Osnovne mehanike 2D platformera

Miketek, Stella

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Stella Miketek

Osnove mehanike 2D platformera

ZAVRŠNI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Stella Miketek

Matični broj:

Studij: Informacijski i poslovni sustavi

OSNOVE MEHANIKE 2D PLATFORMERA

ZAVRŠNI RAD

Mentor:

Doc. dr. sc. Mladen Konecki

Varaždin, lipanj 2024.

Stella Miketek

Izjava o izvornosti

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

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Ovaj rad temelji se na implementaciji osnovnih mehanika 2D platformera kroz pisani i praktični dio. Prikupljanje podataka potrebnih za izradu rada temeljit će se na sekundarnim izvorima podataka. U pisanome dijelu analiziraju se korišteni programski alati i pruža se uvid u njihovu primjenu. Slijedi detaljan opis računalne igre koja će biti razvijena, zajedno s algoritmima koji će biti implementirani za njezinu izradu. Praktični dio rada predstavlja funkcionalni 2D platformer izrađen u Unity-u i mnošto implementiranih mehanika kroz više razina. Rad se fokusira na primjenu teorije u praksi kroz konkretan razvoj videoigre, naglašavajući važnost teorijskog znanja u procesu razvoja. Kroz rad ističe se važnost daljnjeg istraživanja i unapređenja mehanika igara u svrhu stvaranja kvalitetnih korisničkih iskustava.

Ključne riječi: 2D platformer; Unity; računalna igra; algoritmi; mehanike; teorijska analiza; praktična implementacija

Sadržaj

1. Uvod

U današnjem digitalnom dobu računalne igre postaju sveprisutni oblik zabave i razonode. Osim toga, imaju i značajan utjecaj na našu kulturu i na naš način interakcije s tehnologijom. Žanr koji će se obrađivati, ali i izrađivati kroz ovaj rad je 2D platformer koji ostaje među najpopularnijim žanrovima i privlači širok spektar igrača – od onih rekreativnih (eng. *Casual*) do strastvenih (eng. *Hardcore)* ljubitelja videoigara. Razvoj igara zahtijeva duboko razumijevanje mehanika i tehnologija koje stoje iza istih.

Ovaj rad istražuje osnovne mehanike 2D plaformera kroz kombinaciju teorijskog i praktičnog pristupa. Korištenjem Unity platforme za razvoj računalnih igara, istražujemo ključne aspekte već spomenutog žanra – od fundamentalnih mehanika do praktične implementacije.

Osnovni cilj ovoga rada jest pružiti dublje razumijevanje karakteristika i zahtjeva te demonstrirati proces razvoja jednog 2D platformera kroz praktičnu implementaciju. Kroz analizu korištenih programskih alata, detaljan opis računalne igre, te integraciju algoritama u procesu razvoja gradi se teorijsko znanje koje može biti primijenjeno u stvaranju funkcionalnog i kvalitetnog iskustva za igrače.

Motivacija za ovu temu proizlazi iz vlastitog iskustva igranja računalnih igrica duži niz godina. Uživajući u igranju različitih igara, uvijek su me fascinirale mehanike i dinamika istih, te sam istovremeno osjećala izazov u prepoznavanju potencijala za poboljšanje igračkog iskustva. Stoga, iz želje za dubljim razumijevanjem procesa stvaranja igara i mogućnosti unaprjeđenja postojećih mehanika, odlučila sam kroz ovaj rad istražiti područje izrade 2D platformera i time obogatiti svoje znanje.

2. Metode i tehnike rada

U ovome poglavlju će biti detaljno opisane metode i tehnike korištene pri izradi računalne igre. Izrada rada temeljiti će se na kombinaciji teorijskog pristupa sa praktičnom implementacijom kako bi se postigao cjelovit uvid u temu.

Koristit će se analitički pristup prilikom istraživanja teorijskog okvira koji uključuje pregled literature, studija slučaja i analizu postojećih radova u području razvoja računalnih igara posebice 2D platformera. Analizom dostupnih materijala, bit će istraženi koncepti, tehnike i algoritmi koji se koriste u razvoju računalnih igara.

Za eksperimentalni pristup koji će se koristiti prilikom praktične implementacije koristit će se Unity platforma za razvoj videoigara. Kroz spomenutu platformu izvršit će se stvaranje funkcionalne 2D platformerske igre, a spomenuti pristup omogućit će dublje razumijevanje praktične primjene teorijskih koncepata u procesu stvaranja igre.

Od programskih alata koji će biti korišteni u istraživačkim aktivnostima imamo već spomenuti *Unity Game Engine* kao glavni alat za razvoj računalnih igara.

Kombinacija analitičkog i eksperimentalnog pristupa, zajedno s korištenjem odgovarajućih programskih alata, omogućit će detaljno istraživanje teme i postizanje ciljeva postavljenih u ovom istraživačkom radu.

.

3. Izrada 2D platformera u Unityju

Kroz ovu cjelinu teorijski će biti objašnjeni svi potrebni pojmovi i alati potrebni za shvaćanje koncepta, a zatim i za izradu računalne igrice u odabranom alatu – Unity. Prvo slijedi opis, svrha i povijesni razvoj *Unity* platforme za izradu videoigara, zatim će više biti rečeno o žanru 2D platformera – kako je nastao i kako se razvijao do onoga što danas predstavlja, ključne karakteristike i što ga čini popularnim. Za kraj, osvrnuti ćemo se na tipične mehanike koje se koriste u ovom žanru.

3.1. Unity

Unity je platforma već unaprijed pripremljena raznim alatima koji omogućuju izradu kvalitetnih videoigara čak i najvećim početnicima. Svatko ima mogućnost stvarati kreacije koje se natječu s velikim imenima jer su upravo i mnoge popularne igrice današnjice napravljene kroz ovu platformu.

3.1.1. Što je Unity?

Unity je zapravo 'motor za igre' (eng. *Game engine)* temeljen na Microsoftovom programskom jeziku C#, a služi za stvaranje 2D, 3D i drugih vrsta igara. Razvijen je od strane tvrtke *Unity Technologies* još u 2005. godini. (Lacoma, 2023.)

Podaci iz 2022. govore kako je mjesečno bilo preuzimano više od 4 milijarde igrica napravljenih upravo u *Unityju*. Od 1000 najboljih mobilnih igrica, čak 70% je napravljeno u *Unityju*, a *Unity Ads* stvorio je zaradu veću od 1.1 milijardi dolara za osnivače igara. (Technologies, 2024.)

Upravo jednostavnost korištenja pa čak i za nove korisnike je ono što je omogućilo ovakve brojke. Unity je jedno od najvećih i najpopularnijih imena koji se spominju kada je u pitanju izrada videoigara za više od 18 platformi koji podržavaju ovaj *game engine*.

Izrada igara pomoću *Unitya* je dovoljno moćna za sve vrste igara. Od društvenih igara kao što su *Fall Guys: Ultimate Knockout* i *Amoung Us*, 2D igara poput *Ori and the Will of the Wisps*, VR/AR igara poput *Pokemon Go* i *Beat Saber*, tu su i mnogi drugi žanrovi i mnoge druge poznate igre razvijene pomoću ovog diva. (Team, 2022.)

3.1.2. Povijest Unity-a

Od svojih skromnih početaka 2005. godine, Unity game engine je postao jedan od alata koji je postao standard u industriji razvoja računalnih igara. Osnovan s vizijom da omogući što većem broju developera da stvara i razvija videoigre, Unity Technologies je uspio ostvariti svoj cilj kroz kontinuirani razvoj i unaprjeđenje platforme.

Već u prvih par godina slijedi rast popularnosti, a nadolazeće verzije Unitya u 2007., zatim 2010. i 2012. godini proširuju mogućnosti ovog *game engine-a* u svakom smislu. Tako je *Unity 2.0* iz 2007. sadržavao 50 novih značajki od kojih su najvažnije bile optimizirani *engine* za terene, mogućnost raznim developerima da lakše surađuju na projektima i korištenje UDP-a (eng. *User Data Protocol)* za razvoj igara za više igara. U rujnu 2010. godine izlazi *Unity 3.0* s novitetima kao što su proširenje grafičkih značajki tako da se mogu koristiti na računalima i konzolama za videoigre, poboljšano UV mapiranje, audio filteri i mnoge druge mogućnosti. *Unity 4.0* iz 2012. godine uključuje *DirectX 11* i *Adobe Flash* podršku, novi alat za animacije *Mecanim*, te pristup *Linuxu*. (Fletch, 2020.)

Iz navedenih podataka posebno je značajno primijetiti da su u prvim kritičnim godinama razvoja, tvrtka *Unity Technologies* uspjela postići iznimno dobru osnovu za daljnji rast i razvoj. Kroz inovativne značajke, optimizaciju performansi i podršku za brojne platforme, *Unity* je zadobio povjerenje velikog broja developera širom svijeta. Iako se kontinuirano razvija i napreduje, važno je prepoznati da su temelji postavljeni u ranijim godinama razvoja bili ključni za uspjeh *Unityja* kao platforme koju poznajemo danas.

3.2. 2D Platformer

Najjednostavnije objašnjeno – 2D platformeri su igrice u trećem licu u kojima igrač kontrolira dvodimenzionalnog lika u dvodimenzionalnom prostoru. Kretanje je omogućeno u smjerovima naprijed i nazad odnosno lijevo i desno, te skok i pad, gore i dolje. Igrač se s likom kreće po blokovima – platformama i prolazi kroz razne prepreke, zamke i borbe s drugim likovima.

3.2.1. Povijest i razvoj platformera

Platformeri su jedan od najpopularnijih žanrova današnjice, no u jednom trenutku od svog nastajanja nosili su titulu najpopularnijeg žanra kada se procjenjivalo da je jedna trećina svih igara bila upravo ovoga tipa.

Prvi platformer pojavljuje se 1980. godine pod nazivom *Space Panic* koji je imao ljestve, ali ne i mogućnost skakanja. U godini nakon, 1981. izlazi *Donkey Kong* koji se smatra prvim

„pravim“ platformerom jer je imao mogućnost skakanja i preskakivanja preko prepreka. Uskoro izlazi i nastavak *Donkey Kong Jr*., a poslije i *Mario Bros* koji su nudili mogućnost igranja u dvoje. Veliku važnost nosi i igrica *Pitfall!* koja predstavlja prvu igru s nekoliko (256) povezanih ekrana i značajno pripomaže razvoju žanra. Na ovu ideju nastavljaju izlaziti igrice kao što su *Manic Miner* i njegov nastavak *Jet Set Willy*. U 1984. dodjeljena je prva nagrada za najbolju igricu ovog žanra – *Wanted: Monty Mole*. (Konecki, 2022.)

Prvi platformer sa takozvanim 'skrolanjem' ekrana izlazi još u 1981. godini pod nazivom *Jump Bug*. Razvoju žanra značajno doprinosi i *Pac-Land* koji izlazi 1984., a osim kretanja u dva smjera i skrolanja, nudio je novitete kao što su hodanje, skakanje, opruge, pojačanja (eng. *Power-ups)*, jedinstvene razine. Godinu nakon izlazi takozvani arheotip platformera *Super Mario Bros*, a osim tog platformera Nintendo već nadolazeće 2 godine izbacuje platformsku avanturu otvorenog svijeta *Metroid* (1986.) i uvodi nelinearan prelazak nivoa u platformeru *Mega Man* (1987.). Iduća generacija 2D skrolanja su *Super Mario World* (1990.) kao poznati klasik, a *Sonic the Hedgehog* (1991.) je uz navedeno koristio i sustav fizike te omogućavao brz prelazak nivoa. (Konecki, 2022.)

Novi korak u napretku ovog žanra nastao je dolaskom 3D i 2.5D igara. 2.5D igre koristile su 3D grafiku, a 2D mehanike za igranje. Dolaskom ovih noviteta, pada popularnost klasičnih 2D platformera, te na scenu nastupaju novi naslovi kao što su *Vectorman*, *Donkey Kong Country 2*, *Super Mario World 2*, *Clockwork Knight*, *Pandemonium*, *Klonoa* i mnogi drugi. U prvi plan stavljeni su 3D platformeri, no pojavljuju se i izometrični platformeri koji su omogućavali 3D igranje s 2D grafikom, a prvi izometrični platformer je *Congo Bongo* iz 1983. godine. *Alpha Waves* smatra se prvim pravim 3D platformerom, a izlazi 1990. Pravo iznenađenje dolazi s prvom igricom poznatog serijala *Crash Bandicoot* koji ostaje među najvećim imenima i dan danas. U 1996. godini izlazi *Super Mario 64* koji postavlja novi standard. Pratili su ga *Donkey Kong 64* i *Banjo-Kazooie* za koje je Nintendo bio pohvaljen za detaljnost nivoa i zagonetki. (Gaming, 2017.)

Iako su 3D igre postale dominirajući oblik, 2D platformeri nikada nisu potpuno nestali. Broji razvojni timovi i nostalgični igrači nastavljaju stvarati i igrati 2D platformere te time održavati životnost ovog žanra. Dolaskom digitalnih platformi kao što su mobilni uređaji i online trgovine za igre, omogućeno je širenje 2D platformera na novu publiku. Posljednjih godina vidimo ponovan rast popularnosti s pojavom brojnih uspješnih naslova kao što su *Celeste* (2018), *Hollow Knight* (2017.), *Gravity Circuit* (2023.) i *Pizza Tower* (2023.). Ovi i brojni drugi naslovi i njihova igranost pokazuju da ovaj žanr i dalje ima svoju publiku među današnjim igračima.

3.2.2.Karakteristike platformera

Već u samoj definiciji ovog žanra spomenute su neke od karakteristika po kojima prepoznajemo ovaj žanr. Od lika kojim upravlja igrač kroz neko okruženje do mogućnosti prelaženja razina, ostvarivanja najboljeg rezultata ili jednostavno preživljavanja od razine do razine.

Iako se ovaj žanr rapidno razvijao i napredovao te se današnje igrice ovog žanra podosta razlikuju od onih sa samih početaka, postoje neki elementi po kojima je lako prepoznati da se radi upravo o platformeru. Kao prvu stvar navesti ćemo interaktivnu okolinu ili dizajn razine koji govori puno o tome što sve lik kojim upravlja igrač može učiniti. Specifično za platformere je da su ove igre napravljene na način da izazivaju igrača kroz razine koje postaju sve teže kako igrač napreduje. Igranje iz perspektive trećeg lica je slijedeće obilježje ovog žanra, a uz to imamo i vertikalno i horizontalno kretanje kroz razine. (MasterClass, 2021.)

Upravo bi se korištenje skakanja i penjanja za kretanje kroz razine moglo nazvati glavnom karakteristikom ovog žanra s obzirom da razine i okolina imaju neravan teren i platforme koje su često pomične i na različitim visinama. Uz klasično kretanje lijevo i desno, te skakanje, penjanje i padanje, postoje i razne akrobatske sposobnosti koje čine prelaženje razina još zanimljivijim. Tako je za neke razine potrebno koristiti uže i kuku, ponekad je potrebno penjati se po zidovima, skakati s objekata, a ponekad i lebdjeti, koristiti trampolin i/ili se katapultirati na željenu visinu. (Konecki, 2022.)

Može se reći da su platformeri prepoznatljivi po jednostavnim, ali zaraznim mehanikama, te da raznovrsne razine pružaju igračima dinamično iskustvo, dok puzzle elementi dodaju dubini i izazovu. Unatoč evoluciji gaming industrije, 2D platformeri i dalje ostaju popularni među igračima zbog svoje jednostavnosti, ali istovremeno i zaraznosti.

3.2.3.Glavne mehanike

Za početak, definirat ćemo što uopće jesu mehanike u igrama. U igračkom smislu, mehanika je osnovna komponenta igre. Mehanika obuhvaća sva pravila, sve radnje koje igrač može poduzeti u igri i sve algoritme koji se odvijaju.

Stoga glavne mehanike 2D platformera uključuju kretanje (lijevo-desno) lika kojeg kontrolira igrač, skakanje kada je u pitanju kretnja prema 'gore' u svrhu prelaska preko prepreka ili izbjegavanja neprijatelja koje samo po sebi čini još jednu od mehanika koje su ključ u stvaranju 2D platformera. Prepreke i neprijatelji su mehanika koja izaziva igrača i otežava mu prelazak razina. Slijedeća mehanika je najkarakterističnija mehanika za ovaj žanr – platforme. Platforme su ključni elementi dizajna razina jer pružaju put za napredak i izazove koje je potrebno svladati. Na putu prema cilju, igrač se često susreće s mehanikom skupljanja

predmeta poput novčića, pojačanja (*power-ups*) ili ključeva koji otključavaju nova područja ili nove razine. Nekada se za prelazak razine potrebno boriti sa neprijateljima skakanjem na iste ili korištenjem oružja. Uz sve navedeno tu su i već spomenuti puzzle elementi koji zahtijevaju rješavanje logičkih zagonetki ili pronalaženje ispravnog puta u labirintima kako bi se napredovalo kroz igru. Za kraj, tu je i mehanika napretka kroz razine. Za ovaj žanr karakteristično je da se igre sastoje od niza razina ili svjetove koje igrači moraju proći kako bi dovršili igru, te upravo mehanika otežavanja svake slijedeće razine je ono što privlači igrače da nastave igrati.

2D platformeri se temelje na ovim mehanikama. One pružaju igračima dinamično i izazovno igračko iskustvo koje ih drži 'budnima' i motiviranima za daljnje napredovanje za igru. Iako postoji još brdo drugih, spomenute mehanike su ono što bi svaki platformer trebao imati. Kako i zašto su pojedine mehanike bitne, kako se izvode i kako utječu na igračko iskustvo pojasniti ćemo kroz izradu praktičnog dijela rada u slijedećim poglavljima.

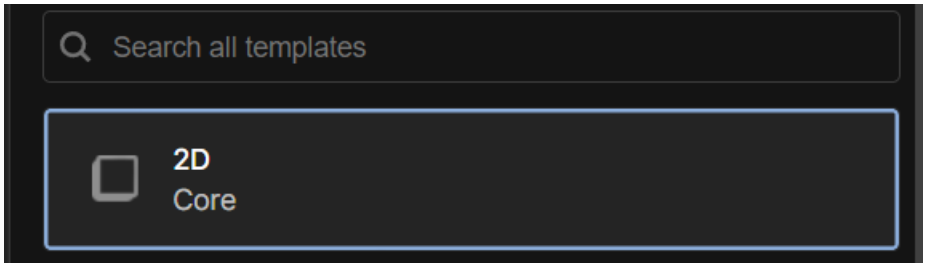
4. Implementacija

U ovom poglavlju proći ćemo detaljnu izradu vlastitog 2D platformera. Započeti ćemo s preuzimanjem *Unityja*, izradom projekta te zatim preuzimanjem potrebnih resursa. Nakon toga ćemo izraditi razine, a potom uvoditi i mehanike od jednostavnijih poput hodanja, skakanja i padanja do naprednijih kao što su sakupljanje stvari (eng. *Items*) i takozvanih *consumablesa*. Uz sve navedeno ubacit ćemo i animacije, brojač skupljenih stvari, pokretne platforme, razne zamke, zvučne efekte, početni i završni ekran, borbu protiv neprijatelja, sustav za skupljanje života.

4.1. Unity i Unity assets store

Za početak upoznati ćemo se s alatima i programima potrebnima za izradu vlastite igrice. Prvo što si trebamo osigurati je *Unity*. Potrebno je posjetiti službenu web stranicu *Unity Technologies* na adresi https://unity.com/ te izraditi korisnički račun i odabrati odgovarajući licencu. U našem slučaju dovoljna će biti licenca za studente koja je besplatna. Nakon toga preuzimamo *Unity Hub* koji je zapravo upravitelj aplikacija koji olakšava instalaciju, upravljanje verzijama i upravljanje projektima. Po završetku instalacije, potrebno ga je pokrenuti i prijaviti se sa svojim *Unity ID*-om. U *Unity Hubu* odabiremo karticu „*Installs*“ i kliknemo „*Add*“ kako bi dodali novu verziju *Unity-a*. Biramo željenu verziju i odabiremo „*Next*“. Potrebno je još odabrati komponente koje želimo instalirati, a u našem slučaju to je *Unity Editor* i zatim odabiremo „*Done*“. Nakon što je instalacija završena možemo početi s izradom igrice.

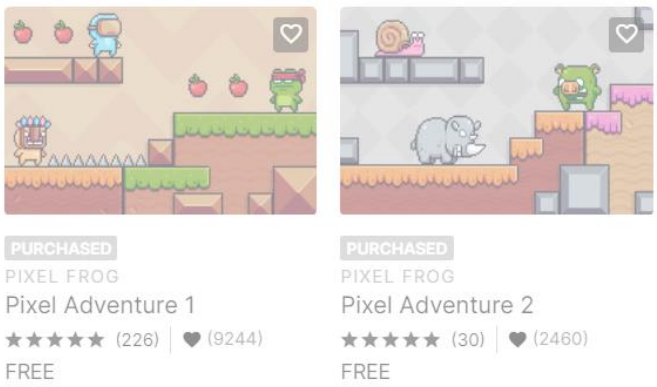
U kartici „*Projects*“ odabiremo „*New project*“ i zatim biramo „*2D Core*“ (Slika 1). Potom je potrebno popuniti podatke o projektu koji izrađujemo – naziv i lokaciju pohrane. Sve potvrdimo s „*Create project*“.



Slika 1: 2D Core opcija prilikom stvaranja novog projekta

4.1.1. Resursi (*Assets*)

Kada govorimo o resursima potrebnima za izradu igrica tada mislimo na alate potrebne za izradu terena i/ili razina, razne pozadine (eng. *Background*), lika kojim upravlja igrač (eng. *Character*), zamke, neprijatelji i sve ostalo što igricu čini igricom. Iako je sve navedeno moguće napraviti samostalno i tako u potpunosti prilagoditi svojim željama, idejama i potrebama, za one koji si žele uštedjeti nešto vremena postoji mogućnost preuzimanja besplatnih resursa. Za to postoji *Unity Asset Store* koji osim besplatnih, nudi i resurse koji se plaćaju. Konkretno u svrhu izrade ovog rada koristit će se resursi koje je moguće naći prilikom pretraživanja naziva *'Pixel Adventures'* (Slika 2).



Slika 2: Prikaz assetsa korištenih u radu

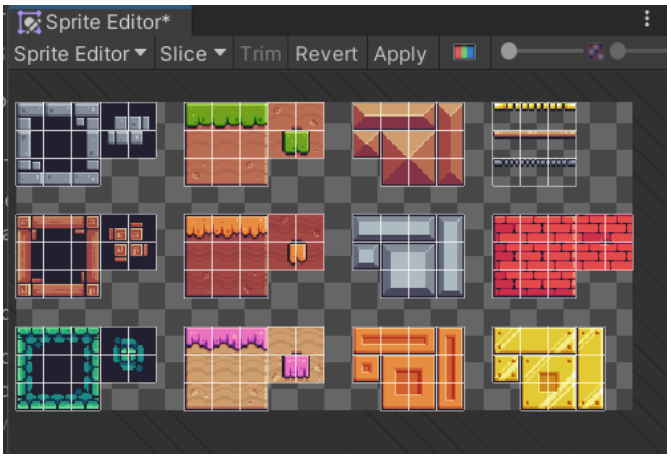
Preuzimanje je poprilično jednostavno – odabiremo opciju „*Add to My Assets*“ i zatim „*Open in Unity*“. U gornjem desnom kutu prozorčića koji se otvara potrebno je odabrati „*Download*“ i nakon što je preuzimanje gotovo potrebno je željene resurse uvesti odabirom opcije „*Import*“ gdje je važno označiti sve elemente.

4.1.2. Izrada razina

Sada kada imamo resurse za izradu razina možemo početi sa izradom istih. Prvo ćemo izraditi teren prve razine po kojem će se kretati igrač. Postoji više načina za ostvariti ovaj dio, lakši način je ako već imate izrezanu (eng. *Sliced*) verziju terena. U slučaju da ona ne postoji, sami trebamo pripremiti resurse koji služe za izradu terena. Spomenuti resursi nalaze se u odjeljku *Project-> Assets -> Pixel Adventure 1/2 -> Assets -> Terrain.*

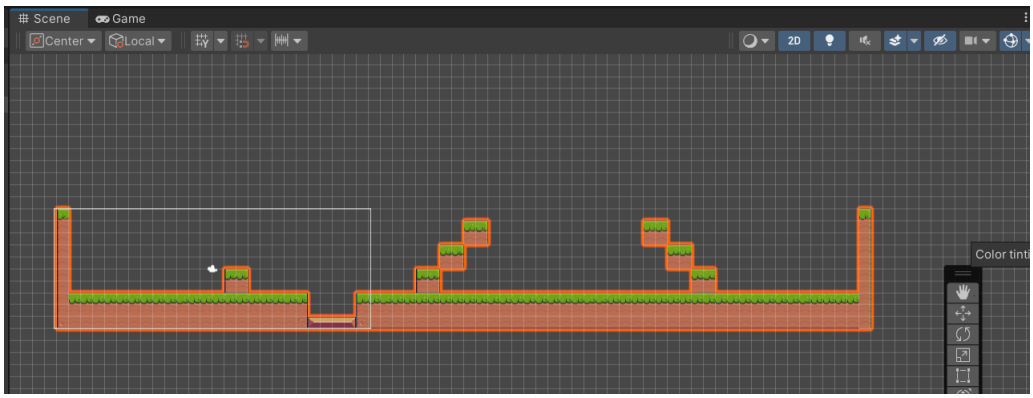
Priprema resursa za teren je poprilično jednostavan proces u kojem je potrebno samo postojeće elemente terena izrezati na manje dijelove. Kada ne postoji već pripremljena verzija priprema se vrši na način da odaberemo element u mapi i kada sam se u *Unityju* s desne

strane pojavi kartica *Inspector* mijenjamo dvije stvari – u polje *Pixels Per Unit* postavljamo vrijednost 16, a *Sprite Mode* postavljamo na *Multiple*. Ove promjene pohranimo s *Apply* i zatim odabiremo *Sprite Editor*. U ovom editoru, u odjeljku *Slice*, u polje *Type* postavljamo *Grid By Cell Size*, a x i y vrijednost pod *Pixel Size* opcijom postavljamo na 16. Promjene primjenjujemo gumbom *Slice* i kao rezultat dobijemo isjeckane kvadratiće raznolikog terena koji možemo koristiti za izradu igre (Slika 3).



Slika 3: Izgled pripremljenih elemenata za teren

Nakon toga potrebno je dodati naš teren na scenu. U lijevom dijelu ekrana u kartici *Hierarchy* odabiremo desni klik miša i iz izbornika biramo *2D Object -> Tilemap -> Rectangular*. Novostvoreni objekt imenujemo proizvoljnim (ali smislenim) imenom, u ovom slučaju *Terrain*. Zatim iz kartice *Window* odabiremo 2D -> *Tile Palette* i kreiramo novu paletu kojoj dajemo isti naziv – *Terrain*. Izrezani teren iz mape *Assets* odaberemo i povučemo u *Tile Palette* odjeljak te odabiremo željeni dio terena i crtamo željenu razinu po sceni (Slika 4).



Slika 4: Primjer izgleda razine

Kako bi estetski ovo ljepše izgledalo, moguće je dodati pozadinu (eng. *Background*). Postupak pripreme i postavljanja pozadine isti je kao kod prošlog objekta. Pozadine se nalaze u mapi *Assets* i podmapi *Backgrounds*. Željenu pozadinu pripremimo na način da joj promijenimo vrijednosti *Pixels Per Unit* i *Sprite Mode*, te ju izrežemo na manje dijelove u *Sprite Editoru* tako što joj promijenimo tip i x i y vrijednosti. U hijerarhiji dodajemo novi objekt i imenujemo ga *Background* te postavimo po sceni.

Kako nam naša pozadina ne bi zaklonila i sakrila teren, potrebno je dodijeliti takozvane slojeve (eng. *Layers*) na teren i pozadinu. To ćemo učiniti tako da odaberemo *Background* ili *Terrain* iz hijerarhije i zatim u *Inspector* dijelu u dodatnim postavkama (*Additional Settings*) otvaramo *Sorting Layer* i dodajemo dva nova sloja pomoću opcije *Add Sorting Layer* i nazivamo ih po elementima kojima ćemo ih dodijeliti – *Background* i *Terrain*. Slažemo ih redom – *Background* (*Layer* 0), *Default* (*Layer* 1) i *Terrain* (*Layer* 2). Za kraj, potrebno je ponovno u hijerarhiji odabrati elemente potrebne za izradu razine i dodijeliti im pripadajuće istoimene slojeve.

Prije nego što krenemo na slijedeće poglavlje, potrebno je još samo izraditi lika kojim će igrač upravljati u svrhu prelaska igrice. U *Hierarchy* dijelu dodajemo novi objekt koji ćemo nazvati *Player*, tako što odabiremo desni klik -> 2D *Object -> Sprites -> Square*. U mapi *Assets* pronađemo već pripremljene izglede igrača i otvorimo onog koji je *'Idle'*. Zatim odaberemo *Player* u hijerarhiji te je potrebno *Drag and drop* prvu sličicu u *Idle* skupini u polje *Sprite* u *Sprite Renderer* dijelu *Inspectora*. Također je potrebno prilagoditi veličinu stoga odabiremo sve sličice iz foldera željenog lika i u *Inspectoru* promijenimo vrijednost polja *Pixels Per Unit* na 16. U ovom trenutku naš lik je vidljiv na sceni, no pokretanjem igrice ne događa se ništa jer je potrebno dodati još neke komponente. Komponente dodajemo tako da odaberemo element Player i u Inspectoru odabiremo *Add Component*. Potrebno je dodati *Rigidbody* 2D (primjena fizike) i *Box Collider* 2D (prepoznavanje sudara/dodira s drugim objektima). Slično je potrebno napraviti i s našim terenom kojem je potrebno dodati komponente *Tilemap Collider* 2D koja automatski generira *'Collider'* za *Tilemap* objekt. No time se svaka naša kockica gleda kao zaseban objekt što baš nije najsretnije rješenje te zato dodajemo i *Composite Collider* 2D kako bi se naš cijeli teren gledao kao jedna cjelina. Dodavanjem prethodnih komponenti, automatski nam se generira i komponenta *Rigidbody* 2D kako teren ne bi 'propao' zajedno s igračem prilikom pokretanja. Potrebno je teren označiti i kao statični element – *Body Type: Static*. Za kraj dodajemo i komponentu *Platform Effector* 2D koja onemogućuje igraču da skoči ako se u padu nalazi uz neku platformu.

4.2. Kretnje i animacije

U ovom trenutku prilikom pokretanja igre lik kojeg smo odabrali pojavljuje se na sceni bez propadanja njega samoga ili cijelog terena. Slijedeći korak je omogućiti osnovne mehanike – kretnje lijevo-desno, te skok.

4.2.1. Osnovno kretanje lijevo-desno

Za implementaciju kretnji potrebna nam je skripta u programskom jeziku C#. Radi preglednosti i organiziranosti radimo novu mapu za skripte. Zatim stvaramo novu skriptu na način da odaberemo desni klik -> *Create* -> *C# Script*. Imenujemo ju *Player Movement* i otvaramo ju dvostrukim klikom. Nakon otvaranja *Editora*, u konkretnom slučaju – *Visual Studio*, možemo početi s kodom. Kako bismo mogli koristiti *Rigidbody2D* komponentu u svom kodu za korištenje sila koje će pomicati našeg lika lijevo – desno, potrebno je napraviti komponentu istog tipa (*Rigidbody* 2D). U funkciji *Start*() koja se pokreće prilikom paljenja igre – dohvaćamo vrijednost komponente *Rigidbody2D* u novonastalu varijablu *rb*. U funkciji *Update*() želimo da se naš lik kreće pritiskom na odgovarajuće tipke. Tako koristimo *rb.velocity* i stvaramo novi vektor koji se sastoji od dvije komponente od kojih se prva veže za kretanje po X-osi, a druga po Y-osi. Pošto se mi za početak želimo moći kretati lijevo-desno, potrebno je provjeriti je li pritisnuta neka od odgovarajućih tipki na tipkovnici (strelice lijevo i desno, slova A i D) i ako je, njihovu vrijednost je potrebno spremiti u novu varijablu *dirX* pomoću već ugrađene mogućnosti *Input.GetAxisRaw(„Horizontal“)*. Ovo omogućuje automatsko pohranjivanje vrijednosti -1 ako je pritisnuta strelica lijevo ili slovo A, ili vrijednosti 1 ako je pritisnuta strelica desno ili slovo D. Upravo ćemo ovu vrijednost pomnoženu s varijablom *moveSpeed* staviti kao prvi atribut našeg vektora. Također, bitno je spomenuti da varijabla *moveSpeed* sadrži atribut *SerializeField* pomoću kojeg je moguće da se ova varijabla prikazuje i uređuje direktno iz *Unity Inspectora*. Drugi atribut vektora zadržavati će trenutnu brzinu po y-osi. U konačnici, kod za kretnju lijevo-desno izgleda ovako:

```
Using System.Collections;
Using System.Collections.Generic;
Using UnityEngine;

Public class PlayerMovement : MonoBehaviour
{
     Private Rigidbody2D rb;
     Private float dirX = 0f;
     [SerializeField] private float moveSpeed = 7f;
```

```
Private void Start()
{
     Rb = GetComponent<Rigidbody2D>();
}

Private void Update()
{
     dirX = Input.GetAxisRaw(„Horizontal“)
     rb.velocity = new Vector2(dirX*moveSpeed, rb.velocity.y);
}
}
```

4.2.2. Skakanje i padanje

U platformerima jedna od osnovnih potreba je skakanje. Stoga ćemo u funkciji *Update*() provjeravati je li igrač pritisnuo tipku za skakanje pomoću *Input.GetButtonDown(„Jump“)*. *Jump* predstavlja string koji označava tipku koja je u *Unity* sustavu za unos postavljena kao inicijalizirana vrijednost za skok – tipka space.

Slijedeći korak je napraviti novu funkciju *IsGrounded()* koja je tipa bool i vraća vrijednost ovisno o tome nalazi li se objekt na tlu ili ne. U njoj pozivamo *Physics2D.BoxCast* koja provjerava nalazi li se objekt na tlu, a sastoji se od 6 atributa. Prva dva atributa vezana su za veličinu i središte *'bounding box'* objekta, 0f je kut rotacije, zatim slijedi definiranje smjera u kojem se provodi provjera, te duljina provjere koju postavljamo na .1f. Zadnji atribut vezan je za sloj s kojim se provjerava sudar. Ako se objekt nalazi na tlu, tj. *BoxCast* detektira sudar – funkcija vraća *true*, u suprotnom vraća *false*. Tu vrijednost pohranjujemo u varijablu *coll*.

Ako su zadovoljena oba uvjeta – pritisnuta je tipka *space* i lik se trenutno nalazi na tlu tada se izvršava slijedeće: U *velocity* atribut objekta *rb* sprema se novonastali vektor sastavljen od dvije komponente. Prva je *rb.velocity.x* koja ostaje nepromijenjena, a druga je *jumpForce* varijabla koju je potrebno deklarirati s mogućnošću uređivanja u samom *Unity Inspectoru*. Uz sve navedeno još je potrebno deklarirati i masku za slojeve – *jumpableGround* te primijeniti skriptu na igrača drag & drop metodom. U konačnici, skoro gotova skripta izuzev dijela zaduženog za animacije izgleda ovako:

```
Using System.Collections;
Using System.Collections.Generic;
Using UnityEngine;
```

```
Public class PlayerMovement : MonoBehaviour
{
     Private Rigidbody2D rb;
     Private Boxcollider2D coll;

     [SerializeField] private LayerMask jumpableGround;

     Private float dirX = 0f;
     [SerializeField] private float moveSpeed = 7f;
     [SerializeField] private float jumpForce = 14f;

Private void Start()
{
     Rb = GetComponent<Rigidbody2D>();
     Coll = GetComponent<BoxCollider2D>();
}

Private void Update()
{
     dirX = Input.GetAxisRaw(„Horizontal“)
     rb.velocity = new Vector2(dirX*moveSpeed, rb.velocity.y);

     if (Input.GetButtonDown(„Jump“) && IsGrounded())
     {
             Rb.velocity = new Vector2(rb.velocity.x, jumpForce);
     }
}

Private bool isGrounded()
{
     Return Physics2D.BoxCast(coll.bounds.center,coll.bounds.size, 0f,
Vector2.down,.1f, jumpableGround);
}
}
```

4.2.3. Skripta za kameru

Kako bi kamera pravilno pratila igrača potrebno je napraviti novu skriptu. Deklariramo jednu privatnu varijablu tipa *Transform* i nazovemo ju *Player* te joj dodamo atribut koji joj omogućuje uređivanje direktno iz *Unity inspectora*. U funkciji *Update*() u varijablu *transform* u

njenu poziciju spremamo novi vektor koji se sastoji od tri komponente - x, y i z vrijednosti pozicije igrača. Kod je poprilično jednostavan, a izgleda ovako:

```
Using System.Collections;
Using System.Collections.Generic;
Using UnityEngine;

Public class CameraController : MonoBehaviour
{
     [SerializeField] private Transform player;

     Private void Update()
     {
            Transform.position = new Vector3(player.position.x,
player.position.y, transform.position.z)
     }
}
```

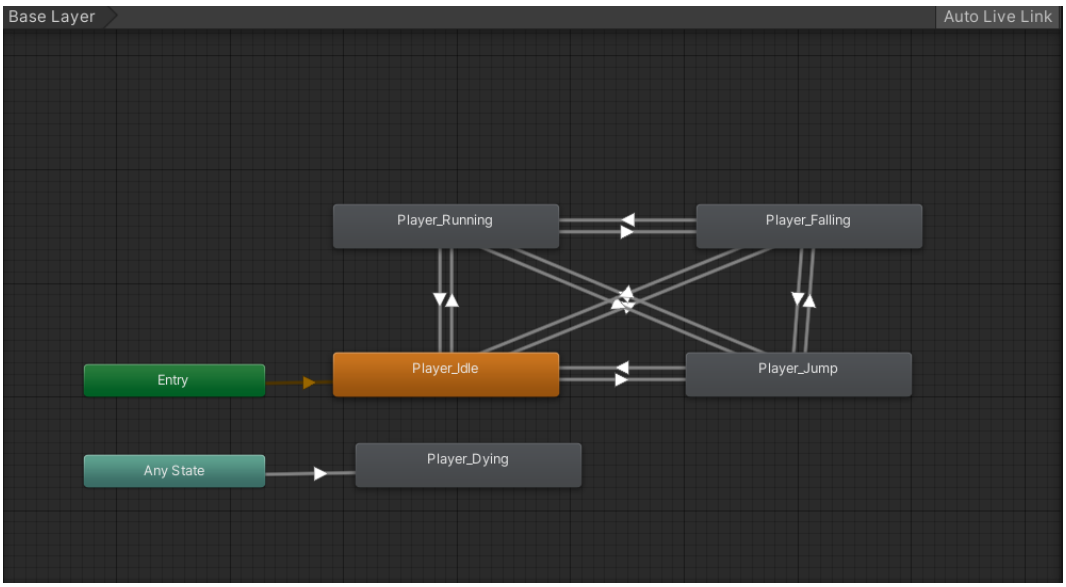
4.2.4. Animacije

Animacije se mogu objasniti kao pokretne slike ili grafike koje daju život likovima, objektima i okolini unutar igre. One omogućuju dinamičnost i interaktivnost igre tako što pružaju iluziju kretanja i akcije likova i ostalih elemenata igre. Svrha im je poboljšati korisničko iskustvo na način da doprinose imerzivnosti i zabavi.

U *Unityju* ih je prvo potrebno pripremiti. Tako ćemo prvo otići u mapu gdje nam se nalazi željeni glavni lik i označimo sve animacije. U *Inspector* dijelu podešavamo vrijednost *Pixels Per Unit* na 16. Potom slijedi izrada nove mape pod nazivom *Animations* i potrebno je stvoriti objekt animacije – desni klik -> *Create* -> *Animation*. Imenujemo ju smisleno prema animaciji o kojoj će se raditi (*Player_Idle*) i *'drag & droppamo*' animacijski objekt na lika kojim upravlja igrač. U slijedećem koraku potrebno je otvoriti animacijsko okno *Window* -> *Animation* -> *Animation*. U polje *Samples* upisujemo 18, a u vremensku lentu desno povlačimo sve željene sličice. Odabirom animacije u njenom *inspectoru* potrebno je označiti *loop time*. Ovo ponovimo za animacije padanja, umiranja, skakanja i trčanja.

Kako bi sve napravljeno bilo povezano potrebno je otvoriti animator. To činimo tako da odemo na *Window -> Animation -> Animator*. U animatoru potrebno je napraviti tranzicije kao na slici (Slika 5). Osim toga potrebno je napraviti i dva parametra – *state* koji je tipa *int* i *death* koji je tipa *trigger*. Sve tranzicije koje idu prema animaciji *Player_Idle* trebaju imati uvjet *state*

equals 0, one tranzicije koje idu prema animaciji trčanja umjesto 0 imaju 1, prema animaciji skakanja 2 i prema animaciji padanja 3. Tranziciji koja ide prema animaciji *Player_death* dodajemo *trigger death.*



Slika 5: Izgled tranzicija animacija na objektu igrača

U skriptu *Player Movement* dodajemo nekoliko linija koda kako bi se animacije mogle zaista izvoditi. Inicijaliziramo varijable *sprite* tipa *SpriteRenderer* i varijablu *anim* koja je tipa *Animator*. Osim toga definiramo i privatni enumeracijski tip varijable *MovementState* koja definira različita stanja u kojima se igrač može nalaziti tijekom igranja i pomaže određivanju prigodne animacije za stanje u kojem se trenutno nalazi igrač. U Start() funkciji dohvaćamo vrijednosti u varijable *sprite* i *anim*, a u funkciji *Update*() pozivamo novu funkciju *UpdateAnimationState*() koja je odgovorna za ažuriranje animacijskog stanja igrača na temelju trenutnog kretanja i situacije u igri, a izgleda ovako:

```
private void UpdateAnimationState()

    {
        MovementState state;

        if (dirX > 0f)
        {
            state = MovementState.running;
            sprite.flipX = false;
        }
```

```
        else if (dirX < 0f)
        {
            state = MovementState.running;
            sprite.flipX = true;
        }
        else
        {
            state = MovementState.idle;
        }

        if (rb.velocity.y > .1f)
        {
            state = MovementState.jump;
        }
        else if (rb.velocity.y < -.1f)
        {
            state = MovementState.falling;
        }

        anim.SetInteger("state", (int)state);
    }
```

4.3. Dodatne mehanike

U ovom trenutku implementirane su osnovne mehanike za kretanje, a za posebno igračko iskustvo dodane su animacije. Već samo s ovim mehanikama moguće je igrati i uživati u 2D platformerima te dosta jednostavnijih minimalističkih igrica koristi samo kretanje i skakanje kao mehanike dovoljne za igranje i prelaženje razina. Ipak, kako bi pridonijeli igračkom iskustvu u ovom radu biti će implementirane i dodatne mehanike kao što su skupljanje voća i pripadni brojač, pokretne i statične zamke, sustav za živote (eng. *Health system*), a u drugoj i trećoj razini biti će i uvedeni i jednostavniji i složeniji neprijatelji. Kako bi sve ovo bilo još povezanije dodati ćemo zvučne efekte, te početni i završni ekran.

4.3.1. Skupljanje jagoda i brojač

Prvi korak je pripremiti element koji skupljamo. Iz resursa pronađemo folder s raznim voćem i odaberemo jedno te mu promijenimo *Pixels Per Unit* vrijednost u 16. Zatim u hijerarhiji radimo novi element *2D object -> sprite -> square ->* imenujemo ga *Collectibles* i povlačimo

prvu sličicu odabranog voća (Slika 6). Na element voća dodajemo komponentu *Box Collider 2D* i označujemo opciju *Is Trigger*. Također, dodajemo novu oznaku tako da odabiremo *Untagged -> Add Tag* -> + -> imenujemo novu oznaku s *Strawberry*.

Slika 6: Prikaz izgleda stvari koje igrač sakuplja

Sada je potrebno još stvoriti skriptu – *ItemCollector*. U ovoj skripti nisu potrebne već unaprijed pripremljene funkcije *Start* i *Update* jer se ova skripta pokreće samo kada se *'triggera'* voćka. Unutar funkcije *OnTriggerEnter2D(Collider2D collision)* provjeravamo unutar *if-a* je li došlo do kolizije s objektom koji nosi oznaku *Strawberry*. Ako je, potrebno je ukloniti pokupljenu voćku, a to je moguće s naredbom *Destroy(collision.gameObject)*. Skriptu je potrebno metodom '*drag & drop'* primijeniti na objekt igrača.

Osim same mehanike skupljanja, dodati ćemo i brojač koji broji koliko voćki je pokupljeno prilikom igranja tako da igrač u svakom trenutku može znati kako napreduje. U istoj skripti dodajemo novu varijablu *strawberries* i postavljamo joj inicijalnu vrijednost na 0. Zatim u if-u nakon što smo uklonili voćku sa scene, povećavamo broj jagoda za 1. Sve ovo ispisujemo u konzolu s naredbom *Debug*. Na scenu dodajemo novi objekt desni klik -> UI -> *Legacy -> Text -> Strawberries counter*. Novonastali objekt uredimo po želji – font, veličina, boja. U skripti *ItemCollector* deklariramo novu privatnu varijablu *strawberryCounter* koja je tipa tekst i sadrži atribut *SerializeField*. U if blok dodajemo pripadajući tekst napravljenoj varijabli pomoću naredbe: *strawberryCounter.text = „Strawberries:“ + strawberry;* Skriptu je potrebno primijeniti na objekt igrača, a u polje *Strawberry Counter* prenijeti naš tekstualni objekt *Strawberry Counter (Text)*. U konačnici ova jednostavna skripta izgleda ovako:

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

public class ItemCollector : MonoBehaviour
{
    private int strawberry = 0;
    [SerializeField] private Text strawberryCounter;

  private void OnTriggerEnter2D(Collider2D collision)
    {
       if (collision.gameObject.CompareTag("Strawberry"))
       {
         AudioManager.instance.Play("Collecting");
         Destroy(collision.gameObject);
         strawberry ++;
         Debug.Log("Strawberries: " + strawberry);
         strawberryCounter.text = "Strawberries: " + strawberry +
"/10";
       }
    }
}
```

4.3.2. Zamke, pokretne platforme i neprijatelji

Glavna uloga zamki i neprijatelja je da otežaju igraču put do cilja bilo da se radi o skupljanju stvari ili prelaska razina. U folderu *Traps* (zamke) postoji više zamki za korištenje, a moguće ih je i napraviti samostalno. Za potrebe ovog rada koristit će se već gotovi šiljci koje je prethodno potrebno prilagoditi postavljanjem vrijednosti *Pixels Per Unit* na 16. U hijerarhiji dodajemo novi objekt odabirom desnog klika -> *2D object -> square* i prenosimo sličicu na *Sprite* polje. Od komponenti potrebno je dodati *Box Collider 2D*.

Slijedeći korak je napraviti skriptu koja će omogućiti da igrač u dodiru s opasnosti pogine i bude vraćen na početak. Skriptu nazivamo *Player_Life* i stvaramo novu privatnu funkciju tipa void *OnCollisionEnter2D(Collision2D collision)*. U if bloku provjerava se je li došlo do kolizije s nekom od zamki, a ovo ćemo najjednostavnije napraviti tako da našim zamkama dodamo novu oznaku – *trap*. U slučaju da je došlo do kolizije poziva se funkcija *Die()* koja je privatna i tipa void, a za nju nam je prethodno potrebno deklarirati varijable *rb* i *anim* koje

poprimaju vrijednosti u funkciji *Start().* U samoj *Die()* funkciji komponenta koja elementu igrača omogućuje kretnje se isključuje kako bi element postao statičan, a kod varijable *anim triggera* se okidač *death*. Kako bi se igrač mogao ponovno oživiti i probati preći razinu iz nekog drugog pokušaja potrebno je pozvati i funkciju *GameOverScreen.Setup()* koja predstavlja postavljanje završnog ekrana o kojem će više biti rečeno kasnije u radu. U konačnici skripta *PlayerLife* izgleda ovako:

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

public class PlayerLife : MonoBehaviour
{
    private Rigidbody2D rb;
    private Animator anim;

    public GameOverScreen GameOverScreen;

    void Start()
    {
        anim = GetComponent<Animator>();
        rb = GetComponent<Rigidbody2D>();
    }
   private void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject.CompareTag("Trap"))
        {
            Die();
        }
    }
    private void Die()
    {
        GetComponent<PlayerMovement>().enabled = false;
        anim.SetTrigger("death");
        GameOverScreen.Setup();
    }
}
```

Za izradu pokretne platforme potrebne su dvije skripte. Prva koja je zaslužna za kretanje platforme od točke do točke, te druga skripta koja omogućuje igraču da 'miruje' na pokretnoj platformi.

Prva skripta sadrži varijable *waypoints* (koja je ujedno i polje *gameObjectova* koji predstavljaju točke duž kojih se treba kretati element), *currentWaypointIndex* (koji pohranjuje indeks trenutne točke kojoj element slijedi), *speed* (za brzinu kretanja elementa) i *spriteRenderer* (kao referenca na *SpriteRenderer* komponentu elementa). U metodi *Start*() dohvaćamo komponentu elementa *SpriteRenderer*, dok u metodi *Update*() provjeravamo udaljenost između trenutne pozicije elementa i trenutne točke kako bi element znao da prelazi na slijedeću točku. Pri dolasku do zadnje točke, element se okreće u suprotnom smjeru. U skripti to izgledao ovako:

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

public class WayPointFollower : MonoBehaviour

{
    [SerializeField] private GameObject[] waypoints;
    private int currentWaypointIndex = 0;
    [SerializeField] private float speed = 2f;
    private SpriteRenderer spriteRenderer;

    private void Start()
    {
        spriteRenderer = GetComponent<SpriteRenderer>();
    }

    private void Update()
    {
         if(Vector2.Distance(waypoints[currentWaypointIndex].transform
.position, transform.position) < .1f)
        {
            currentWaypointIndex ++;
            if(currentWaypointIndex >= waypoints.Length)
            {
                currentWaypointIndex = 0;
```

```
                }
            }

            If   (waypoints[currentWaypointIndex].transform.position.x   >
transform.position.x)
            {
                spriteRenderer.flipX = true;
            }
            else
            {
                spriteRenderer.flipX = false;
            }
            transform.position = Vector2.MoveTowards(transform.position,
            waypoints[currentWaypointIndex].transform.position,
            Time.deltaTime * speed);
        }
    }
```

Druga skripta je dosta jednostavna i sadrži dvije metode – *OnTriggerEnter2D* i *OnTriggerExit2D* s proslijeđenom varijablom *collision* tipa *Collider2D*. Po nazivu metoda moguće je zaključiti da prva metoda provjerava kada se drugi objekt, konkretno igrač, sudari s objektom pokretne platforme što rezultira time da se objekt platforme postavlja kao roditelj (eng. *Parent*) objekta igrača i to mu omogućava da prati kretnje platforme. Druga metoda poništava prethodnu onoga trenutka kada se element igrača odmakne od elementa pokretne platforme. Skripta izgleda ovako:

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

public class StickyPlatform : MonoBehaviour
{
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.gameObject.name == "Player")
        {
            collision.gameObject.transform.SetParent(transform);
        }
    }
```

```
    private void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.gameObject.name == "Player")
        {
            collision.gameObject.transform.SetParent(null);
        }
    }
}
```

Od neprijatelja unutar ove igrice implementirane su dvije vrste – jednostavniji i kompleksniji. Jednostavniji neprijatelj je plava kornjača s bodljikavim oklopom čija skripta izgleda ovako:

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

public class EnemyTurtle : MonoBehaviour
{

    [SerializeField] private Transform[] waypoints;
    private int currentWaypointIndex = 0;
    [SerializeField] private float speed = 2f;
    private SpriteRenderer spriteRenderer;
    private Animator anim;

    private void Start()
    {
        spriteRenderer = GetComponent<SpriteRenderer>();
        anim = GetComponent<Animator>();
    }

    private void Update()
    {
      Collider2D[]     colliders     =     Physics2D.OverlapCircleAll
(transform.position, 0.2f);

       UpdateAnimationState();
```

```
        }

        private void UpdateAnimationState()
        {
             if(Vector2.Distance(waypoints[currentWaypointIndex].position,
transform.position) < .1f)
            {
                currentWaypointIndex ++;
                if(currentWaypointIndex >= waypoints.Length)
                {
                    currentWaypointIndex = 0;
                }
            }

                 if     (waypoints[currentWaypointIndex].position.x     >
transform.position.x)
            {
                spriteRenderer.flipX = true;
            }
            else
            {
                spriteRenderer.flipX = false;
            }

             anim.SetBool("isWalking",true);

             transform.position = Vector2.MoveTowards(transform.position,
waypoints[currentWaypointIndex].position, Time.deltaTime * speed);



        }

        [SerializeField] private float damage;

        private void OnTriggerEnter2D(Collider2D collision)
        {
            if (collision.tag == "Player")
            {
                collision.GetComponent<Health>().TakeDamage(damage);
            }
        }
```

```
}
```

Iz koda je vidljivo da kornjača ima mogućnost kretanja lijevo desno pomoću iste logike primijenjene i na pokretnu platformu. Uz to u metodi *UpdateAnimationState*() pokreće se i animacija kretanja kornjače, a u metodi *OnTriggerEnter2D*() provjerava se je li došlo do sudara s drugim objektom i ako se radi o sudaru s elementom s oznakom *Player* tada se dohvaća *Health* komponenta igrača i nanosi mu se šteta u obliku uzimanja života.

Kompleksniji neprijatelj napravljen je pomoću dvije skripte – *MeleeEnemy* i *EnemyPatrol*. Prva skripta upravlja ponašanjem neprijatelja kada se igrač nalazi na kratkoj udaljenosti. Provjerava ima li igrača u blizini, a ako je odgovor pozitivan i ako je prošlo dovoljno vremena od prethodnog napada, pokreće se animacija *'Enemy_attack*'. U metodi *Awake* postavljaju se inicijalne vrijednosti, dok se u metodi *Update*() provjerava je li igrač u dometu napada i pokreće se napad u slučaju da jest. U metodi *SeeingPlayer*() koristi se *boxcast* da otkrije postoji li igrač u dometu napada. *OnDrawGizmos*() metoda crta gizmos koji ocrtava domet napada. U metodi *AttackPlayer*() prilikom napada poziva se metoda *DamagePlayer*() koja nanosi štetu igraču koji se nalazi u dometu neprijatelja. Kod u konačnici izgleda ovako:

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

public class MeleeEnemy : MonoBehaviour
{
    private Health playerHealth;

    [SerializeField] private float cooldown;
    [SerializeField] private float range;
    [SerializeField] private int _damage;

    [SerializeField] private float colliderDistance;
    [SerializeField] private BoxCollider2D coll;
    [SerializeField] private LayerMask playerLayer;
    private float timer = Mathf.Infinity;

    private Animator anim;

    private EnemyPatrol enemyPatrol;
```

```
        private void Awake()
        {
            anim = GetComponent<Animator>();
            enemyPatrol = GetComponentInParent<EnemyPatrol>();
            playerHealth = GetComponent<Health>();
        }

        private void Update()
        {
            timer = timer + Time.deltaTime;

            if(SeeingPlayer())
            {
                if (timer >= cooldown)
                {
                    anim.SetTrigger("attack");
                    timer = 0;
                }
            }

             if (enemyPatrol != null)
                enemyPatrol.enabled = !SeeingPlayer();

        }



        private bool SeeingPlayer()
        {
              RaycastHit2D hit = Physics2D.BoxCast(coll.bounds.center + transform.right * range *transform.localScale.x * colliderDistance,
              new Vector3(coll.bounds.size.x * range, coll.bounds.size.y, coll.bounds.size.z), 0, Vector2.left, 0 , playerLayer);

        if(hit.collider != null)
          {
                playerHealth = hit.transform.GetComponent<Health>();
          }

return hit.collider != null && hit.collider.CompareTag ("Player");
        }
```

```
        private void OnDrawGizmos()
        {
            Gizmos.color = Color.red;
          Gizmos.DrawWireCube(coll.bounds.center + transform.right * range
* transform.localScale.x * colliderDistance, new Vector3(coll.bounds.size.x
*range,coll.bounds.size.y, coll.bounds.size.z));
        }
        private void DamagePlayer()
        {
             if (playerHealth != null && SeeingPlayer())
            playerHealth.TakeDamage(_damage);
        }

        public void AttackPlayer()
    {
        DamagePlayer();
    }
    }
```

Druga spomenuta skripta zadužena je za patroliranje neprijatelja u trenutcima kada igrač nije unutar dometa napada. Potrebno je inicijalizirati varijable u koje će se spremati podatak o krajnjoj lijevoj i desnoj točki između kojih će se kretati neprijatelj. Također potrebne su varijable za brzinu kretanja neprijatelja ali i vrijeme mirovanja te tajmer mirovanja koji će služiti kako bi implementirali kratki odmor objekta neprijatelja nakon što dođe do jedne od rubnih točaka. Od metoda imamo metodu *Awake* koja se koristi za inicijalizaciju objekta prije nego što se pokrene igra. U ovom slučaju koristi se za postavljanje početne skale neprijatelja kako bi se osiguralo da isti uvijek započinje svoje kretnje na jednaki način. U metodi *OnDisable* osigurava se da se u trenutku mirovanja izvodi odgovarajuća animacija. Kroz metodu *Update* provjerava se trenutni smjer kretanja neprijatelja i provodi se kretanje u tom smjeru sve dok objekt ne dođe do rubne točke. Nakon toga poziva se metoda *DirectionChange* koja postavlja neprijatelja u stanje mirovanja te nakon određenog vremena okreće ga u drugom smjeru i ponavlja se kretnja prema suprotnoj rubnoj točki. Kretnja u određenom smjeru i određenom brzinom pomoću metode *MoveInDirection*. Izgled konačne skripte vidljiv je u nastavku:

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

```
public class EnemyPatrol : MonoBehaviour
{
    [SerializeField] private Transform leftEdge;
    [SerializeField] private Transform rightEdge;
    [SerializeField] private Transform enemy;

    [SerializeField] private float speed;
    private Vector3 initScale;
    private bool movingLeft;

    [SerializeField] private float idleDuration;
    private float idleTimer;
    [SerializeField] private Animator anim;

    private void Awake()
    {
        initScale = enemy.localScale;
    }

    private void OnDisable()
    {
        anim.SetBool("isMoving", false);
    }

    private void Update()
    {
        if (movingLeft)
        {
            if (enemy.position.x >= leftEdge.position.x)
                MoveInDirection(-1);
            else
                DirectionChange();
        }
        else
        {
            if (enemy.position.x <= rightEdge.position.x)
                MoveInDirection(1);
            else
                DirectionChange();
```

```
            }
        }

        private void DirectionChange()
        {
            anim.SetBool("isMoving", false);
            idleTimer += Time.deltaTime;

            if(idleTimer > idleDuration)
                movingLeft = !movingLeft;

            enemy.localScale = new Vector3(initScale.x * (movingLeft ? -1
: 1), initScale.y, initScale.z);
        }

        private void MoveInDirection(int _direction)
        {
            enemy.position = new Vector3(enemy.position.x +
Time.deltaTime * _direction * speed, enemy.position.y, enemy.position.z);
            anim.SetBool("isMoving", true);
        }
    }
```

Kao zadnju vrstu zamki imamo rotirajuće oštrice. Na njih je potrebno primijeniti skriptu *WayPointFollower* kao i kod pokretnih platformi, ali za razliku od platformi oštrice trebaju raditi neku štetu igraču. Za to nam je potrebna nova skripta koja izgleda ovako:

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

    public class TakingDamage : MonoBehaviour
    {
        [SerializeField] private float speed = 2f;
        [SerializeField] private float damage;
        public float damageTimeout = 1f;
        private bool canTakeDamage = true;

        private void OnTriggerEnter2D(Collider2D collision)
        {
```

```
        if (canTakeDamage && collision.tag == "Player")
        {

            collision.GetComponent<Health>().TakeDamage(damage);
            StartCoroutine(damageTimer());
        }
    }

    private IEnumerator damageTimer()
    {
        canTakeDamage = false;
        yield return new WaitForSeconds(damageTimeout);
        canTakeDamage = true;
    }

    private void Update()
    {
        transform.Rotate(0, 0, 360 * speed * Time.deltaTime);
    }
}
```

Iz koda je vidljivo da su nam dovoljne četiri varijable – *speed, damage, damageTimeout* i *canTakeDamage*. Unutar prve metode *OnTriggerEnter2D* imamo if u kojem se provjerava jesu li zadovoljena dva uvjeta. Prvi uvjet je da igrač može primiti štetu odnosno da je u varijablu *canTakeDamage* pohranjena vrijednost *true*. Drugi uvjet vezan je za oznaku objekta s kojim je došlo do kolizije, te ako se radi o oznaci '*Player*' tada se izvršava naredba koja dohvaća *Health* atribut igrača i radi mu određenu štetu. U metodi *damageTimer* osigurava se da igrač ne može prebrzo ostati bez svih života već da ima kratki period vremena za skloniti se na sigurno prije zadobivanja novih oštećenja. Metoda *Update* koristi se radi estetskih razloga kako bi se oštrica uistinu rotirala i time pridonesla korisničkom iskustvu.

4.3.3. Sustav za živote

Sustav za živote (eng. *Health system*) koristi se za praćenje i vizualizaciju života igrača tijekom igre. Za izradu istog potrebne su dvije skripte i preuzeti resurs koji predstavlja traku života. U skripti *HealthBar* vizualno prikazujemo traku života tako da postavimo mogućnost biranja početnog broja života s maksimalnim odabirom 10. U funkciji *Update*() ažuriramo trenutno stanje zdravlja igrača. Ova jednostavna skripta u konačnici izgleda ovako:

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

public class HealthBar : MonoBehaviour
{
   [SerializeField] private Health playerHealth;
   [SerializeField] private Image totalhealthBar;
   [SerializeField] private Image currenthealthBar;

    private void Start ()
    {
        totalhealthBar.fillAmount = playerHealth.currentHealth / 10;
    }

    private void Update()
    {
        currenthealthBar.fillAmount = playerHealth.currentHealth /
10;
    }
}
```

Skripta *Health* služi kako bi upravljali zdravljem igrača. Sastoji se od dvije metode – *Awake*() u kojoj se dohvaćaju inicijalne vrijednosti i već više puta spomenute metode *TakeDamage*() koja oduzima živote igraču kada primi štetu. Ako broj života padne na nulu, igrač umire i pokreće se prikladna animacija koja onemogućava igraču kretanje. Izgled konačne skripte:

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

public class Health : MonoBehaviour
{
    [SerializeField] private float startingHealth;
    public float currentHealth {get; private set;}
    private Animator anim;
```

```
        private bool dead;

        private void Awake()
        {
            currentHealth = startingHealth;
            anim = GetComponent<Animator>();
        }

        public void TakeDamage (float _damage)
        {
            currentHealth = Mathf.Clamp(currentHealth - _damage,0,
startingHealth);

            if (currentHealth > 0 )
            {
               Debug.Log("Ouch!");
            }
            else
            {
                if(!dead)
                {
                anim.SetBool("grounded", true);
                GetComponent<PlayerMovement>().enabled = false;
                anim.SetTrigger("death");
                dead = true;
                }
            }
        }
    }
```

4.3.4. Zvučni efekti

Zvučni efekti igraju ključnu ulogu u stvaranju imerzivnog iskustva u 2D platformerima. Osim što podižu atmosferu i ambijent, koriste i kao povratna informacija igraču o akcijama i interakcijama unutar igre kao što su skok, sakupljanje stvari, primanja štete i slično. Ključni su za stvaranje uzbudljivog i kvalitetnog igračkog iskustva. U našem konkretnom primjeru za implementaciju zvučnih efekata bile su potrebne dvije skripte.

Sound skripta predstavlja jednostavnu klasu koja sadrži podatke o zvuku kao što su naziv, audio zapis samog zvuka, jačina, ton zvuka i ponavljanje zvuka. Svaki od navedenih atributa može se postavljati i direktno iz *Unity Inspector-a*. Konačna skripta izgleda ovako:

```
using UnityEngine;
using UnityEngine.Audio;

[System.Serializable]
public class Sound {

    public string name;

    public AudioClip clip;
    public AudioMixerGroup mixer;

    [Range(0f, 1f)]
    public float volume = 1;

    [Range(-3f, 3f)]
    public float pitch = 1;

    public bool loop = false;

    [HideInInspector]
    public AudioSource source;
}
```

U skripti *AudioManager* kontrolira se reprodukcija zvukova u igri. Unutar metode *Awake*() koja se poziva prilikom pokretanja igre, provjerava se postoji li već jedna instanca *AudioManagera*, ako postoji – uništava se trenutna instanca, a u suprotnom slučaju postavlja trenutnu instancu na ovu i osigurava da *AudioManager* preživi između prelaska scena. Metodama *Play* i *Stop* pokreće i zaustavlja se određeni zvuk. Prikaz spomenute skripte:

```
using System;
using UnityEngine;
using UnityEngine.Audio;

public class AudioManager : MonoBehaviour {

    public static AudioManager instance;
    public Sound[] sounds;
```

```
    void Awake ()
    {
        if (instance != null)
        {
            Destroy(gameObject);
            return;
        } else
        {
            instance = this;
            DontDestroyOnLoad(gameObject);
        }

        foreach (Sound s in sounds)
        {
            s.source = gameObject.AddComponent<AudioSource>();
            s.source.clip = s.clip;
            s.source.volume = s.volume;
            s.source.pitch = s.pitch;
            s.source.loop = s.loop;
        }
    }

    public void Play(string sound)
    {
        Sound s = Array.Find(sounds, item => item.name == sound);
        s.source.Play();
    }
    public void Stop(string sound)
    {
        Sound s = Array.Find(sounds, item => item.name == sound);
        s.source.Stop();
    }
}
```

Nakon postavljanja svih zvukova unutar *AudioManagera* potrebno ih je dodati na odgovarajućim trenutcima unutar skripti – npr. u skripti *PlayerMovement* kod metode koja omoguće igraču skok postavlja se jednostavna linija koda koja pokreće zvuk pod nazivom *Jump*:

```
rb.velocity = new Vector2(rb.velocity.x, jumpForce);
AudioManager.instance.Play("Jump");
```

4.3.5. Start i Game Over ekrani

Početni (eng. *Start*) i završni (eng. *Game Over)* ekrani neizostavni su dijelovi kako 2D platformera tako i svih ostalih igrica. Implementacija istih je izvediva na više načina, a u konkretnom primjeru Start ekran biti će prikazan kao nova scena dok će Game Over ekran biti implementiran unutar samih scena razina.

Game Over ekran sadrži varijablu koja pohranjuje unaprijed postavljeni indeks svake od scena. Unutar metode *Setup*() postavlja se kao aktivan ekran i natpisom *Game over* i tri gumba – *Start over*, *Restart level* i *Quit game*. Pritiskom na prvi gumb funkcijom *LoadScene* učitava se prva razina, dok se odabirom drugog guma učitava trenutna scena odnosno element igrača se postavlja na početak razine na kojoj je igrač izgubio život i daje mu priliku da stigne do kraja razine iz drugog pokušaja. Treći gumb samo izlazi iz cijele aplikacije, a konačna skripta vidljiva je u nastavku.

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

public class GameOverScreen : MonoBehaviour
{
     public int sceneBuildIndex;

    public void Setup ()
    {
        gameObject.SetActive(true);
    }

    public void StartOverButton()
    {
        SceneManager.LoadScene("Level1");
    }

    public void RestartButton()
```

```
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }

    public void QuitButton()
    {
        Application.Quit();
    }
}
```

Početni ekran ostvarili smo na drugi način. Napravili smo novu scenu, dodali željenu pozadinu i tekst te stvorili jedan gumb s natpisom Play. Pritiskom na spomenuti gumb pomoću *LoadSceneAysnc*(1) poziva se scena prve razine i omogućuje se igraču početak igranja. Skripta je vrlo jednostavna i kratka, a izgleda ovako:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
public class MainMenu : MonoBehaviour
{
    public void PlayGame()
    {
        SceneManager.LoadSceneAsync(1);
    }
}
```

Zadnja skripta korištena za izradu ovog rada je vrlo jednostavna skripta pod nazivom *Endpoint*. Po nazivu se da zaključiti da se radi o kodu koji se izvršava nakon što se elementom igrača dođe do kraja razine:

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

public class EndPoint : MonoBehaviour
{
    public int sceneBuildIndex;
```

```
    private void OnTriggerEnter2D(Collider2D other)
    {

        print("Trigger entered");

        if (other.tag == "Player")
        {
      print("Switching to " + sceneBuildIndex);
      SceneManager. LoadScene(sceneBuildIndex, LoadSceneMode.Single);
        }
}
}
```

Zapravo se radi o samo jednoj metodi koja se pokreće na okidač koji se nalazi na bilo kojem objektu kojeg odredimo kao krajnju točku pojedine razine. Ako je došlo do kontakta s objektom koji nosi oznaku '*Player*' potrebno je prebaciti scenu. Broj indeksa scene upisuje se direktno u *Inspectoru* na svakoj razini.

5. Zaključak

U ovom završnom radu fokus je bio na izradi 2D platformera. Izrada ove igrice bilo je jedno novo i obogaćujuće iskustvo jer kroz sam proces osim zadovoljavajućeg konačnog rezultata, suočavala sam se s brojnim problemima i izazovima koji su uključivali razna istraživanja kako bih pronašla zadovoljavajuće rješenje.

Stjecanje iskustva rada u intuitivnom i fleksibilnom sučelju Unity-a bilo je vrlo zanimljivo s obzirom na činjenicu da su upravo na toj platformi stvorene i mnoge poznate igrice današnjice. Zahvaljujući raznim alatima za vizualizaciju programiranja, animiranja i implementaciji zvučnih efekata vrijeme potrebno za izradu jednostavne igrice je svedeno na minimum, a postignute su sve željene funkcionalnosti u igri i stvoren je jedinstven korisnički doživljaj. Izradom ove igre, stekla sam dublje razumijevanje različitih aspekata razvoja igara od samog programiranja skripti do dizajna i umjetničke strane izrade. Prilikom izrade praktičnog dijela cilj je bio držati se onoga što sam naučila tijekom istraživanja potrebnog za teorijski dio ovoga rada. Povodom toga implementirane su razne mehanike koje su glavna karakteristika ovog tipa igre, te je postignut napredak kroz razine u smislu težine pa tako prva razina sadrži samo zamke, druga razina uz brdo zamki sadrži i jednostavnog neprijatelja, dok treća razina uz nekoliko zamki i jednostavnih neprijatelja, sadrži i kompleksnijeg neprijatelja.

U konačnici, ovo iskustvo izrade vlastite igrice predstavlja vrijedan korak u mom osobnom razvoju. Osim dubljeg uvida u proces razvoja igara, bogatija sam i za nove načine rješavanja problema i nove načine kreativnog izražavanja.

Literatura

Fletch. (13. Listopad 2020.). *Povijest Unityja*. Preuzeto 25. Ožujak 2024. iz UNITY.HR: https://unity.hr/povijest-unityja/

Gaming, N. (23. Ožujak 2017.). *The evolution of platform games in 9 steps*. Preuzeto 26. Ožujak 2024. iz Redbull: https://www.redbull.com/in-en/evolution-of-platformers

Konecki, M. (2022.). *Razvoj računalnih igara.* Preuzeto 26. Ožujak 2024. iz Moodle: https://elfarchive2223.foi.hr/mod/resource/view.php?id=95246

Lacoma, T. (15. Rujan 2023.). *What is Unity?* Preuzeto 25. Ožujak 2024. iz Android Police: https://www.androidpolice.com/what-is-unity/

MasterClass. (9. Srpanj 2021.). *Learn About Platform Game: 7 Examples of Platform Games*. Preuzeto 26. Ožujak 2024. iz Masterclass: https://www.masterclass.com/articles/platform-game-explained

Team, C. &. (22. Veljača 2022.). *Top Games Made with Unity: Unity Game Programming*. Preuzeto 25. Ožujak 2024. iz Create&Learn: https://www.create-learn.us/blog/top-games-made-with-unity/

Technologies, U. (2024.). *Welcome to Unity*. Preuzeto 25. Ožujak 2024. iz Unity: https://unity.com/our-company

Popis slika