Izrada puzzle platformer igre u alatu Unity

Ćurko, Domagoj

Master's thesis / Diplomski rad

2021

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:211:241066

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Domagoj Ćurko

Izrada puzzle platformer igre u alatu Unity

DIPLOMSKI RAD

Varaždin, 2021.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Domagoj Ćurko

Matični broj: 44905/16–R

Studij: Informacijsko i programsko inženjerstvo

Izrada puzzle platformer igre u alatu Unity

DIPLOMSKI RAD

Mentor:

Prof. dr. sc. Danijel Radošević

Varaždin, studeni 2021.

Domagoj Ćurko

Izjava o izvornosti

Izjavljujem da je moj 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

U ovom diplomskom radu ću predstaviti žanr igara poznat kao "puzzle platformer" , te ću proučiti principe dobrog dizajna razina u videoigrama. Također ću opisati instrumentarij programskog alata Unity za izradu takvih igara. U praktičnom dijelu rada izradit će se puzzle platformer računalna igra, te će se prikazati primjena naučenih principa za dobar dizajn razina. Glavni lik, odnosno igrač, koristi magnetizirane sjekire kako bi promijenio magnetska svojstva objekata oko sebe i time ostvario prolazak razine. Veliku pozornost obratiti ću na dobre principe dizajna razina. Sam programski kod pisan je u programskom jeziku C# u alatu Microsoft Visual Studio. Cilj rada je pružiti uvid u proces izrade video igre, te pokazati kako se principi dobrog dizajna razina mogu primijeniti u realnom praktičnom projektu.

Ključne riječi: unity, puzzle platformer, dizajn razina, igra, algoritmi

Sadržaj

1. Uvod

Tema ovog diplomskog rada je Izrada puzzle platformer igre u alatu Unity. Proučiti ću žanr igara pod nazivom „puzzle platformer“, te ću predstaviti ideju za videoigru čija će izrada sačinjavati praktični dio ovog diplomskog rada. Proučiti ću principe dobrog dizajna razina, te razložiti kako se ti principi mogu primijeniti u igri ovog žanra. Kako bih izradio videoigru koristiti ću alat Unity, koji koristi C# programski jezik i Microsoft Visual Studio aplikaciju kao uređivač teksta. Ideju za igru koju ću izraditi osmislio sam samostalno razmišljajući o zanimljivim mehanikama za dodati u jednu drugu videoigru na kojoj sam radio, ali mi se koncept prelaženja razina korištenjem magnetizma toliko svidio da sam odlučio oko njega izraditi novu igru.

Već nekoliko godina bavim se izradom videoigara (eng. game development). Kada sam se prvi put upustio u svijet game development-a, uskoro sam shvatio da napredno znanje kodiranja i algoritama nije pretežito korisno. Prije tog iskustva, imao sam svakakve ideje za vlastite videoigre, ali mislio sam da je najveća prepreka moje znanje, odnosno manjak znanja, o programiranju. U sklopu studija naučio sam programirati, i mislio sam da me je to učinilo spremnim za izradu svoje prve igre. I uistinu, uspješno sam izradio i objavio svoju vlastitu video igru, i usput shvatio da je potrebna razina razumijevanja programiranja mnogo manja nego sam prvotno očekivao. Međutim, igru nije preuzelo mnogo ljudi, a i oni koji jesu, nisu imali najbolje dojmove.

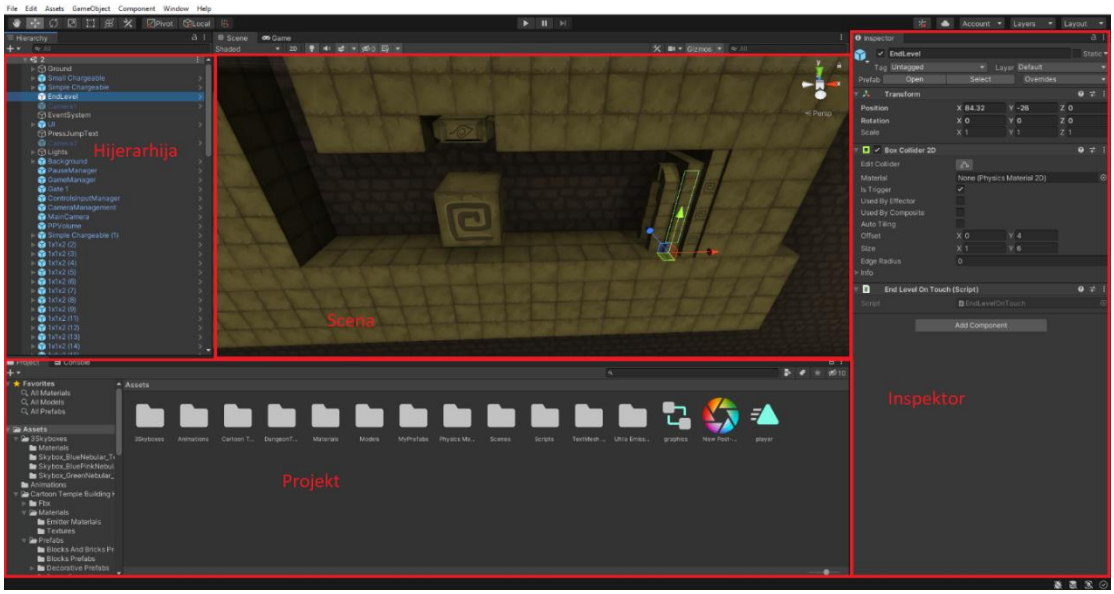
Iako i dalje smatram da su glavne mehanike igre bile zabavne i dobro izvedene, te su mi mnogi igrači to i sami rekli, razine nisu bile dobro osmišljene. Neke razine igrači jednostavno nisu mogli prijeći jer se od njih zahtijevalo razumijevanje mehanike koju prethodno nisam dovoljno predstavio. Kroz to iskustvo sam naučio da najveći problem nije izrada igre, nego napraviti igru zabavnom. Veliku ulogu u zabavi ima dizajn razina, te ću iz tog razloga u sklopu ovog diplomskog rada proučiti što sve možemo napraviti kako bi poboljšali iskustvo igranja korištenjem nekih principa dobrog dizajna razina, te ću sve te principe prikazati na praktičnom primjeru.

2. Unity

Unity je popularni alat za izradu video igara (eng. Game Engine). Postao je javno dostupan u lipnju 2005. godine, a danas podržava izradu projekata za više od 25 različitih platformi. Njime se možemo koristiti za izradu 2D i 3D igara, a podržava i razvoj igara koje koriste virtualnu i proširenu stvarnost. Na tržištu postoji mnogo opcija za developere računalnih igara, a Unity se među konkurencijom ističe iznimnim mogućnostima, kvalitetom i lakoćom korištenja, te činjenicom da ga je moguće koristiti potpuno besplatno u slučajevima gdje kompanija ili pojedinac koji koristi Unity platformu zarađuje manje od 100,000 američkih dolara godišnje. Velik postotak game developera čine upravo učenici i studenti koji nemaju sredstva za priuštiti skupi game engine, što čini Unity vrlo privlačnom opcijom. Danas je Unity jedan od najpopularnijih alata za izradu video igara na tržištu.[1]

2.1. Unity korisničko sučelje

Alat Unity koristi jednostavan i intuitivan dizajn korisničkog sučelja. Kada po prvi put izradi novi projekt u alatu Unity, korisnik se susreće sa bazičnim korisničkim sučeljem (UI), koje se sastoji od 4 prozora: hijerarhija, inspektor, scena i projekt. Na slici dolje prikazano je Unity korisničko sučelje sa nažnačenim dijelovima.



Slika 1: Unity korisničko sučelje

Prozor Projekt (eng. Project) služi kao prikaz direktorija u kojem je sadržan projekt, te kroz njega možemo pristupiti svim skriptama, modelima, slikama, zvučnim efektima te ostalim asset-ima. Kada želimo kreirati novu skriptu u našoj igri, potrebno je desnim klikom u Projekt prozor i odabirom opcije New -> C# Script kreirati novu skriptu, koju kasnije možemo koristiti u igri.

Svaka scena u Unity-u predstavlja zasebni 3D prostor, te jedna scena često predstavlja jednu razinu u igri ili dio razine.

U pogledu Scena (eng. Scene View) prikazan je renderirani 3D prostor trenutne scene na kojoj radimo, te su učitani svi objekti koje smo postavili u trenutnu scenu.

U prozoru Hijerarhija (eng. Hierarchy) prikazani su objekti iz trenutne scene u obliku izbornika s izlistanim imenima objekata u sceni. Klikom na bilo koji objekt u hijerarhiji odabiremo taj objekt te se on odabire i u sceni, a isto tako možemo odabrati objekt u sceni, te će se on označiti u hijerarhiji.

U prozoru Inspektor (eng. Inspector) prikazani su podaci o trenutno odabranom projektu.

Svi podaci jednog objekta učahureni su u takozvane „komponente“. Komponente predstavljaju skripte koje su pridružene objektima, kako bi im se dodijelili podaci i ponašanje. Tako recimo svaki objekt ima komponentu (skriptu) naziva Transform, u kojoj su zabilježeni podaci o poziciji, rotaciji i veličini objekta. Kako bi na primjer pomicali objekt kroz prostor, trebamo u kodu dohvatiti njegovu Transform komponentu i promijeniti varijablu koja predstavlja jednu od njegovih koordinata u 3D prostoru.

2.2. Unity Asset Store

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 video igre.

Takva imovina naziva se „asset“ u svijetu game developmenta, a činjenica da postoji mnoštvo potpuno besplatnih asseta na platformi Unity Asset Store koje možemo koristiti u izradi svojih igara uvelike pojednostavljuje proces izrade videoigre. To mogu potvrditi i iz osobnog iskustva, jer nikad nisam bio dobar u vizualnom dizajnu, te nikad nisam mogao stvoriti koherentnu scenu u igri koristeći vlastite modele.

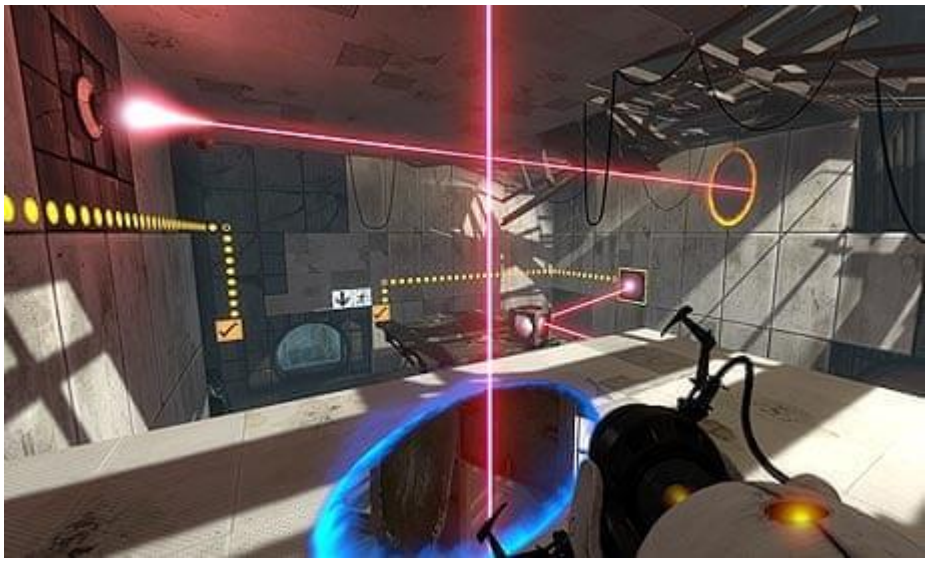
Tu mi je Unity Asset Store skratio vrijeme provedeno na učenje i proizvodnju 3D modela, te bih jednostavno preuzeo paket asseta koji tematski pristaje mojoj ideji igre.

Iako se u game development krugovima ne podržava korištenje asseta samo sa Asset Store-a, većini igraća to neće smetati ukoliko se svi modeli i 2D slike tematski uklapaju.

3. Puzzle platformer žanr igara

Puzzle platformer žanr je podžanr Platformer žanra, često nazivanog „jump 'n' run“ žanr. To je žanr videoigara u kojima je glavna misija pomicanje igrača kroz prostor, često oslanjajući se na vještinu samog kretanja u igri, kao i na sposobnost igrača da riješi zagonetke koje mu igra predstavlja. Često su zagonetke u takvim igrama isprepletene i usko povezane s središnjom mehanikom igre.[2]

Središnja mehanika igre predstavlja srž igre, odnosno radnju koju igrač iznova i iznova ponavlja u videoigri. Tako na primjer možemo reći da je pucanje iz oružja glavna mehanika većine akcijskih videoigara iz prvog lica, ili da je glavna mehanika u popularnoj puzzle platformer igri „Portal 2“ stvaranje portala koje igrač koristi kako bi se navigirao kroz razine.



Slika 2: Igra Portal 2 [8]

Puzzle platformer igre bile su jako popularne u ranim danima videoigara. Puzzle platformer igre 1998. godine su zauzimale 15% udjela tržišta video igara, dok je već u 2006. godini njihova popularnost naglo pala i sačinjavale su samo 2% tržišta, donekle kao posljedica razvijanja drugih žanrova kao što su pucačine iz prvog lica. Ipak, i danas postoji velik broj igrača koji su spremni igrati puzzle platformer igre, pa se neke igre u tom žanru prodaju u milijunima kopija.[2]

Za ovaj diplomski rad izradit ću igru ovog žanra, te ću objasniti kako primijeniti principe dobrog dizajna razina.

3.1. Principi dobrog dizajna razina

Dobar dizajn razina jako je bitan za videoigru, a to je pogotovo istina za igre puzzle platformer žanra. Kod igara u puzzle platformer žanru zabava se uvelike zasniva na kombinaciji kvalitete dizajna razina i zanimljivosti središnje mehanike, te koliko dobro ta mehanika i dizajn razina rade skupa. U ovom poglavlju proučiti ćemo neke principe dobrog dizajna razina, te kako neke popularne puzzle platformer igre koriste te principe kako bi povećali kvalitetu svoje igre.

3.2. Postupno učenje igrača

Kako bi razine u videoigri bile zanimljive, bitno je da igrač na svakoj razini nauči nešto novo kako bi uspio prijeći razinu. Nije dobra praksa imati razine u igri koje ne zahtijevaju rješavanje novih problema, jer će igraču takve razine brzo postati dosadne radi previše ponavljajućeg iskustva. Zato je korisno u svakoj razini predstaviti igraču barem jedan problem koji je na neki način drugačiji od svih prijašnjih.[3]

Također je bitno učenje igrača provoditi postupno. Nije dobro pred igrača staviti problem za koji on još nije spreman jer se od njega traži razumijevanje tri nove mehanike koje nije imao vremena svladati. Kako bi to izbjegli, dobro se držati pravila da se u svakoj zagonetci od igrača shvati samo jedna nova mehanika.

Na primjer, u prvoj razini igre Super Mario Bros od igrača se zahtijeva da nauči skakati.

Slika 3: Super Mario Bros

Kako je prikazano na slici gore, na samom početku razine igra predstavlja prvu prepreku, odnosno protivnika. Ovu razinu nije moguće prijeći ukoliko igrač ne nauči kako preskočiti tog protivnika, što osigurava da je igrač uistinu naučio skakati i spreman je za nove izazove.

3.3. Izbjegavati ponavljanje

Dobra praksa je izbjegavati ponavljanje ideja i zagonetki u više razina ili unutar iste razine, jer igrač nakon prvog rješavanja tog problema već zna kako riješiti zagonetku. Ponavljanje motiva je u redu ukoliko se na neki način promijeni iskustvo igrača.[4]

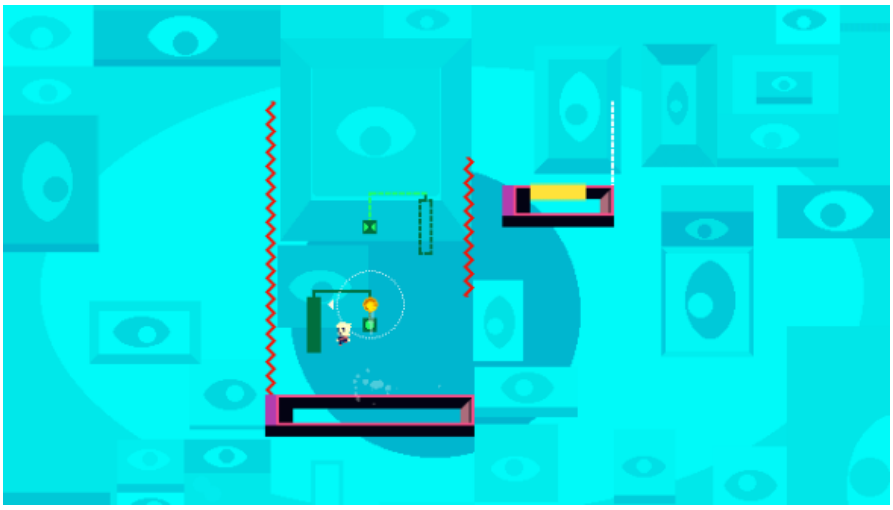
Savršen primjer toga je igra Halo 3, koja u misiji „The Ark“, nakon što igrač probije svoj put kroz razinu, zahtijeva od igrača da ide unatrag i prijeđe istu razinu još jedan put. Međutim, prvi put je igrač razinu prelazio kao pješak, a drugi put prelazi razinu u tenku. Iako je razina ostala ista, iskustvo igranja nije isto radi te promjene, te igra ostaje zabavna.[5]

Slika 4: Igra Halo 3 [5]

3.4. Traženje višestrukih primjena za mehanike

Jedna od najbitnijih pravila kod proizvodnje igara puzzle platformer žanra je ne zatrpati igrača sa mnoštvom mehanika i pravila igre, jer to može dovesti do pre kompliciranih kontrola i zbunjivanja igrača. Umjesto toga, dobro je pronaći što više primjena osnovnih mehanika. Tako broj mehanika igre ostaje niži što smanjuje frustraciju igrača.

Na primjer, u igri HackyZack postoji mehanika gdje bi, ukoliko igrač stisne odgovarajući prekidač, nestali određeni blokovi u razini, što bi igraču omogućilo prolaz. Zatim su proizvođači te igre dodali mogućnost da se istom akcijom pritisne prekidač koji stvara određene blokove kako bi igrač prešao razinu. Dakle, radi se o samo jednoj mehanici, pritiskanje prekidača, ali ona ima dvije različite funkcije.[4]



Slika 5: Igra HackyZack [4]

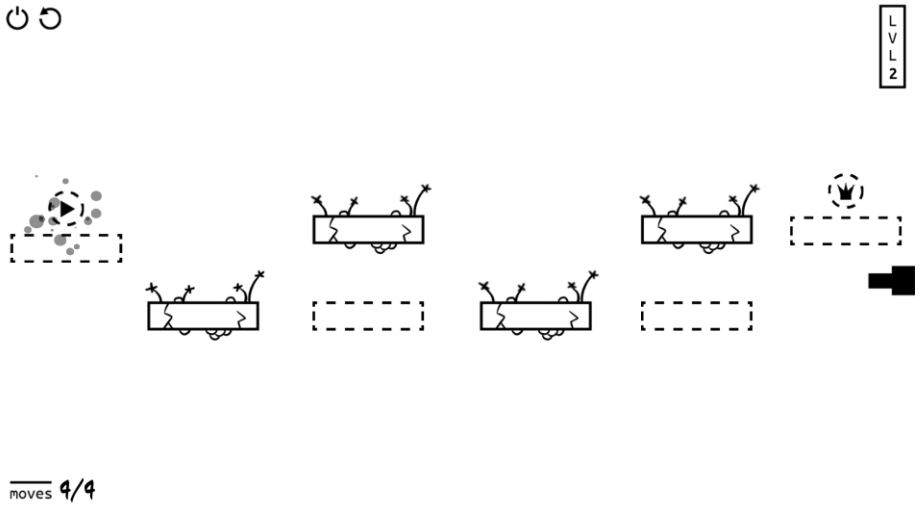
3.5. Reći igraču što napraviti, ali ne i kako

Kada dizajniramo razine, često smo u kušnji objasniti igraču što treba napraviti kako bi riješio problem ili zagonetku. To dizajneri razina često rade iz straha da igrači ne bi zapeli na nekom dijelu njihove igre, što bi onemogućilo daljnju igru.

Međutim, tim pristupom riskiramo da igrač pri rješavanju problema ne osjeća dovoljno zadovoljstvo, jer se osjeća kao da je vođen za ruku do rješenja. Bolji pristup je pobrinuti se da igrač zna što mu je činiti, ali da sam mora odgonetnuti kako to izvesti.[3]

Kako bi se pobrinuli da igrač zna što mu je činiti, moramo u dizajniranju razine staviti fokus na cilj, te na elemente s kojima igrač može vršiti interakciju kako bi postigao taj cilj. Međutim, kako će igrač iskoristiti te elemente i koji način rješavanja problema će izmisliti najbolje je ostaviti samom igraću.

Jednostavan primjer primjene tog principa je razina iz jedne prethodne puzzle platformer igre na kojoj sam radio, pod imenom Blank.



Slika 6: Igra Blank

U ovoj razini (prikazanoj u slici iznad) igraču je jasno da mora doći od lijeve do desne strane razine, te da može aktivirati maksimalno četiri platforme kako bi si omogućio prelazak razine, ali na njemu samom je da pronađe odgovarajući put. Dakle, igraču je jasan cilj, ali ne i način na koji se on postiže.

3.6. Dodatni izazovi

Svaku igru igrati će igrači vrlo različitih vještina. Igra namijenjena za širu publiku ne smije biti pre teška jer bi se tako mnogim igračima onemogućio prolazak kroz cijelu igru. Međutim, poželjno je da igra pruži dodatni izazov igračima kojima sam prelazak igre nije dovoljan izazov.[6]

Primjer pružanja takvog opcionalnog izazova igračima možemo vidjeti u igri Celeste. Ukoliko igrač želi prijeći samo glavnu priču igre, za to će mu trebati svega nekoliko sati i većini igrača to neće biti prevelik izazov. Međutim, glavna priča samo je dio onoga što igra nudi.[6]

Igrač si može zadati cilj sakupljanja jagoda, čije sakupljanje nije obavezno, a jagode se uglavnom na mjestima do kojih je teško doći jer je za potrebna velika vještina. To omogućuje da svaki igrač prilagodi težinu svojoj razini umijeća. Primjer jedne razine sa jagodom koju igrač može skupiti možemo vidjeti na slici ispod.[6]

Slika 7: Igra Celeste [9]

Vidimo da je jagoda na vrlo nedostupnom mjestu, okružena s opasnim bodljama, dok je put za igrače koji ne žele taj dodatni izazov puno jednostavniji.

4. Izrada igre

Igra koju ću izraditi kao praktični dio ovog rada biti će igra u kojoj je cilj na svakoj razini doći do kraja razine. Igra će koristiti 3D modele za glavnog lika i okolinu, a pogled na razinu će biti sa strane (tzv. „side view“). Igrač će moći koristiti dvije vrste sjekira, plave i crvene. Igrač može baciti sjekiru u neki interaktivni objekt, čime magnetizira taj objekt, ovisno o boji sjekire. Kako bi prešao razinu, morati će prevladati nekoliko zagonetki, a zagonetke se uglavnom rješavaju korištenjem te glavne mehanike.

4.1. Uvođenje asset-a

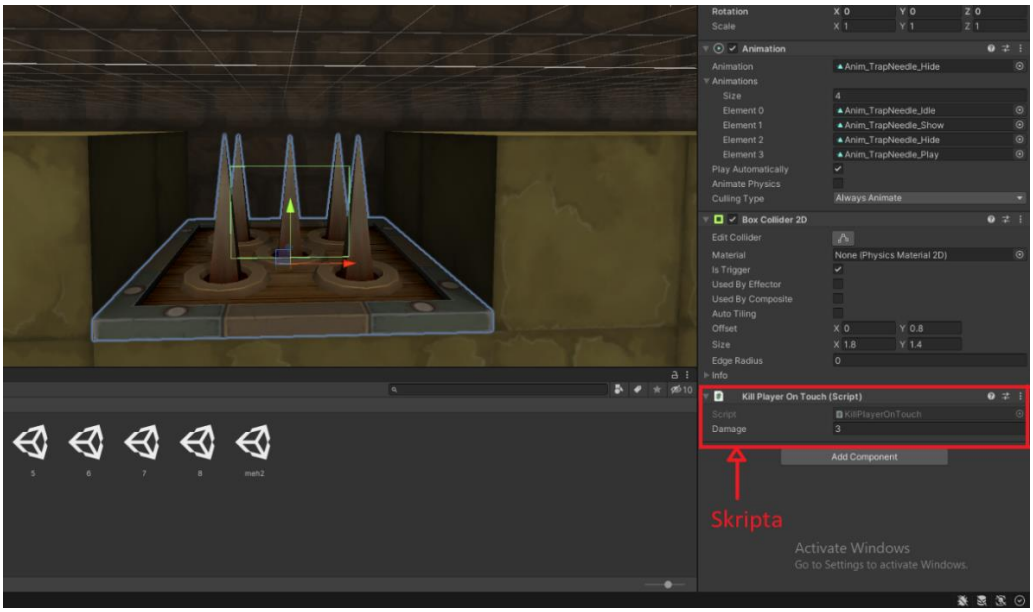
Svaka igra koristi asset-e. Pojam „asset“ (imovina) u svijetu razvoja igara predstavlja svaku vrstu digitalne imovine koju koristimo u svrhu izrade same igre, bilo da je to 3D model, zvučni efekt, 2D slika za korisničko sučelje ili nešto drugo.

Kako bi dodali asset u naš projekt te ga mogli koristiti u našoj igri, potrebno je samo željenu datoteku postaviti unutar direktorija u kojem se nalazi naš projekt, a moguće je i koristiti Unity sučelje tako da se direktno u prozor Projekt povuče željena datoteka.

4.2. Scena

Svaka scena u našoj igri predstavlja jednu razinu. Kako bi napravili novu scenu, potrebno je u Projekt prozoru kliknuti desnu tipku miša, te odavrati opciju new->Scene. U tu scenu zatim je moguće dodavati objekte i na njih „prikvačiti“ željene skripte, te se na taj način ostvaruje željeno ponašanje.

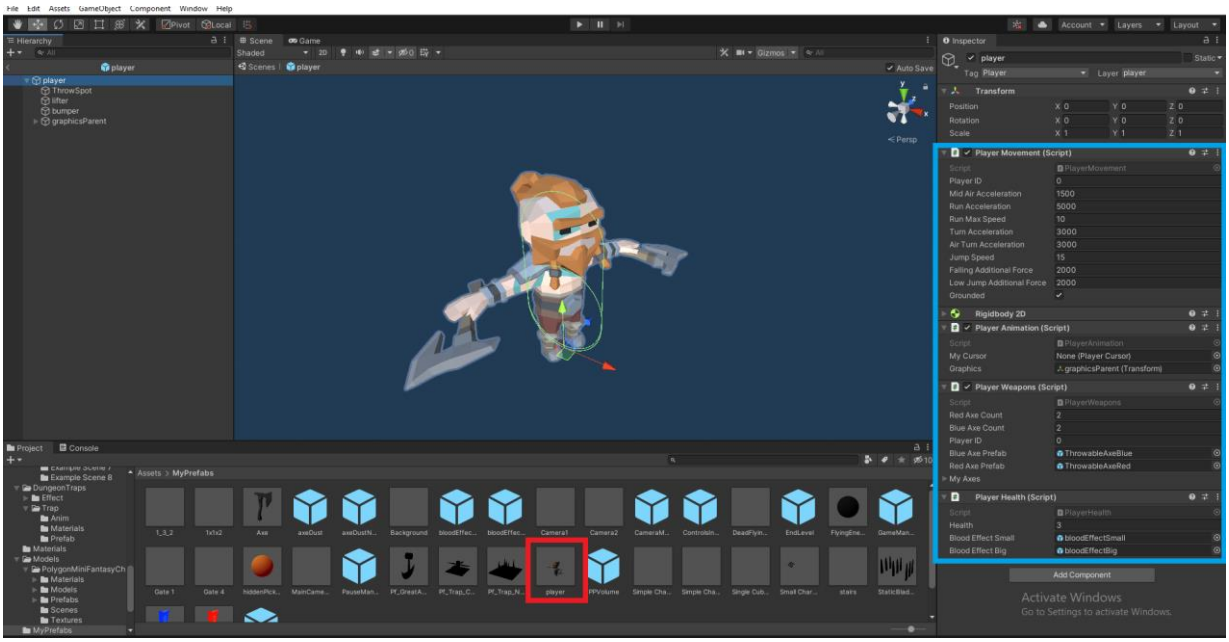
Na primjer, na slici ispod možemo vidjeti kako je na objekt koji treba ozlijediti igrača ukoliko ga on dotakne dodana skripta KillPlayerOnTouch.cs, koja se brine za tu radnju.



Slika 8: Skripte

4.3. Igrač

Model za glavnog lika koji predstavlja igrača preuzet je sa prije spomenutog web mjesta, Unity Asset Store-a. Igrač preuzima vikinga koji ima mogučnost bacanja magnetiziranih sjekira. Objekt igrača izgleda kao na slici dolje:



Slika 9: Igrač

Objekt igrača nalazi se u mapi „MyPrefabs“, u koju sam spremao sve takozvane „Prefab“ objekte. Taj pojam u Unity-u označava tip objekta kojeg je također moguće instancirati putem skripte. Kako bi kreirali takav objekt, potrebno ga je označiti u prozoru Hijerarhija i povući u željenu mapu. Dvostrukim klikom na stvoreni Prefab objekt (na slici gore označeno crvenom bojom) otvaramo taj objekt te ga možemo uređivati.

Prefab objekti su kao svojevrsni nacrti za objekte u igri. Budući da objekt glavnog lika imamo u svakoj sceni u igri, bilo bi dosta teško svaku promjenu na igraču (npr. brzinu kretanja) mijenjati na svakom objektu igrača u našoj igri. To bi zahtijevalo da ulazimo u svaku zasebnu scenu, odaberemo objekt glavnog lika i promijenimo mu željenu vrijednost. Ako u igri imamo 100 razina, jasno je da će takva mala promjena ubrzo postati pravi izazov.

Međutim, kada napravimo Prefab objekt, svaka promjena koju napravimo na njemu odraziti će se na svim instancama tog objekta, dakle kada Prefab objektu promijenimo brzinu kretanja, ta promjena će se propagirati na svaku scenu i igri i zadati će se nova vrijednost. Prefab-ove koristimo za sve objekte koji će se učestalo pojavljivati u igri, kao neprijatelji, zamke, interaktivni predmeti, i slično.

Na slici gore plavom bojom naznačene su skripte koje su dodane na objekt igrača. Skripta „Rigidbody2D“ koja je vidljiva među njima zaslužna je za dodavanje fizike u našu igru, kao što su gravitacija, vektori brzine, sile i slično. Kad god u skriptama želimo utjecati silom na neki objekt, ili mu zadati novi vektor brzine, pristupati ćemo njegovoj Rigidbody2D komponenti.

Također su na slici gore vidljive i skripte PlayerMovement (kretnja igrača), PlayerAnimation (animacija igrača), PlayerWeapons (oružja igrača) te PlayerHealth (zdravlje igrača). Te četiri skripte čine velik dio ove igre, pa ću ih svaku zasebno objasniti.

4.3.1. Kretnja igrača

Pogledajmo sada skriptu koja je zaslužna za ostvarivanje kretnje igrača, PlayerMovement.cs. Na početku skripte inicijalizirane su varijable koje se koriste u ostatku skripte:

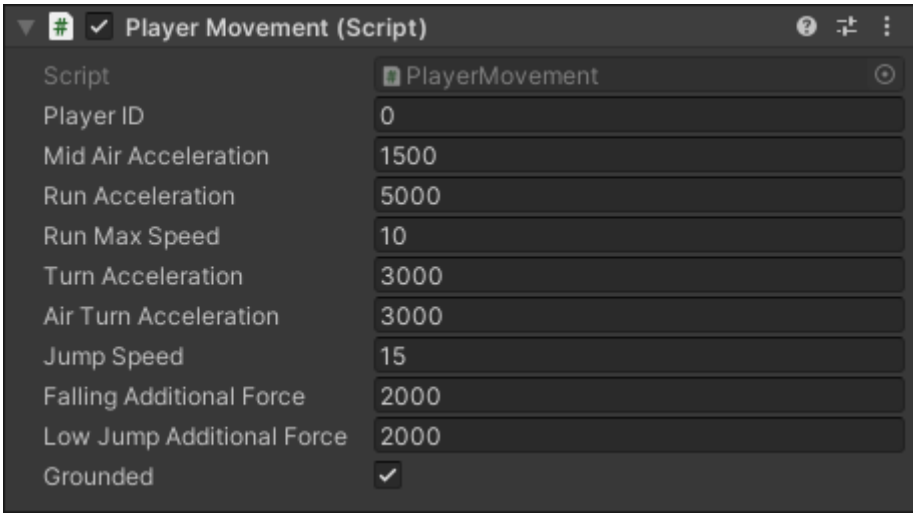
```
public int playerID; // 0-keyboard and mouse, 1-joystick1, 2-joystick2

Rigidbody2D rgb;
public float midAirAcceleration;
public float runAcceleration;
public float runMaxSpeed;
public float turnAcceleration;
public float airTurnAcceleration;
public float jumpSpeed;
```

```
public float fallingAdditionalForce;
public float lowJumpAdditionalForce;
public bool grounded;
// Start is called before the first frame update
void Start()
{
    rgb = GetComponent<Rigidbody2D>();
}
```

Varijabla playerID služi za raspoznavanje igrača, jer je igru moguće igrati u paru s drugom osobom. Varijabla rgb odnosi se na komponentu Rigidbody2D samog igrača, te možemo vidjeti da se vrijednost te varijable inicijalizira u Start() funkciji pomoću metode GetComponent(), koja daje komponentu zadane vrste na objektu. Funkcija Start() je Unity funkcija koja se pokreće automatski čim se objekt, u ovom slučaju igrač, instancira. Varijabla grounded je varijabla tipa boolean koja ima vrijednost „true“ ukoliko je igrač na tlu, a u protivnom ima vrijednost „false“. Ostale varijable predstavljaju parametre za kretnju kao što su maksimalna brzina trčanja, brzina skakanja i slično.

Sve varijable koje su pri deklaraciji označene kao public (npr. public float runMaxSpeed;), ili su označene s [SerializeField] atributom (npr. [SerializeField] float runMaxSpeed;) prikazivati će se i u Inspektor prozoru unutar Unity sučelja, te će se njihova vrijednost ondje moći mijenjati. To je vrlo zgodno za vrijednosti poput brzine trčanja, koje želimo moći brzo mijenjati zbog lakšeg testiranja.



Slika 10: Varijable u prozoru Inspektor

Slično metodi Start(), metoda Update() je još jedna Unity metoda, međutim ona se ne izvodi pri instancijaciji objekte, nego se izvodi prije svaki renderirane sličice u igri (npr. 60 puta u sekundi). Pogledajmo Update() metodu u skripti PlayerMovement.cs:

```
// Update is called once per frame
    void Update()
    {
        //ground check
        grounded = false;
        if (Physics2D.Raycast(transform.position-Vector3.up*0.001f,
Vector2.down, 0.01f)) grounded = true;
        else if (Physics2D.Raycast(transform.position - Vector3.up * 0.001f
+ Vector3.right * 0.2f, Vector2.down, 0.05f)) grounded = true;
        else if (Physics2D.Raycast(transform.position - Vector3.up * 0.001f
- Vector3.right * 0.2f, Vector2.down, 0.05f)) grounded = true;
        else if (Physics2D.Raycast(transform.position + Vector3.up * 0.212f
+ Vector3.right * 0.413f, Vector2.down, 0.2f)) grounded = true;
        else if (Physics2D.Raycast(transform.position + Vector3.up * 0.212f
- Vector3.right * 0.413f, Vector2.down, 0.2f)) grounded = true;
        else if (Physics2D.Raycast(transform.position + Vector3.up * 0.322f
+ Vector3.right * 0.44f, Vector2.down, 0.4f)) grounded = true;
        else if (Physics2D.Raycast(transform.position + Vector3.up * 0.322f
- Vector3.right * 0.44f, Vector2.down, 0.4f)) grounded = true;



        if (grounded)
        {
            //jumping
            if (ControlsInputManager.pressedJump[playerID])
            {
                if(rgb.velocity.y > 0)
                {
                    rgb.velocity += new Vector2(0, jumpSpeed);
                }
                else rgb.velocity = new Vector2(rgb.velocity.x, jumpSpeed);
                ControlsInputManager.pressedJump[playerID] = false;
            }
        }
    }
```

U ovoj metodi varijabla grounded postavlja se u početku na „false“, a zatim se pomoću Unity metode Physics2D.Raycast() iz pozicije stopala igrača ispaljuje sedam virtualnih zraka koje putuju malu udaljenost, te ukoliko išta pogode (na primjer tlo), mijenjaju vrijednost varijable grounded u „true“. Tako je riješena provjera je li igrač na tlu. To nam je potrebno kako bi znali

da li igrač trenutno može skočiti ili ne, da nema te provjere igrač bi mogao skakati i dok je u zraku. Zatim vidimo dio koda koji je zaslužan za skakanje. Provjerava se da li je igrač pritisnuo tipku za skok, te ukoliko je, igračeva brzina se ažurira preko rgb (Rigidbody2D) komponente.

Posljednja metoda u ovoj skripti je FixedUpdate(). To je još jedna od glavnih Unity metoda. Ona je slična Update() metodi, ali za razliku od nje, ne izvršava se prije renderiranja svake sličice, već se izvršava uvijek u jednakim vremenskim razmacima, npr. svakih 50 milisekundi. Zbog toga što se odvija uvijek u jednakim razmacima, često se koristi za upravljanje silama i drugim fizikalnim svojstvima. Pogledajmo implementaciju te metode u skripti za kretnju igrača:

```
private void FixedUpdate()
    {
        if (grounded)
        {
            //fastStop/turn
            if (rgb.velocity.x > 0.7 &&
ControlsInputManager.holdingLeftRight[playerID] != 1)
            {
                rgb.AddForce(turnAcceleration * Vector2.left);
            }
            else if (rgb.velocity.x < -0.7 &&
ControlsInputManager.holdingLeftRight[playerID] != -1)
            {
                rgb.AddForce(turnAcceleration * Vector2.right);
            }
            //running
            else rgb.AddForce(Vector2.right * runAcceleration *
Mathf.Clamp01(1 - Mathf.Abs(rgb.velocity.x) / runMaxSpeed) *
ControlsInputManager.holdingLeftRight[playerID]);


        }
        else
        {
            //mid air movement
            if (rgb.velocity.x > 0.7 &&
ControlsInputManager.holdingLeftRight[playerID] == -1)
            {
                rgb.AddForce(airTurnAcceleration * Vector2.left);
            }
            else if (rgb.velocity.x < -0.7 &&
ControlsInputManager.holdingLeftRight[playerID] == 1)
            {
```

```
                rgb.AddForce(airTurnAcceleration * Vector2.right);
            }
            else rgb.AddForce(Vector2.right * midAirAcceleration *
Mathf.Clamp01(1 - Mathf.Abs(rgb.velocity.x) / runMaxSpeed) *
ControlsInputManager.holdingLeftRight[playerID]);


            //faster falling
            if (rgb.velocity.y < 0) rgb.AddForce(Vector2.down *
fallingAdditionalForce);
            if (rgb.velocity.y > 0 &&
!ControlsInputManager.holdingJump[playerID]) rgb.AddForce(Vector2.down *
lowJumpAdditionalForce);
            //
        }
    }
```

Ova funkcija provjerava korisnikove unose, te na temelju njih djeluje na objekt igrača pripadajućim silama koristeći rgb (Rigidbody2D) komponentu. Klasa koju možemo vidjeti da se gore koristi za provjeru korisnikovih unosa (eng. *input*) zove se ControlsInputManager. To je klasa koju sam razvio kako bih uveo razinu apstrakcije u korisničke input-e, te tako omogućio lako dodavanje podrške za igranje pomoću kontrolera.

Neću objašnjavati tu cijelu skriptu jer to nije fokus ovog rada, ali sve što o njoj valja znati je da ona sadrži niz varijabla koje su tipa jednodimenzionalnog polja, te uređuje te varijable u Update() funkciji na temelju korisnikovih pritisaka tipki na tipkovnici ili gumba na kontroleru, na primjer ovako:

```
holdingJump[0] = Input.GetButton("Jump0") || Input.GetKey(KeyCode.UpArrow);
holdingJump[1] = Input.GetButton("Jump1");
holdingJump[2] = Input.GetButton("Jump2");
```

Tako se stanje inputa svakog igrača (da li drži tipku za skok ili ne) sprema u polje holdingJump[], te tako više igraća može u isto vrijeme imati različito stanje inputa, ovisno o njihovom identifikacijskom broju, koji može biti 0, 1 ili 2. Svaki igrač koristi svoj playerID (identifikacijski broj igrača) kako bi pristupio vlastitim vrijednostima.

4.3.2. Animiranje igrača

Pogledajmo sada skriptu koja je zaslužna za ostvarivanje animacije igrača, PlayerAnimation.cs. Na početku su deklarirane varijable koje se koriste u ostatku skripte:

```
    public PlayerCursor myCursor;
    Rigidbody2D rgb;
    Animator anim;
    [SerializeField] Transform graphics;
    // Start is called before the first frame update
    void Start()
    {
        rgb = GetComponent<Rigidbody2D>();
        anim =
transform.Find("graphicsParent").Find("graphics").GetComponent<Animator>();
    }
```

Također je prikazana metoda Start(), koja inicijalizira vrijednost varijabli rgb i anim, koje predstavljaju komponente tipa Rigidbody2D i Animator, respektivno. O komponenti Rigidbody2D smo već pričali, a komponenta Animator služi nam, kako i samo ime govori, za animiranje objekata u video igri. Pogledajmo metodu Update() u ovoj skripti.

```
// Update is called once per frame
void Update()
    {
        if (myCursor != null)
        {
            if (myCursor.aimingDir.x > 0)
               transform.localScale = new Vector3(1, 1, 1);
            if (myCursor.aimingDir.x < 0)
               transform.localScale = new Vector3(-1, 1, 1);

            if (Mathf.Abs(rgb.velocity.x) > 0.15f)
            {
                //running forward
                if ((rgb.velocity.x * myCursor.aimingDir.x) > 0)
                {
                    anim.SetInteger("relativeMovementDir", 1);
                    graphics.localEulerAngles = new
Vector3(graphics.localEulerAngles.x, graphics.localEulerAngles.y,
Mathf.LerpAngle(graphics.localEulerAngles.z, -8, Time.deltaTime * 5f));
                }
```

```
                //running backward
                else
                {
                    anim.SetInteger("relativeMovementDir", -1);
                    graphics.localEulerAngles = new
Vector3(graphics.localEulerAngles.x, graphics.localEulerAngles.y,
Mathf.LerpAngle(graphics.localEulerAngles.z, 16, Time.deltaTime * 5f));
                }
            }
            else
            {
                anim.SetInteger("relativeMovementDir", 0);
                graphics.localEulerAngles = new
Vector3(graphics.localEulerAngles.x, graphics.localEulerAngles.y,
Mathf.LerpAngle(graphics.localEulerAngles.z, 0, Time.deltaTime * 5f));
            }
        }
    }
```

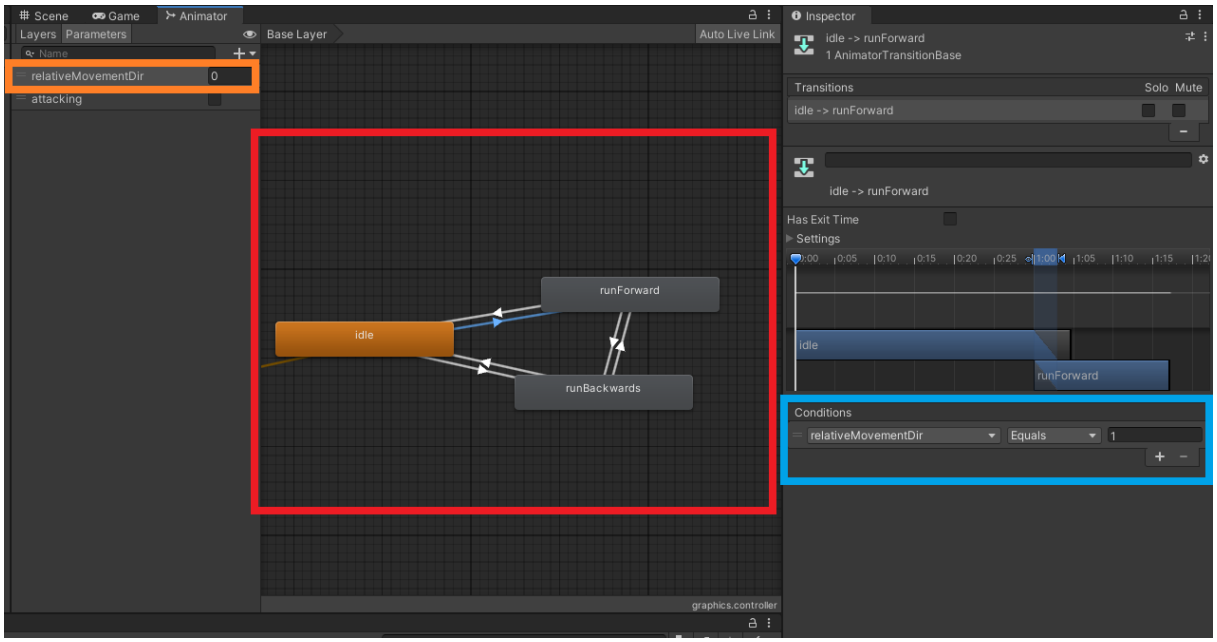
Ova skripta koristi komponentu Rigidbody2D kako bi dobila podatke o trenutnoj brzini igrača, te skriptu PlayerCursor za pristup poziciji kursora. Ovisno o tome nalazi li se kursor lijevo ili desno od igrača, objektu igrača se po potrebi skalira veličina po x osi sa -1, što daje privid da se igrač zaokrenuo za 180 stupnjeva.

Zatim skripta provjerava u kojem smjeru se kreće igrač, te ovisno o tome pušta animaciju za trčanje unaprijed ili unatrag. To radi tako da komponenti Animator postavi vrijednost parametra „relativeMovementDir“, a Animator preuzima odgovornost puštanja pripadajuće animacije. Više o tome kako on to radi kasnije. Zatim ova skripta naginje igrača u smjeru u kojem trči, što daje dodatni efekt težine i momenta, kako je vidljivo na slici ispod.

Slika 11: Igrač trči

4.3.3. Animator komponenta

Pogledajmo sada kako Animator komponenta koristi parametar „relativeMovementDir“ kojeg joj je pružila skripta PlayerAnimation.cs.



Slika 12: Animator

Kako je prikazano na slici gore, u Unity alatu moguće je uređivati animator komponentu koristeći Animator prozor. S lijeve strane, na slici iznad naznačeno narančastom bojom, prikazana je vrijednost parametra kojeg prosljeđuje PlayerAnimation skripta.

U prozor naznačen na slici gore crvenom bojom postavljaju se stanja, a svako stanje predstavlja jednu animaciju. Na primjer, tu vidimo animaciju za trčanje naprijed, trčanje unatrag i animaciju koja se izvršava dok igrač miruje.

Stanja međusobnu povezujemo prijelazima (eng. transitions). Na slici gore vidimo da je odabran prijelaz od stanja „idle“ (animacija stajanja) i stanja „runForward“ (animacija trčanja naprijed). U dijelu slike označenom plavom bojom, unutar prozora Inspektor, postavljamo za svaki prijelaz uvjet pod kojim se on izvršava. Tako vidimo da se odabrani prijelaz treba izvršiti kada je parametar relativeMovementDir jednak 1.

Kada smo s Unity Asset Store-a preuzeli potrebne animacije, te uspješno implementirali skriptu PlayerAnimation i samu Animator komponentu, igrač se uistinu animira ovisno o njegovoj kretnji.

4.3.4. Zdravlje igrača

Sljedeća skripta koju ćemo proučiti biti će skripta koja je zaslužna za ostvarivanje funkcionalnosti zdravlja glavnog lika, PlayerHealth.cs. Sve skripte (klase) koje su zaslužne za ostvarivanje funkcionalnosti zdravlja, bilo glavnog lika, neprijatelja ili objekta, nasljeđuju od klase Health.

```
public abstract class Health : MonoBehaviour
{
    [SerializeField] float health = 3;

    public virtual void TakeDamage(float damage)
    {
        health -= damage;
        if (health <= 0)
        {
            Die();
        }
    }

    public virtual void Die()
    {
```

```
    }
}
```

To je abstraktna klasa koja deklarira varijablu health, koja predstavlja zdravlje, te implementira dvije virtualne metode, TakeDamage() i Die(). Metoda TakeDamage() oduzima željenu vrijednost od varijable health, te ukoliko je nova vrijednost varijable health manja ili jednaka 0, poziva funkciju Die(), koja predstavlja umiranje lika u igri. Funkcija Die() je ovdje prazna, ali budući da je definirana kao virtualna metoda, klase djeca (klase koje nasljeđuju od klase Health) mogu je implementirati.

Ovdje je prikazan početak skripte PlayerHealth.cs:

```
public class PlayerHealth : Health
{
    [SerializeField] GameObject bloodEffectSmall;
    [SerializeField] GameObject bloodEffectBig;

    public override void TakeDamage(float damage)
    {
        base.TakeDamage(damage);
        StartCoroutine(PlayerDamaged());
    }
    public override void Die()
    {
        base.Die();
        StartCoroutine(PlayerDeath());
    }
```

Dakle, skripta PlayerHealth nasljeđuje od skripte Health, te implementira (nadjačava) dvije njezine metode, TakeDamage() i Die(). I jedna i druga metoda prvo izvršavaju funkcionalnost u roditeljskoj klasi, a zatim pokreću odgovarajuću korutinu (eng. coroutine) pozivom naredbe StartCoroutine(). Korutine su vrsta metoda koje se ne moraju izvršiti u trenutku kada su pozvane, nego se mogu izvršavati kroz duži period vremena. To nam omogućuje da unutar njih koristimo vrlo korisne metode WaitForSeconds() i WaitForSecondsRealtime(), koje zaustavljaju rad metode na određeno vrijeme. To nam omogućuje na primjer da kod smrti igrača prvo izvršimo potrebne efekte umiranja, a tek nakon nekog vremena resetiramo razinu. Pogledajmo kako je to izvedeno:

```
IEnumerator PlayerDamaged()
```

```
    {
        Instantiate(bloodEffectSmall, transform.position + Vector3.up *
0.7f, Quaternion.identity);
        yield return null;
    }
IEnumerator PlayerDeath()
    {
        Instantiate(bloodEffectBig, transform.position + Vector3.up * 0.7f,
Quaternion.identity);
        foreach (MonoBehaviour x in
transform.root.GetComponentsInChildren<MonoBehaviour>())
        {
            if (x != this)
            {
                x.enabled = false;
            }
        }
        Rigidbody2D rgb =
transform.root.GetComponentInChildren<Rigidbody2D>();
        rgb.velocity = Vector2.up * Random.Range(11f, 14) + Vector2.right *
Random.Range(-5, 5);
        int rotDir = Random.Range(0,2);
        rgb.freezeRotation = false;
        if(rotDir == 0)rgb.angularVelocity = Random.Range(1000,1300);
        if (rotDir == 1) rgb.angularVelocity = -Random.Range(1000, 1300);

        float timer = 0.5f;
        while (timer > 0)
        {
            Time.timeScale -= 0.7f * Time.deltaTime * 2;
            Time.timeScale = Mathf.Clamp(Time.timeScale, 0.3f, 1f);
            timer -= Time.deltaTime;
            yield return null;
        }
        Time.timeScale = 0.3f;

        yield return new WaitForSecondsRealtime(2);
        Time.timeScale = 1;
        Application.LoadLevel(Application.loadedLevel);
        yield return null;
    }
```

U isječku koda gore vidi se da su korutine metode tipa IEnumerator. U korutini PlayerDamaged() samo se instancira efekt krvi, koji signalizira igraču da je ozlijeđen.

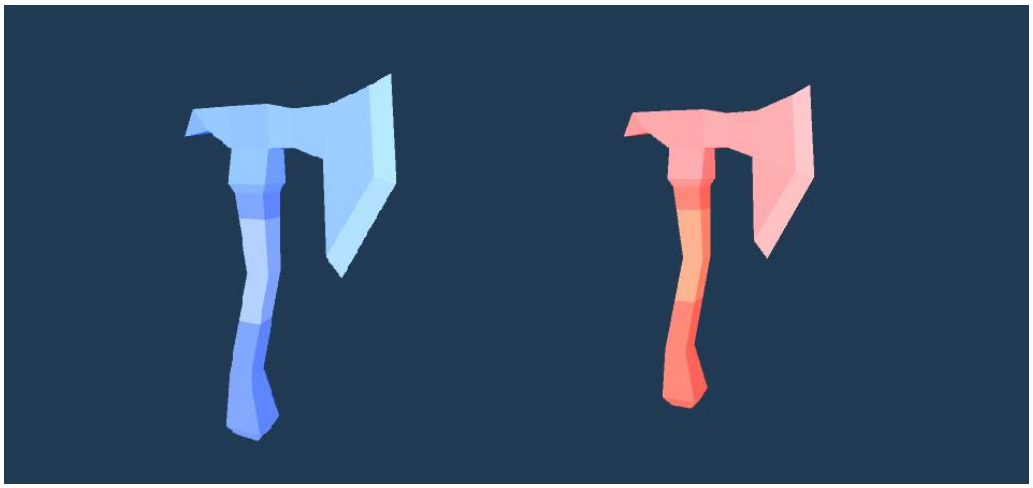
U PlayerDeath() korutini također se instancira efekt krvi, a zatim se isključuju sve skripte na objektu igrača (kako se više ne bi mogao kretati), te se objektu igrača pridodaje nasumična brzina. Također, vrijeme se naredbom `Time.timeScale = 0.3f;` postavlja na 0.3 puta svoje uobičajene brzine, što daje efekt usporene snimke. Zatim se pomoću metode čeka dvije sekunde, nakon čega se ponovno učitava trenutna razina.

4.3.5. Igračeve sjekire

Sljedeća skripta koju ćemo proučiti biti će skripta koja je zaslužna za mogućnost glavnog lika da baci jednu od magnetiziranih sjekira, PlayerWeapons.cs. Pogledajmo deklaraciju varijabli i Start() metodu te skripte:

```
    public int redAxeCount = 2;
    public int blueAxeCount = 2;
    bool currentlyThrowing = false;
    Animator anim;
    public int playerID;
    [SerializeField] GameObject blueAxePrefab;
    [SerializeField] GameObject redAxePrefab;
    public List<Axe> myAxes = new List<Axe>();
    // Start is called before the first frame update
    void Start()
    {
        anim =
transform.Find("graphicsParent").Find("graphics").GetComponent<Animator>();
    }
```

Varijable redAxeCount i blueAxeCount označavaju koliko sjekira svake boje igrač ima na raspolaganju. Varijable blueAxePrefab i redAxePrefab predstavljaju Prefab objekte igračevih sjekira za bacanje.



Slika 13: Sjekire

Sagledajmo sada metodu Update().

```
    void Update()
    {
        playerID = GetComponent<PlayerMovement>().playerID;
        if (ControlsInputManager.pressedLeftAttack[playerID])
        {
            if (!currentlyThrowing && blueAxeCount > 0)
            {
                StartCoroutine(ThrowAxe(blueAxePrefab));
                blueAxeCount--;
            }
        }
        if (ControlsInputManager.pressedRightAttack[playerID])
        {
            if (!currentlyThrowing && redAxeCount > 0)
            {
                StartCoroutine(ThrowAxe(redAxePrefab));
                redAxeCount--;
            }
        }

        if (ControlsInputManager.pressedReload[playerID]) // returns every
axe to player
        {
            foreach(Axe axe in myAxes)
            {
                axe.Reload();
```

```
            }
        }
    }
```

Provjerava se pritišće li igrač tipku za bacanje sjekira jedne ili druge vrste, te ukoliko pritišće i ima još sjekira te boje, poziva korutinu ThrowAxe(). Ako igrač pritisne botun za „reload“, vračaju mu se sve sjekire koje je bacio tako što se na svakoj od njih poziva metoda Reload(). Više o tome kasnije. Pogledajmo sada implementaciju korutine ThrowAxe() :

```
    IEnumerator ThrowAxe(GameObject axe)
    {
        currentlyThrowing = true;
        anim.SetBool("attacking", true);
        yield return new WaitForSeconds(0.22f);
        PlayerCursor cursor = GetComponent<PlayerAnimation>().myCursor;
        GameObject newAxe = Instantiate(axe,
transform.Find("ThrowSpot").position, Quaternion.identity);
        newAxe.transform.localScale = transform.localScale;
        newAxe.GetComponent<Rigidbody2D>().velocity = cursor.aimingDir *
18;
        newAxe.GetComponent<Axe>().player = gameObject;
        myAxes.Add(newAxe.GetComponent<Axe>());
        yield return new WaitForSeconds(0.07f);
        anim.SetBool("attacking", false);
        currentlyThrowing = false;
        yield return null;
    }
```

Ova korutina izvršava animaciju bacanja sjekire, stvara novu sjekiru ispred objekta igrača, te joj zadaje novu brzinu prema smjeru kursora.

Opisao sam sve najbitnije skripte koje daju ponašanje objektu igrača, a sada ću predstaviti neke druge važne skripte koje čine ovu igru.

4.4. Sjekire

Svaka sjekira koja se stvori kada ju igrač baci na sebi nosi skriptu Axe.cs. Ovako izgleda njezina Update() metoda:

```
void Update()
```

```
    {
        if(myChargeable !=null)Debug.Log(myChargeable.name);
        if(!stuck)transform.Rotate(Vector3.forward,rotDir*(-3240) *
transform.localScale.x*Time.deltaTime);


        float distanceToStart = Vector3.Distance(startPos,
transform.position);
        float distanceToPlayer =
Vector3.Distance(player.transform.position, transform.position);
        if (distanceToStart > range && !stuck && !currentlyReloading)
        {
            Reload();
        }
        if (distanceToPlayer > range && !stuck && !currentlyReloading)
        {
            Reload();
        }
        if (distanceToPlayer > rangeWhenStuck && stuck &&
!currentlyReloading)
        {
            Reload();
        }
    }
```

Ukoliko sjekira nije već zaglavljena u neki objekt, ona se rotira da se dobije efekt prave sjekira koja izgleda kao da ju je netko bacio. Zatim, provjerava se je li sjekira otišla pre daleko od igrača, te ako je, poziva se metoda Reload(), koju sam već u prijašnjem poglavlju spomenuo.

```
public void Reload()
    {
        if (!currentlyReloading)
        {
            StartCoroutine(ReturnToPlayer());
        }
    }
```

Metoda Reload() samo poziva korutinu ReturnToPlayer(), koja vraća sjekiru igraču. Ovo je implementacija te korutine:

```
    IEnumerator ReturnToPlayer()
    {
        if (!currentlyReloading)
```

```
        {
            currentlyReloading = true;
            if (myChargeable != null) myChargeable.currentSign = 0;
            rotDir = -1;
            stuck = false;
            transform.parent = null;
            Rigidbody2D rgb = GetComponent<Rigidbody2D>();
            GetComponent<Collider2D>().enabled = false;
            rgb.gravityScale = 0;

            while (true)
            {
                Vector3 destination = player.transform.position +
Vector3.up * 0.7f;
                rgb.velocity = ((destination -
transform.position).normalized * 20);
                if ((transform.position - destination).magnitude < 0.5f)
break;
                yield return null;
            }

            if (charge == -1)
player.GetComponent<PlayerWeapons>().redAxeCount++;
            if (charge == 1)
player.GetComponent<PlayerWeapons>().blueAxeCount++;
            player.GetComponent<PlayerWeapons>().myAxes.Remove(this);
            currentlyReloading = false;
            Destroy(gameObject);
        }

        yield return null;
    }
```

Ukoliko je sjekira zabijena u neki objekt (ako referenca myChargeable nije jednaka null vrijednosti) ona se otkači od tog objekta, te postavlja njegov naboj na nula. Zatim se pomoću while petlje sjekiri daje brzina u smjeru igrača kako bi mu se vratila. Kada dođe do igrača, u igračevoj skripti PlayerWeapons povećava se raspoloživi broj sjekira takvog naboja, te sjekira nestaje.

Sada ću predstaviti jedan od najbitnijih dijelova koda u ovoj igri, koji je zaslužan za magnetiziranje objekata.

```
private void OnCollisionEnter2D(Collision2D other)
    {
        if(myChargeable == null && !hitSomething)
        {
            hitSomething = true;
            if (other.transform.GetComponent<Chargeable>() != null)
            {
                Instantiate(hitEffectSmall, other.contacts[0].point,
Quaternion.identity);
                stuck = true;
                Chargeable thing =
other.transform.GetComponent<Chargeable>();
                thing.currentSign = charge;
                GetComponent<Rigidbody2D>().velocity = new Vector3(0, 0,
0);
                GetComponent<Rigidbody2D>().angularVelocity = 0;
                GetComponent<Collider2D>().enabled = false;
                GetComponent<Rigidbody2D>().isKinematic = true;
                if (thing.axeGoesOnRoot) transform.parent =
thing.transform.root;
                else transform.parent = thing.transform;
                myChargeable = thing;
                thing.RegisterAxe(this);
            }
            else
            {
                Instantiate(hitEffect, other.contacts[0].point,
Quaternion.identity);
                if (!currentlyReloading)
                {
                    StartCoroutine(ReturnToPlayer());
                }

            }
        }
    }
```

Metoda OnCollisionEnter2D() je Unity metoda koja se pokreće kada se objekt na kojem je skripta dodirne s nekim drugim objektom koji ima fizikalne granice (eng. collider).

Prvo se provjerava ima li taj drugi objekt komponentu tipa Chargeable, što naznačuje da se objekt može magnetizirati. Ukoliko nema, pokreće se korutina ReturnToPlayer(), koja vraća sjekiru igraću. Ukoliko ima, sjekira se prikvači na taj objekt, te se naboj tog objekta postavi na pozitivan (plavo) ili negativan (crveno) naboj. Objekti istoznačnog naboja se odbijaju, a oni različitog predznaka se međusobno privlače.

Slika 14: Magnetizirani objekt

4.5. Magnetiziranje objekata

Svakom objektu koji može pomoću sjekira biti magnetiziran mora biti pridružena skripta koja nasljeđuje klasu Chargeable. Razlog tomu je što želimo imati jedinstvenu apstrakciju, ali i dalje želimo imati mogučnost da se različiti objekti drugačije ponašaju ukoliko su magnetizirani.

Ispod je prikazan početni dio koda klase Chargeable.cs:

```
public int currentSign;
public int charge;
public int chargeStreght;
public float range;
protected Rigidbody2D rgb;
protected MeshRenderer[] mr;
public static event Action<Chargeable> OnMagneticPulse;
```

```
    [SerializeField] Material noChargeMaterial;
    [SerializeField] Material negativeChargeMaterial;
    [SerializeField] Material positiveChargeMaterial;
    public bool axeGoesOnRoot = true;
    Axe myAxe;
```

Na početku skripte zadane su varijable koje se u ostatku skripte koriste. Varijabla currentSign prima vrijednost 0, 1 ili -1, ovisno je li i kojom sjekirom je magnetiziran objekt. Varijabla chargeStreght može biti drugačija za svaki objekt, a predstavlja jačinu magnetskog polja predmeta, ukoliko je on magnetiziran, dok varijabla charge predstavlja currentSign * chargeStreght. Varijabla range govori koliko daleko djeluje magnetno polje objekta.

Bitno je obratiti pozornost na sljedeću liniju koda:

```
public static event Action<Chargeable> OnMagneticPulse;
```

Ovdje je deklariran događaj (eng. event) OnMagneticPulse. To je događaj koji će slušati sve „magnetizirane“ instance klase Chargeable, te će ga svaka od njih i odašiljati. To ću uskoro malo bolje objasniti.

Varijable označene s [SerializeField] su materijali objekta za određena stanja, kako bi mogli odrediti kako će objekt vizualno izgledati kada je magnetiziran. Pogledajmo dalje kod ove klase:

```
    private void Start()
    {
        DoStart();
    }
    public virtual void DoStart()
    {
        mr = transform.root.GetComponentsInChildren<MeshRenderer>();
        rgb = GetComponentInChildren<Rigidbody2D>();
        OnMagneticPulse += ProcessMagneticPulse;
    }
    private void OnDestroy()
    {
        OnMagneticPulse -= ProcessMagneticPulse;
        if (myAxe != null)
        {
            myAxe.transform.parent = null;
```

```
            myAxe.gameObject.SetActive(true);
            myAxe.Reload();
        }
    }
```

Gore su prikazane funkcije Start(), DoStart() i OnDestroy() klase Chargeable. Metoda Start() jednostavno poziva metodu DoStart(). To se možda na prvi pogled čini suvišno, međutim korisno je kako bi postigli da druge klase mogu nadjačati funkcionalnost koja se izvršava pri instanciranju objekta.

Metoda DoStart() postavlja neke osnovne reference, te događaju OnMagneticPulse dodaje metodu ProcessMagneticPulse(). Metoda OnDestroy() poziva se kada će se instanca objekta uništiti, te se u njoj metoda ProcessMagneticPulse() te instance miče iz slušača OnMagneticPulse eventa.

Dolje je prikazana implementacija funkcija FixedUpdate() i ProcessMagneticPulse():

```
    void FixedUpdate()
    {
        charge = currentSign * chargeStreght;
        if(charge != 0)
        {
            OnMagneticPulse(this);
        }

        if (charge == 0)
        {
            SetMaterial(noChargeMaterial);
        }
        else if (charge < 0)
        {
            SetMaterial(negativeChargeMaterial);
        }
        else if (charge > 0)
        {
            SetMaterial(positiveChargeMaterial);
        }
    }

    protected void ProcessMagneticPulse(Chargeable other)
    {
        if (charge != 0)
```

```
        {
            if (other != this)
            {
                float distance = (other.transform.position -
transform.position).magnitude;
                if(distance < range || distance < other.range)
                {
                    Vector2 dir = other.transform.position -
transform.position;
                    rgb.AddForce(dir.normalized * -charge * other.charge /
dir.magnitude);
                }
            }
        }
    }
```

Funkcija FixedUpdate() u pravilnim intervalima (na primjer 30 puta u sekundi) pokreće događaj OnMagneticPulse(), čime djeluje na druge magnetizirane objekte u svom dometu. Također se postavlja odgovarajući materijal objektu, ovisno o njegovoj magnetiziranosti.

U metodi ProcessMagneticPulse() sadržana je glavna mehanika ove igre. Provjerava se magnetni naboj i predznak tijela, te tijela koje je na njega djelovalo, te se primjenjuju odgovarajuće odbojne ili privlačne sile.

Kada su implementirane spomenute skripte, moguće je prelaziti razine korištenjem mehanike magnetiziranja objekta.

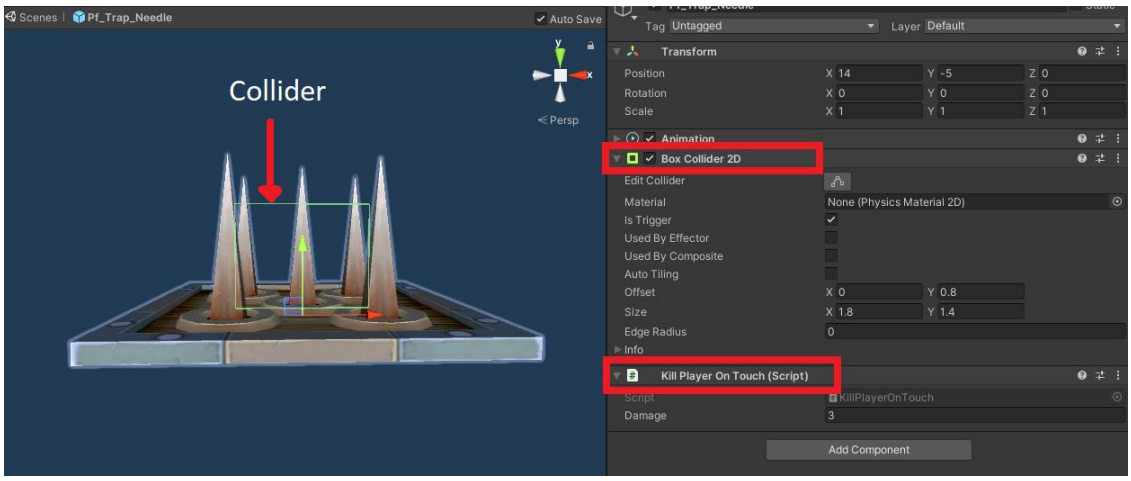
Slika 15: Privlačne sile

4.6. Prepreke

Kako bi igrač mogao u igri izgubiti (umrijeti), implementirao sam objekte koji rade štetu igraču ako ga dotaknu. Pogledajmo primjer jednog takvog objekta:

Slika 16: Prepreka

Ovaj objekt je animiran tako da mu se bodlje spuštaju i uzdižu. U trenutku kada se uzdignu, aktivira se nevidljiva BoxCollider2D komponenta na objektu, a taj objekt na sebi ima i skriptu za ozljeđivanje igrača ukoliko dotakne igrač taj collider.



Slika 17: KillPlayerOnTouch i BoxCollider2D

Pogledajmo kako je implementirana skripta KillPlayerOnTouch.cs:

```
public class KillPlayerOnTouch : MonoBehaviour
{
    [SerializeField] float damage = 3;
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.transform.tag == "Player")
collision.transform.root.GetComponentInChildren<PlayerHealth>().TakeDamage(
damage);
    }
    private void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.transform.tag == "Player")
collision.transform.root.GetComponentInChildren<PlayerHealth>().TakeDamage(
damage);
    }
}
```

Funkcije OnTriggerEnter2D() i OnCollisionEnter2D() su Unity funkcije koje se okidaju kada nešto dotakne objekt na kojem je skripta. U ovom slučaju provjerava se je li taj objekt igrač, i ako je, dohvaća se njegova PlayerHealth komponenta te se na njoj poziva metoda TakeDamage(), koja ga ozljeđuje.

Slika 18: Igrač - smrt

4.7. Neprijatelji

Još jedna prepreka igraču su neprijatelji, koji mogu letjeti i ozljeđuju igrača pri dodiru. U igri su prikazani kao leteće svijetleće kugle.

Slika 19: Neprijatelj

Svakom neprijatelju pridružena je skripta ChargeableSimple.cs, koja nasljeđuje od klase Chargeable. To znači da igrač može magnetizirati neprijatelja, te tako izbjeći prijetnju.

Svakom neprijatelju je također pridružena FlyingEnemyMovement skripta, koja, kako ime govori, omogućuje kretanje letećih neprijatelja. Ovako je implementirana:

```
EnemyMovementState state;
Transform target;
Vector2 targetPos;
Vector2 startPos;
public float range;
public float stateCheckInterval;
public float flapInterval;
public float flapForceVertical;
public float flapForceHorizontal;
```

```
    public float HorizontalMoveRange;
    public float VerticalMoveRange;
    Rigidbody2D rgb;
    // Start is called before the first frame update
    void Start()
    {
        startPos = transform.position;
        rgb = GetComponent<Rigidbody2D>();
        StartCoroutine(DetermineState());
        StartCoroutine(Flapping());
    }
```

U početku se deklariraju varijable, te metoda Start(). U metodi Start() pokreću se dvije korutine, DetermineState() i Flapping(). Prikazane su ovdje:

```
IEnumerator DetermineState()
    {
        while (true)
        {
            float minDistance = 1000;
            state = EnemyMovementState.RandomMovement;
            foreach (PlayerMovement x in
FindObjectsOfType<PlayerMovement>())
            {
                float distance = Vector2.Distance(x.transform.position,
transform.position);
                if (distance < range && distance < minDistance)
                {
                    target = x.transform;
                    state = EnemyMovementState.FollowingPlayer;
                    minDistance = distance;
                }
            }
            if(state == EnemyMovementState.RandomMovement)
            {
                targetPos = startPos + new Vector2(Random.Range(-
HorizontalMoveRange,HorizontalMoveRange), Random.Range(-VerticalMoveRange,
VerticalMoveRange));
            }
            yield return new
WaitForSeconds(stateCheckInterval*Random.Range(0.95f,1.05f));
        }
    }
```

```
    IEnumerator Flapping()
    {
        while (true)
        {
            Flap();
            yield return new WaitForSeconds(flapInterval *
Random.Range(0.95f, 1.05f));
        }
    }
```

Korutina DetermineState() otprilike jednom po sekundi provjerava u kojem stanju neprijatelj treba biti, odnosno je li igrač dovoljno blizu da ga počne hvatati, ili da nastavi lutati. Korutina Flapping() otprilike jednom u sekundi poziva funkciju Flap().

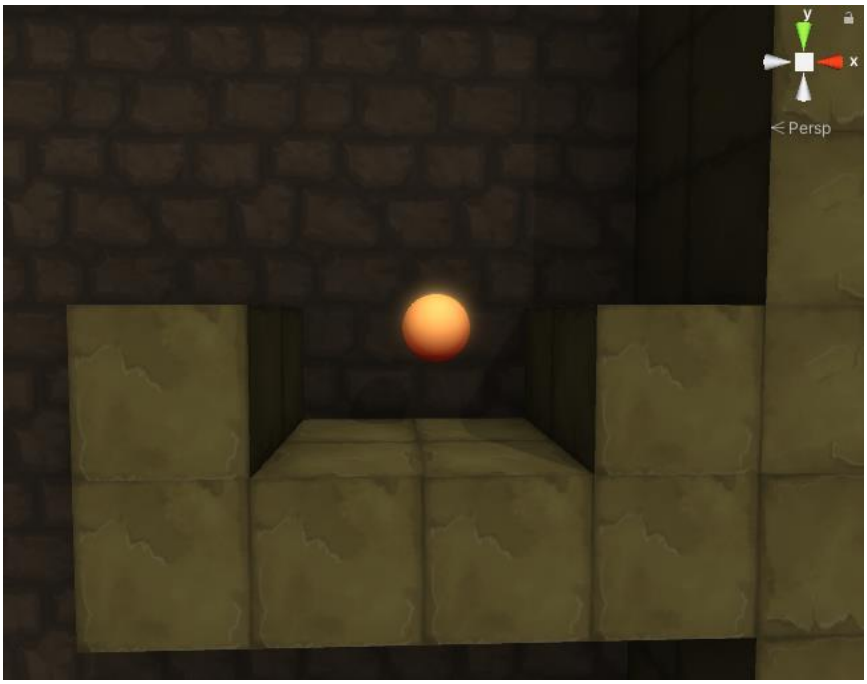
```
    void Flap()
    {
        float HorForce = flapForceHorizontal;
        if (state == EnemyMovementState.FollowingPlayer)
        {
            targetPos = target.position + Vector3.up;
            HorForce *= 2;
        }
        if (Mathf.Sign(rgb.velocity.x) != Mathf.Sign(targetPos.x -
transform.position.x)) HorForce *= 2;

        //target below
        if (targetPos.y < transform.position.y)
        {
            rgb.AddForce(new Vector2(HorForce * Mathf.Sign(targetPos.x -
transform.position.x), flapForceVertical * 0.5f));
        }
        //target above
        else
        {
            rgb.AddForce(new Vector2(HorForce * Mathf.Sign(targetPos.x -
transform.position.x), flapForceVertical));
        }
    }
```

Funkcija Flap() dodaje objektu letećeg neprijatelja brzinu u odgovarajućem smjeru, ovisno o poziciji igrača ili prema nasumično odabranoj poziciji, ukoliko neprijatelj ne vidi igrača.

4.8. Skriveno blago

Na svakoj razini postoji jedno skriveno blago, koje izgleda kao na slici ispod. Kada ga igrač dotakne, odigra se efekt sakupljanja i objekt nestaje.



Slika 20: Skriveno blago

Na objektu se nalazi skripta PickUp():

```
public class PickUp : MonoBehaviour
{
    [SerializeField] GameObject pickUpEffect;
    private void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.transform.tag == "Player")
        {
            Instantiate(pickUpEffect, transform.position,
transform.rotation);
            Destroy(gameObject);
        }
```

```
    }
}
```

Metoda OnCollisionEnter2D() okida se kada neki drugi objekt dotakne objekt na kojem je ova skripta. Ukoliko je taj objekt uistinu igrač, naredbom Instantiate() stvara se efekt sakupljanja, te se objekt na kojem je skripta uništava.

Slika 21: Efekt sakupljanja blaga

5. Primjena principa dizajna razina

U ovom poglavlju prikazati ću kako sam u izradi razina za ovu igru primijenio principe dobrog dizajna razina koje sam proučio i objasnio u poglavlju 3.

5.1. Postupno učenje igrača

U svojoj igri, pazio sam da igrač nauči potrebne mehanike prije nego li ih je potrebno koristiti u kompleksnim zagonetkama. Tako na primjer većina zagonetki od igrača zahtijeva osnovno razumijevanje mehanike skakanja.

Međutim, vjerojatno neće svaki igrač samostalno shvatiti da se može više skakati ako dulje drži tipku za skok. Zato sam već u prvoj razini stavio jednostavnu prepreku koju igrač mora preskočiti, a to je nemoguće osim ako ne drži tipku za skok malo duže pritisnutu.

Slika 22: Učenje skoka

Spomenuta prepreka prikazana je na slici gore. Time osiguravam da će igrač naučiti tu mehaniku te povećavam šansu da neće biti problema i frustracije kada ju kasnije bude morao koristiti.

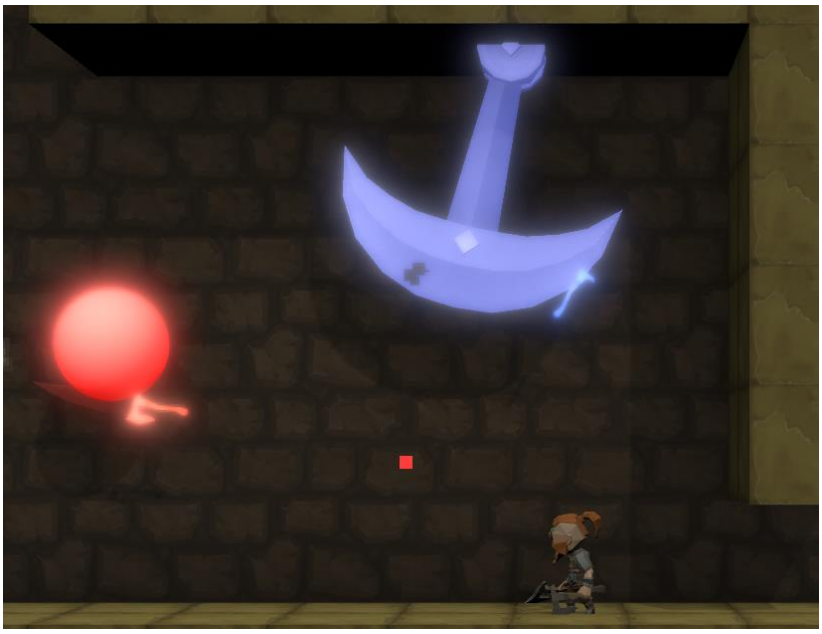
5.2. Izbjegavati ponavljanje

Budući da sam odabrao vrlo raznovrsnu mehaniku za glavnu mehaniku svoje igre, imao sam mogućnost stvoriti puno različitih zagonetki sa nekoliko osnovnih elemenata, te ni ja ni testeri moje igre nismo dobili dojam da se zagonetke ponavljaju.

Slika 23: Različite zagonetke

5.3. Traženje višestrukih primjena za mehanike

U mojoj igri igrač se može samo kretati i bacati sjekire. To slijedi načelo da treba imati što manje mehanika sa što više primjena. Tako na primjer igrač u mojoj igri magnetizmom može pomicati blokove koji mu priječe put, ali može i magnetizirati neprijatelja i neku od opasnih prepreka, kako bi se neprijatelj privukao prema opasnoj prepreci. Na taj način ga igrač može pobijediti, kako je prikazano na slici ispod.



Slika 24: Višestruka primjena mehanike

Tu vidimo da ista mehanika uistinu ima više primjena, a ako na ovoj igri nastavim raditi u budućnosti, dodao bih prekidače koje igrač sa sjekirom može pritisnuti, lance koje može presjeći i magnetizirati, te mnoge druge primjene za samo jednu jednostavno shvatljivu mehaniku.

5.4. Reći igraču što napraviti, ali ne i kako

U igri koju sam izradio iskustvo igranja zasniva se na rješavanju zagonetki. Igraču je jasno u kojem smjeru treba ići i gdje treba doći, ali ne i kako to ostvariti. Igrač, oružan moćima vladanja silama magnetizma sam stvara svoj put, koji ne mora uvijek biti jedinstven.

Slika 25: Reći igraču što napraviti, ali ne i kako

Na slici gore vidimo primjer jedne takve zagonetke. Igraču je u ovoj točci u igri jasno koji se blokovi mogu magnetizirati, te da je potrebno popeti se kroz prolaz gore desno koristeći te blokove i magnetizam.

5.5. Dodatni izazovi

Kako sam već spomenuo u prijašnjem poglavlju, u ovoj igri postoji dodatan izazov, koji nije nužno izvršiti za prijelaz igre, te služi samo kao testiranje vještina za one koji žele izazovnije iskustvo.

To su skrivena blaga, koja su sakrivena na teško dostupnim lokacijama u razini, te zahtijevaju naprednije razumijevanje mehanika nego što je potrebno za prijelaz bazične igre.

Slika 26: Dodatni izazovi

Pozicija jednog takvog skrivenog blaga prikazana je na slici gore. To daje mogućnost da i lošiji igrači mogu iskusiti cijelu priču igre, a za one napredne postoji dostojan izazov.

6. Zaključak

Tema ovog diplomskog rada je izrada puzzle platformer igre u alatu Unity. U sklopu ovog rada proučio sam žanr igara pod nazivom „puzzle platformer“, osmislio ideju za igru koja je služila kao praktični dio rada, te istu izradio. Za izradu video igre koristio sam alat Unity, a igru sam programirao u C# programskom jeziku, koristeći se pri tom alatom Microsoft Visual Studio. Proučio sam neke od najbitnijih principa dobrog dizajna razina i objasnio njihovu primjenu na vlastitoj video igri. Pokazao sam kako rade najbitnije skripte i sustavi koji sačinjavaju ovu igru.

Većina ljudi koji pokušavaju ući u svijet izrade video igara fokusiraju se na kompleksne programerske vještine i programiranje mnoštva mehanika. Iako su to sve dobre i korisne stvari kod izrade igara, više puta sam se uvjerio da igra može biti bolja i zabavnija s manje mehanika, koje su na pametan način iskorištene korištenjem pametnog dizajna razina.

Kroz ovaj diplomski rad razvio sam svoje vještine za izradu igara i shvatio sam vrijednost dobrog dizajna razina, te kako nedostatak istog može imati velik utjecaj. Planiram se nastaviti baviti proizvodnjom video igara, i mislim da će mi znanje koje sam dobio izrađujući ovaj rad uvelike pomoći u tome.

Popis literature

[1] Wikipedia, Unity (game engine), 2021. [na internetu]. Dostupno:

https://en.wikipedia.org/wiki/Unity_(game_engine) [pristupano 5.11.2021]

[2] Wikipedia, Platform game, 2021. [na internetu]. Dostupno:

https://en.wikipedia.org/wiki/Platform_game [pristupano 5.11.2021]

[3] Dan Taylor, Ten Principles of Good Level Design (Part 1), 2021. [na internetu]. Dostupno:

https://www.gamedeveloper.com/design/ten-principles-of-good-level-design-part-1- [pristupano 5.11.2021]

[4] Guest Blogger, Designing Fun Platforming Levels: Tips and Best Practices, 2021. [na internetu]. Dostupno: https://developer.amazon.com/blogs/appstore/post/a08b3316-4fdc-400b-884d-1ada24b485c1/designing-fun-platforming-levels-tips-and-best-practices [pristupano 5.11.2021]

[5] Dan Taylor, Ten Principles of Good Level Design (Part 2), 2021. [na internetu]. Dostupno:

https://www.gamedeveloper.com/design/ten-principles-of-good-level-design-part-2- [pristupano 5.11.2021]

[6] Eledris, How to Design Breathtaking 2D Platformer Levels, 2021. [na internetu]. Dostupno:

https://eledris.com/design-2d-platformer-levels/ [pristupano 5.11.2021]

[7] Unity, Scripting API Manual, 2021. [na internetu]. Dostupno:

https://docs.unity3d.com/ScriptReference/ [pristupano 5.11.2021]

[8] The Guardian, Portal 2 – review, 2021. [na internetu]. Dostupno:

https://www.theguardian.com/technology/gamesblog/2011/apr/19/portal-2-game-review [pristupano 5.11.2021]

[9] IGN, Celeste Wiki Guide, 2021. [na internetu]. Dostupno:

https://www.ign.com/wikis/celeste/Chapter_1-_Forsaken_City [pristupano 5.11.2021]

Popis slika

Popis slika treba biti izrađen po uzoru na indeksirani sadržaj, te upućivati na broj stranice na kojoj se slika može pronaći.