Izrada 2D platformera u programskom alatu Unity

Bednarek, Adrian

Undergraduate thesis / Završni rad

2022

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:897390

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Adrian Bednarek

IZRADA 2D PLATFORMERA U PROGRAMSKOM ALATU UNITY

ZAVRŠNI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Adrian Bednarek

JMBAG: 0016139222 6

Studij: Poslovni Sustavi

IZRADA 2D PLATFORMERA U PROGRAMSKOM ALATU UNITY

ZAVRŠNI RAD

Mentor:

Doc. dr. sc. Mladen Konecki

Varaždin, 2022

.

Adrian Bednarek

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Tema ovog rada je izrada vlastite računalne igre koja spada u žanr 2D platformera u programskom alatu Unity. U uvodu ću opisati vlastitu motivaciju, mišljenje i iskustvo prema računalnim igricama. Ukratko ću opisati alat Unity, njegovu povijest i njegove mogućnosti. Nakon toga slijedi opis igre. U opisu igre ću prvo objasniti žanr 2D platformer, njegovu povijest i trenutnu popularnost na primjeru. Nakon opisa žanra slijedi opis žanra koji ja namjeravam izraditi te detaljan opis igre njena pravila i mogućnosti. U sljedećem poglavlju ću se isključivo usredotočiti na izradu same igre. Prikazat ću detaljne postupke izrade same igre zajedno sa slikama i linijama koda uz dodatna pojašnjenja. Bit će objašnjeni postupci koji su me doveli do određenog rješenja kao i razlozi, dodatni prijedlozi poboljšanja. U zaključku ću opisati vlastito iskustvo i mišljenje pri izradi igre kao i prema alatu Unity.

Budući da se rad dodatno sastoji praktičnog dijela, u početku sam se upoznavao sa alatom Unity i alatom Visual Studio kako bih naučio osnovne koncepte za izradu igre. To sam postigao istraživanjem službene stranice Unity, razne forum stranice koje su sadržavale diskusije i mišljenja na moju temu i dodatnim proučavanje edukacijskih video uradaka povezanih za moj rad. Osim sadržaja sa interneta, također sam primijenio vlastito iskustvo i ideju pri izradi igre koje sam stekao tijekom akademskog života.

Ključne riječi: Unity, računalne igre, 2D platformer, skripte, animacija, metode, varijable, Spark.

Sadržaj

1. Uvod

Tijekom djetinjstva, igranje računalnih igra iz 90-ih i 2000-ih godina je bila moja omiljena aktivnost koju sam dijelio sa većinom vršnjaka moje dobi u vrtiću i školi. To su bile igre raznih žanrova, ali ponajviše 2D platformeri. Jednostavno upravljanje kontrolama, kreativni dizajn likova i pikselna umjetnost pozadine su elementi koji su me privlačili kod takvih računalnih igri. Igre poput Jazz Jackrabbit, Captain Claw, Mario Forever su samo neke od igrica koje su vršnjaci moje dobi uključujući i mene uživali igrajući te su ostavili pozitivan dojam na meni prema game developerima.

Ovu temu rada sam odabrao iz više razloga koje ću u nastavku navesti. Kako sam u djetinjstvu uživao u računalnim igrama, igrajući zajedno sa prijateljima na istoj tipkovnici, htio sam taj isti užitak podijeliti kroz igru koju bih izradio. Odlično iskustvo tijekom igranja igri iz djetinjstva me je potaknulo da iskažem zahvalnost autorima igre na način da izradim vlastitu verziju igre za koju se nadam da može biti zabavna za buduće igrače. Budući da nisam nikada u životu izradio neku igricu, uvidio sam priliku da to napravim u ovom radu. Htio sam si postaviti nekakav izazov da izradim funkcionalnu igricu počevši od potpune nule. Na taj način mogu poticati druge koji nisu nikada izradili igricu, a žele da također vide da je to izvedivo unatoč drugim obavezama. Osim navedenih razloga, također želim napraviti ovaj rad da mi služi kao dokaz stečenog znanja iz akademskog života. Izradom ovog rada imam priliku saznati tuđa mišljenja, što nije u redu, što mogu napraviti bolje i sl. Stoga bih htio izraditi igricu koja bi osobno bila zabavna za mene i naučiti iz svega što sam izradio kako bih primjenjivao naučeno i u ostalim aktivnostima tijekom života.

U nastavku ću ukratko opisati alat Unity, vidjeti njene mogućnosti i kratku usporedbu, a nakon toga slijedi poglavlje gdje ću opisati 2D platformer. Uz opis žanra, opisat ću igru koju planiram izraditi, sva pravila i mogućnosti koja će biti izvediva u igri. Nakon opisa igre slijedi praktična izrada igre, gdje ću prikazati vlastiti proces izrade igre u alatu Unity te uz alat Visual Studio. Nakon izrade igre slijedi zaključak gdje ću opisati svoja iskustva pri izradi igre, što bi moglo biti bolje i osobni stav prema alatu.

2. Unity

Unity je alat za izradu 2D/3D računalnih igri koji u sebi ima ugrađen IDE(Integrated development environment – integrirano okruženje za razvoj). Kada je 2005. godine predstavljen engine za igre Unity, jedan od njegovih glavnih ciljeva bio je započeti s programiranjem igara pod vodstvom kreatora, a istovremeno demokratski upravljati razvojem igara od strane ostalih developera. Unity je kompatibilan s operativnim sustavom Microsoft Windows unatoč tome što je prvobitno razvijen za Mac OS X [1].

Unity je besplatan alat za sve developere osim za one gaming kompanije čiji prihod prelazi 100 000 dolara godišnje. Engine predstavlja sve značajke i komponente koje Unity sadrži u obliku softvera, a služi kao pomoć pri izradi igre [2]. Pod značajkama se misli definirana fizika unutar igre, 3D/2D objekti, definiranje sudara, manipulacija objekata i sl. IDE predstavlja sučelje koje spaja alate u jedan editor kojim možemo manipulirati prozore i resursima pomoću drag and drop sustava [2]. Unity pruža navigaciju resursa po direktorijima i mapama, to znači da bilo što postavimo u direktoriju preko editora, Unity će kreirati kopiju instance u vlastitom direktoriju. Vrlo korisnica značajka je kreiranje animacija i tranzicija gdje jednostavnim drag and drop nekog sprite-a možemo u sekundi vremena kreirati animaciju. Iste animacije se za navedeni objekt može uređivati u prozoru animator. Uz Unity možemo koristiti programski jezik po izboru, ali u većini slučajeva će se koristiti Visual Studio sa C# u kojem su već definirani paketi za Unity. Glavni razlog zašto je dobro izrađivati igre u Unity je prvenstveno zbog minimalnog kodiranja. U većini slučajeva će sam editor pružati svoje značajke i komponente za koje bi se inače tražilo puno vremena da se kodira. Instalacija je poprilično jednostavna. Na glavnoj stranici postoji link za skidanje u kojem biramo verziju te će se ista verzija instalirati na našem računalu. Tijekom instalacije je potrebno odabrati koje značajke želimo imati te nakon instalacije se logiramo ili registriramo pri čemu je Unity spreman za korištenje.

Unity je u odnosu na ostale alate poput Unreal Engine i Cryengine vrlo otvoren alat. Pod time se misli da nudi izradu igre na različitim platformama poput Andorid, Windows, Mac i igrače konzole [2]. Unity je vrlo jednostavan za korištenje te je zbog toga vrlo popularan za developere početnike. Glavna mana Unity alata u odnosu na druge, jest u tome što zaostaje pri izradi visokokvalitetne grafike [2]. Također, kako imamo brojne značajke i usluge koje nam alat pruža, ipak neki korisnici se žale na postojeće bugove koji se mogu pojaviti tijekom testiranja igre.

3. Opis igre

U nastavku ću opisati 2D platformer igricu. Objasnit ću kako se kretala kroz povijest te kako stoji danas. Nakon toga ću opisati igru koju ja planiram izraditi.

3.1. 2D Platformer opis

3.1.1. Definicija

2D platformer je pod-žanr akcijskih igra. Tipično je za takve igre da koriste vrlo jednostavne kontrole poput trčanja i skakanja za likove kako bi došli do kraja mape. Takve igre su vrlo lagane za shvatiti, ako postoji također mogu imati minimalno razvijenu vlastitu priču i animacije napada (ako se radi o čistom platformeru) [10]. Igra se svodi na tome da će igrač trčati/hodati iz jednog terena na drugi vrlo često izbjegavajući neprijatelje i skupljati bodove. Na kraju razine, igrača može dočekati posljednji protivnik koji može biti jači u odnosu na druge ili će se prikazati rezultati uspjeha pri čemu igrač prelazi na sljedeću razinu. Kod takvih igara, izgled je vrlo jednostavan, ali i dalje primamljiv, sav fokus je bio na izradi izazovnih mapa. Od igrača bi se većinom očekivala preciznost i snalažljivost u 2D prostoru, te čitanje pokreta od protivnika bi bio ključ uspjeha. 2D platformeri se vrlo lako mogu miješati sa preostalim žanrovima pri čemu mogu otvarati vrata za nove kreativne ideje. Jednostavnost i zabava je ono što čini taj žanr primamljivim za sve uzraste i dan danas.

3.1.2. Povijest

2D platformeri se prvi puta pojavljuju u 80-ima kada su lansirane igre poput Space Panic i Donkey King. Neki ne prihvaćaju Space Panic kao prvi platformer jer ne sadrži sve elemente platformera poput skakanja [9]. Unatoč tome, svaka igra u 80-ima je imala svoju bitnu ulogu u razvijanju žanrova igre koje poznajemo dan danas pa tako i za 2D platformere. U 90-im godinama su 2D platformeri imali svoj vrhunac pažnje kada su se velikom brzinom počele lansirati raznolike igre tog žanra. Postoji vrlo mnogo dobrih igra tog doba, neke vrijedne spominjanja su Sonic the Hedgehod, Mega Man 3, Captain Claw, Jazz Jackrabbit i još mnoge druge. Igre tog doba je krasila njihova jednostavnost i kreativnost pri izradi likova. Isprobavajući dizajnirati likove nalik životinjama se pokazalo vrlo uspješnim i vrlo primamljivim pogotovo za mlađu publiku. Na slici možemo vidjeti primjer Sonic the Hedgehod igre. Kako se igrači vole postaviti u poziciji glavnog lika, vrlo bitnu ulogu ima dizajn glavnog lika.

Slika 1: Sonic the Hedgehod

Dodavanje nekih posebnih vještina i mogućnosti poput brzog trčanja koje životinja ne bi imala u realnom svijetu i kreativnost u dizajniranju je ono što tu igru čini primamljivom. Igrajući tu igru kao dijete, sjećam se da sam se vrlo lako mogao zamisliti kao da sam ja dio igre u ulozi glavnog lika Sonica. Htio sam čim brže trčati uz precizno i efikasno izbjegavanje robotiziranih životinja. Kako se djeca vole igrati razne igre koje uključuju trčanje i skakanje, primjenjujući te iste atribute na igru koju zovemo 2D platformer, dajemo im istu tu mogućnost u virtualnom okruženju. Kako Sonic u igri želi biti čim brži u odnosu na ostale ne bi li došao do pobjede, igra je nastojala imitirati slične osjećaje i želje koje imaju i djeca. Stoga možemo zaključiti da igre koje su pobudile radoznalost i želje kod djece, otvarajući im vrata za nove ideje, pokazale su se vrlo uspješnim.

3.1.3. Kombinacija žanrova

Danas su 2D platformere dijelom zamijenili 3D igre nudeći vrlo preciznu grafiku sa vrlo interaktivnim likovima i pričama. U odnosu na 2D platformere, takve igre su obično vrlo skupe i zahtjevne za igrati i vrlo često će postojati minimalni zahtjevi kako bi se iste igre mogle pokrenuti. Jedan dobar načina oživljavanja 2D platforera jest kroz kombinaciju drugih žanrova. Kako je ovaj svijet poprilično dobro upoznat sa igricama, kombinirajući 2D platformere sa drugim žanrovima, nudimo veću kompleksnost igre, a igračima nov doživljaj i još veći zahtjev pri razumijevanju igre. Kao i na slici za primjer sam uzeo Super Smash Bros.

Slika 2: Super Smash Bros

Smash je igra sa kombiniranim žanrom 2D platformera i borilačkih vještina koja se pokazala jako uspješnom [21]. Oživljavanje raznolikih legendarnih likova iz 2D platformera poput Sonic, Pacman, Mega man, Mario u novoj igri, raznolike kombinacije napada i prelazak mape čine ovu igru vrlo jedinstvenom. U odnosu na klasični 2D platformer, Smash nudi razne načine igranja u borilačkim vještinama sa detaljnim kombinacijama napada protiv drugih igrača paralelno kao i kod klasičnih 2D platformera, prelazak mape zajedno sa drugim igračima. Danas se sve više pažnje posvećuje e-sportu gdje se igrači diljem svijeta međusobno natječu u igrama radi osvajanja visokih nagrada i slave. Usvajanje borilačkih vještina s naglaskom na 2D platformera, ova igra čini dobru kompetenciju suvremenim igrama u e-sportovima.

3.2. Opis izrade žanra

2D platformer koji ja namjeravam izraditi, bit će u kombinaciji avanture gdje će igrač osim prelaska razine, pokušavati pronaći skrivena mjesta sa dodatnim bodovima. Kao i na primjeru Captain Claw, igrač će imati mogućnost koristiti više načina napadanja.

Slika 3: Captain Claw

Igraču će biti omogućeno pucanje, ali i borba šakama. Isto tako primjenjivat ću sličan način poraza igrača kao i u Captain Claw igrici, gdje glavni lik padne sa cijele mape. Svi ostali elementi će biti slični poput Jazz Jackrabbit igri sa vlastitim dodatnim idejama.

3.3. Spark Rabbit

Igra koju izrađujem zvat će se Spark Rabbit. Radi se o narančastom zecu kao glavnoj ulozi koji skuplja bodove u obliku povrća i voća, a za svakih 10000 bodova, Spark dobiva ekstra život. Prilikom sakupljanja bodova Spark se susreće sa neprijateljima koji mu žele naškoditi. Razlikujemo 4 različita neprijatelja: zmija, minotaur, pčela i pauk. Za svakog poraženog protivnika Spark će dobiti određeni broj bodova. Uz sebe će imati svoj pištolj sa kojim će pucati zelene metke. Osim pištolja Spark će se moći boriti šakama i nogom. Bitna razlika je u tome što Spark nanosi više štete svojim protivnicima ako koristi šake dok pištoljem može pucati svoje neprijatelje iz daljine. Dodatna prednost Sparka je što unatoč tome što je brz, on može trčati još brže držanjem tipke „M“. Sparku će biti omogućen skok zajedno sa duplim skokom kako bi lakše izbjegavao nevolje. Osim brzog trčanja, Spark ima još jednu skrivenu sposobnost. Dvoklikom na desnu/lijevu strelicu, Spark aktivira ultra brzinu pri čemu u djeliću sekunde postaje otporan na štetu i prođe određenu udaljenost u velikoj brzini, poprimajući blještavu animaciju otkud i dolazi njegovo ime Spark.

Igra će imati 2 razine kako bismo demonstrirali dolazak na cilj i prijelaz na sljedeću razinu kao i nastavljanje razine nakon gubitka života. U 1. razini, Spark će se nalaziti u šumovitom kraju gdje u početku isprobava svoje mogućnosti, prolazeći kroz kratak vodič. Nakon toga Spark će imati svoje prve prepreke koje će morati sam izbjeći ili se suočiti sa njima. Osim voća, Sparku će biti omogućena dodatna pomoć. Kako bi olakšali Sparku pustolovine, imat će šansu pokupiti svoje omiljeno povrće: „Mrkva“, koja će mu resetirati HP i odmah povratiti snagu za nastavak dalje. Naravno, tu je i još veća nagrada, istražujući mapu igrač može naići na ekstra živote koji će mu povećati šansu da završi razinu. Kako bismo očarali pustolovine Sparka, u pozadini će svirati muzika koja će se mijenjati prema njegovim HP-ima. Ako Spark izgubi 2 srca koja predstavljaju njegov HP, promijenit će se glazba kako bismo ga obavijestili da pripazi malo. Ako Spark ima manje od 3 HP-a, ponovo će se promijeniti glazba, kako bi upozorila igrača da bude još oprezniiji. Sa 0 HP-a cijeli teren uključujući i nebo će mijenjati boje velikom brzinom koja će tjerati Sparka da brzo pronađe mrkvu kako bi obnovio snagu, u suprotnom ako Spark izgubi posljednji HP, vraća se na početak. Kroz pustolovine, Spark će naići na lisice koje predstavljaju checkpoint. U najgorem slučaju, ako Spark izgubi život, moći će nastaviti pustolovine od zadnjeg checkpointa. Kako smo prije naveli da će Spark imati priliku naići na nagrade, kao malu pomoć, moći će se

poslužiti trampolinima i teleportom koji će mu pomoći da dođe do skrivenih mjesta. Igrač će završiti sa razinom kada prođe kroz zastavu sa crno bijelim kvadratićima.

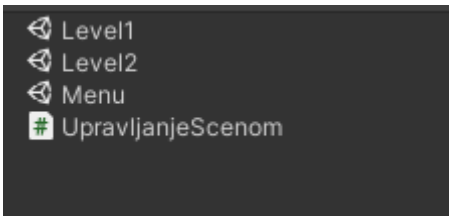
Druga razina predstavlja malo veći izazov. Spark će se nalaziti u napuštenom gradu gdje će se sresti sa novim neprijateljima. Dok u 1. razini su većinom prevladavale pčele, na 2. razini će se pojaviti crni paukovi. Kako bi postavili veći izazov, borbe šakom/nogom su neučinkovite protiv paukova. Jedini način da ih se riješi jest da ih se izbjegne pod svaku cijenu ili se može koristiti pištolj. Paukovi mogu napadati sa stropa, stoga gravitacija na njih nema utjecaja. Ako Spark uspije doći na kraj druge razine, promijenit će se nova glazba u pozadini, a cijeli teren će ponovno mijenjati boje. To će biti znak da se približava glavnom neprijatelju skeletonu. U odnosu na druge, skeleton će imati mnogo više HP-a i snage u odnosu na druge neprijatelje koje je igrač susretao te će predstavljati mnogo veći izazov. Igrač će imati priliku primijeniti naučeno kroz igru u borbi protiv svog glavnog neprijatelja ne bili završio 2. razinu. Spark će na raspolaganju imati mrkve za resetiranje HP tijekom borbe skeletona. Ako igrač uspije poraziti skeletona, dočekat će ga pobjedničke trube, a 2. razina će biti završena, a s time i cijela igra. Na kraju će se prikazati ukupni bodovi koje je igrač uspio sakupiti tijekom igre. Na početku i tijekom igre će biti omogućen izbornik i postavke kako bi igrač mogao izaći iz igre ili postaviti glasnoću glazbe.

4. Izrada igre

U ovoj cjelini slijedi praktični postupak izrade igre Spark Rabbita, 2d platformera. Kao što je već i prije najavljeno, igra će biti izrađena u alatu Unity, sa postojećim open-source sprite-ovima, glazbom i sličnim detaljima. Mala napomena, budući da se izradom Unity igre susrećem prvi put, rješenja koja ću navesti neće biti idealna, uvijek će postojati bolja rješenja koja vode boljoj optimalizaciji i kontroli igre.

4.1. Scene

Ova igra će se sastojati od 3 scena. Prve dvije scene će predstavljati razine kroz koju igrač prolazi dok posljednja scena predstavlja glavni izbornik koji se pojavljuje na početku pri otvaranju igre.



Slika 4: Direktorij scena

4.1.1. Skripta scene

U skripti scena se nalaze najosnovnije naredbe za kretanje po scenama. Dodao sam sljedeće metode: povratak na izbornik, na početnu razinu, ponavljanje trenutne razine, prijelaz na sljedeću razinu. To su otprilike sve potrebne kombinacije koje ću koristiti tijekom igre. Budući da se bavimo scenama potrebno je dodatno koristiti biblioteku pod nazivom **UnityEngine.SceneManagement**, u suprotnom nećemo moći koristiti naredbe za manipulaciju scena. U nastavu prilažem primjer koda za upravljanje scenama.

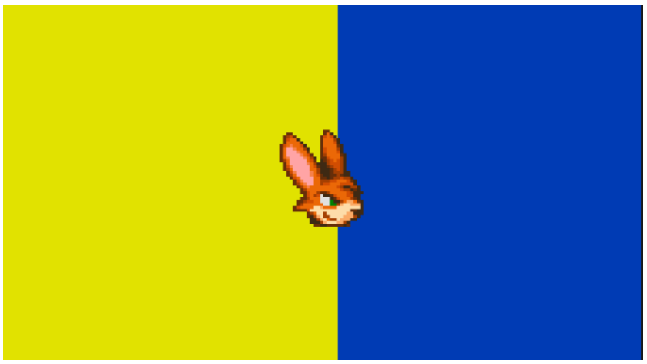
```
1.  public class UpravljanjeScenom : MonoBehaviour
2.  {
3.      [SerializeField]
4.      Tranzicija tranzicijaScene;
5.
6.      private void Start()
7.      {
8.         tranzicijaScene.PrikaziTranziciju();
9.      }
10.     IEnumerator UcitavanjeLevela(int indeks)
11.     {
12.         tranzicijaScene.ZapocniDruguTranziciju();
```

```
13.           yield return new WaitForSeconds(1.2f);
14.           SceneManager.LoadScene(indeks);
15.       }
16.       public void SljedeciLevel()
17.       {
18.
      StartCoroutine(UcitavanjeLevela(SceneManager.GetActiveScene().buildIndex +
      1));
19.       }
20.       public void NaMenu()
21.       {
22.           StartCoroutine(UcitavanjeLevela(0));
23.       }
24.       public void NaPocetniLevel()
25.       {
26.           StartCoroutine(UcitavanjeLevela(1));
27.       }
28.       public void RestartLevel()
29.       {
30.
      StartCoroutine(UcitavanjeLevela(SceneManager.GetActiveScene().buildIndex));
31.       }
32.   }
```

Možemo vidjeti da sam osim scena koristio i Tranzicija kao dodatnu klasu. Detaljnije o tome ću pričati u sljedećem poglavlju. Bitno je za naglasiti da sam koristio dodatnu metodu tipa IEnumerator. Ukratko, unutar te metode možemo izvršiti određenu liniju koda te pričekati nekoliko framova ili sekunda nakon čega će se izvršiti sljedeća linija koda, uz paralelno funkcioniranje ostalih dijelova koda. To nam je vrlo korisno jer pri prijelazu u sljedeću scenu, mi želimo odvojiti sekundu vremena da uredno učita tranziciju o kojoj ću spomenuti nešto više u nastavku. Još dodatno za naglasiti, da bi se pokrenule metode tipa IEnumerator, potrebno je izvršiti metodu *StartCoroutine()*.

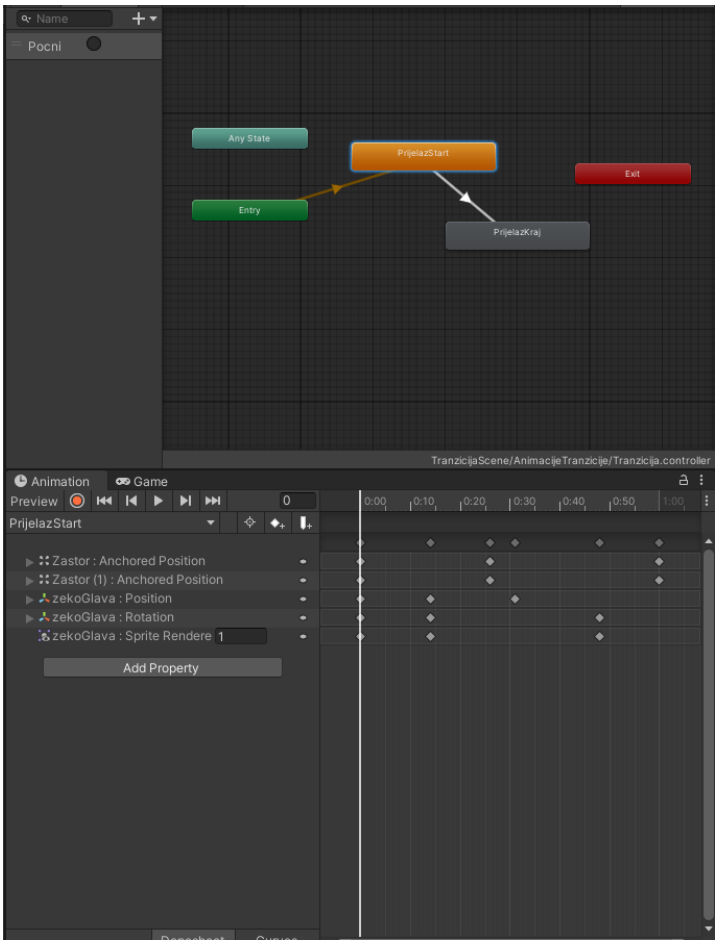
4.1.2. Tranzicija scene

Kod tranzicije scene sam koristio tri slike koje sam svrstao u objekt naziva tranzicija. Slike se sastoje od dva zastora različite boje i ikone Sparka. U primjeru možemo vidjeti sliku.



Slika 5: Tranzicija

Razlog zašto želimo koristiti tranzicije scena jest, prilikom učitavanja na sljedeću scenu, trenutna scena se obično zamrzne te igrač ima dojam kao da je igra zamrznuta. Tranzicijom dajemo nešto ljepši dojam pri učitavanju scene. Pri kreiranju slike koju želimo vidjeti kao tranziciju, na objektu roditelja je potrebno kreirati novu animaciju koju imenujemo po volji. Kada to učinimo, u „Animation“ prozoru kliknemo na gumb „Recording“, kako bi snimili događaj pri promjeni i jednostavno kreiramo animaciju po volji. S druge strane potrebno je kreirati još jednu animaciju koja je suprotna od prve kako bi se poklapale, te će izgledati kao da se tranzicija zatvara - otvara. Ovako izgleda u mom primjeru.



Slika 6: Animacija tranzicije i prozor „Animator“

Prva animaciju koju smo napravili za tranziciju će se događati svaki puta kada se učita nova scena, dok druga će biti kada scena završava. U animatoru sam postavio parametar tipa „trigger“ pomoću kojeg započinjemo završnu tranziciju scene. Za prvu scenu nisu potrebni nikakvi uvjeti jer ona će automatski sama započeti budući da je povezana sa „Entry“. U nastavku ću prikazati primjer skripte za tranziciju scene.

```
1.  public class Tranzicija : MonoBehaviour
```

```
2.  {
3.      Animator animator;
4.      private bool zapocelaDrugaAnimacija = false;
5.      private void Start()
6.      {
7.          animator = GetComponent<Animator>();
8.          Invoke("MakniTranziciju",1.5f);
9.      }
10.     private void MakniTranziciju()
11.     {
12.         if (!zapocelaDrugaAnimacija)
13.         {
14.             gameObject.SetActive(false);
15.         }
16.         zapocelaDrugaAnimacija=false;
17.     }
18.     public void PrikaziTranziciju()
19.     {
20.         gameObject.SetActive(true);
21.     }
22.     public void ZapocniDruguTranziciju()
23.     {
24.         PrikaziTranziciju();
25.         zapocelaDrugaAnimacija = true;
26.         animator.SetTrigger("Pocni");
27.     }
28. }
```

Budući da će se prva animacija tranzicije automatski pokrenuti, u *Start()* metodi sam osim referenciranja animatora postavio metodu *Invoke().* Metoda *Invoke()* će pokrenuti metodu koju smo odredili unutar nje nakon određenog vremena koji smo postavili kao drugi parametar. Na taj način postavljamo objekt nevidljivim pri čemu se animacija neće vidjeti nakon što završi. Dalje imamo metodu *ZapocniDruguTranziciju()* koja kao što joj i ime kaže, pokreće drugu animaciju tranzicije. Tu metodu koristimo u klasi UpravljanjeScenom pri prijelazu na drugu scenu. Sve ostalo su samo pomoćne metode/varijable kako bi ostvarili zadan rezultat i s time završavamo sa ovim poglavljem.

4.2. Glavni igrač Spark

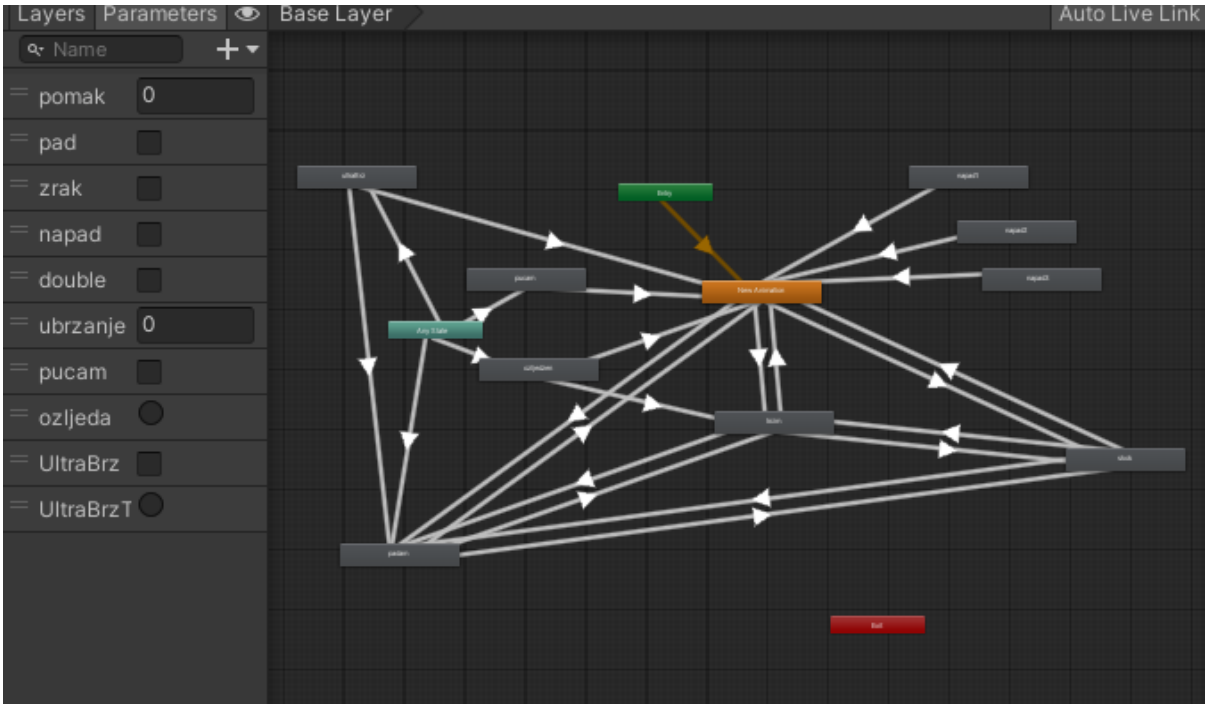
U mojoj igri glavnu ulogu igrača ima narančasti zec zvan Spark. Osnovne animacije Sparka su trčanje, brzo trčanje, ultra brzina, pucanje, borba šakama i nogom, animacija boli i skokovi, padovi. Koristio sam open-source sprite-ove za njega dostupne na stranici: https://opengameart.org/ . Na slici možemo vidjeti primjer sprite-a.

Slika 7: Izgled Sparka

U početku nakon što sam izradio animacije za Sparka, prvo koristeći drag and drop metodu, nakon toga u „Animation“ prozoru pod „Create new Clip“ kreirao ostale animacij. Također je trebalo dodati preostale komponente. Budući da su nam automatski dodane komponente „SpriteRenderer“ i „Animator“, bilo je potrebno ručno dodati „RigidBody2D“ i po želji od „Collider2D“ sam odabrao „CapsuleCollider2D“. Na slici možemo vidjeti zelenu liniju koja predstavlja „CapsuleCollider2D“ dok dijamantni oblici predstavljaju djecu objekte za našeg glavnog igrača. Dva gornja objekta predstavljaju područje napada dok 3 donja utvrđuju tlo te će mi pomoći pri postavljanju zadane animacije kada je na podu.

Slika 8: Spark

Ukratko „RigidBody2D“ će nam omogućiti manipulaciju kretanja Sparka dok nam „Collider2D“ omogućuje kolizije sa drugim objektima koji imaju collidere. Spark ima 4 različite skripte na sebi, a to su : „PlayerHP“, „PlayerKontroler“, „PlayerRespawn“ i „PlayerSound“. U nastavku ću prikazati hijerarhiju animacija.



Slika 9: Prikaz prozora „Animator“ za Sparka

Spark ima najviše animacije od ostalih likova u igri stoga je i prozor „Animator“ vrlo neuredan. Negdje sam koristio **trigger** i **bool** parametar za istu animaciju, ali to je bilo sve u svrhu da se ograniči ostalim animacijama da se izvode. Jedino animacije napada nemaju ulaznu vezu. Budući da sam ih htio nasumično pokretati, to sam sve izveo iz skripte stoga nisam imao potrebu koristiti „Animator“ za njih. Sve skripte ću detaljnije objasniti u sljedećim poglavljima.

4.2.1. „PlayerHP“ skripta

U „PlayerHP“ skripti upravljamo stanjem našeg igrača, njegovim HP-om i životima. U sljedećem primjeru možemo vidjeti primjer skripte:

```
1.      private int hp = 5;
2.      private int zivoti = 3;
3.
4.      [SerializeField]
5.      GameObject HPsrce;
6.
7.      [SerializeField]
```

```
8.      GameObject Zivot;
9.
10.     private PlayerSound sound;
11.     private void Start()
12.     {
13.         sound = GetComponent<PlayerSound>();
14.         ZivotiOdProslogLevela();
15.         OsvjeziPrikazZivota();
16.         OsvjeziPrikazHPa();
17.     }
```

U ovom primjeru imamo prikazane inicijalizirane početne varijable i metode koje se pokreću na početku. Za HP i živote se već podrazumijeva zašto imamo stoga neću ih objašnjavati. Stavio sam za inicijalizaciju dva objekta koja su također HP i život koji su tipa GameObject. Radi se o slikama života koji su u obliku srca dok život ima broj i ikonu Sparka. Stoga metode: *OsvjeziPrikazZivota()* i *OsvjeziPrikazHPa()* postavljaju prikaz stanja da igrač može vidjeti stanje rezultata i to je u pravilu cijela priča. Kod inicijalizacije **sound**, koristio sam u svrhu da Spark napravi zvuk kada dobi život, bolje bi bilo da sam postavio tu varijablu u drugoj klasi, ali zbog malih komplikacija sam ju ovdje ostavio. Sljedeće ću pokazati neke od metoda, neću sve jer ih većina samo služe u pomaganju drugih metoda i poprilično su jednostavne.

```
18.     public void smanjiHPiliZivot()
19.     {
20.         smanjiHP();
21.
22.         if(!AkoImaHP())
23.         {
24.             smanjiZivote();
25.         }
26.     }
```

Kada se desi da igrač dobiva štetu, ovu metodu ćemo pozvati te ćemo smanjiti živote ili HP. Bitno je za naglasiti da je sva šteta u cijeloj igri ista, te će Spark za svaku štetu gubiti HP manje.

```
27.     public void ResetHP()
28.     {
29.         hp = 5;
30.         OsvjeziPrikazHPa();
31.     }
32.     public void DobivaZivot()
33.     {
34.         zivoti += 1;
35.         sound.PustiZivotZvuk();
36.         OsvjeziPrikazZivota();
37.     }
38.     public int stanjeZivota()
39.     {
40.         return zivoti;
41.     }
42.     private void OsvjeziPrikazHPa()
43.     {
44.         for (int i = 0; i < 5; i++)
```

```
45.           {
46.               if (i < hp)
47.               {
48.                   HPsrce.GetComponentsInChildren<SpriteRenderer>()[i].enabled
    = true;
49.               }
50.               else
51.               {
52.                   HPsrce.GetComponentsInChildren<SpriteRenderer>()[i].enabled
    = false;
53.               }
54.           }
55.       }
```

Ovdje imamo ostale metode, nisam stavio sve jer su većinom samo metode koje provjeravaju vrijednosti, ništa više. Ovdje imamo primjer metode gdje osvježavamo prikaz HP-a za igrača. U pravilu provjeravamo naše trenutno stanje HP-a te prema toj vrijednosti toliko će biti vidljivih sličica srca. Varijablu **sound** koja je tipa PlayerSound proizlazi iz druge skripte koju ću također kasnije objasniti. Trenutno ju ovdje koristim kod metode *DobivaZivot()* radi zvuka. Glavni razlog zašto sam ju baš postavio ovdje, a ne sa ostalim zvukovima je zbog toga što igrač može dobiti na dva načina. Kada sakupi 10 000 bodova ili kada pokupi život u obliku itema. Iz tog razloga, umjesto da zvuk koristim kod skripte itema, koristio sam upravo u toj skripti. Ovdje još ima sljedeću metodu koju bih htio objasniti.

```
56.       private void ZivotiOdProslogLevela()
57.       {
58.           if (Manager.ekstraZivoti != 0)
59.           {
60.               zivoti = Manager.ekstraZivoti;
61.               Manager.ekstraZivoti = 0;
62.           }
63.       }
```

Budući da su životi i HP **private**, njihove vrijednosti će biti jednake kao i na početku pri prijelazu na drugu scenu. Iz tog razloga stavio sam statičku varijablu za živote koja će pohraniti vrijednosti prije prelaska na sljedeću razinu. Ovo je vrlo nezgodni način na koji sam izveo ovaj problem, ali da uštedim na vremenu postavio sam ovu metodu. Ovdje još prilažem slike HP-a koje sam koristio.

Slika 10: HP srce (Vlastita izrada)

4.2.2. „PlayerRespawn“ skripta

U „PlayerRespawn“ skripti određujemo gdje će se Spark pojaviti kada izgubit sav HP ili kada se treba pojaviti na početku razine. U nastavku ću prikazati cijelu skriptu.

```
1.  void Start()
2.      {
3.          coll = GetComponent<CapsuleCollider2D>();
4.          animator = GetComponent<Animator>();
5.          PlayerKontrole = GetComponent<PlayerKontroler>();
6.          hpManager = GetComponent<PlayerHP>();
7.          RespawnPozicija();
8.      }
9.      private void RespawnPozicija()
10.     {
11.         respawn = StartCheckPoint.transform.position;
12.         transform.position = respawn;
13.     }
14.     private void OnTriggerEnter2D(Collider2D collision)
15.     {
16.         if (collision.tag == "Respawn")
17.         {
18.             respawn = collision.transform.position;
19.             collision.GetComponent<BoxCollider2D>().enabled = false;
20.             collision.GetComponent<Animator>().Play("stoji");
21.         }
22.     }
23.     public void VratiSeNaCheckPoint()
24.     {
25.         VratiZadanePostavkePlayera();
26.         transform.position = respawn;
27.     }
28.     private void VratiZadanePostavkePlayera()
29.     {
30.         PlayerKontrole.NeBoli();
31.         animator.Play("New Animation");
32.         hpManager.ResetHP();
33.         coll.enabled = true;
34.         animator.speed = 1;
35.     }
36. }
```

U početku imamo metodu *RespawnPozicija()* koja će pamtiti prvu poziciju gdje se Spark treba pojaviti, a ona je referenca na objekt tipa GameObject „Početak“ koji je uglavnom slika starta. Kasnije će se njegova vrijednost mijenjati prema checkpointovima kroz koje će Spark prolaziti. Imamo jednu *OnTriggerEnter()* metodu koja će aktivirati smo ako će Spark doći u kontakt sa drugim objektima koji imaju na sebi posebnu oznaku „Respawn“. Radi se o tome, da kad god Spark prođe kroz checkpoint, tom objektu će se maknuti collider, promijenit će se animacija te će **respawn** varijabla dobiti vrijednost pozicije checkpointa. Razlog zašto trebamo isključiti collder jest da se ne bi više puta izvršio na isti objekt metoda *OnTriggerEnter2D()*.

Sljedeću metodu vrijednu objašnjenja, imamo *VratiZadanePostavkePlayera().* Ta metoda nam je jako bitna jer kada Spark izgubi život, maknut će mu se collider i smrznuti animacija,

sve u svrhu da javimo igraču da je izgubio život. Nakon toga se te postavke postavljaju na prvobitno stanje. Također je potrebno resetirati HP i postaviti početnu animaciju.

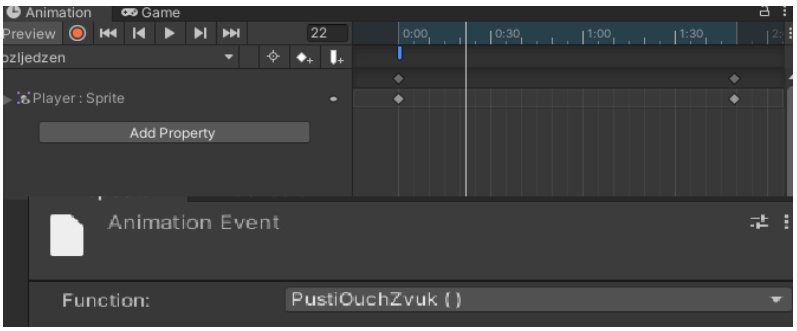
4.2.3. „PlayerSound“ skripta

„PlayerSound“ skripta sadrži sve zvukove Sparka te su umetnute uglavnom kod animacija Sparka, iznimka jedino zvuk za dobivanje života, prije sam napomenuo da sam ju stavio u HP skriptu. U nastavku ću prikazati skriptu.

```
1.      private void Start()
2.      {
3.          hpManager = GetComponent<PlayerHP>();
4.      }
5.      private void Awake()
6.      {
7.          sourceZvuk = gameObject.AddComponent<AudioSource>();
8.          sourceZvuk.volume = .7f;
9.      }
10.     private void PustiultraBrzZvuk()
11.     {
12.         sourceZvuk.PlayOneShot(ultraBrzZvuk);
13.     }
14.     private void PustiOuchZvuk()
15.     {
16.         if (hpManager.AkoImaHP())
17.         {
18.             sourceZvuk.PlayOneShot(OuchZvuk);
19.         }
20.         else
21.         {
22.             sourceZvuk.PlayOneShot(Dead);
23.         }
24.     }
25.     private void PustiMetakZvuk()
26.     {
27.         if (mozePustiti)
28.         {
29.             mozePustiti=false;
30.             sourceZvuk.PlayOneShot(Metak);
31.             Invoke("ZvukCD", .12f);
32.         }
33.     }
```

U početku imamo dvije početne metode*: Awake()* i *Start()*. Bitna razlika između tih metodi je da će *Awake()* započeti odmah pri učitavanju skripte dok će Start() započeti pri korištenju skripte. Stoga, *Awake()* metoda je brža u odnosu na *Start()* metodu. U mom primjeru, nije bilo potrebe postaviti *Awake(),* jer Spark pri učitavanju neće raditi zvukove, ali htio sam naglasiti da želim imati postavljen **sourceZvuk** koji je tipa AudioSource. AudioSource predstavlja izvor iz kojeg potječe zvuk, razlikujemo još AudioClip koji predstavlja sam taj zvuk i mi uglavnom trebamo barem 1 AudioSource za varijable tipa AudioClip kako bi se čuo zvuk.

Dalje sam koristio „PlayerHP“ skriptu čisto radi provjere da li je Spark izgubio sav HP, jer tada može imati zvuk za „Ouch!“ ili zvuk umiranja. Ostali zvukovi se puštaju u samim animacijama koje možemo vidjeti na sljedećoj slici.



Slika 11: Postavljanje „Animation event“ kod animacije „ozljedzen“

Možemo primijetiti jedan maleni pravokutnik. On nam omogućuje da pokrenemo neku funkciju iz skripte. U našem primjeru smo za animaciju „ozlijedzen“, koristili metodu *PustiOuchZvuk()*. Taj isti princip smo primijenili i za zvuk umiranja, pucanja metaka, ultra brz i za tri animacije napada.

4.2.4. „PlayerKontroler“ skripta

U „PlayerKontroler“ skripti se nalaze sve animacije Sparka, sva njegova kretanja te uvjeti za te animacije. Također sadrži i *OnTriggerEnter2D()* metode gdje nanosi štetu protivniku. To je najveća skripta od svih ostalih koje sam kreirao i trebala bi biti podijeljena u manje dijelove, no kako sam to radio u vrijeme kada sam se tek učio jezik C#, nisam baš previše radio objektno orijentirano, do neke razine sam razdvojio u skripte objašnjene prije, no nadam se da će biti razumljivo. Ovdje sad možemo vidjeti početne metode koje se pokreću u metodi *Update()* i *Start().*

```
1.      void Start()
2.      {
3.          animator = GetComponent<Animator>();
4.          rb2 = GetComponent<Rigidbody2D>();
5.          sprite = GetComponent<SpriteRenderer>();
6.          metakref = Resources.Load("metak");
7.          HPmanager = GetComponent<PlayerHP>();
8.
9.          PostaviTimerZaDvoklik();
10.         ZapamptiPozicijuPlayeraI();
11.     }
12.     void Update()
13.     {
14.         trenutna_pozicija = transform.position.y;
15.
```

```
16.             KretanjeAnimacija();
17.             NapadanjeAnimacija();
18.             OzljedzenAnimacija();
19.             UltraBrzaAnimacija();
20.             SkokAnimacija();
21.             PucanjeAnimacija();
22.             OkreniPlayera();
23.             BrzoTrcanjeAnimacija();
24.
25.             zadnja_pozicija = trenutna_pozicija;
26.         }
```

U metodi *Start()* referenciramo komponente koje ćemo koristiti za : „Animator“, „Rigidbody2D“, „SpriteRenderer“, te dohvaćanje skripte „PlayerHP“ te referenciramo objekt zvan **metak**, koji nam služi u animaciji pucanja. U metodi *Update()* će se za svaki frame vrtjeti metode koje kontroliraju sve animacije Sparka koje smo naveli. Za svaku od njih ću izdvojiti posebno poglavlje. Osim metoda također imamo varijable **trenutna** i **zadnjaPozicija** sa kojom pamtimo poziciju Sparka u odnosu na prošli frame. Njih ću u objasniti u navedenim animacijama. Metoda *PostaviTimerZDvoklik()* ukratko rečeno, pamti vrijeme trajanja između dva klika od strane igrača, kako bismo mogli odrediti da li se radi o dvokliku.

4.2.5. Kretanje animacija

U nastavku ću prikazati metodu za animaciju kretanja.

```
27.         private void KretanjeAnimacija()
28.         {
29.             if (MoguLiHodatiNaPodu()||MoguLiSeKretatiPoZraku())
30.             {
31.                 horizontala = Input.GetAxisRaw("Horizontal") * brzinatrcanja;
32.                 rb2.velocity = new Vector2(horizontala, rb2.velocity.y);
33.                 PostavljanjeAnimacijeKretanja();
34.             }
35.         }
```

Kako bi se kretali provjeravamo uvjete za kretanje na terenu i kretanje po zraku. Valja napomenuti da se kretanje ne odnosi na animaciju trčanja, već općenito micanje. Ako su jedan od uvjeta dozvoljeni postavit ćemo neku brzinu pod varijablom **brzinatrcanja** te će se Spark kretati. Dobro je napomenuti da sam metodu *Input.GetAxisRaw()* koristio kako bih spriječio izglađivanje prilikom micanja Sparka. U smislu, da kada se prestane micati, vrijednost će biti 0, a kada se miče, bit će 1 sa umnoškom **brzinatrcanja**. U nastavku ću prikazati metodu *PostaviAnimacijuKretanja().*

```
36.         private void PostavljanjeAnimacijeKretanja()
37.         {
38.             animator.SetFloat("pomak", Mathf.Abs(horizontala));
39.             animator.SetFloat("ubrzanje", 1f);
40.         }
```

Pomak nam u animaciji služi da utvrdimo da li se Spark uopće miče, dok **ubrzanje** nam služi da li Spark samo trči ili brzo trči. Na temelju toga će se i animacija staviti. Razlog zašto sam stavio **ubrzanje** tipa **float** jer i samu animaciju ubrzavamo pa na taj način ne trebam 2 parametra već koristim u obje svrhe vrijednost od **ubrzanje**. Metode sa kojim provjeravam uvjete kretanja za Sparka, neću prikazati, ali čisto da izjasnim, one provjeravaju parametre ostalih animacija te vraća **true** ili **false**.

4.2.6. Napadanje animacija

U nastavku ću prikazati metodu za animaciju napadanja.

```
41.      private void NapadanjeAnimacija()
42.      {
43.          if (MozeNapasti())
44.          {
45.              if (JesamLiNaPodu())
46.              {
47.                  rb2.velocity = Vector3.zero;
48.              }
49.              IzvediAnimacijuIDamage();
50.              Invoke("resetAttack", 0.3f);
51.          }
52.      }
```

U pravilu kako bismo nanijeli štetu prvo provjeravamo uvjete za napadanje. Budući da nemam sprite-ove napadanja prilikom hodanja, postavio sam Sparka da stane kada se želi boriti, ako je na podu. U zraku to ne vrijedi, stoga sam u if uvjetu postavio kretanje Sparka na 0. Metoda *resetAttack()* predstavlja cooldown napada kako igrač ne bi mogao velikom brzinom nanositi štetu protivnicima. U nastavku ćemo pogledati uvjete kako bi Spark mogao napasti.

```
53.      private bool MozeNapasti()
54.      {
55.          if(Input.GetKeyDown("n") && !animator.GetBool("napad") &&
     !ONcooldown && !boli && HPmanager.AkoImaHP())
56.          {
57.              return true;
58.          }
59.          return false;
60.      }
```

Kao što i u **if** uvjetu piše, Spark može napasti samo ako pritisne tipku „N“ uz uvjet da mu animacija napadanja neaktivna, ako nema nikakvu animaciju u timeru ili cooldown-u, ako nije primio štetu te ako ima HP. Alternativno sam mogao provjeru HP-a postaviti odmah na početku za sve animacije, ali ovako je isto uredu iako nije efikasno. U nastavku ću prikazati metodu gdje se postavlja animacija.

```
61.      private void IzvediAnimacijuIDamage()
62.      {
63.          int BrojAnimacije = NasumicniBrojNapada();
```

```
64.           animator.SetBool("napad",true);
65.           animator.Play("napad" + BrojAnimacije);
66.
67.           float damage = OdrediDamage(BrojAnimacije);
68.           NanesiDamage(damage);
69.       }
```

Pri izvađanju animacije nam u početku treba nasumični broj koji nam generira metoda *NasumicniBrojNapada().* Razlog tome je što želim da pri pritisku tipke, Spark nasumično odabere hoće li udarati nogom, lijevom rukom ili desnom rukom, nakon čega postavljamo animaciju. Također ako Spark napada nogom nanosit će više štete iz kojeg sam razloga i postavio varijablu da zapamti o kojoj se animaciji radi. U nastavku ću prikazati metodu *NanesiDamage().*

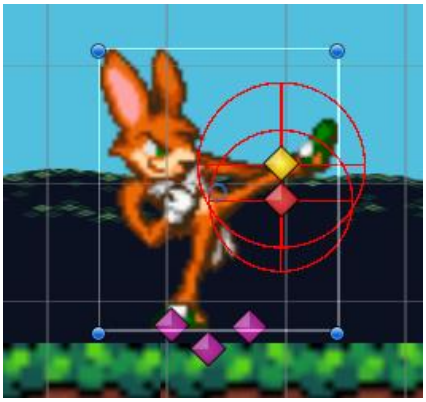
```
70.       private void NanesiDamage(float damage)
71.       {
72.           Collider2D[] udariSakom =
      Physics2D.OverlapCircleAll(udaracSakom.transform.position,
      radiusUdarcaSake, EnemyLayer);
73.           foreach (Collider2D protivnik in udariSakom)
74.           {
75.               if (protivnik.name.Contains("Boss"))
76.               {
77.                   protivnik.GetComponent<BossHP>().SmanjiHP(damage);
78.               }
79.               else
80.               {
81.                   protivnik.GetComponent<EnemyHP>().SmanjiHP(damage);
82.               }
83.           }
84.       }
```

Prije nego što sam radio ovu metodu bilo je potrebno napraviti prazan objekt i postaviti ga kao dijete naspram objekta „player“. Taj objekt predstavlja udarac rukom ili udarac šakom. Postoji jedna izvrsna metoda koja pamti dodire svih objekta u nekom području koja se zove *Physics2D.OverlapCircleAll().* U njoj sam pohranio poziciju objekta udarca šakom i svojevoljno odredio radius kako bih imao krug koji nanosi štete neprijateljima, te je još potrebno nadodati layer koji želimo pamtiti. U **foreach** petlji provjeravamo koga smo udarili ako smo ikoga udarili, te ćemo koristiti njihovu skriptu kako bismo smanjili hp. U nastavku ću prikazati Gizmos.

```
85.       private void OnDrawGizmos()
86.       {
87.           //Udarac nogom
88.           Gizmos.color = new Color(1, 0, 0);
89.           Gizmos.DrawWireCube(Vector3.zero, Vector3.one);
90.           Gizmos.DrawWireSphere(udaracNOgom.transform.position,
      radiusUdarcaNoge);
91.
92.           //Udarac sakom
93.           Gizmos.color = new Color(1, 0, 0);
94.           Gizmos.DrawWireCube(Vector3.zero, Vector3.one);
95.           Gizmos.DrawWireSphere(udaracSakom.transform.position,
      radiusUdarcaSake);
```

```
96.     }
```

Budući da na ekranu ne možemo vidjeti krug koji smo si zadali kao područje nanošenja štete, Gizmos nam nudi svoju pomoć svojim metodama da si sami nacrtamo u prozoru „Scene“. U nastavku prikazujem kako to izgleda na Sparku.



Slika 12: Spark, animacija napada

Prema slici, uz Gizmos možemo lakše podešavati veličinu područja štete. U mom primjeru su dvije Gizmos instance: jedan za šake, drugi za nogu i s time završavamo animacijom napada.

4.2.7. Ozlijeđen animacija

U pravilu kako bi Spark bio ozlijeđen, postoji uvjeti koje mora zadovoljiti kako bi pokrenuo animaciju. U nastavku ću prikazati metodu *OzljedzenAnimacija():*

```
97.  private void OzljedzenAnimacija()
98.      {
99.          if (KadaJeOzljedzen())
100.         {
101.             HPmanager.smanjiHPiliZivot();
102.             if (HPmanager.AkoImaHP())
103.             {
104.                 poluNevidljiv = true;
105.
106.                 rb2.velocity = new Vector2(0, skaci + 5);
107.
108.                 StartCoroutine(PoluNevidljiv(3.5f));
109.             }
110.             else
111.             {
112.                 PlayerGubiZivotAnimacija();
113.             }
114.         }
115.         boli = false;
116.     }
```

Kako bi Spark mogao biti ozlijeđen metoda *KadaJeOzljeđen()* provjerava te uvjete od kojih su: parametar **boli** mora biti **true**, ne smije biti polunevidljiv, ne smije biti na cooldownu. Pod cooldownu se podrazumijeva kada izvodi svoju ultra brzu animaciju, jer tada ne može biti ozlijeđen. Nakon što se utvrdi da je Spark ozlijeđen, utvrđuje se da li Spark ima HP, ako da, oduzet će mu se HP. To utvrđuje skripta „PlayerHP“, spomenutu ranije. Ovdje provjeravamo kako bismo pravilno izveli animaciju. Ako gubi HP, postat će polunevidljiv u suprotnom gubi život te se radi druga animacija. U nastavku prikazujem metodu *Polunevidljiv().*

```
117.      IEnumerator PoluNevidljiv(float trajanje)
118.      {
119.          float pamtiTrajanje = Time.time;
120.          while (true)
121.          {
122.              if (trajanje <= Time.time - pamtiTrajanje)
123.              {
124.                  poluNevidljiv = false;
125.                  break;
126.              }
127.              sprite.color = new Color(1, 1, 1, 0.2f);
128.              yield return new WaitForSeconds(0.3f);
129.              sprite.color = Color.white;
130.              yield return new WaitForSeconds(0.3f);
131.          }
132.      }
```

Već sam ranije spomenuo da IEnumerator nam omogućuje čekanje na izvađanje linije koda uz paralelno funkcioniranje ostalih linija kodova. U ovom primjeru postavljamo timer koji će nam poslužiti da znamo na koliko dugo je Spark polunevidljiv. U **while** petlji svakih 0.3 sekunde, Spark postaje vidljiv pa nakon toga nevidljiv. Ova petlja će trajati sve dok ne prekoračimo dozvoljeno trajanje pri čemu će petlja završiti. U nastavku prikazujem metodu koju koriste svi protivnici kada ozljede Sparka.

```
133.      public void ozljedzen()
134.      {
135.          if (!boli && !poluNevidljiv&&!ONcooldown)
136.          {
137.              animator.SetTrigger("ozljeda");
138.              Invoke("ResetOzljeda", .4f);
139.              boli = true;
140.          }
141.      }
```

U pravilu se pokreće animacija ozljede te se postavlja parametar **boli** na **true**. Razlog zašto sam stavio *ResetOzljeda()* metodu je iz razloga što parametar **trigger** zna potrajati više nego što je potrebno te iz tog razloga zna ponekad istu animaciju ponavljati 2 puta. Ovaj if uvjet koji sam postavio je zapravo nepotreban. To je isti uvjet koji sam postavio kod metode *KadaJeOzljedzen()* te kao rezultat će se ponavljati 2 puta što je nepotrebno.

4.2.8. Skok animacija

Skok sam po sebi nije kompliciran za postaviti, ali ponekad trebamo odlučiti koliki prioritet ima skok u odnosu na druge radnje. To vrijedi i za njenu animaciju kao i Spark-ovo gibanje. U nastavku će biti metoda *SkokAnimacija().*

```
142.      private void SkokAnimacija()
143.      {
144.          PostaviUvjeteSkakanja();
145.          NaKlikSkoci();
146.      }
```

Prvo postavljamo uvjete skakanja. Ako smo na podu tada možemo skakati u suprotnom smijemo iskoristiti dupli skok. Također postavljamo parametre te utvrđujemo da li padamo ili skačemo. To je povezano sa zadnjom i trenutno pozicijom Spark-a. Ako je trenutna pozicija viša od zadnje to znači da Spark ide prema gore stoga ćemo postaviti animaciju skakanja ako ne dira pod, u suprotnom pada. U nastavku je metoda *PostaviUvjeteSkakanja()* i neke od pomoćnih metoda.

```
147.      private void PostaviUvjeteSkakanja()
148.      {
149.          if (JesamLiNaPodu())
150.          {
151.              MoguSkakati();
152.          }
153.          else
154.          {
155.              mozeskakati = false;
156.              PostavljanjeAnimacijePadanjaSkakanja();
157.          }
158.      private void NaKlikSkoci()
159.      {
160.          if (Input.GetKeyDown("space") && HPmanager.AkoImaHP())
161.          {
162.              if (mozeskakati)
163.              {
164.                  Skoci();
165.              }
166.              else if (dupliSkok)
167.              {
168.                  IskoristiDupliSkok();
169.              }
170.          }
171.      }
172.      private void Skoci()
173.      {
174.          mozeskakati = false;
175.          animator.SetBool("zrak", true);
176.          rb2.velocity = new Vector2(rb2.velocity.x, skaci);
177.      }
178.      }
179.      private void IskoristiDupliSkok()
180.      {
181.          dupliSkok = false;
182.          animator.SetBool("pad", false);
183.          animator.SetBool("zrak", true);
184.          rb2.velocity = new Vector2(rb2.velocity.x, skaci);
```

```
185.     }
```

U navedenim primjerima možemo vidjeti parametre **zrak** i **pad**. **Zrak** se odnosi na to da Spark skače dok **pad** se odnosi da pada uz uvjet da ne dira podloug. Bio bi bolji neki drugi naziv umjesto **zrak**, no nema veze. Spark će biti u mogućnosti iskoristiti dupli skok tek, onda kada će **dupliSkok** biti **true**, nakon čega se **dupliSkok** postavlja na **false** kako ne bi više puta iskoristio skok. To isto vrijedi za normalan skok uz varijablu **mozeSkakati**. Također paralelno postavljamo animacije skoka. Kada Spark dotakne tlo, **mozeSkakati** i **dupliSkok** varijable će se resetirati, no i dalje će za dupli skok vrijediti ograničenja kako ga ne bi iskoristili dok je na tlu.

4.2.9. Pucanje animacija

Za animaciju pucanja osim sprite-ova Sparka potreban nam je i sprite-ove za metke. Sprite-ove za metke sam zasebno tražio te ovdje možemo vidjeti iste. Također prilažem sliku i za oružje.

Slika 13: Pištolj izgled

Slika 14: Metci izgled

Budući da sam koristio sprite-ove kao glavnu bazu za izrađivanje animacija Sparka, jednostavno sam i oružje spojio sa sprite-om Sparka. U nastavku ću prikazati liniju koda za pucanje.

```
186. private void PucanjeAnimacija()
187.     {
188.         if (MoguLiPucati())
189.         {
190.             animator.SetBool("pucam", true);
191.             if (JesamLiNaPodu())
192.             {
193.                 rb2.velocity = Vector3.zero;
194.             }
195.             AnimacijaMetka();
196.
197.             Invoke("resetBullet", 0.12f);
198.         }
199.     }
```

Kao što i vidimo na primjeru, kao i sve ostale animacije, kako bismo pucali moramo zadovoljiti određene uvjete. Uglavnom metoda *MoguLiPucati()* provjerava sljedeće uvjete: da li imamo HP, da li ne pucamo trenutno, da li nismo u cooldownu te da li smo pritisnuli tipku za pucanje. Nakon što zadovoljimo uvjete postavljamo parametar **pucam** na **true** i možemo pucati. Budući da nemamo animaciju za pucanje tijekom hodanja, postavio sam da Spark stane kada želi pucati na podu, u zraku se može slobodno kretati. Također smo postavili *resetBullet()* metodu kako bi smo odredili maksimalnu brzinu pucanja za Sparka. U nastavku ću prikazati metodu *AnimacijaMetka().*

```
200.      private void AnimacijaMetka()
201.      {
202.          GameObject metak = (GameObject)Instantiate(metakref);
203.          metak.GetComponent<metakskripta>().StartMetak(obrni);
204.          PozicijaMetka(metak);
205.      }
```

U pravilu mi smo u početku referencirali metak koristeći klasu Resources. To je izvrsna klasa koja nam omogućuje učitavanje objekata iz mape „Resources“ i također pretraživanje drugih mapa. Koristeći metodu *Instantiate()* kopiramo naš objekt koji predstavlja metak i tu kopiju pohranjujemo u novu varijablu koja će predstavljati metak koji pucamo. Na taj način smo napraviti iz razloga da ne moramo više puta referencirati objekt klasom Resources. Varijabla **obrni** nam služi za zrcaljenje Sparka, ali i metka. Od kreiranog objekta dohvaćamo skripte za metke koje ću prikazati u nastavku.

```
1.        public void StartMetak(bool obrni)
2.        {
3.            sprite = GetComponent<SpriteRenderer>();
4.            sprite.sortingOrder = 0;
5.            rb2 = GetComponent<Rigidbody2D>();
6.            rb2.constraints = RigidbodyConstraints2D.FreezePositionY;
7.            if (obrni)
8.            {
9.                rb2.velocity = new Vector2(20, 0);
10.               sprite.flipX = true;
11.           }
12.           else { rb2.velocity = new Vector2(-20, 0); }
13.
14.           Invoke("DestroySelf", .7f);
15.
16.       }
```

U skripti za metke smo trebali definirati u koju stranu će se metak gibati koju određuje varijabla **obrni,** bilo je također potrebno zamrznuti os **y**, kako bi nam metci se kretali isključivo ravno. Ako metak ne doživi sudar, pozvat će se metoda *DestroySelf()* koja će uništiti objekt metak. U nastavku ću još prikazati metodu *OnCollisionEnter2D().*

```
17.       private void OnCollisionEnter2D(Collision2D collision)
18.       {
19.           if (gameObject.name.Contains("metak"))
20.           {
```

```
21.             if (collision.gameObject.CompareTag("Ground") ||
    collision.gameObject.CompareTag("enemy") )
22.             {
23.                  Destroy(gameObject);
24.             }
25.         }
26.         else if (gameObject.name.Contains("metakPcela"))
27.         {
28.             if (collision.gameObject.CompareTag("Ground") ||
    collision.gameObject.CompareTag("Player"))
29.             {
30.                 Destroy(gameObject);
31.             }
32.         }
33.     }
```

Ukratko, metak će se uništiti sa objektom koji na sebi ima oznaku „Ground“ što je zapravo teren isto tako sam postavio i za Sparka kao „player“ i sve protivnike pod oznakom „enemy“. Tu istu skriptu sam koristio za metke od protivnika, u našem primjeru pčele zbog čega sam i preispitivao radi li se trenutno o objektu naziva „metak“ ili „metakPcela“. S ovime završavamo animacijom pucanja.

4.2.10. „UltraBrz“ animacija

„UltraBrz“ animacija je jedna od mojih najdražih animacija kod Sparka, te upravo ta animacija čini ovu igricu zabavnijom. Radi se o animaciji obliku zaleta gdje Spark u jedinici sekunde prođe veliku brzinu.

Slika 15: „UltraBrz“ animacija

U nastavku ću prikazati primjer koda kako sam izveo ovu animaciju.

```
34.     private void UltraBrzaAnimacija()
35.     {
36.         DesniDvoklik();
37.         LijeviDvoklik();
38.     }
39.     private void LijeviDvoklik()
40.     {
41.         if (Input.GetKeyDown("left") && !boli && !ONcooldown)
42.         {
43.             float VrijemeLijevogKlika = Time.time - DvoklikLijevo;
44.             if (VrijemeLijevogKlika < .2f)//ako je dvoklik
45.             {
46.                 PostaviAnimacijuUltraBrz(-1);
```

```
47.                 Invoke("MakniUltraBrzinu", 0.12f);
48.                 Invoke("MakniUltraBrzCD", ultraBrzinaCDSec);
49.             }
50.             DvoklikLijevo = Time.time;
51.         }
52.     }
```

Budući da se ta animacija izvodi dvoklikom lijevo ili desno, trebao sam kreirate dvije zasebne metode za dvije zasebne tipke. Nakon što zadovoljimo uvjete potrebno je provjeriti da li se radi o dvokliku. To radimo varijablom **DvoklikLijevo** koja u sebi pohranjuje vrijeme zadnje pritisnute tipke. Njezinom razlikom sa realnim vremenom nam daje točnu razliku pritiska između dvije tipke te u tom slučaju će se izvesti animacija ultra brzina. U nastavku ću prikazati metodu *PostaviAnimacijuultraBrz().*

```
53.     private void PostaviAnimacijuUltraBrz(int strana)
54.     {
55.         ONcooldown = true;
56.         UltraBrzinaKretanje = true;
57.         animator.SetBool("UltraBrz",true);
58.         animator.SetTrigger("UltraBrzTrigger");
59.         rb2.velocity = new Vector2(ultraBrzina*strana, 0);
60.     }
```

U pravilu kod ove metode trebamo znati stranu okretaja koja će biti isključivo vrijednosti 1 ili -1 koju množimo sa varijablom **ultraBrzina**. Ono što bi moglo biti zbunjujuće da sam kod ove animacije postavio dvije varijable ograničenja, **UltraBrzinaKretanje** i **Oncooldown**. **UltraBrzinaKretanje** se odnosi isključivo za kretanje i ona će samo utjecati na kretanje Sparka, u smislu koliko brzo će proći određeni interval dok **Oncooldown** daje ograničenje „UltraBrz“ animaciji, ali i ostalim animacijama da se izvedu. To je i mene u početku malo zbunilo kada sam to radio, moglo bi se i bolje to napraviti, ali nadam se da se to jasno vidi.

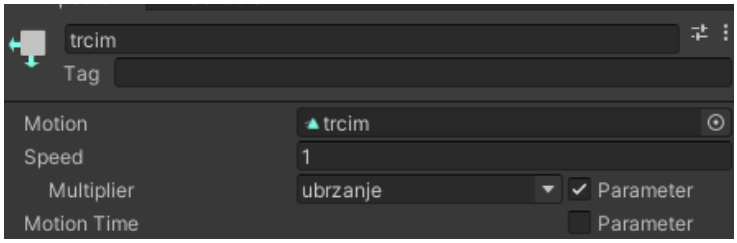
4.2.11. Ostale animacije

Pod ostale animacije sam postavio one animacije koje bi mogle biti manje interesantne. Uglavnom su to brzo trčanje koja je replika od običnog trčanja i smjer okretanja Sparka. U nastavku ću prvo prikazati brzo trčanje, ali neću ulaziti previše u detalje.

```
61.     private void BrzoTrcanjeAnimacija()
62.     {
63.         if
    (mozeBrzoTrcati()&&(MoguLiSeKretatiPoZraku()||MoguLiHodatiNaPodu()))
64.         {
65.             rb2.velocity = new Vector2(horizontala * 2, rb2.velocity.y);
66.             animator.SetFloat("ubrzanje", 3f);
67.         }
68.     }
```

U pravilu za brzo trčanje vrijedi sva pravila kao i za obično trčanje, jedino dodatno treba pridržati tipku više. Kod zadovoljenih uvjeta postavljamo vrijednost parametra **ubrzanje**

na 3 dok je kod obično trčanja na 1. To smo učinili kako bismo koristili tu vrijednost za ubrzavanje same animacije klikom na samu animaciju te postavili vrijednost od „Multiplier“ na **ubrzanje**. To možemo vidjeti na slici.



Slika 16: Podešavanje animacije „trcim“

Sljedeće bi imali metodu *OkreniPlayera()* te da ne duljim u nastavku ću je i prikazati.

```
69.     private void OkreniPlayera()
70.     {
71.         if (Input.GetKey("left"))
72.         {
73.             obrni = false;
74.             sprite.flipX = true;
75.         }
76.         if (Input.GetKey("right"))
77.         {
78.             obrni = true;
79.             sprite.flipX = false;
80.         }
81.         OkreniGizmosNapadanja();
82.     }
```

Svatko ima svoju verziju kako će zrcaliti svog igrača, u mom slučaju postavljam **sprite.flipX** na **true** prema potrebi. Ovo rješenje zadovoljava moje potrebe, ali mnogo bolje bi bilo da umjesto toga postavim **transform.LocalScale.X** na -1 jer u tom slučaju bih zrcalio sve objekte i sve komponente uz Sparka te bih usput zrcalio komponentu „Collider2D“ i Gizmos, ali nema veze.

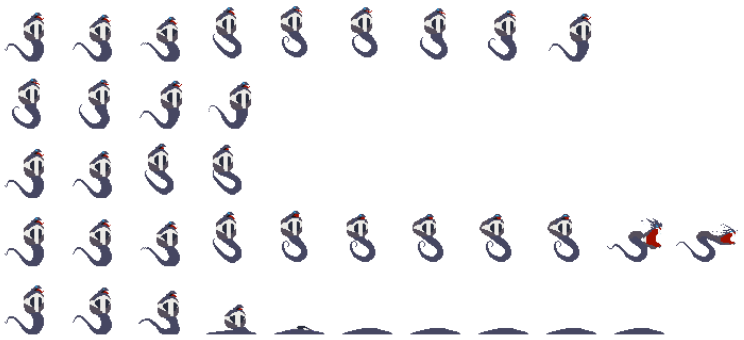
4.3. Neprijatelji

Za svakog neprijatelja sam koristio zasebnu skriptu, ali svi osim skeletona imaju istu skriptu za HP. Razlikuje sljedeće neprijatelje: Zmija, Minotaur, Pčela i Pauk te na kraju i završni neprijatelj Skeleton. Ono što je većina 2D platform igrica radilo je to da pri samom dodiru na neprijatelja, igrač bi gubio HP. U mom primjeru igrač može biti u kontaktu neprijatelja dokle god ne dodiruje njegovo područje napada. Iznimka je kod pauka gdje sam

postavio da odmah gubi HP čisto radi raznolikosti i otežavanju razine. U sljedećim poglavljima ću zasebno opisati neprijatelje.

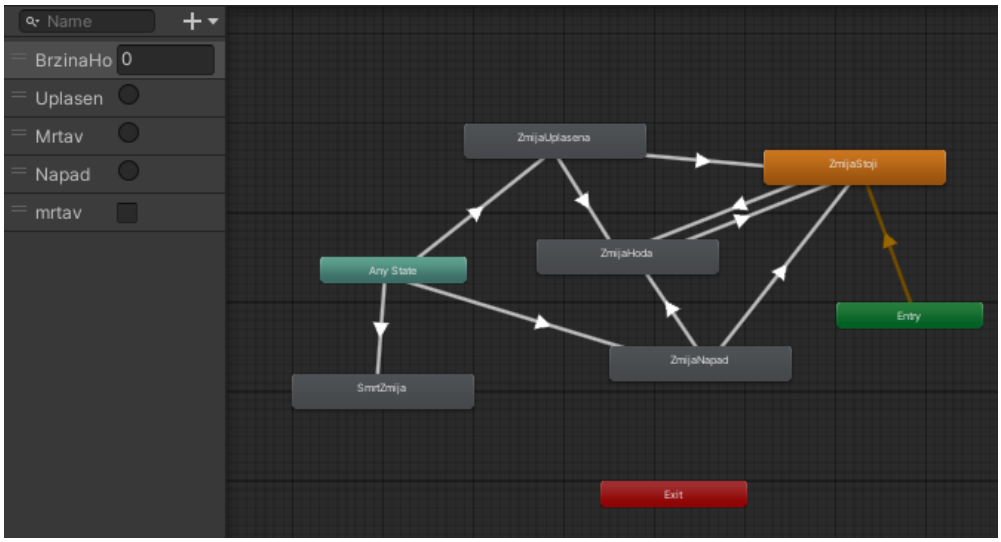
4.3.1. Zmija

Zmija će imati 5 osnovne animacije, a to su : kretanje, stajanje, napadanje, ozljeda i smrt. U nastavku su prikazani sprite-ovi zmije.



Slika 17: Zmija izgled

Isto kao i za Sparka za zmiju kao i za sve protivnike je bilo potrebno ručno dodati komponente: „RigidBody2D“ i „Collider2D“. U nastavku ću prikazati sliku animatora zmije.



Slika 18: Prikaz prozora „Animator“ za zmiju

U pravilu zmija će većinom spontano hodati i stajati tek prilikom kontakta sa Sparkom će proći u animaciju napada i boli pri čemu se natrag vraća na stajanje ili hodanje. U nastavku ću prikazati skriptu zmije.

4.3.1.1. „Zmija“ skripta

U nastavku ću prikazati metode *Start()* i *Update()* metodu nakon čega ću ih objasniti.

```
1.      void Start()
2.      {
3.          rb = GetComponent<Rigidbody2D>();
4.          sprite = GetComponent<SpriteRenderer>();
5.          animator = GetComponent<Animator>();
6.          udarac = hitBox.GetComponent<BoxCollider2D>();
7.          hpManager = GetComponent<EnemyHP>();
8.          sprite.color = BojaZmije;
9.      }
10.     private void Update()
11.     {
12.         PostaviAnimacijuHodanja();
13.         NapadanjePlayera();
14.         PrimanjeStete();
15.         if (KretanjeDopusteno())
16.         {
17.             KrecemSe = true;
18.             StartCoroutine(KreciSe(RandomBroj()));
19.         }
20.     }
```

U početku u *Start()* metodi referenciramo komponente. Također sam pridružio i „EnemyHP“ skriptu koju ću objasniti nakon što prođem kroz sve protivnike. Osim komponenti postavio sam i varijablu **BojaZmije** tipa Color. Ona nam je potrebna pri ozljedi zmije da u djeliću sekunde promijenimo njenu boju u crvenu zatim vratimo natrag. U *Update()* metodi imamo kretanja zmije. Metoda *PostaviAnimacijuHodanja()* nam samo prati da li se naša zmija trenutno kreće te usklađuje parametar sa animacijama. U nastavku ću prikazati metodu *NapadanjePlayera().*

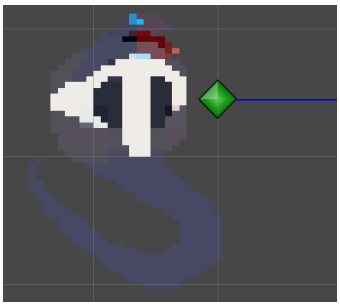
```
21.     private void NapadanjePlayera()
22.     {
23.         if (VidimPlayera() && !Napada && hpManager.ImaLiHP())
24.         {
25.             Napada = true;
26.             animator.SetTrigger("Napad");
27.             rb.velocity = new Vector2(0, rb.velocity.y);
28.             Invoke("NeNapadaj", 1.2f);
29.         }
30.     }
```

Zmija da bi mogla napasti mora proći kroz uvjete, a to su : zmija mora vidjeti igrača, ako trenutno ne napada i ako ima HP. **Napad** postavljamo na **true** kako ne bismo prekinuli animaciju dok napada te zaustavljamo zmiju da se ne kreće. *NeNapadaj()* metoda postavlja **Napada** na **false**, ali zapravo bi se trebala zvati *MozeNapasti().* U nastavku ću prikazati kako utvrditi da li zmija vidi Sparka.

```
31.     private bool VidimPlayera()
32.     {
33.         Vector2 granicaVidika = OdrediVidik();
34.         RaycastHit2D dodir =
    Physics2D.Linecast(vidok.transform.position,granicaVidika, 1 <<
    LayerMask.NameToLayer("Action"));
```

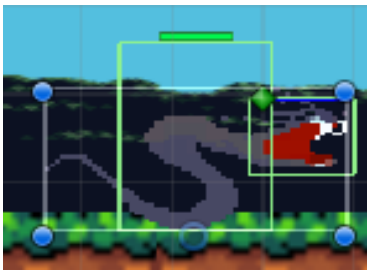
```
35.           if (dodir.collider && hpManager.ImaLiHP())
36.           {
37.               if (dodir.collider.tag == "Player")
38.               {
39.                   return true;
40.               }
41.           }
42.           Debug.DrawLine(vidok.transform.position,granicaVidika, Color.blue);
43.           return false;
44.       }
```

Da bismo utvrdili da li zmija može gledati, prvo trebao odrediti njezino područje gledanja. Kod zmije sam dodao dodatni dijete objekt koji će nam pomoći oko toga. Mi u pravilu trebamo odabrati neku proizvoljnu udaljenost od objekta te povući liniju. *OdrediVidik()* metoda nam samo vodi računa ako je zmija okrenuta da okrene i tu liniju. Unity ima korisnu metodu Physics2D.*Linecast()* pomoću koje vučemo pravac između dvije točke i mi smo postavili poziciju objekta i proizvoljnu poziciju koja će reagirati na layer „Action“. Ta metoda vraća vrijednost tipa RaycastHit2D koju ja pohranjujem u zasebnu varijablu dodir. Na temelju te varijable utvrđujemo, postoji li kolizija između naše zamišljene linije i Sparka. Debug.*DrawLine()* nam omogućuje da nacrtamo tu liniju koju ću prikazati.



Slika 19: Prikaz plave linije kod zmije

Prilikom napadanja, kod zmije sam postavio i drugi „Collider2D“ koji će biti „Trigger“. „Trigger“ će nam omogućiti da ne radimo nikakvu koliziju sa objektima već da nam javi kada dođe u kontaktu. Taj collider će nam predstavljati područje štete te će se isključiti prema potrebi. Što možemo vidjeti na slici.



Slika 20: Područje napada kod zmije

Kako sam taj collider postavio kao „Trigger“, to mi omogućuje korištenje metode *OnTriggerEnter2D()* kada će se dogoditi nekakav kontakt. U nastavku prikazujem kod.

```
45.     private void OnTriggerEnter2D(Collider2D collision)
46.     {
47.         if (collision.tag == "Player")
48.         {
49.             collision.GetComponent<PlayerKontroler>().ozljedzen();
50.         }
51.     }
```

Metoda je poprilično jednostavna, pri kontaktu sa objektom naznakom „Player“ koristi njegovu skriptu i metodu *ozljedzen().* Nisu potrebna dodatna objašnjenja za tu metodu budući da sam ju prije objasnio. Također collider koji predstavlja područje štete, će se isključiti pri završetku animacije i te metode sam upravo i koristio kod animacija. Metode su vrlo jednostavne stoga ih neću prikazati. U nastavku ću još prikazati kretanje zmije.

```
52.     private IEnumerator KreciSe(int broj)
53.     {
54.         switch (broj)
55.         {
56.             case 1:
57.                 {
58.                     HodamLijevo();
59.                     break;
60.                 }
61.             case 2:
62.                 {
63.                     HodamDesno();
64.                     break;
65.                 }
66.             case 3:
67.                 {
68.                     Stojim();
69.                     break;
70.                 }
71.         }
72.         yield return new WaitForSeconds(RandomBroj());
73.         KrecemSe=false;
74.     }
```

KreciSe() metoda je tipa IEnumerator jer sam htio da u određenom vremenu zmija promijeni animaciju. Varijabla **broj** će odrediti gdje će se kretati zmija i taj broj sam stavio da

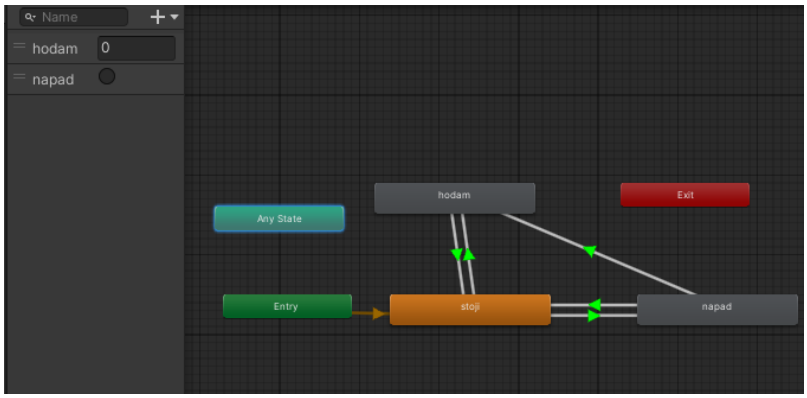
je nasumičan. Kao rezultat zmija će se kretati potpuno nasumično. Varijabla **KrecemSe** sam koristio kako bih sinkronizirao ulazak u metodu.

4.3.2. Minotaur

Osim po izgledu minotaur se razlikuje i po načinu kretanja te mu nisam stavio collider kao područje napada, već sam se služio gizmosima isto kao i kod Sparka. U igri sam kombinirao collidere i gizmose čisto radi raznolikosti da mogu vidjeti razliku između njih. U nastavku ću prikazati sprite minotaura.

Slika 21: Izgled minotaura

Za razliku od zmije, minotaur ima samo tri animacije: stajanje, hodanje i napadanje. U nastavku ću prikazati sliku animatora.



Slika 22: Prikaz prozora „Animator“ za minotaura

U odnosu na prošlog protivnika, minotaur nema animaciju umiranja niti za bol stoga je i animator puno jednostavniji. Animacija za bol mi nije toliko bitna jer mogu jednostavno promijeniti njegovu boju kao znak udarca, dok kod umiranja je drugačija stvar. Kod umiranja

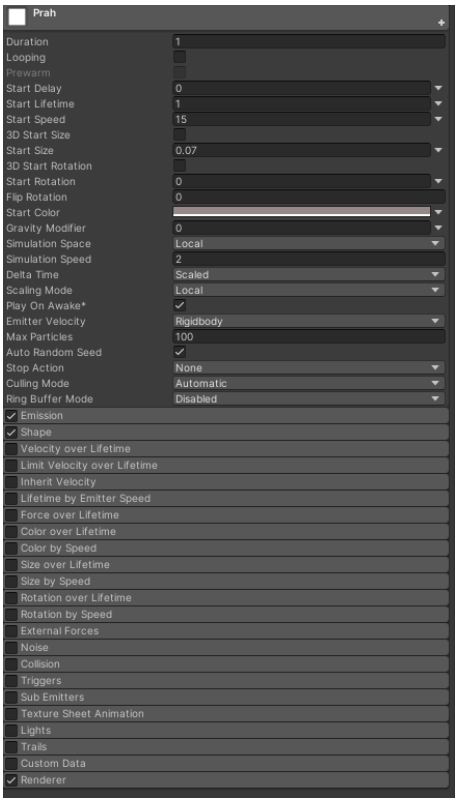
sam dodao još jedan objekt koji će se pojaviti tijekom njegove smrti, no o tome ću još kasnije pričati.

4.3.2.1. Skripta minotaura

U nastavku prikazujem skriptu minotaura.

```
75.      void Start()
76.      {
77.          FlipanjeMinotaura = transform.localScale;
78.          smjerKretanja = DESNO;
79.          animator = GetComponent<Animator>();
80.          rb2 = GetComponent<Rigidbody2D>();
81.          sprite = GetComponent<SpriteRenderer>();
82.          hpManager = GetComponent<EnemyHP>();
83.          prahReferenca = Resources.Load("Prah");
84.      }
85.      private void FixedUpdate()
86.      {
87.          OdrediSmjerKretanja();
88.          AnimacijaHodanja();
89.          AnimacijaNapadanja();
90.          AnimacijaPrimanjeStete();
91.      }
```

Evo kod metode Start() dohvaćamo osnovne komponente, te onim komponenta, varijabli i skripta također dohvaćamo objekt zvan prahReferenca. Kako sam prije napomenuo da nemamo animaciju za smrt, postavljanje animacije praha nam rješava taj problem. U metodi FixedUpdate() sam postavio metode za kretanje kao i određivanja smjera. Koristio sam FixedUpdate() umjesto Update(), ali to mi ne čini toliko razlike. FixedUpdate() će se pokrenuti više puta po frame-u za razliku od Update(). Budući da neću prikazati metodu za umiranje zbog jednostavnosti, jednostavno ću ukratko prikazati „Prah“ objekt, njezinu komponentu u sljedećoj slici.



Slika 23: Komponenta „Particle System“

Komponenta „Particle System“ je još jedna jako zanimljiva komponenta Unity-a, a ona u pravilu prikazuje animirajuće čestice. Kao na slici vidimo da imamo vrlo mnogo izbora za oblikovanje njihovog ponašanja i izgleda. Kreirao sam prazan objekt koji sadrži tu komponentu gdje sam posložio opcije da može dati efekt nekakvog praha sa kojim sam se poslužio da prikažem smrt minotaura. U nastavku vidimo rezultate ovog efekta.

Slika 24: Poraz minotaura

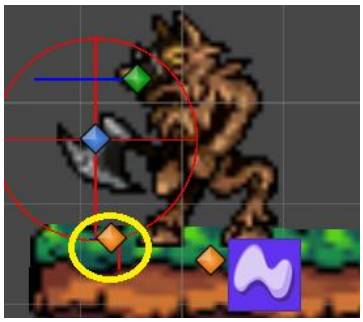
Objekt praha se kreira na potpuno isti način kao i metak kod Sparka stoga ga neću prikazati. No uglavnom valja napomenuti da se prah pojavljuje odmah kada uništimo objekt minotaur te nakon sekunde uništimo i prah objekt. Sljedeće ću prikazati metodu AkoDiramZidRub().

```
92.      private void AkoDiramZidRub()
93.      {
94.          if ((DodirujemLiZid() || JesamLiNaRubu()) && JesamLiNaPodu())
95.          {
96.              if (smjerKretanja == LEVO)
97.              {
98.                  ZrcaljenjeMinotaura(DESNO);
99.              }
100.             else
101.             {
102.                 ZrcaljenjeMinotaura(LEVO);
103.             }
104.         }
105.     }
```

Ova metoda uglavnom određuje kada će se minotaur okrenuti. U pravilu kod minotaura sam postavio tri prazna objekta, jedan je za provjeru poda, drugi provjerava rub terena i zid dok treći provjerava da li se nalazi Spark u blizini. U ovom primjeru koda provjeravamo rub terena ili zid koristeći drugi objekt. Budući da su metode *DodirujemLiZid()* i *JesamLiNaRubu()* poprilično slične pokazat ću samo jednu od njih u nastavku.

```
106.     private bool DodirujemLiZid()
107.     {
108.
109.         float udaljenostZida = udaljenost;
110.
111.         if (smjerKretanja == LEVO)
112.         {
113.             udaljenostZida = -udaljenost;
114.         }
115.
116.         Vector3 DodirZida = ground.position;
117.         DodirZida.x += udaljenostZida;
118.
119.         Debug.DrawLine(ground.position, DodirZida, Color.blue);
120.         if (Physics2D.Linecast(ground.position, DodirZida, 1 <<
     LayerMask.NameToLayer("Ground")))
121.         {
122.             return true;
123.         }
124.         return false;
125.     }
```

U početku pohranimo našu proizvoljnu duljinu vidika koju sam nazvao **udaljenost**, a varijablu za pohranu **udaljenostZida**. Novom vektoru **DodirZida** ćemo pridružiti tu vrijednost i nacrtati liniju od pozicije našeg objekta do pozicije vektora **DodirZida**. Kao što sam to radio i kod zmije, koristimo Physics2D.*LineCast()* koja će nam vratiti **true** ako dodirujemo zid, a u suprotnom **false**, te će se naš minotaur okrenuti po potrebi. Dodatno napominjem da je potrebno postaviti layer za sav teren pod „Ground“. Na slični način primjenjujemo za dodir ruba terena, samo što vrijednost ne pridodajemo os **x** kao u ovom primjeru već os **y**. U nastavku ću prikazati kako ove linije izgledaju kod minotaura.



Slika 25: Prikaz objekta s linijom kod minotaura

Vrlo slabo se vidi međutim stavio sam žutu kružnicu radi lakšeg snalaženja. Crvena linija provjerava rub terena, dok plava linija kod istog objekta provjerava rub zida. Drugi objekt koji isto provjerava teren sam postavio čisto radi provjere da li uopće dodirujemo pod kako bih postavio kao dodatno ograničenje. U suprotnom, ako se desi da će naš minotaur naprimjer biti u zraku, okretat će se beskonačno brzo što nije baš divno za vidjeti. Objekt koji je do glave minotaura sa plavom linijom, javlja ako vidi Sparka, budući da sam već kod zmije prikazao, neću prikazati ponovo. U nastavku ću prikazati kako sam izveo napadanje kod minotaura.

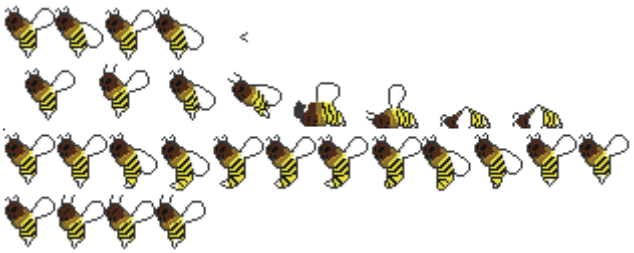
```
126.     private void AnimacijaNapadanja()
127.     {
128.         if (VidimPlayera() && !napada)
129.         {
130.             napada = true;
131.             rb2.velocity = new Vector2(0, rb2.velocity.y);
132.             Invoke("napad", 0.5f);
133.         }
134.     }
135.     private void napad()
136.     {
137.         animator.SetTrigger("napad");
138.         Invoke("Udarac",0.4f);
139.         napada = false;
140.     }
141.     private void Udarac()
142.     {
143.         Collider2D[] udarioPlayera =
     Physics2D.OverlapCircleAll(udarac.transform.position, radius, playerLayer);
144.         foreach (Collider2D player in udarioPlayera)
145.         {
146.             if (player.CompareTag("Player"))
147.             {
148.                 player.GetComponent<PlayerKontroler>().ozljedzen();
149.             }
150.         }
151.     }
```

Napadanje je nešto slično kao i kod Sparka jer sam koristio Gizmos, uglavnom minotaur će napasti samo ako vidi Sparka pri čemu pokreće svoju metodu *napad()*. Kod metode *napad()* će započeti animaciju napada te kroz pola sekunde će se stvoriti područje

štete. To područje štete što smo ga nacrtali pomoću Gizmos klasom što će provjeriti da li smo oštetili Sparka te koristeći njegovu skriptu ćemo smanjiti mu HP. Ponavljam da se kod Sparka u svim slučajevima štete smanjuje HP za 1 stoga nije bilo potrebno definirati količinu štete. Primanje štete kod minotaura je potpuno isto kao i kod zmije stoga neću ga prikazivati. Još da napomenem, parametar **napad** sam koristio kako bih minotauru ograničio ostale pokrete.

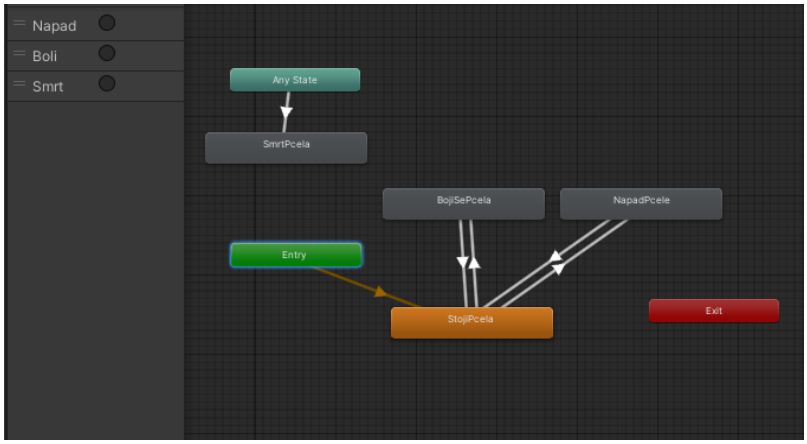
4.3.3. Pčela

Pčela je jedini protivnik koji može letjeti i pucati iz daljine. Budući da sam prikazao pucanje kod Sparka, neću ga ponovo objašnjavati kod pčele jer je otprilike isto. U nastavku ću prikazati sprite-ove pčele.



Slika 26: Izgled pčele

Pčela se sastoji od 4 animacija: letenje, pucanje, umiranje i primanje štete. U nastavku ću prikazati hijerarhiju animacije.



Slika 27: Prikaz prozora „Animator“ za pčelu

Animacije su poprilično jednostavne, pčela će većinom koristiti animaciju stajanja gdje u pravilu samo lebdi. Ostale animacije će se aktivirati pri kontaktu sa Sparkom.

4.3.3.1. „Pčela“ skripta

U nastavku ću prikazati *Start()* i *Update()* metode.

```
1.      void Start()
2.      {
3.          metakReferenca = Resources.Load("metakPcela");
4.          rb2 = GetComponent<Rigidbody2D>();
5.          animator = GetComponent<Animator>();
6.          sprite = GetComponent<SpriteRenderer>();
7.          hpManager = GetComponent<EnemyHP>();
8.          pocetnaPozicija = transform.position.y;
9.      }
10.     void Update()
11.     {
12.         if (hpManager.ImaLiHP())
13.         {
14.             AnimacijaNapadanje();
15.             AnimacijaKretanja();
16.             OkreniSePremaPLayeru();
17.             OdrediPutanjuKretanja();
18.         }
19.         AnimacijaPrimanjaStete();
20.
21.     }
```

Osim komponenti također smo referencirali objekt za metke kao i kod Sparka u *Start()* metodi. U *Update()* metodi imamo primjer izvađanja svih animacija. U nastavku ću prikazati kretanje pčele.

```
22.     private void AnimacijaKretanja()
23.     {
24.         if (gore)
25.         {
26.             rb2.velocity = new Vector2(rb2.velocity.x, brzina);
27.         }
28.         else
29.         {
30.             rb2.velocity = new Vector2(rb2.velocity.x, -brzina);
31.         }
32.     }
```

Kretanje je poprilično jednostavno, gleda se stanje parametra **gore** te će se pčela micati gore-dolje. U nastavku ću prikazati bitniju metodu *OdrediPutanjuKretanja().*

```
33.     private void OdrediPutanjuKretanja()
34.     {
35.         if (transform.position.y + razlika < pocetnaPozicija)
36.         {
37.             gore = true;
38.         }
39.         else if (transform.position.y - razlika > pocetnaPozicija )
40.         {
41.             gore = false;
42.         }
43.     }
```

Varijabla **pocetnaPozicija** smo definirali još kod *Start()* metode gdje smo pohranili poziciju pčele pri učitavanju skripte. Ta će se vrijednost uvijek biti konstanta. Razlika je

proizvoljna veličina koja će definirati koliko daleko će se pčela kretati gore-dolje, te ako prelazi gornju-donju granicu promijenit će se vrijednost parametra **gore**, a i time smjer kretanja. U nastavku ću prikazati napadanje pčele.

```
44.      private void AnimacijaNapadanje()
45.      {
46.          if (VidimPlayera())
47.          {
48.              animator.SetTrigger("Napad");
49.          }
50.      }
51.      private void Pucaj()
52.      {
53.          GameObject metak = (GameObject)Instantiate(metakReferenca);
54.          metak.GetComponent<metakskripta>().StartMetak(sprite.flipX);
55.          metak.transform.SetParent(transform);
56.          OkreniMetak(metak);
57.      }
```

Kod napadanja Sparka, utvrđuje se, da li ga pčela uopće vidi, već sam prije pokazivao kod minotaura i zmije metodu *VidimPlayera()* stoga ju neću prikazati ponovo. Postavit će se animacija napadanja, a metoda *Pucaj()* će se isključivo pokrenuti kod animacije pucanja gdje sam tu metodu i postavio. Na taj način pčela neće pucati u obliku strojnice. U primjeru ću prikazati sliku pčele kako napada.

Slika 28: Prikaz napadanja pčele

Kada dođe do kolizije između plave linije pčele i Sparka, aktivirat će se parametar **trigger** za napad pri čemu će pčela ispaliti metak. U nastavku ću prikazati metode kada pčela prima štetu.

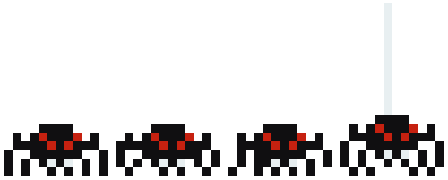
```
58.      private void AnimacijaPrimanjaStete()
59.      {
60.          if (hpManager.MozeAnimiratiStetu)
61.          {
62.              Potresi();
63.              hpManager.MozeAnimiratiStetu = false;
64.              StaviCrvenuBoju();
65.              if (hpManager.ImaLiHP())
66.              {
67.                  EnemyPrimaStetu();
68.              }
69.              else
```

```
70.             {
71.                 EnemyUmire();
72.             }
73.         }
74.     }
75.     private void EnemyUmire()
76.     {
77.         AnimacijaSmrti();
78.         DobivanjeBodova();
79.     }
80.
81.     private void AnimacijaSmrti()
82.     {
83.         animator.SetTrigger("Smrt");
84.         rb2.gravityScale = 1;
85.         gameObject.layer = 10;
86.         hpManager.UnistiHPSlider();
87.         rb2.velocity = Vector2.zero;
88.     }
```

U pravilu ako pčela nije mrtva poprimit će crvenu boju i pokrenuti metodu *Potresi(),* te će pokrenut animaciju primanja štete. *Potresi()* metodu ću objasniti nešto kasnije kada sve protivnike objasnim. Ako pčela nema HP pokrenut će se metoda za umiranje budući da sam postavio pčelu da na nju ne utječe gravitacija, kod smrti sam postavio na 1 kako bi pala na tlo. Indeks layera za pčelu postavljamo na 10 da ne može imati kontakt sa Sparkom, zaustavimo njeno kretanje i s time smo imitirali smrt pčele. *DobivaBodove()* metodu ću također naknadno objasniti.

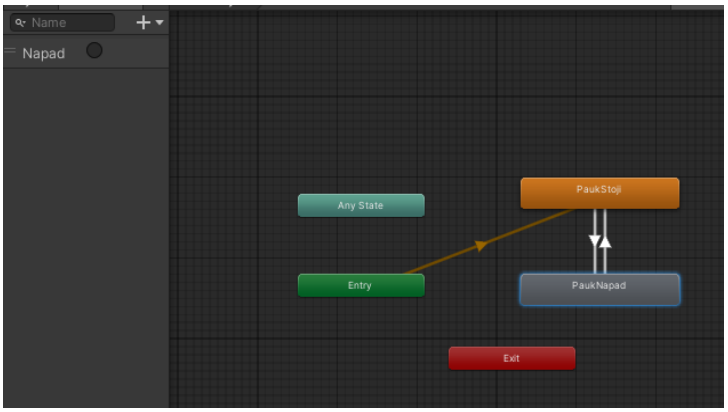
4.3.4. Pauk

Pauk je jedini protivnik na kojeg udarci nemaju nikakvog efekta te ga je potrebno poraziti za pištoljem. On također pri kontaktu automatski nanosi Sparku štetu. Mogli bismo reći da je cijeli collider od pauka „Trigger“ čija metoda će zadati štetu Sparku. Budući da cijeli collider je „Trigger“, pauk može hodati kroz zidove. Prikazat ću sprite-ove pauka.



Slika 29: Izgled pauka

Pauk u pravilu ima dvije animacije, hodanje i spuštanje mrežom. U nastavku ću prikazati animator.



Slika 30: Prikaz prozora „Animator“ za pauka

Animator pauka je poprilično jednostavan jedino je važno za naglasiti da se napad odnosi na spuštanje mrežom. Pauk će u svakom kontaktu sa Sparkom nanositi štetu.

4.3.4.1. „Pauk“ skripta

U nastavku ću prikazati Start() i Update() metodu.

```
1.      void Start()
2.      {
3.          rb = GetComponent<Rigidbody2D>();
4.          animataor = GetComponent<Animator>();
5.          sprite = GetComponent<SpriteRenderer>();
6.          pocetnaPozicija = transform.position.x;
7.          prahReferenca = Resources.Load("Prah");
8.          hpManager = GetComponent<EnemyHP>();
9.      }
10.
11.     void Update()
12.     {
13.         NasumicniBroj = UnityEngine.Random.Range(1,5);
14.         AnimacijaKretanje();
15.         AnimacijaNapadanje();
16.         AnimacijaPrimanjeStete();
17.     }
```

U *Start()* metodi dohvaćamo komponente, skripte i objekte koje su nam potrebne. „Prah“ sam već objašnjavao stoga ga neću ovdje prikazati. U pravilu kao i kod zmije u *Update()* metodi kod pauka sam postavio nasumični broj koji će predstavljati vrijeme napadanja. Na taj način Spark ne može očekivati napad od pauka. Isto kao i kod pčele, postavio sam kretanje na način da pohranim početnu poziciju i kreiram razliku koju ne smije prekoračiti, u suprotnom mijenja smjer. Neću je prikazivati kako je sličan kodu kod pčele, razlika je što mijenjamo os **x** umjesto os **y**. U nastavku ću prikazati napadanje pauka.

```
18.     private void AnimacijaNapadanje()
19.     {
20.         Napadni();
21.         if (SinkronizacijaNapadanja)
22.         {
```

```
            SinkronizacijaNapadanja = false;
            StartCoroutine(OdrediVrijemeNapadanja(NasumicniBroj));
        }
    }
    private IEnumerator OdrediVrijemeNapadanja(float broj)
    {
        yield return new WaitForSeconds(broj);
        MozeNapasti = true;
        SinkronizacijaNapadanja = true;
    }
```

Kao što sam prije rekao, pauk će nasumično napasti na način da uđe u metodu *OdrediVrijemeNapadanja()* koja će nakon određenih broj sekundi kroz varijablu **broj** postaviti parametre za napadanje. U nastavku ću prikazati ostale metode napadanja.

```
    private void Napadni()
    {
        if (MozeNapasti)
        {
            rb.velocity = Vector2.zero;
            IzvediNapad();
        }
    }
    private void IzvediNapad()
    {
        MozeNapasti = false;
        napad = true;
        animataor.SetTrigger("Napad");
        Invoke("NastaviSeKretati", 0.56f);
    }
```

Napad koji aktiviramo kod parametra **trigger** se odnosi na spuštanje sa mreže, pauk može pri svakom kontaktu nauditi Sparku. Kada izvodi napad potrebno je zaustaviti kretanje pauka te postavljamo *Invoke()* metodu kako bismo nakon pola sekundi zaustavili napad. Varijabla **napad** se postavlja na **false** te se pauk može nastavljati kretati.

4.3.5. Kada se protivnik trese

Kod svakog protivnika sam postavio metodu *Potresi().* Pri svakom nanošenju štete ova metoda će se aktivirati. U nastavku ću prikazati njene metode.

```
    private void Potresi()
    {
        pozicijaPrijeTresenja = transform.position;
        TreseSe = true;

        if (TreseSe)
        {
            transform.position = pozicijaPrijeTresenja +
UnityEngine.Random.insideUnitCircle * new Vector2(0.5f, 1f);
            Invoke("ResetTresi", 0.2f);
        }
    }
    private void ResetTresi()
    {
        transform.position = pozicijaPrijeTresenja;
```

```
            TreseSe = false;
        }
```

Smisao te metode je da osim što promijenimo boju protivniku pri nanošenju štete, želimo mu malo poremetiti poziciju. To postižemo sa klasom Random, njenom varijablom **insideUnitCyrcle**. Ova varijabla će nam vratiti nasumičnu poziciju unutar nekog kruga te sam u svom primjeru postavio neki vektor sa kojim će se množiti kako bismo povećali intenzitet i zbrojili sa trenutnom pozicijom. Bitno je da nakon određenog vremena vratimo protivnika na početnu poziciju kako se ne bi cijelo vrijeme vrtio u krug ili da ne zaglavi u nekom objektu.

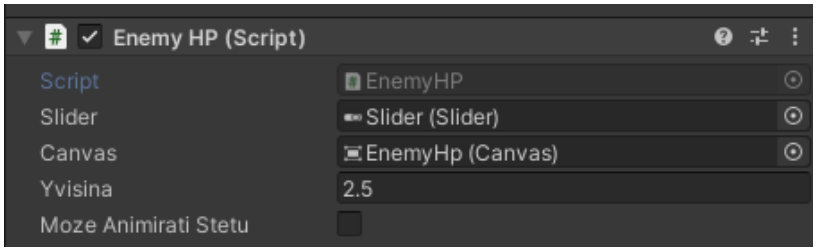
4.3.6. „EnemyHP“ skripta

„EnemyHP“ skriptu sam koristio na svakom protivniku osim skeletona. U nastavku ću prikazati cijelu skriptu.

```
    [SerializeField]
    Slider slider;

    [SerializeField]
    Canvas canvas;

    [SerializeField]
    private float Yvisina;

    Slider HPslider;
    private float hp = 1;
    public bool MozeAnimiratiStetu = false;
    void Start()
    {
        KreairajSlider();
    }
    private void Update()
    {
        PozicionirajHPslider();
    }
    public bool ImaLiHP()
    {
        if (hp < 0)
        {
            return false;
        }
        return true;
    }
    public void SmanjiHP(float damage)
    {
        MozeAnimiratiStetu = true;
        hp -= damage;
        HPslider.value = hp;
    }
    private void KreairajSlider()
    {
        if (slider != null && canvas!=null)
        {
            HPslider = Instantiate(slider);
            HPslider.transform.SetParent(canvas.transform);
```

```
41.             HPslider.gameObject.SetActive(true);
42.         }
43.
44.     }
45.     private void PozicionirajHPslider()
46.     {
47.         if (HPslider != null)
48.         {
49.             HPslider.transform.position = new Vector3(transform.position.x,
    transform.position.y + Yvisina, transform.position.z);
50.         }
51.     }
52.     public void UnistiHPSlider()
53.     {
54.         Destroy(HPslider.gameObject);
55.     }
```

Ova skripta ima poprilično jednostavne metode, jedino bi za početak bilo dobro objasniti objekt **HPslider**. U *Start()* metodi će se pokrenuti *KreirajHPSlider()* koji će kreirati objekt ***slider*** tipa Slider za svakog protivnika iznad njihovog sprite-a. Kod kreiranja objekta **slider** bilo je potrebno unaprijed kreirati objekt tipa Canvas i unutar njega postaviti objekt **slider** koje sam referencirao. Referencu sam prekopirao u novu varijablu i nju sam posebno smjestio u objekt **canvas**. Razlog tome je da za sve UI objekte je potrebno postaviti unutar **canvasa** kako bi oni mogli biti vidljivi. Taj **HPslider** koji smo kreirali će se uvijek ažurirati pri nanošenju štete. U *Update()* metodi samo postavio *PozicionirajHPSlider()* kako bi slider cijelo vrijeme pratio protivnika. Nakon što protivniku ponestane **HPslider** će se automatski uništiti. Ono što bi bilo mnogo bolje jest da sam postavio *PozicionirajHPSlider()* u *FixedUpdate()* metodi, jer mi želimo cijelo vrijeme da se ažurira pozicija **HPslidera**, ovako će malo izgledati kao da šteka kada prati protivnika. Sve ostale metode su poprilično jednostavne stoga ih nema smisla objašnjavati. U nastavku ću još prikazati sliku skripte.

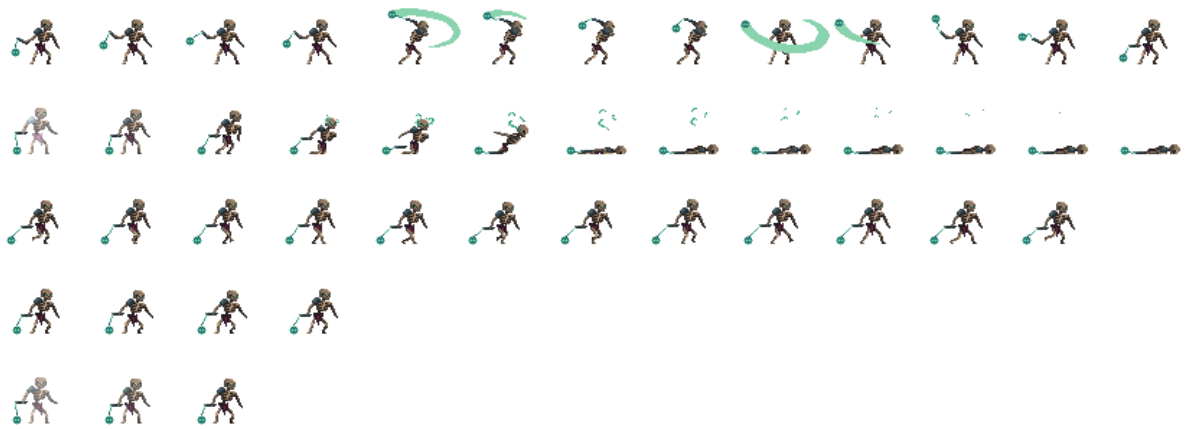


Slika 31: Izgled komponente sa skriptom „EnemyHP“

Sliku skripte sam postavio čisto da možemo vidjeti da je unutar njega potrebno postaviti objekte **canvas** i **slider** koje će skripta kopirati u svoju korist. **Yvisina** je visina na koju će biti postavljen **slider**.

4.3.7. Skeleton

Skeleton je posljednji protivnik („Final Boss“) koji se pojavljuje na kraju 2 razine. Budući da za skeletona želimo da se kreće malo „naprednije“ u odnosu na ostale protivnike, koristit ćemo „StateMachineBehaviour“. To su u pravilu metode koje se izvode unutar pojedine animacije. U nastavku ću prikazati sprite-ove skeletona.



Slika 32: Izgled skeletona

Animacije koje ću koristiti kod skeletona jesu: napadanje, čekanje, hodanje, primanje štete i smrt. U nastavku ću objasniti najvažnije skripte, neću sve jer su vrlo jednostavne i uglavnom služe samo za postavljanje parametara drugih klasa.

4.3.7.1. Skripte skeletona

U nastavku ću prikazati „CekajNaPlayera“ skriptu.

```
1.  public class CekajNaPlayera : StateMachineBehaviour
2.  {
3.      Vector2 pozicijaBossa;
4.      override public void OnStateUpdate(Animator animator, AnimatorStateInfo
    stateInfo, int layerIndex)
5.      {
6.          pozicijaBossa =
    Camera.main.WorldToScreenPoint(animator.transform.position);
7.          if (Screen.safeArea.Contains(pozicijaBossa))
8.          {
9.              animator.SetTrigger("Hodaj");
10.         }
11.     }
12.     public override void OnStateExit(Animator animator, AnimatorStateInfo
    stateInfo, int layerIndex)
13.     {
14.         BossHP.HpSlider.gameObject.SetActive(true);
15.     }
16.
17. }
```

Ukratko, prva metoda će se izvesti jednom čim se animacija pokrene, druga metoda će se izvesti tijekom izvođenja animacije i posljednja metoda se izvodi jednom prije izlaska. Ono što sam uradio ovdje jest, da skeleton izvodi svoju animaciju čekanja sve dok ne upadne pod ekran kamere pri čemu počne pratiti Sparka i kreira mu se **HPslider**. U nastavku ću prikazati skriptu „PratiPlayera“.

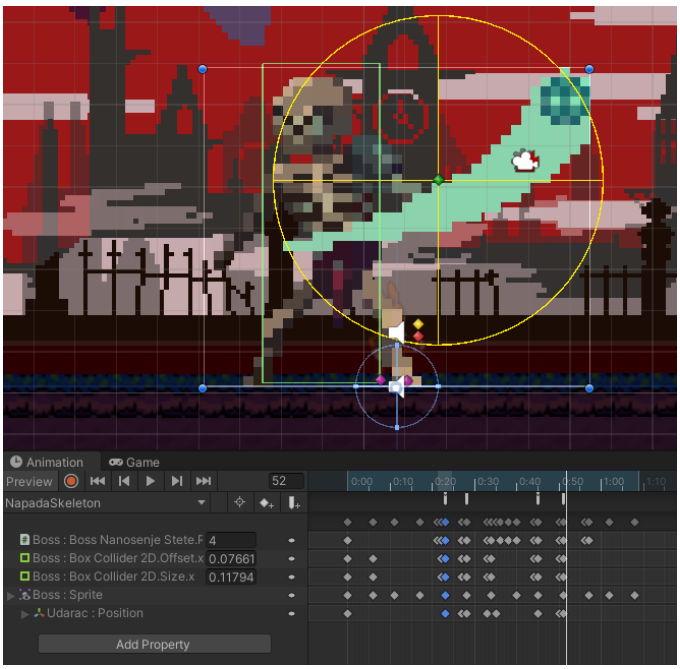
```
1.  public class PratiPlayera : StateMachineBehaviour
2.  {
3.      Transform boss;
4.      Transform player;
5.      private float udaljenost;
6.      private Vector2 kretanjePremaPlayeru;
7.      Rigidbody2D rb;
8.
9.      [SerializeField]
10.     float RangeNapadanja;
11.
12.     [SerializeField]
13.     private float brzina;
14.     override public void OnStateEnter(Animator animator, AnimatorStateInfo
    stateInfo, int layerIndex)
15.     {
16.         boss = animator.transform;
17.         player = GameObject.FindGameObjectWithTag("Player").transform;
18.         rb = animator.GetComponent<Rigidbody2D>();
19.     }
20.     override public void OnStateUpdate(Animator animator, AnimatorStateInfo
    stateInfo, int layerIndex)
21.     {
22.         udaljenost = Vector2.Distance(player.position, boss.position);
23.         Vector2 meta = new Vector2(player.position.x, boss.position.y);
24.         kretanjePremaPlayeru = Vector2.MoveTowards(boss.position, meta,
    Time.fixedDeltaTime * brzina);
25.         AkoDodzemDoPlayera(animator);
26.
27.     }
28.     private void AkoDodzemDoPlayera(Animator animator)
29.     {
30.         if (udaljenost <= RangeNapadanja)
31.         {
32.             animator.SetTrigger("Napad");
33.         }
34.         else
35.         {
36.             rb.MovePosition(kretanjePremaPlayeru);
37.         }
38.     }
39.     override public void OnStateExit(Animator animator, AnimatorStateInfo
    stateInfo, int layerIndex)
40.     {
41.         animator.ResetTrigger("Napad");
42.     }
43. }
```

U metodi *OnStateEnter()* dohvaćamo poziciju Sparka i skeletona kao i komponentu. Budući da u ovoj skripti skeleton prati Sparka, potrebno je odrediti udaljenost između njih što radimo pomoću metode Vector2.Distance() te želimo da se skeleton postepeno kreće prema

Sparku što postižemo Vector2.*MoveTowards()* metodom. Vrijednost koju dobivamo pohranimo u metodi *MovePosition()* kako bi pomaknuli skeletona. Također sam odredio neku udaljenost koja predstavlja maksimalnu udaljenost koju može imati skeleton prema Sparku da bi se mogla pokrenuti animacija napadanja. Kod skripte za upravljanje HP-om je u pravilu sve isto jedino valja napomenuti sljedeće metode.

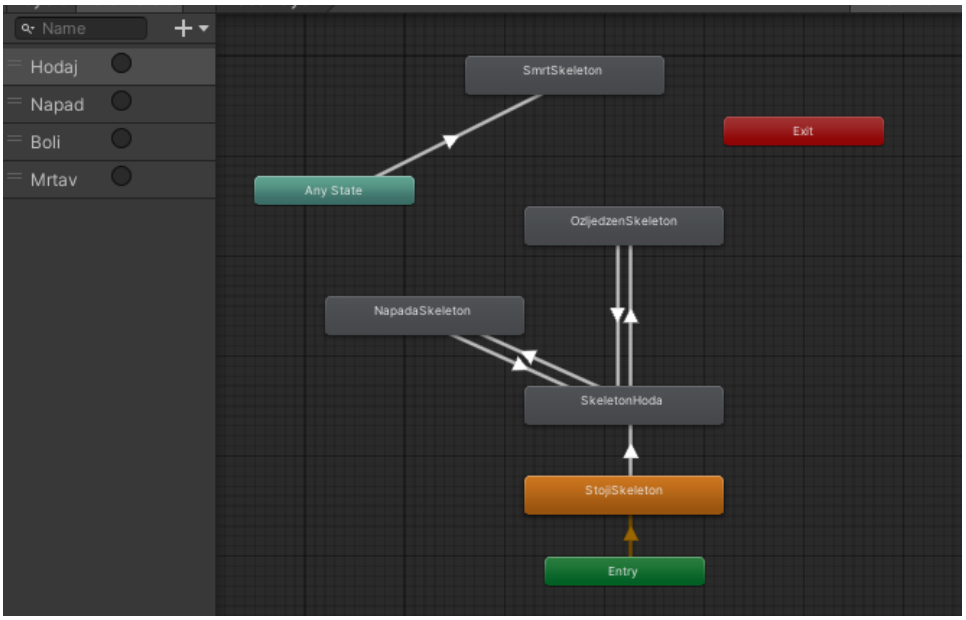
```
44.      private void GameFinished()
45.      {
46.          if (!CoroutineSinkronizacija)
47.          {
48.              CoroutineSinkronizacija = true;
49.              StartCoroutine(NakonPorazaNaMenu());
50.          }
51.      }
52.      private IEnumerator NakonPorazaNaMenu()
53.      {
54.          yield return new WaitForSeconds(3f);
55.          mrtav = false;
56.          CoroutineSinkronizacija = false;
57.          upravljanjeScenom.NaMenu();
58.      }
```

U odnosu na ostale protivnike, pri porazu skeletona igra je završena, stoga sam koristio metodu *NakonPorazaNaMenu()* kako bih prešao na sljedeću scenu s odgodom vremena. Skriptu napadanja neću prikazati jer je otprilike ista kao i kod Sparka i minotaura, jedina je razlika da sam njihove metode ubacio u same animacije što ću prikazati u sljedećoj slici.



Slika 33: Postavljanje animation event kod animacije „NapadaSkeleton“ za skeletona

Pomoću „Animation Even“ sam dodao metode za nanošenje štete u samu animaciju napadanja pri čemu, će se Gizmos povećati i smanjiti po potrebi. Bilo je također potrebno dodati dijete objekt udarac, kako bi se pozicionirao taj Gizmos. Budući da će skeleton napasti 2 puta nakon prvog napada, radius za Gizmos se smanji na nula te prilikom drugog napada taj isti radijus se prilagodi tom napadu. Kada je skeleton izvan animacije napada, Gizmos i dalje ima neki svoj radijus, no budući da je izvan animacije napada, neće se pokrenuti metode za nanošenje štete, stoga nema nikakvog utjecaja na Sparka. U nastavku ću prikazati sliku animacija skeletona. Animacije su mu poprilično jednostavne, ulazi u animaciju čekanja i na toj animaciji ostaje sve dok ne upadne u kadar kamere pri čemu će početi pratiti Sparka i napadati ga.



Slika 34: Prikaz prozora „Animator“ za skeletona

Animacija smrti se može desiti u bilo kojem trenutku iz čega je i postavljen u „Any State“. Nakon smrti, skeleton ostaje u toj animaciji te će igra prijeći na scenu s izbornikom.

4.4. Stvari

U ovom poglavlju ću prikazati skripte i sprite-ove ostalih stvari na koje Spark može naići, a da ide u njegovu korist. U igri će se nalaziti trampolini, teleport i povrće. Sve nabrojane stvari ću opisati u nastavku.

4.4.1. Trampolin

Trampolin je jedan od elemenata koja čini 2D platformer mnogo zabavnijim i interesantnijim. Gotovo u svakoj uspješnoj 2D platformeru možemo vidjeti nekakve oblike trampolina, osobna inspiracija su mi igre poput: Sonic, Mario Forever, Jazzjack Rabbit i dalje. U nastavku ću prikazati sprite istog.

Slika 35: Izgled trampolina

Kod trampolina bilo je potrebno osim „BoxCollider2D“ postaviti i linijski „Collider2D“ na gornjoj površini trampolina gdje bismo željeli da se Spark odbija. U nastavku ću prikazati skriptu.

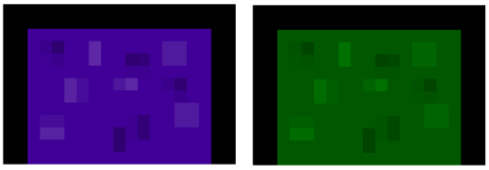
```
1.  public class Trampolin : MonoBehaviour
2.  {
3.      [SerializeField]
4.      private float VisinaSkakanja;
5.      private void OnTriggerEnter2D(Collider2D collision)
6.      {
7.          if (collision.CompareTag("Player"))
8.          {
9.              PlayerSkace(collision);
10.         }
11.     }
12.     private void PlayerSkace(Collider2D collision)
13.     {
14.         float x =
    collision.gameObject.GetComponent<Rigidbody2D>().velocity.x;
15.         collision.gameObject.GetComponent<Rigidbody2D>().velocity = new
    Vector2(x,VisinaSkakanja);
16.     }
17. }
```

Bitno je za napomenuti da sam linijskog collidera postavio kao „Trigger“ kako bi smo mogli koristiti *OnTriggerEnter2D()* metodu. U pravilu je potrebno pri koliziji utvrditi da li se radi o Sparku te će se dohvatiti njegova komponenta „Rigidbody2D“ preko koje ćemo postaviti neku silu na os **y** koristeći „velocity“.

4.4.2. Teleport

Teleport u igri potiče igrače na istraživanje razine, otkrivanje novih stvari, ali isto tako se mogu namjerno postaviti mnogo teleportova gdje samo jedan može biti izlaz. Postoje igrači koji žele proći razinu igre čim brže, ali isto tako postoje oni koji žele istraživati. Za

uporne istražitelje bih na nekim opasnim mjestima postavio teleport koji vode na neotkrivena mjesta koja mogu voditi na dodatne nagrade. U nastavku ću prikazati primjer teleporta.



Slika 36: Izgled teleporta (Vlastita izrada)

U mom primjeru imamo ljubičasti i zeleni teleport čisto radi lakšeg razlikovanja iako nije bilo potrebno. U nastavku ću prikazati skripte teleporta.

```
1.  public class Teleport : MonoBehaviour
2.  {
3.      private Transform[] teleport;
4.      void Start()
5.      {
6.          teleport = GetComponentsInChildren<Transform>();
7.      }
8.      public void ChildTrigger(Transform childTransform, Collider2D Player,
    Vector3 pozicijaTeleporta)
9.      {
10.         foreach (Transform t in teleport)
11.         {
12.             if (!t.Equals(childTransform))
13.             {
14.                 Player.transform.position = t.position+pozicijaTeleporta;
15.             }
16.         }
17.     }
18. }
```

Budući da su nam potrebna 2 teleporta koja vode prolaz isključivo između sebe postavio sam ih u objekt roditelja i za roditelja sam postavio ovu skriptu. U pravilu napravio sam vlastitu metodu koja bi funkcionirala kao „Trigger“ budući da sam kod teleportove postavio collider sa triggerom. Metoda će samo utvrditi kod kojeg teleporta nije napravljena kolizija te će Sparka teleportirati na tom mjestu. U nastavku ću prikazati metodu kod objekt djece.

```
1.  public class Teleportiranje : MonoBehaviour
2.  {
3.      private Teleport parent;
4.      private void Start()
5.      {
6.          parent = GetComponentInParent<Teleport>();
7.      }
8.      private void OnTriggerEnter2D(Collider2D collision)
```

```
9.      {
10.         if (collision.CompareTag("Player"))
11.         {
12.             TeleportirajPlayera(collision);
13.         }
14.     }
15.
16.     private void TeleportirajPlayera(Collider2D collision)
17.     {
18.         if (collision.transform.position.x>transform.position.x)
19.         {
20.             parent.ChildTrigger(transform, collision, Vector3.left*2);
21.         }
22.         else
23.         {
24.             parent.ChildTrigger(transform, collision, Vector3.right * 2);
25.         }
26.
27.     }
28. }
```

U primjeru djece želimo pohraniti objekt roditelja kako bismo mogli pristupili njegovoj metodi. U metodi TeleportirajPlayera() provjeravamo sa koje strane se nalazio Spark čisto da ga možemo teleportirati sa suprotne strane, isto kao i u realnom svijetu dok prođemo kroz vrata. Ako prođemo sa lijeve strane nalazit ćemo se na desnoj strani, tu istu logiku sam prenio i na primjeru gdje prosljeđujem vrijednosti o poziciji.

4.4.3. Povrće i bodovi

Svaki 2D platformer će uvijek imati nekakve bodove za skupljati, koji mogu služiti za dodatne nagrade poput ekstra živote ili da predstavljaju konačne rezultate. Sa bodovima želimo igrača poticati na otkrivanje novih mjesta, na avanturu i sl. U mom primjeru sam postavio povrće za skupljanje bodova što možemo vidjeti i na slici.

Slika 37: Izgled bodova za sakupljanje (Vlastita izrada)

Osim povrća sam također postavio i ekstra živote za pokupiti koji je namjerno drugačiji čist radi informiranja igrača. Kako bi skupljanje bodova bilo interesantnije postavio

sam ih da se pomiču gore-dolje. U nastavku ću samo mali dio koda prikazati jer je dosta sličan kod pčela.

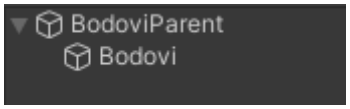
```
1.      private void Lebdenje()
2.      {
3.          if (gore)
4.          {
5.              rb2.velocity = new Vector2(rb2.velocity.x, rb2.velocity.y +
    brzina);
6.          }
7.          else
8.          {
9.              rb2.velocity = new Vector2(rb2.velocity.x, rb2.velocity.y -
    brzina);
10.         }
11.     }
```

Glavna razlika u odnosu na pčele jest, da se varijabla brzina samo oduzima ili zbraja sa os **y** „velocity“. Na taj način imamo promjenu smjera sa izglađivanjem što izgleda ljepše kod sakupljanja bodova. U nastavku ću prikazati preostale metode koje sam koristio.

```
12.     private void OnTriggerEnter2D(Collider2D collision)
13.     {
14.         if(collision.tag == "Player")
15.         {
16.             PustiZvuk();
17.             PlayerDobiBodove();
18.             ProvjeriJeliMrkvaIliZivot(collision);
19.             Destroy(gameObject);
20.         }
21.     }
22.     private void ProvjeriJeliMrkvaIliZivot(Collider2D collision)
23.     {
24.         if (gameObject.name.Contains("Mrkva"))
25.         {
26.             collision.gameObject.GetComponent<PlayerHP>().ResetHP();
27.         }
28.         else if (gameObject.name.Contains("Zivot"))
29.         {
30.             collision.gameObject.GetComponent<PlayerHP>().DobivaZivot();
31.         }
32.
33.     }
34.     private void PlayerDobiBodove()
35.     {
36.         manager.PlayerDobivaBodove(int.Parse(brBodova));
37.         GameObject bod = Instantiate(bodovi, transform.position,
    Quaternion.identity);
38.         bod.SetActive(true);
39.         bod.transform.GetChild(0).GetComponent<TextMeshPro>().text =
    brBodova;
40.     }
```

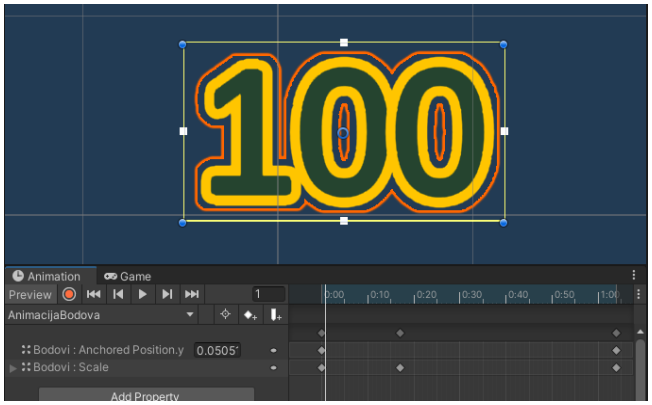
Sve stvari za sakupljanje je potrebno postaviti kao „Trigger“ dok im odredimo collider. Kada Spark dođe u kontakt sa takvim stvarima pokrenut će se *OnTriggerEnter2D()* metoda. U *OnTriggerEnter2D()* metodi će se pokrenuti zvuk, Spark će dobiti bodove te se uništi navedeni objekt. Budući da ekstra život daje život, a mrkva resetira HP potrebno je preispitati

da li se radi o navedenim stvarima te će se pokrenuti metode kod klase PlayerHP. Također je bilo potrebno animirati dobivene bodove. Kreirao sam zaseban objekt sa tekstom pod nazivom bodovi i postavio mu prazan objekt roditelja kao što je prikazano na slici.



Slika 38: Objekti za bodove

Objekt **Bodovi** sam koristio za animiranje dok sam njegov roditelj koristio za pozicioniranje. U nastavku ću prikazati sliku i animaciju bodova.



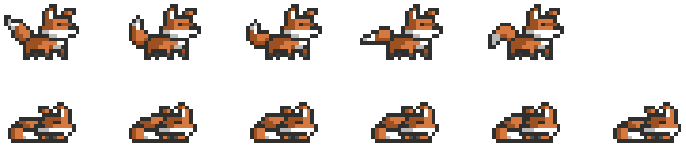
Slika 39: Animiranje teksta za bodove

U pravilu klikom na snimanje sam povukao prema gore tekst bodova i smanjio mu veličinu na nula što će dati glatku animaciju kada se unište stvari za skupljanje. Te bodove primijenio kod svih protivnika kada se poraze.

4.4.4. CheckPoints

U igri možemo primijetiti više vrsta chechpointova. Kada se igrač tek pojavi u igri onda se pojavljuje u nekakvom startu. Kako bi igrač prešao na drugu razinu potrebno je doći do cilja koji predstavlja isto nekakav checkpoint. Ako igrač izgubi život, on se može pojaviti negdje otkuda je nedavno nastavio što bi bio nekakav save checkpoint. U nastavku možemo vidjeti sliku checkpointova koje sam ja koristio.

Slika 40: Start (Vlastita izrada)



Slika 41: Izgled lisice kao checkpointa

Slika 42: Kraj razine (Vlastita izrada)

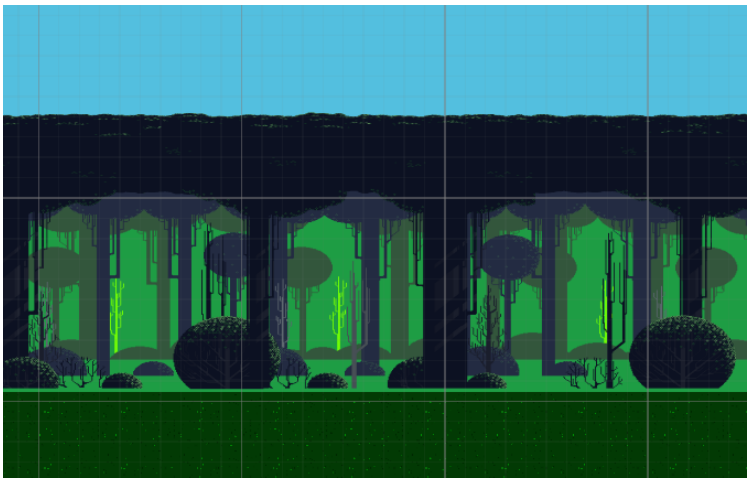
Učitavanje kod starta sam izveo na način da sam referencirao objekt **Start** u skripti „PlayerKontrole“. Poziciju Sparka sam postavio na poziciju objekta starta. Cilj sam izveo na način da sam kreirao collider koji sam postavio kao „Trigger“. Kada Spark dođe do cilja, metoda *OnTriggerEnter2D()* u skripti za cilj će učitati novu scenu. U nastavku ću prikazati skriptu koju sam koristio kod save checkpoint-ova (lisice). U nastavku ću samo mali dio skripte pokazati koji sam koristio kod lisice.

```
1.       void Update()
2.      {
3.          GledajUPlayera();
4.          PozicijaPlayera = player.transform.position.x;
5.
6.      }
```

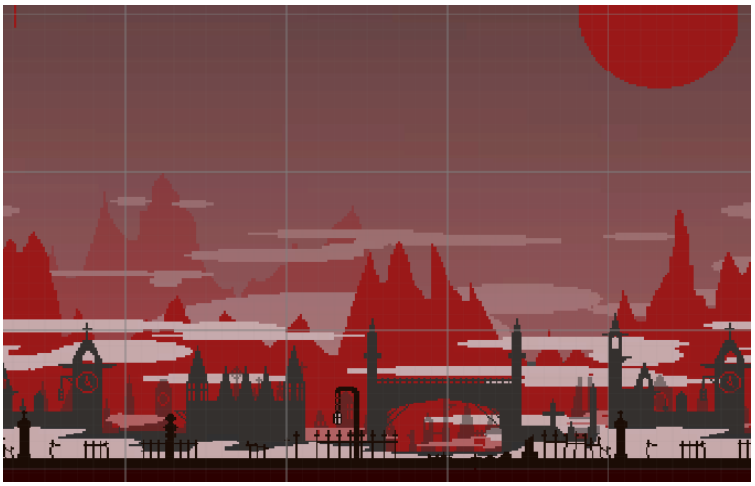
Detalje metode neću prikazati, uglavnom nakon što će Spark proći kroz checkpoint, lisica će iz animacije spavanja preći u animaciju stajanja. Također pohranjujem poziciju Sparka u toj skripti kako bi lisica uvijek bila okrenuta prema Sparku. Nastavljanje od checkpointa, već sam opisao kod „Respawn“ skripte stoga ju neću ponavljati.

4.5. Pozadinska slika

Slika pozadine je nešto što će svaka igra uvijek imati. Starije verzije igre će imati vrlo jednostavne pozadine, ponekad i jednobojne dok za suvremene slike će se utrošiti sate da se dizajniraju. U svom primjeru sam koristio dvije open-source pozadinske slike, svaka na svojoj razini koje možemo vidjeti na slici.



Slika 43: Pozadinska slika 1. razine



Slika 44: Pozadinska slika 2. razine

Pozadinske slike sam slagao po slojevima kako bih mogao napraviti parallax efekt. Više o tome ću prikazati u nastavku.

4.5.1. Parallax efekt

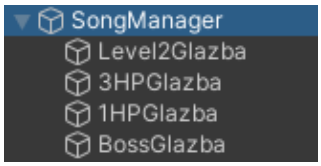
Parallax efekt je efekt slike da se njezini slojevi kreću različitim brzinama. Kako je Spark glavni fokus igre, pozadinska slika će pratiti Sparka. Slojevi za koje se smatra da su bliži Sparku će pratiti sporije u odnosu na slojeve dalje. Na taj način postižemo ugodan efekt koji krasi svaku igru. U nastavku ću prikazati skriptu [3].

```
7.  public class Background1 : MonoBehaviour
8.  {
9.      private float duzinaSlike, pozicijaSlike;
10.     public GameObject cam;
11.     public float eff;
12.     float udaljenostOdKamere;
13.     float efekt;
14.     void Start()
15.     {
16.         pozicijaSlike = transform.position.x;
17.         duzinaSlike = GetComponent<SpriteRenderer>().bounds.size.x;
18.     }
19.     void FixedUpdate()
20.     {
21.         udaljenostOdKamere = (cam.transform.position.x * (1 - eff));
22.         efekt = (cam.transform.position.x * eff);
23.         transform.position = new Vector3(pozicijaSlike + efekt,
    transform.position.y, transform.position.z);
24.
25.         if ( udaljenostOdKamere > pozicijaSlike + duzinaSlike)
26.         {
27.             pozicijaSlike += duzinaSlike;
28.         }
29.         else if(udaljenostOdKamere< pozicijaSlike - duzinaSlike)
30.         {
31.             pozicijaSlike -= duzinaSlike;
32.         }
33.     }
```

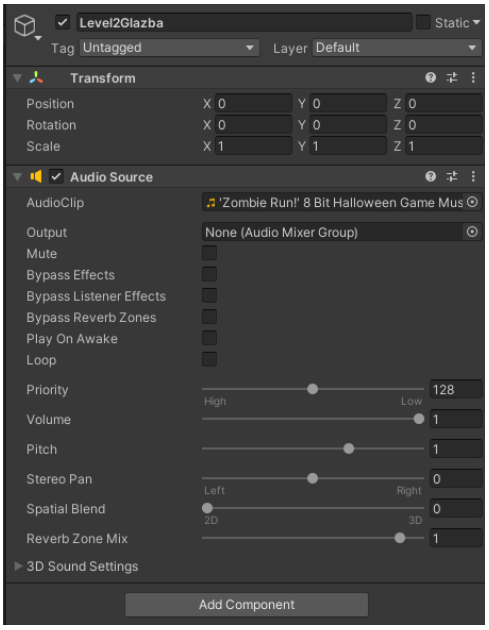
U metodi Start() smo dohvatili duzinu slike i njezinu poziciju. Koristio sam metodu *FixedUpdate()* jer daje obećavajući rezultat, kod metode *Update()* slike nisu glatko pratile kameru. Efekt smo postigli na način da smo usporili brzinu kojom će slika pratiti kameru sa varijablom efekt. Varijabla **udaljenostOdKamere** nam je služila za provjeru pozicije slike od kamere pri čemu bi je pomaknuli za njezinu dužinu.

4.6. Glazba

Kako bih puštao glazbu u pozadini koristio sam prazan objekt **SongManager** i u njemu sam unaprijed postavio djecu objekte tipa AudioSource koji sadrže glazbu u sebi kao što je prikazano na slici.



Slika 45: Objekt „SongManager“



Slika 46: Primjer objekta sa komponentom „AudioSource“

Sve glazbe koje sam koristio sam su open-source i copyright free. U svojoj igri sam htio postaviti više glazbi u istoj razini da vidim kakav će doživljaj dati. Promjenom glazbe sam htio dati dodatni znak igraču da mu je smanjen HP. Svakih put kada igrač ima 2 HP manje promijenit će se glazba dok pri resetiranju HP-a pustit će se glazba trenutne razine. Na taj način igrač ne mora previše gledati u HP već može utvrditi prema promijeni glazbe.

4.6.1. „UpravljanjePjesmom“ skripta

U nastavku ću prikazati metodu *Start()* i *Update()* za skriptu „UpravljanjePjesmom“.

```
1.      void Start()
2.      {
3.          Glazba = GetComponentsInChildren<AudioSource>();
4.          stanjeHPa = hpManager.StanjeHPa();
5.          indeksAktivnePjesme = 10;
6.          NastaviGlazbuPremaHP();
7.      }
8.      void Update()
9.      {
10.         if (!hpManager.AkoImaHP())
11.         {
12.             PrekiniGlazbu();
13.
```

```
14.          }
15.          else if (MozePromijenitiGlazbu())
16.          {
17.              stanjeHPa = hpManager.StanjeHPa();
18.              NastaviGlazbuPremaHP();
19.          }
20.      }
```

Potrebno je u metodi Start() dohvatiti djecu komponente tipa AudioSource koje ćemo puštati tijekom izvođenja igre. Varijabla **indeksAktivnePjesme** ima dvije uloge. Služit će kao sklopka za puštanje glazbe, isto tako ćemo njome znati koja je trenutno glazba puštena. Proizvoljno sam postavio vrijednost 10, jer ako bih postavio indeks koji pripada pjesmi tada ne bih mogao pustiti tu pjesmu. To sam učinio s razlogom da se ista pjesma ne pusti više puta. U *Update()* metodi će se mijenjati pjesme po potrebi, te će nam za to biti potrebna skripta od Sparka prema kojoj ćemo provjeravati HP. U nastavku ću prikazati metodu *PromjenaGlazbe().*

```
21.  private void PromjenaGlazbe()
22.      {
23.          if (hpManager.StanjeHPa() <= 1)
24.          {
25.              if (indeksAktivnePjesme != 2)
26.              {
27.                  indeksAktivnePjesme = 2;
28.                  PustiPjesmu();
29.              }
30.          }
31.          else if (hpManager.StanjeHPa() > 3)
32.          {
33.              if (indeksAktivnePjesme != 0)
34.              {
35.                  indeksAktivnePjesme = 0;
36.                  PustiPjesmu();
37.              }
38.          }
39.          else if (hpManager.StanjeHPa() < 4 || hpManager.StanjeHPa() > 1)
40.          {
41.              if (indeksAktivnePjesme != 1)
42.              {
43.                  indeksAktivnePjesme = 1;
44.                  PustiPjesmu();
45.              }
46.          }
47.      }
```

Kod promjene glazbe, provjeravamo HP igrača te će pustiti pjesmu prema istoj. Kod puštanje pjesme postavljamo indeks te pjesme te ćemo je svaku puta provjeravati kako ne bismo istu pjesmu pustili više puta. Ovo rješenje je napravljeno pomoću **if else** uvjeta, ali smatram da bi bilo mnogo bolje sa **switch case** uvjetom. U nastavku ću još prikazati metodu *NastaviGlazbuPremaHP().*

```
48.      public void NastaviGlazbuPremaHP()
49.      {
50.          if (!BossGlazba)
```

```
51.          {
52.              PromjenaGlazbe();
53.          }
54.          else if (BossGlazba && indeksAktivnePjesme != 3)
55.          {
56.              indeksAktivnePjesme = 3;
57.              PustiPjesmu();
58.          }
59.      }
```

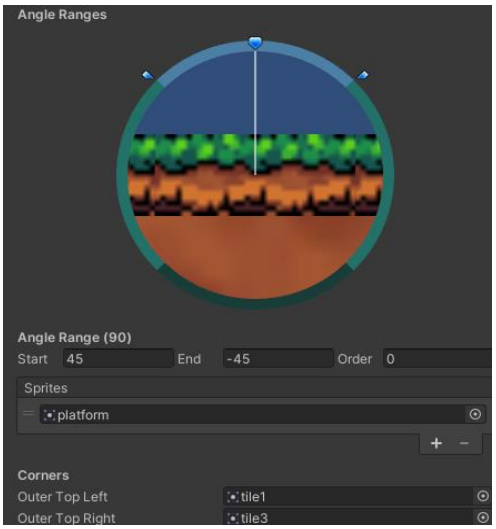
Kod ove metode sam htio prikazati kada se glazba treba promijeniti kada Spark dođe do posljednjeg protivnika. U tom slučaju ćemo zanemariti HP i provjeravati vrijednost **bool** varijable **BossGlazba**. Ta varijabla će imati vrijednost **true** dokle god se Spark bori sa skeletonom. Ta će nam varijabla pomoći pri sprječavanju da puštamo preostalu glazbu i njoj uglavnom postavljamo vrijednost iz drugih skripti.

4.7. Teren

Teren sam napravio koristeći komponentu „SpriteShape“. Ono što je bitno kod terena jest da se uklapa uz pozadinu slike, možda čak i uz protivnike. Kod kreiranja mape ne postoje granice kreativnosti. U nastavku ću prikazati slike terena.

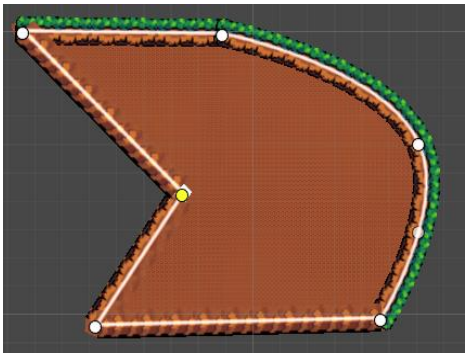
Slika 47: Izgled terena

„SpriteShape“ komponenta nam je jako korisna jer nam omogućuje da organski oblikujemo sprite-ove terena. Pri oblikovanju komponente „Spriteshape“ bilo je potrebno kreirati prazni objekt koji sadrži u sebi komponentu „Spriteshape“. Kod komponente „Spriteshape“ možemo ubaciti nekoliko sprite-ova koji će se mijenjati ovisno prema kojoj strani ga okrenemo. Na slici možemo vidjeti primjer izrade.



Slika 48: Komponenta „SpriteShape“

Izradom objekta sa komponentom „SpriteShape“ možemo ih pohraniti u nekom direktoriju te ih kopirati u scenu prema potrebi. Velika prednost te komponente je to da ih možemo organski preobličivati bez da mijenjamo rezoluciju iste. To radi na način da prilikom rastezanja terena, „SpriteShape“ komponenta će paralelno napraviti kopije sprite-ova na rastegnutim mjestima što možemo vidjeti na slici.



Slika 49: Oblikovanje terena pomoću komponente „SpriteShape“

U mom primjeru, ja nisam radio pretjerane oblike, no dobro je znati da je takva funkcionalnost moguća unutar Unity-a. U većini slučajeva, ja sam koristio samo pravilne kvadratne oblike i pravokutne što mi je bilo i više nego dovoljno kako bih svojim mapa dao sjaj. U nastavku ću prikazati skriptu terena [20].

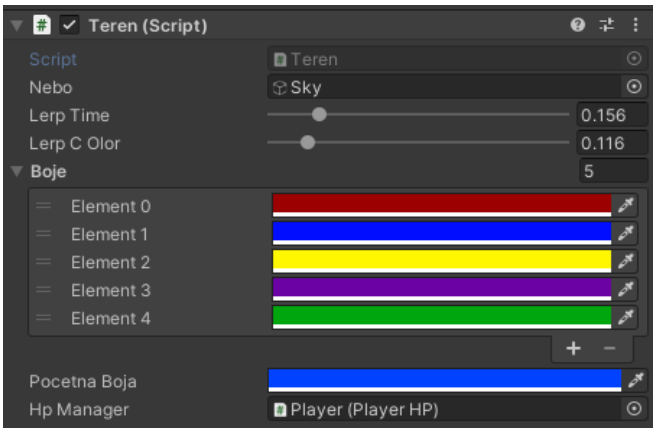
```
1.  void Update()
2.      {
3.          if (hpManager.StanjeHPa() < 1 || UpravljanjePjesmom.BossGlazba)
4.          {
5.              stop = false;
6.              foreach (SpriteRenderer sr in spriteNebo)
```

```
            {
                sr.color = MijenjajBoje(sr.color);
            }
            foreach (SpriteShapeRenderer sr in spriteRenderer)
            {
                sr.color = MijenjajBoje(sr.color);
            }
        }
        else if (!stop)
        {
            stop = true;
            foreach (SpriteRenderer sr in spriteNebo)
            {
                sr.color = Color.white;
            }
            foreach (SpriteShapeRenderer sr in spriteRenderer)
            {
                sr.color = pocetnaBoja;
            }
        }
    }
```

Skripta u pravilu je nepotrebna kod terena, ali ono što sam dodatno htio učiniti kada Spark nema više HP-a jest neprestano mijenjati boju terena, što će dati party izgled. Ovaj efekt mi se jako sviđa i jako mi je drago što sam ga stavio, te sam ga također primijenio tijekom borbe kod skeletona. Kod skripte sam uglavnom pratio stanje HP-a od Sparka te u trenutku kada nema HP, cijeli teren uključujući pozadinsku sliku nebo, počinje neprestano mijenjati boju. U nastavku ću prikazati metodu *MijenjajBoje()* [20].

```
    private Color MijenjajBoje(Color color)
    {
        t = Mathf.Lerp(t, 1f, lerpTime * Time.deltaTime);
        if (t > .9f)
        {
            t = 0f;
            indeksBoje++;
            indeksBoje = (indeksBoje >= boje.Length) ? 0 : indeksBoje;
        }
        return Color.Lerp(color, boje[indeksBoje], t * lerpCOlor);
    }
```

Koristeći metodu *Lerp()* u kombinaciji sa vremenom dobivamo lijepu tranziciju boje koju možemo prilagođavati varijablama **LerpTime** i **LerpColor**. **LerpTime** predstavlja vrijeme između dviju tranzicija, a **LerpColor** vrijeme same tranzicije boje u drugu. Na slici možemo vidjeti podešavanje iste skripte. U uvjetu provjeravamo da li je varijabla t poprimila vrijednost 1 pri čemu će se promijeniti indeks polja što znači da se prelazi na sljedeću boju.



Slika 50: Prikaz skripte „Teren“ kao komponenta

U skripti sam postavio polje tipa Color **Boje** proizvoljne veličine, iz kojih će se mijenjati tranzicije. Varijabla **PocetnaBoja** u pravilu nije potrebna, no ja sam ju koristio kako bih promijenio boju terena kada Spark dođe na drugu razinu.

4.8. Izbornik

Glavni izbornik je prva stvar koju vidimo kada upalimo igru. Kako bi u Unity napravili glavni izbornik potrebno je postaviti objekt tipa Canvas u kojem ćemo postaviti gumbe, te ovisno koliko imamo opcija i postavka potrebno ih je organizirati prema istim. U primjeru možemo vidjeti moj izbornik.

Slika 51: Glavni izbornik

Slika 52: Primjer objekta za izbornik

Kada kreiramo gumb potrebno im je dodatno podesiti veličinu, izgled i promjenu boje pri dodiru/kliku. Kako će izgledati glavni izbornik to će ovisiti od osobe do osobe, ideju koju sam ja primijenio na primjeru jest, da ga prikažem u obliku prve razine. Također sam dodao Sparka kao i ostale protivnike kako bi upotpunili doživljaj. Također sam kod naslova animirao slova na isti način kao i kod bodova i pčela koristeći skriptu za kretanje dolje/gore. U nastavku ću prikazati skriptu za izbornik.

```
1.      public void PlayGame()
2.      {
3.          Manager.bodovi = 0;
4.          StartCoroutine(UcitajLevel(SceneManager.GetActiveScene().buildIndex
    + 1));
5.      }
6.      public void QuitGame()
7.      {
8.          StartCoroutine(IzadziIzIgre());
9.      }
```

Metode koje su prikazane će se koristiti za gumbe koje smo si postavili na početku izbornika. Za gumb **Options** nije bilo potrebno raditi metodu jer Unity gumbovi imaju vlastitu funkcionalnost koji možemo postaviti u kojem možemo postaviti vidljivost nekog objekta. U mom primjeru maknemo vidljivost za objekta s izbornikom i stavimo vidljivost na objekt **Options**. U nastavku možemo vidjeti primjer slike za postavke.

Slika 53: Postavke

Postavke će kod svake igre varirati. U mom primjeru je poprilično jednostavan gdje možemo postaviti zvuk igre. Htio sam postaviti zasebno zvuk za igru i glazbu igre, ali iz nekog razloga mi nije radilo stoga sam odustao da ne izgubim na vremenu. U nastavku ću prikazati primjer metoda koje sam koristio za postavljanje zvuka.

```
1.     private void PromijeniGlasnocu()
2.      {
3.          glasnocaGame = GameSlider.value;
4.          PlayerPrefs.SetFloat("GameVolume", glasnocaGame);
5.          Ucitaj();
```

```
    }

    public void Ucitaj()
    {
        glasnocaGame = PlayerPrefs.GetFloat("GameVolume");
        GameSlider.value = glasnocaGame;
        AudioListener.volume = glasnocaGame;
    }
```

Unity ima jednu vrlo zanimljivu klasu PlayerPrefs koja nam omogućuje pohranjivanje vrijednosti čak i kada igra ne radi. To je vrlo korisna klasa te mi je žao što nisam na početku znao za nju nego tek na kraju. U mom primjeru koristio sam ju kako bih pohranio vrijednost objekta **slider** te kada se igra upali, **slider** i razina zvuka će se ravnati po pohranjenoj vrijednosti. Za podešavanje zvuka sam koristio AudioListener koji nam omogućuje podešavanje zvuka u cijeloj sceni. AudioListener je po defaultu postavljen na glavnoj kameri. Kada bih radio opcije za više zvukova tada bih koristio AudioSource za podešavanje zvuka, dok bi mi AudioListener služio da mute-am sve zvukove.

4.8.1. Izbornik tijekom igre

Izbornik tijekom igre je preporučljiv za svaku igru. U starijim verzijama i u amaterskim igrama možemo primijetiti da ne postoji uvijek izbornik tijekom pauze već jednostavno se izlazi iz te scene u scenu gdje se nalazi glavni izbornik. U mom primjeru možemo vidjeti izbornik tijekom igre.

Slika 54: Izbornik tijekom igre

Pri izradi izbornika tijekom igre, koriste se gotovo iste tehnike jedina je razlika u načinu kada ćemo prikazati njihovu vidljivost. U svom primjeru, a i u većini slučajeva, kada se aktivira izbornik tijekom igre, smrzava se vrijeme igre i potamni pozadina. U nastavku ću prikazati metode skripte.

```
    void Update()
    {
```

```
3.          if (Input.GetKeyDown(KeyCode.Escape) && !GameOver.Gameover)
4.          {
5.              if (pauzirano)
6.              {
7.                  Pokreni();
8.              }
9.              else
10.             {
11.                 Pauziraj();
12.             }
13.         }
14.     }
15.     void Pauziraj()
16.     {
17.         Time.timeScale = 0f;
18.         PauseUI.SetActive(true);
19.         pauzirano = true;
20.     }
21.     public void Pokreni()
22.     {
23.         Time.timeScale = 1f;
24.         PauseUI.SetActive(false);
25.         YesNo.SetActive(false);
26.         GameOptions.SetActive(false);
27.         pauzirano =false;
28.     }
```

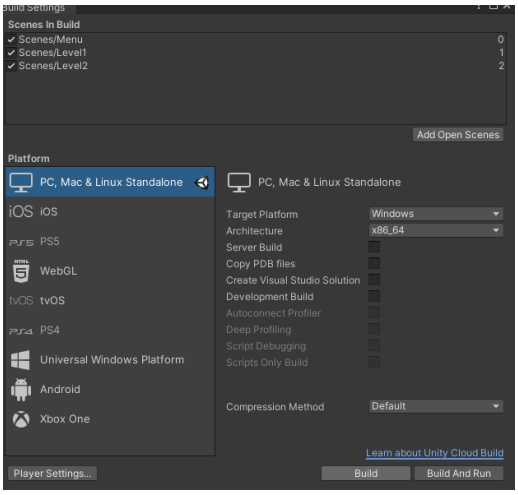
Izbornik tijekom igre otvaramo sa tipkom „Escape“ stoga je potrebno cijelo vrijeme pratiti da li će igrač pritisnuti tu tipku. Potrebno je koristiti neki parametar koji će služiti kao prekidač da bi se samo jednom izvela metoda, u mom primjeru **pauzirano**. Pomoću Time**.timeScale** smrzavamo igru i postavljamo objekte vidljivim, a sve ostalo se svodi na isti način kao i kod glavnog izbornika. U sljedećoj slici vidimo kako bi izgledala završena igra.

Slika 55: Kraj igre

Ovo sam slikao tijekom uređivanja scene stoga, možemo zanemariti živote i HP. Ovaj ekran radi na isti način kao i glavni izbornik razlika je u tome što sam u skripti koristio javnu **static** varijablu **gameOver**. Kada je ta vrijednost **true**, prikazat će se zaslon gotove igre, a pri učitavanju izbornika, **gameOver** će biti **false**.

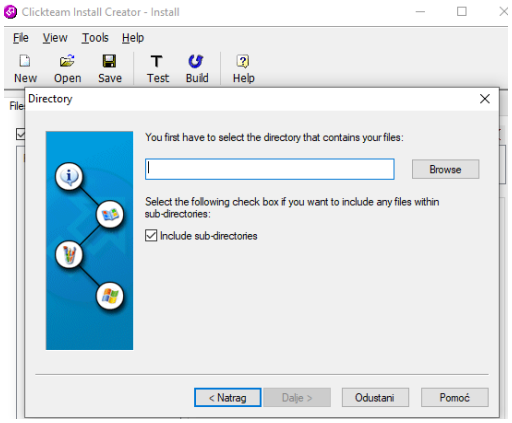
4.9. Kreiranje instalacije

Nakon izrade igre bilo je potrebno kreirati izvršnu datoteku. Postupak je vrlo jednostavan, potrebno je otići na „Build settings“, zatim na „Player settings“. Tamo će biti moguće postaviti zadanu ikonu igre i još mnoge druge opcije. Kada smo zadovoljni sa podešavanjima, kliknemo „Build„ zatim će Unity kreirati mapu sa izvršnom datotekom.



Slika 56: Prikaz „Build settings“ prozora

Kod izrade instalacije igre koristio sam program Install Creator. Program je vrlo jednostavan za korištenje stoga neću prikazati detaljnu instalaciju. Sve što je potrebno jest pratiti čarobnjak za instaliranje, popunjavati prazna područja te će se kreirati instalacijska datoteka.



Slika 57: Install Creator

5. Zaključak

Tijekom izrade igre suočio sam se sa brojnim izazovima, sa padovima i uspjesima, frustracijama i ponosom kada sam dovršavao igru. U početku sam htio izraditi vlastiti dizajn likova, shvativši koliko to vremena oduzima odlučio sam se na otvorene sprite-ove.

Nakon instalacije Unity, jako mi se svidjelo to što je Visual Studio bio povezan, te nisam trebao ništa ručno povezivati. Budući da sam se izradom u Unity počeo baviti već tijekom zimskog semestra ove godine, nisam bio upoznat sa C#. Kao rezultat toga, nisam imao objektno orijentiranu organizaciju koda po klasama i skriptama, stoga kod mi je bio poprilično neuredan. Kod Unity alata mi se nije svidjelo to što sam morao ručno u postavkama omogućiti automatsko praćenje pravopisa pri korištenju Visual Studio. Zbog toga mi je trebalo neko vrijeme da pronađem podešavanje u postavkama te sam imao spor napredak prilikom proučavanja alata. Jako mi se svidio način na koji se kreiraju prazni objekti, animacije i tranzicije. S time nisam imao previše problema, ali nisam previše zadovoljan sa prozorom animator. Bilo je mi je potrebno mnogo vremena da se prilagodim tom prozoru kako bih bio zadovoljan sa postavljenim animacijama. U početku sam svu animaciju manipulirao preko skripte i radilo je iznenađujuće dobro. Unatoč tome, budući da s tim Unity gubi svoju glavnu svrhu, odlučio sam postaviti uvjete animacija u animatoru. Ono što bi bilo super za Unity jest da u animatoru svaka animacija osim uvjeta i parametara ima i svoj prioritet. Isto kako i procesi na računalu imaju svoj prioritet izvršenja tako je i za mene logično da to sadrži i animator. Smatram, da bi s time animator bio mnogo jednostavniji, nudio bi širi pristup razvijanju animacija. Jako mi se sviđaju komponente za postavljanje kolizije na objektima, međutim pojavljivao se problem u kojem nisam mogao animirati collider ako bi ih imao više. Isto tako, pri postavljanju poligon collider, bio sam zbunjen kada sam mu moga mijenjati oblik, ali se nije pohranio tijekom izvađanja igre. U tom slučaju, ne bih uopće volio imati mogućnost uređivanja tog collidera ako mi nije omogućena. Također bih volio, da mi je dodavanje scena u igru omogućeno u prozoru umjesto pod postavkama, u suprotnom bi mi bilo bolje da su scene dodane po defaultu.

Unatoč zamjerkama, na temelju vlastitog iskustva, od 1 do 5, alatu bih dao ocjenu 3,5. Glavna zamjerka su bugovi i malo veća očekivanja pri brzini rada i učitavanja, s druge strane, dostupnost brojnim značajkama je jedna od jakih strana. Budući da je široko otvoren ostalim platformama, jednostavan za korištenje, Unity je odličan za sve one koji se planiraju baviti izradom računalnih igri.

Popis literature

[1] Unity.hr (13. listopada, 2020) : *Povijest Unityja*

Preuzeto 24.8.2022 s: https://unity.hr/povijest-unityja/

[2] Unity.hr (13. listopada, 2020): *Što je Unity?*

Preuzeto 24.8.2022 s: https://unity.hr/sto-je-unity/

[3] Dani (25. veljača 2019): *Unity Parallax Tutorial - How to infinite scrolling background*

Preuzeto 5.3.2022 s: https://youtu.be/zit45k6CUMk

[4] Unity Forum

Preuzeto 15.3.2022 s: https://forum.unity.com/

[5] Unity Answers

Preuzeto 2.7.2022 s: https://answers.unity.com/index.html

[6] The Spriters Resources

Preuzeto 2.10.2021 s: https://www.spriters-resource.com/

[7] Stack Overflow

Preuzeto 26.6.2022 s: https://stackoverflow.com/questions

[8] Microsoft Documentation

Preuzeto 15.7.2022 s: https://docs.microsoft.com/

[9] Codex Gamicus: *2D platform video games*

Preuzeto 24.8.2022 s: https://gamicus.fandom.com/wiki/2D_platform_video_games

[10] iD Tech (22. listopada 2012): *10 Types of Platforms in Platform Video Games*

Preuzeto 26.8.2022 s: https://www.idtech.com/blog/10-types-of-platforms-in-platform-video-games

[11] Brackeys (14. listopada 2018) : *MAKE ORGANIC LEVELS!! Unity Sprite Shape*

Preuzeto 17.6.2022 s: https://youtu.be/GSo_fU1JdfM?list=PLPV2KyIb3jR5QFsefuO2RlAgWEz6EvVi6

[12] Brackeys (15. srpnja 2018): *2D Movement in Unity (Tutorial)*

Preuzeto 2.6.2022 s: https://youtu.be/dwcT-Dch0bA?list=PLPV2KyIb3jR5QFsefuO2RlAgWEz6EvVi6

[13] Brackeys (8. srp 2018.): *How to make a 2D Game in Unity*

Preuzeto 18.6.2022 s: https://youtu.be/on9nwbZngyw?list=PLPV2KyIb3jR5QFsefuO2RlAgWEz6EvVi6

[14] Brackeys (15. prosinca 2019): *MELEE COMBAT in Unity*

Preuzeto 26.7.2022 s: https://youtu.be/sPiVz1k-fEs?list=PLPV2KyIb3jR5QFsefuO2RlAgWEz6EvVi6

[15] Brackeys (26. siječnja 2020): *How to make a BOSS in Unity!*

Preuzeto 18.7.2022 s: https://youtu.be/AD4JIXQDw0s

[16] Lost Relic Games (6. prosinca 2019): *Enemy AGRO AI System in Unity For Beginners (2D Game Dev Tutorial)*

Preuzeto 15.7.2022 s: https://youtu.be/nEYA3hzZHJ0

[17] Lost Relic Games (18. ožujka 2019): *GameDev Tutorial: Howto make enemies Flash and Explode in Unity*

Preuzeto 10.3.2022 s: https://youtu.be/WgLd6EahyVU

[18] Unity Documentation

Preuzeto s https://docs.unity3d.com/Manual/

[19] Hooson (5. travnja 2021): *How To Make A Volume Slider In 4 Minutes - Easy Unity Tutorial*

Preuzeto 11.6.2022 s: https://youtu.be/yWCHaTwVblk

[20] Hamza Herbou (20. listopada 2019) : *Color Lerp in Unity, Best practice*

Preuzeto 11.7.2022 s: https://youtu.be/C_f2ChrcSSM

[21] YouGov (prosinac 2021): *Super Smash Bros. is the most popular fighting game*

Preuzeto 20.8.2022 s: https://today.yougov.com/topics/technology/articles-reports/2022/01/04/us-super-smash-bros-most-popular-fighting-game

Popis slika

Popis korištenih resursa

[1] OpenSurge Team, Stephen Challener (Spark, sprite)

Dostupno 25.2. 2022. na: https://opengameart.org/content/surge-of-opensurge-for-ultimate-smash-friends

[2] Stephen Challener (Minotaur, sprite)

Dostupno 15.3. 2022. na: https://opengameart.org/content/minotaur-with-axe

[3] AstroBob (Skeleton, sprite)

Dostupno 25.7. 2022. na: https://astrobob.itch.io/animated-pixel-art-skeleton

[4] KingKelpo (Pčela, sprite)

Dostupno 5.7. 2022. na: https://kingkelp.itch.io/bee

[5] Elthen's Pixel Art Shop (Lisica, sprite)

Dostupno 7.3. 2022. na: https://elthen.itch.io/2d-pixel-art-fox-sprites

[6] manyufan233 (Zmija, sprite)

Dostupno 12.7. 2022. na: https://manyufan233.itch.io/animated-pixel-snake-monster

[7] KoopsFan (Trampolin, slika)

Dostupno 2.8. 2022. na: https://www.pixilart.com/art/smm-smb1-smas-trampoline-76dabe97ad2ae46

[8] SpinachChicken (Pauk, sprite)

Dostupno 15.7. 2022. na: https://opengameart.org/content/2d-spider-animated

[9] MortMort (Grass and Dirt, tileset)

Dostupno 21.2. 2022. na: https://mnrart.gumroad.com/l/OWUQQ

[10] stext25 (DarkFantasyCity, assets)

Dostupno 2.7. 2022. na: https://stext25.itch.io/dark-fantasy-city

[11] edermuniz14 (Pixel Art Forest, assets)

Dostupno 25.6. 2022. na: https://www.gamedevmarket.net/asset/free-pixel-art-forest/

[12] qubodup (Sword Hit, zvuk)

Dostupno 5.8. 2022. na: https://pixabay.com/sound-effects/sword-hit-7160/

[13] Nightflame (Swinging staff whoosh, zvuk)

Dostupno 5.8. 2022. na: https://pixabay.com/sound-effects/swinging-staff-whoosh-strong-08-44658/

[14] Snowflakes (whip01, zvuk)

Dostupno 5.8. 2022. na: https://pixabay.com/sound-effects/whip01-6952/

[15] Mrdropex (monster attack sound, zvuk)

Dostupno 5.8. 2022. na: https://pixabay.com/sound-effects/monster-attack-sound-46185/

[16] HeatleyBros, "8 Bit Go!" (glazba izbornika)

Dostupno 2.7. 2022. na: https://youtu.be/-ykzvHU6si8

[17] NCS, Dirty Palm, Oblivion (glazba nakon 3 HP)

Dostupno 2.7. 2022. na: https://youtu.be/8Yue9YYdNLM

[18] HeatleyBros, "8 Bit Adventure!" (glazba 1 razine)

Dostupno 2.7. 2022. na: https://youtu.be/Wsw-86zjb8I

[19] HeatleyBros, "Zombie Run!" (glazba 2 razine)

Dostupno 2.7. 2022. na: https://youtu.be/u9LL6sqrunU

[20] HeatleyBros, "Halloween Dash!" (glazba nakon 1 HP)

Dostupno 2.7. 2022. na: https://youtu.be/hvZxWep3fH4

[21] Neffext, Graveyard (boss glazba)

Dostupno 2.7.2022. na: https://youtu.be/W7NWnqB8dQE

[22] Clickteam, Install Creator

Dostupno 16.8.2022. na: https://install-creator.en.softonic.com/