristi se zbrajanje
  }
}
```

Kôd 11. Praćenje igrača usporedbom pozicija

U kôdu se uspoređuju pozicije igrača i *Tracker Glitcha* te ovisno o tome određuje u kojem smjeru će se *Tracker Glitch* pomicati. Korutina `ExplosionTimer` omogućuje *Tracker Glitchu* da se nakon određenog vremena praćenja igrača uništi (Kôd 12). Vrijeme je određeno metodom `WaitForSeconds` i njenim ulaznim parametrom.

```
private IEnumerator ExplosionTimer()
{
  yield return new WaitForSeconds(3);
  isAwake = false;
  animator.SetBool("Dead", true);
}
```

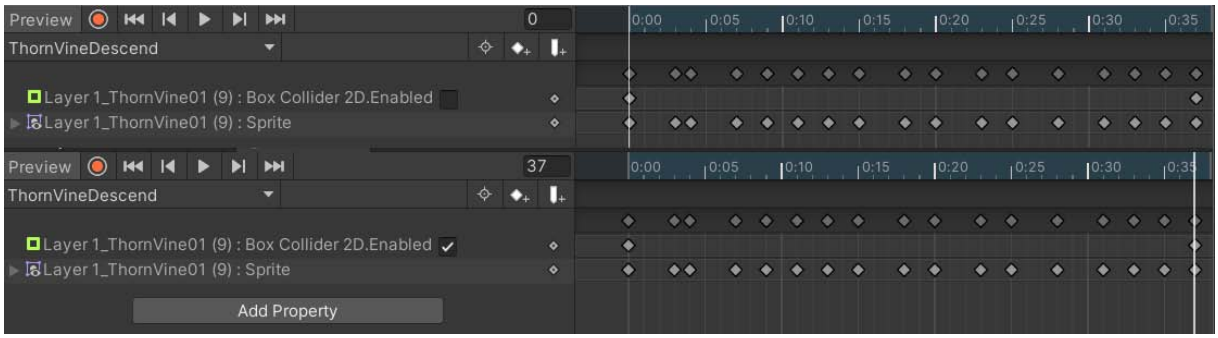
Kôd 12. Timer korutina

4.4.3. Thorn Vine

Thorn Vine predstavlja viseću trnovitu lijanu koja se spušta na igrača kada joj priđe. Ulaženjem igrača u lijanin prostor detekcije pokreće se animacija spuštanja.

```
>>> Animator.SetBool("inRange", true);
```

Pri završetku animacije se uključuje collider komponenta lijane, što joj omogućuje smanjivanje životnih bodova igrača pri sudaru s njim.



Slika 17. Uključivanje komponente Collider pomoću animacijskih keyframeova

4.4.4. Tree Boss

Kao zadnji neprijatelj na kraju nivoa, *Tree Boss* predstavlja personificirano drvo koje koristi prije spomenute trnovite lijane kako bi napalo igrača.

Isprva izgleda kao obično drvo, ali kad se igrač približi, pomoću metode `OnTriggerEnter2D` se pokreće animacija buđenja (Slika 18).

Slika 18. Usporedba spavajućeg i budnog stanja neprijatelja

Nakon buđenja kreće faza u kojoj drvo izvodi nasumične napade (Kôd 13).

```
if (isAwake)
{
  //...
}
else if (attackReady && !damagePhase)
{
  attackIndex = Random.Range(1, 4);
  switch(attackIndex)
  {
      //...
  }
}
```

Kôd 13. Nasumično generiranje indeksa napada

Generira se nasumični broj koji predstavlja indeks napada. Taj broj se stavlja u switch grananje kako bi se napadi nasumično odrađivali. Napadi su implementirani korištenjem metoda i animacijskih događaja (Kôd 14).

```
private void SlamAttack()
{
  attackCount += 1;
  attackReady = false;
  animator.SetBool("Slam", true);
  currentDamageValue = 2;
  knockback = Vector2.up * slamKnockupForce;
  StartCoroutine(AttackCooldown(2));
}
private void VineAttack()
{
  attackCount += 1;
  attackReady = false;
  animator.SetBool("vineAttack", true);
  StartCoroutine(AttackCooldown(3));
}
private void EveryOtherVine()
{
  attackCount += 1;
  attackReady = false;
  animator.SetBool("everyOtherVine", true);
  StartCoroutine(AttackCooldown(3));
}
```

Kôd 14. Metode individualnih napada

Na kraju animacije svakog napada uključuje se okidač (engl. *trigger*) koji igraču nanosi štetu na životne bodove, te se pokreće korutina `AttackCooldown`. Ona onemogućuje neprijatelju da napadne za vrijeme određeno ulaznim parametrom (Kôd 15).

```
private IEnumerator AttackCooldown(float cooldown)
{
  yield return new WaitForSeconds(cooldown);
  attackReady = true;
}
```

Kôd 15. Korutina za održavanje stanke između napada

```
private void Bite()
{
  attackReady = false;
  animator.SetBool("Bite", true);
  currentDamageValue = 3;
  if (player.transform.position.x < transform.position.x)
    knockback = Vector2.left * biteKnockbackForce;
  else
    knockback = Vector2.right * biteKnockbackForce;
  StopAllCoroutines();
  StartCoroutine(AttackCooldown(1));
}
```

Kôd 16. Izvedba blizinskog napada

Izvršenjem deset nasumično generiranih napada, što ne uključuje `Bite` napad (Kôd 16), neprijatelju je omogućen novi `Spit` napad (Kôd 17).

```
if (attackCount >= 10 && attackReady)
{
  StopAllCoroutines();
  damagePhase = true;
  bigVine.SetActive(true);
  bigVine.GetComponent<BigBossVine>().bombHit = false;
  Spit();
}
private void Spit()
{
  attackCount = 0;
  attackReady = false;
  animator.SetBool("Spit", true);
  StartCoroutine(AttackCooldown(4));
}
private void InitiateSpit()
{
  Vector2 spitDirection = (player.transform.position –
                           transform.position).normalized;
  GameObject spawnedBomb = Instantiate(circleBomb,
                                       transform.position,
                                       Quaternion.identity);
  spawnedBomb.GetComponent<Rigidbody2D>().AddForce(spitDirection *
                                        100, ForceMode2D.Impulse);
}
```

Kôd 17. Novi napad nakon deset izvršenih napada

`Spit` napadom neprijatelj izbaci bombu usmjerenu na igrača. Instanciranje bombe se odrađuje stvaranjem prefab objekta. Prefab je objekt koji nije sam aktivan u sceni, već služi kao nacrt za instanciranje objekata. Kreira se tako da se jednostavno povuče odabrani objekt iz prozora Hierarchy u prozor Project (Slika 19).

Slika 19. Primjer Prefab objekta

Nakon što neprijatelj izbaci bombu, ulazi u fazu mirovanja. Pojavljuje se veća verzija trnovite lijane koja će ciljati igrača. Igrač mora pokupiti tu bombu i odvesti je do neprijatelja. Primanje bombe je implementirano Kôd 18.

```
if(isCarrying)
{
  if (characterController.facingRight)
  {
    this.transform.position = new Vector3(
        player.gameObject.transform.position.x + 2,
        player.gameObject.transform.position.y-1, 0);
  }
  else
  {
    this.transform.position = new Vector3(
        player.gameObject.transform.position.x - 2,
        player.gameObject.transform.position.y - 1, 0);
  }
}
```

Kôd 18. Nošenje bombe

Nošenje je implementirano tako da bomba prati poziciju igrača. Nakon što igrač odnese bombu do neprijatelja, mora namamiti lijanu na stropu da se spusti na nju. Ako lijana dotakne bombu izvršava se sljedeći kôd (Kôd 19).

```
else if (collision.gameObject.CompareTag("Interactable"))
{
  collision.gameObject.GetComponent<CircleBomb>().Explode();
  if (Mathf.Abs(boss.transform.position.x- transform.position.x)
      <= 8)
  {
    bossAI.Damage();
  }
  isAttacking = false;
  attackOnCooldown = false;
  bossAI.damagePhase = false;
  bossAI.attackReady = false;
}
```

Kôd 19. Eksplozija bombe

Ako se interakcija između bombe i lijane dogodi u blizini neprijatelja, on prima štetu. Igrač zatim mora ponoviti isto četiri puta prije nego što je neprijatelj poražen i igra završena.

4.5. Spremanje igre

Implementiran je sustav u kojem igrač može spremiti svoj napredak tako da se željene vrijednosti spremaju u datoteku.

Za primjer će se koristiti podaci o igraču (Kôd 20).

```
[System.Serializable]
public class PlayerData
{
  public float[] position;
  public int maxHealth;
  public float maxStamina;
  public float speed;
  public float runSpeed;
  public float jumpForce;
  public float crouchSpeed;
  public bool doubleJumpUnlocked;
  public int shardsCollected;
  public PlayerData (Stats playerStats)
  {
    position = new float[3];
    position[0] = playerStats.gameObject.transform.position.x;
    position[1] = playerStats.gameObject.transform.position.y;
    position[2] = playerStats.gameObject.transform.position.z;
    //ostale vrijednosti inicijalizirane pridruživanjem
  }
}
```

Kôd 20. Klasa s podacima koji će se spremati

Atribut `System.Serializable` znači da se vrijednosti klase mogu spremati u datoteke. Konstruktorom se inicijaliziraju vrijednosti koje se žele spremiti.

Pozicija igrača (spremljena u varijablu position) je tipa `Vector3`. Taj tip se ne može izravno spremiti u datoteku, zato ga treba razbiti na komponente. `Vector3` je tip koji sprema tri vrijednosti: X, Y i Z koordinatu. Svaka zasebna koordinata je tipa float, dakle `Vector3` je moguće spremiti u float niz s duljinom tri.

Varijable se spremaju metodom `SavePlayer` (Kôd 21).

```
public static void SavePlayer(Stats playerStats)
{
  BinaryFormatter formatter = new BinaryFormatter();
  string path = Application.persistentDataPath + "/player.stats";
  FileStream stream = new FileStream(path, FileMode.Create);
  PlayerData data = new PlayerData(playerStats);
  Debug.Log(path);
  formatter.Serialize(stream, data);
  stream.Close();
}
```

Kôd 21. Metoda za spremanje vrijednosti

Pomoću klase `BinaryFormatter` se datoteka sprema u binarnom formatu, što otežava mogućnost manipulacije vrijednostima. Korištenjem Unity standardnog direktorija za spremanje podataka, kreira se tok kojim se podaci spremaju. Tok je nakon korištenja potrebno zatvoriti kako ne bi došlo do izmjene podataka.

```
public static PlayerData LoadPlayer()
{
  string path = Application.persistentDataPath + "/player.stats";
  if (File.Exists(path))
  {
    BinaryFormatter formatter = new BinaryFormatter();
    FileStream stream = new FileStream(path, FileMode.Open);
    PlayerData data = formatter.Deserialize(stream) as PlayerData;
    stream.Close();
    return data;
  }
}
```

Kôd 22. Metoda za učitavanje vrijednosti

Prvo se provjerava postoji li datoteka, ako postoji počinje postupak učitavanja datoteka u instancu klase PlayerData (Kôd 22).

4.6. Glavni izbornik

Izrada izbornika se radi pomoću UI objekata kao što su tipke, tekst i slike. Pri kreiranju UI objekta kreira se Canvas objekt koji služi kao roditeljski objekt za ostale UI objekte.

Interakcije u izbornicima rade se tipkama (*button*). One omogućuju zvanje određenih metoda (Slika 20), ili obavljanje jednostavnih radnji kao isključivanje i uključivanje objekata preko događaja *on click* (Slika 21).

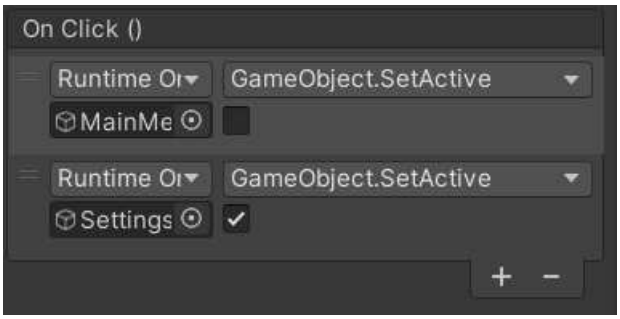
Slika 20. On click event

Pritiskom na tipku *Play* u sučelju poziva se metoda `Play` (Kôd 23).

```
using UnityEngine;
using UnityEngine.SceneManagement;
...
public void Play()
{
  SceneManager.LoadScene("Level1");
}
```

Kôd 23. Metoda `Play`

Za rad sa scenama potrebno je uključiti namespace `SceneManagement`.



Slika 21. Uključivanje i isključivanje objekata On click eventom

4.6.1. Sustav nadogradnje

Za održavanje nekog osjećaja napretka, igrač može poboljšati svoja svojstva: životne bodove, energiju potrebnu za akcije kao što su trčanje i skakanje te brzinu. Klikom na tipku pozvat će se sljedeće metoda `UpgradeHealth`, koja je odgovorna za poboljšanje željenog svojstva (Kôd 24).

```
public void UpgradeHealth()
{
  if(upgradedHealth < healthPoints.Length)
  {
    upgradedHealth += 1;
    playerStats.maxHealth = baseHealth + upgradedHealth;
    playerStats.health = playerStats.maxHealth;
    playerStats.shardsCollected -= 1;
  }
}
```

Kôd 24. Metoda `UpgradeHealth` za povećavanje životnih bodova

Sučelje za sustav poboljšanja prikazano je Slika 22.

Slika 22. Sučelje za sustav poboljšanja

5. ZAKLJUČAK

Izrada videoigre je dugotrajan proces, neovisno o engineu koji se koristi. Samostalni programeri prisiljeni su posjedovati znanje i iskustvo za više različitih alata. Osim samog korištenja enginea, moraju se upoznati s alatima za razvijanje ostalih aspekata igre (izgled, zvučni efekti, glazba u igri).

Izrada ovog praktičnog rada trajala je tek nekoliko mjeseci, što uključuje planiranje, dublje učenje korištenih programa, crtanje 2D spriteova, popravak te optimizaciju kôda. Važno je napomenuti da je, čak i nakon toliko provedenog vremena u izradi, ovo je još uvijek vrlo rana verzija videoigre s ciljem kasnijeg nadograđivanja. Izrada većih videoigara, koje imaju sadržaj dovoljan za više sati igranja, traje godinama.

Izrada ovog rada omogućila je uvid u razne dijelove Unityja i njegovih sustava kao što su rad s datotekama i spremanje vrijednosti u njih i optimizacija kôda. Ta iskustva bit će korisna pri nadograđivanju postojeće i izradi novih videoigara u budućnosti.

U procesu izrade ove videoigre naletilo se na nekoliko problema koji su utjecali na brzinu razvoja. Najveći problem, koji je opasan u industriji videoigara, a najviše za samostalne programere, je gubljenje motivacije i inspiracije. Gubitak motivacije i inspiracije uzrokuje provođenje vremena na jednom banalnom problemu, u slučaju ovog rada to je najčešće bio dizajn terena. Zato je tijekom izrade videoigre važno održati viziju krajnjeg proizvoda radi održavanja motivacije. Zbog vremena potrebnog za izradu videoigre, što uključuje dugotrajni rad na vizualnom dizajnu i kôdiranju, iznimno je važno uzimati i pauze.

Osim jasno istaknutih negativnih učinaka samostalnog rada, postoje jednako važni pozitivni učinci. Kao samostalni programer, nije potrebno održavati komunikaciju s ostalim članovima tima te razjasniti svoju ideju o rješavanju problema. Sve potrebne informacije i ideje nalaze se na jednom mjestu, tako da je nemoguće doći do greške u komunikaciji ili neslaganja.

Kôd izrade videoigre, važno je imati vanjski uvid u proizvod. Programer koji radi videoigru zna sve o njoj, stoga mu je nemoguće odrediti težinu i neke očite probleme. Povratne informacije bile su važne pri razvoju i modificiranju određenih pogleda videoigre.

Na kraju, iskustva stečena tijekom izrade ovog rada bit će neprocjenjiva za buduće projekte. Kroz učenje i primjenu raznih sustava Unity enginea i novih alata, postavljeni su temelji za razvoj složenijih i sadržajnijih igara u budućnosti.

Unatoč tome što je ova igra još uvijek u ranoj fazi razvoja, postignut je značajan napredak, a stečena znanja pružaju solidnu osnovu za daljnje usavršavanje i nadogradnju projekta. Izrada većih, komercijalno uspješnih igara zahtijeva godine rada i timski napor, no ovo iskustvo pokazalo je da je, uz posvećenost i kontinuirano učenje, moguće samostalno razviti kvalitetan i funkcionalan proizvod.

LITERATURA

1. Joe Hocking (2015), Unity in Action: Multiplatform Game Development in C#
2. Unity Scripting API, https://docs.unity3d.com/ScriptReference (zadnje pristupljeno 28. 8. 2024.)
3. Unity User Manual, https://docs.unity3d.com/Manual (zadnje pristupljeno 20. 8. 2024.)
4. Wikipedia, https://en.wikipedia.org/wiki/Unity_(game_engine) (pristupljeno 25. 8. 2024.)

SAŽETAK

Ovaj završni rad temelji se na analizi Unity enginea i mogućnosti njegove razvojne okoline, pružajući uvid u svaku komponentu potrebnu za razvoj videoigre. Spomenuti su vizualni stilovi i važnost njihovog učinka na razvoj, koje je potrebno znanje o dodatnim alatima te potrebna priprema prije početka razvoja. Korak po korak, korištenjem ugrađenih Unity klasa i metoda u jeziku C# rješavani su problemi vezani uz logiku igre i ponašanje objekata u okolini te ostalih problema, kao što su animacije i spremanje vrijednosti u datoteke korištenjem sustava koje pruža Unity.

Ključne riječi: Unity, videoigra, vizualni stilovi, jezik C#, animacije

SUMMARY

This thesis is based on the analysis of Unity engine and possibilities with its development environment, providing insight into every component needed for development of a videogame. It mentions visual style and the significance of their effect on development, needed knowledge of additional tools as well as necessary preparation before the start of development. Step by step, problems regarding the game logic and behavior of objects in the environment were solved with the usage of built in Unity classes and methods in C# language and other problems like animations and storing values into files using systems provided by Unity.

Keywords: Unity, videogame, visual style, C# language, animations