Izrada 2D igre s generiranim razinama

Ćurko, Domagoj

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Domagoj Ćurko

Izrada 2D igre s generiranim razinama

ZAVRŠNI RAD

Varaždin, 2019.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Domagoj Ćurko

Matični broj: 44905/16–R

Studij: Informacijski sustavi

Izrada 2D igre s generiranim razinama

ZAVRŠNI RAD

Mentor:

Prof. dr. sc. Danijel Radošević

Varaždin, kolovoz 2019.

Domagoj Ćurko

Izjava o izvornosti

Izjavljujem da je moj završni 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

U ovom završnom radu ću predstaviti žanr igara temeljen na proceduralnom generiranju razina, poznat kao "Roguelike", algoritam za kreiranje razina te instrumentarij programskog alata Unity za izradu takvih igara. U praktičnom dijelu rada izradit će se 2D video igra koja koristi algoritam za proceduralno generiranje razina. Glavni lik, odnosno igrač, koristi luk i strijelu kako bi uništio neprijatelje te probio put do blaga u nasumično generiranim razinama. Veliku pozornost obratiti ću na samo generiranje razina, te ću objasniti algoritam koji sam koristio i zašto je baš on bio pogodan za igru koju sam izradio. Sam programski kod pisan je u programskom jeziku C# u alatu Microsoft Visual Studio. Cilj rada je pružiti uvid u proces izrade video igre, od njenog začeća gdje ona postoji samo kao ideja, pa do samog kraja realizacije.

Ključne riječi: unity; 2D igra; nasumična generacija; roguelike; algoritmi;

Sadržaj

1. Uvod

Tema ovog završnog rada je Izrada 2D igre s generiranim razinama. Ukratko ću proučiti žanr igara pod nazivom „roguelike“, te ideju igre čija će izrada biti praktični dio rada. Sama igra će koristiti algoritam za proceduralno generiranje razina, što je i jedna od glavnih obilježja roguelike žanra. Igra će biti izrađena u alatu Unity, koji olakšava izradu video igara instrumentarijem koji ću pobliže objasniti. Za pisanje programskog koda koristiti ću alat Microsoft Visual Studio. Ideja za video igru koja je praktični dio ovog završnog rada došla je od sličnih igara, kao što je na primjer Spelunky [2]. Proći ću kroz sve bitne korake izrade ove video igre, s velikim fokusom na samo generiranje razina.

Obradom ove teme želim ukazati na izazove izrade video igara, te na koji način se ti izazovi mogu prebroditi. Industrija video igara postaje sve utjecajnija grana informatike. Tržište za video igre je ogromno, te se nadam da ovim radom mogu ukazati na činjenicu da izrada video igara nije nešto nedostižno. Na obradu ove teme motivirala me činjenica da mi je već nekoliko ljudi reklo kako su jako zainteresirani za izradu video igara, ali kako ne znaju gdje početi. Mnogo ljudi ima predodžbu da moraju biti izvrsni programeri kako bi se upustili u takav poduhvat. Iako ne mogu reći da za izradu video igre nije potrebno minimalno razumijevanje programiranja, mogu sa sigurnošću reći da je potrebna razina jako mala i da se može usvojiti u nekoliko dana. Naime, ja sam se počeo baviti izradom video igara kao hobi nakon prve godine studija. Počeo sam ne vjerujući potpuno da ću išta postići, i mislio sam da ću morati učiti programirati na fakultetu još godinama prije nego što mogu izraditi nešto što nalikuje na video igru. Međutim, kada sam pogledao nekoliko vodića na YouTube-u i čuo za alat Unity, u nekoliko dana sam došao do rezultata. Vidio sam da problem kod izrade video igara uopće nije napisati neku naredbu, jer se bilo koja naredba može pronaći na internetu u nekoliko sekundi. Pravi izazov je znati prepoznati problem, a onda osmisliti rješenje. Na primjer, nije problem saznati poziciju nekog objekta u prostoru, nego prepoznati da uvećavanjem vrijednosti neke od koordinata te pozicije možemo dobiti predodžbu kretanja tog objekta. Zbog toga ću izraditi ovu igru, kako bih dao ideju o tome što je zapravo izazovno kod izrade video igara, te koliko intuitivna neka rješenja kompleksnih problema mogu biti. Upravo to je razlog zbog kojeg sam se zaljubio u izradu video igara i zbog kojeg sam odabrao baš ovu temu za svoj završni rad.

2. Unity

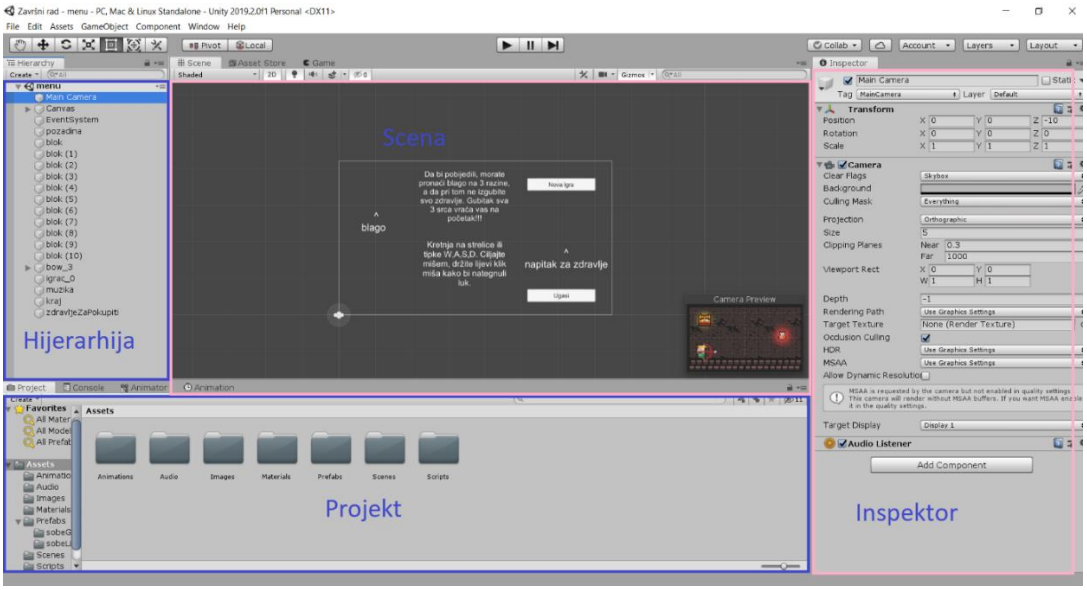
Unity je popularni alat za izradu video igara. Prvi put je najavljen i postao dostupan u lipnju 2005. godine kao alat za izradu igara samo za Mac OS X operacijski sustav, ali danas je proširen da podržava izradu igara za više od 25 različitih platformi. Služi za izradu 2D i 3D igara, kao i igara koje koriste virtualnu i proširenu stvarnost. Kada je najavljen 2005. godine od strane Unity Technologies-a, cilj Unity alata bio je demokratizacija procesa izrade video igara tako da omoguće velikom broju ljudi pristup potrebnom instrumentariju. Unity danas ima besplatnu opciju za kompanije koje godišnje zarađuju manje od 100,000 američkih dolara. Činjenica da Unity ima besplatnu opciju zasigurno je uvelike doprinijela njegovoj popularnosti, te je danas on jedan od najpopularnijih i po mojem mišljenju najboljih alata za izradu video igara na tržištu.

2.1. Unity Asset Store

Jedan od razloga zbog kojih sam se upustio u izradu video igara bilo je postojanje Unity Asset Store-a. Unity Asset Store je platforma na kojoj korisnici mogu objavljivati svoj rad, kao na primjer 2D slike ili 3D modele, animacije, audio datoteke i gotovo sve ostale vrste imovine potrebne za izradu bilo kakve igre. Za takvu imovinu se koristi pojam „assets“. Ideja je da kreatori koji koriste Unity međusobno dijele i prodaju assete, te tako jedni drugima olakšavaju i ubrzavaju izradu igara. Meni je to oduvijek puno značilo jer sam znao programirati, ali sam mislio da je izrada igara nešto meni nedostižno jer nemam talent za umjetnost. Pokazalo se da je moj strah bio uzaludan, jer za izradu igre uistinu nije potrebno znati sve. Dovoljno je imati viziju zabavne igre, a modeli, animacija i muzika se vrlo jednostavno i brzo mogu besplatno preuzeti ili kupiti putem Asset Store-a ili sličnih platformi.

2.2. Unity korisničko sučelje

Korisničko sučelje alata Unity vrlo je jednostavno. Sastoji se od nekoliko ključnih dijelova, a to su hijerarhija, inspektor, scena i projekt. Na slici 1 ispod možemo vidjeti raspored tih dijelova kakav on bude pri prvom otvaranju alata Unity nakon instalacije.



Slika 1: Korisničko sučelje alata Unity

U prozoru Projekt možemo vidjeti sadržaj mape u koju smo spremili svoj projekt. Sve što želimo koristiti u našoj igri mora se prvo nalaziti u ovom prozoru. Dakle, ako želimo dodati novo ponašanje u našu igru, moramo prvo u prozoru Projekt napraviti novu C# skriptu, a tek nakon toga ju dodati na neki objekt u sceni.

Scene možemo shvatiti kao odvojene svjetove u alatu Unity, pa se često scene koriste kao razine tako da se u svakoj sceni izgradi jedna razina igre. Prozor Scena služi kako bi dobili jasniju predodžbu o tome gdje se koji objekt nalazi i u kojem je odnosu s drugim objektima u trenutno otvorenoj sceni. Prikazuje 2D ili 3D prostor, ovisno o tome kakvu igru radimo. Međutim, i 2D igre su u alatu Unity zapravo igre u 3D prostoru koje koriste samo dvije od tri dimenzije, pa se jednim klikom uvijek može prebaciti iz 2D u 3D pogled.

Na prozor Hijerarhija možemo gledati kao na presliku prozora Scena. On zapravo sadrži popis svih objekata u trenutno otvorenoj sceni, te daje jednostavniju alternativu za odabir bilo kojeg objekta ili više objekata.

Prozor Inspektor je prozor u kojem vidimo podatke o odabranom objektu. Kad odaberemo neki objekt, u Inspektoru vidimo sve komponente tog objekta. Svaki objekt ima komponentu Transform, koja određuje njegovu poziciju, veličinu i rotaciju u trodimenzionalnom prostoru. Ostale komponente bilo kojeg objekta su skripte koje smo pridružili tom objektu. Kroz ovaj prozor dodjeljujemo nova ponašanja objektima tako da im pridružujemo potrebne skripte. Na primjer, na slici gore vidimo da je objektu Main Camera (glavna kamera) pridružena skripta Camera, koja postoji kao dio osnovnog paketa alata Unity te koja sadrži svo ponašanje koje definira kameru.

3. Roguelike žanr igara

Ovaj žanr dobio je naziv po video igri Rogue. [7] Roguelike je podžanr igara igranja uloga, za kojeg su karakteristična obilježja kretanje kroz nasumičo generirane razine, grafika temeljena na kvadratićima koje zovemo tile-ovi, te stalna smrt. Taj pojam označava pojavu u nekim igrama gdje se igrač nakon smrti u igri ne vraća na zadnje mjesto gdje je spremio igru, već na sami početak igre. To znači da su u svakom susretu s neprijateljem u video igri puno veći ulozi, jer igrač može izgubiti sate koje je proveo u igri sa nekoliko krivo pritisnutih kontrola. Nekad se smatralo da se video igra mora, kako bi bila smatrana roguelike naslovom, temeljiti na igri na krugove, gdje igrač čeka svoj red i tada odradi svoj potez, zatim opet čeka svoj red i tako dalje. Danas se pojam roguelike najčešće pridodaje igrama koje imaju barem neke od ovih značajki.

Za ovaj završni rad izraditi ću igru ovog žanra. Sadržavati će nasumično generirane razine i stalnu smrt, što su dva glavna obilježja roguelike žanra.

3.1. Nasumična generacija razina

Nasumična generacija ili proceduralna generacija pojam je koji se u dizajnu igara koristi kako bi se opisao postupak generiranja sadržaja ne ručno, već uz pomoć algoritma koji stvara sadržaj na temelju slučajnih brojeva. Postoji mnogo načina da se u nekoj igri implementira nasumična generacija. Neke video igre koriste u potpunosti nasumično generiran sadržaj pomoću algoritma, kao na primjer igra No Man's Sky[1]. Takve igre često ne izgledaju zanimljivo, jer ako je sav sadržaj nasumično generiran, možemo pukom šansom dobiti cijelu razinu koja je sastavljena od ravnog puta do cilja na kojem ne stoji niti jedan neprijatelj ili bilo koji drugi oblik izazova. Iz tog razloga u video igrama se najčešće koristi generiranje razina koje je hibrid nasumičnog generiranja i umjetničkog dizajna razina.

U video igri je vrlo mala šansa da ćemo, koristeći samo nasumičnu generaciju, naići na nešto što nalikuje dvorcu. Međutim, ako kreator igre napravi nekoliko dijelova dvorca, na primjer soba, a zatim igra nasumično poslaže te dijelove koje je kreator ručno izradio, može postići efekt nepredvidljivosti koji prati nasumiču generaciju, ali i dalje donekle oblikovati iskustvo koje će igrač imati u tom dvorcu. Primjer toga možemo vidjeti na slici 2, koja prikazuje jedan od nasumično generiranih dvoraca u igri Cube World.

Zbog toga ću i ja u svojoj igri koristiti sličan tip nasumične generacije. To je nužno jer se moja ideja zasniva na prelasku razina, a nasumična generacija bi to mogla onemogućiti. Više o tome ću reći u poglavlju Izrada igre.

Slika 2: Dvorac u igri Cube World [4]

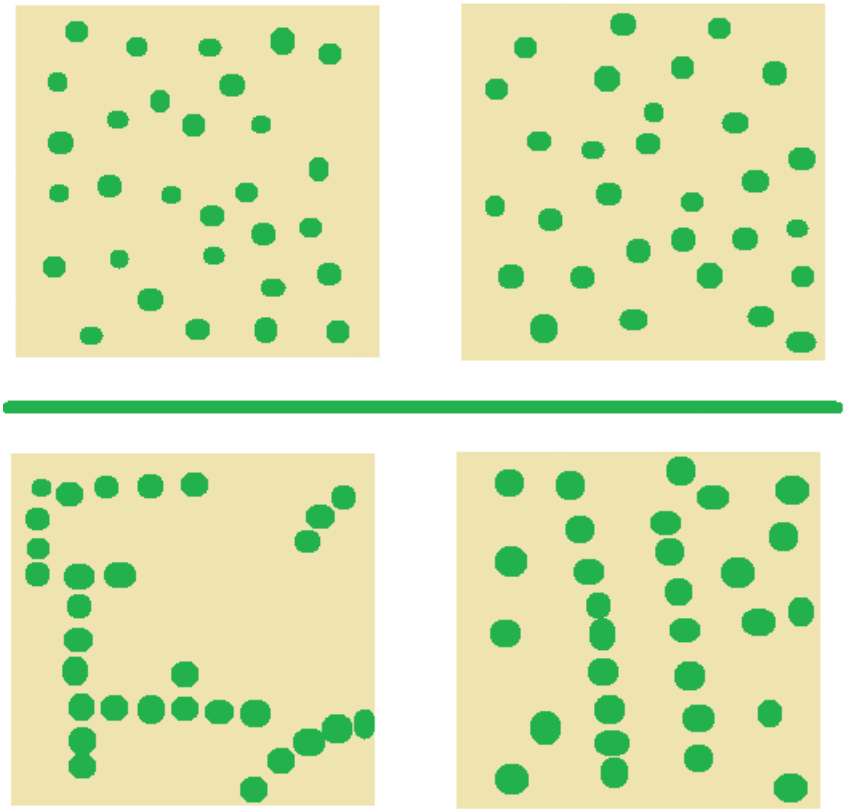
3.2. No Man's Sky

Primjer igre koja koristi nasumičnu generaciju je No Man's Sky. Fokus je u ovoj video igri stavljen na istraživanje i skupljanje resursa u galaksiji koja je ispunjena s nezamislivo velikim brojem planeta, od kojih je igraču na svaki omogućeno slijetanje i istraživanje. Gotovo svi elementi u igri su nasumično generirani, uključujući solarne sustave, planete i njihove ekosisteme, floru, faunu i njihova ponašanja. [1]

Slika 3: Igra No Man's Sky [5]

Iako nasumična generacija u igri No Man's Sky stvara zanimljive planete svojim algoritmom, za neke igrače oni uskoro postanu dosadni. Iako se dva planeta razlikuju, i dalje oba imaju travu, stabla i kamenje koji su nasumično „razbacani“ po planetima. Slika 4

demonstrira zašto smatram da potpuno nasumična generacija sadržaja nije uvijek dobro rješenje za neke igre. Zamislimo da su sve zelene točke na slici 4 stabla na nekoj razini. U gornjem dijelu slike vidimo primjer dviju razina koje su nastale potpuno nasumičnom generacijom, a u donjem dijelu slike 4 vidimo primjer dviju razina koje je umjetnik ručno izradio.



Slika 4: Usporedba nasumičnih i rukom izrađenih razina

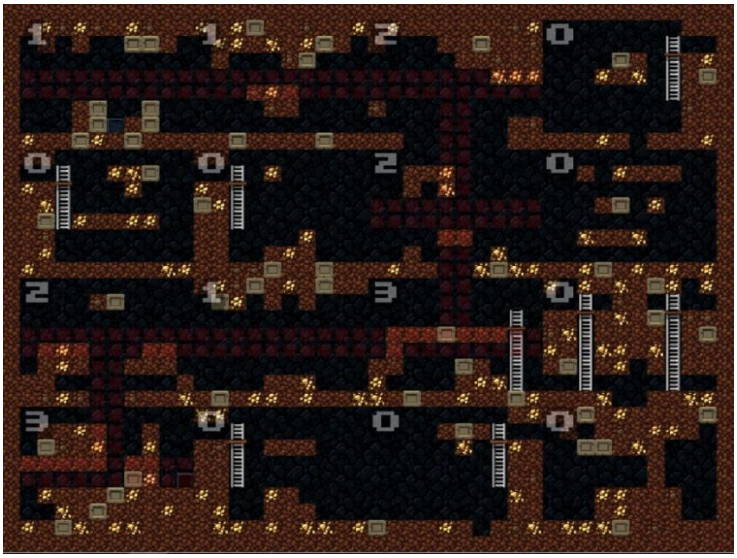
Dvije razine koje je ručno izradio umjetnik ostavljaju jači dojam, te su međusobno puno više različite i upamtljive nego one koje su generirane potpuno nasumično. Razlog tomu je što umjetnik koji gradi razinu postavlja stabla tako da dobije što zanimljivije uzorke i motive, na primjer put kroz šumu, dok će nasumična generacija gotovo uvijek dati dosadniji raspored stabala. Pogledajmo jednu video igru koja koristi kombinaciju umjetničkog i nasumičnog generiranja razina.

3.3. Spelunky

Spelunky je jedinstveni platformer sa nasumično generiranim razinama koje nude novo i izazovno iskustvo pri svakoj novoj igri. [2]

Slika 5: Spelunky [2]

Pogledajmo kako Spelunky pristupa nasumičnoj generaciji razina, uz pomoć slike 6. Svaka razina temelji se na dvodimenzionalnom polju soba, ili „mreži“, dimenzija 4 vertikalno i 4 horizontalno. Igrač uvijek počinje u jednoj od gornjih soba. Kako bi bili sigurni da će igrač moći doći do dna razine, odnosno do kraja, prvo se stvara takozvani „kritični put“. Njega čine sobe koje je odredio algoritam. Na slici 6 kritični put naznačen je crvenkastim kvadratićima. [3]



Slika 6: Generacija razina u igri Spelunky [3]

Svaka soba koja je na kritičnom putu mora biti određenog tipa, odnosno mora imati izlaze na točno određene strane kako bi bili sigurni da će se igrač moći slobodno kretati kritičnim putem. Ostale sobe mogu biti bilo kojeg tipa, jer se u njih može ali ne mora biti moguće ući. Postojanje kritičnog puta osigurava da će svaku razinu biti moguće prijeći. [3]

Odlučio sam da će se u mojoj igri koristiti algoritam za generiranje razina sličan kao u igri Spelunky, osim što će u mojoj igri dimenzije razina biti promjenjive i igrač će putovati od dolje prema gore kroz razinu.

4. Izrada igre

Igra koju ću izraditi kao praktični dio ovog završnog rada biti će 2D igra u kojoj je cilj na svakoj razini pronaći blago, koje označava kraj razine. Igrač će biti opremljen lukom i strijelom, te će kroz nasumično generirane razine morati pronaći svoj put do blaga kroz brojne neprijatelje.

4.1. Uvoz asseta

Pojam asset u doslovnom prijevodu znači imovina. Označava sve slike, modele, zvukove, animacije, skripte i svu ostalu imovinu koju koristimo kao dijelove za izradu video igre. Datoteke se u naš projekt uvoze jednostavno lijevim klikom miša na datoteku i povlačenjem ili u mapu gdje smo spremili naš projekt na računalu, ili direktno u Projekt prozor u alatu Unity.

4.2. Kretnja igrača

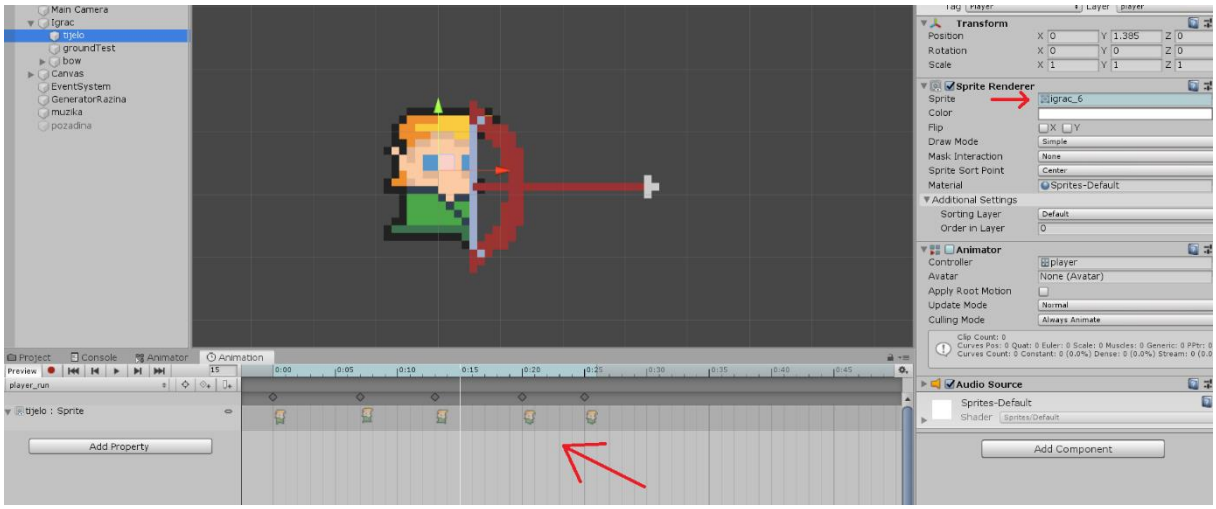
Kako bi dobili predodžbu kretanja igrača, potrebne su nam dvije animacije - jedna za prikaz igrača dok stoji na mjestu i jedna dok se kreće.

4.2.1. Animacija

U 2D igrama postoji nekoliko načina animiranja. Igrač se može sastojati od više slika (jedna za glavu, jedna za trup, jedna za svaku ruku i nogu i tako dalje), koje se onda pomiću kroz vrijeme kako bi dale iluziju kretanja. Ja sam za ovu igru odabrao nešto jednostavniji pristup. Sve animacije u ovoj igri su zapravo postignute mijenjanjem slike, ili sprite-a, kroz vrijeme. Na svakoj sličici igrač je postavljen u jednu pozu, a kada se te sličice brzo mijenjaju dobiva se iluzija pokreta, slično kao u filmu. Na ovaj način su se animirale mnoge klasične igre, kao na primjer Super Mario Bros. Na slici 7 možemo vidjeti sličice koje ćemo koristiti za kretnju igrača.

Slika 7: Sličice za animiranje igrača

Prve dvije s lijeve strane od ovih devet sličica koristiti ćemo za izradu animacije koja će se prikazivati dok igrač stoji, a četvrtu, petu, šestu i sedmu sličicu s lijeva koristiti ćemo za animaciju trčanja.

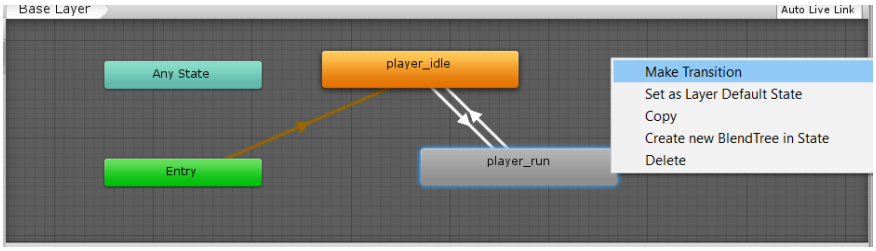


Slika 8: Prozor Animacija i svojstvo Sprite

Na slici 8 gore možemo vidjeti otvoren prozor Animation, koji u alatu Unity služi za animiranje. Brojevna crta pri vrhu tog prozora predstavlja protok vremena, a na nama je da okomitu bijelu crtu koju vidimo u tom prozoru postavimo na vremensku poziciju koju želimo i zadamo koja će se slika u tom trenutku prikazivati. To možemo napraviti jednostavnim povlačenjem određene sličice na željeno mjesto u prozor Animation, ili izravnim mijenjanjem svojstva Sprite u skripti Sprite Renderer koju možemo vidjeti u Inspektoru na desnoj strani ekrana. Na ovaj način ćemo izraditi sve animacije potrebne za našu igru, kao natezanje luka ili mahanje krilima letećih neprijatelja. Međutim, kada nakon izrade ove animacije pokrenemo igru, igrač će biti animiran, ali će izgledati kao da trči na mjestu, jer mu mi ni na koji način još nismo naznačili da se on treba kretati. Animacija nam služi samo kao vizualno sredstvo, ali sama po sebi ne donosi nikakve nove mehanike. Kako bi postigli da se igrač zapravo kreće kada pritisnemo neku tipku, potrebna su još dva koraka, postavljanje animatora i pisanje skripte za kretanje.

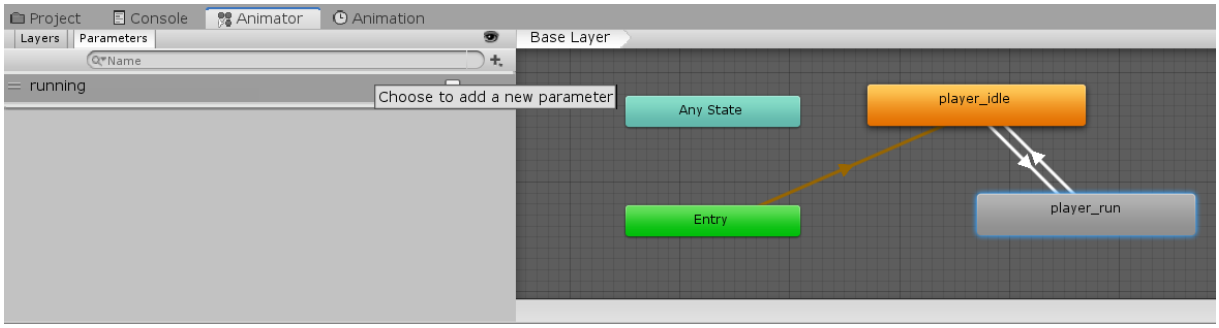
4.2.2. Animator

Odabirom objekta igrač i odabirom prozora Animator Unity nam nudi mogućnost povezivanja animacija i određivanja uvjeta prijelaza iz jedne u drugu animaciju, kako je vidljivo na slici 9.



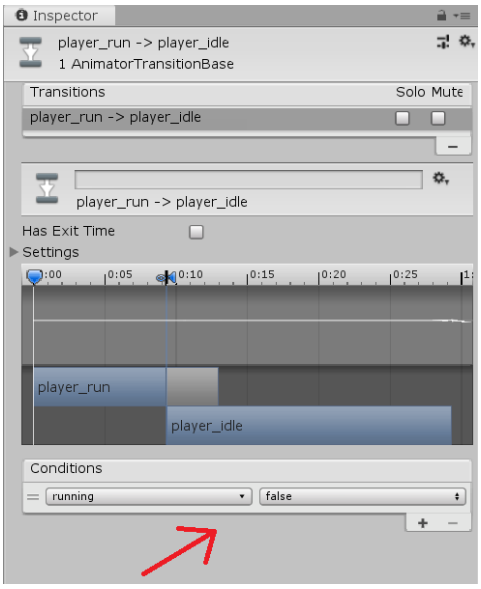
Slika 9: Animator i dodavanje nove poveznice

Nakon što smo napravili poveznice između animacija za željeni objekt, u ovom slučaju za našeg igrača, potrebno je stvoriti parametar čijom će se promjenom potaknuti promjena animacije koja je trenutno aktivna na odabranom objektu.



Slika 10: Animator i dodavanje parametara

Kao što možemo vidjeti na slici 10, ja sam taj parametar nazvao „running“, koja na engleskom jeziku znači „trčanje“. Zatim je potrebno odabrati pojedinu vezu između animacija u animatoru i odrediti pod kojim uvjetom će se aktivirati ta veza. Dakle, na vezi koja ide od animacije player_run do animacije player_idle postaviti ćemo uvjet za aktivaciju veze running=false, a za vezu koja pokazuje u suprotnom smjeru running=true. Na slici 11 prikazan je dio prozora Inspektor koji nam omogućuje postavljanje tih uvjeta.



Slika 11: Uvjet aktivacije poveznice

Ovim smo dobili mogućnost mijenjanja parametra running iz skripte za kretanje, te će to automatski potaknuti da se izvrši odgovarajuća animacija.

4.2.3. Skripta za kretanje

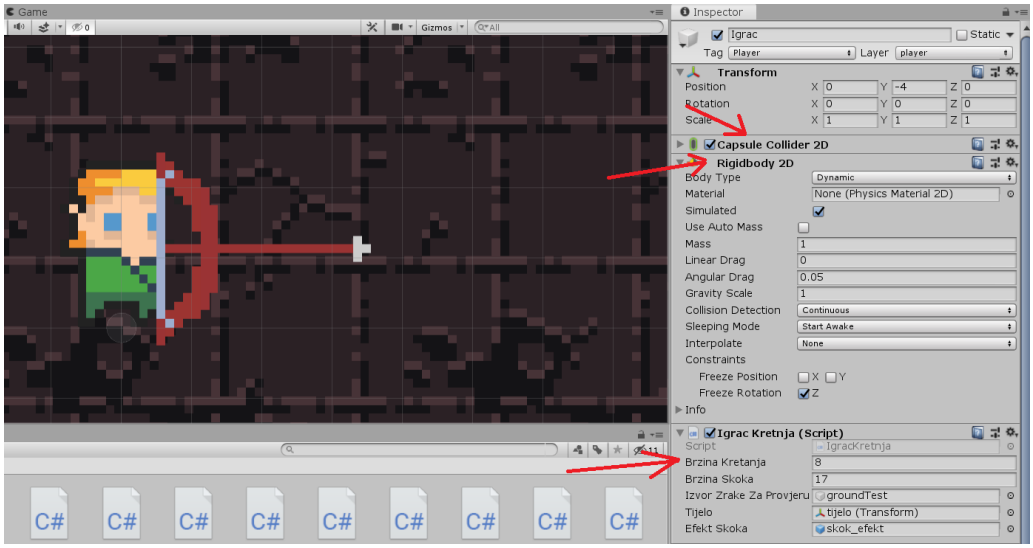
Kad kreiramo novu skriptu u alatu Unity i otvorimo je pomoću alata Visual Studio, ona će uvijek izgledati ovako:

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

Unutar klase, ali izvan funkcija Start i Update je mjesto gdje ćemo definirati varijable koje želimo koristiti u cijeloj skripti, ili im želimo pristupati iz drugih skripti. Ako ispred neke varijable napišemo ključnu riječ „public“, njoj ćemo moći pristupiti iz drugih skripti te će vrijednost te varijable biti vidljiva u prozoru Inspektor, gdje istu vrijednost možemo izravno mijenjati. Prikazivanje varijable i mogućnost promjene njene vrijednosti u Inspektoru vrlo je korisno za testiranje igre. Na primjer, bilo bi nezgodno da ako želimo podesiti brzinu kretanja igrača nakon testiranja svaki put moramo otvarati skriptu i mijenjati vrijednost varijable „brzina“ u samoj skripti. Ako želimo tu funkcionalnost, ali ne želimo da je varijabla „vidljiva“ iz drugih skripti, možemo prije deklariranja varijable napisati ključni pojam SerializeField, ovako:

```
[SerializeField] float brzinaKretanja;
```

Slika 12 prikazuje varijablu vidljivu u inspektoru, te joj u Inspektoru možemo i mijenjati vrijednost.



Slika 12: Varijabla brzinaKretanja vidljiva u inspektoru

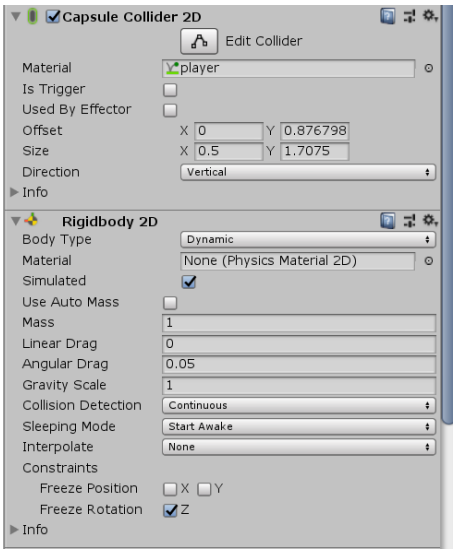
Svaka nova skripta sadrži dvije metode, Start() i Update(). Video igre rade tako da se određen broj puta u sekundi, najčešće 30 ili 60, za prikaz na monitoru „priprema“ nova slika. Slično kao u filmu, to nam daje iluziju kretanja i promjene u virtualnom svijetu. Metoda Start() se poziva prije prve generirane sličice u kojoj postoji objekt na kojem je ova skripta aktivna, a metoda Update() se poziva za svaku novu sličicu. Postoje još metode Awake() i FixedUpdate(), koje su slične metodama Start() i Update() respektivno, te preporučam bilo kome tko hoće nešto dublje znanje o samom programiranju igara da ih samostalno prouči. Za potrebe

shvačanja ovog rada biti će dovoljno razumijevanje metoda Start() i Update(). Pogledajmo konačnu inačicu skripte za kretanje igrača:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class IgracKretnja : MonoBehaviour
{
    bool naTlu;
    [SerializeField] float brzinaKretanja;
    [SerializeField] float brzinaSkoka;
    [SerializeField] GameObject izvorZrakeZaProvjeruPoda;
    [SerializeField] Transform tijelo;
    Rigidbody2D rgb;
    [SerializeField] GameObject efektSkoka;
    // Start is called before the first frame update
    void Start()
    {
        rgb = GetComponent<Rigidbody2D>();
    }

    // Update is called once per frame
    void Update()
    {
        naTlu = false;
        float xInput = Input.GetAxis("Horizontal");
        //kretnja lijevo/desno
        rgb.velocity = new Vector2(brzinaKretanja*xInput,
rgb.velocity.y);

        //provjera za skok

         if(Physics2D.Raycast(izvorZrakeZaProvjeruPoda.transform.positio
         n, Vector2.down,0.1f))
        {
            naTlu = true;
        }
        //skakanje
        if ((Input.GetKeyDown(KeyCode.W) ||
Input.GetKeyDown(KeyCode.UpArrow)) && naTlu)
        {
```

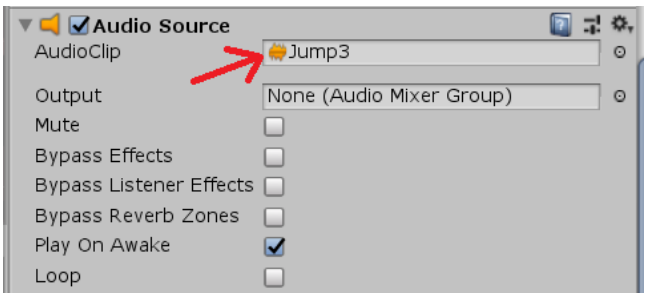
```
            rgb.velocity = new Vector2(rgb.velocity.x, brzinaSkoka);
            GetComponent<AudioSource>().Play();
            Instantiate(efektSkoka,
izvorZrakeZaProvjeruPoda.transform.position, Quaternion.identity);
        }

        //animiranje kretnje
        if (Mathf.Abs(rgb.velocity.x) > 0.5f)
        {
            tijelo.GetComponent<Animator>().SetBool("running", true);
        }
        else
        {
            tijelo.GetComponent<Animator>().SetBool("running", false);
        }


        if (Camera.main.WorldToScreenPoint(transform.position).x <
Input.mousePosition.x)
        {
            tijelo.localScale = new Vector2(1, tijelo.localScale.y);
        }
        else if (Camera.main.WorldToScreenPoint(transform.position).x >
Input.mousePosition.x)
        {
            tijelo.localScale = new Vector2(-1, tijelo.localScale.y);
        }
    }


}
```

U metodi Start() tražimo referencu na komponentu Rigidbody2D objekta na kojem je ova skripta, u ovom slučaju taj objekt će biti igrač. Kako bismo mogli upravljati brzinama nekog objekta i na njega mogli vrlo jednostavno djelovati silama, na primjer gravitacijom, on mora sadržavati komponentu Rigidbody2D. Ako želimo i da naš objekt ne prolazi kroz druge objekte, odnosno da ispituje kolizije s drugim objektima, taj objekt mora imati još i komponentu Collider2D. To je ovdje definitivno slučaj, jer ne želimo da igrač može prolaziti kroz pod i zidove. Na slici 13 vidimo te dvije komponente.



Slika 13: Komponente CapsuleCollider2D i Rigidbody2D

U metodi Update() obavljaju se 4 funkcije. Kod komentara „kretanje lijevo/desno“ vidimo na koji način se objektu mijenja brzina na temelju igračevih komandi. U odjeljku ispod komentara „provjera za skok“ iz pozicije igračevih nogu ispaljuje se virtualna zraka kojom se očitava stoji li igrač trenutno na nečemu ili je u zraku. Ova provjera služi kako igrač ne bi mogao skakati dok je u zraku, pa se tu i varijabla „naTlu“ postavlja na odgovarajuću vrijednost ovisno o tome pogodi li ispaljena virtualna zraka nešto ili ne. Ispod komentara „skakanje“ provjerava se da li je igrač pritisnuo tipku W ili strelicu prema gore na tipkovnici, i ako je, igraču se postavi određena brzina po osi y(prema gore). Zatim se uz skakanje izvrši odgovarajući audio efekt pomoću komponente AudioSource, koju smo također dodali na objekt igrača i postavili odgovarajuću audio datoteku koju AudioSource treba reproducirati. Na ovaj način pomoću komponente AudioSource reproduciraju se svi zvukovi u izrađenoj igri. Komponentu AudioSource na objektu koji predstavlja igrača i pripadajuću audio datoteku vidimo na slici 14:

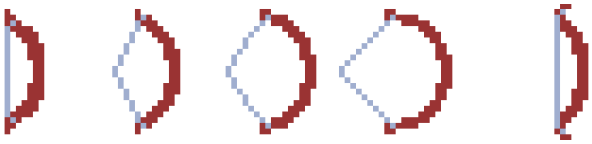


Slika 14: Komponenta AudioSource

Ispod komentara „animiranje kretnje“ postavi se odgovarajuća animacija stajanja ili trčanja, ovisno o apsolutnoj vrijednosti x komponente vektora brzine. To se postiže postavljanjem vrijednosti parametra u animatoru kojeg smo u prijašnjem poglavlju postavili. Nakon toga se još igrač okreće prema kursoru tako da se x komponenta njegove veličine postavi na -1 ili 1, ovisno o tome na kojoj strani igrača se kursor nalazi. Ovo radimo kako bi igrač uvijek bio okrenut u smjeru gdje trenutno cilja. Time je završena skripta za kretanje igrača.

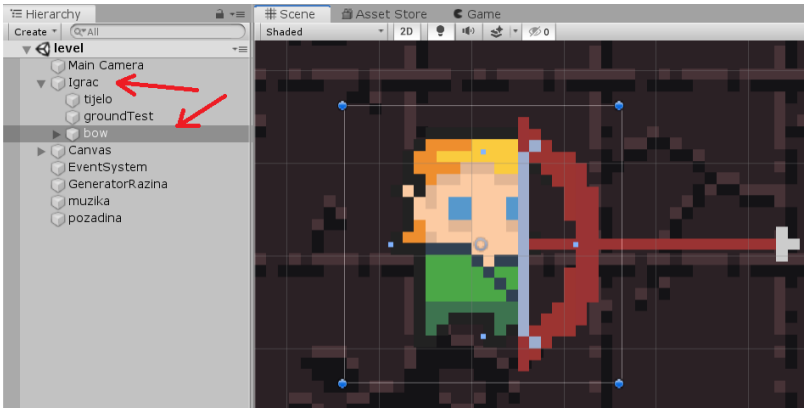
4.3. Luk i strijela

Na isti način kao i kod animiranja kretnje igrača, za luk sam izradio dvije animacije, jednu animaciju za luk dok miruje, i jednu za luk koji se napinje. Sličice koje sam koristio vidimo na slici 15.



Slika 15: Sličice za animaciju luka

Zatim sam postavio animator sa parametrom „aiming“ što na engleskom znači „natezanje“, te sam njega koristio kao uvjet za prijelaz iz animacije u animaciju. Zatim sam luk postavio kao podobjekt, ili objekt dijete, objekta igrača. Time sam postigao da se luk kreće zajedno s igračem, ali da je i dalje odvojeni objekt koji mogu nezavisno animirati, što ne bismo mogli da je luk dio slike samog igrača. Na slici 16 prikazano je kako to izgleda u hijerarhiji.



Slika 16: Objekt luk kao dijete objekta igrača

4.3.1. Luk

Pogledajmo skriptu Luk.

```
[SerializeField] GameObject arrowSlot;
[SerializeField] GameObject arrow;
[SerializeField] AudioClip[] zvukovi;
bool aiming = false;
```

Na početku skripte deklariraju se varijabla „aiming“, koja će služiti za praćenje stanja luka, odnosno da li je trenutno nategnut ili ne. Varijabla „arrowSlot“ sadrži referecu na objekt koji predstavlja mjesto gdje se ispaljene strijele stvaraju. Varijabla „arrow“ sadrži referecu na objekt koji predstavlja strijelu. Lista „zvukovi“ sadrži tri audio datoteke, od kojih ćemo svaku reproducirati uz određenu razinu nategnutosti luka.

```
void Update()
    {
        //rotacija
        Vector3 mousePos = Input.mousePosition;
        Vector3 objectPos =
Camera.main.WorldToScreenPoint(transform.position);
        mousePos.x = mousePos.x - objectPos.x;
        mousePos.y = mousePos.y - objectPos.y;
        float angle = Mathf.Atan2(mousePos.y, mousePos.x) *
Mathf.Rad2Deg;
        transform.rotation = Quaternion.Euler(new Vector3(0, 0, angle));
        //natezanje
        if (Input.GetMouseButtonDown(0))
        {
            if (aiming == false) StartCoroutine(Aim());
        }
    }
```

Neposredno ispod komentara „rotacija“ nalazi se kod koji je zaslužan za okretanje luka prema kursoru. Ovo omogućuje igraču da cilja. Ispod komentara natezanje vidimo dio koda zaslužan za natezanje luka. Ukoliko je igrač pritisnuo lijevu tipku miša, i ukoliko nije već počeo natezati luk, pokreće se korutina Aim().

Korutine su vrste metoda koje osobno jako često koristim. Razlog tomu je prvenstveno to što nam korutina pruža mogućnost jednostavnog upravljanja vremenom. U običnoj funkciji može biti izazovno izvesti neki kod tek nakon par sekundi, dok korutine imaju tu ugrađenu funkcionalnost. Ako želimo da između izvršavanja dviju uzastopnih naredbi prođe neki period

vremena, dovoljna je jedna naredba unutar korutine da nam to omogući. Ovdje tu specifičnu naredbu nisam koristio, ali sam iskoristio jednu drugu specifičnost korutina na koju ću poslije ukazati. Pogledajmo kako izgleda ta korutina:

```
IEnumerator Aim()
    {
        GetComponent<AudioSource>().clip = zvukovi[0];
        GetComponent<AudioSource>().Play();
        int power = 10;
        float timer = 0;
        GetComponent<Animator>().SetBool("aiming", true);
        while (Input.GetMouseButton(0))
        {
            timer += Time.deltaTime;
            if (timer > 0.5f && power<25)
            {
                power = 25;
                GetComponent<AudioSource>().clip = zvukovi[1];
                GetComponent<AudioSource>().Play();
            }
            if (timer > 1 && power<70)
            {
                power = 70;
                GetComponent<AudioSource>().clip = zvukovi[2];
                GetComponent<AudioSource>().Play();
            }
            yield return null;
        }
        ShootArrow(power);
        GetComponent<Animator>().SetBool("aiming", false);
        yield return null;
    }
```

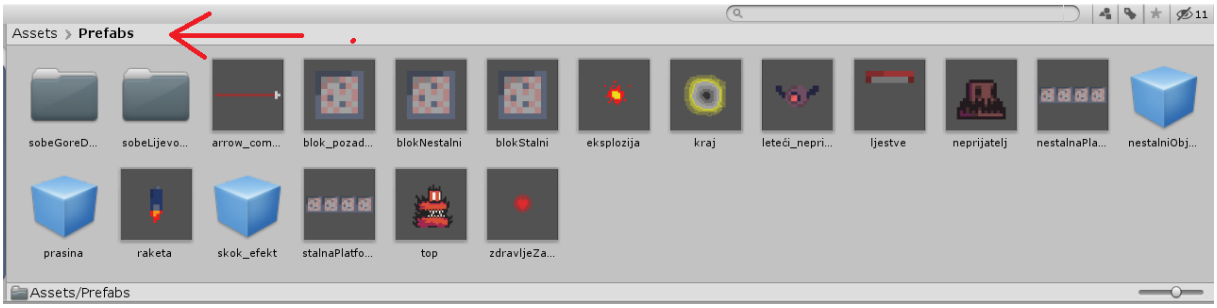
Čim se korutina pokrene reproducira se zvuk koji pripada prvoj razini nategnutosti luka. Postavi se parametar animatora kako bi se odigrala animacija natezanja. Također se odmah inicijaliziraju varijable „timer“ i „power“, koje služe za praćenje vremena i snage napetosti luka respektivno. Zatim ulazimo u while petlju, u kojoj se provjerava koliko je vremena prošlo od pokretanja korutine pomoću varijable „timer“, te se varijabla „power“ postavlja na odgovarajuću vrijednost. While petlja se odvija sve dok igrač i dalje drži pritisnuti lijevi klik miša. Sve to nam

omogućuje zadnja naredba unutar while petlje, zbog koje i nisam koristio običnu funkciju već korutinu. Ta naredba je „yield return null;“, a ona zaustavlja izvođenje koda unutar korutine do generiranja sljedeće sličice u igri. Time postižemo da se sav kod unutar while petlje odvija jednako često kao i kod unutar Update() metode.

Nakon izlaska iz while petlje, zaustavlja se animacija natezanja, te se poziva funkcija ShootArrow(), koja je zadužena za ispaljivanje strijele.

```
void ShootArrow(float speed)
    {
        GameObject spawnedArrow =
Instantiate(arrow,arrowSlot.transform.position + 2.789f *
transform.right, transform.rotation);
        spawnedArrow.GetComponent<Rigidbody2D>().velocity =
spawnedArrow.transform.right * speed;
        switch (speed)
        {
            case 10:
                spawnedArrow.GetComponent<StrijelaKretnja>().snaga = 1;
                break;
            case 25:
                spawnedArrow.GetComponent<StrijelaKretnja>().snaga = 2;
                break;
            case 70:
                spawnedArrow.GetComponent<StrijelaKretnja>().snaga = 3;
                break;
        }
    }
```

Funkcija ShootArrow() prvo stvara strijelu naredbom Instantiate, a zatim joj daje potrebnu brzinu, ovisno o tome koliko smo dugo natezali luk. Nakon toga se ovisno o brzini strijele zadaje javna varijabla „snaga“, koja se nalazi u skripti StrijelaKretnja stvorene strijele. Potrebno je naglasiti da kako bismo stvorili neki objekt, on prvo mora postojati kao asset koji možemo referencirati. To se postiže povlačenjem objekta iz prozora Hijerarhija u prozor Projekt. Time nastaje takozvani Prefab, što možemo gledati kao ideju objekta kojeg možemo stvarati, te će svaki objekt stvoren iz te ideje naslijediti sva svojstva te ideje. Slika 17 prikazuje sve Prefab-ove koje sam napravio i koristio tijekom izrade ove igre.



Slika 17: Prefabs

To je vrlo korisno kada, na primjer, u igri imamo tisuće i tisuće travki svijetlo zelene boje. Bilo bi jako teško promijeniti boju svakoj travki zasebno ukoliko bismo uvidjeli da bi u našoj igri bolje izgledale travke tamno zelene boje. Međutim, pomoću prefab-a dovoljno je samo na samom prefab-u napraviti promjene, a te promjene će se automatski propagirati na sve objekte koji potječu od njega.

4.3.2. Strijela

Pogledajmo skriptu StrijelaKretnja, koja se nalazi na objektu koji predstavlja strijelu.

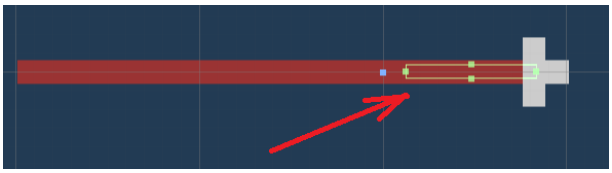
```
using UnityEngine;
public class StrijelaKretnja : MonoBehaviour
{
    [SerializeField] GameObject prasinaPriUdaru;
    bool stuck = false;
    public int snaga;
    // Update is called once per frame
    void Update()
    {
        if (!stuck)
        {
            Vector2 dir =
transform.GetComponent<Rigidbody2D>().velocity;
            float angle = Mathf.Atan2(dir.y, dir.x) * Mathf.Rad2Deg;
            transform.rotation = Quaternion.AngleAxis(angle,
Vector3.forward);
        }
    }
    private void OnCollisionEnter2D(Collision2D drugoTijelo)
    {
```

```
        Instantiate(prasinaPriUdaru, transform.position,
Quaternion.identity);
        GetComponent<Rigidbody2D>().velocity = Vector3.zero;
        GetComponent<Rigidbody2D>().gravityScale = 0;
        GetComponent<Rigidbody2D>().isKinematic = true;
        stuck = true;
        GetComponent<BoxCollider2D>().enabled=false;
        if(drugoTijelo.transform.tag == "enemy" ||
drugoTijelo.transform.tag == "raketa")
        {
            Vector3 originalScale = transform.localScale;
            transform.SetParent(drugoTijelo.transform);
            transform.localScale = originalScale;
        }
        if (drugoTijelo.transform.tag == "enemy")
        {
drugoTijelo.gameObject.GetComponent<NeprijateljZdravlje>().OduzmiZdravlj
e(snaga);
        }
    }
}
```

Dio koda u Update() funkciji postavlja rotaciju strijele koja odgovara vektoru smjera njene brzine. Slika 18 prikazuje kako je strijela u zraku izgledala prije i poslije implementacije tog koda.

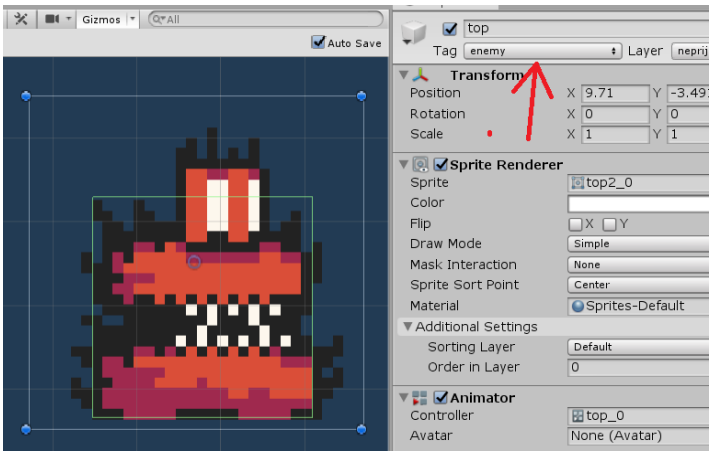
Slika 18: Strijela prije i poslije rotacije

Ispod Update() funkcije vidimo funkciju OnCollisionEnter2D(). To je funkcije koja se izvršava pri dodiru collider-a na objektu na kojem je skripta i collider-a na nekom drugom objektu. Prethodno smo strijeli dodali komponentu BoxCollider2D, kako je vidljivo na slici 19 ispod.



Slika 19: Collider strijele

Strijela se zaustavlja neovisno o tome u što je udarila. Ako tijelo koje je strijela udarila ima oznaku „enemy“ ili „raketa“, strijela će postati objekt dijete objekta kojeg je pogodila. To daje iluziju da se strijela zabila u tijelo protivnika, jer se ona sad kreće zajedno s protivnikom kojeg je pogodila. Također, krajnja selekcija tipa if provjerava ima li pogođeni objekt oznaku „enemy“. Na slici 20 naznačeno je crvenom strelicom gdje se objektima dodjeljuje oznaka.



Slika 20: Oznaka neprijatelja

Ako ima, to znači da je pogođeni objekt neprijatelj i da mu je potrebno oduzeti zdravlje pozivom javne funkcije OduzmiZdravlje() u skripti NeprijateljZdravlje koja je pridružena objektu koji predstavlja neprijatelja. Pogledajmo kako je implementirano zdravlje neprijatelja.

4.4. Neprijatelj

Svaki neprijatelj ima potpuno odvojenu skriptu za kretanje, ali svaka radi na sličan način te koristi instrumentarij alata Unity sa sličan način kao skripta IgračKretnja pa ih neću zasebno objašnjavati budući da se čitanjem ovog završnog rada do ove točke već steklo znanje potrebno za pisanje takvih skripti. Ono što je zajedničko svim neprijateljima, i ono što želim opisati je način na koji neprijatelji primaju „udarce“ strijele. Sličan sistem koristim u svim svojim

igrama, i želim pokazati koliko jednostavno se može izvesti. Pogledajmo kako je implementirano zdravlje neprijatelja:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class NeprijateljZdravlje : MonoBehaviour
{
    [SerializeField] int trenutnoZdravlje;
    [SerializeField] GameObject eksplozija;
    public void OduzmiZdravlje(int kolikoZdravljaOduzeti)
    {
        trenutnoZdravlje -= kolikoZdravljaOduzeti;
        if (trenutnoZdravlje <= 0) StartCoroutine(UnistiSe());
    }
    IEnumerator UnistiSe()
    {
        Instantiate(eksplozija, transform.position,
Quaternion.identity);
        Destroy(gameObject);
        yield return null;
    }
}
```

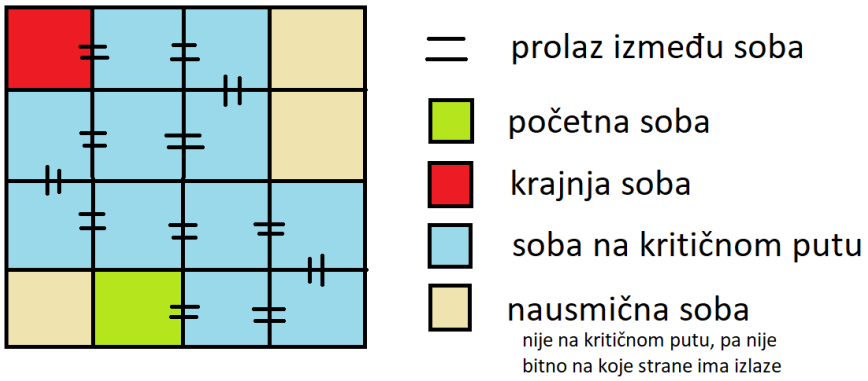
Praćenje zdravlja neprijatelja svodi se na praćenje vrijednosti jedne varijable „trenutnoZdravlje“, koju pozivom funkcije OduzmiZdravlje() igračeva strijela umanjuje, a ako zdravlje padne na vrijednost nula ili manje, neprijatelj se uništava. Na isti način funkcionira i zdravlje igrača.

4.5. Generacija razina

Kada smo napravili prefab-ove svih neprijatelja kako bi ih mogli dinamički stvarati po razinama, spremni smo upustiti se u programiranje nasumične generacije razina. Kad god želimo imati nasumičnu generaciju u nekoj igri, moramo krenuti od ideje.

4.5.1. Ideja

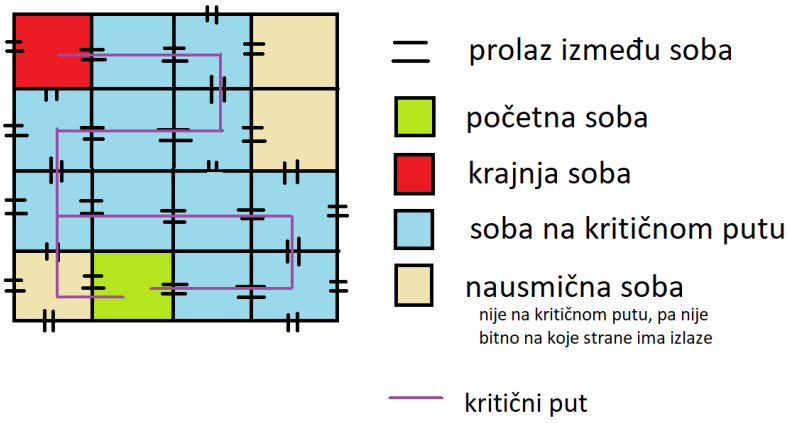
Znamo da je igračev cilj pronaći blago na svakoj razini, odnosno doći do kraja razine. To znači da kad se bavimo nasumičnom generacijom moramo biti jako oprezni. Potpuno nasumična generacija može igraču prepriječiti put do kraja razine. Zato sam uvidio da bi bilo dobro prvo generirati jedan „kritični put“ od početka do kraja razine koji je prohodan kako bi bili sigurni da će igrač uvijek moći pronaći blago na kraju razine. Kako bih si to olakšao, odlučio sam da će sve razine u mojoj igri biti sastavljene od mnoštva „soba“, koje imaju izlaze na različite strane, što će mi omogućiti da izradim prohodan kritični put. Najzahtjevniji dio ovog pothvata je samo generiranje kritičnog puta. Odlučio sam da će svaka razina biti pravokutnog oblika, sačinjena od pravokutnih soba. Igrač će uvijek počinjati razinu u nekoj od najnižih soba te razine, te će se morati postupno penjati prema gornjim sobama razine, gdje će u nekoj od njih biti blago. Ovakva ograničenja pomažu nam kod izrade nasumične generacije, jer sad imamo jasnu predodžbu kako bi kritični put trebao izgledati. Sobe koje nisu na kritičnom putu mogu biti generirane potpuno nasumično, jer nije ni bitno hoće li igrač moći doći do njih ili ne. Na slici 21 možemo vidjeti vizualni prikaz jedne razine generirane na ovaj način.



Slika 21: Primjer nasumično generirane razine

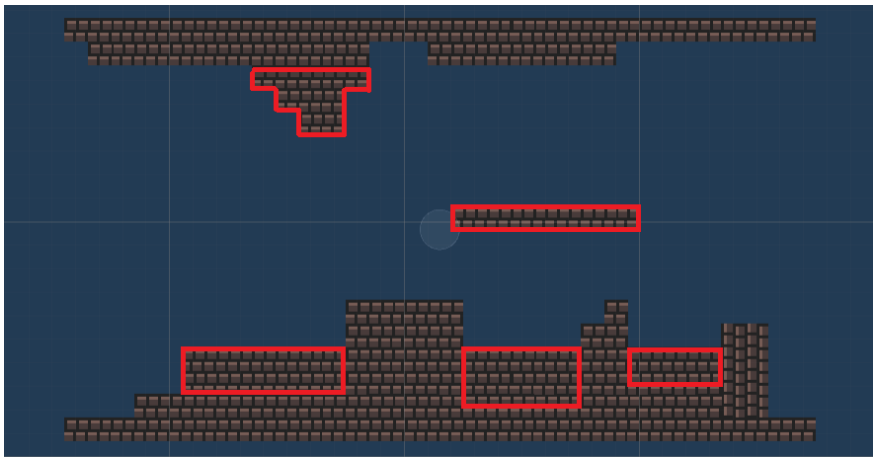
4.5.2. Sobe

Kako bi dobio dodatan efekt varijacije među razinama, podijelio sam sobe u kategorije ovisno o tome na koje strane imaju izlaze. Odlučio sam se na dvije kategorije, sobe koje imaju izlaze na lijevu i desnu stranu i sobe koje imaju izlaze na sve četiri strane. Razlog tomu bio je jednostavnost, ali i činjenica da sa samo dva tipa soba postoji veliki potencijal za vertikalno spajanje soba koje nisu na kritičnom putu, što mi se činilo kao zanimljiva ideja jer bi to moglo pružiti ponekad i više od jednog kritičnog puta, na način koji je prikazan na slici 22.



Slika 22: Nasumična generacija s dva kritična puta

Kako bih mogao putem skripte stvarati sobe, prvo sam morao napraviti prefab-ove soba. Odlučio sam da ću za svaki tip sobe napraviti nekoliko prefab-ova soba. To će mi omogućiti da kad mi treba na primjer soba s izlazima lijevo i desno, ne moram uvijek stvarati istu, nego mogu nasumično odabrati jednu sobu tog tipa. Kako bih još više igraču dao osjećaj nasumičnosti, u svakoj sobi postoje objekti koji imaju šansu da nestanu pri učitavanju razine. Takvi objekti na slici 23 ispod su naznačeni crvenim obrubom. To sam postigao tako što sam na te objekte dodao skriptu koja generira jedan nasumični broj od 0 do 99, te ako je taj broj manji od šanse za nestanak objekta iskazane u postotcima, taj objekt bi se uništio pozivom funkcije Destroy().



Slika 23: Objekti koji mogu nestati

4.5.3. Skripta GeneratorRazina

Sada kada smo odlučili na koji način naša generacija razina treba funkcionirati i kada smo pripremili potrebne sobe, možemo početi s proučavanjem skripte za nasumično generiranje razina.

```
     [SerializeField] GameObject jedanBlok;
    [SerializeField] int dimenzijeRazine;
    [SerializeField] List<GameObject> sobeTip1; //sobe s izlazima na
lijevu i desnu stranu
    [SerializeField] List<GameObject> sobeTip2; //sobe s izlazima
lijevo, desno, gore i dolje
    bool kraj = false;
    int[,] sobe;
    Vector2Int trenutnaSoba;
    Vector2Int prijasnjaSoba;
    Vector2Int pocetnaSoba;
    Vector2Int krajnjaSoba;
    [SerializeField] GameObject ObicniNeprijatelj;
    [SerializeField] GameObject LeteciNeprijatelj;
    [SerializeField] GameObject Top;
    [SerializeField] GameObject Srce;
    public int brojObicnihNeprijatelja;
    public int brojLetecihNeprijatelja;
    public int brojTopova;
    public int brojSrca;
    [SerializeField] GameObject krajRazine;
```

Na početku skripte GeneratorRazina deklariraju se razne varijable. Htio bih posebno obratiti pozornost na dvodimenzionalno polje tipa int pod imenom „sobe“. To dvodimenzionalno polje služi kao preslika razine koju treba generirati. To polje možemo zamisliti kao šahovnicu širine i visine n polja. Svako polje predstavlja jednu sobu koju ćemo generirati. Prije stvaranja samih soba, prvo ćemo pomoću algoritma nad tom šahovnicom odrediti kojem polju pripada koji tip sobe.

```
void Start()
    {
        PrepisiVrijednostiIzPostavki();
        sobe = new int[dimenzijeRazine,dimenzijeRazine];
```

```
        trenutnaSoba = new Vector2Int(Random.Range(0, dimenzijeRazine),
0);
        pocetnaSoba = trenutnaSoba;
        sobe[trenutnaSoba.x, trenutnaSoba.y] = 1;
        OdrediOstaleSobe();
        StvoriSveSobe();
        StvoriGranice();
        StvoriZdravljeZaPokupiti();
        StvoriNeprijatelje();
        UništiNeprijateljeOkoIgraća();
    }
```

U metodi Start() prvo se poziva funkcija PrepisiVrijednostiIzPostavki(), koja inicijalizira postavke razine tako da ih prepiše iz skripte PostavkeLevela.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public static class PostavkeLevela
{
    public static int zdravljeIgraca;
    public static int dimenzijeRazine;
    public static int brojObicnihNeprijatelja;
    public static int brojLetecihNeprijatelja;
    public static int brojTopova;
    public static int brojSrca;
    public static void PostaviPostavke(int redniBrojRazine)
    {
        switch (redniBrojRazine)
        {
            case 1:
                zdravljeIgraca = 3;
                dimenzijeRazine = 4;
                brojObicnihNeprijatelja = 10;
                brojLetecihNeprijatelja = 5;
                brojTopova = 5;
                brojSrca = 2;
                break;
            case 2:
```

```
                zdravljeIgraca =
GameObject.Find("Igrac").GetComponent<IgracZdravlje>().trenutnoZdravlje;
                dimenzijeRazine = 5;
                brojObicnihNeprijatelja = 12;
                brojLetecihNeprijatelja = 10;
                brojTopova = 15;
                brojSrca = 3;
                break;
            case 3:
                zdravljeIgraca =
GameObject.Find("Igrac").GetComponent<IgracZdravlje>().trenutnoZdravlje;
                dimenzijeRazine = 6;
                brojObicnihNeprijatelja = 20;
                brojLetecihNeprijatelja = 30;
                brojTopova = 25;
                brojSrca = 4;
                break;
                break;
        }
    }
}
```

Ovom statičnom klasom ostvario sam mogućnost jako jednostavnog proširivanja svoje video igre. Sve što je potrebno je napisati postavke za svaku razinu, a skripta GeneratorRazina će se pobrinuti za ostalo. To je jedan od bitnih principa kada se programiraju video igre. Uvijek je dobro što više izdvajati funkcionalnosti koje se mogu izdvojiti u zasebne skripte. To je dobra praksa jer nam omogućuje da radimo velike promjene u kasnijim fazama izrade igre na efikasan i modularan način.

Kada se dvodimenzionalno polje „sobe“ inicijalizira, svim vrijednostima na „šahovnici“ dana je vrijednost 0. Ukoliko želimo da neka soba ima izlaze samo na lijevu i desnu stranu, vrijednost ćemo postaviti na 1, a ukoliko želimo da neka soba ima izlaz na sve strane postaviti ćemo vrijednost pripadajućeg polja na 2. Sva polja koja imaju nakon izvršenja algoritma vrijednost 0 su polja koja nisu na kritičnom putu pa nije bitno koju ćemo sobu na pripadajućoj poziciji stvoriti.

Nakon prepisivanja postavki za razinu, nasumično se odabire soba koja je u dnu „šahovnice“ koja će imati ulogu početne sobe, te se vrijednost tog polja postavlja na 1.

Dalje u funkciji Start() skripte GeneratorRazina možemo vidjeti da se poziva funkcija OdrediOstaleSobe(). Ova funkcija zapravo je srž ovog završnog rada. Ona vrši algoritam koji određuje kritični put na šahovnici.

```
void OdrediOstaleSobe()
    {
        while (!kraj)
        {
            prijasnjaSoba = trenutnaSoba;
            bool uspjeh = false;
            while (!uspjeh)
            {
                int slucajniBroj = Random.Range(1, 6);
                if(slucajniBroj >= 1 && slucajniBroj <= 2)
                {
                    uspjeh = ProbajPomakULijevuStranu();
                }
                else if (slucajniBroj >= 3 && slucajniBroj <= 4)
                {
                    uspjeh = ProbajPomakUDesnuStranu();
                }
                else if (slucajniBroj == 5)
                {
                    uspjeh = ProbajPomakGore();
                }
            }
            if(trenutnaSoba.y == dimenzijeRazine-1 &&
                ((trenutnaSoba.x == 0 && sobe[1,dimenzijeRazine-1]>0)
                || (trenutnaSoba.x == dimenzijeRazine - 1 &&
sobe[dimenzijeRazine - 2, dimenzijeRazine - 1] > 0)))
            {
                kraj = true;
                krajnjaSoba = trenutnaSoba;
            }
        }
    }
```

Varijable „prijasnjaSoba“ i „trenutnaSoba“ su tipa Vector2Int, što znači da svaka od njih sadrži dvije cjelobrojne vrijednosti. One označuju poziciju polja na šahovnici koja predstavljaju

prijašnju sobu koju smo odredili i novu sobu koju tek želimo odrediti respektivno. Pamtiti poziciju sobe koju smo prethodno odredili korisno je kako bi se kod pomaka prema gore, što znači da je nova soba na kritičnom putu iznad prethodne, vrijednost polja koje korespondira prethodnoj sobi mogla postaviti na 2, što znači da i ta soba mora imati izlaz prema gore. To je je nužno jer, da bi se iz donje sobe prešlo u gornju, donja mora imati izlaz prema gore, a gornja izlaz prema dolje.

Sve dok nismo došli do kraja, pomoću while petlje događaju se pomaci po šahovnici tako da se na temelju nasumičnog broja odabire hoće li se pokušati trenutnu sobu (varijablu „trenutnaSoba“, koja će zapravo predstavljati sljedeću sobu na kritičnom putu) pomaknuti u lijevo, u desno ili prema gore. Ovo su funkcije koje provjeravaju je li pomak u određenom smjeru moguć, i ako je moguće rade pomak u tom smjeru i vraćaju vrijednost „true“, u suprotnom samo vrate vrijednost „false“ kako bi se ponovno pokušao pomak u neku stranu.

```
bool ProbajPomakULijevuStranu()
    {
        if (trenutnaSoba.x > 0)
        {
            if(sobe[trenutnaSoba.x - 1, trenutnaSoba.y] == 0)
            {
                trenutnaSoba.x -= 1;
                sobe[trenutnaSoba.x, trenutnaSoba.y] = 1;
                return true;
            }
        }
        return false;
    }
    bool ProbajPomakUDesnuStranu()
    {
        if (trenutnaSoba.x < dimenzijeRazine - 1)
        {
            if (sobe[trenutnaSoba.x + 1, trenutnaSoba.y] == 0)
            {
                trenutnaSoba.x += 1;
                sobe[trenutnaSoba.x, trenutnaSoba.y] = 1;
                return true;
            }
        }
        return false;
```

```
    }
    bool ProbajPomakGore()
    {
        if (trenutnaSoba.y < dimenzijeRazine - 1)
        {
            sobe[trenutnaSoba.x, trenutnaSoba.y] = 2;
            trenutnaSoba.y += 1;
            sobe[trenutnaSoba.x, trenutnaSoba.y] = 2;
            return true;
        }
        return false;
    }
```

Ovaj postupak pomicanja po dvodimenzionalnom polju ponavlja se sve dok ne dođemo do kraja. Kraj se očita kada se nalazimo na gornjem lijevom ili gornjem desnom polju šahovnice, te se varijabli „krajnjaSoba“ tipa Vector2Int pridruži vrijednost zadnjeg polja na kritičnom putu, što je dobro zapamtiti jer je to soba u kojoj ćemo trebati stvoriti blago.

Sada kada je funkcija OdrediOstaleSobe() odredila kritični put, možemo stvoriti sve sobe. Za to je zaslužna funkcija StvoriSveSobe() koja se odmah zatim poziva.

```
void StvoriSveSobe()
    {
        for(int i = 0; i < dimenzijeRazine; i++)
        {
            for (int j = 0; j < dimenzijeRazine; j++)
            {
                if (sobe[i, j] == 0)
                {
                    StvoriSporednuSobu(new Vector2Int(i, j));
                }
                if (sobe[i,j] == 1)
                {
                    StvoriSobuOdređenogTipa(new Vector2Int(i, j),
sobeTip1);
                }
                if (sobe[i, j] == 2)
                {
                    StvoriSobuOdređenogTipa(new Vector2Int(i, j),
sobeTip2);
```

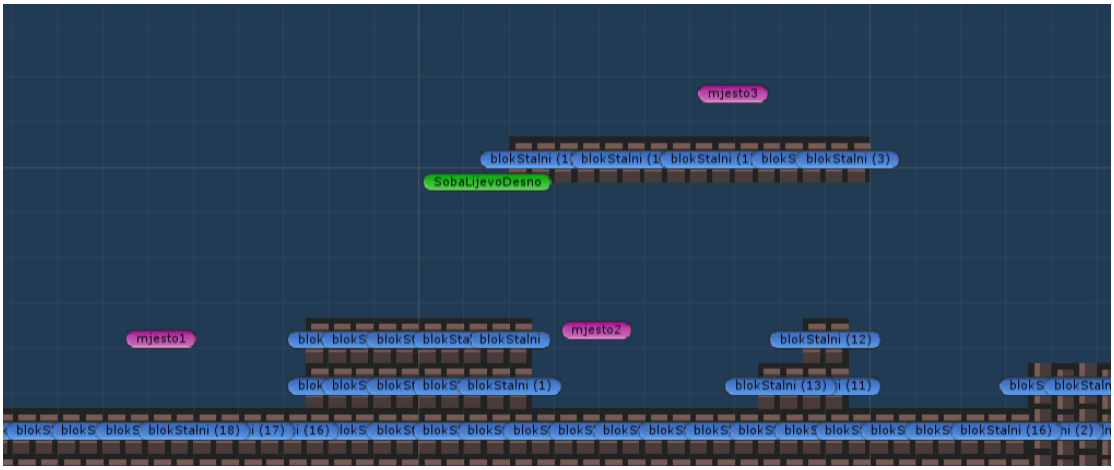
```
                }
            }
        }
    }
```

Prolazimo kroz sva polja šahovnice, odnosno dvodimenzionalnog polja cjelobrojnih brojeva pod nazivom „sobe“, te ovisno o vrijednosti na polju šahovnice koje je trenutno na radu stvaramo pripadajuću vrstu sobe. To nam omogućavaju funkcije StvoriSobuOdređenogTipa() i StvoriSporednuSobu().

```
void StvoriSobuOdređenogTipa(Vector2Int pozicija,List<GameObject>
listaMogucihSoba)
    {
        int nasumicniBroj = Random.Range(0, listaMogucihSoba.Count);
        GameObject sobaZaStvoriti = listaMogucihSoba[nasumicniBroj];
        Vector2 pozicijaNoveSobe = new Vector2(pozicija.x * 32,
pozicija.y * 18);
        GameObject stvorenaSoba = Instantiate(sobaZaStvoriti,
pozicijaNoveSobe, Quaternion.identity);
        //bojanje kriticnog puta
        //if (sobe[pozicija.x, pozicija.y] > 0)
        //{
          //  foreach(var slika in
stvorenaSoba.GetComponentsInChildren<SpriteRenderer>())
            //{
              //  slika.color = Color.red;
            //}
        //}
        if(pozicija == pocetnaSoba)
        {
            GameObject igrac = GameObject.Find("Igrac");
            Transform mjesta =
stvorenaSoba.transform.Find("mjestaZaStvoritiIgraca");
            int slucajniBroj = Random.Range(0, mjesta.childCount - 1);
            igrac.transform.position =
mjesta.GetChild(slucajniBroj).position;
        }
        if (pozicija == krajnjaSoba)
        {
            Transform mjesta =
stvorenaSoba.transform.Find("mjestaZaStvoritiIgraca");
            int slucajniBroj = Random.Range(0, mjesta.childCount - 1);
```

```
                Instantiate(krajRazine,
    mjesta.GetChild(slucajniBroj).position, Quaternion.identity);
            }
        }
void StvoriSporednuSobu(Vector2Int pozicija)
        {
            int nasumicniBroj = Random.Range(0,4);
            if(nasumicniBroj >= 0 && nasumicniBroj <= 2)
            {
                StvoriSobuOdređenogTipa(pozicija, sobeTip1);
            }
            if (nasumicniBroj == 3)
            {
                StvoriSobuOdređenogTipa(pozicija, sobeTip2);
            }
        }
```

Funkcija StvoriSobuOdređenogtipa() nasumično odabire jednu od soba traženog tipa i stvara ju na potrebnoj poziciji. Također i provjerava je li soba koju stvara početna soba, te ako jest postavlja igrača na jednu od pogodnih početnih pozicija. Na slici 24 možemo vidjeti neka takva mjesta pod nazivima „mjesto1“, „mjesto2“ i „mjesto3“ koja sam za svaku sobu ručno postavio kako se igrač ne bi pri pokretanju igre stvorio unutar zidova. Nakon toga još se provjerava je li soba koju trenutno stvaramo krajnja soba, te ako je, stvaramo blago na nekom od pogodnih mjesta.



Slika 24: Potencijalne pozicije za igrača

Funkcija StvoriSporednuSobu() na temelju nasumičo generiranog broja stvara sobu bilo kojeg tipa.

Nakon funkcije StvoriSveSobe() poziva se funkcija StvoriGranice(), koja služi kako bi osigurali da igrač ne može ispasti s razine, što je bez ove funkcije moguće ako na primjer neka od najnižih soba ima izlaz prema dolje.

```
void StvoriGranice()
    {
        //lijeva granica
        for(int i = 0; i<dimenzijeRazine*18; i++)
        {
            Vector2 pozicija = new Vector2(-15.5f, -8.5f + i);
            Instantiate(jedanBlok, pozicija, Quaternion.identity);
        }
        //desna granica
        for (int i = 0; i < dimenzijeRazine * 18; i++)
        {
            Vector2 pozicija = new Vector2(-16.5f + dimenzijeRazine *
32, -8.5f + i);
            Instantiate(jedanBlok, pozicija, Quaternion.identity);
        }
        //donja granica
        for (int i = 0; i < dimenzijeRazine * 32; i++)
        {
            Vector2 pozicija = new Vector2(-15.5f + i, -8.5f);
            Instantiate(jedanBlok, pozicija, Quaternion.identity);
        }
        //gornja granica
        for (int i = 0; i < dimenzijeRazine * 32; i++)
        {
            Vector2 pozicija = new Vector2(-15.5f + i, -8.5f +
dimenzijeRazine * 18);
            Instantiate(jedanBlok, pozicija, Quaternion.identity);
        }
    }
```

Stvaranje granica oko razine postiže se stvaranjem zasebnih blokova uzduž lijeve, desne, gornje i donje granice razine.

Zatim se poziva funkcija StvoriZdravljeZaPokupiti().

```
void StvoriZdravljeZaPokupiti()
    {
        int brojIteracija = 0;
        Vector2 pozicijaIgraca =
GameObject.Find("Igrac").transform.position;
        int brojStvorenihSrca = 0;
        while (brojStvorenihSrca < brojSrca)
        {
            brojIteracija++;
            if (brojIteracija > 5000)
            {
                Debug.Log("Ovdje je problem!");
                break;
            }
            Vector2 potencijalnaPozicija = new Vector2(Random.Range(-16,
32 * dimenzijeRazine - 16),
                Random.Range(-9, 18 * dimenzijeRazine - 9));
            if (!Physics2D.OverlapCircle(potencijalnaPozicija, 1f)
                && (Mathf.Abs(pozicijaIgraca.x - potencijalnaPozicija.x)
> 12
                || Mathf.Abs(pozicijaIgraca.y - potencijalnaPozicija.y)
> 8))
            {
                Instantiate(Srce, potencijalnaPozicija,
Quaternion.identity);
                brojStvorenihSrca++;
            }
        }
    }
```

Ova funkcija iterativno odabire nasumične pozicije unutar granica razine, te provjerava postoji li na toj poziciji već neki objekt pozivom metode Physics2D.OverlapCircle. Ukoliko se na toj poziciji već ne nalazi neki objekt i ukoliko ta pozicija nije pre blizu početne pozicije igrača, na njoj se stvara jedan „napitak“ kojeg igrač može pokupiti kako bi obnovio dio zdravlja. Ovo se ponavlja sve dok se ne stvori broj napitaka koji smo definirali za ovu razinu u klasi PostavkeLevela. Obratimo pažnju još na varijablu „brojIteracija“, koja služi kako bi while petlja prestala iterirati ako premašimo 5000 iteracija. U alatu Unity s while petljama treba biti jako oprezan. Ukoliko se nikad ne ispuni uvjet prekida neke while petlje, naša igra će se zamrznuti

i onemogućiti nastavak igranja, jer se nikad neće izaći iz while petlje, a to je potrebno kako bi se učitala sljedeća sličica. To se u ovoj specifičnoj skripti moglo dogoditi ako je razina pre mala i nema dovoljno mjesta za sve napitke koje smo specificirali. Unutar petlje će se odabrati nasumična pozicija, vidjeti da je ta pozicija zauzeta te će petlja krenuti u novu iteraciju. Kako naš kod ne bi „zaglavio“ u slučaju takvog spleta okolnosti, uveli smo sigurnosnu mjeru tako da pratimo broj izvršenih iteracija.

Odmah zatim poziva se funkcija StvoriNeprijatelje(), koja stvara specificirani broj svake vrste neprijatelja. Pogedajmo na koji način stvara jednu od tih vrsta neprijatelja:

```
//obicni neprijatelji
        brojIteracija = 0;
        int brojStvorenihObicnihNeprijatelja = 0;
        while (brojStvorenihObicnihNeprijatelja < 
brojObicnihNeprijatelja)
        {
            brojIteracija++;
            if (brojIteracija > 5000)
            {
                Debug.Log("Ovdje je problem!");
                break;
            }
            Vector2 potencijalnaPozicija = new Vector2(Random.Range(-16, 
32 * dimenzijeRazine - 16),
                Random.Range(-9, 18 * dimenzijeRazine - 9));
            if (!Physics2D.OverlapCircle(potencijalnaPozicija, 1.5f)
                && (Mathf.Abs(pozicijaIgraca.x - potencijalnaPozicija.x) 
> 12
                || Mathf.Abs(pozicijaIgraca.y - potencijalnaPozicija.y) 
> 8))
            {
                Instantiate(ObicniNeprijatelj, potencijalnaPozicija, 
Quaternion.identity);
                brojStvorenihObicnihNeprijatelja++;
            }
        }
```

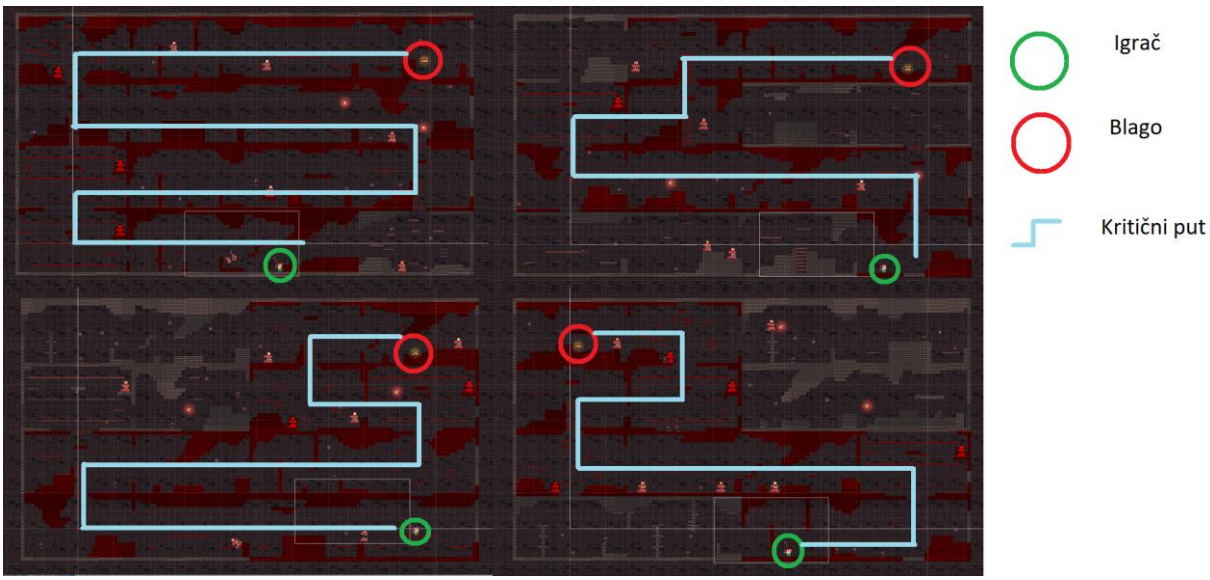
Ovo funkcionira na istom principu kao i funkcija koja stvara napitke, te se svi neprijatelji stvaraju na isti način.

Zadnja funkcija koja se poziva u skripti GeneratorRazina je UništiNeprijateljeOkoIgraća().

```
void UništiNeprijateljeOkoIgraća()
    {
        Vector2 pozicijaIgraca =
GameObject.Find("Igrac").transform.position;
        Collider2D[] lista = Physics2D.OverlapCircleAll(pozicijaIgraca,
12f);
        foreach(Collider2D tijelo in lista)
        {
            if(tijelo.tag == "enemy")
            {
                Destroy(tijelo.gameObject);
            }
        }
    }
```

Ova funkcija pronađe sve objekte u određenom radijusu oko igrača koji imaju oznaku neprijatelja, te ih uništi. Ovo radimo kako igrač ne bi bio opkoljen neprijateljima pri pokretanju razine.

Nakon toga, razina je potpuno generirana sa svim neprijateljima, napitcima i blagom. Slika 25 prikazuje nekoliko razina generiranih korištenjem ovog algoritma.



Slika 25: Nekoliko nasumično generiranih razina

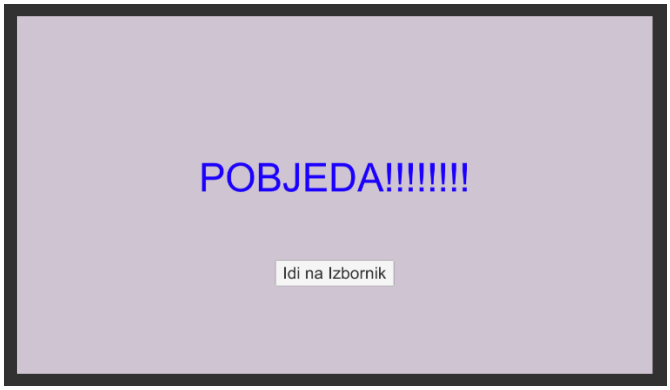
4.5.4. Skripta Kraj

Kako bismo mogli prijeći svaku razinu i cijelu igru, napisao sam skriptu Kraj koju sam dodao na objekt blaga.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
public class Kraj : MonoBehaviour
{
    public static int redniBrojSljedećeRazine = 2;
    private void OnTriggerEnter2D(Collider2D drugoTijelo)
    {
        if(drugoTijelo.tag == "Player")
        {
            if (redniBrojSljedećeRazine < 4)
StartCoroutine(NoviLevel());
            else StartCoroutine(ZavrsiCijeluIgru());
        }
    }
    IEnumerator NoviLevel()
    {
        GetComponent<AudioSource>().Play();
        Time.timeScale = 0;
        yield return new WaitForSecondsRealtime(2);
        Time.timeScale = 1;
        PostavkeLevela.PostaviPostavke(redniBrojSljedećeRazine);
        redniBrojSljedećeRazine++;
        SceneManager.LoadScene("level");
        yield return null;
    }
    IEnumerator ZavrsiCijeluIgru()
    {
        GetComponent<AudioSource>().Play();
        Time.timeScale = 0;
        yield return new WaitForSecondsRealtime(2);
        Time.timeScale = 1;
        SceneManager.LoadScene("GameWon");
        yield return null;
    }
```

```
}
```

Kada igrač dotakne blago, ova skripta će pokrenuti novu razinu ili će završiti cijelu igru, ovisno o tome koju razinu smo sad prešli. Ukoliko je potrebno završiti cijelu igru jer je igrač prošao sve razine, učitava se scena „GameWon“, koja sadrži poruku o pobjedi te mogućnost povratka na glavni izbornik, kako je vidljivo na slici 26.



Slika 26: GameWon scena

4.5.5. Glavni izbornik

Sve što je preostalo kako bi naša igra postala zaokružena cjelina je glavni izbornik. Svi elementi izbornika su primarno estetskog značaja, osim dva gumba za pokretanje nove igre i izlazak iz aplikacije, koje možemo vidjeti u gornjem desnom i donjem desnom kutu slike 27.

Slika 27: Glavni izbornik

Pritiskom na gumb „Nova igra“ poziva se funkcija NovaIgra(), a pritiskom na gumb „Ugasi“ funkcija Ugasi().

```
public void NovaIgra()
    {
        Application.LoadLevel("level");
        PostavkeLevela.PostaviPostavke(1);
    }
    public void Ugasi()
    {
        Application.Quit();
    }
```

Ovim smo igraču omogućili pokretanje prve razine, te je naša video igra sada završena.

5. Zaključak

Tema ovog završnog rada je izrada 2D igre s generiranim razinama. Proučio sam žanr igara pod nazivom „roguelike“, te razradio ideju igre čija je izrada bila praktični dio rada. Igru sam izradio pomoću alata Unity i Microsoft Visual Studio. Implementirao sam algoritam za nasumičnu generaciju razina, te sam objasnio programski kod nekoliko skripti za koje smatram da zajedno pokazuju znanje potrebno za izradu video igre.

Način na koji se nasumična generacija implementira kako bi omogućila željeni način igranja video igre uistinu mi je predstavljao velik izazov. Glavna mehanika moje igre je pucanje strijele iz luka, te sam se bojao da će nasumična generacija proizvesti razine s puno prepreka koje će ometati tu mehaniku, a samim tim i zabavu. Međutim, proučavanjem sličnih igara, kao Spelunky, došao sam do kreativnih rješenja do kojih bi teško samostalno došao. Rekao bih da je najbitnija stvar koju sam kroz ovaj rad naučio bila važnost proučavanja drugih igara, kako za inspiraciju tako i za načine rješavanja naizgled kompleksnih problema, kao na primjer nasumične generacije razina.

Iako sam već nekoliko igara izradio u alatu Unity, kroz ovaj završni rad stekao sam mnoge vještine i znanja, kako o alatu Unity, tako i o izradi video igara općenito. Volio bih se izradom video igara jednog dana baviti profesionalno, a nakon ovog projekta osjećam se korak bliže ostvarenju tog sna. Drago mi je da sam za svoj završni rad odabrao baš ovu temu, te bih alat Unity preporučio svima koji su zainteresirani za izradu video igara.

Popis literature

[1] No Man's Sky Wiki, Procedural generation, 2019. [na internetu]. Dostupno: https://nomanssky.fandom.com/wiki/Procedural_generation [pristupano 11.9.2019]

[2] Steam, Spelunky, 2019. [na internetu]. Dostupno: https://store.steampowered.com/app/239350/Spelunky/ [pristupano 11.9.2019]

[3] Darius Kazemi, Spelunky Generator Lessons, 2019. [na internetu]. Dostupno: http://tinysubversions.com/spelunkyGen/ [pristupano 11.9.2019]

[4] Gamepedia, Cube World - Castle, 2019. [na internetu]. Dostupno: https://cubeworld.gamepedia.com/Castle [pristupano 11.9.2019]

[5] HumbleBundle, No Man's Sky, 2019. [na internetu]. Dostupno: https://www.humblebundle.com/store/no-mans-sky [pristupano 11.9.2019]

[6] Unity, Scripting API Manual, 2019. [na internetu]. Dostupno: https://docs.unity3d.com/560/Documentation/ScriptReference/index.html [pristupano 11.9.2019]

[7] Carter Dotson, The Beginner's Guide to Roguelikes, 2019. [na internetu]. Dostupno: https://www.lifewire.com/what-are-roguelikes-4117411 [pristupano 11.9.2019]

Popis slika