Ritmička akcijska igra sinkronizirana s ulaznim zvučnim zapisom

Čalušić, Darko

Master's thesis / Diplomski rad

2018

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:248649

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-04-26**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU

ELEKTROTEHNIČKI FAKULTET

Sveučilišni diplomski studij računarstva

RITMIČKA AKCIJSKA IGRA SINKRONIZIRANA SA ULAZNIM ZVUČNIM ZAPISOM

Diplomski rad

Darko Čalušić

Osijek, 2018 godina.

SADRŽAJ

1. UVOD 1
1.1. Zadatak završnog rada 1
2. GLAVNI DIO RADA 2
2.1. Opis razvojnog okruženja Unity 2
2.2. Izrada takt detektora 3
2.3. Izrada igre 6
2.3.1. Kreiranje avatara 6
2.3.2. Kreiranje ambijenta 9
2.3.3. Kreiranje glazbe i glazbenih efekata 12
2.3.4. Izrada nivoa za igranje 13
2.3.5. Kreiranje početnog zaslona 16
2.3.6. Kreiranje zaslona pobjede 18
2.3.7. Kreiranje zaslona gubitka 19
2.3.8. Kreiranje grafičkog korisničkog sučelja (GUI-a) 20
2.4. Testiranje igre 23
3. ZAKLJUČAK 26
LITERATURA 27
SAŽETAK 28
ABSTRACT 29
ŽIVOTOPIS 30
PRILOZI 31

1. UVOD

Industrija video igara je jedan od nabrže rastućih ekonomskih sektora na svijetu. Time što igre postaju sve više kompleksne, potreban je sve veći broj osoba za njihovo kreiranje. Međutim, postoje i manje zahtjevne igre koje spadaju u skupinu nezavisnih igara (engl. *Indie Games* – Independent video games). Igra opisana u ovome diplomskom radu spada u skupinu nezavisnih i akcijskih igara.

Akcijske igre predstavljaju žanr videoigre u kojima se naglašavaju fizičke sposobnosti igrača koristeći se koordinacijom ruku i vremenom reagiranja. Postoji veliki broj podgrupa ovog žanra, uključujući strategije, igre u areni, igre iz prvog lica i mnoge druge. Specifičnost ove igre zasniva se na sinkronizaciji sa ulaznim zvučnim zapisom koji dodatno dočarava njen ambijent. Igrač će kontroliranjem avatara, koji putuje kroz izmišljeni svijet, zaobilaziti različite prepreke u ritmu ambijentalne glazbe. Također, različiti objekti u igri će reagirati na ulazni zvučni zapis.

Diplomski rad se sastoji od definiranja kratkog zadatka završnog rada, glavnog dijela rada u kojemu je opisano razvojno okruženje – Unity, pravljenje takt (*engl.* beat) detektora, opisani elementi igre, nastanak pozadinske glazbe i glazbenih efekata, početnog zaslona, zaslona pobjede i drugih. Iza tog slijedi testiranje igre i zaključak.

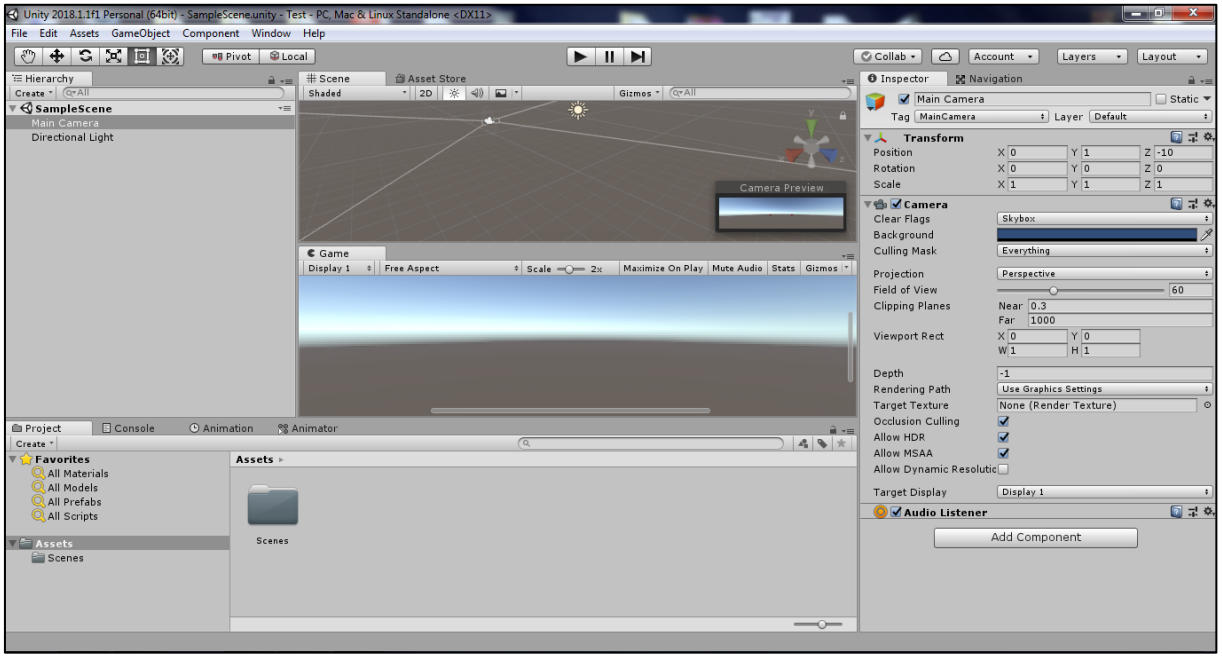
1.1. Zadatak završnog rada

Zadatak ovog završnog rada je realizacija ritmičke akcijske igre koja je sinkronizirana sa ulaznim zvučnim zapisom. Također je potrebno napraviti glazbu koja će biti korištena prilikom igranja.

2. GLAVNI DIO RADA

2.1. Opis razvojnog okruženja Unity

Unity je platforma za pravljenje igara razvijena od strane Unity Technologies. Ovo razvojno okruženje se primarno koristi za izradu 2D i 3D video igara i simulacija na različim uređajima poput računala, Android uređaja, iOS mobilnih uređaja, Play Station-a i mnogih drugih. Prva inačica programa izdana je 8.6.2005. za OS X i proširila se do sada na 27 platformi. Zadnja stabilna inačica je 2018.1.5. izdana 15.6.2018. koja je korištena za izradu projekta. Unity platforma napisana je u C++ i C# programskim jezicima. Postoje 4 licence za njegovo korištenje: *Personal*, *Plus*, *Pro* i *Enterprise*. U ovome radu korištena je *Personal* licenca bez ikakvih pogodnosti. Za pisanje igara potrebno je poznavati C# programski jezik kao i razvojno okruženje Unity-a.



Sl. 2.1.1 *Razvojno okruženje Unity-a*

2.2. Izrada takt detektora

Iz razloga što se igra opisana u ovom diplomskom radu temelji na ritmu glazbe, potrebno je napraviti algoritam i napisati odgovarajuću skriptu koja otkriva kada dođe do takta u pjesmi. Takt se u glazbi definira kao osnovna jedinica vremena. Ponavljanje taktova u pjesmi se karakterizira kao ritam pjesme. Metrički gledano, u decibelima (dB – jedninici za mjerenje glasnoće zvuka), kada se dogodi takt, dođe do naglog porasta, što je ključno za pravljenje gore navedene skripte. Glavna logika ovog algoritma se sastoji u uspoređivanju trenutne energije zvuka sa lokalnom energijom zvuka. Ukoliko je trenutna energija zvuka veća od lokalne energije zvuka, tada se smatra da se dogodio takt. Prije svega potrebno je poznavati da postoji više vrsta uzorkovanja prilikom snimanja. Neki od njih su: 44100 Hz, 48000 Hz, 96000 Hz i drugi. Dane veličine predstavljaju tipove gdje se zvuk uzorkuje onoliko puta u sekundi kolika je frekvencija uzorkovanja. Naprimjer, kod 44100 Hz, zvuk se uzorkuje 44100 puta u sekundi gdje svako uzorkovanje može biti spremljeno u 8, 16 i 24 bitne vrijednosti. Pjesma napravljena za ovaj diplomski rad snimljena je u 44100 Hz u 16 bitne vrijednosti. U nastavku slijedi razrada algoritma.

$$e = e_{stereo} = e_{left} + e_{right} = \sum_{k=i_0}^{i_0+1024} a[k]^2 + b[k]^2 \qquad (2\text{-}2\text{-}1)$$

Trenutna energija zvuka (**e**) predstavlja sumu lijevog i desnog kanala, gdje 1024 predstavlja broj uzorkovanja koje obrađujemo a varijabla **i** predstavlja mjesto uzorkovanja. Trenutna energija zvuka u ovome slučaju predstavlja period od 0.023 sekunde (1024 / 44100). U Unity-ju je korištena integrirana funkcija za izračunavanje amplituda za oba kanala (Sl. 2.2.1) – a[k] i b[k] (a predstavlja polje vrijednosti amplituda za lijevi kanal a b za desni).

```
public void GetSpectrumData(float[] samples, int channel, FFTWindow window);
```

Sl. 2.2.1 *Integrirana funkcija Unity-ja za izračun amplitude zvuka*

gdje je:

-samples: polje/broj za popunjavanje uzoraka (treba biti jednak potenciji broja 2),
-channel: kanal iz kojeg se popunjavaju uzorci; desni, lijevi,
-window: smanjenje propuštanja signala iz frekvencijskih pojaseva.

$$< E >= \frac{1}{43} * \sum_{i=0}^{43} (E[i])^2 \qquad (2\text{-}2\text{-}2)$$

Prosječna lokalna energija (**<E>**) predstavlja sumu polja trenutne energije zvuka, gdje je E[0] najnovija izračunata energija zvuka od trenutna 1024 uzorka (u ovome slučaju **e** – trenutna energija zvuka), a E(42) najstarije izračunata energiju zvuka (u 43 iteracije). Razlog što se koriste 43 iteracije je taj što se prilikom izračuna trenutne energije zvuka mora koristiti potencija broja 2, što je 1024, a za lokalnu energiju zvuka se uzima vrijednost jedne sekunde, što su 44100 uzorka. Stoga u 44100 uzorka može stati 43 uzorka trenutne energije zvuka, što daje: 43 * 1024 = 44032 što je približno jednoj sekundi (prosječnoj lokalnoj energiji zvuka). U Unity-u je to urađeno na sljedeći način:

```
float E = sumLocEnergy ()/his_Buff.Length;
```

Sl. 2.2.2 *Računanje prosječne lokalne energije*

```
float[] shiftingHistoryBuffer = new float[his_Buff.Length];

for (int i = 0; i<(his_Buff.Length-1); i++) {
        shiftingHistoryBuffer[i+1] = his_Buff[i];
}

shiftingHistoryBuffer [0] = e;

for (int i = 0; i<his_Buff.Length; i++) {
        his_Buff[i] = shiftingHistoryBuffer[i];
}
```

Sl. 2.2.3 *Polja za spremanje trenutnih energija zvuka*

```
float sumLocEnergy() {
        float E = 0;

        for (int i = 0; i<his_Buff.Length; i++) {
                E += his_Buff[i]*his_Buff[i];
        }
        return E;
}
```

Sl. 2.2.4 *Funkcija sume polja trenutne energije zvuka*

Prilikom uspoređivanja trenutnog intenziteta zvuka sa prosječnim, može doći do greške ukoliko se dogode nagle promjene u samom intenzitetu zvuka. Stoga je potrebno uvesti konstantu koja će utvrditi osjetljivost samog algoritma. Ta konstanta nije ista za sve vrste glazbe i potrebno je napraviti algoritam koji će prilagoditi konstantu u ovisnosti o intenzitetu energije zvuka

$$V = \frac{1}{43} * \sum_{i=0}^{43} (E[i] - < E >)^2 . \quad (2\text{-}2\text{-}3)$$

Varijanca energija (**V**) predstavlja prosjek sume razlike trenutne energije zvuka i prosječne lokalne energije. Time je se dobilo da što je varijanca veća, algoritam postaje osjetljiviji.

$$C = (-0.0025714 * V) + 1.5142857 \quad (2\text{-}2\text{-}4)$$

Naposlijetku, konstanta **C** je dobivena pravljenjem linearne regresije na osnovu najbolje prilagođenih vrijednosti. Što je varijanca veća, konstanta postaje manja i obrnuto.

$$e > C * < E > \quad (2\text{-}2\text{-}5)$$

Detekcija takta se otkriva u posljednjem koraku. Ukoliko je trenutna energija zvuka (**e**) veća od prosječne energije zvuka pomnožena sa konstantom (**<E> * C**), dolazi do detekcije takta. U samom kodu su urađene dodatne promjene kako bi se algoritam usavršio. Uvedena je nova konstanta koja sprječava detekciju na određeno vrijeme iz razloga što su se taktovi pojavljivali u jako intenzivnim vremenima zbog čega je u ovome slučaju dolazilo do neželjenih pojava.

2.3. Izrada igre

2.3.1. Kreiranje avatara

Nakon što je odabrana svemirska tema igre, potrebno je napraviti avatara kojega kontroliramo. Izgled avatara je preuzet sa internet stranice https://itch.io/.

Sl. 2.3.1 *Izgled avatara*

Kao što je napomenuto u uvodu, igra testira refleksivne sposobnosti igrača. Avatar će tijekom čitavog nivoa biti u pokretu, stoga je potrebno napraviti potrebne animacije. U besplatnom programu paint.net napravljene i dovršene su dodatne potrebne animacije za: trčanje, letenje, hodanje, skakanje i klizanje po tlu. U Unity-ju animacije se prave iz *sprite* animacijske liste (Sl. 2.3.2).

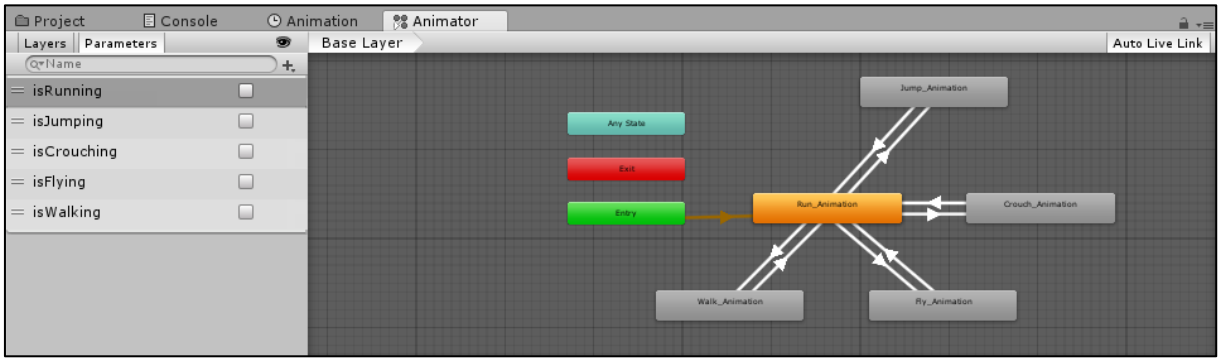
Sl. 2.3.2 *Sprite animacijska lista skoka*

Nakon ubacivanja napravljenog *sprite-a* animacije u Unity-u, potrebno je napraviti potrebnu animaciju. Animacija se pravi klikom na prozor animacije.

Sl. 2.3.3 *Postupak pravljenja animacije*

Odgovarajuću animaciju pravimo klikom na dugme *Add Property*. Zatim imenujemo animaciju i spremamo na željeno mjesto. Nakon toga u prozor animacije povlačimo odgovarajuće slike na mjesto u vremenskoj domeni koje smo prethodno obradili u Unity-u. Pregledavanje dovršene animacije se vrši dugmetom *Play* u animacijskom prozoru.

Poslije dovršenih animacija, potrebno napraviti parametre na osnovu kojih će se odgovarajuća pokretati. To se vrši klikom na *Window -> Animator* u Unity-u.



Sl. 2.3.4 *Prozor animatora*

Dodavanje parametara se vrši klikom na dugme +. U igri je dodano 5 *boolean* parametara: *isRunning*, *isJumping*, *isCrouching*, *isFlying* i *isWalking*. Na osnovu prijelaza napravljeno je aktiviranje određenih animacija. Pristup spomenutim parametrima se vrši nakon dohvaćanja *Animator* komponente na *Player* objektu. Dohvaćanje komponenti se vrši u funkciji Start koja se pokreće prilikom pokretanja određene scene.

```
anim = GetComponent<Animator> ();
```

Sl. 2.3.5 *Dohvatanje animator komponente*

Pristup boolean parametrima se odvija pomoću funkcije SetBool.

```
public void SetBool(string name, bool value);
```

Sl. 2.3.6 *SetBool funkcija*

gdje je:

-name: ime animacije koju želimo pokrenuti,
-value: *bool* parametar (*true* / *false*).

Kontroliranje avatara napravljeno je u klasi *CharacterController2D* funkcijom *Move()*.

```
public void Move(float move, bool crouch, bool jump, bool fly, bool walk)
```

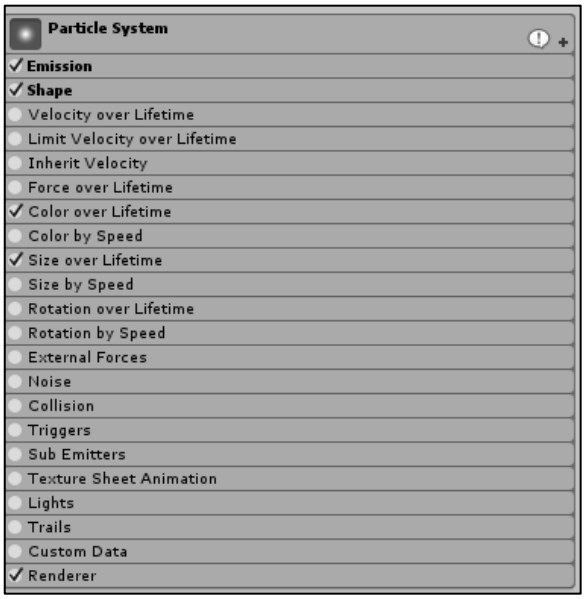
Sl. 2.3.7 *Move funkcija*

Pozivanjem spomenute funkcije u integriranoj funkciji *FixedUpdate*, omogućava se izvršavanje *Move()* funkcije onoliko puta koliko se izvršavaju fizikalne veličine. Time je osigurano da ukoliko se igra pokrene na računalima visokih performansi, neće doći do bržeg izvršenja određenih funkcija. Ukoliko bi se funkcija *Move()* pozivala u funkciji *Update*, tada bi se izvršavala svaki frame zbog čega dolazi do bržeg izvršavanja na računalima viskokih performansi.

Određenom logikom u kodu klase *PlayerMovement* i funkcije *Move*, napravljeno je kontroliranje avatara i odgovarajućih animacija kako je unaprijed bilo definirano.

2.3.2. Kreiranje ambijenta

Iz razloga što se radnja igre odvija u svemiru, potrebno je napraviti simulaciju zvijezda. Simulacija je napravljena uz pomoć *Particle Systema* u Unity-u (Sl. 2.3.8).



Sl. 2.3.8 *Particle System u Unity-u*

U odjeljku *Emission* namješta se ponašanje kao i duljina trajanja emitiranja čestica. U odjeljku *Shape* definirano je emitiranje čestica u obliku kocke i postavljen je odgovarajući broj čestica. *Color over Lifetime* i *Size over Lifetime* su namješteni po vlastitom izboru zbog ljepšeg prikazivanja. U *Renderer* odjeljku se postavlja se materijal emitiranja. Nakon podešavanja svih parametara dobiveno je ponašanje kao na sljedećoj slici (Sl. 2.3.9).

Sl. 2.3.9 *Particle System pozadine*

Particle System pozadine je nakon toga postavljen kao *child* objektu *Player* kako bi se naslijedio položaj i dočarao efekt hodanja kroz svemir.

Podloga po kojoj se avatar kreće predstavlja 3D objekte. Postavljeno je da se prilikom detekcije takta, materijal objekta mijenja boju. Takvo ponašanje napravljeno je pomoću *UnityEvent-a*.

```
public ChangeColourHandler onBeatChange;
```

Sl. 2.3.10 *Definiranje UnityEvent-a*

Na slici (Sl. 2.3.10) prikazano je definiranje varijable tipa Event.

```
[System.Serializable]
public class ChangeColourHandler : UnityEngine.Events.UnityEvent
{

}
```

Sl. 2.3.11 *Definiranje klase UnityEvent-a*

Nakon napisane klase, potrebno je navedeni *Event* aktivirati. To se postiže naredbom *Invoke()*. Zatim je potrebno napisati klasu koja će mijenjati boju pripadajućeg objekta prilikom takta.

```
PlayerMovement changeCol = FindObjectOfType<PlayerMovement> ();
changeCol.onBeatChange.AddListener (changeObjectColour);
```

Sl. 2.3.12 *Definiranje osluškujuće funkcije na onBeatChange*

Navedenim dijelom koda (Sl. 2.3.12) postignuto je da izvršavanje koda u funkciji imena *changeObjectColour* bude sinkronizirano sa detekcijom takta.

```
void changeObjectColour(){
    rend.material.color = new Color (r,g,b);
}
```

Sl. 2.3.13 *Funkcija changeObjectColour*

gdje je:

-*rend*: Dohvaćena renderer komponenta pripadajućeg 3D objekta,

-r, g, b: Random vrijednosti između 0 i 1.

Izvršavanje određenih radnji kontrolirano je pomoću *2DboxCollider-a*. Radi olakšanog igranja napravljeno je da prilikom detekcije takta, varijabla zaslužna za ispitivanje vrijednosti i pritiska određenog gumba na tipkovnici, produlji svoju vrijednost za 0.2s. Osim što je produljena vrijednost takta, produljena je i varijabla za ispitivanje pritiska gumba za skakanje. Time je omogućeno da igrač ima 0.2 sekunde da reagira na određenu pojavu takta. Uvjet koji treba biti ispunjen prilikom određenih radnji je:

- Avatar se mora nalaziti u *2DboxCollider-u*,
- Mora se dogoditi takt,
- Igrač mora pravovremeno reagirati.

Radnje koje su sinkronizirane sa taktovima u igri su:

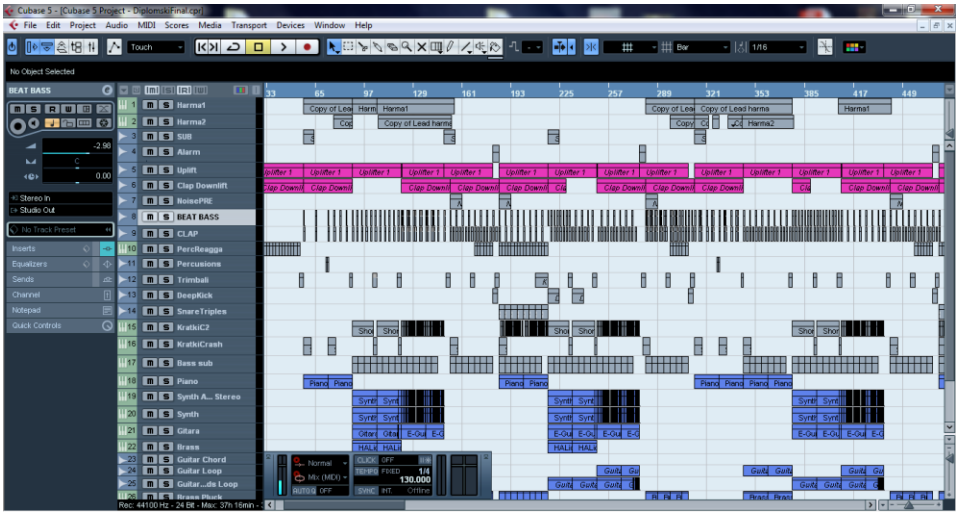
- Skakanje,
- Klizanje po tlu,
- Letenje,
- Hodanje,
- Pritisak strelica (gore, dolje, lijevo i desno),
- Mijenjanje boje 3D objekata.

2.3.3. Kreiranje glazbe i glazbenih efekata

Svi glazbeni resursi potrebni za realizaciju igre rađeni su u naprednom glazbenom programu Cubase 5.

Sl. 2.3.14 *Grafičko sučelje Cubase 5.*

Nakon što je odabrana tema igre, potrebno je napraviti i snimiti sve glazbene resurse, počevši od početnog zaslona, zaslona kraja i glazbe koja prati tijek igre. S obzirom da je ova igra ambijentalnog karaktera, potrebno je napraviti pozadinsku glazbu koja je visoke kvalitete i priliči prethodno izrađenom nivou igre. Glazba koja je snimljena za glavni nivo ima tempo od 130 BPM (engl. *beats per minute,* Sl. 2.3.15). Spada u skupinu EDM pjesama. Sastoji se od 26 različitih instrumenata. Na isti način su snimljene pozadinske pjesme za početni zaslon, zaslon pobjede i zaslon gubitka.



Sl. 2.3.15 *Sadržaj pjesme glavnog nivoa*

2.3.4. Izrada nivoa za igranje

Prvi korak izrade odgovarajućeg nivoa sastoji se u pronalasku položaja avatara u trenutcima u kojima je ritam naglašen. Kako bi ovo bilo moguće, u skripti je potrebno pristupiti komponenti *Transform* koja posjeduje položaj koordinata.

```
player = GameObject.Find ("Player").GetComponent<Transform> ();
```

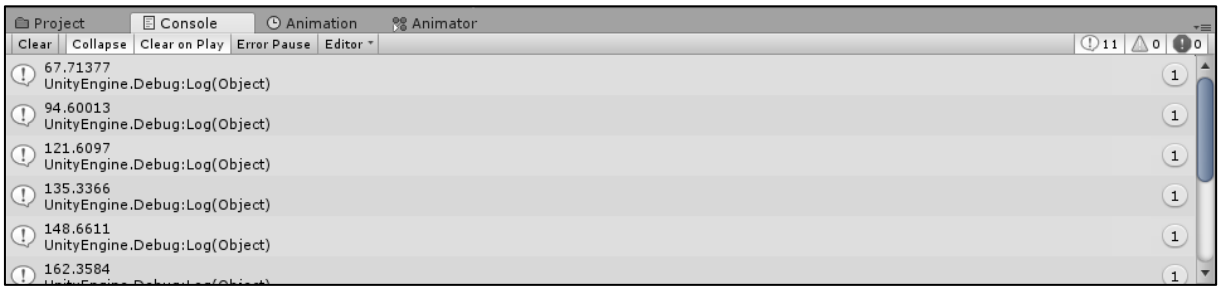
Sl. 2.3.16 *Pristup komponenti Transform*

Na prethodnoj slici (Sl. 2.3.16) je prikazano pristupanje komponenti Transform objekta pod nazivom Player. Ukoliko se želi pristupiti odgovarajućoj komponenti željenog objekta, potrebno je upisati ime objekta pod navodnicima i pod komponentom upisati ime željene komponente. Tada pozivom na player.pristup, gdje je pristup željena varijabla napisane komponente, pristupamo odgovarajućim podatcima. U PlayerMovement skripti, u funkciji koja osluškuje događaj taktova, potrebno je dodati sljedeću liniju koda (Sl. 2.3.17).

```
Debug.Log (player.position.x);
```

Sl. 2.3.17 *Ispisivanje x položaja player-a*

Nakon toga je potrebno postaviti objekte po kojima će se avatar kretati. Kako bi se radnja napravila dinamičnijom, napravljeni su odjeljci u kojima će avatar iz trčanja preći u hodanje i to u onim mjestima gdje glazba ima manju dinamiku. To je postignuto slušanjem glazbe i bilježenjem određenih x koordinata. Upravo iz razloga što je horiontalna brzina avatara konstantna (osim u slučaju kad hoda), postavljena je podloga po kojoj se kreće i računate x koordinate u ritmu.



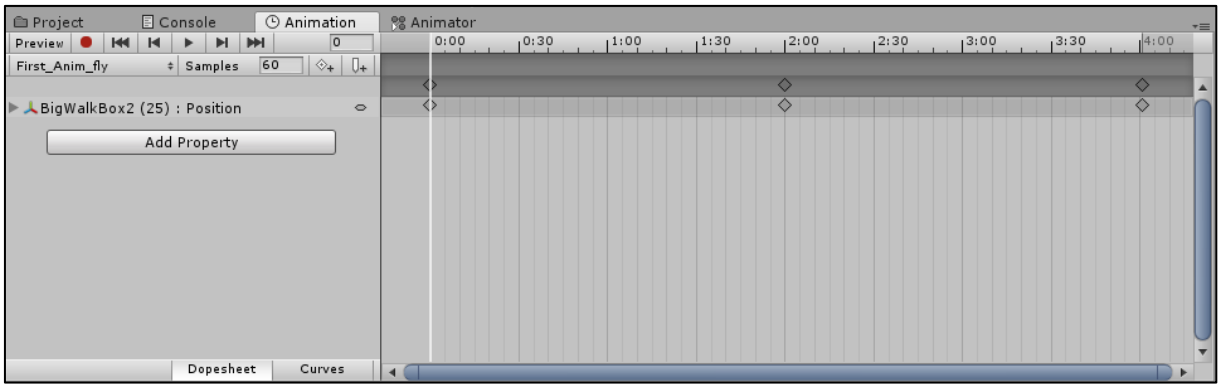
Sl. 2.3.18 *Ispisane x koordinate položaja avatara*

Nakon što su koordinate zabilježene (Sl. 2.3.18) i napravljena dinamika nivoa (na osnovu dinamike pjesme), potrebno je postaviti prethodno napravljene objekte koje će igrač morati zaobilaziti. Napravljeni objekti spremljeni su u *prefabs* u Unity-ju. Oni se ponašaju kao *template* za kreiranje novih instanci objekta u sceni.

Kreirani template u sceni igranja su:

1. Početak hodanja,
2. Kraj hodanja,
3. Odjeljak klizanja po tlu,
4. Početak letenja,
5. Kraj letenja,
6. Strelica gore,
7. Strelica dolje,
8. Strelica lijevo,
9. Strelica desno,
10. Kraj nivoa,
11. Odjeljak za skakanje,
12. Objekt po kojemu se kreće,
13. Prepreke.

Spomenuti objekti se zatim postavljaju u ritmu glazbe. Kako bi se igra napravila još dinamičnijom, postavljen je dio pokretnih prepreka prilikom letenja. Pokretne prepreke su napravljene pomoću Unity animacija. Ukolko se želi animirati 3D objekt, potrebno ga je označiti.



Sl. 2.3.19 *Ispisane x koordinate položaja avatara*

Nakon označavananja potrebno je kliknuti dugme *Record* u prozoru animacije (Sl. 2.3.19) i imenovati animaciju.. Od toga trenutka je objekt u stanju snimanja animacije. U ovome slučaju je potrebno animirati poziciju objekta. To je učinjeno na način da se na vremenskoj liniji označi vrijeme gdje se objekt treba nalaziti i pomjeri objekt. Na taj način je napravljena animacija trenutnog objekta.

Igra posjeduje opciju pauze. Pritiskom gumba P na tipkovnici aktivira se zaslon pauze (Sl. 2.3.20) sa ponuđenim izbornikom.

Sl. 2.3.20 *Zaslon pauze*

Kako bi se zaustavila igra, potrebno je zaustaviti vrijeme u igri. To se potiže sljedećom linijom koda (Sl. 2.3.21).

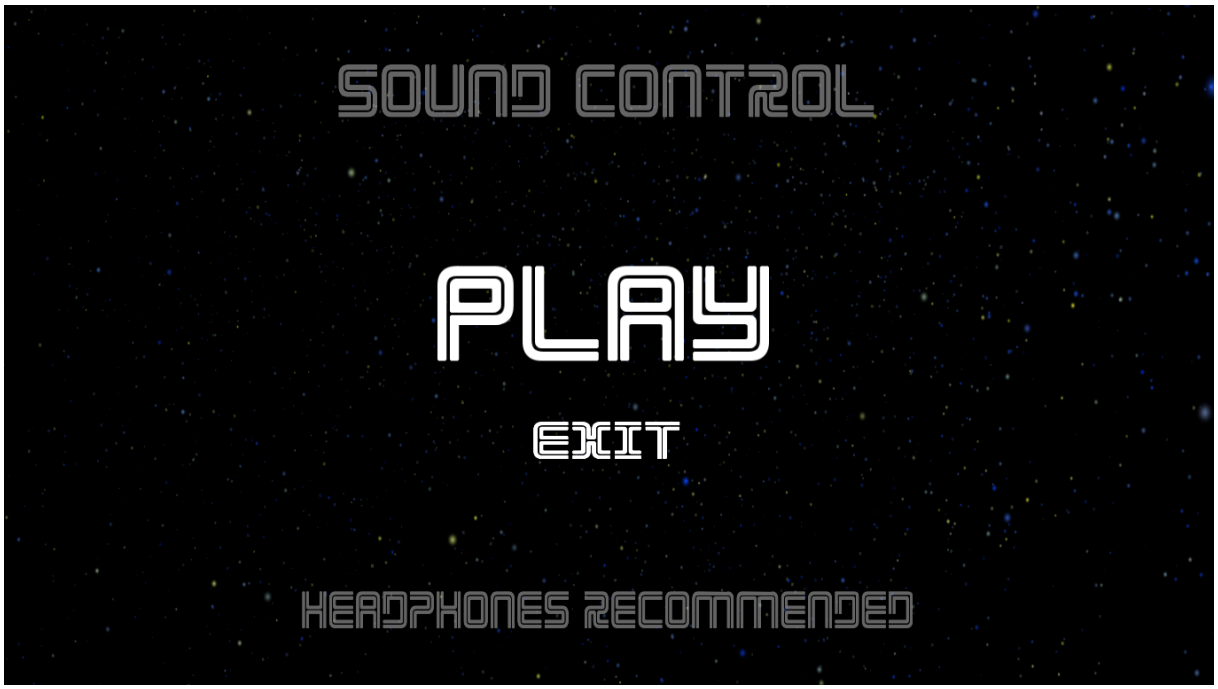
```
Time.timeScale = 0;
```

Sl. 2.3.21 *Zaustavljanje vremena u igri*

Međutim, ta naredba ne zaustavlja zvuk te se zvuk reproducira normalnim tokom. Kako skripta za detekciju takta radi na osnovu reproduciranog zvuka, potrebno je zvuk zaustaviti kako bi se mogao nastaviti reproducirati pri nastavljanju igre i nastaviti detektirati taktove. Dodana je komponenta tipa *AudioSource* koja reproducira glavnu pozadinsku pjesmu i koja se pauzira tijekom aktiviranja zaslona pauze.

2.3.5. Kreiranje početnog zaslona

Prilikom ulaska u igru otvara se početni zaslon. Pozadina ovog zaslona je animirana na način kao na slici (Sl. 2.3.8) uz uključenu komponentu *Noise*. Ova komponenta dodaje turbulentno strujanje čestica. U podopcijama Strength i Frequency namješta se odgovarajuća brzina i snaga turbulencije. Izgled početnog zaslona je kao na sljedećoj slici (Sl. 2.3.22).



Sl. 2.3.22 *Izgled početnog zaslona*

Na početnom zaslonu nalaze se dvije komponente tipa *Button* – Play i Exit. Dugme Play je animirano na način da se u padajućem izborniku komponente *Button* opcija *Transition* postavi na *Animation*. Zatim se klikom na dugme *Visualize* otvara prozor imenovanja animacije. Nakon toga se pravljenje animacije vrši na način kao što je objašnjeno na slici Sl. 2.3.3.

Gumb Exit je postavljen na opciju *Color Tint* u padajućem izborniku.

Svi prijelazi iz scena rađeni su sljedećim dijelom koda gdje varijabla N predstavlja redoslijed scene koja se nalazi u izborniku *File -> Build Settings* (Sl. 2.3.23).

```
public void LoadLevel_N(){
    Application.LoadLevel (N);
}
```

Sl. 2.3.23 *Dio koda za učitavanje odgovarajuće scene*

Nakon učitavanja početnog zaslona i potvrđivanja igranja pritiskom dugmeta *Play*, otvara se zaslon učitavanja nivoa. Za vrijeme trajanja ovog zaslona učitava se predani nivo. Učitavanje scene predstavlja asinkronu funkciju gdje Unity učitava sve potrebne resurse i objekte u pozadini izvršavanja niti.

```
IEnumerator LoadNewScene() {
    yield return new WaitForSeconds(2);
    AsyncOperation async = Application.LoadLevelAsync(scene);
    while (!async.isDone) {
        yield return null;
    }
```

Sl. 2.3.24 *Nit za učitavanje nivoa*

gdje je:

-LoadNewScene(): Funkcija učitavanja nivoa,

-LoadLevelAsync(): Asinkrono učitavanje nivoa,

-scene: Varijabla tipa integer.

2.3.6. Kreiranje zaslona pobjede

Nakon uspješno završenog nivoa, otvara se zaslon pobjede. Na njemu se nalaze dvije komponente tipa *Button – Main menu* i *Exit*. Ptitiskom na dugme Main menu vraća se početni zaslon, dok pritisak na dugme Exit izlazi iz igre.

Sl. 2.3.25 *Zaslon pobjede*

Na ovom zaslonu, nakon osvojenih bodova prilikom igranja, ispisuje se rezultat. Varijabla rezultata se nalazi u skripti *PlayerMovement*. Bodovi na zaslonu pobjede su ispisani na način da je varijabla *score* definirana kao *static*.

```
{class name}.{static member name}
```

Sl. 2.3.26 *Pristupanje static varijablama*

Na slici iznad (Sl. 2.3.26) napisana je sintaksa pristupa varijablama tipa *static*. Iz razloga što je spomenuta varijabla score tipa *integer*, potrebno ju je parsirati u tip *string*. To se postiže dodavanjem integrirane funkcije na kraj pristupa varijable - *.ToString()*;.

2.3.7. Kreiranje zaslona gubitka

Ukoliko *Rewind* varijabla dođe do nule, pojavljuje se zaslon gubitka (Sl. 2.3.27). Ovaj zaslon posjeduje tri komponente tipa Button – *Try Again?, Main menu i Exit*. Pritiskom na dugme *TryAgain?* otvara se ponovno učitavanje nivoa. *Main menu* pokreće početni zaslon, dok pritisak na dugme *Exit* izlazi iz aplikacije.

Sl. 2.3.27 *Izgled zaslona gubitka*

2.3.8. Kreiranje grafičkog korisničkog sučelja (GUI-a)

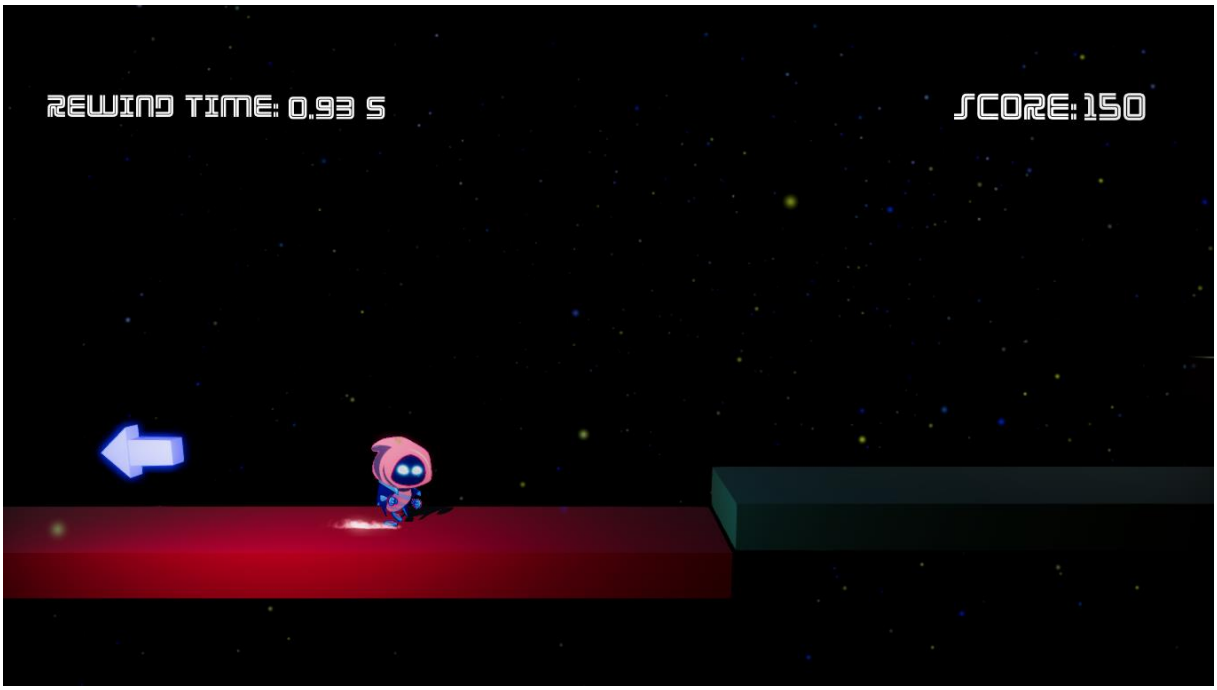
Cilj napravljenje igre je postizanje što većeg broja bodova. Bodovi se skupljaju na način da se izvršavaju određene radnje. Na kraju svake uspješno izvršene radnje se dobije određeni broj bodova. Broj bodova se nalazi u gornjem desnom kutu zaslona igranja (Sl. 2.3.29). U Unity-ju je to odrađeno na način da je dodana komponenta tipa *Text* u *Canvas* scene za igranje.

```
public void addScoreUp(){
    scoreScore += 10;
}
```

Sl. 2.3.28 *Funkcija povećavanja bodova*

Pozivom funkcije *addScoreUp()* (Sl. 2.3.28), povećavaju se bodovi. Funkcija se poziva na kraju uspješno izvršene radnje klizanja po tlu, na kraju uspješno izvršenog skakanja, letenja i pritiska na označene strelice. Varijabla *scoreScore* se ispisuje na grafičkom sučelju.

Kako bi se omogućio oporavak od krivo i netočno izvršene radnje, dodana je funkcija vraćanja vremena. Dopušteno, odnosno preostalo vrijeme vraćanja vremena se nalazi u gornjem lijevom kutu (Sl. 2.3.29).



Sl. 2.3.29 *Izgled korisničkog sučelja*

Vraćanje vremena je napravljeno na način da se prilikom svakog pozivanja funkcije *FixedUpdate* sprema položaj avatara koji se kontrolira i trenutno vrijeme pjesme. Spremanje je ograničeno na dani broj sekunda, što je u ovome slučaju postavljeno na četiri sekunde. Spremanje se vrši u komponentu tipa *List*. Kako bi se omogućio rad sa varijablom tipa *List*, mora se u zaglavlje dokumenta dodati *using System.Collections.Generic;*

```
public PointsInTime (Vector3 _position, float _songTime)
{
    position = _position;
    songTime = _songTime;
}
```

Sl. 2.3.30 *Funkcija PointsInTime*

gdje je:

-position: varijabla u koju se sprema položaj avatara

-songTime: varijabla u koju se sprema vrijeme pjesme

```
void Record (){

    if (pointsInTime.Count > Mathf.Round (recordTime / Time.fixedDeltaTime)) {
        pointsInTime.RemoveAt (pointsInTime.Count - 1);

    }

    pointsInTime.Insert (0, new PointsInTime(transform.position , (float)audioSong.time));

}
```

Sl. 2.3.31 *Funkcija snimanja podataka (Record)*

Funkcija snimanja je napravljena na FIFO načinom upravljanja memorijom (Sl. 2.3.31). Svako pozivanje navedene funkcije u *FixedUpdate* će spremiti položaj avatara i vrijeme pjesme na nultu poziciju u polju, dok će ostale pomjeriti za jedno mjesto unaprijed. Kada se spremi onoliko podataka u polje koje je veće od postavljenog vremena vraćanja, spremljeni podaci se brišu.

Vraćanje vremena se poziva pritiskom tipke R ili pogrešnim kontroliranjem avatara.

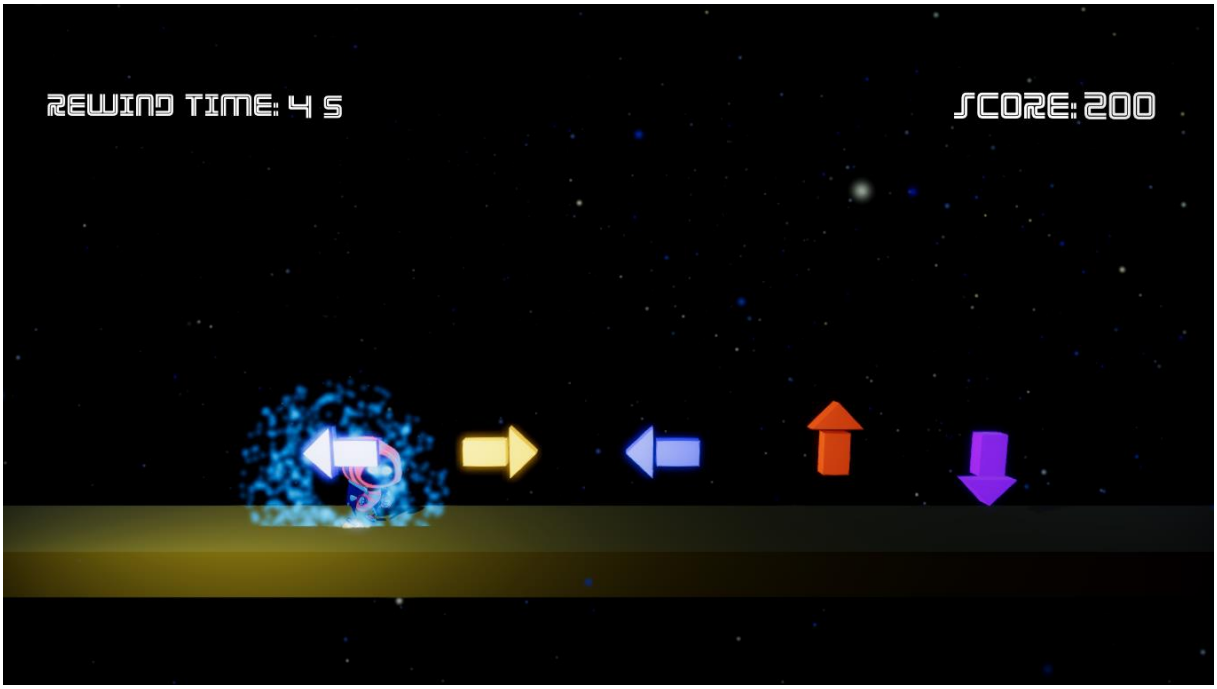
```
void Rewind(){

    if (pointsInTime.Count > 0) {

        PointsInTime pointInTime = pointsInTime[0];
        transform.position = pointInTime.position;
        audioSong.time = pointInTime.songTime;
        pointsInTime.RemoveAt (0);

    } else {

        StopRewind ();
    }
}
```

Sl. 2.3.32 *Funkcija vraćanja podataka (Rewind)*

Funkcija *Rewind* je zaslužna za vraćanje spremljenih podataka (Sl. 2.3.32). Ukoliko postoji određeni broj elemenata polja, koji je pri tome veći od nula, tada se spremljene varijable položaja avatara i vremena pjesme pridodaju trenutnom položaju avatara i vremenu pjesme. To se radi na način da se nultom (zadnje spremljenom) podatku pristupi, pridodaju se podaci i taj isti podatak se briše. Nakon toga se postupak ponavlja se dok se polje u potpunosti ne isprazni. Ukoliko je polje prazno, poziva se funkcija zaustavljanja vraćanja i nastavljanja igre normalnim tokom. Prilikom vraćanja vremena, potrebno je *Player* komponenti *Rigidbody2D* promijeniti tip u *Kinematic*. Kada je objektu postavljen tip u *Kinematic*, na njega ne djeluju nikakve implementirane sile, već se isključivo kontrolira putem skripte. Nakon što je uspješno izvršeno vraćanje, ponovno se postavlja tip na *Dynamic*.

2.4. Testiranje igre

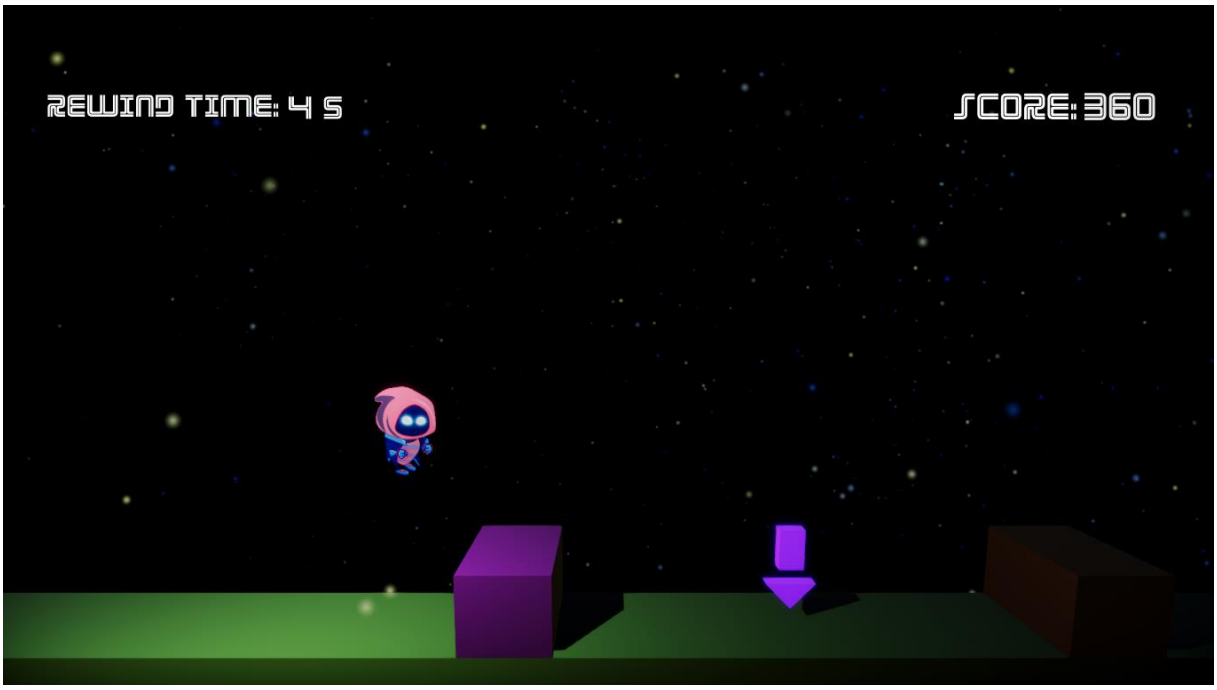
Nakon uspješno napravljene igre, slijedi testiranje.



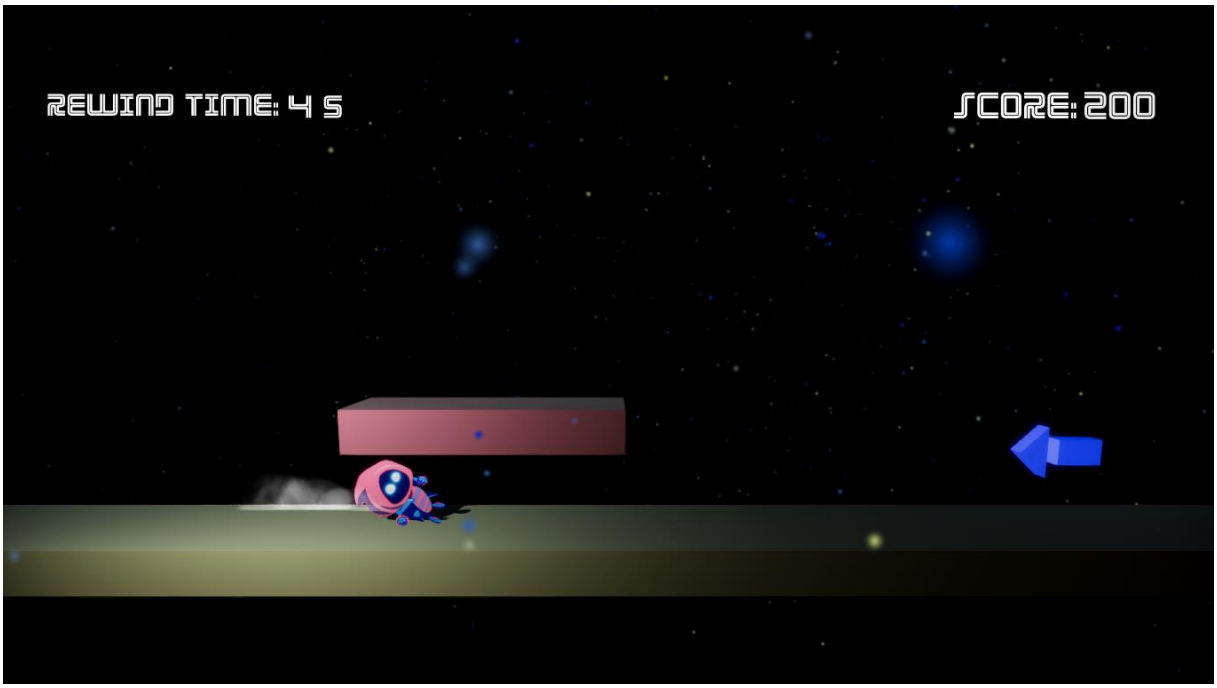
Sl. 2.4.1 *Uspješno pritisnuta strelica u lijevo*

Sl. 2.4.2 *Prolazak područja letenja*

Na prethodnim slikama (Sl. 2.4.1 i Sl. 2.4.2) vidljivo je kontroliranje avatara. Ukoliko je označena strelica ispravno i pravodobno stisnuta, reproducira se napravljeni efekt. Efekt je napravljen u *Particle System editoru* (Sl. 2.3.8) uz odgovarajuće izmjene. Također je i napravljen efekt hodanja i trčanja uz prethodno napravljenoj animaciji prašine.



Sl. 2.4.3 *Pravovremena izvršena akcija skakanja*



Sl. 2.4.4 *Ispravno izvršeno područje klizanja*

Na slikama (Sl. 2.4.3 i Sl. 2.4.4) prikazane su ispravne i pravovremene potrebne akcije. Prilikom klizanja po tlu, potrebno je držati dugme C do kraja napravljenog područja klizanja.

Sl. 2.4.5 *Uspješno izvršen napravljeni nivo*

Ukoliko se uspješno izvrši napravljeni nivo, potrebno je da se otvori zaslon na pobjedu sa napisanim postignutim rezultatom što je uspješno napravljeno. Pritiskom na dugme Exit, kojemu se može dodatno pristupiti u pauziranoj igri, izlazi se iz igre što je uspješno testirano.

3. ZAKLJUČAK

Igra se temelji na ritmu i sposobnosti igrača da procijeni i reagira na određenu dinamiku pjesme. Uspješno je završena skripta za detekciju takta. Međutim, uz dodatno proučavanje moguće ju je proširiti na detekciju određenih frekvencija odnostno spektra frekvencija i time povećati njeno djelovanje i preciznost.

Osnovno znanje za izradu igre usvojeno je na predmetu razvoj računalnih igara. Uz pomoć literature koja se nalazi na službenim stranicama Unity-a, riješeni su problemi koji su se pojavili. Izrada prepoznavanja takta je bio prvi korak u pravljenju igre. Skripta je testirana u procesu pravljenja i nakon izrade gotove i dobili su se ispravni rezultati. Nakon uspješnog prvog dijela, slijedio je dio pravljenja svih glazbenih resursa i kontroliranja avatara.

Glazbeni resursi su napravljeni uz pomoć višegodišnjeg iskustva rada u studiju. Za njegovu izradu potrebno je predznanje na području ekvalizatora, kompresora i sve pripadajuće opreme za njeno nastajanje. Kontroliranje avatara, zajedno sa sinkronizacijom ritmom pjesme, je bio zadnji korak u izradi.

Skriptu *BeatDetectorFinal* koja se nalazi u prilogu je moguće koristiti u drugim aplikacijama u kojima je poželjno detektirati ritam pjesme.

LITERATURA

[1] Wikipedija, Akcijske igre (Action games) (Stranica posjećena 20.6.2018.) https://en.wikipedia.org/wiki/Action_game

[2] Wikipedija, Unity (Unity Game Engine) (Stranica posjećena 21.6.2018.) https://en.wikipedia.org/wiki/Unity_(game_engine)

[3] Beat Detection Algorithm (Stranica posjećena 28.6.2018.) http://archive.gamedev.net/archive/reference/programming/features/beatdetection/

[4] Beat Detection Algorithm (Unity Questions) (Stranica posjećena 28.6.2018) https://answers.unity.com/questions/733587/beat-detection-algorithm.html?childToView=1487400#comment-1487400

[5] Beat Detection Spectrum Algorithm (Github) (Stranica posjećena 3.7.2018.) https://github.com/allanpichardo/Unity-Beat-Detection

[6] Particle System (Unity) (Stranica posjećena 30.7.2018.) https://www.youtube.com/watch?v=agr-QEsYwD0

[7] Rewind Time in Unity (Stranica posjećena 15.7.2018.) https://www.youtube.com/watch?v=eqlHpPzS22U

SAŽETAK

Cilj ovoga rada bio je opisati način sinkroniziranja ulaznog zvučnog zapisa sa kontrolama i pravljenje vlastite glazbe za njeno korištenje. Ritmička akcijska igra napravljena je u razvojnom okruženju Unity. Igra nije zahtjevna i zbog toga ju je moguće igrati na mnogim računalima slabijih performansi. Iz razloga što Unity ima mogućnost izbacivanja napravljenih aplikacija na različite platforme, ovu je igru moguće urediti za Android i iOS uređaje. Igru je moguće proširiti dodavanjem još jednog zaslona odabira pjesama. Teorijski dio rada sastoji se od detaljnog opisa nastajanja skripte za detekciju takta, postupak nastajanja avatara, kontroliranja avatara, sinkronizacija kontrola sa ulaznim zvučnim zapisom i pravljenje objekata potrebnih za kreiranje nivoa.

Ključne riječi: Ritmička akcijska igra, Detekcija takta, ritam, Unity, C#

ABSTRACT

The purpose of this practical work was to closely describe synchronizing an inbound audio recording with game controls and designing adequate proprietary music for use with the mentioned synchronization functionality. The rhythmic action game was developed using the Unity development environment. The game is not overly demanding in terms of computer hardware and can be played on many kinds of lower performance computers. Due to Unity boasting the capability to export applications to multiple platforms, this game could easily be run on Android, iOS and other mobile operating systems alike. The game could be further enriched by adding an another screen, possibly equipped with an interactive music player functionality. A more theoretical aspect of this work included a detailed description of how the beat detection script was developed, as well as the designing of avatar, his controls, the synchronization of controls with the inbound audio data and the development of various objects necessary for level design.

Keywords: Rithm action game, Beat detection, Rythm, Unity, C#

ŽIVOTOPIS

Darko Čalušić rođen je 25.03.1995 u Ozimici, Žepče, Bosna i Hercegovina. Stanuje u Osijeku na adresi Frankopanska 5. Trenutno pohađa 2. godinu diplomskoga studija, smijer, informacijske i podatkovne znanosti. Preddiplomski smijer završava 2016. godine u Osijeku na Fakultetu eletrotehnike, računarstva i informacijskih tehnologija. Srednju školu je pohađao u Katoličkom školskom centru Don Bosco, Žepče, kao tehničar za mehatroniku. Tijekom školovanja istaknuo se na brojnim natjecanjima iz matematike, informatike i AutoCad-a. Osim natjecanja iz znanja, istaknuo se na glazbenim natjecanjima i natjecanjima u karateu. Posjeduje glazbeni studio 5. godina. Od tehničkih znanja koje posjeduje, ističe znanja u: Pascalu, C, C++, C#, Javi, HTML-u, Php-u, Laravel-u, MySQL-u i pravljenju Wordpress stranica. Od ostalih znanja istaknuo bi poznavanje glazbenog programa Cubase.

PRILOZI

DVD

-Unity projekt

-Rad u .docx i .pdf formatu