55. srečanje mladih raziskovalcev Slovenije 2021

RAZVOJ IN FAZE IZDELAVE VIDEO IGRE

Raziskovalno področje **RAČUNALNIŠTVO ALI TELEKOMUNIKACIJE**

Raziskovalna naloga

Mentor: Jernej Feguš

Avtorji: Jure Kozar, Tilen Bezjak, Miha Šket

Šola: SERŠ

Leto: 2021

Kraj: Maribor

KAZALO VSEBINE

KAZALO SLIK

ZAHVALA

Za pomoč in organizacijo pri raziskovalni nalogi se zahvaljujemo mentorju, sošolcem in staršem, ki so nam stali ob strani in nas spodbujali med potekom raziskovanja in pisanju naloge. Posebna zahvala je namenjena vsem zaposlenim na naši šoli, ki so nam omogočili vse potrebno za raziskovanje.

POVZETEK

Cilj oziroma namen naše raziskovalne naloge je bil, da sebi in ostalim ljudem pokažemo, kako zahtevno je narediti video igro v programskem okolju Unity. V raziskovalni nalogi bomo najprej opisali programsko okolje, v katerem bomo igro razvili, jezik, s katerim bomo programirali ter programa za obdelovanje zvoka in slike, ki bosta igro dodatno nadgradila. Poiskali bomo probleme, s katerimi se bomo srečali med izdelavo, jih analizirali in odpravili z najboljšimi možnimi rešitvami. Natančno bomo prikazali vse korake izdelave video igre (programska koda, urejanje likov in ozadja igre v programu Adobe Photoshop in urejanje zvoka v programu Bosca Ceoil).

1 UVOD

Še pred 20 leti je bilo zelo težko ustvariti novo video igro. Da bi jo v takratnem času lahko razvili, smo bili omejeni z napredkom tehnologije in viri informacij, ki niso bili dovolj zanesljivi. V današnjem času se lahko skoraj vsak odloči in s pomočjo naprednih programov ter obsežnim virom informacij ustvari svojo video igro. Naš cilj v raziskovalni nalogi je prikazati širši družbi, kako težko je narediti video igro, katere faze so težje in katere ne ter koliko časa in volje potrebujemo, da projekt uspe v celoti.

1.2 Hipoteze

1. Igro lahko izdelamo v enem mesecu.
2. Video igro je v programskem okolju Unity enostavno narediti.
3. Osnove premikanja lika je najlažje sprogramirati.

1.3 Razvojno okolje

1.3.1 Unity

Unity je najbolj priljubljeno okolje za razvijanje iger. Program Unity je v lasti podjetja Unity Technologies, napisan v programskem jeziku C# in C++. Vsebuje veliko funkcij in je dovolj prilagodljiv, da lahko ustvarimo skoraj vsako igro, ki si jo lahko predstavljamo. S programskimi funkcijami lahko tako začetnik kot tudi napreden razvijalec ustvari video igro. Program so uporabili za ustvarjanje iger kot Pokemon Go, Hearthstone, Rimworld, Cuphead in še mnoge druge. V programu Unity lahko izdelujemo tako dvodimenzionalne kot tridimenzionalne igre. Programsko okolje je prijazno tako umetnikom kot zagretim programerjem, saj orodja ponujajo ustvarjanje dvodimenzionalnih in tridimenzionalnih animacij, ki jih lahko ustvarimo iz nič. Igre lahko razvijamo na zelo visoki grafični ravni, saj nam program ponuja močno senčenje, končno obdelavo in visoko resolucijske svetlobne učinke. Glavni programski jezik, ki se uporablja v programskem okolju, je C#, dodaja se mu lahko tudi več drugih jezikov kot so JavaScript, Boo, IronPython, Lua in C++.

1.3.1.1 Zakaj smo izbrali okolje Unity?

To okolje smo si izbrali predvsem zato, ker je najprijaznejše za začetnike. V splošnem je Unity najbolj znano razvojno okolje, kar posledično s sabo prinese veliko informacij za ljudi, ki se s programom prvič srečajo. Tudi za izkušene razvijalce se vedno najde kakšna nova informacija. Za razliko od drugih programskih okolij ima Unity največjo izbiro orodij, s katerimi lahko ustvarimo igro. Hoteli smo se naučiti programirati v programskem jeziku C#, za kar je bilo okolje Unity najboljša izbira.

Slika 1: Programsko okolje Unity

1.3.2 Adobe Photoshop

Adobe Photoshop je program, s pomočjo katerega urejamo slike in animacije. Za Photoshop smo se odločili zato, ker nam ponuja največ orodij, s katerimi lahko oblikujemo slike, je enostaven za uporabo ter prijazen za začetnike. Je najbolj znan urejevalnik slik na svetu, kar nam posledično omogoča veliko informacij in navodil za uporabo. Adobe Photoshop nam je pomagal izdelati vse slikovne elemente video igre.

Slika 2: Adobe Photoshop

1.3.3 Bosca Ceoil

Bosca Ceoil je program za začetnike, specifično narejen za izdelavo zvočnih učinkov za video igre. Program je primeren tudi za tiste, ki nimajo nobenega znanja o glasbi oziroma zvočnih učinkih. Že vizualno nas je program takoj pritegnil, zato smo se odločili, da za našo igro uporabimo Bosca Ceoil. V igri bo prisotna predvsem klasična inštrumentalna glasba s

posebnimi zvočnimi efekti (skok, padec, eksplozija…). Bosca Ceoil je dostopen vsakemu uporabniku, saj je zastonj.

Slika 3: Bosca Ceoil

1.4 Razvoj igre

1.4.1 Izbira tipa igre

Kako smo izbrali tip igre? V izboru smo imeli več vrst iger, in sicer od tistih podobnih Tetrisu do tistih bolj naprednih, kot so na primer Minecraft, Stardew Valley, Old School RuneScape ali Dead Cells. Vsi smo bili navdušeni nad igro Celeste, ki je naš največji navdih. Celotna scena in lik temeljita na navdihu iz te igre. Zdelo se nam je, da je za začetnike v izdelovanju video iger to najboljša izbira.

1.4.2 Zakaj smo si izbrali pixel art?

Izbira slikovne podobe video igre ni bila enostavna. Kot prvo smo se vprašali, kako naj igra izgleda. Strinjali smo se, da je pixel art najprimernejši za začetniško video igro.

Prednosti

- Minimalistična podoba video igre (z manj pokažeš več).
- Enostavnejša vključitev v kodo, brez pomoči drugih programov.
- Cenovno dostopen program, s katerim smo pixel art oblikovali.
- Zavzema manj prostora.
- Ne potrebuje boljših računalniških komponent.
- Spomin na igre iz otroštva.
- Pixel art ostane enak skozi čas in razvoj drugih iger (ostane klasičen).
- Enostavnejša izdelava (časovno hitreje).
- Na spletu je dostopnih veliko informacij, zato se lahko hitro naučimo.

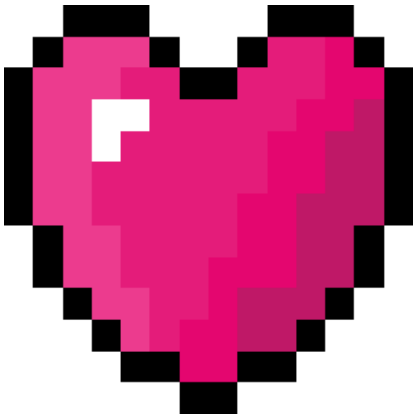
Slabosti

- Slika je manj kvalitetna, kar posledično odvrne ljudi, ki želijo videti visoko kvalitetne 2D igre.
- Pri izdelavi animacij smo lahko rahlo omejeni.
- Težje je poudariti podrobnosti.
- Težave pri izstopanju igre (zasičenost trga).

1.4.3 Potek izdelave slikovnega materiala

Ker smo na področju pixel arta in dela s slikami začetniki, smo se najprej morali naučiti osnov. Za začetek smo poskusili samostojno ustvariti nekaj naključnih slik. Prva težava, s katero smo se srečali, je bila povezana s senčenjem. Senčenje je v pixel artu zelo pomembno, saj lahko z odtenki različnih barv poudarimo podrobnosti na sliki. Veliko težav smo imeli z

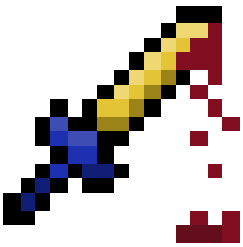
oblikovanjem slike. Naš cilj je bil, da slika izgleda čim bolj tridimenzionalno. Probleme nam je delala barvna paleta, s katero smo s težavo izbirali prave odtenke barv, ki bi se med seboj ujemale. Takoj smo vedeli, da je za pixel art potrebna majhna umetniška žilica, s katero lahko ustvarimo prave umetnine.



Slika 4: Prvi primer pixel arta (avtorski posnetek zaslona)

Ugotovili smo, da je na trgu grafična tablica, s katero bi naš pixel art naredili natančneje in hitreje. Takoj smo zgrabili priložnost in jo kupili. Izkazalo se je, da so bila naša predvidevanja pravilna, saj nam je tablica zelo olajšala in skrajšala delo. Napredek je bil očiten.

Slika 5: Grafična tablica

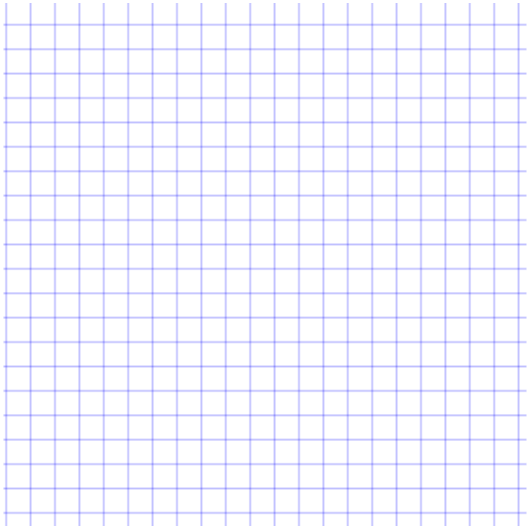


Slika 6: Drugi primer pixel arta (avtorski posnetek zaslona)

Po dvotedenskem učenju in odpravljanju težav smo se odločili, da začnemo z ustvarjanjem glavnega lika igre. Takoj smo se soočili s prvo težavo. Sprva smo si želeli, da bi bil glavni lik volkodlak. A že na začetku ustvarjanja smo ugotovili, da je naloga, ki smo si jo zadali, prezahtevna. Volkodlak bi vseboval preveč podrobnosti, katerih sami nismo znali izraziti. Težave bi nam delala predvsem dlaka in zobje, ki bi jih morali poudariti, zato smo se skupinsko odločili, da bo glavni lik igre človek. Predvidevali smo, da bomo imeli težave s senčenjem in poudarjanjem podrobnosti, kar se je izkazalo za pravilno. Težave smo odpravili tako, da smo si pomagali z vodiči na platformi YouTube.

1.4.3.1 Izbira mreže in barv

Za risanje smo izbrali kvadratno mrežo, saj bo igra dvodimenzionalna. Ta mreža je standardna in se običajno uporablja za pixel art. Pomembno je, da se vse črte sekajo pod pravim kotom.

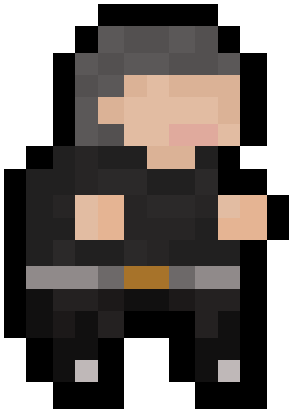


Slika 7: Kvadratna mreža

Odločili smo se, da bodo barve glavnega lika temnejše, scena pa svetlejša. Obleka glavnega lika bo temnosive barve z odtenki črne. Nosil bo svetlosiv pas z zlato sponko. Njegova koža bo bele barve s poudarjenimi usti. Oči lik ne bo imel, saj smo vzeli inspiracijo iz igre Celeste. Ugotovili smo, da bi bile oči premajhna podrobnost in da v igri ne bi izstopale. V tem primeru se opazijo slabosti pixel arta, saj smo bili omejeni glede podrobnosti lika. Tukaj smo se srečali s težavo glede velikosti lika. Edina rešitev, ki smo jo ugotovili, je bila, da smo lik povečevali

in zmanjševali, dokler se nam ni zdelo, da je razmerje med likom in dejansko sceno igre dobro.

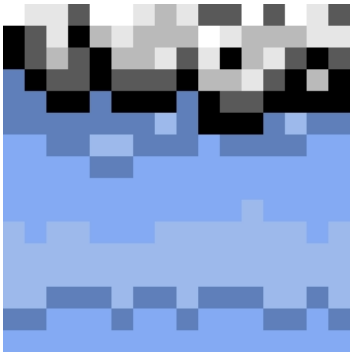
Ko smo se odločili, kako bi naj lik izgledal, smo ga začeli risati. S pomočjo grafične tablice in pisala smo ustvarili enostavno obliko lika. Obliko je bilo težko določiti, saj iz same oblike težko razberemo, kaj bi naj predstavljala. Ko je bila oblika končana, smo se lotili osnovnega barvanja. Vse barve, ki smo si jih izbrali že prej, smo tudi uporabili. Zadnja in tudi najtežja naloga je bila senčenje. Pri senčenju smo imeli res precej težav. Hoteli smo, da dvodimenzionalen lik daje občutek tridimenzionalnosti. S pomočjo video vodičev in nekaj vaje nam je to tudi uspelo. Naš cilj je bil, da lik izgleda čim bolj enostavno.



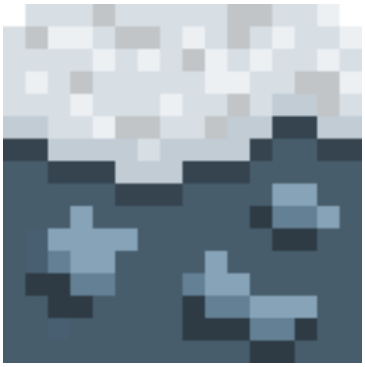
Slika 8: Končna oblika glavnega lika igre (avtorski posnetek zaslona)

1.4.3.2 Oblikovanje scene

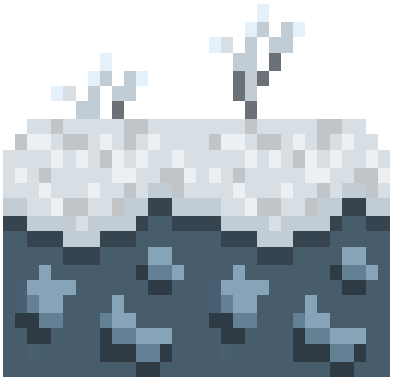
Tako kot pri ustvarjanju glavnega lika smo tudi tu uporabili kvadratno mrežo. Odločili smo se, da bo tema scene zimska. Vključevala bo sivo in belo barvo. Najprej smo se odločili izdelati otočke, po kateri bo lik skakal. Naše spretnosti so se iz dneva v dan izboljševale, zato težav s risanjem otočkov nismo imeli. Odločili smo se, da bo velikost otočkov 16 krat 16 slikovnih pik. Prvi otoček nam ni bil všeč, zato ga na koncu nismo uporabili.



Slika 9: Prvi primer otočka (avtorski posnetek zaslona)

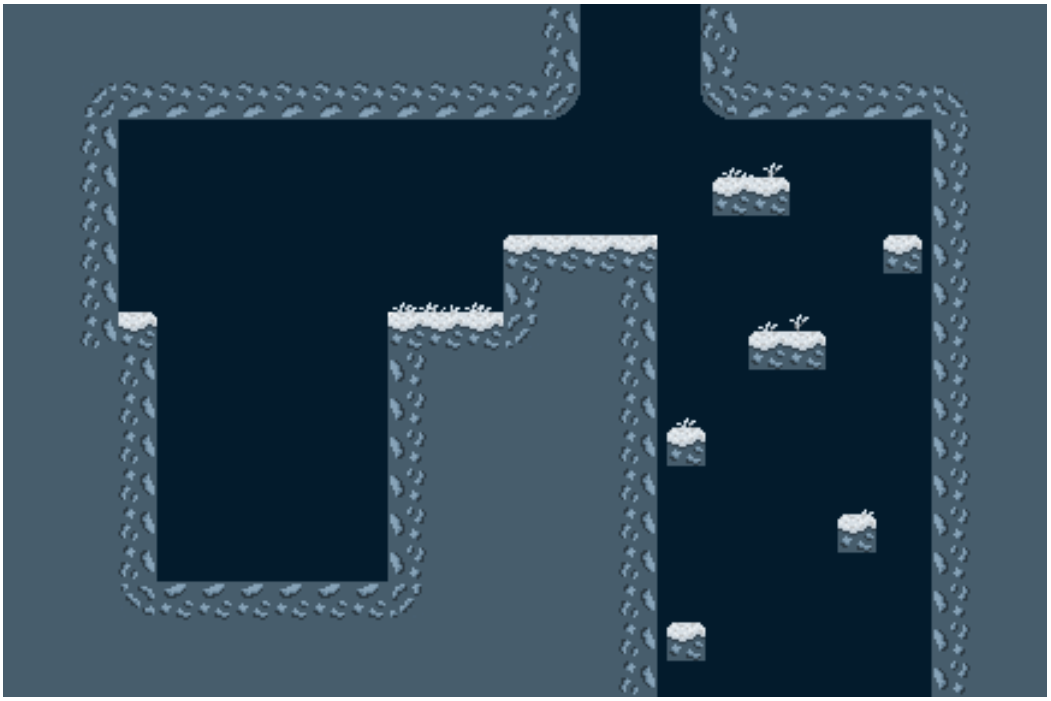


Slika 11: Končan otoček (avtorski posnetek zaslona)



Slika 10: Druga verzija končanega otočka (avtorski posnetek zaslona)

Ko smo dokončali vse otočke, smo se lotili scene. Zdelo se nam je, da bo problem velikost celotne scene. Ugotovili smo, da s tem ne bo težav, saj lahko v programskem okolju Unity spreminjamo velikost scene poljubno. Pozorni smo bili tudi na to, da lik lepo izstopa in da scena ni pretemna. Na platformo scene smo dodali še nekaj podrobnosti, kot sta kamenje in trava. Razporejenost otočkov po sceni smo spreminjali glede na težavnost nivoja igre. Prehod z enega nivoja na drugega smo naredili tako, da smo iz ene scene v drugo speljali majhen tunel na vrhu slike, ki igralcu omogoči prehod na naslednji nivo.

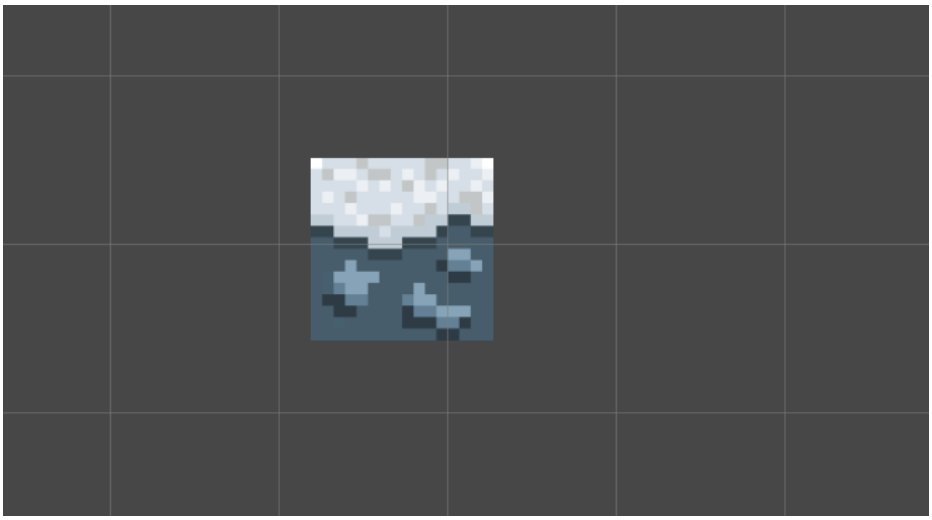


Slika 12: Primer scene (avtorski posnetek zaslona)

1.4.4 Programiranje igre

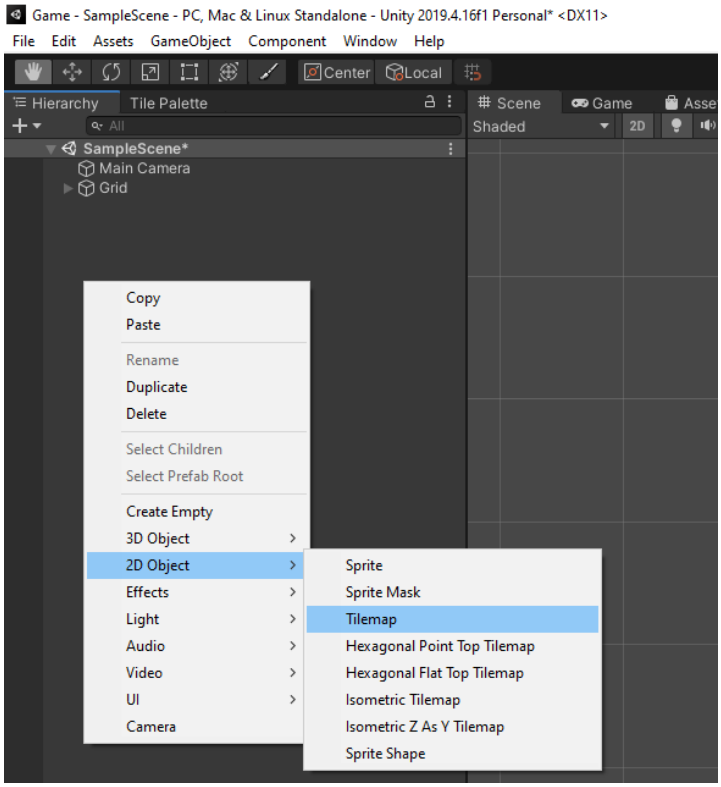
Za izdelavo video igre smo potrebovali programsko okolje, ki smo si ga izbrali. Na uradni spletni strani podjetja, ki je program ustvarilo, smo si naložili Unity Personal. Programsko okolje je zastonj, ker naša igra ne ustvarja več kot 100.000 € dohodka. Seznanili smo se z vsemi podanimi pravicami in tako je bil prenos končan. Ob prvem pogledu na program smo bili zelo zmedeni in začudeni. Prve dni uporabe smo se še lovili z lokacijo vseh orodij programa. Da smo sploh razumeli delovanje orodij v Unity Personal, smo morali pogledati

nekaj video vodičev, ki so nam podali osnovno razumevanje programa. Najtežje se nam je bilo prilagoditi na izgled programa, saj ni bil podoben nobenemu programu, s katerim smo programirali že prej. Prva težava, s katero smo se srečali, je bila velikost slik. Ko smo sliko prilepili v program Unity, je bila slika premajhna. Rešitev smo iskali v orodjih programa, kar pa ni bila najlažja naloga, saj ima Unity veliko orodij s podobnimi funkcijami. Drugo težavo nam je predstavljala mreža v Unityu. Ko smo sliko prilepili na mrežo, ta ni bila poravnana s kvadrati.

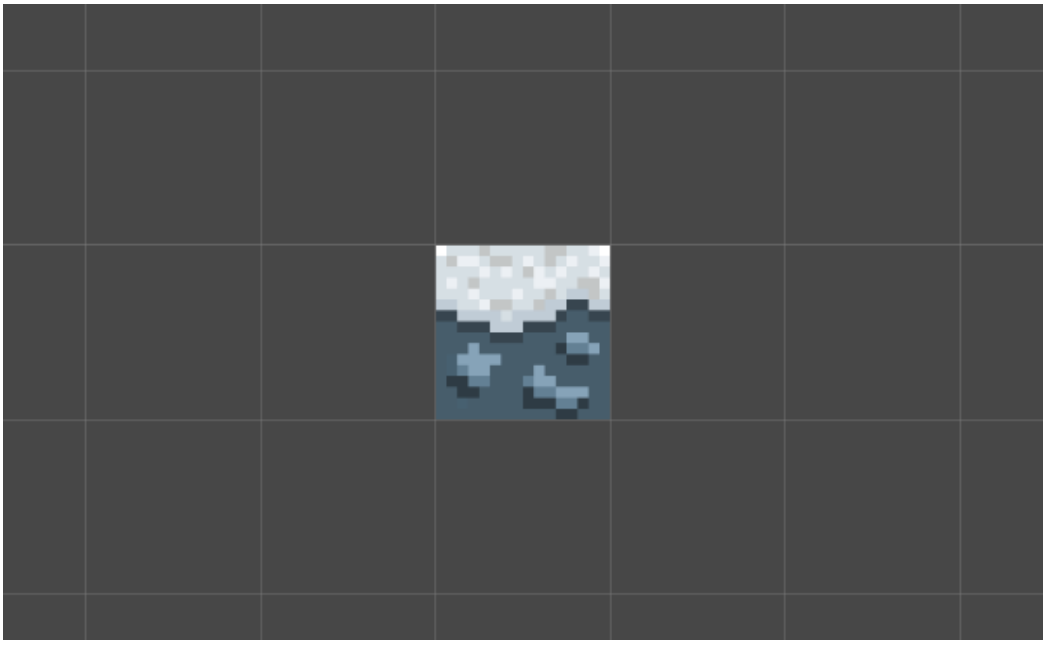


Slika 13: Druga pojavljena težava (avtorski posnetek zaslona)

Težavo smo odpravili z desnim klikom na Hierachy, izbrali možnosti 2D in Tilemap. Ta funkcija nam je omogočila sliko poravnati s kvadratkom v mreži.

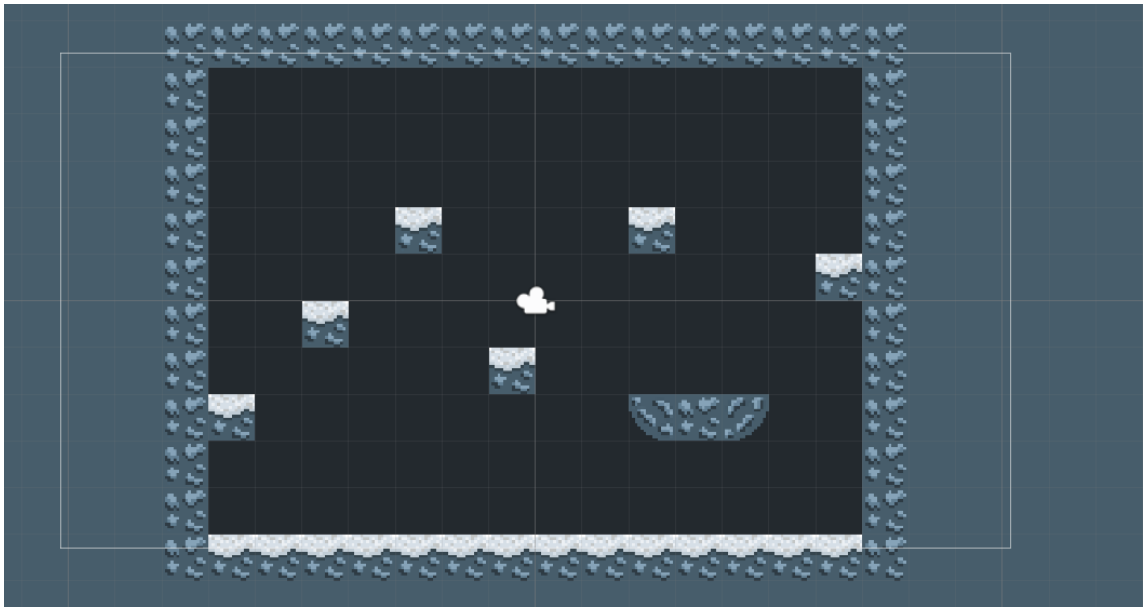


Slika 14: Pot do rešitve prvega problema (avtorski posnetek zaslona)



Slika 15: Končna rešena težava (avtorski posnetek zaslona)

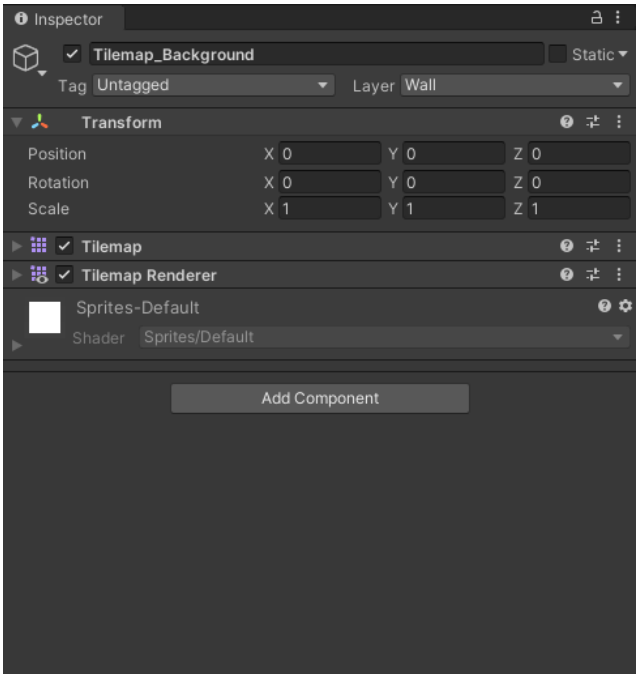
V nadaljevanju smo tako s preostalimi bloki sestavili celotno ozadje. Pri tem delu se s težavami nismo več srečali. Nastalo ozadje je postalo naša poizkusna arena, v kateri smo preizkušali gibe lika ter njegov odziv na okolico.



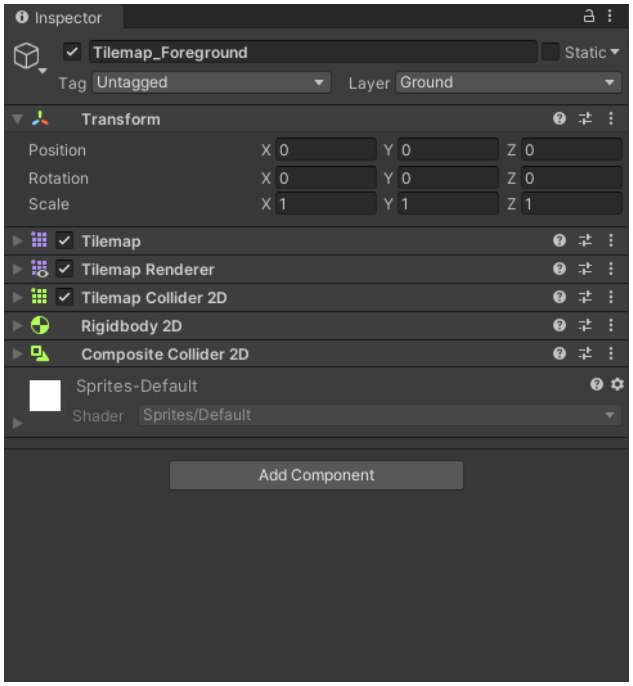
Slika 16: Poizkusna arena (avtorski posnetek zaslona)

Da bi se lik po areni premikal normalno, smo morali ustvariti nekakšen prazen prostor. To smo naredili tako, da smo v »Gridu« dodali dva prazna objekta, ki smo ju poimenovali Tilemap_Background in Tilemap_Foreground. Razlika med objektoma je, da se tako imenovanega sprednjega ozadja lik lahko dotika, zadnje pa mu dopušča prosto premikanje po areni. V objekt Tilemap_Background smo dodali komponente Tilemap in Tilemap Renderer.

Objekt Tilemap_Background vsebuje komponente Tilemap, Tilemap Renderer, Tilemap Collider 2D, Rigidbody 2D in Composite Collider 2D. V ta objekt smo dodali še plast z imenom Ground, kar pomeni, da smo dodali tla.

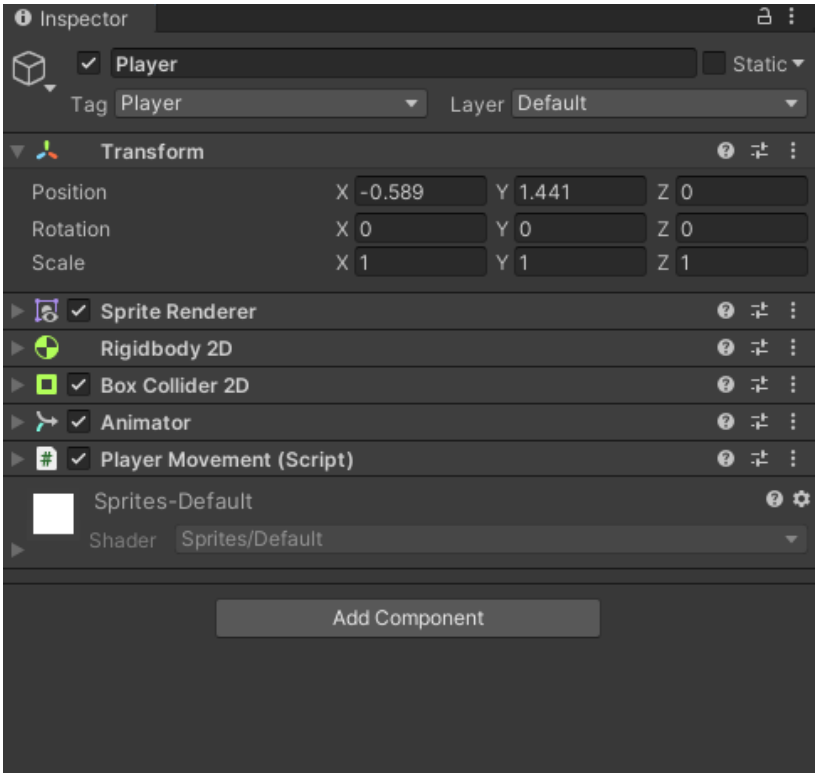


Slika 17: Tilemap_Background (avtorski posnetek zaslona)



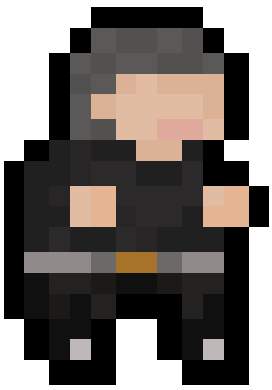
Slika 18: Tilemap_Foreground (avtorski posnetek zaslona)

V naslednjem koraku smo dodali lik igre. Najprej smo ustvarili nov prazen objekt in ga poimenovali Player. Za boljšo preglednost smo igralcu dodali oznako Player. Da se bo lik lahko premikal, smo mu dodali naslednje komponente: Sprite Renderer, Rigidbody 2D, Box Collider 2D, Animator in novo komponento z imenom Player Movement.



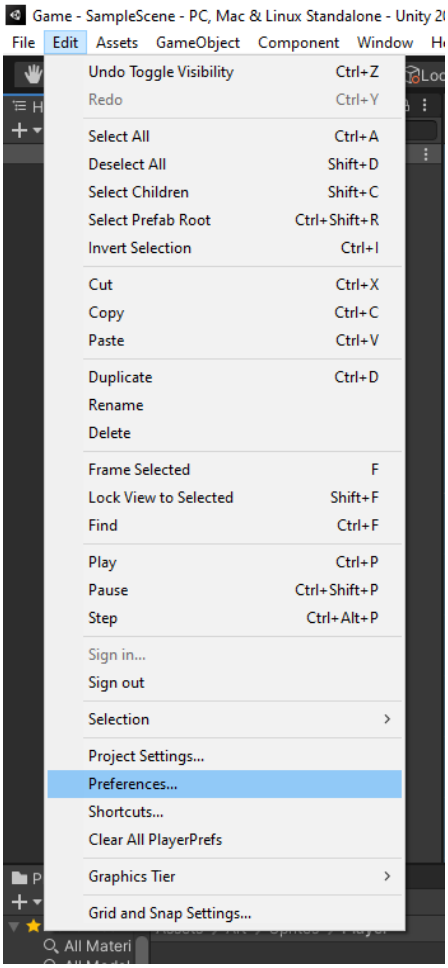
Slika 19: Komponente za objekt Player (avtorski posnetek zaslona)

V komponento Sprite Renderer smo dodali sliko lika, ki stoji pri miru. V enaki komponenti smo dodali še Sorting Layer in ga nastavili na entity.

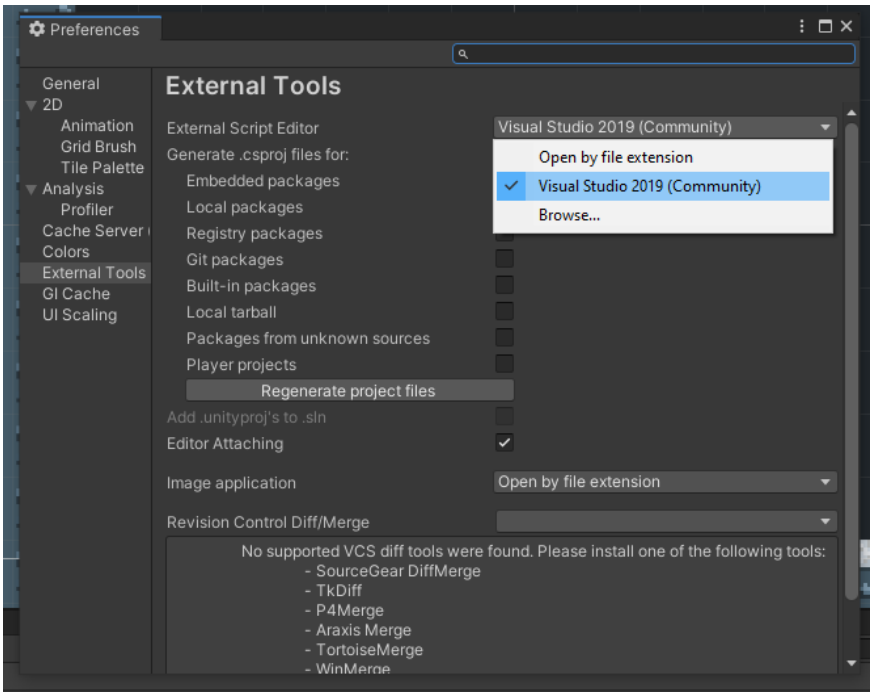


Slika 20: Začetna postavitev lika (avtorski posnetek zaslona)

Ko smo končali s podobo ozadja, smo se lotili programiranja kode. Kodo smo programirali v programskem okolju Visual Studio Community 2019. Visual Studio je bilo potrebno povezati z okoljem Unity, pri čemer smo se ponovno srečali s težavo. Ugotovili smo, da je treba za povezavo z Unityjem vključiti knjižnico s komando imenovano »using UnityEngine;«. Pobrskali smo po spletu in hitro našli potrebno rešitev. V programu Unity smo v orodni vrstici pod oznako uredi poiskali možnost Preferences.



Slika 21: Pot do rešitve drugega problema (avtorski posnetek zaslona)



Slika 22: Rešitev drugega problema (avtorski posnetek zaslona)

Odpre se nam okno z več možnostmi. Izberemo External Tools in v desnem okvirčku označimo možnost Visual Studio 2019 (Community). S tem smo kodo v Visual Studiu in okolju Unity povezali. Ko smo kodo naknadno spreminjali, se je spreminjala tudi igra v Unityju.

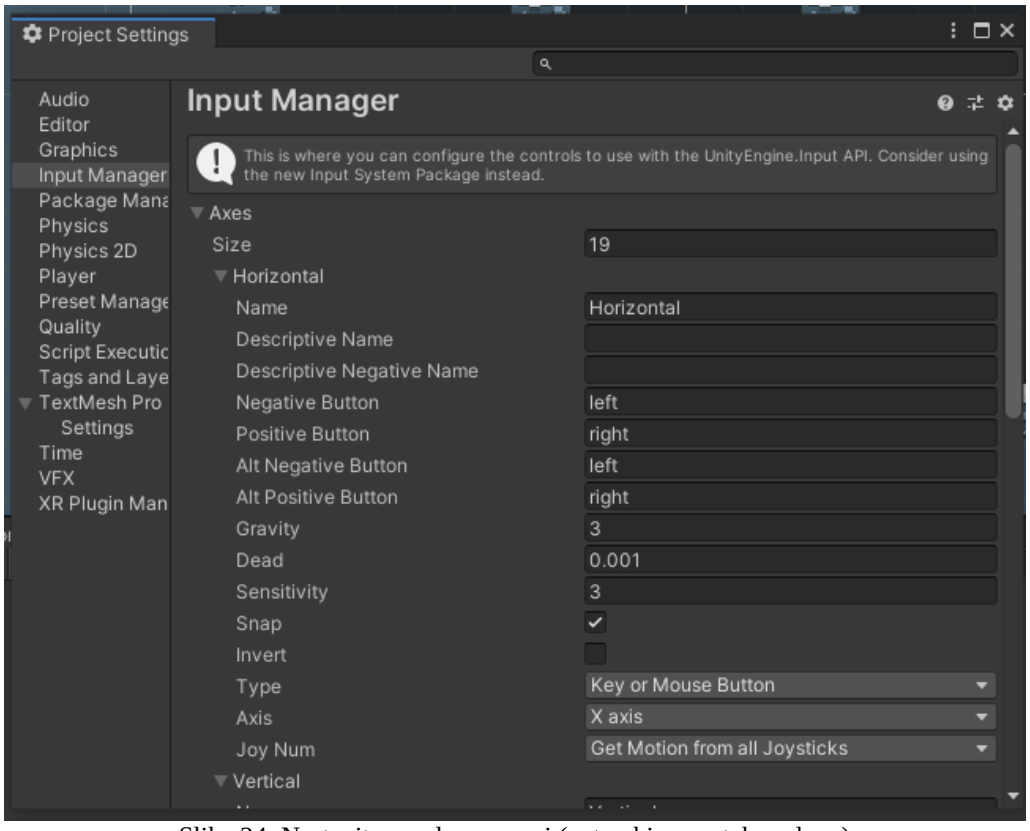
1.4.4.1 Programiranje kode za premikanje lika

Vedeli smo, da bo programiranje kode za premikanje lika zelo pomembno in zahtevno, zato smo si že na začetku malenkost olajšali delo. Hoteli smo, da je koda pregledna in razumljiva. Odločili smo se, da bomo za razvrščanje oziroma boljšo preglednost spremenljivk v programu uporabili več glav. Glava v programu združi spremenljivke pod enakim naslovom, tako da so urejene ter da jim je enostavno spreminjati vrednosti.

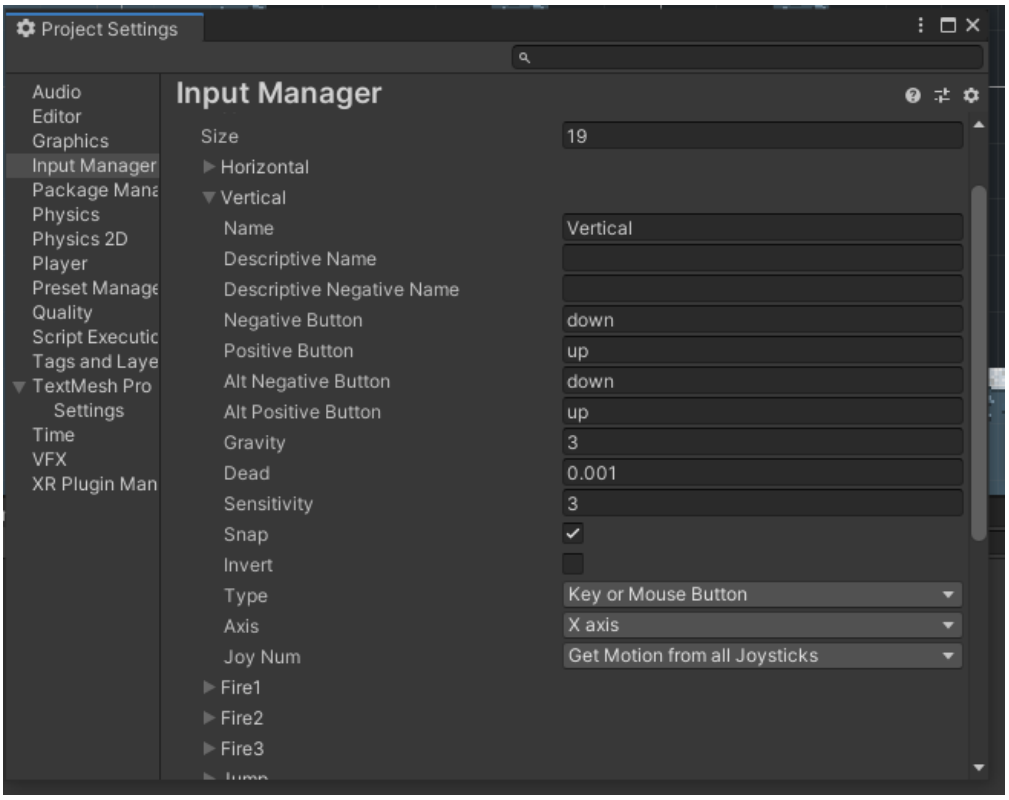
```
[Header("Movement Variables")]
```

Slika 23: Primer glave v programu (avtorski posnetek zaslona)

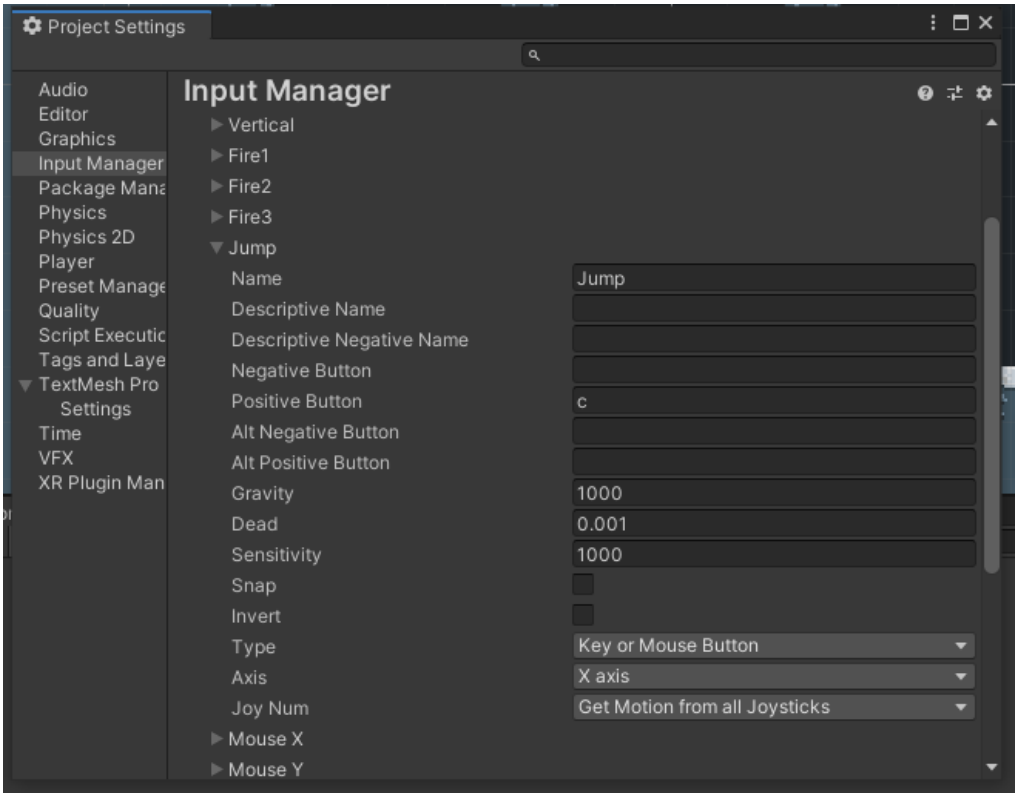
Prvi korak pri pisanju kode je bil, da omogočimo vnos za premikanje s tipkovnico (puščicami). Pri tem smo se srečali s problemom. Težave nam je povzročalo razumevanje v povezavi med nastavitvami v Unityju in kako jih dejansko vključiti v kodo. S pomočjo video vodiča smo ugotovil, da je potrebno v Unityju ustvariti vodoravno in navpično os ter skok. Nastaviti jim je treba tipki, s katerima vnesemo spremembo gibanja. Nato moramo v kodi napisati ukaz, s katerim povemo, da so puščice tipke, s katerimi spremenimo gibanje lika in jih pravilno enako poimenovati kot v Unityju.



Slika 24: Nastavitve vodoravne osi (avtorski posnetek zaslona)



Slika 25: Nastavitve navpične osi (avtorski posnetek zaslona)



Slika 26: Nastavitve za skok (avtorski posnetek zaslona)

```
private Vector2 GetInput()
{
    return new Vector2(Input.GetAxisRaw("Horizontal"), Input.GetAxisRaw("Vertical"));
}
```

Slika 27: Ukaz za vnos navpičnega in vodoravnega gibanja (avtorski posnetek zaslona)

Naslednja težava, ki nas je ovirala do končanja igre, je bila, da nismo mogli dostopati do komponent Rigidbody in Animator. Obe komponenti sta zelo pomembni za fizično premikanje lika in animacijo. Rešitev smo odkrili na blogu ljudi, ki so se že srečali s podobno težavo. Ugotovili smo, da je potrebno za spreminjanje teh lastnosti v kodi dodati funkcijo start. Ta funkcija se izvede samo ob zagonu igre in omogoči spreminjanje lastnosti v Unity–ju. Ustrezno smo morali uporabiti tudi funkcijo update, ki se izvede v vsaki sličici na sekundo. Njeno izvajanje je odvisno od tega, kaj funkcija vsebuje. Ta koda do zdaj še nima svojega učinka, temveč samo zazna pritisk na gumb.

```
private void Start()
{
    rb = GetComponent<Rigidbody2D>();
    anim = GetComponent<Animator>();
}
```

Slika 28: Funckija start (avtorski posnetek zaslona)

```
private void Update()
{
    horizontalDirection = GetInput().x;
    verticalDirection = GetInput().y;
```

Slika 29: Funkcija update (avtorski posnetek zaslona)

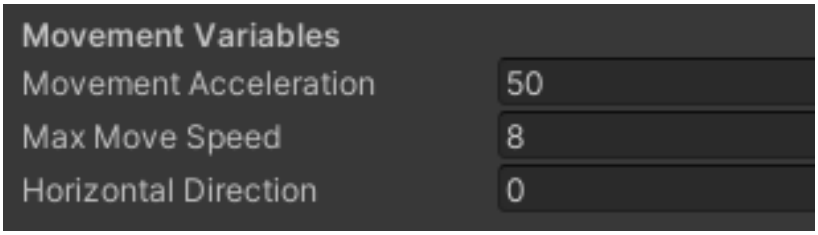
Obe funkciji sta prisotni ob vsakem novem ustvarjanju skripte, kar pomeni, da sta ob odločitvi dodajanja nove lastnosti liku funkciji že vključeni v kodi.

Vedeli smo, da ta koda lika še ne premakne, zato smo mu morali dodati fizične lastnosti, s katerimi se bo lahko premikal. V skripti PlayerMovement smo ustvarili novo funkcijo imenovano MoveCharacter. To funkcijo smo poklicali v Unityjevi funkciji FixedUpdate. Za razliko od funkcije Update, se ta funkcija lahko izvede enkrat, ničkrat ali večkrat na sličico na sekundo. Ker nismo prav vedeli, kako bi se lotili kode, smo si pomagali z video vodičem in spletno stranjo Unityja. Na spletni strani smo našli dodatne informacije za komponento Rigidbody2D. Tukaj smo se prvič srečali s fiziko v programiranju igre. Liku je bilo potrebno dodati silo, ki pa se izračuna po formuli masa*pospešek. Logika, ki stoji za to formulo, je, da večja kot je masa, večja je sila, potrebna za pospešitev do določene hitrosti. Napisali smo torej kodo, s katero omogočimo vnos za premik lika ter tej kodi dodali fizično lastnost lika v obliki programske kode.

```
private void MoveCharacter()
{
    rb.AddForce(new Vector2(horizontalDirection, 0f) * movementAcceleration);
```

Slika 30: Funkcija MoveCharacter (avtorski posnetek zaslona)

Spremenljivkam, ki smo jih ustvarili na začetku programa, smo dodali določene vrednosti. Ker smo ugotovili, da se lik premika nenadzorovano in da se ob pritisku na gumb zelo hitro premakne, smo morali hitro poiskati rešitev. Našli smo jo na video vodiču, ki nam je prikazal, da lahko spremenljivke v Unityju spremenimo do te mere, da se bo lik lahko normalno premikal. S poskušanjem smo prišli do vrednosti, ki najbolje ustrezajo konceptu naše igre.



Slika 31: Vrednosti spremenljivk za premikanje lika (avtorski posnetek zaslona)

V funkciji MoveCharacter smo morali dodati if stavek, ki omeji pospešek lika, da lahko lažje nadzorujemo njegovo premikanje.

```
private void MoveCharacter()
{
    rb.AddForce(new Vector2(horizontalDirection, 0f) * movementAcceleration);
    if(Mathf.Abs(rb.velocity.x) > maxMoveSpeed)
    {
        rb.velocity = new Vector2(Mathf.Sign(rb.velocity.x) * maxMoveSpeed, rb.velocity.y);
    }
}
```

Slika 32: If stavek prikazan v kodi (avtorski posnetek zaslona)

Premikanje lika se je že zdelo bližje končnemu, a nas je še vseeno motilo to, da je lik drsel po podlagi. Ko smo na tipkovnici pritisnili tipko za premik v desno ali levo, lik ni napravil enega samega koraka, ampak ga je nekako podrsalo na eno izmed strani. Tudi to rešitev smo našli preko video vodiča in spletne strani Unity. Spet smo morali uporabiti novo funkcijo, imenovano ApplyGroundLinearDrag. Učinek te funkcije se je takoj poznal. Funkcija v glavnem naredi upor na liku in mu onemogoči drsenje na eno izmed strani.

```
private void ApplyGroundLinearDrag()
{
    if (Mathf.Abs(horizontalDirection) < 0.4f)
    {
        rb.drag = groundLinearDrag;
    }
    else
    {
        rb.drag = 0f;
    }
}
```

Slika 33: Funkcija ApplyGroundLinearDrag (avtorski posnetek zaslona)

Premikanje lika se nam še vedno ni zdelo popolno, saj je med hitro spremembo smeri gibanja še vedno zdrsel na eno ali drugo stran. Ugotovili smo, da se tudi ta problem da rešiti z dodajanjem ukaza, ki liku omogoči hitro spremembo smeri gibanja, ne da bi oddrsal na eno izmed strani. Rešitev smo spet našli preko video vodiča, kjer nam je bila funkcija tega ukaza zelo dobro razložena.

```
private bool ChangingDirection =>
    (rb.velocity.x > 0f && horizontalDirection < 0f) || (rb.velocity.x < 0f && horizontalDirection > 0f);
```

Slika 34: Funkcija ChangingDirection (avtorski posnetek zaslona)

To na novo ustvarjeno funkcijo smo morali priključiti funkciji ApplyGroundLinearDrag, da je celotna stvar lahko delovala.

```
private void ApplyGroundLinearDrag()
{
    if (Mathf.Abs(horizontalDirection) < 0.4f || ChangingDirection)
    {
        rb.drag = groundLinearDrag;
    }
    else
    {
        rb.drag = 0f;
    }
}
```

Slika 35: Dodajanje funkcije ChangingDirection (avtorski posnetek zaslona)

S tem zadnjim ukazom smo bili s premikanjem na levo in desno stran zadovoljni.

Naslednji korak v izpopolnjevanju premikanja je seveda skok. Pri kodiranju skoka smo se srečali s številnimi težavami in napakami v igri. Tako kot za vsako drugo spremembo gibanja smo tudi za to morali ustvariti novo funkcijo z imenom Jump. Osnovna funkcija vsebuje ukaz, da po prejetem vnosu na tipkovnici lik skoči in ponovno pristane na tleh. Tudi to funkcijo smo morali vključiti v funkcijo Update.

```
private void Jump()
{
    rb.velocity = new Vector2(rb.velocity.x, 0f);
    rb.AddForce(Vector2.up * jumpForce, ForceMode2D.Impulse);
}
```

Slika 36: Funkcija Jump (avtorski posnetek zaslona)

Učinek funkcije smo takoj preizkusili v programskem okolju Unity. Presenetilo nas je, da ima ta ukaz zelo veliko pomanjkljivosti. Ugotovili smo, da lik lahko skoči ob vsakem pritisku na gumb za skok ne glede na to, ali je njegova pozicija v zraku ali na tleh. Hitro smo se lotili dela in skušali ugotoviti, kako bi ta problem rešili. Z znanjem, ki smo ga pridobili v dosedanjem programiranju igre, smo ugotovili, da moramo najprej določiti, kje so tla. Zato smo ustvarili več novih struktur z imeni groundLayer tipa LayerMask, groundRaycastLength in onGround.

```
[Header("Ground Collision Variables")]
[SerializeField] private float groundRaycastLength;
private bool onGround;
```

Slika 37: Strukturi groundRaycastLength in onGorund (avtorski posnetek zaslona)

```
[Header("LayerMask")]
[SerializeField] private LayerMask groundLayer;
```

Slika 38: Struktura groundLayer (avtorski posnetek zaslona)

S strukturo groundRaycastLength dodamo liku črto, ki je iz središča lika usmerjena proti tlom. Pomembno je, da ji določimo dolžino in LayerMask po imenu groundLayer. Sama koda nam je seveda povzročala težave, zato smo si pomagali z video vodičem. Da se je ukaz pravilno izvedel, smo morali dodati novo funkcijo z imenom CheckCollisions.

```
private void CheckCollisions()
{
    //Ground Collisions
    onGround = Physics2D.Raycast(transform.position * groundRaycastOffset,
        Vector2.down, groundRaycastLength, groundLayer);
```

Slika 39: Funkcija CheckCollisions (avtorski posnetek zaslona)

Prikazana koda sama po sebi ne prikaže črte, ampak moramo za njen prikaz napisati še eno funkcijo imenovano OnDrawGizmos. Ta ukaz nam olajša delo pri spreminjanju dolžine črte, saj sedaj lahko vidimo kam črta sega. Njeno dolžino smo nastavili na 0,75. Barvo lahko določimo poljubno.

```
private void OnDrawGizmos()
{
    Gizmos.color = Color.green;
    //Ground Check
    Gizmos.DrawLine(transform.position , transform.position + Vector3.down * groundRaycastLength);
```

Slika 40: Funkcija OnDrawGizmos (avtorski posnetek zaslona)

Da bo lahko prejšnji ukaz deloval, smo morali dodati novo spremenljivko CanJump tipa bool. Ukaz v spremenljivki se izvede le takrat, ko na tipkovnici pritisnemo tipko, ki jo imamo določeno za skok in ko se lik dotakne tal oziroma neke ravne površine (lahko je tudi otoček).

```
private bool CanJump => Input.GetButtonDown("Jump") && onGround;
```

Slika 41: Spremenljivka CanJump (avtorski posnetek zaslona)

Sama spremenljivka še ne stori nič, zato smo jo morali poklicati v pravem trenutku izvajanja programa. If stavek, ki smo ga za izvajanje skoka zapisali v funkciji Update, smo morali sedaj prestaviti v funkcijo FixedUpdate in jo ustrezno spremeniti.

```
if (CanJump)
{
    Jump();
}
```

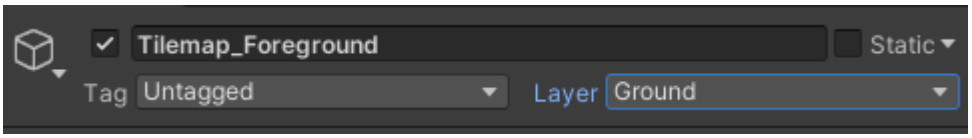
Slika 42: Spremenjen if (CanJump) stavek v funkciji FixedUpdate (avtorski posnetek zaslona)

Za pravilno delovanje moramo tudi funkcijo CheckCollisions klicati v funkciji FixedUpdate.

```
private void FixedUpdate()
{
    CheckCollisions();
```

Slika 43: Klic funkcije CheckCollisions (avtorski posnetek zaslona)

Po zapisu kode smo v programskem okolju Unity brez težav ustvarili novo plast z imenom Ground. Posledično smo tudi igralnemu objektu z imenom Player morali nastaviti Ground Layer na ground.



Slika 44: Novoustvarjena plast Ground (avtorski posnetek zaslona)

Slika 45: Sprememba v igralnem objektu (avtorski posnetek zaslona)

Sedaj smo popravili težavo glede neskončnega skakanja v zraku. Tu pa nas je pričakala še druga težava. Problem smo opazili pri padanju lika, saj je to bilo počasno in neenakomerno. Do rešitve smo prišli s pomočjo bloga programerja, ki je svojim bralcem predlagal rešitev v obliki linearnega upora v zraku. Ta pomaga, da se z likom v zraku lažje premikmo. Ustvarili smo novo funkcijo z imenom ApplyAirLinearDrag.

```
private void ApplyAirLinearDrag()
{
    rb.drag = airLinearDrag;
}
```

Slika 46: Funkcija ApplyAirLinearDrag (avtorski posnetek zaslona)

To funkcijo smo morali klicati še v funkciji FixedUpdate.

```
if (onGround)
{
    ApplyGroundLinearDrag();
}
else
{
    ApplyAirLinearDrag();
}
```

Slika 47: Klic funkcije ApplyAirLinearDrag (avtorski posnetek zaslona)

Premikanje v zraku je bilo nedvomno lažje, a je lik še vedno zelo počasi padal na tla. Dodati smo morali dve novi spremenljivki imenovani fallMultiplier in lowJumpFallMultiplier. Ti dve spremenljivki smo uporabili v novi funkciji imenovani FallMultiplier. Vključiti smo jo morali tudi v funkcijo FixedUpdate.

```
private void FallMultiplier()
{
    if (rb.velocity.y < 0)
    {
        rb.gravityScale = fallMultiplier;
    }
    else if(rb.velocity.y > 0 && !Input.GetButton("Jump"))
    {
        rb.gravityScale = lowJumpFallMultiplier;
    }
    else
    {
        rb.gravityScale = 1f;
    }
}
```

Slika 48: Funkcija FallMultiplier (avtorski posnetek zaslona)

Z delovanjem celotnega premikanja smo bili zadovoljni. Ker smo hoteli narediti igro še bolj napredno in zanimivo, smo se odločili dodati dvojni skok. Dodali smo dve novi spremenljivki imenovani extraJumps in extraJumpValue. V funkciji FixedUpdate smo spremenili if stavek in mu dodali nov ukaz ustvarjen iz teh dveh spremenljivk.

```
if (onGround)
{
    extraJumpValue = extraJumps;
    ApplyGroundLinearDrag();
```

Slika 49: Sprememba if stavka v funkciji FixedUpdate (avtorski posnetek zaslona)

Tudi v funkcijo Jump smo morali dodati ukaz, ki se izvede, če je zadnji skok že bil porabljen, da s tem preprečimo izvajanje še več skokov.

```
if (!onGround) {
    extraJumpValue--;
}
```

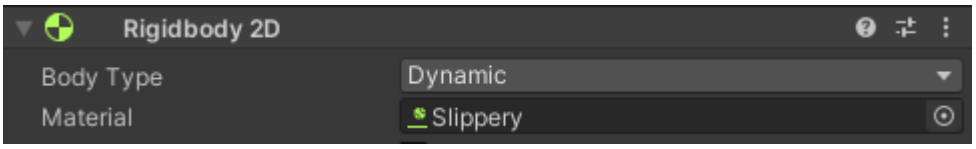
Slika 50: If stavek v funkciji Jump (avtorski posnetek zaslona)

Dodatno smo morali spremeniti tudi funkcijo CanJump in ji dodati lastnosti za dvojni skok.

```
private bool CanJump =>
    Input.GetButtonDown("Jump") && (onGround || extraJumpValue > 0);
```

Slika 51: Spremenjena funkcija CanJump (avtorski posnetek zaslona)

Po zadnji spremembi v funkciji CanJump smo v okolju Unity preizkusili delovanje dvojnega skoka. Bili smo zadovoljni z videnim, a smo še vseeno videli prostor za napredek in popravke. V tem zagonu in poizkusu igre smo odkrili napako v programu. Ko se je lik dotaknil zida, se je nanj prilepil. To težavo smo lahko odpravili v nekaj enostavnih korakih. V okolju Unity smo ustvarili nov fizični material 2D imenovan Slippery, ga dodali objektu ter mu nastavili lastnosti trenje in poskočnost na vrednost 0.



Slika 52: Dodajanje materiala v objekt Player (avtorski posnetek zaslona)

Slika 53: Nastavljanje vrednosti za trenje in poskočnost (avtorski posnetek zaslona)

V naslednji točki smo razmislili, kako bi igralcu olajšali igranje in mu omogočili, da igra na najvišji možni kvaliteti. Bili smo mnenja, da lahko lik izboljšamo in tako uporabniku kar se da pomagamo na poti do cilja. Iz različnih video vodičev smo izvedeli, da lahko igralcu pomagamo s tako imenovanim »Coyote time«. Ta ukaz igralcu s počasnejšimi reakcijami pomaga do skoka ne glede na to, ali je igralec še vedno na trdi površini oziroma je že v padanju. Ta čas tako imenovane navidezne trde površine je v bistvu zelo majhen (nekaj milisekund).

Da bi nam ta ukaz uspel, smo morali dodati novi spremenljivki z imenoma hangTimeCounter. in hangTime. Nato smo v funkciji FixedUpdate spet spremenili if stavek in mu dodali nov ukaz, ustvarjen iz navedenih spremenljivk.

```
if (onGround)
{
    extraJumpValue = extraJumps;
    ApplyGroundLinearDrag();
    hangTimeCounter = hangTime;
}
```

Slika 54: Sprememba if stavka z dodajanjem spremenljivk hangTimeCounter in hangTime (avtorski posnetek zaslona)

S pomočjo video vodiča smo ugotovili, kako nastaviti čas navidezne trde površine. V enaki funkciji FixedUpdate smo morali spremeniti še stavek else z ustreznim ukazom.

```
else
{
    ApplyAirLinearDrag();
    FallMultiplier();
    hangTimeCounter -= Time.fixedDeltaTime;
```

Slika 55: Sprememba else stavka v funkciji FixedUpdate (avtorski posnetek zaslona)

Spremeniti smo morali še ukaz spremenljivke CanJump.

```
private bool CanJump =>
    Input.GetButtonDown("Jump") && (hangTimeCounter > 0f || extraJumpValue > 0);
```

Slika 56: Dodajanje hangTimeCounter spremenljivke v ukaz spremenljivke CanJump (avtorski posnetek zaslona)

Zmotila nas je še ena stvar v programu. Če smo hoteli, da lik med pristajanjem na tla skoči še enkrat, se nam je odriv zdel zelo počasen. Hoteli smo to spremeniti in hitro našli rešitev s tako imenovanim »Jump Bufferjem«. Spet smo dodali dve novi spremenljivki, imenovani jumpBufferLength in jumpBufferCounter.

Za razliko od prejšnje rešitve smo morali sedaj dodati if stavek v funkcijo Update.

```
if (Input.GetButtonDown("Jump"))
{
    jumpBufferCounter = jumpBufferLength;
}
else
{
    jumpBufferCounter -= Time.deltaTime;
}
```

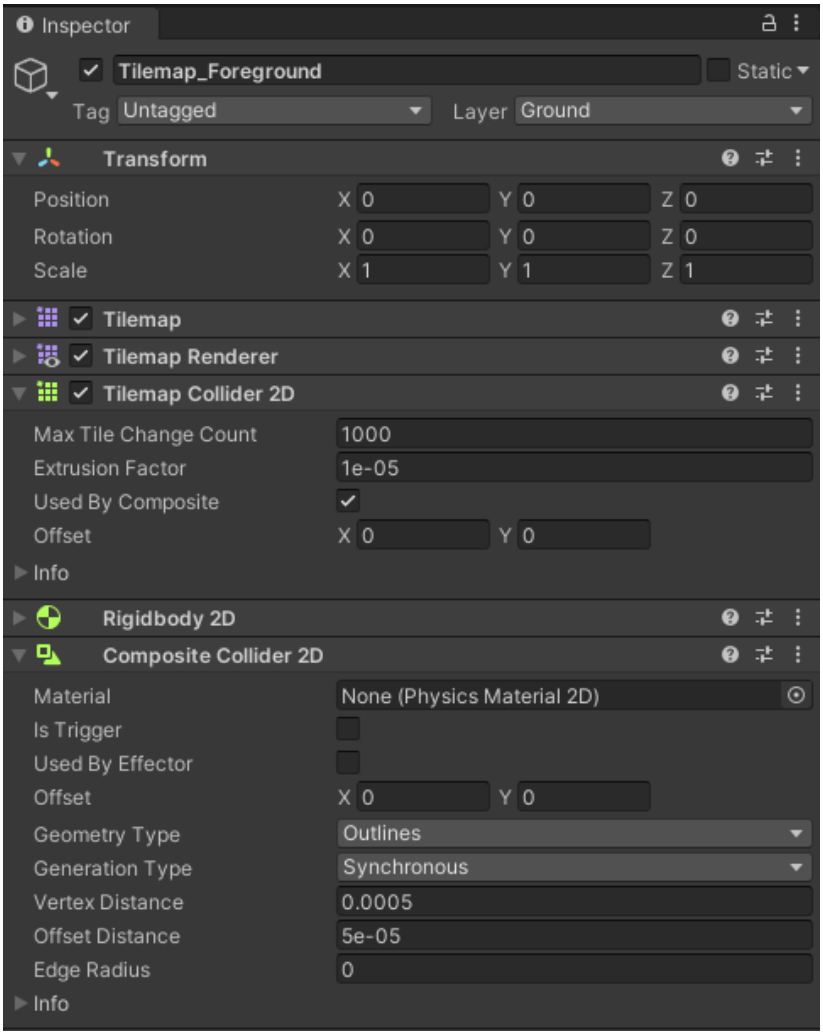
Slika 57: Nov dodan if stavek v funkciji Update (avtorski posnetek zaslona)

V Jump funkciji smo nato vrednost jumpBufferCounter postavili na 0 in v funkcijo CanJump prav tako dodali enako spremenljivko.

```
private bool CanJump =>
    jumpBufferCounter > 0f && (hangTimeCounter > 0f || extraJumpValue > 0);
```

Slika 58: Dodajanje jumpBufferCounter spremenljivke v ukaz CanJump (avtorski posnetek zaslona)

S preizkusom smo potrdili delovanje kode in bili zadovoljni z napredkom. A znova smo med preizkusom zaznali novo napako. Zgodilo se nam je, da smo se med premikanjem levo in desno na tleh zaleteli v nevidno oviro. Tako nam je bilo nadaljnje premikanje onemogočeno. Težavo smo rešili kar v Unityju in sicer v mreži Tilemap_Foreground. Dodali smo možnost Composite Colider 2D ter v možnosti Tilemap Colider 2D odkljukali Used By Composite. Do rešitve nam je pomagal odličen programer, ki nam je s svojimi podobnimi izkušnjami zagotovil delujočo rešitev.



Slika 59: Sprememba v mreži Tilemap_Foreground (avtorski posnetek zaslona)

Opazili smo novo napako v programu. Ko je lik skočil na mestu, je skočil nižje, kot bi skočil med premikanjem v eno ali drugo smer. Ta problem smo lahko enostavno rešili. Funkcijo ApplyAirLinearDrag smo dodali v Jump funkcijo in problem je bil rešen.

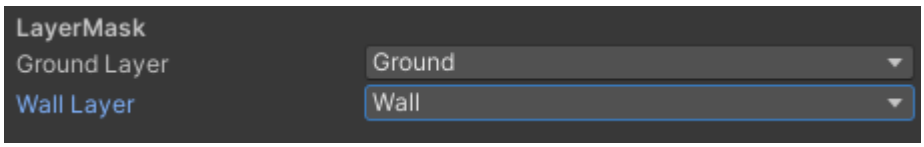
Z namenom, da bi igro še bolj popestrili, smo dodali tako imenovan »Wall movement«. Ta nam omogoči drsenje po zidu navzdol, odrivanje od zidu in nasploh hitrejšo pot do želenega cilja. V kodo programa smo dodali novo spremenljivko z imenom wallLayer. Da bi poenostavili preglednost vseh spremenljivk, smo dodali novo glavo programa, kjer bodo zbrane vse spremenljivke, ki imajo funkcijo v stenskem premikanju. Tja smo dodali spremenljivke wallRaycastLength, onWall in onRightWall. Vse tri spremenljivke smo ustrezno uporabili v funkcijah CheckCollisions in OnDrawGizmos.

```
//Wall Collisions
onWall = Physics2D.Raycast(transform.position, Vector2.right, wallRaycastLenght, wallLayer) ||
    Physics2D.Raycast(transform.position, Vector2.left, wallRaycastLenght, wallLayer);
onRightWall = Physics2D.Raycast(transform.position, Vector2.right, wallRaycastLenght, wallLayer);
```
Slika 60: Uporaba spremenljivk v stenskem premikanju (avtorski posnetek zaslona)

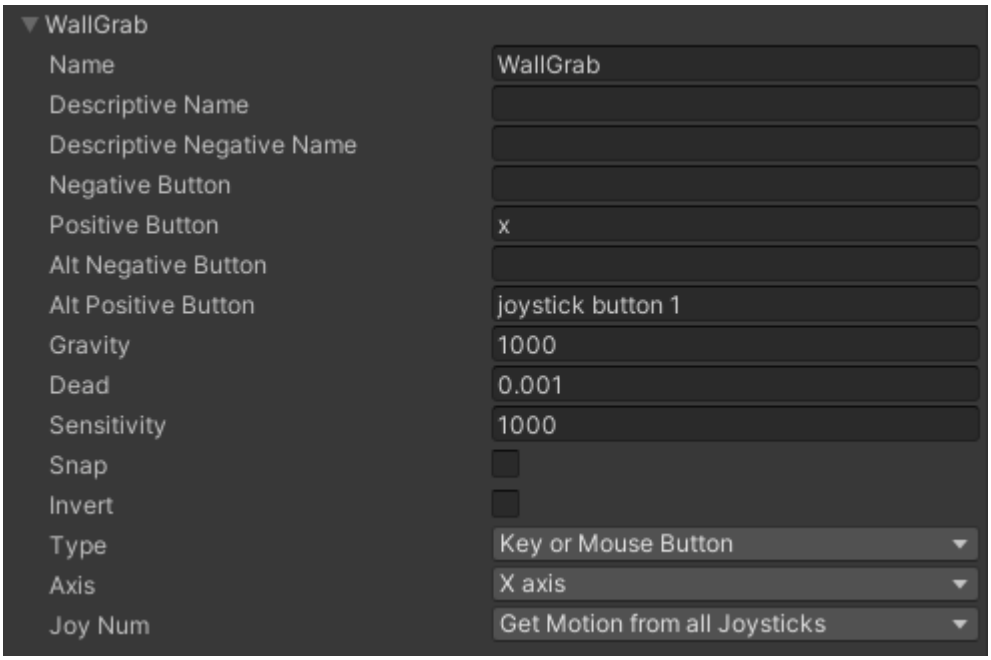
```
//Wall Check
Gizmos.DrawLine(transform.position, transform.position + Vector3.right * wallRaycastLenght);
Gizmos.DrawLine(transform.position, transform.position + Vector3.left * wallRaycastLenght);
```

Slika 61: Uporaba spremenljivk v funkciji OnDrawGizmos (avtorski posnetek zaslona)

Nato smo v programu Unity pod nastavitvijo Input Manager ustvarili novo možnost imenovano WallGrab. Še prej smo kot stensko plast v Unityju nastavili steno.



Slika 62: Nastavitev nove plasti Wall Layer (avtorski posnetek zaslona)



Slika 63: Nova možnost, imenovana WallGrab (avtorski posnetek zaslona)

V kodi smo dodali novo spremenljivko wallGrab in ji po navodilih iz video vodiča napisali ukaz. Da se lik sploh lahko prime za steno, smo morali v funkcijo FixedUpdate dodati stavek if, ki konstantno preverja, ali je lik na steni ali ni.

```
private bool wallGrab => onWall && !onGround && Input.GetButton("WallGrab");
```

Slika 64: Nova spremenljivka wallGrab (avtorski posnetek zaslona)

Za preverjanje, ali je lik na desni oziroma levi steni, smo morali dodati novo funkcijo imenovano StickToWall. Z ukazom, ki ga ta funkcija vsebuje, omogočimo liku spuščanje po levi ali desni steni.

```
private void StickToWall()
{
    if(onRightWall && horizontalDirection >= 0f)
    {
        rb.velocity = new Vector2(1f, rb.velocity.y);
    }
    else if(!onRightWall && horizontalDirection <= 0f)
    {
        rb.velocity = new Vector2(-1f, rb.velocity.y);
    }

}
```

Slika 65: Funkcija StickToWall (avtorski posnetek zaslona)

Posledično smo morali dodati še funkcijo, ki bo omogočala liku, da se samo obdrži na steni, ne da bi se premikal. Imenovali smo jo WallGrab.

```
private void WallGrab()
{
    rb.gravityScale = 0f;
    rb.velocity = new Vector2(rb.velocity.x, 0f);
    StickToWall();
}
```

Slika 66: Funkcija WallGrab (avtorski posnetek zaslona)

Da bi lik sploh lahko zdrsel po steni navzdol, smo dodali tudi novo spremenljivko, ki mu bo to omogočala. Ukaz v bistvu pove, da lik lahko začne drseti po steni navzdol, če je le dovolj visoko nad tlemi oziroma se tal ne dotika. Tudi to spremenljivko smo morali dodati v if stavku funkcije FixedUpdate.

```
private bool wallSlide =>
    onWall && !onGround && !Input.GetButton("WallGrab") && rb.velocity.y < 0f;
```

Slika 67: Spremenljivka wallSlide z njenim ukazom (avtorski posnetek zaslona)

Sami bi s programiranjem te kode imeli veliko težav, zato smo kodo pisali s pomočjo video vodičev ter z uporabno razlago. Seveda nas je vse zanimalo, kaj kateri del kode izvede. Naslednji korak v programiranju je bil dodajanje še dveh spremenljivk v glavo. Ti se imenujeta wallRunModifier in wallSlideModifier. Slednjo spremenljivko smo morali dodati v funkcijo WallSlide.

```
private void WallSlide()
{
    rb.velocity = new Vector2(rb.velocity.x, -maxMoveSpeed * wallSlideModifier);
    StickToWall();
}
```

Slika 68: Funkcija WallSlide (avtorski posnetek zaslona)

Osnove drsenja po steni navzdol so bile končane, zato smo se lahko lotili naslednjega koraka, in sicer teka po steni navzgor. Začeli smo z ustvarjenjem nove spremenljivke in njenim ukazom, ki smo ga spet morali vključiti v novo funkcijo kot tudi v funkcijo FixedUpdate za sprotno preverjanje.

```
private bool wallGrab =>
    onWall && !onGround && Input.GetButton("WallGrab") && !wallRun;
private bool wallSlide =>
    onWall && !onGround && !Input.GetButton("WallGrab") && rb.velocity.y < 0f && !wallRun;
private bool wallRun => onWall && verticalDirection > 0f;
```

Slika 69: Spremenljivke za premikanje po steni (avtorski posnetek zaslona)

```
private void WallRun()
{
    rb.velocity = new Vector2(rb.velocity.x, verticalDirection * maxMoveSpeed * wallRunModifier);
    StickToWall();
}
```

Slika 70: Funkcija WallRun s novo spremenljivko verticalDirection (avtorski posnetek zaslona)

S tem smo drsenje in tek po zidu končali. Sedaj smo se lotili težjega dela in sicer skoka s stene. Ustvarili smo novo spremenljivko z imenom isJumping in ji vrednost nastavili na »false«. Nato smo morali spremeniti funkcijo Jump in ji dodati nove ukaze.

```
private void Jump(Vector2 direction)
{
    if (!onGround && !onWall) {
        extraJumpValue--;
    }
    ApplyAirLinearDrag();
    rb.velocity = new Vector2(rb.velocity.x, 0f);
    rb.AddForce(direction * jumpForce, ForceMode2D.Impulse);
    hangTimeCounter = 0f;
    jumpBufferCounter = 0f;
    isJumping = true;
}
```

Slika 71: Spremenjena funkcija Jump (avtorski posnetek zaslona)

Kot smo predvidevali, smo morali ustvariti novo funkcijo WallJump, ji dodati ukaz, v katero smer naj lik skoči glede na položaj stene (če je na levi steni, naj skoči v desno in obratno). Na koncu funkcije pokličemo še funkcijo Jump, ki nam omogoča odriv od stene.

```
private void WallJump()
{
    Vector2 jumpDirection = onRightWall ? Vector2.left : Vector2.right;
    Jump(Vector2.up + jumpDirection);
}
```

Slika 72: Nova funkcija WallJump (avtorski posnetek zaslona)

Da bodo vsi ukazi delovali pravilno in se preverjali vsako sličico na sekundo, smo v funkcijo FixedUpdate morali dodati vse na novo ustvarjene spremenljivke ter poklicati vse funkcije.

```
if (onGround)
{
    extraJumpValue = extraJumps;
    ApplyGroundLinearDrag();
    hangTimeCounter = hangTime;
}
else
{
    ApplyAirLinearDrag();
    FallMultiplier();
    hangTimeCounter -= Time.fixedDeltaTime;
    if(!onWall || rb.velocity.y < 0f)
    {
        isJumping = false;
    }
}
```

Slika 73: Dodajanje stavka else za odskok od stene (avtorski posnetek zaslona)

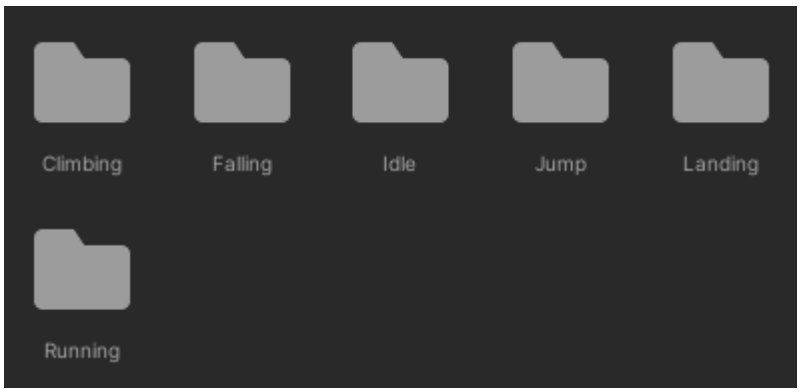
```
if (CanJump)
{
    if(onWall && !onGround)
    {
        WallJump();
    }
    else
    {
        Jump(Vector2.up);
    }
}
```

Slika 74: If stavek s klicem funkcije WallJump (avtorski posnetek zaslona)

```
if(!isJumping)
{
    if (wallGrab) WallGrab();
    if (wallSlide) WallSlide();
    if (wallRun) WallRun();
    if (onWall) StickToWall();
}
```

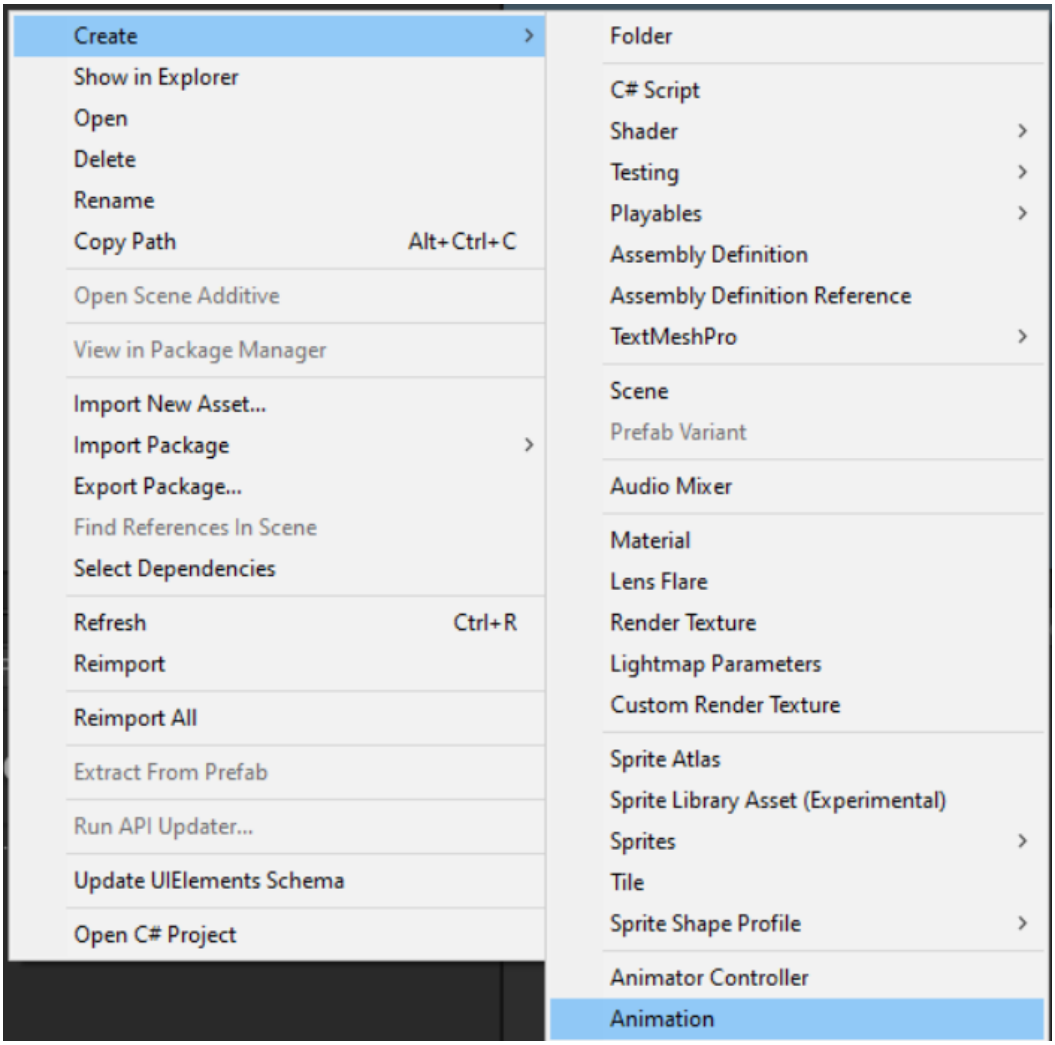
Slika 75: Dodajanje vseh funkcij v if stavek (avtorski posnetek zaslona)

Na tej točki smo videli, da je delovanje »wall movementa« zadovoljivo za naš standard programiranja. Bili smo zadovoljni, da nam je to uspelo, saj je bilo razumevanje nekaterih delov kode precej težko. Vedeli smo, da je konec programiranja že blizu. Zadnja ovira, ki nas je pričakala, so bile animacije. Animacije so za izgled igre zelo pomembne. Igralcu močno izboljšajo igralno izkušnjo in ga približajo igri. V programskem okolju Unity smo ustvarili več podmap in jim dodelili ustrezna imena glede na animacijo.

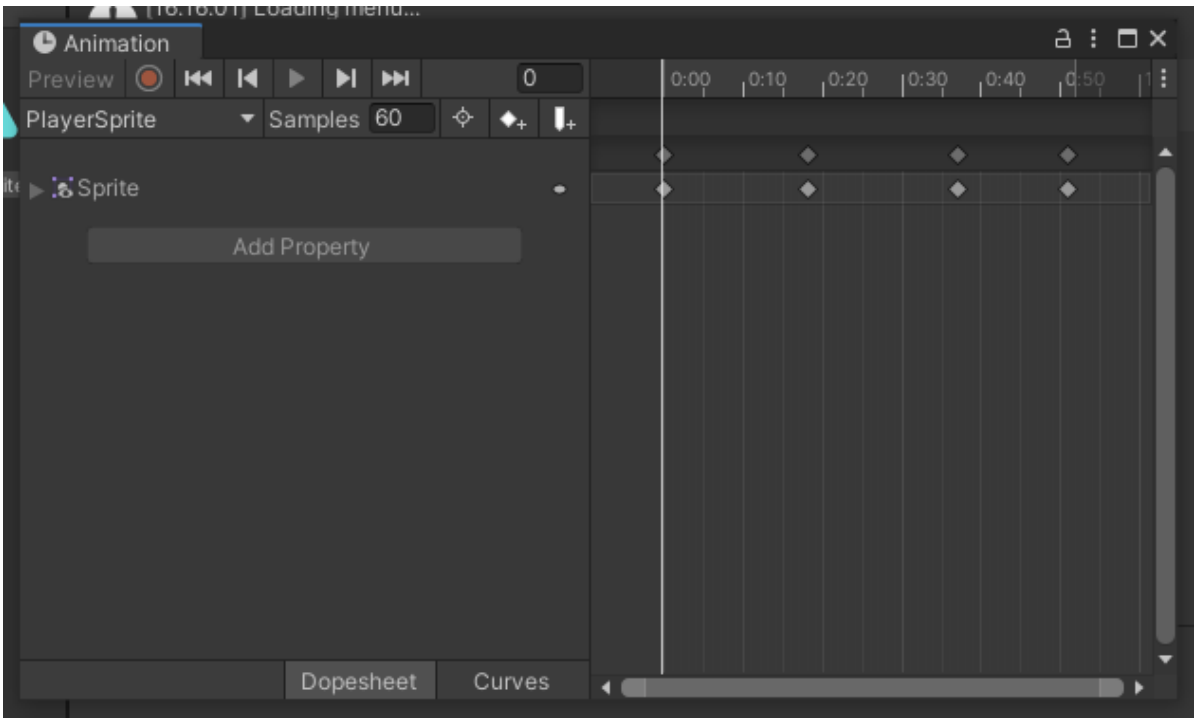


Slika 76: Ustvarjene podmape za animacije (avtorski posnetek zaslona)

Ko smo mape ustvarili, smo jim dodali ustrezne sličice, ki bodo prikazovale posamezne animacije. Sličice smo ustvarili že pred začetkom programiranja igre. Unity ima zelo priročno funkcijo, ki nam omogoča, da animacije naredimo že v samem programskem okolju. Da smo lahko dostopali do tega ustvarjanja animacije, smo morali v mapo dodati posebno orodje, ki nam bi to omogočalo. Dodali smo ga tako, da smo v mapi z desnim klikom odprli možnosti in izbrali Animation. Ko se nam je v mapi ustvaril okvirček z imenom New Animation, smo ga morali povleči v objekt Player. To pomeni, da se je lahko animacija izvajala med igro.

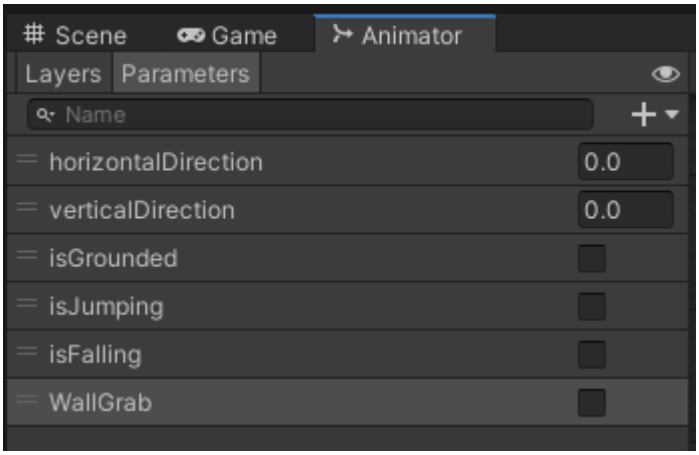


Slika 77: Postopek ustvarjanja animacije (avtorski posnetek zaslona)

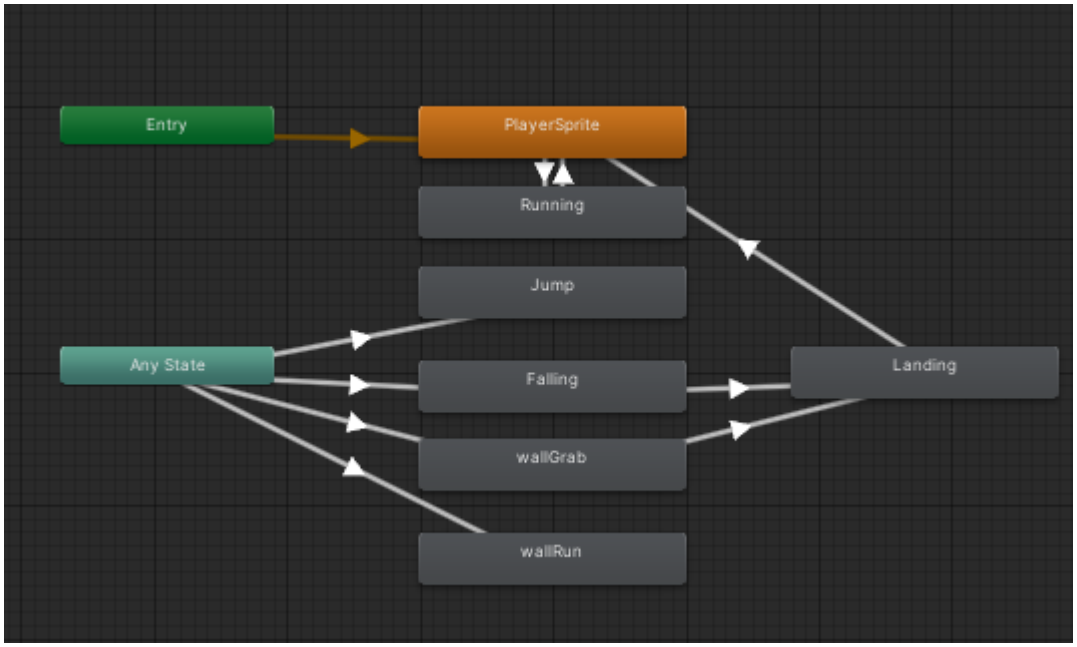


Slika 78: Okno, v katerem ustvarjamo animacijo (avtorski posnetek zaslona)

V vsaki podmapi smo ustvarili po eno animacijo iz sličic, le v podmapi Climbing smo morali ustvariti dve (eno za vzpenjanje in eno za spuščanje). Ko smo imeli animacije končane, smo jih morali med seboj povezati. To smo storili s pomočjo animatorja, orodja v okolju Unity. Animator nam je pomagal, da smo lahko določili, ob katerih pogojih se izvede oziroma izvaja animacija. Smiselno smo jih med seboj povezali in jim dodali parametre z določenimi vrednostmi.



Slika 79: Parametri animatorja (avtorski posnetek zaslona)



Slika 80: Povezave v animatorju (avtorski posnetek zaslona)

Na sliki je razvidno, da ima lik animacijo mirujočega stanja. Ko se lik premakne, se animacija spremeni v animacijo teka in obratno, če se lik ustavi. Iz kateregakoli premika lahko lik preide v animacijo skoka, padanja, držanja stene in teka po steni. Animaciji padanja in držanja stene pa vsebujeta tudi posledično animacijo pristanka. Vidimo, da se iz te zadnje animacije pristanka glede na premikanje animacija znova lahko spremeni.

Ko smo končali z nastavljanjem animatorja, smo se posvetili kodi. Opazili smo, da se lik med skokom s stene vizualno ne obrne v drugo smer, zato smo dodali funkcijo Flip, ki to stori. Funkcija omogoči, da se lik kar najhitreje obrne v drugo smer in gibanje nadaljuje.

```
private void Flip()
{
    facingRight = !facingRight;
    transform.Rotate(0f, 180f, 0f);
}
```

Slika 81: Funkcija flip (avtorski posnetek zaslona)

Zatem smo se lotili programiranja animacij. Da smo lahko dostopali do nove funkcije Animation, smo morali dovoliti dostop do te funkcije. Deklarirati smo morali komponento Animator z imenom anim. Ker je to privatna komponenta, smo jo morali vključiti v funkcijo Start, da smo si omogočili dostop do funkcije Animation. Kot prvo smo morali poklicati funkcijo Flip, ki se je izvedla pod določenimi pogoji.

```
private void Animation()
{
    if((horizontalDirection < 0f && facingRight || horizontalDirection > 0f && !facingRight)
        && !wallGrab && !wallSlide)
    {
        Flip();
    }
```

Slika 82: Klic funkcije Flip v funkciji Animation (avtorski posnetek zaslona)

V nadaljevanju pisanja kode smo se osredotočili na programiranje animacij. Kode ni bilo težko razumeti, saj je njena funkcija enaka funkciji animatorja v Unityju.

```
if(onGround)
{
    anim.SetBool("isGrounded", true);
    anim.SetBool("isFalling", false);
    anim.SetBool("WallGrab", false);
    anim.SetFloat("horizontalDirection", Mathf.Abs(horizontalDirection));
}
else
{
    anim.SetBool("isGrounded", false);
}
```

Slika 83: If stavek za animacije na tleh (avtorski posnetek zaslona)

```
if(isJumping)
{
    anim.SetBool("isJumping", true);
    anim.SetBool("isFalling", false);
    anim.SetBool("WallGrab", false);
    anim.SetFloat("verticalDirection", 0f);
}
```

Slika 84: If stavek za animacijo skoka (avtorski posnetek zaslona)

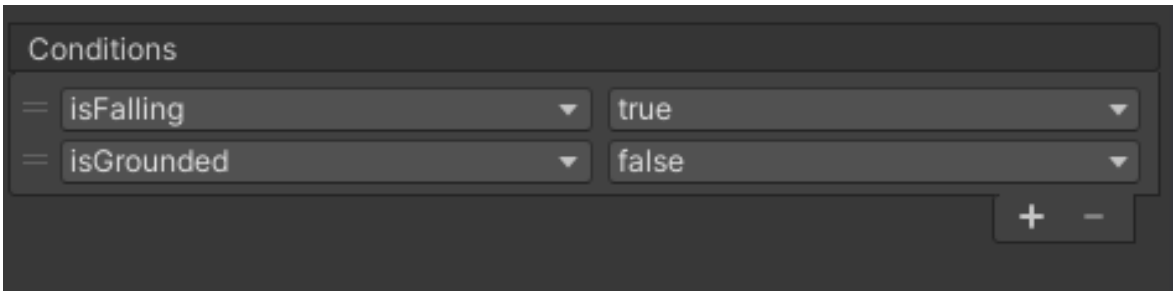
```
else
{
    anim.SetBool("isJumping", false);
    if(wallGrab || wallSlide)
    {
        anim.SetBool("WallGrab", true);
        anim.SetBool("isFalling", false);
        anim.SetFloat("verticalDirection", 0f);
    }
    else if(rb.velocity.y < 0f)
    {
        anim.SetBool("isFalling", true);
        anim.SetBool("WallGrab", false);
        anim.SetFloat("verticalDirection", 0f);
    }
    if(wallRun)
    {
        anim.SetBool("isFalling", false);
        anim.SetBool("WallGrab", false);
        anim.SetFloat("verticalDirection", Mathf.Abs(verticalDirection));
    }
}
```

Slika 85: Else stavek z več if stavki za preverjanje ostalih pogojev za animacije (avtorski posnetek zaslona)

Po zapisu te kode smo animacije preizkusili tudi v igri. Animacije so se nam zdele zelo dobre, saj so igro vizualno zelo izboljšale. Smo pa med preizkusom odkrili več napak, ki smo jih želeli odstraniti. Prvo napako smo opazili med mirovanjem lika. Animacija lika se je hitro naključno spreminjala med animacijo padanja in pristanka. Ugotovili smo, da moramo v animatorju stanju padanja dodati nov pogoj imenovan isGrounded in ga postaviti na vrednost »false«.

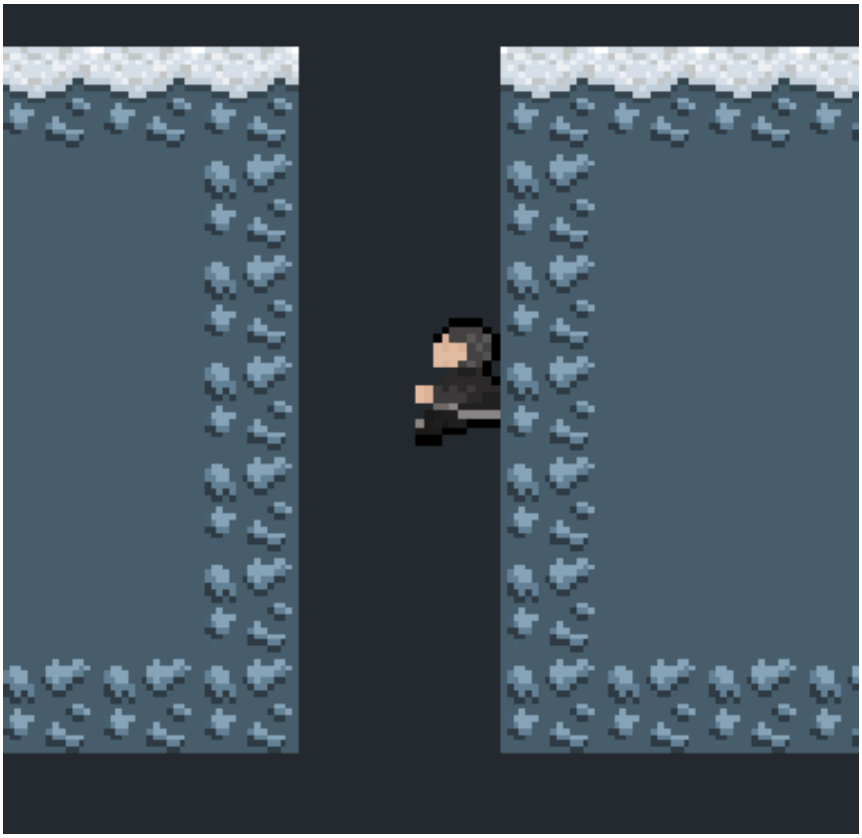
Slika 86: Falling animacija v stanju mirovanja (avtorski posnetek zaslona)

Slika 87: Landing animacija v stanju mirovanja (avtorski posnetek zaslona)



Slika 88: Nov dodan pogoj stanju Falling (avtorski posnetek zaslona)

Zadnja težava je bila v tem, da se je lik proti steni obrnil s svojim zadnjim delom oziroma hrbtom. Ker nas je to zmotilo, smo poiskali rešitev. Našli smo jo v dveh if stavkih, ki smo ju dodali funkciji StickToWall.



Slika 89: Druga težava pri animacijah (avtorski posnetek zaslona)

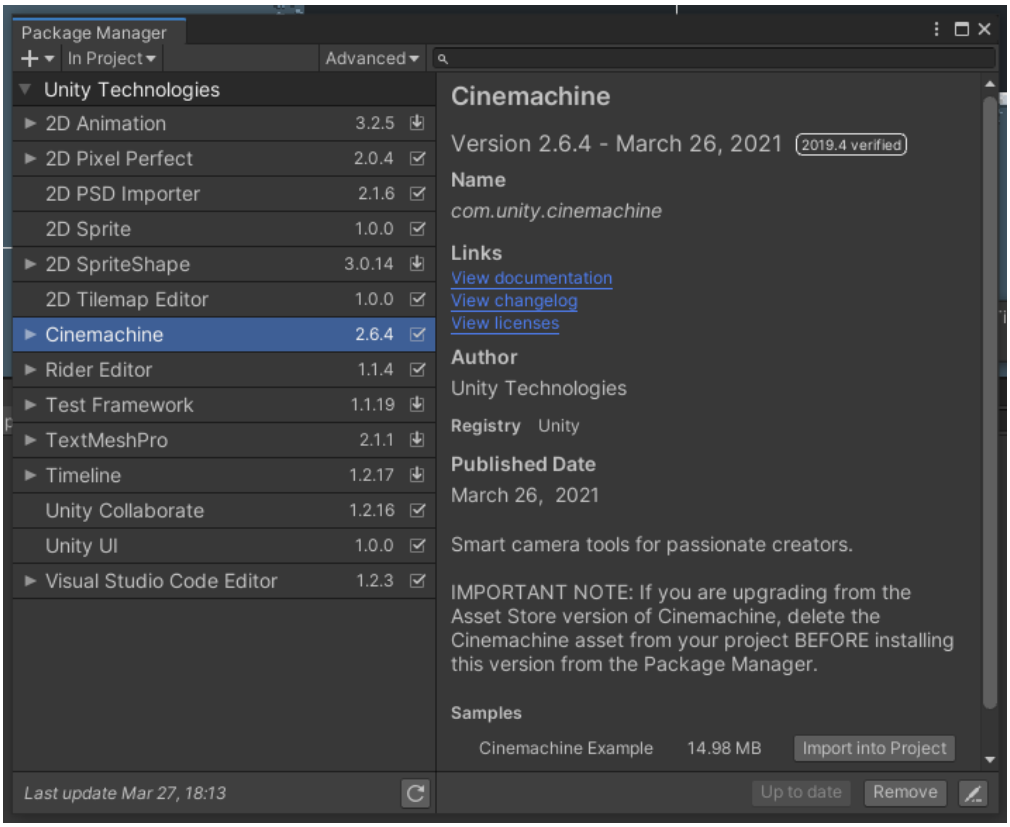
```
if(onRightWall && !facingRight)
{
    Flip();
}
else if(!onRightWall && facingRight)
{
    Flip();
}
```

Slika 90: If stavek za popravo druge težve pri animacija (avtorski posnetek zaslona)

Ta dva popravka sta bila zadnja v povezavi s premikanjem lika in animacijami. Bili smo zelo zadovoljni s končnim izgledom in občutkom premikanja igralca.

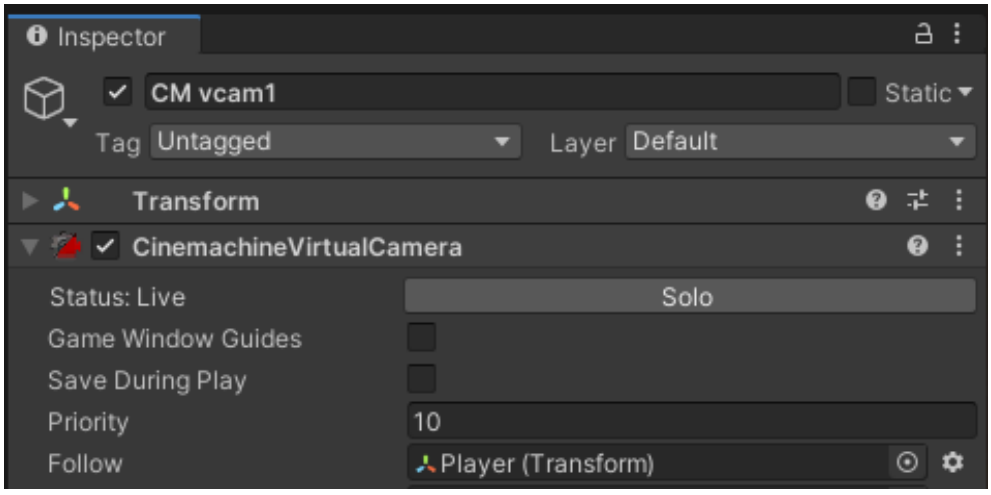
1.4.4.2 Kamera in pogled

Pri izbiri kamere in pogleda na igro smo si bili enotni. Vsi smo se strinjali, da mora biti kamera postavljena na liku in ne na posameznih stopnjah. Pozicija kamere je za igro izjemnega pomena, saj lahko s spreminjanjem pozicije vplivamo na dojemanje igre. Okolje Unity kamero že ima, a z njo nismo bili zadovoljni. Preko Unityja smo pridobili izboljšano kamero z več možnostmi, ki smo jih lahko poljubno spreminjali.

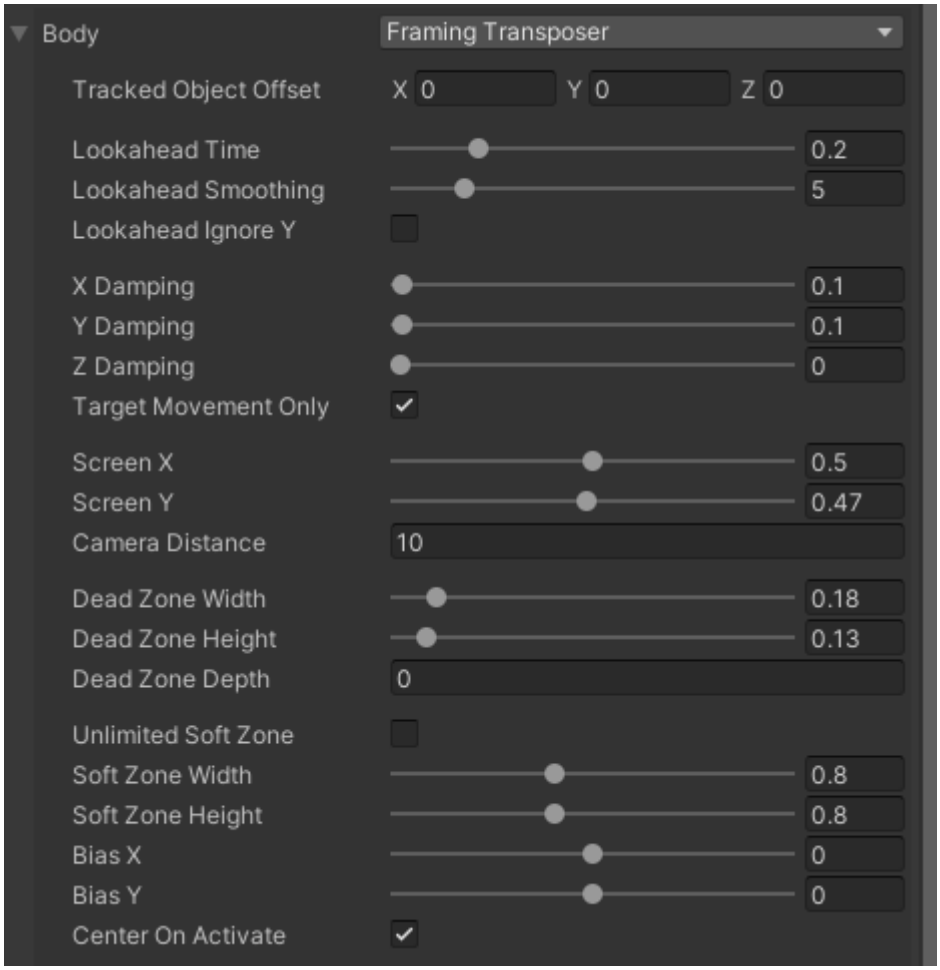


Slika 91: Izbira kamere (avtorski posnetek zaslona)

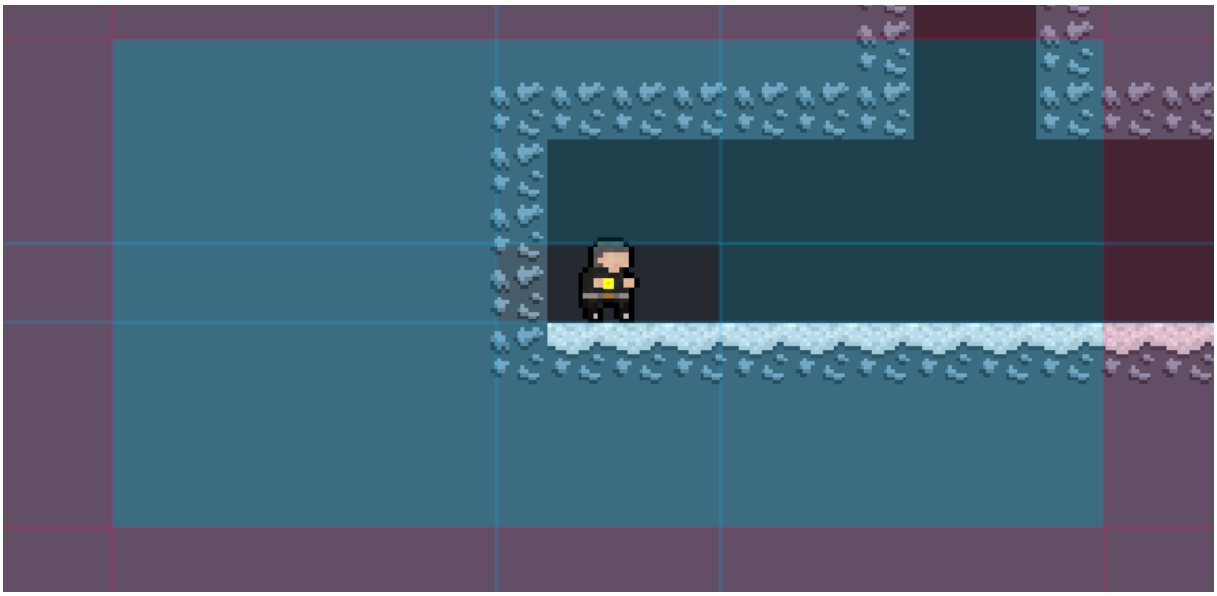
Ko smo kamero dodali v igro, smo začeli spreminjati njene nastavitve. Veliko časa smo potrebovali za spreminjanje in preizkušanje različnih nastavitev. Na spodnjih slikah lahko vidimo, kakšne so bile naše končne nastavitve.



Slika 92: Izbira sledenja kamere (avtorski posnetek zaslona)

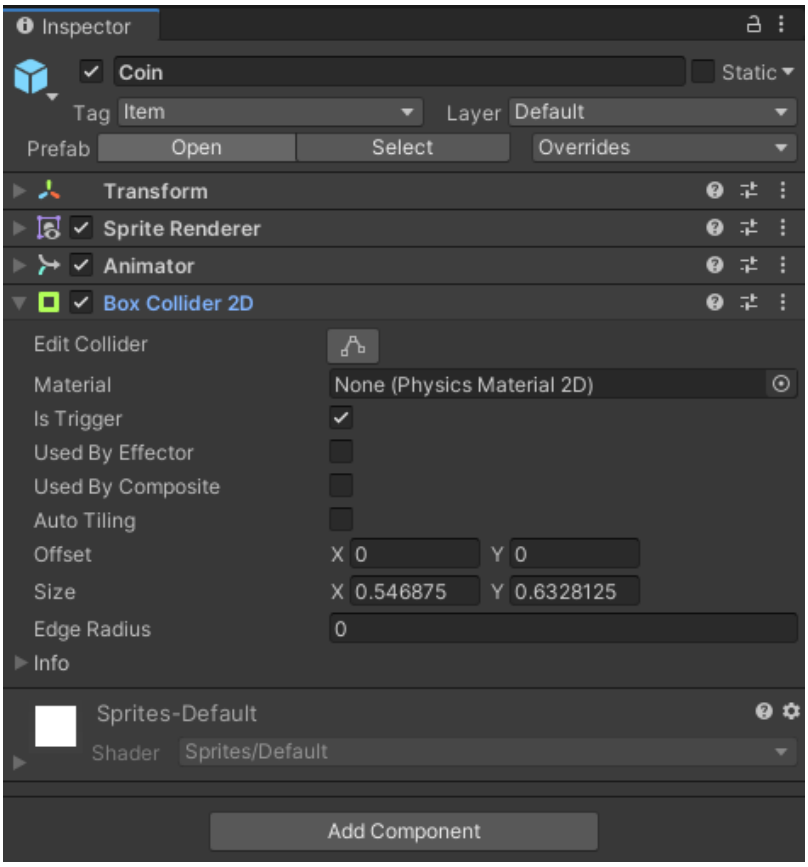


Slika 93: Nastavitve v telesu kamere (avtorski posnetek zaslona)


Slika 94: Pozicija kamere med igro (avtorski posnetek zaslona)

1.4.4.3 Zbirateljski predmet (kovanci)

Smiselno se nam je zdelo ustvariti predmet, ki ga lahko igralec med igro pobira. V našem primeru so to bili kovanci. V okolju Unity smo ustvarili nov igralni objekt in ga poimenovali Coin. V nastavitvah smo nastavili pozicijo kovanca in nato dodali tri komponente imenovane Sprite Renderer, Animator in Box Collider 2D. Najpomembneje je bilo, da smo v komponenti Box Collider 2D odkljukali možnost Is Trigger.

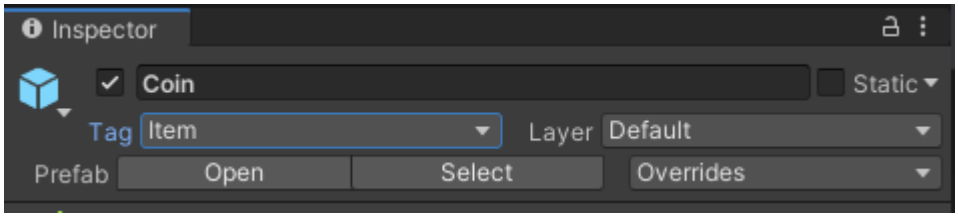


Slika 95: Nastavitve kovanca (avtorski posnetek zaslona)

Nato smo ustvarili mapo predmeti in v njej ustvarili animacijo vrtenja kovanca.

Slika 96: Animacije kovanca (avtorski posnetek zaslona)

Prav tako smo morali objektu Coin dodati novo oznako, katero lahko potem uporabimo v kodi. Kodo za delovanje smo napisali s pomočjo video vodiča. Ukaz v bistvu deluje tako, da se sproži, ko z likom vstopimo v bližino območja kovanca. Ta se uniči, posledično pa kovanec izgine iz igre.



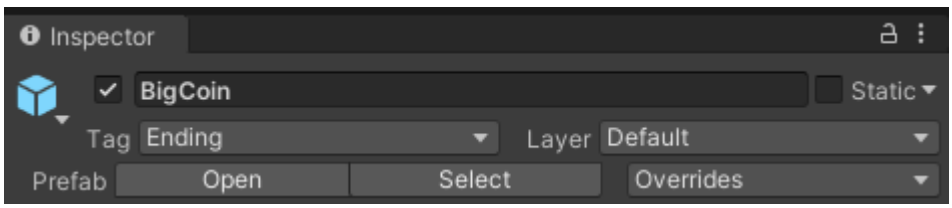
Slika 97: Dodajanje oznake objektu Coin (avtorski posnetek zaslona)

```
private void OnTriggerEnter2D(Collider2D collision)
{
    if(collision.tag == "Item")
    {
        Destroy(collision.gameObject);
        coin++;
    }
}
```

Slika 98: Funkcija za pobiranje kovanca (avtorski posnetek zaslona)

Število kovancev v igri ni določeno, zato smo lahko poljubno izbrali, koliko kovancev bo igra vsebovala. Da smo to naredili, smo ustvarili novo mapo imenovano PreFeb, ki nam je omogočila neskončno dodajanje kovancev v igro oziroma v novo mapo CollectableHolder. V enako mapo smo dodali še večji kovanec. Ko se je igralec dotaknil tega velikega kovanca, je to pomenilo konec igre.

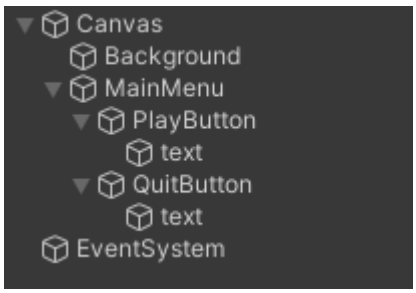
Slika 99: Mapa PreFab (avtorski posnetek zaslona)



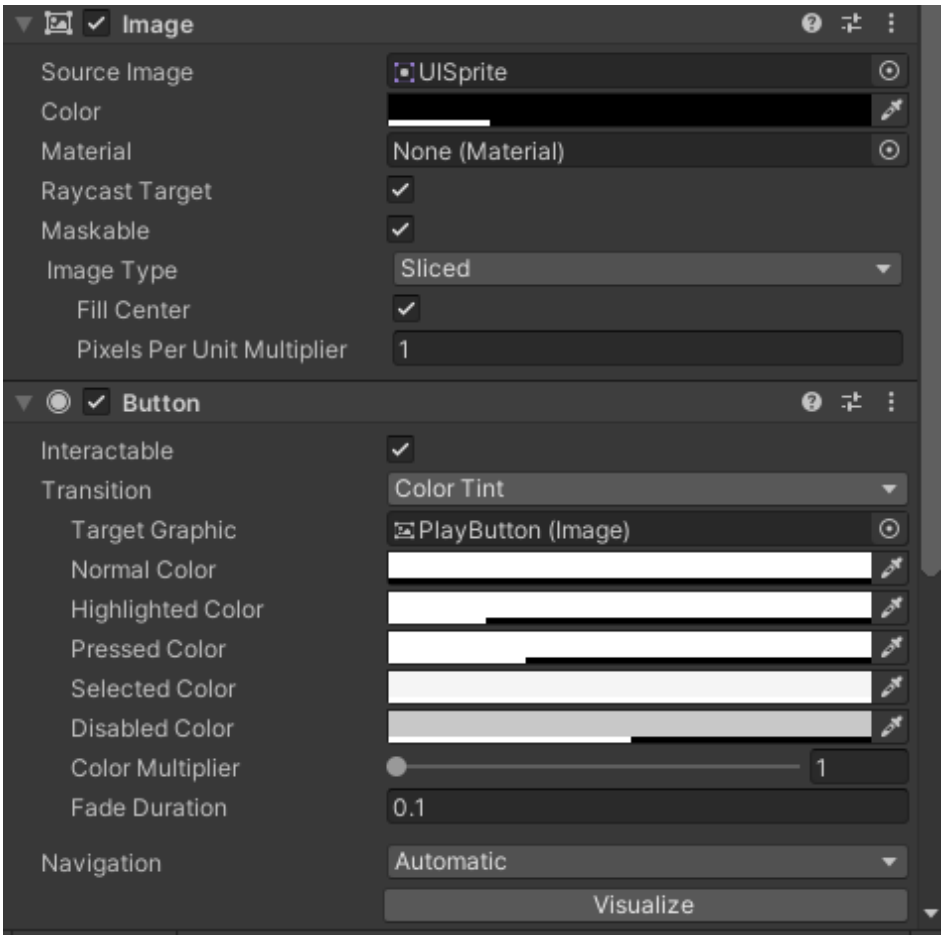
Slika 100: Novi objekt imenovan BigCoin (avtorski posnetek zaslona)

1.4.4.4 Meni

Da bi bila igra res zaključena, smo dodali še meni. Prvi meni je omogočal izbiro med igraj in zapusti igro. Dodali smo novo platno v igri. Platnu smo dodali objekt Main Menu, v njem pa ustvarili dva gumba imenovana PlayButton in QuitButton. Gumboma smo dodali posebne vizualne nastavitve, ki so nam ugajale.

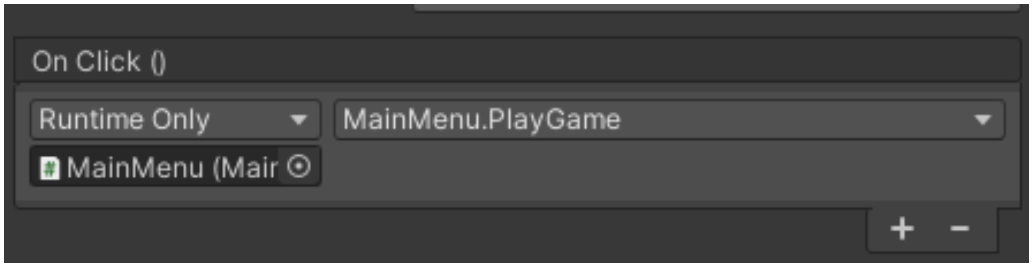


Slika 101: Vsi dodani objekti (avtorski posnetek zaslona)

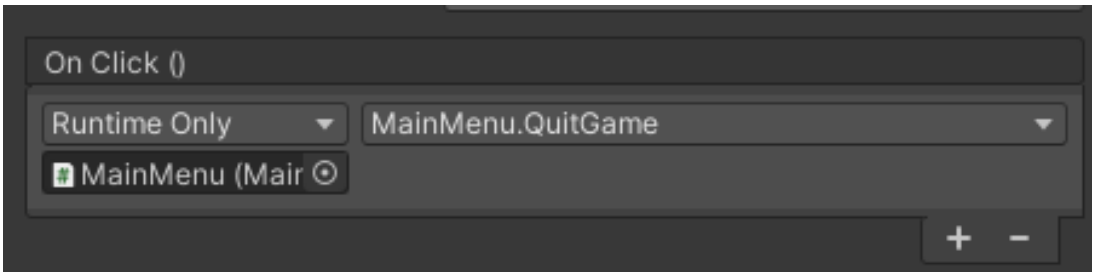


Slika 102: Nastavitve obeh gumbov (avtorski posnetek zaslona)

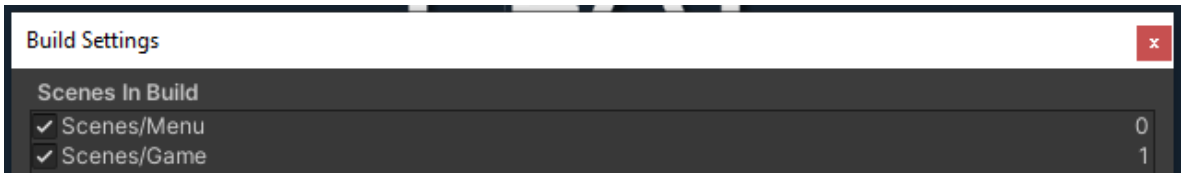
Ker pa nastavitve same po sebi ne delujejo, smo morali dodati še kodo za delovanje menija. Objektu Main Menu smo dodali skripto in jo poimenovali enako kot objekt. V nastavitvah gumba smo dodali gumbom učinke. V skripti smo napisali kodo, ki je omogočila igralcu zapustiti ali vstopiti v igro. Pred tem smo morali v Unityju označiti zaporedje odpiranja scen.



Slika 103: Posledica pritiska na gumb Play (avtorski posnetek zaslona)



Slika 104: Posledica pritiska na gumb Quit (avtorski posnetek zaslona)

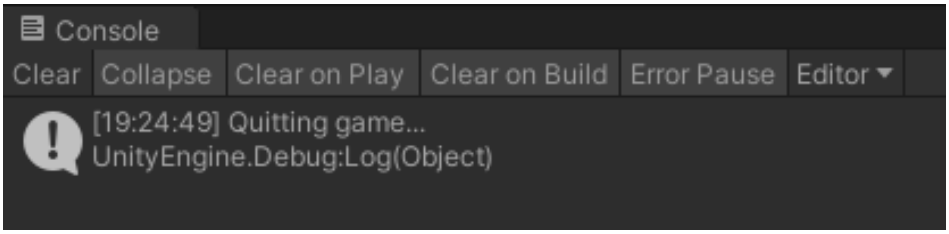


Slika 105: Zaporedje odpiranja scen (avtorski posnetek zaslona)

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class MainMenu : MonoBehaviour
{
    public void PlayGame()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
    }

    public void QuitGame()
    {
        Debug.Log("Quitting game...");
        Application.Quit();
    }
}
```

Slika 106: Koda za Main Menu (avtorski posnetek zaslona)

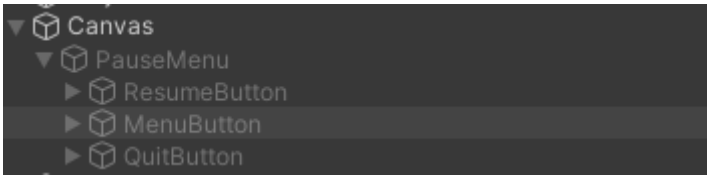
V kodi smo dodali funkciji imenovani PlayGame in QuitGame. Prva naloži sceno igre, druga pa zapre program. Da bi se program v aplikaciji res zaprl, nam v Unityju pove obvestilo v konzoli.



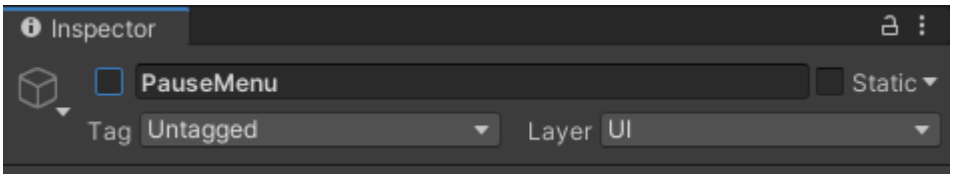
Slika 107: Obvestilo o uspešnem zaprtju programa (avtorski posnetek zaslona)

Slika 108: Končni izgled prvega menija (avtorski posnetek zaslona)

Drugi meni, ki smo ga naredili, se imenuje pavza. Meni se pojavi samo takrat, ko igralec pritisne gumb escape. Ustvarjanje vseh objektov in gumbov je bilo enako kot pri ustvarjanju prvega menija. Razlika je bila, da objekt PauseMenu ni bil označen kot viden, saj se je moral pojaviti samo ob pritisku na gumb.



Slika 109: Objekti in gumbi v Pause meniju (avtorski posnetek zaslona)



Slika 110: Ne označen PauseMenu (avtorski posnetek zaslona)

Spet smo objektu dodali novo skripto s kodo za ta meni. V kodi smo najprej dodali dve novi spremenljivki, in sicer GameIsPaused in pauseMenuUI. Nato smo v if stavku zapisali, kaj se zgodi, če igralec pritisne gumb escape.

```
public static bool GameIsPaused = false;
public GameObject pauseMenuUI;
```

Slika 111: Novi spremenljivki v Pause meniju (avtorski posnetek zaslona)

```
private void Update()
{
    if (Input.GetKeyDown(KeyCode.Escape))
    {
        if(GameIsPaused)
        {
            Resume();
        }
        else
        {
            Pause();
        }
    }
}
```

Slika 112: If stavek v funkciji Update za Pause meni (avtorski posnetek zaslona)

```
public void Resume()
{
    pauseMenuUI.SetActive(false);
    Time.timeScale = 1f;
    GameIsPaused = false;
}
```

Slika 113: Funkcija Resume (avtorski posnetek zaslona)

```
void Pause()
{
    pauseMenuUI.SetActive(true);
    Time.timeScale = 0f;
    GameIsPaused = true;
}
```

Slika 114: Funkcija Pause (avtorski posnetek zaslona)

```
public void LoadMenu()
{
    Time.timeScale = 1f;
    Debug.Log("Loading menu...");
    SceneManager.LoadScene("Menu");
}
```

Slika 115: Funkcija LoadMenu (avtorski posnetek zaslona)

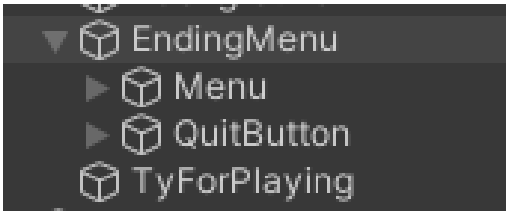
```
public void QuitGame()
{
    Debug.Log("Quitting game...");
    Application.Quit();
}
```

Slika 116: Funkcija QuitGame (avtorski posnetek zaslona)

Slika 117: Končni izgled drugega menija (avtorski posnetek zaslona)

Vsaka funkcija posebej izvede ukaz glede na to, kaj igralec pritisne na zaslonu. V funkciji Resume vidimo, da če igralec pritisne izbiro resume, se igra nadaljuje, v funkciji LoadMenu se odpre glavni meni in v funkciji QuitGame igralec igro zapusti.

Zadnji in tretji meni smo naredili po enakem postopku kot prejšnja dva. Spet smo dodali novo plast ter objekte in gumba z določenimi funkcijami. Ta meni se bo prikazal samo takrat, ko igro končamo.



Slika 118: Objekt in gumba v zadnjem meniju (avtorski posnetek zaslona)

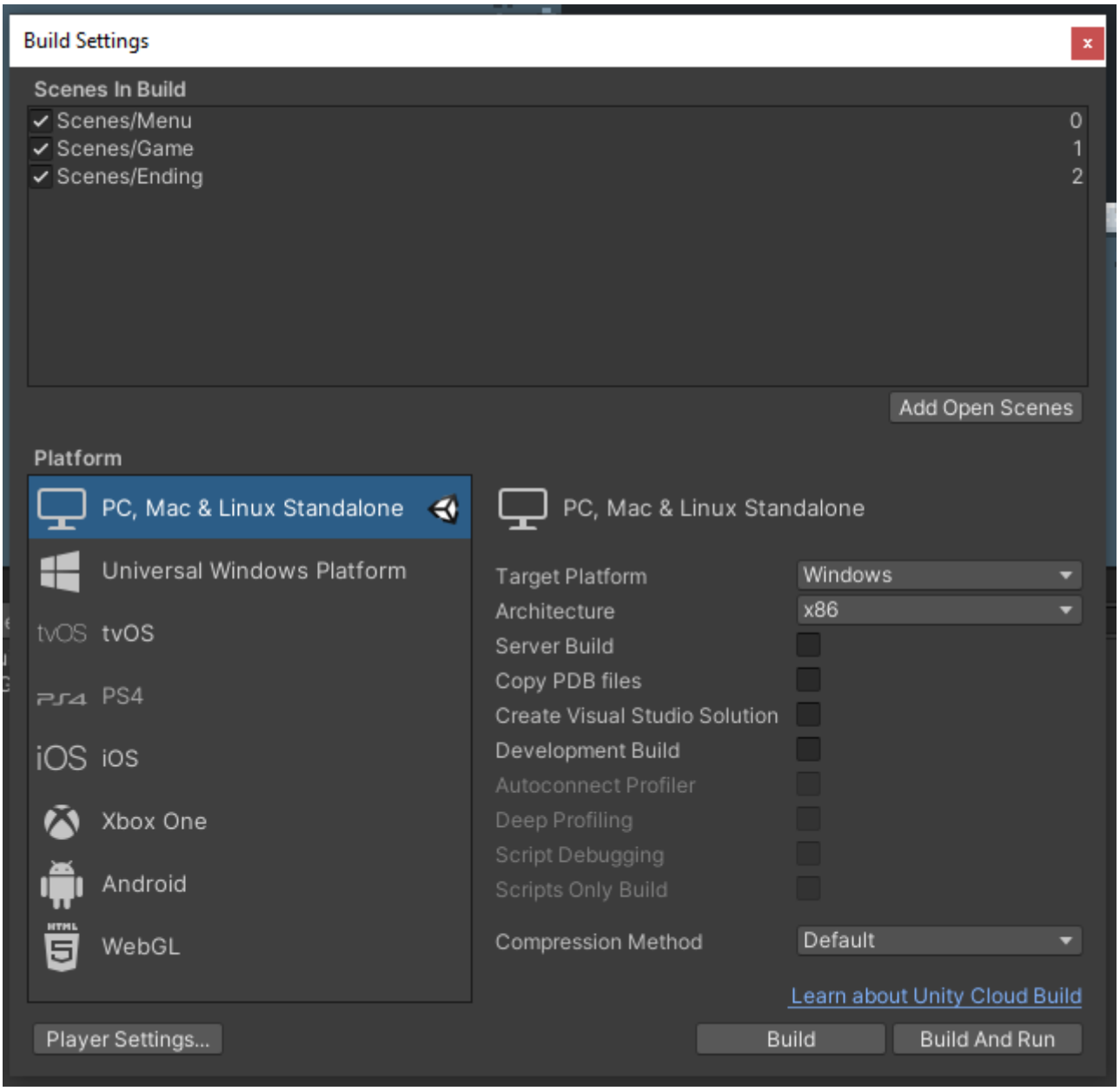
```
public void QuitGame()
{
    Debug.Log("Quitting game...");
    Application.Quit();
}

public void MainMenu()
{
    Debug.Log("Loading menu...");
    SceneManager.LoadScene("Menu");
}
```

Slika 119: Koda za zadnji meni (avtorski posnetek zaslona)

Funkcija kode zadnjega menija je enaka kot pri prejšnjih dveh. Ko igralec izbere možnost Main Menu, se mu odpre glavni meni, ko pa želi igro zapustiti, izbere možnost Quit Game.

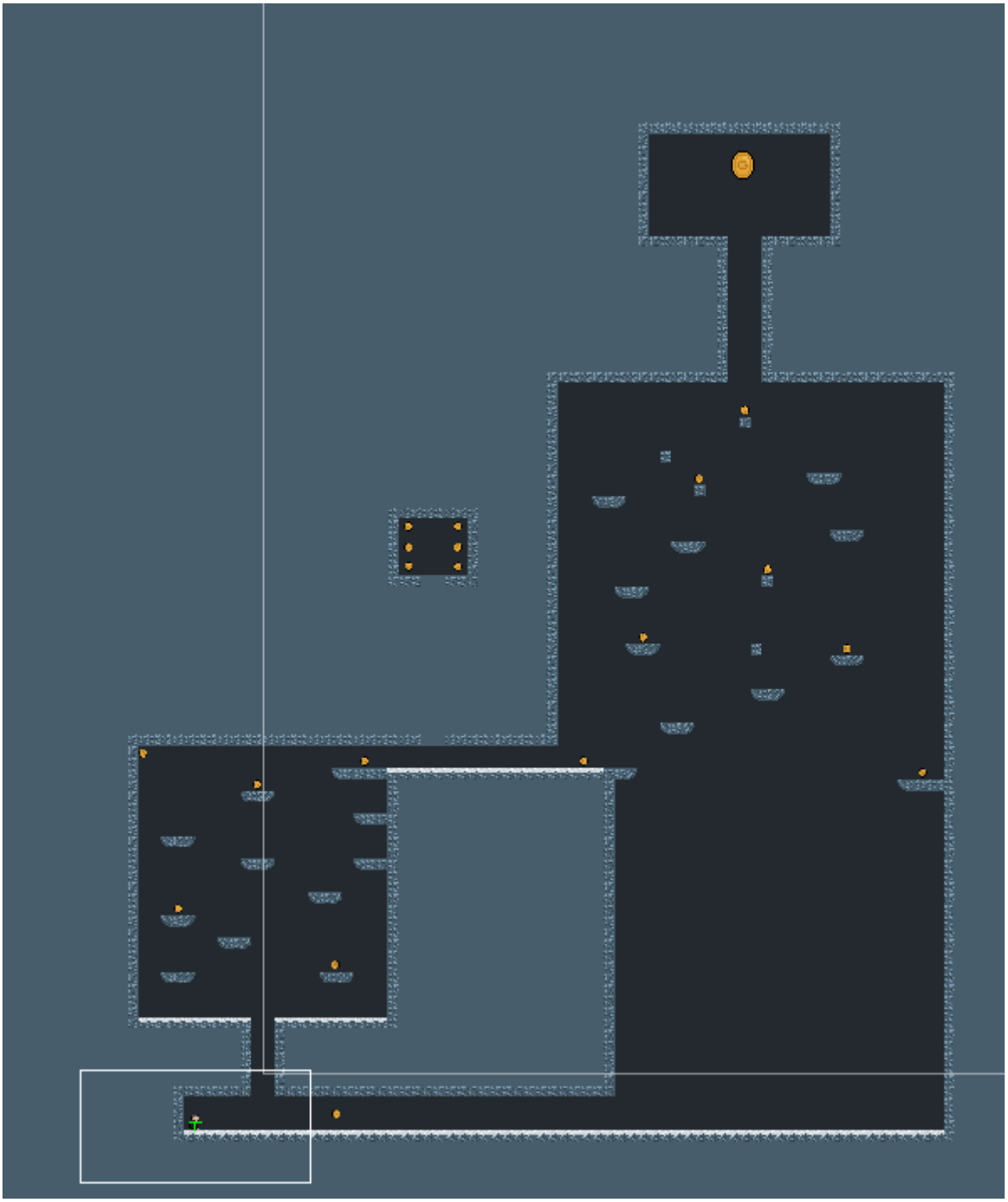
Slika 120: Končni izgled zadnjega menija (avtorski posnetek zaslona)

Zadnje dejanje ustvarjanja video igre je bil izvoz v datoteko tipa .exe. To smo storili tako, da smo v možnosti Build Settings izbrali platformo, na kateri hočemo, da igra deluje ter pritisnili izbiro Build and Run. Ta izbira nam je omogočila, da datoteko igre shranimo v poljubno mapo in jo nato zaženemo.



Slika 121: Izvoz igre v .exe verzijo (avtorski posnetek zaslona)

Zadnji korak pri ustvarjanju celotnega projekta je bil težko pričakovan, hkrati pa tudi zelo vesel trenutek za nas vse. Ne glede na vse ovire se nam je zaključna podoba zdela odlična, hkrati pa moramo povedati, da nam je zmanjkalo časa za izdelavo zvoka in zvočnih učinkov.



Slika 122: Končna podoba igre (avtorski posnetek zaslona)

2 DRUŽBENA ODGOVORNOST

V raziskovalni nalogi smo predstavili vse faze izdelave video igre. Menimo, da smo z našimi rezultati pokazali, kateri deli programiranja bodo ustvarjalcu povzročali težave, kako težko je ustvariti grafično podobo igre in koliko časa potrebujemo za ustvarjanje celotnega projekta. Raziskovalna naloga bi lahko imela določen vpliv na ljudi, ki se z razvijanjem video iger še niso ukvarjali, a bi se v tem želeli na novo preizkusiti.

3 ZAKLJUČEK

Za raziskovanje pravilnosti hipotez, ki smo si jih zadali na začetku, smo potrebovali veliko časa in energije. Na koncu lahko rečemo, da so rezultati raziskovalne naloge jasni. Menimo, da programiranje video igre zahteva svoj čas ter voljo. V ustvarjanju takšnega projekta te mora voditi želja po uspehu in dosegu cilja. Ugotovili smo, da smo za dokončanje celotnega projekta potrebovali 1 mesec s čimer lahko zagotovo potrdimo našo prvo hipotezo. Povemo lahko, da je poleg razvijanja igre naš čas vzela tudi šola in učenje. Zelo smo bili presenečeni nad okoljem Unity, saj je njegova uporaba za začetnike enostavna, vsebuje pa tudi veliko uporabnih funkcij, ki med programiranjem pridejo prav. Lahko se strinjamo, da igre v programskem okolju Unity ni enostavno narediti. Celoten projekt vsebuje veliko malenkosti, ki jih, če nisi pozoren, ne opaziš. Unity zahteva dobro povezovanje in razumevanje kodiranja ter funkcij posameznih delov kode, zato menimo, da lahko našo prvo hipotezo ovržemo. Vsak izkušen programer bo dejal, da je osnovno programiranje premikanja najenostavnejši del programiranja. Tukaj bi se z njim strinjali. Menimo, da vsak igričar v svetu ve, kako deluje princip premikanja, saj je ta bil znan tudi nam. Če na projekt pogledamo kot celoto, lahko rečemo, da so osnove premikanja (premik levo, premik desno in skok) najenostavnejši del za programiranje in razumevanje in s tem našo tretjo in tudi zadnjo hipotezo potrdimo.

1. Igro lahko izdelamo v enem mesecu. – POTRJENO

2. Video igro je v programskem okolju Unity enostavno narediti. - OVRŽENO

3. Osnove premikanje lika je najlažje sprogramirati. - POTRJENO

4 VIRI

4.1 Spletni viri

Kaj je Unity https://conceptartempire.com/what-is-unity/ (15.2.2021)

Kaj je Adobe Photoshop https://www.cbronline.com/what-is/what-is-adobe-photoshop-4984426/ (16.2.2021)

Kaj je Bosca Ceoil https://boscaceoil.net/ (16.2.2021)

Kako ustvariti pixel art https://medium.com/pixel-grimoire/how-to-start-making-pixel-art-2d1e31a5ceab (16.2.2021)

Zakaj uporabiti pixel art https://ecommerce-platforms.com/articles/why-pixel-art-can-be-a-perfect-training-ground-for-graphic-design (16.2.2021)

Unity Store https://store.unity.com/#plans-individual (22.2.2021)

Komplet vodičev za začetnike v okolju Unity https://www.youtube.com/playlist?list=PLpj8TZGNIBNy51EtRuyix-NYGmcfkNAuH (22.2.2020)

Kako ustvariti 2D platformo https://youtu.be/UbPiCgCkHTE (22.2.2021)

Premikanje lika v Unity-ju https://www.youtube.com/watch?v=dwcT-Dch0bA (22.2.2021)

Osnovna matematika v premikanju lika https://youtu.be/DPfxjQ6sqrc (22.2.2021)

Nasveti za kodiranje igre https://youtu.be/8QPmhDYn6rk (22.2.2021)

Višji skok lika https://youtu.be/j111eKN8sJw (22.2.2021)

Razlaga komponente Rigidbody2D https://docs.unity3d.com/ScriptReference/Rigidbody2D.html (26.3.2021)

Razlaga LayerMask-a https://docs.unity3d.com/ScriptReference/LayerMask.html (26.3.2021)

Razlaga spremenljivke Raycast https://docs.unity3d.com/ScriptReference/Physics.Raycast.html (26.3.2021)

Razlaga funkcije OnDrawGizmos https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnDrawGizmos.html (26.3.2021)

Rešitev za onemogočeno premikanje na tleh (nevidna ovira) https://gamedev.stackexchange.com/questions/168600/player-gets-stuck-on-edges-between-tilemapcollider2d-tiles (26.3.2021)

Wall Movement https://youtu.be/JIASeoOU274 (26.3.2021)

Kamera v Unity-ju https://www.youtube.com/watch?v=2jTY11Am0Ig (26.3.2021)

Start meni v Unity-ju https://www.youtube.com/watch?v=zc8ac_qUXQY (26.3.2021)

Pause meni v Unity-ju https://youtu.be/JivuXdrIHK0 (26.3.2021)

Unity Prefabs https://docs.unity3d.com/Manual/Prefabs.html (26.3.2021)

4.2 Slikovni viri

Slika 1: Unity Logo. (16.2.2021) Dostopno na: https://unity3d.com/files/images/ogimg.jpg
Slika 2: Adobe Photoshop Logo. (16.2.2021) Dostopno na: https://en.wikipedia.org/wiki/Adobe_Photoshop#/media/File:Adobe_Photoshop_CC_icon.svg
Slika 3: Bosca Ceoil Logo (16.2.2021) Dostopno na: https://boscaceoil.net/images/boscalogo.png
Slika 4: Prvi primer pixel arta (avtorski posnetek zaslona)
Slika 5: Drugi primer pixel arta (avtorski posnetek zaslona)
Slika 6: Grafična tablica. (17.2.2021) Dostopno na: https://wcm-cdn.wacom.com/-/media/images/products/pen-tablets/one-by-wacom-2017/one-by-wacom-g4.jpg?h=640&la=en&w=960&rev=ece0a73abb1e4b63a34a8a147dfe008f&hash=F43A63D93E61AFEEC4CE3163A0F7875F
Slika 7: Kvadratna mreža (17.2.2021) Dostopno na: https://ecommerce-platforms.com/wp-content/uploads/2016/11/doc88img03-1549926059.png
Slika 8: Končna oblika glavnega lika igre (avtorski posnetek zaslona)
Slika 9: Prvi primer otočka (avtorski posnetek zaslona)
Slika 10: Druga verzija končanega otočka (avtorski posnetek zaslona
Slika 11: Končan otoček (avtorski posnetek zaslona)
Slika 12: Primer scene (avtorski posnetek zaslona)
Slika 13: Druga pojavljena težava (avtorski posnetek zaslona)
Slika 14: Pot do rešitve prvega problema (avtorski posnetek zaslona)
Slika 15: Končna rešena težava (avtorski posnetek zaslona)
Slika 16: Poizkusna arena (avtorski posnetek zaslona)
Slika 17: Tilemap_Background (avtorski posnetek zaslona)
Slika 18: Tilemap_Foreground (avtorski posnetek zaslona)
Slika 19: Komponente za objekt Player (avtorski posnetek zaslona)
Slika 20: Začetna postavitev lika (avtorski posnetek zaslona)
Slika 21: Pot do rešitve drugega problema (avtorski posnetek zaslona)
Slika 22: Rešitev drugega problema (avtorski posnetek zaslona)
Slika 23: Primer glave v programu (avtorski posnetek zaslona)
Slika 24: Nastavitve vodoravne osi (avtorski posnetek zaslona)
Slika 25: Nastavitve navpične osi (avtorski posnetek zaslona)
Slika 26: Nastavitve za skok (avtorski posnetek zaslona)
Slika 27: Ukaz za vnos navpičnega in vodoravnega gibanja (avtorski posnetek zaslona)
Slika 28: Funckija start (avtorski posnetek zaslona)
Slika 29: Funkcija update (avtorski posnetek zaslona)
Slika 30: Funkcija MoveCharacter (avtorski posnetek zaslona)
Slika 31: Vrednosti spremenljivk za premikanje lika (avtorski posnetek zaslona)
Slika 32: If stavek prikazan v kodi (avtorski posnetek zaslona)
Slika 33: Funkcija ApplyGroundLinearDrag (avtorski posnetek zaslona)
Slika 34: Funkcija ChangingDirection (avtorski posnetek zaslona)
Slika 35: Dodajanje funkcije ChangingDirection (avtorski posnetek zaslona)
Slika 36: Funkcija Jump (avtorski posnetek zaslona)