Izrada višeplatformske strateške videoigre u programskom alatu Godot

Mrkonjić, Luka

Master's thesis / Diplomski rad

2023

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

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:711616

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Luka Mrkonjić

IZRADA VIŠEPLATFORMSKE STRATEŠKE VIDEOIGRE U PROGRAMSKOM ALATU GODOT

DIPLOMSKI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Luka Mrkonjić

Matični broj: 45913/17-R

JMBAG: 0016129448

Studij: Informacijsko i programsko inženjerstvo

IZRADA VIŠEPLATFORMSKE STRATEŠKE VIDEOIGRE U PROGRAMSKOM ALATU GODOT

DIPLOMSKI RAD

Mentor:

Doc. dr. sc. Mladen Konecki

Varaždin, lipanj 2023.

Luka Mrkonjić

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

Tema ovog rada je izrada strateške videoigre unutar programa Godot. Kroz rad će se proći kroz detalje samog alata, u što spadaju njegova povijest, stanje na tržištu te prednosti i mane u usporedbi s drugim alatima. Zatim će biti objašnjene mehanike i funkcionalnosti same videoigre. U konačnici proći će se kroz sve bitne logičke cjeline koje će biti potonje objašnjene.

Ključne riječi: algoritmi, programiranje, godot, razvoj videoigara, strateška videoigra.

Sadržaj

1. Uvod

Videoigre su jedan od mojih hobija, koje su mi oduvijek bile fascinantne. Logičko povezivanje različitih kreativnih grana kao što su crtanje, 3D modeliranje, kreiranje glazbe i slično, po meni predstavlja jedan od najzanimljivijih mogućih slučajeva s kojim programer može vršiti raditi, a i u konačnici sama interakcija koju krajnji korisnik ima s videoigrom. Upravo mi je programski dio izrade videoigara oduvijek bio najfascinantniji, jer se u njemu konzistentno nalaze različite vrste problema koje programer mora riješiti na pametan način, često misleći izvan kutije. Iako sam proučavao neke programske probleme i rješenja s kojim se drugi kreatori videoigara suočili, dosad sam nisam nikako napravio videoigru, što mi je oduvijek bila želja.

S ovim radom želim spojiti svoju želju za izradom videoigre s proučavanjem game engine-a Godot. Godot je relativno novi game engine na tržištu pa mi je s ovim radom ujedno cilj uvidjeti koje su to glavne prednosti Godot-a u usporedbi s drugim sličnim alatima na tržištu te kakvu budućnost može očekivati.

Tako će se prva cjelina ovog rada baviti samim game engineom. Objasniti će se što je uopće game engine, što je Godot, njegova povijest i radna okolina. Provjerit će mu se stanje na tržištu i prikazati će se njegove prednosti i mane.

Poslije toga objasniti će se kako bi sami projekt trebao funkcionirati. Ukratko će se proći kroz glavnu inspiraciju za sami projekt te će biti objašnjene mehanike samog projekta. Zadnji i najveći dio će prolaziti kroz tehničke detalje projekta. Iako se ne će proći kroz svaku liniju koda za najbitnije funkcionalnosti algoritam će biti objašnjen kao i način na koji to sve skupa funkcionira.

2. Metode i tehnike rada

Iako će se ovaj rad kratko dotaknuti teoretske tematike Godot-a, rad će pretežito biti projektne prirode i kao takav, tu nema nekih kompleksnih metodika istraživanja. Međutim jedna od metodika koja će se koristiti za analizu popularnosti Godota će biti pregled statistika na Google Analitycsu te Similar.

Programska strana ovog rada je nešto kompleksnija. Projekt će se izvršavati unutar Godot 4.0.3. Mono verzije, jer će program biti pisan unutar C#. Kao editor koda i sustav za debugiranje koristiti će se Visual Studio Code. Za kreaciju mapa i pojedinih likova koristiti će se Blender 3.5. Za modifikaciju, izradu likova kao i generalno uređivanje slika koristiti će se Gimp. Za verzioniranje rada će biti korišten Git, a osim lokalnog repozitorija rad će biti spremljen na GitHubu.

Link za online git repozitorij je idući: https://github.com/lukamrko/3dRpgMy.

Ovaj rad će biti pisan unutar Worda. Od dodatnih programa tu će biti korišten Chocolatey za preuzimanje svih programa, LightShot za dohvaćanje slika zaslona te Gimp za uređivanje slika.

Pošto izrada videoigri vrlo lako postane hardverski intenzivan zadatak, smatram da je jako bitno navesti specifikacije računala na kojem ću obavljati rad. Rad će biti obavljen na stolnom računalu s procesorom IntelCore i7-3770, 3.4GHz, grafičkom karticom Graphics Radeon RX 580 Series, radne memorije 16GB DDR3 667 MHz i operacijskom sustavu Windows 10.

Naposljetku za sami projekt sam koristio predložak rada otvorene licence "Godot Tactical RPG". Navedeni projekt je napisan u potpuno drugom jeziku od mog projekta, ali je i dalje moj projekt jako inspiriran dijelovima tog projekta. Također većina slika za moj projekt je direktno ili modificirano preuzeta iz navedenoga projekta.

Pozadinsku glazbu bez naknade sam pronašao na YouTubeu na idućem linku: https://www.youtube.com/watch?v=s5668QCRabg. Za zvukove, koji nastaju tijekom napada, koristio sam besplatne zvukove preuzete sa stranice: pixabay.com

3. Teorijska pozadina Godota

Kroz ovu cjelinu proći će se kroz teoretsku pozadinu Godota. Prvo će biti objašnjeno što je to uopće game engine, zatim će se objasniti što je to Godot i ukratko će se proći kroz njegovu povijest. Nakon toga će se pregledati trenutno stanje Godota na tržištu u usporedbi s drugim game enginima te će biti dati razlozi zašto bi netko koristio Godot. Na kraju ove cjeline ukratko će biti prikazana radna okolina Godota.

3.1. Game engine

Prije nego li se krene objašnjavati Godot, potrebno je razumjeti što je to game engine. U početku razvoja videoigara, u 1980-im i ranim 1990-im, razvoj je se najčešće sastojao od nekolicine ljudi koji su jednu igru ciljno radili za jednu platformu. Međutim s vremenom zbog poboljšanja hardware bilo je moguće raditi kompleksnije programe pa tako i kompleksnije igre. S kompleksnijim igrama dolazi i do većeg obujma posla, što dovodi i do novih problema. Neki od novih problema su bili što za svaku novu igru treba "izmišljati kotač ponovno", to jest mnogo logičkih stvari koje je napravljeno u nekim prijašnjim igrama treba ponavljati i ponovno pisati, što znači da developeri nisu imali neki standard tijekom rada. Jedna od težih stvari tijekom ovog prepisivanja bi bila fizika. Također paralelni rad u grupama postaje otežan, jer je praktički ne moguće raditi na nekoj funkcionalnosti koja ovisi o nekoj drugoj funkcionalnosti koja dosad nije implementirana, ili samom činjenicom da bi dobar dio sustava bio usko povezan (engl. tightly coupled).

Kako bi se ovaj problem donekle riješio, nastali su game engini. Prema [2] game engine se može smatrati programom koji je proširiv i može služiti kao podloga za različite videoigre bez velike modifikacije. Važno je napomenuti da ne postoji neko službeno tijelo koje je dalo definiciju za game engine i da je za različite game engine potrebna različita razina modifikacije kako bi se kreirala nova igra. Također bitno je napomenuti da za izradu same videoigre nije potrebno koristiti već postojeći engine, već programer može napraviti svoj iako je ovo dosta težak pothvat. S druge pak strane također je moguće napraviti videoigru, bez da se pritom napravi neovisni game engine, odnosno direktno kodirati logiku same igre. Ovakve igre u pravilu najčešće nisu mehanički kompleksne i njihova daljnja proširivost je iznimno teška, ali u konačnici programska okolina na kojoj su napravljene može se smatrati kao game engine specifičan za tu videoigru.

Danas se na tržištu može pronaći dosta izbora za game engine, među kojim su najpopularniji Unreal Engine i Unity. Zajednička točka ovim enginu, kao i Godotu je što

omogućavaju višeplatformski razvoj, grafičko sučelje za rad s različitim stavkama videoigre i skriptama, ugrađena fizika, uvoz objekata (engl. asset importer) i još neke druge manje mogućnosti.

3.2. Godot i povijest Godota

Godot je besplatni, sveobuhvatni, multi-platformski game engine koji omogućuje lagano kreiranje 2D i 3D videoigara [3].

Predstavljen je javnosti kao *open source* game engine početkom 2014. godine [4]. Izvori o Godot-u prije ovog vremena su praktički nepostojani, ali ono što se može znati je da ga je Juan Lenitsky započeo s radom na kosturu samog engine još 2001.godine [5]. Iako je ova inicijalna verzija Godota bila dosta primitivna, u njoj se započele mnogo stvari koje se danas uzimaju zdravo za gotovo, među kojim bih ponajprije istaknuo programski jezik GDscript o kojem će se nešto više pričati u kasnije dijelu rada.

Nakon početnog izbacivanja, Godot je konzistentno dobivao ažuriranja. Ažuriranja su praćenje brojkama, a prilikom ostvarivanja nekih većih prekretnica u mogućnostima samog engine, mijenjala bi se major verzija. Tako je trenutno najnovija verzija Godot 4.0.3.

Godot 1.0 i Godot 2.0 po mom mišljenju nije ponudio neke bitne promjene. Godot 1.0 je izašao krajem 2014. godine i njegov glavni fokus je bio ispravljanje bugova koji su bili pronađeni i nastali tijekom njegove javne objave [6]. Godot 2.0 je izašao rano u 2016. godini, te neki od njegovih glavnih značajki su bili poboljšanje rada s scenama, u što su spadala mogućnost nasljeđivanja scena, bolje instanciranja istih pa čak i mogućnost korištenja kontrole verzioniranja, zahvaljujući novom formatu u kojem su se scene spremale [7]

Godot 3.0 je verzija u kojoj je dodano mnogo novih velikih značajki, koje su ponovno uzdigle Godot na tržištu Game engina i zbog čega je Godot ponovno ušao u radijus game developera. Ova verzija izašla je početkom 2018. godine, i neke od glavnih značajki su bile:

- Potpora za pisanje skriptu unutar C# (prije je bilo moguće samo unutar GDscripta)
- Novi 3D render engine
- Masivna proširenja za materijale
- Novi model fizike zvan Bullet Physics

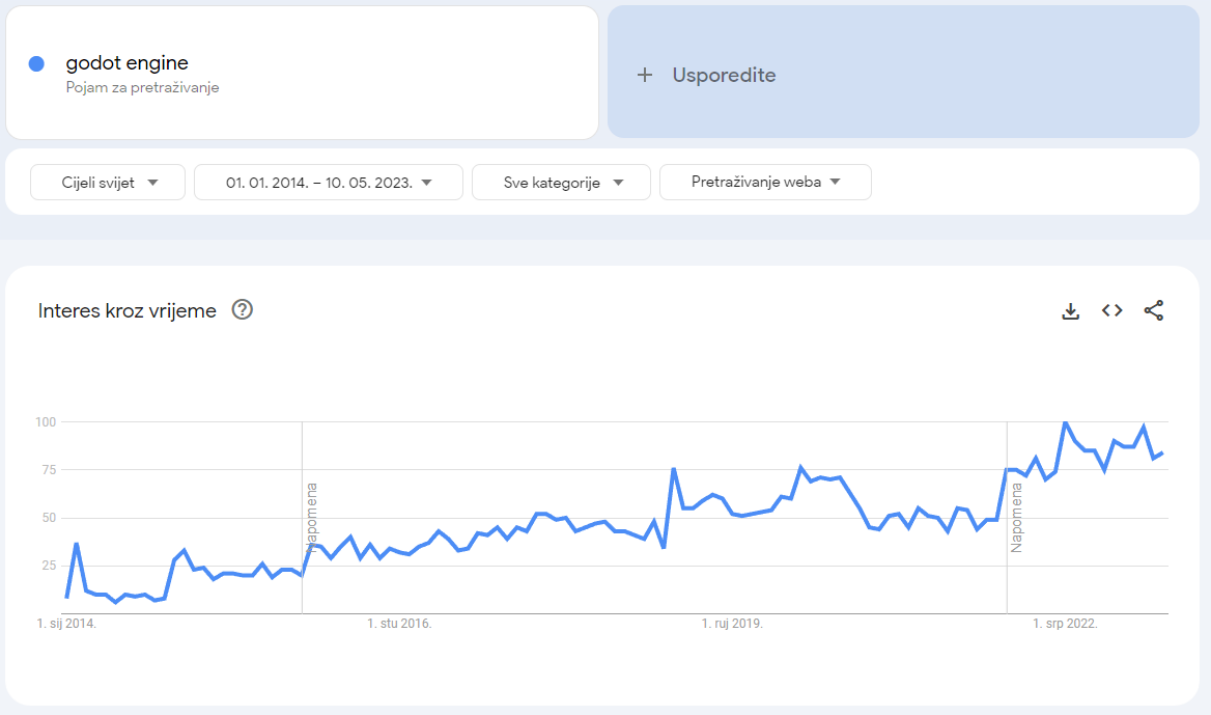
ali i mnoštvo drugih [8].

Zadnja u nizu velikih ažuriranja je Godot 4.0 verzija koja je službeno izašla 1.4.2023. S ovim najnovijim ažuriranjem Godot je ponovno dobio pregršt novih mogućnosti, od koje bih

specifično naveo novi 3D renderer u obliku Vulkana te pomicanje C# verzije s Mono na .NET [9].

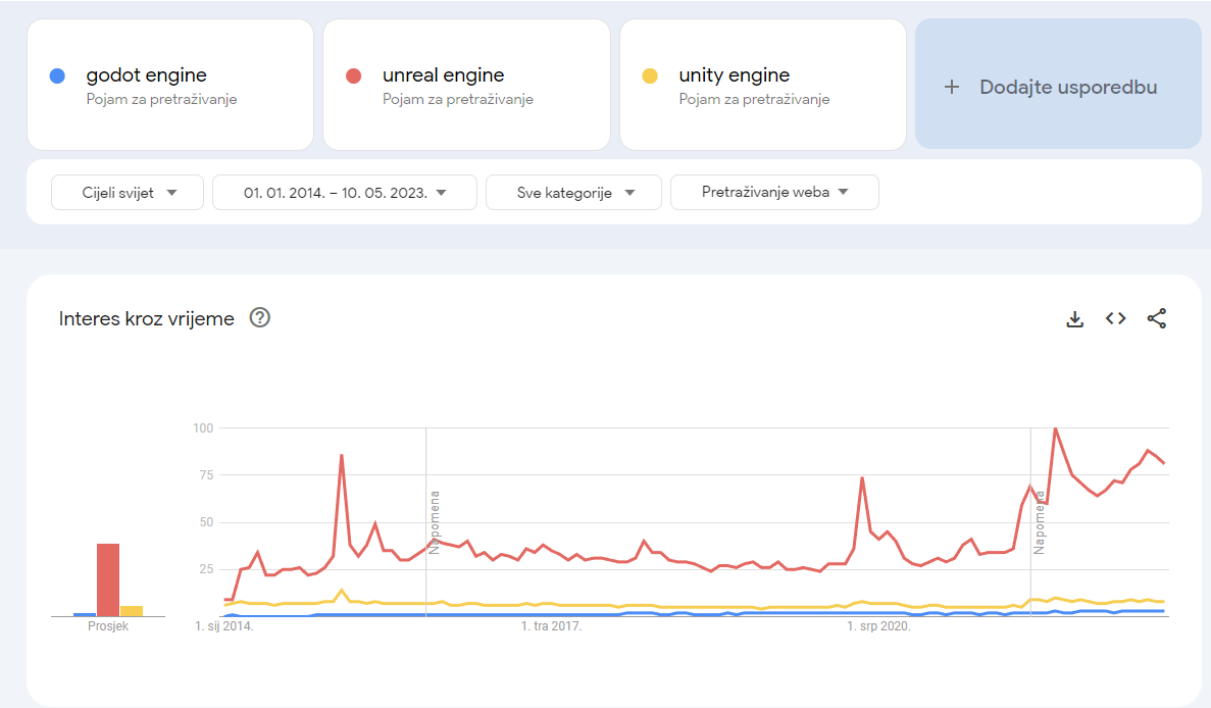
3.3. Stanje Godota na tržištu game engina

Kao što je ranije bilo spomenuto Godot je relativno mlad engine, što znači da je tek nedavno počeo konkurirati divovima na tržištu, kao što su Unity ili Unreal Engine. Iako ne postoji nužno neka konkretna brojka, popularnost pojedinog engina je moguće aproksimirati gledajući više različitih stavki i onda uzimanje zaključka iz toga. Prva stavka koja će se gledati je statistika pretraživanja.



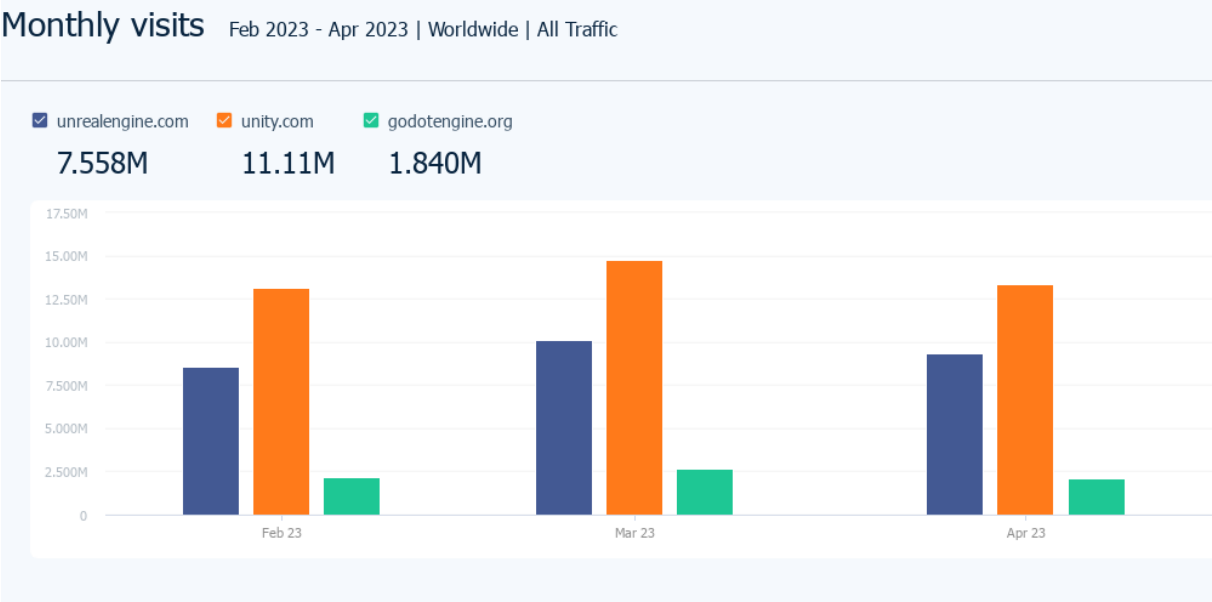
Slika 1 - Google trendovi za Godot (Izvor: [10])

Prema grafu sa slike 1 može se vidjeti kako od izlaska Godota pa sve do trenutka pisanja ovog diplomskog rada njegova popularnost raste. Dva velika porasta mogu se vidjeti tijekom početka 2019. i sad nedavno, tijekom početka 2023. Porast u 2019. se može pripisati izlasku verzije 3.1, koja je se pretežito doticala unaprjeđivanja i popravljanja značajki dovedenih u verziji 3.0. Nedavni porast početkom 2023. se može pripisati tome što je Godot službeno izbacio svoju 4.0 verziju, u kojoj je daleko glavna prednost bila unaprjeđenje 3D dijela sustava.



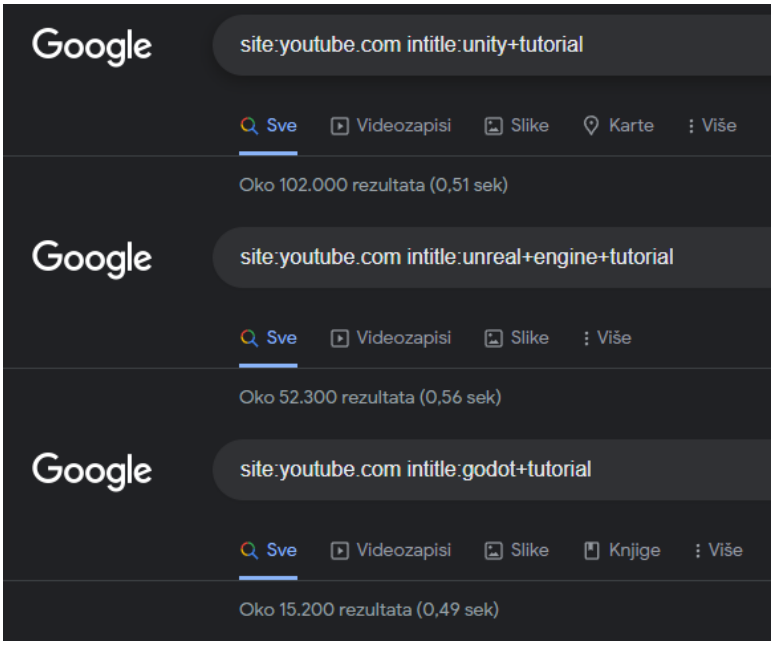
Slika 2 - Google trendovi za Godot(plavo), Unreal Engine(crveno), Unity(žuto) (Izvor:[11])

Međutim ako se sagleda slika 2 vidljivo je da prema popularnosti Godot obuhvaća minimalan udio u tržištu. Štoviše i Unity, engine koji je puno duže na tržištu i koji ima ogroman market zauzima isto jako malen udio u cjelokupnom tržištu.



Slika 3 - Similar web rezultati za Godot, Unity i Unreal Engine (Izvor: [12])

Prijašnji rezultati se također mogu odraziti i sa slike 3. Slika je kreirana koristeći stranicu Similar Web i prikazuje koliko mjesečnih posjeta u zadnja 3 mjeseca imaju prethodno spomenuta 3 engina. Iako je u usporedbi s google trends, situacija ipak nešto ravnopravnija, pa čak i činjenicom da Unity ima prednost nad Unreal Engineom, vidljivo je da Godot tu zauzima tek jedan mali dio, iako je verzija 4.0 izašla u 4 mjesecu.



Slika 4 - Youtube prikaz rezultata tutorial za pojedine engine. Kreirano koristeći Google pretraživanje

Ono što se vidjelo na ranijim rezultatima se ponovno slično može vidjeti i na slici 4. Tako je broj tutoriala na YouTubu, jednoj od najposjećenijih stranica za takvu vrsta sadržaja, ponovno mnogo manji za Godot nego li je za druga dva popularnija engina. Ova statistika se ponajprije tiče samostalnih i neovisnih kreatora videoigara pošto su oni pretežito ciljana publika takvih vrsta videa.

S ovim prethodnim primjerima nije korištena najpouzdaniju metriku za prikaz točno specifičnih podataka jer takva metrika ne postoji, ali smatram da je jako dobro ukazano da Godot još nije ni blizu dva glavna diva u industriji što se tiče popularnosti. Popularnost ili njen nedostatak ne nastaje zbog jedne stavke, ali po meni glavni razlog zašto je Godot na svojoj trenutnoj poziciji je što je puno mlađi engine od ostalih. Tako je primjerice prva verzija Unreal izašla 1998., dok je Unity 2005. Unreal engine je također imao dodatnu prednost što je sa svojim izlaskom ujedno izbacio i tada mega popularnu videoigru, *Unreal*.

Zbog starosti i jednostavno količine podataka i mogućnosti koji su ostali engini akumulirali kroz vrijeme, nerealno je pretpostaviti da će Godot to u bilo koje skorije vrijeme zasjeniti jedan od ova dva diva. Prema tome bolje je pitati kome je Godot namijenjen i zašto bi se netko odlučio na Godot, a ne na neki drugi engine.

3.4. Zašto Godot

Gledajući prošlo poglavlje pojedinac se može zapitati zašto bi uopće koristio Godot, kad postoje drugi game engini, koji u dosta aspekata čak i nude više nego li Godot. Po meni postoje dva glavna razloga, a to su iznimno mali zahtjevi sklopovlja i MIT licenca nad samim enginom.

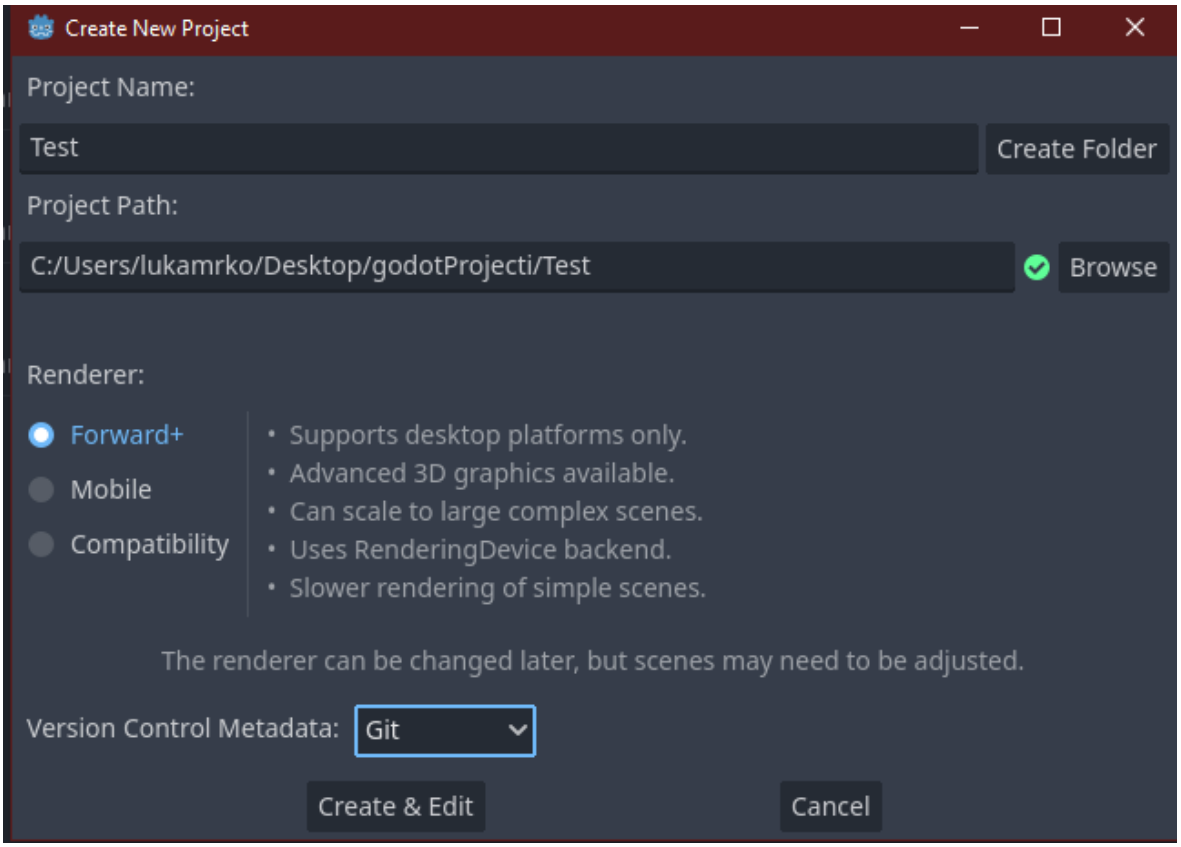
Pod male zahtjeve sklopovlja ubrajam dvije stavke. Prva stavka je iznimno malen prostor koji je potreban za instalaciju samog Godot-a. Tako verzija Godota, bez C# dijela, zauzima samo 50MB, dok je i C# verzija 60ak MB. Za usporedbu Unity zauzima 2GB, dok Unreal Engine zauzima 6GB. Druga stavka je sama računalna konfiguracija koja je potrebna za pokretanja Godota. Iako je nju teže empirijski odrediti, Godot se vrlo lako i brzo pokreće i na slabijim računalnim konfiguracijama. Kao razlog ovome, pretpostavio bih da pošto je engine mnogo novije rađen, tako je vjerojatno puno više pratio različite tehničke standarde i izbjegao nepotrebni (engl. *boilerplate code*) ili zastarjeli kod (engl. *legacy code*), što je dovelo do većeg nivoa performanse. Dodatni razlog je da neke stvari jednostavno još nedostaju pa ih samim time nije potrebno učitavati.

MIT licenca nad enginom podrazumijeva da je engine sam po sebi otvoreni kod (engl. *open source*). Ovaj razlog je po meni puno bitniji zbog fleksibilnosti koji nudi kreatorima. Pošto je kod samog engine open source, programer je u stanju dodati, promijeniti pa čak i ukloniti bilo koju značajka engina kako mu to odgovara. Čak i u slučaju da se naiđe na neki bug, nije nužno čekati izdavača engina da ga popravi, već ga u teoriji svaki programer može sam popraviti. Zadnje, i po meni najbitnije, je činjenica da pošto je MIT licenca nad samim enginom, svaki rezultat rada na tom enginu, što bi u ovom slučaju uglavnom bile videoigre, kreator ima potpunu slobodu nad tim rezultatom. Od tih sloboda najznačajnija je da kreator nije dužan platiti nikakvu tantijemu samom Godotu kada prodaje igru, što nije slučaj drugih game engina.

Uzimajući navedeno u obzir ja bih rekao da su glavna meta tržišta Godota mali i srednje neovisni (engl. *indie*) kreatori videoigara. U ovaj dio tržišta posebice bih naglasio Unity dio, pošto oba imaju veliku podršku neovisnim kreatorima videoigara, oba mogu koristiti C\# za svoje skripte i tijek rada (engl. *workflow*) im je pretežito sličan, ali kod Unityja postoje određene tantijeme koje je potrebno platiti kad se dođe do neke razine prodaje, dok Godot to nema.

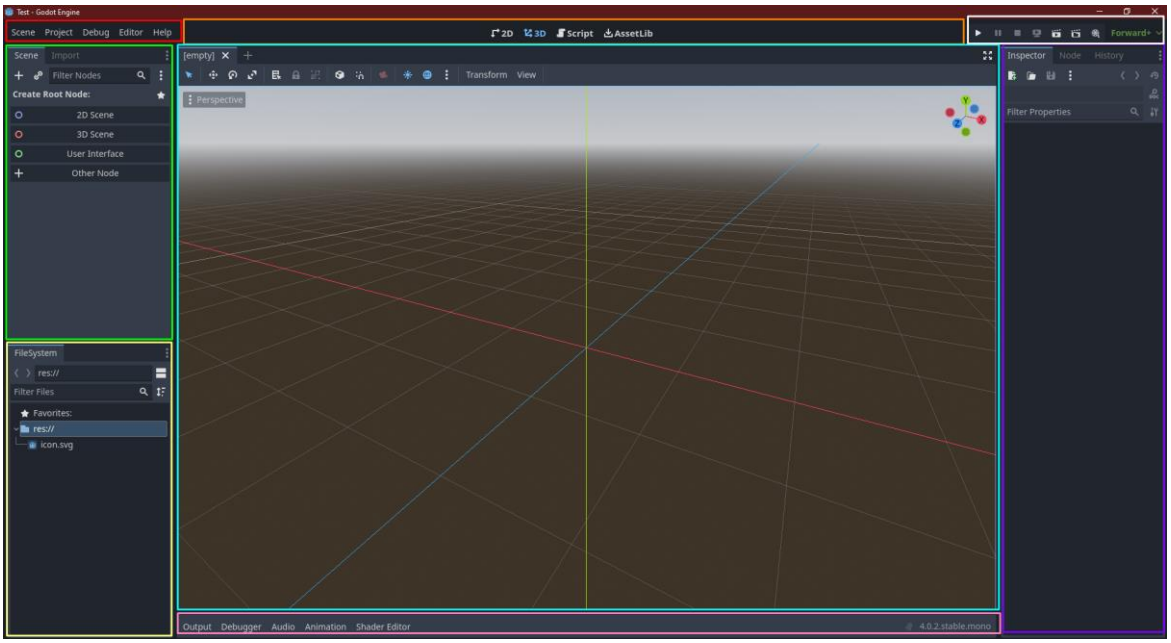
3.5. Godot radna okolina

Prilikom pokretanja Godota otkriva nam se prozor \textit{Project Manager}, u kojoj se nalaze korisnikovi trenutno kreirani projekti. Ova okolina trenutno nije bitna te za idući korak potrebno je kliknuti na *New Project* u gornjem desnom kutu. Klikom na taj gumb otvara nam se nova mini izbornik kao na slici 5.



Slika 5 - Tipovi projekta unutar Godota

U ovom dijelu se daje naziv projektu, ali i puno bitnije odlučuje s kojim Rendererom će Godot raditi. Glavna ideja tih renderera je da što imaju više i bolje opcije, time mogu ići na manji broj platforma, odnosno ne će biti moguće izvesti igru nad mobilnim i web platformama. Pošto ću se u ovom radu fokusirati na izradu igre za Windows i Linux, a ne mobilne igre ili web igre, koristiti ću Forward+ renderer. Prilikom kreacije projekta otvara nam se prozor kao na slici 6.



Slika 6 - Glavni ekran u Godotu

Na slici se može vidjeti zadana scena koja nastaje kreiranjem novog projekta. U gornjem lijevom crvenom pravokutniku nalaze se generalna opcije za projekt, u kojima se između ostalog nalaze, spremanje i učitavanje scena, dokumentacija, kao i postavke za projekt, ali i cjelokupni editor.

Unutar zelenog pravokutnika se nalaze svi čvorovi (engl.*node*) koji tvore trenutačnu scenu. Upravo ovaj dio predstavlja osnovni građevni blok svake scene i zbog toga svaka scena mora sadržavati bar jedan korijenski (engl. *root*) čvor. Iako u ovom slučaju Godot kao korijenske čvorove nudi 2D scenu, 3D scenu i User Interface, najosnovniji korijenski čvor koji je moguće napraviti unutar Godota predstavlja čvor koji se samo zove *Node* i koji ima najosnovnije karakteristike čvora. Bilo bi besmisleno ovamo govoriti o svakom čvoru, ali više o njima će biti rečeno kad se dođe do pojedinih scena unutar projekta.

U narančastom pravokutniku se nalaze različiti pogledi na trenutnu scenu. 2D i 3D nude pregled scene u zadanom načinu, *Script* predstavlja pregled skripte nad trenutnom scenom ili objektom, dok koristeći *AssetLib* je moguće dodati neke vanjske scene sa svojom logikom u trenutni prostor. Tirkizni pravokutnik je povezan s narančastim pravokutnikom u smislu da on prikazuje ono što je odabrano unutar narančastog pravokutnika. Tako će se prilikom prikaza 2D ili 3D scene, vizualno moći uređivati pojedine objekte ili čvorove unutar trenutne scene. Unutar *Script* pregleda otvara se ugrađeni editor koda za skriptu nad zadanom scenom ili čvorom.

Unutar bijelog pravokutnika se pokreće projekt. U njemu je također moguće pokrenuti samo neku specifičnu scenu, kao primjerice nivo radi dodatnog testiranja, odabrati renderer, pa čak i pokrenuti scenu kao video.

U ljubičastom pravokutniku se nalaze sva svojstva trenutno označenoga čvora, kao i ugrađeni signali koje je moguće povezati na pojedini čvor. Funkcionalnost za automatske signale nije najbolje realizirana za C# pa je u ovom projektu ta funkcionalnost zanemarena, a realizirana je kroz kod.

Unutar žutog pravokutnika se nalazi preglednik datoteka cjelokupnog projekta. Tu bi se spremaju scene projekta, ali je poželjno spremiti i sve ostale potrebne resurse za igru kao što su glazba, teksture, materijali i slično, kako bi se njihovoj putanji moglo lokalno pristupiti. Pomoću ovog preglednika je moguće mijenjati trenutno odabranu scenu.

U rozom pravokutniku na dnu nalaze se neke izlazne informacije, kao što su audio, animacije ili uređivač shadera. Meni najkorisniji dio ovog prozora je bio što se tu ispisuju bilo kakvi problemi s gradnjom projekta, iznimke (engl. *exception*) koje su bačene tijekom pokretanje same aplikacije, kao i izlazi same konzole, to jest korištenjem jednostavnog koda *GD.Print()* moguće je ispisati bilo koji izraz unutar koda u konzolu.

4. Detalji projekta i funkcionalnosti

4.1. Glavna inspiracija za projekt - "*Into the Breach*"

Glavna inspiracija za projektni dio ovog diplomskog rada je neovisna videoigra *Into The Breach* od autora i izdavača *Subset Games*. To je strateška igra na poteze u kojoj igrač upravlja s tri *mecha* lika i gdje se bori protiv divovskih insekta. Za razliku od uobičajenih strateških igri na poteze, gdje postoji dobar dio igre koji ovisi o nasumičnom faktoru, *Into the Breach* je to praktički u potpunosti eliminirao i igra je skoro u potpunosti deterministička.

Deterministička prirode igre je postignuta na način da je igra podijeljena u tri faze. U prvoj fazi insekti se stvaraju na mapi, pomiču i biraju svoje akcije. Zatim igrač pomiče svoje mechove, ali igračeve akcije su odmah učinjene. Kada igrač završi sa svim svojim akcijama, insekti nastavljaju sa svojim potezom, te sada čine akcije koji su odredili u prvoj fazi.

Slika 7 - Prosječni ekran unutar igre Into The Breach(Izvor [13])

Kod svih prethodno spomenutih akcija njihov ishod je poznat. Primjerice u prvoj fazi kada insekti biraju svoju akcije, one su igraču vidljive. Tada ukoliko insekt odabere napasti polje desno od sebe, tada će igrač vidjeti tu akciju. Također šteta koju će insekt učiniti je statična i ne može nasumično varirati. Kad igrač igra vidljivo mu je dokle se može kretati, šteta koju čini insektima je također statična i polje na koje ih pomiče sa svojim napadom.

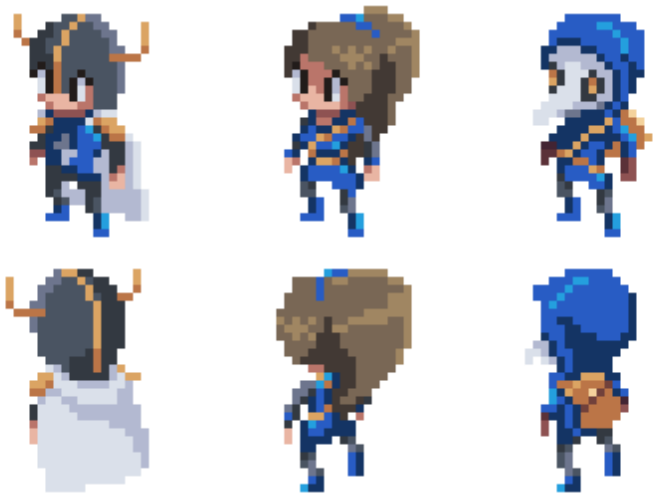
Uvjeti za izgubiti i pobijediti pojedini nivo su dosta jednostavni. Igrač gubi ukoliko mu sva tri mecha umru, ili ukoliko mu insekti unište previše resursa (resursi su u ovom slučaju polja na kojim se nalaze zgrade, elektrane i slično). Igrač dobiva pojedini nivo ukoliko nije izgubio određeni broj poteza.

4.2. Projekt ovog rada - "*Heroes and Bones*"

Projekt ovog diplomskog rada je videoigra zvana *Heroes and Bones*. Videoigra će biti jako slična prethodno spomenutoj videoigri *Into the Breach*. Igrač će preuzeti ulogu nad tri lika, boriti će se protiv pet različitih vrsta protivnika i bit će četiri nivoa. Način na koji će igrač izgubiti igru je da izgubi dva od tri totema ili sve svoje likove. Igra se može podijeliti u tri faze koje jako nalikuju na *Into the Breach*. Detalji svake faze će biti spomenuti u kasnijem podpoglavlju *Pravila i faze*.

4.2.1. Likovi

Kao što je spomenuto ranije igrač kontrolira tri različita lika, a to su vitez, strijelac i alkemist. Računalno kontrolirani protivnici imaju pet različitih likova, a to su kostur-ratnik, kostur-strijelac, kostur-bombaš, kostur-doktor i kostur-heroj.



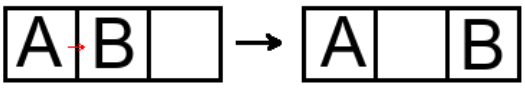
Slika 8 - Prikaz igračevih slika s lijeva na desno: vitez, strijelac i alkemist

Svi likovi dijele atribute da imaju određen broj polja do kojeg se mogu pomaknuti u jednom potezu, visinu koju mogu skočiti, udaljenost polja s kojeg mogu napasti, štetu koju rade te količina zdravlja koju imaju. Za sve likove je važno napomenuti da iako su oni ponašaju kao 3D modeli, na nivou su prikazani pomoću 2D slika.

4.2.1.1. Igračevi likovi

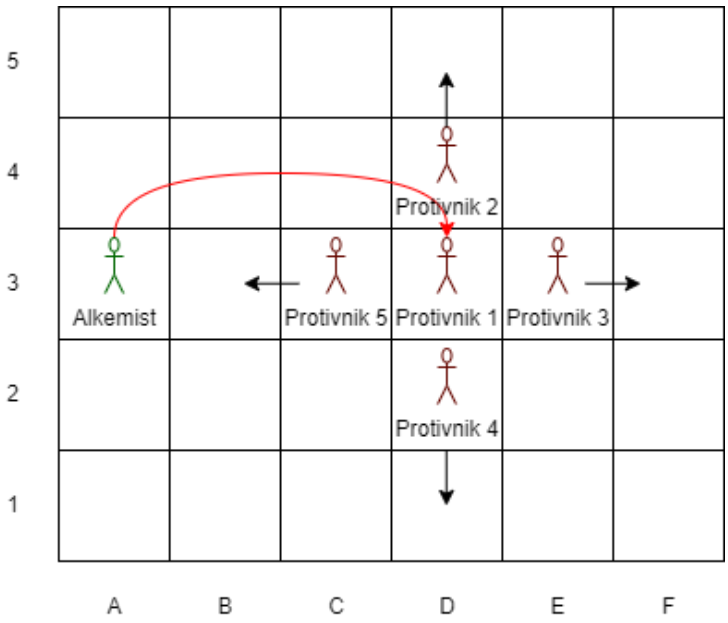
Na slici 8 su vidljive slike igračevih likova. Vitez ima najviše zdravlja od igračevih likova i udara najjače, ali zato ima najmanji broj polja koje može proći na mapi, kao i najmanji skok. Strijelac s druge pak strane, ima najveću mobilnost, u smislu da može proći najviše na mapi kao i skočiti najviše od igračevih likova, može udariti s udaljenosti, ali za razliku od viteza ima nešto manje zdravlja i slabija udara.

Važna stvar koju oba ova igračeva lika rade je što sa svojim udarcem guraju protivnika u drugu stranu, i to ponašanje je vidljivo na slici 9. Neka detaljnija ponašanja prilikom guranja će biti spomenuta kasnije



Slika 9 - Ako igrač A koji je lijevo udari protivnika nad B, tada se protivnik nad B miče udesno

Alkemist je što se tiče svojih statusa mobilniji od viteza, ali manje je mobilan od strijelca, te ima najmanje zdravlje od igračevih likova. Ono što njega čini izuzetno posebnim je vrsta napada koju posjeduje.



Slika 10 - Alkemistov napad. Protivnik1 će izgubiti jedno zdravlje, a ostali protivnici će biti gurnuti.

Za početak udara protivnika kojeg je naciljao na polju i radi minimalnu štetu, odnosno oduzima mu jedno zdravlje. No zatim se gledaju sve četiri strane svijeta u usporedbi s početno pogođenim poljem i ukoliko je protivnik pronađen na nekoj od strana, pomiče se u tom smjeru i pritom mu se ne radi direktna šteta.

Primjerice ako je postojao lik iznad protivnika kojeg je originalno pogodio, taj lik će se pomaknuti jedno polje gore. Ovaj udarac je dosta fleksibilan jer je moguće pogoditi u prazno polje ili čak prijateljskog lika kako bi došlo do ove reakcije. Ovo ponašanje je vidljivo na slici 10.

4.2.1.2. Protivnički likovi

Kao što je ranije spomenuto, svi likovi imaju neke zajedničke atribute, a jedna od glavnih razlika između protivničkih i igračevih likova je da protivnički likovi imaju puno bolju mobilnost na mapi, i što se tiče same kretnje i što se tiče skakanja, ali zato imaju puno manje zdravlja i rade dosta manju štetu. Ovo je balansirano činjenicom da protivnici započinju nivo s četiri lika, te im se svaki potez stvore dodatna četiri lika, s čime jako brzo brojčano nadjačaju igračeva tri lika. Također bitno je napomenuti da protivnički napadi ne pomiču likove po mapi.

Kod protivnika kostur ratnik se može gledati kao najosnovniji lik po kojim se ostali likovi mogu ravnati. Posjeduje četiri zdravlja, a može napasti u bliskom dosegu pritom radeći minimalnu štetu od jednog zdravlja. Kostur-strijelac za razliku od kostur-ratnika ima samo jedno zdravlje, također radi minimalnu štetu, ali zato ima domet od tri polja. Može puno više skočiti, a i samo količina kretanja mu je nešto superiornija od kostura ratnika. Po svojoj dinamici ova 2 kostura dosta naliče na igračeve likove viteza i strijelaca, ali razlika ipak postoji.

Kostur-bombaš je jedinstven lik po tome što nikad neće proživjeti dulje od jednog poteza. Razlog ovome je što na kraju svog poteza eksplodira, te čini maksimalnu štetu svim susjednim poljima, što ga čini jednim od najopasnijih likova u igri. Također njegov skok i kretanje na mapi je u praktičnom smislu neograničeno, što znači da može doći do bilo koje točke na mapi. Ovaj lik također nema klasični napad, nego će približavanjem svog cilja započeti svoju akciju eksplodiranja. Ukoliko i igrač ubije ovog protivnika, akcija eksplodiranja će biti aktivirana pa zbog toga treba biti dodatno oprezan.

Kostur-doktor kao što mu ime nalaži liječi prijateljske likove i on je jedina takva vrsta lika u igri. Njegovo liječenje funkcionira na principu da ako je trenutno zdravlje nekog protivničkog lika manje nego li je njegovo maksimalno moguće zdravlje, tada će mu kostur-doktor obnoviti jedno zdravlje. Ovo liječenje nema neki minimalni potreban radijus kako bi došlo do njega, odnosno funkcionira preko cijele mape. Lik je sličan kosturu-bombašu po tome što nema nikakav napad. Međutim kombinacija da može liječiti s neograničenog dometa kao i to da lik nema napad je dovela do toga da bi idealna strategija lika uvijek bila da samo stoji

tamo gdje se kreira i liječi svoje kosture. Pošto ovo igraču ne bi bilo zabavno učinjeno je da se kostur-doktor i dalje približava igraču, ali će doći maksimalno na dva polja od igrača. U usporedbi s nekim drugim alternativama ovo mi se činio kao najbolji omjer poštenosti igre i odluke igrača što mora napustiti povoljniju poziciju kako bi se suočio s ovim protivnikom.

Zadnji lik je kostur-heroj. Gledano čisto po samim atributima kostur-heroj je najjači lik u igri, ali je zato najsporiji kostur. Tako on ima devet zdravlja, dok druga lik u igri što se tiče zdravlja, vitez ima samo šest, a i udarci mu odnose tri zdravlja. Što se tiče svog ponašanja on se može smatrati zapravo pojačanom verzijom kostur-ratnika, ali za razliku od kostur-ratnika on ima još jedan jedinstveni efekt. Taj efekt je da ukoliko se bilo koji protivnički lik u trenutku napada nalazi blizu njega, tada će napad tog lika oduzimati jedno dodatno zdravlje.

Slika 11 - Svi protivnički likovi. Od lijeva na desno: kostur-ratnik, kostur-strijelac, kostur-bombaš, kostur-doktor i kostur heroj. Luk preuzet s [14]

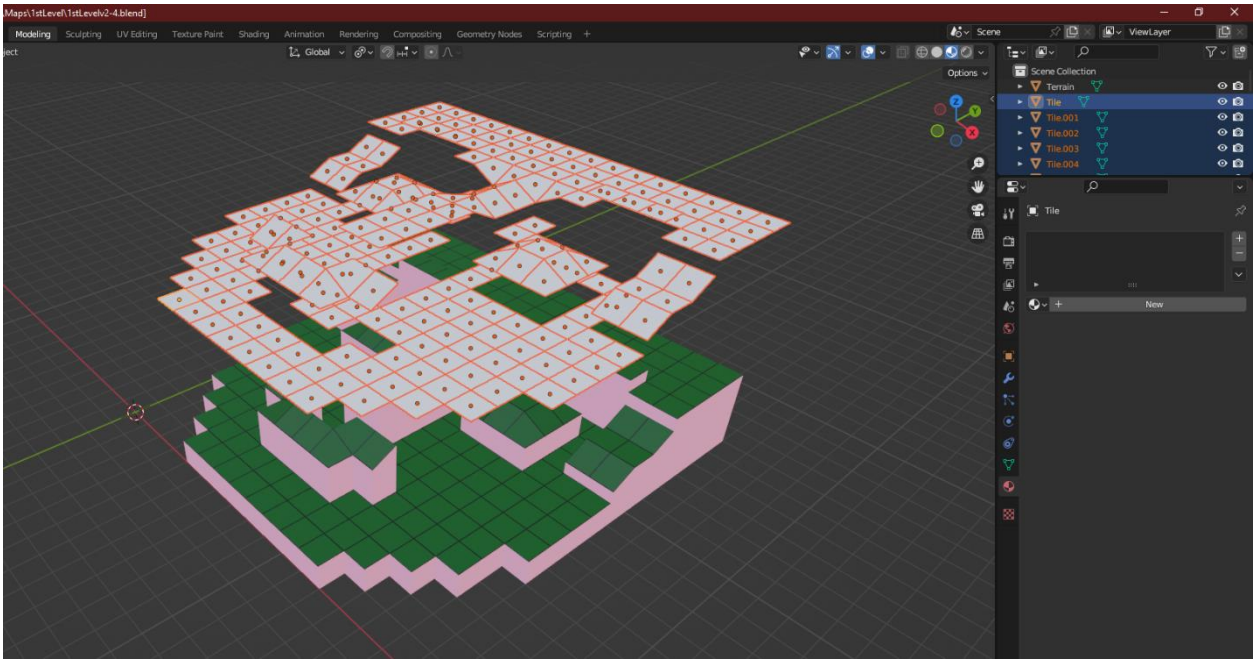
4.2.2.Nivoi

Prilikom izrade nivoa glavna stvar na koju je trebalo obratiti pozornost je činjenica da se likovi kreću po poljima (engl. *tiles*). Ta polja su ovdje u obliku kvadrata, te s jednim pokretom je moguće pomaknuti se za jedan kvadrat vertikalno ili horizontalno, ali ne dijagonalno. Za dijagonalnu kretnju bilo bi potrebno potrošiti dva boda za kretnju.

Druga bitna stvar prilikom izrade nivoa je činjenica da su nivoi trodimenzionalni. Ova trodimenzionalnost se u ovom projektu pretežito očituje kroz način na koje se likovi miču kroz nivoe, to jest ako je pojedino polje previsoka za pojedinog lika on će ga morati zaobići.

Sami nivoi su građeni unutar Blender-a. Unutar Git repozitorija postoje kratke upute kako se nivoi grade, ali u ovom radu se neće detaljno opisati jer to nije fokus rada. Bitan je način na koji su modeli posloženi kako bi igra mogla funkcionirati.

Model je prvo kreiran, uzimajući cijelo vrijeme u obzir da gledano s ptičje perspektive plohe budu jednaki kvadrati. Nakon što je izgled mape zadovoljavajući sve gornje plohe su duplicirane, skrivene, te zatim učinjene svojim posebnim tijelom. Izgled ovakvog objekta je vidljiv na slici 12.



Slika 12 - Primjer mape. Polja su podignuta radi dodatne vidljivosti osnovnog terena.

Ova podjela je učinjena radi lakšeg implementiranje logike u igri, a detalji ove logike će biti objašnjeni u kasnijim cjelinama, koje se budu doticale tehničkog djela. Zasada je važno reći da je samo glavni teren vidljiv igraču cijelo vrijeme. Na tom terenu se nalazi zeleni materijal za podlogu, smeđi materijal za zidove (na prethodnoj slici rozi, ali unutar igre smeđi), te crni materijal da se lakše vidi podjela samih polja.

Nevidljive polja unutar igre se nalaze malo iznad terena i većinu su vremena skriveni, međutim na njima se zapravo obavlja logika. Također iako su većinu vremena nevidljiva igraču, upravu su ta polja ono što će se obojati kako bi igrač znao što može napasti ili dokle može ići. Prema svemu ovome može se reći da je teren statički dio nivoa, dok su polja dinamički.

4.2.3.Pravila i faze

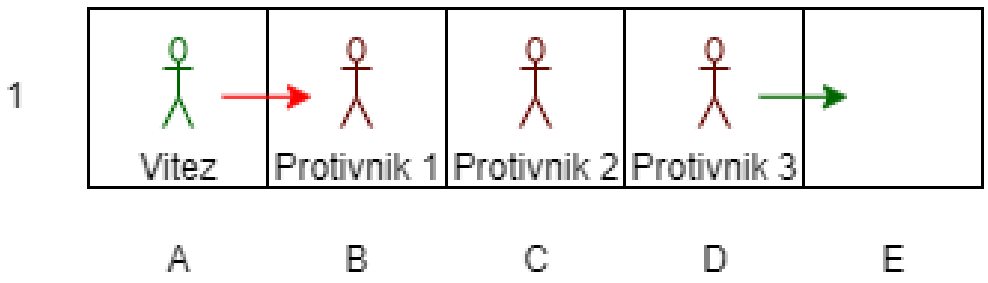
Kao što je ranije u radu spomenuto igra ima tri faze. U ovoj cjelini će biti detaljno opisano šta sve točno spada u pojedinu fazu i njihovu međusobnu interakciju.

Prva faza je protivnička. U njoj protivnici prvo kalkuliraju što mogu napasti, zatim se pomiču prema odabranoj meti i ovisno jesu li u dometu mete, pripremaju svoj napad. Za napad je važno napomenuti da je on relativan. Primjerice protivnik namjerava napasti igrača, koji se nalazi jedno polje sjeverno. U drugoj fazi protivnik je poguran jedno polje južno. Protivnik će kasnije učiniti svoj napad, ali igrač ne će biti oštećen već se napad dogoditi na polju koje se nalazi za jedan kvadrat sjeverno od protivnika.

Na tom polju je u međuvremenu možda došao drugi igrač, protivnik ili pak ništa, te će napad biti izvršen nad njima. Još jedna bitna specifičnost ove faze je da protivnici mogu imati dvije strategije. Jedna strategija je da udaraju igračeve likove koje su im najbliži, dok im je druga da se u potpunosti fokusiraju na toteme kako bi dobili.

Druga faza je igračeva faza. Igrač kao i protivnik se može kretati i izvršavati napade, ali ključna stvar kod igrača je što njegovi napadi guraju likove po ekranu. Ovu funkcionalnost je zapravo srž same igre, jer će protivnici praktički uvijek brojčano biti nadmoćniji nad igračem.

Kako igračeve jedince guraju protivnike već je ranije bilo spomenuto, ali postoje neki specifični slučajevi. Važna stvar koju treba imati prilikom guranja je domino efekt koji povlači za sobom. Naime ako se protivnik gura na polje, gdje se već nalazi neka drugi protivnik, tada originalni protivnik na kojem je započeto guranje ne će biti gurnut već će gurnut biti drugi protivnik. Oba protivnika će izgubiti jedno zdravlje, no samo će posljednji protivnik biti gurnut. Ovaj efekt je rekurzivan, to jest ako se u lancu nalaze četiri protivnika, samo će zadnji protivnik u lancu zapravo biti poguran, dok će svi protivnici u lancu poprimiti određenu štetu. Primjer ovog domino efekta je vidljiv na slici 13.



Slika 13 - Domino efekt. Nakon što vitez udari protivnika 1 samo će protivnik 3 biti poguran dalje. Protivnik 1 i protivnik 3 će izgubiti jedno zdravlje, dok će protivnik 2 izgubiti dva zdravlja (jedno jer ga je netko gurnuo, jedno jer je nekog gurnuo)

Efekt guranja također prestaje ukoliko zadnja lik u lancu je gurnut na polje koje je previsoko za njega. U tom slučaju će primiti štetu kao da je bila uguran u drugog lika, ali lanac guranja prestaje. Lik također na ovaj način mogu biti izbačene s mape. Ukoliko je pojedini lik izbačen s mape automatski umire.

Treća faza je ponovno protivnička faza u kojoj protivnik samo odradi akciju koju je odlučio na kraju svoje prve faze. Po završetku ove faze potez završava i započinju provjere je li nivo gotov, odnosno je li nivo dobiven ili izgubljen. Nivo se vodi da je izgubljen ukoliko je igrač izgubio sve svoje likove ili dva od tri totema.

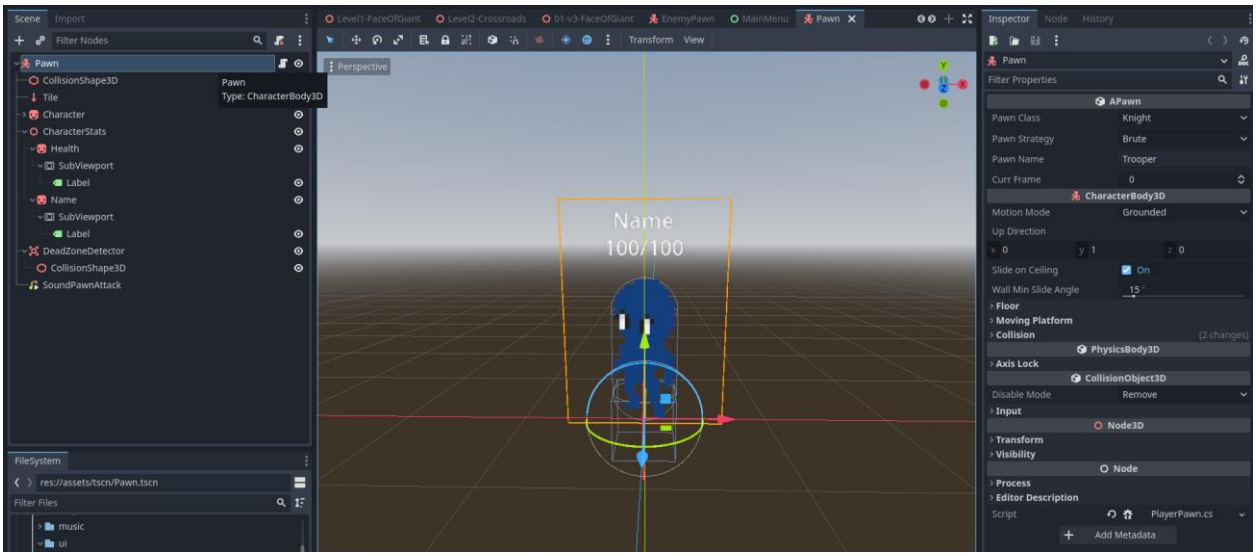
Ako igrač nije izgubio, a prošao je određen broj poteza, koji je prikazan u gornjem desnom ekranu tada igrač dobiva nivo i može preći na idući. Ukoliko nijedan od ovih uvjeta nije ispunjen tada se izvrše pripreme za idući potez. U ove pripreme spada stvaranje novih protivničkih likova na fiksnim lokacijama, kao i osvježavanja svih postojećih likova u smislu mogućnosti izvršavanja akcija i kretanja.

5. Tehnički detalji projekta

Ovo poglavlje će se baviti tehničkom izvedbom samog projekta. U prvom dijelu bit će objašnjena tema osnovnih građevnih blokova u samom Godotu, dok će se u kasnijem dijelu napraviti detaljna analiza ključnih dijelova koda i njegova interakcija s različitim dijelovima Godot sustava.

5.1. Osnovni građevnih blokovi Godota - čvorovi i scene

Čvorovi predstavljaju najmanju jedinicu unutar Godota. Najosnovniji čvor unutar Godota se jednostavno zove *node*. U nekom trenutku ostali čvorovi nasljeđuju ovaj čvor, direktno ili indirektno. Različite vrste čvorova uz najosnovniji tip, najčešće implementiraju i neko drugo sučelje. Zahvaljujući toj implementaciji čvorovi se međusobno razlikuju po svojim ponašanjima i svojstvima. Tako primjerice čvor za puštanje zvukova, ne će imati svojstvo za gravitaciju ili svoj 3D oblik, koji će čvor pod nazivom "*CharacterBody3D*" imati.



Slika 14 - Primjer scene igrača.

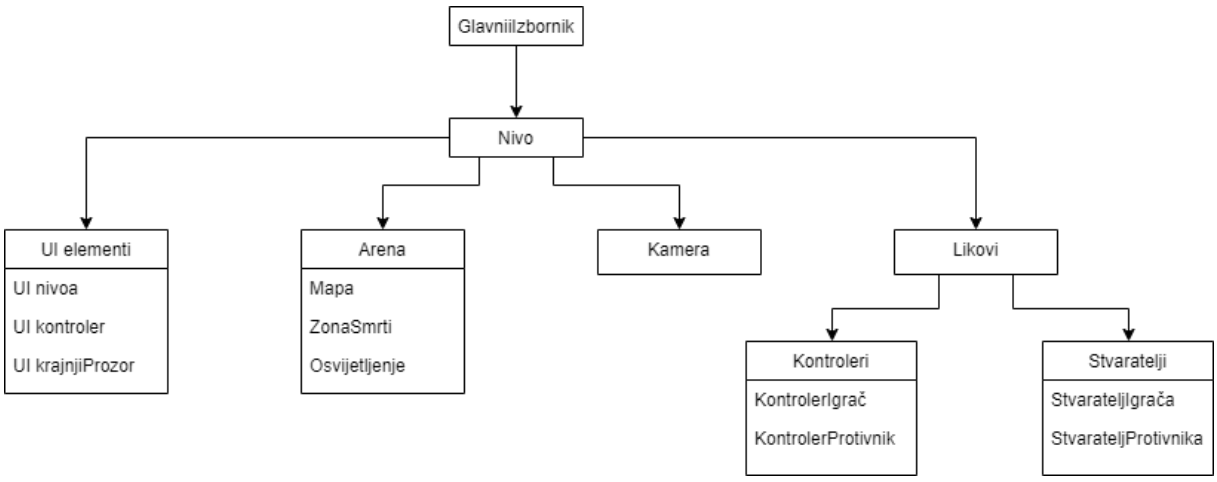
Iako postoji mnogo različitih, tri glavna čvora koji se koriste direktno ili indirektno preko svoje djece su "2D čvor" (engl. *2D Node*) koji se koristi za rad s dvodimenzionalnim objektima, "Kontrola" (engl. *Control*) koja se koristi za crtanje korisničkog sučelja te "3D čvor" (engl. *3D Node*) koja služi za rad s 3D objektima. Na različite čvorove je moguće umetnuti jedan ili više drugih čvorova, te s ovim pristupom se kreira stablo čvorova. Stablo čvorova se zatim zove scena. Primjer jedne ovakve scene je vidljiv na slici 14.

Na slici je vidljiva scena igrača. Na vrhu se sastoji od čvora "*CharacterBody3D*" te ima veću količinu svojih čvorova koji tvore različitu logiku lika. Tako se tu nalazi oblik kolizije, zraka svjetlosti, animator pa čak i neki dijelovi sučeljnih čvorova za prikaz zdravlja i imena. Jedanput napravljena scena se može ponovno koristiti. Tako se u ovom slučaju za kreiranja tri igrača, ova scena se instancira tri puta s različitim svojstvima. Upravo se s ovakvim korištenjem scena radi cjelokupna logika same igre.

Činjenicu koju je potrebno napomenuti je interakcija između koda i čvorova. Na apsolutno svaki čvor je moguće zakačiti skriptu, ali neko nepisano pravilo prilikom izrade igre je da se većinski koristi samo jedna skripta po sceni i to u samom korijenu scene. Razlog ovome je što je relativnim putanjama lako moguće doći do bilo kojeg čvora unutar scene i zatim tako prikupljenim čvorovima je vrlo lako moguće dodati različita ponašanja, signale ili pak promijeniti svojstva.

5.2. Opći pregled projekta

Na slici 15 se može vidjeti generalizirani prikaz kako sami projekt funkcionira. Pojedini dijelovi su izostavljeni radi jednostavnijeg prikaza, ali oni će biti kasnije objašnjeni kroz odgovarajuće cjeline.



Slika 15 - Opći pregled projekta

Na početku igre igrač ulazi u cjelinu izbornik. Ovdje se igraču učita te po potrebi generira dio konfiguracije za cijelu igru. Uz to ova cjelina nije nešto kompleksna, već je njena glavna uloga da preusmjeri igrača na nivo.

Nivo se može smatrati glavnom složenom jedinicom u ovom projektu. Razlog ovome je što posjeduje najveći broj ostalih složenih jedinica te sam vrši neki vrstu interakcije praktički s cijelim sustavom. Prvotno u njemu se nalazi glavni UI elementi projekta. Tu se nalazi UI nivoa koji govori o trenutnom potezu te na kojem potezu pojedini nivo završava, UI kontroler koji prikazuje dostupne kontrole igraču kad je njegov dio poteza te UI za krajnji prozor koji se samo prikaže kada igrač završi nivo, uspješno ili neuspješno.

Unutar nivoa se također nalaze arena na kojoj se nivo odvija. U ovom kontekstu u arenu spada 3D model nivoa koji uključuje teren kao i nevidljiva polja što je bilo već spomenuto. Uz to tu se još nalazi osvjetljenje pojedinog nivoa, kao i zona smrti. Zona smrti u ovom slučaju predstavlja veliki nevidljivi kvadrat koji se nalazi ispod mape. Kolizija između tog kvadrata te bilo kojeg lika u igri će rezultirati smrću tog lika.

Kamera je jedinica koja omogućuje da se scene, u ovom slučaju trodimenzionalne, prikažu na igračevu ekranu. Za razliku od ostalih jedinica, ona je nešto jednostavnije implementacije, te se u njoj uglavnom nalaze kontrole za njeno pomicanje i rotiranje.

Zadnja logička jedinica koja se može podijeliti bila bi vezana za same likove. Ova logička jedinica se zatim može podijeliti na stvaratelje i kontrolere. Stvaratelji su zaduženi za kreiranja likova. U ovom slučaju igrači se kreiraju samo na "nultom" potezu, dok se protivnički likovi kreiraju na "nultom", ali i na kraju svakog ostalog poteza. Kontroleri se isto dijele na kontroler igrača i kontroler protivnika. Slične su funkcije te svaki od tih kontrolera ima ovlasti nad istoimenim likovima. Kontroler protivnika služi kako bi protivnički likovi imali neku logiku i da odigraju svoj dio potez, dok se u kontroleru igrača također nalazi logika za odrađivanje poteza, ali ta logika se zove igračevim signalima.

5.3. Pregled pojedinih logičkih jedinica

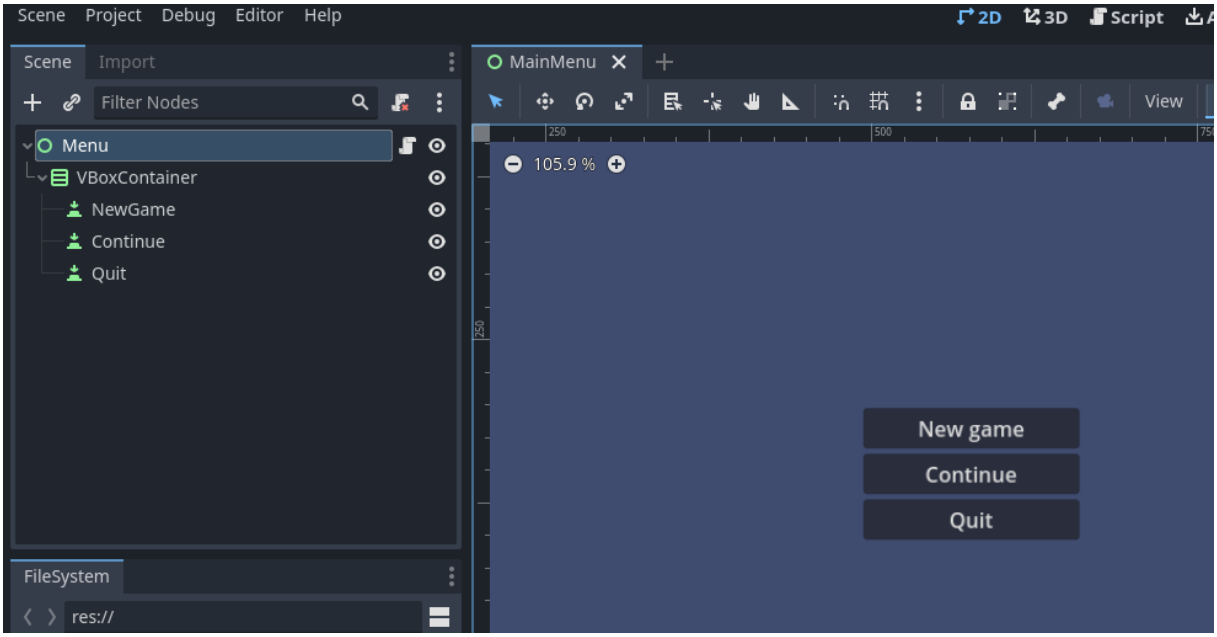
U prethodnom poglavlju bio je prikazan nekakav opći pregled samog projekta, a u ovom djelu će se proći kroz sve bitne logičke jedinice, počevši od jedinice najviše razine.

5.3.1. Izbornik (engl. *Main menu*)

Izbornik je početna scena koja se prikaže igraču prilikom ulaska u samu igru te je vidljiva na slici 16. Scena je u potpunosti sastavljena od kontrolnih, odnosno sučeljnih elemenata i jedna od jednostavnijih u cijelom projektu. Igrač u ovom dijelu može kliknuti na tri gumba koja mu nude tri opcije, a to su: započeti novu igru, nastaviti prijašnju igru te zatvoriti igru.

Funkcionalnosti su dosta jednostavne. Jedina stvar koju je potrebno napomenuti je da nastavak prijašnje igre nastavlja na nultom potezu nekog nivoa do kojeg se ranije došlo. Pošto su gumbi ovdje prvi put spomenuti valja napomenuti da oni funkcioniraju na principu signala.

Signali u ovom kontekstu su zapravo Godotova implementacija *Observer* uzorka dizajna. Kao podsjetnik *Observer* uzorak dizajna je uzorak u kojem objekt, nazvan *subjekt*, ima kolekciju objekta koji ga prate, nazvani *Observeri*, te subjekt šalje obavijesti tim objektima kad se njegovo određeno interno stanje promijeni [15].



Slika 16 - Glavni izbornik

U ovom dijelu koda se također kreiraju konfiguracijske datoteke ukoliko prije nisu postojale. Jedna datoteka nazvana *defaultLevelConfig.cfg* bilježi osnovne informacije o svakom nivou, a to su koja vrsta protivnika se može stvoriti, koliko je rundi potrebno za dobiti te putanja nivoa koji se treba stvoriti.

Druga konfiguracijska datoteka je nazvana *currentLevelConfig.cfg* i ona bilježi samo broj trenutnog nivoa i ona omogućuje da funkcionalnost nastavka prijašnje igre. Prednost ovih konfiguracijskih datoteka, je ta što daju konfigurabilnost određenim dijelovima igra, a da pritom nije potrebno ponovno rekompajlirati cijeli projekt. Tako je moguće samo promijeniti broj trenutnog nivoa na 4 i nastavak igre će nas voditi na zadnji nivo, ili staviti da je potreban samo jedan potez kako bi se pojedini nivo prešao.

5.3.2.Nivo (engl. *Level*)

Kao što je ranije bilo rečeno nivo glavna scena ovog projekta. Međutim sama logika na kojoj nivo funkcionira nije komplicirana, već nivo delegira većinu zadataka drugima. Ipak, za potpuno razumijevanje nivoa bit će potrebno razumjeti i kako ostale scene funkcioniraju, o kojima će nešto više biti rečeno kasnije.

Slika 17 - Scena osnovnog nivoa.

Na slici 17 je vidljiva scena nivoa. Sva četiri nivoa unutar igre imaju ovaj isti raspored, samo s različitim svojstvima. Kao što je vidljivo, čvorovi i scene koje tvore jedan nivo, odgovaraju logičkim jedinicama koje su nešto ranije bilo spomenute.

Kako bi se razumjela logika ove scene, ali i većina scena koja dolazi, potrebno je znati ponašanje dviju osnovnih metoda koja su implementirane u Godot čvorovima, a to su *Ready* i *Process*. *Ready* je funkcija koju čvor pozove prilikom instanciranja te u objektnom programiranju je nešto najsličnije konstruktoru. S druge strane *Process* je metoda koju aplikacija nastoji zovnuti na svakoj sličici (engl. *frame*) u kojoj je zadani čvor aktivan i njeno prvo zvanje se zove tek nakon funkcije *Ready*. Metoda prima jedan parametar, *delta* i taj parametar ukazuje na vremensku razliku između sadašnje sličice i prošle sličice igre. Zbog načina na koji *Process* funkcionira, u kodu dolazi do paralelizma, na način da dok jedan čvor zove *Process*, neki drugi u isto vrijeme može obavljati svoju funkciju procesa. Primjer u ovom projektu je da dok nivo koristi svoju proces metodu, sve ostali aktivni igračevi i protivnički likovi

koriste svoju *Process* metodu. Ovo ponašanje će detaljnije biti opisano kada se bude govorilo o kontrolerima i njihovim likovima.

Unutar scene nivoa, unutar metode *Ready* dohvaćaju se reference na ostale scene unutar prozora. Zatim se iz konfiguracije dohvate informacije o trenutnom, ali i idućem nivou, ukoliko taj nivo postoji. Za trenutni nivo se uzimaju informacije o tome koja vrsta protivnika se može stvoriti, te koliko poteza traje nivo, dok se idući nivo uzima kako bi znalo što raditi u slučaju kad igrač pređe nivo i ako postoji idući nivo da se on brže učita. Zatim se inicijalno stvore igračevi i protivnički likovi, te se oba kontrolera obavijeste o novonastalim likovima. Detaljnije objašnjenje nad stvaranjem likova će biti objašnjeno kasnije, ali zasada je važno napomenuti da tek u ovom trenutku postoji neki lik u igri.

Unutar *Process* dijela nalaze se metode koje omogućuju kretanje i rotiranje kamere korištenjem tipkovnice, ali i metode koja zapravo upravlja cijelom igrom, *TurnHandler*. Metoda ima sljedeći izgled:

```
public void TurnHandler(double delta)
{
    if (EnemyController.ShouldApplyForce())
    {
        EnemyController.DoForcedMovement(delta);
    }
    else if (PlayerController.ShouldApplyForce())
    {
        PlayerController.DoForcedMovement(delta);
    }
    else if (EnemyController.CanFirstAct())
    {
        EnemyController.FirstAct(delta);
    }
    else if (PlayerController.CanAct())
    {
        PlayerController.Act(delta);
   }
    else if (EnemyController.CanSecondAct())
    {
        EnemyController.SecondAct(delta);
    }
    else
    {
        TurnOverOperation();
```

```
        }
    }
```

Ovaj dio koda predstavlja trodiobu jednog poteza, uz neke dodatke. Ideja ovog pristupa je vidjeti u kojoj fazi se pojedini kontroler nalazi te ukoliko se ušlo u funkcionalnost jednog kontrolera da se slobodno preskoče druge metode do iduće sličice *FirstAct* i *SecondAct* unutar protivničkog kontrolera odgovaraju zapravo prvoj i drugoj protivničkoj fazi što je ranije bilo rečeno. Isto tako metoda *Act* unutar igračevog kontrolera odgovara igračevoj fazi. Jedino ponašanje kontrolera koje dosad nije bilo spomenuto je "Forsirano kretanje", iz metode *DoForcedMovement*.

Forsirano kretanje može nastati kada igrač učini napad s nekim od svojih likova. Razlika između forsiranog i normalnog kretanja je ta što odluka za kretanje nije došla od samog lika, kao što je to slučaj prilikom normalnog kretanja, već je rezultat nekog vanjskog podražaja, u ovom slučaju igračeva napada, koji pojedini lik mora poštovati. Kao što je ranije bilo rečeno, pošto u jednom trenutku može biti više aktivnih procesa, važno je onemogućiti ulazak u neke funkcije, dok su pojedini likovi još u stanju kretanja.

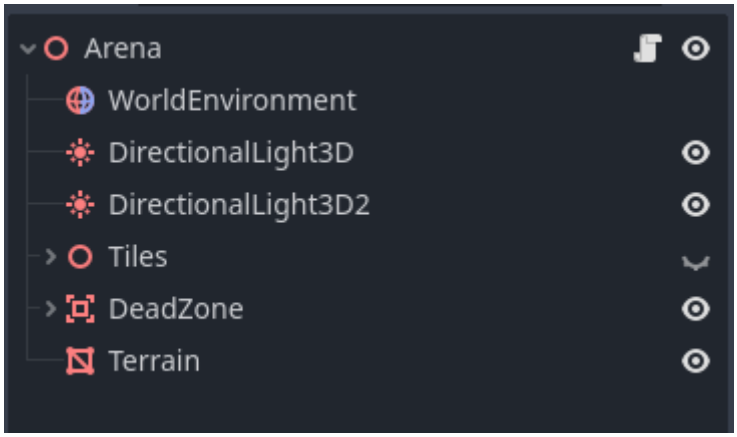
Nakon što su svi kontroleri odradili ono što su mogli odraditi, poziva se *TurnOverOperation* koja odrađuje kraj poteza. Prvo će provjeriti je li nivo izgubljen, a ako nije provjerava je li dobiven. Ukoliko je bilo koja od ovih tvrdnji istinita tada se prikaže UI kontrola *EndInfoWindow* sa odgovarajućim izborima. Sama kontrola je od početka aktivna, ali je učinjena skrivena do ovog trenutka. Ukoliko nivo još nije gotov tada se stvore novi protivnici i poveća broj trenutne runde. Također kontroleri igrača i protivnika se osvježe, kako bi u idućoj sličici ponovno mogli ulaziti u svoje različite faze poteza. Kod za kraj poteza bez popratnih privatnih metoda je idući:

```
var isGameLost = CheckIfGameIsLost();
if (isGameLost)
{
    GameLost();
}
if (currentRound == LevelInfo.RoundsToWin)
{
    LevelWonOperation();
}
currentRound++;
valueCurrentRound.Text = currentRound.ToString();
var enemies = EnemyController.SpawnEnemies();
PlayerController.NotifyAboutNewEnemies(enemies);
```

```
PlayerController.Reset();
EnemyController.Reset();
```

5.3.3. Arena

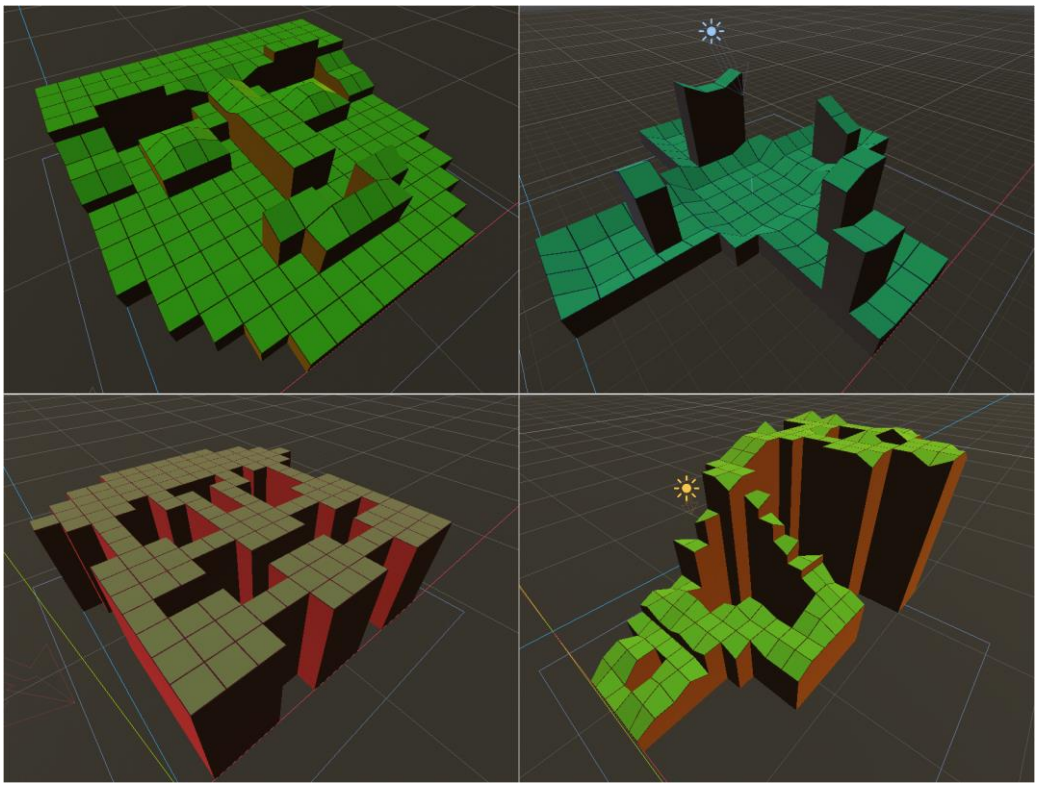
Scena arena predstavlja mapu na kojoj se nivoi odvijaju. Sve scene arene se sastoje od elemenata kao na slici 18. *Terrain* predstavlja sami teren, odnosno 3D objekt mape kroz jedan objekt, dok *Tiles* predstavlja kolekciju gdje je svako polje svoj zaseban objekt. Kao što je vidljivo kolekcija polja originalno ne bude vidljiva, već se ona prikazuje samo kad je potrebna, dok je teren ono što će igrač vidjeti većinu vremena. *DeadZone* predstavlja zonu smrti koja se nalazi ispod mape, ali o noj je već bilo rečeno nešto ranije.



Slika 18 - Scena arene

WorldEnviroment predstavlja pozadinu na kojoj se igra odvija, dok oba *DirectionalLighta* predstavljaju osvjetljenje. Razlog zašto se koriste dva, a ne primjerice jedno osvjetljenje, je zato što s jednim osvjetljenjem određeni dijelovi mape budu u potpunosti tamni, jer igra ne može detektirati nikakvu svjetlost u tom području. Čak i s dva osvjetljenja pojedini dijelovi mape će ostati potpuno tamni, ali tako je napravljeno zbog kreativne vizije.

Dodatna mogućnost prilikom rada s osvjetljenjem je mogućnost dobivanja različitih efekata i samog doživljaja nivoa, samo mijenjajući intenzitet i boju osvjetljenja. Tako se na slici 19 mogu vidjeti sve četiri mape u ovom projektu. Iako se razlikuju po paleti boje, materijali koji se koriste za objekte su identični te je jedina razlika u boji i intenzitetu svjetlosti.



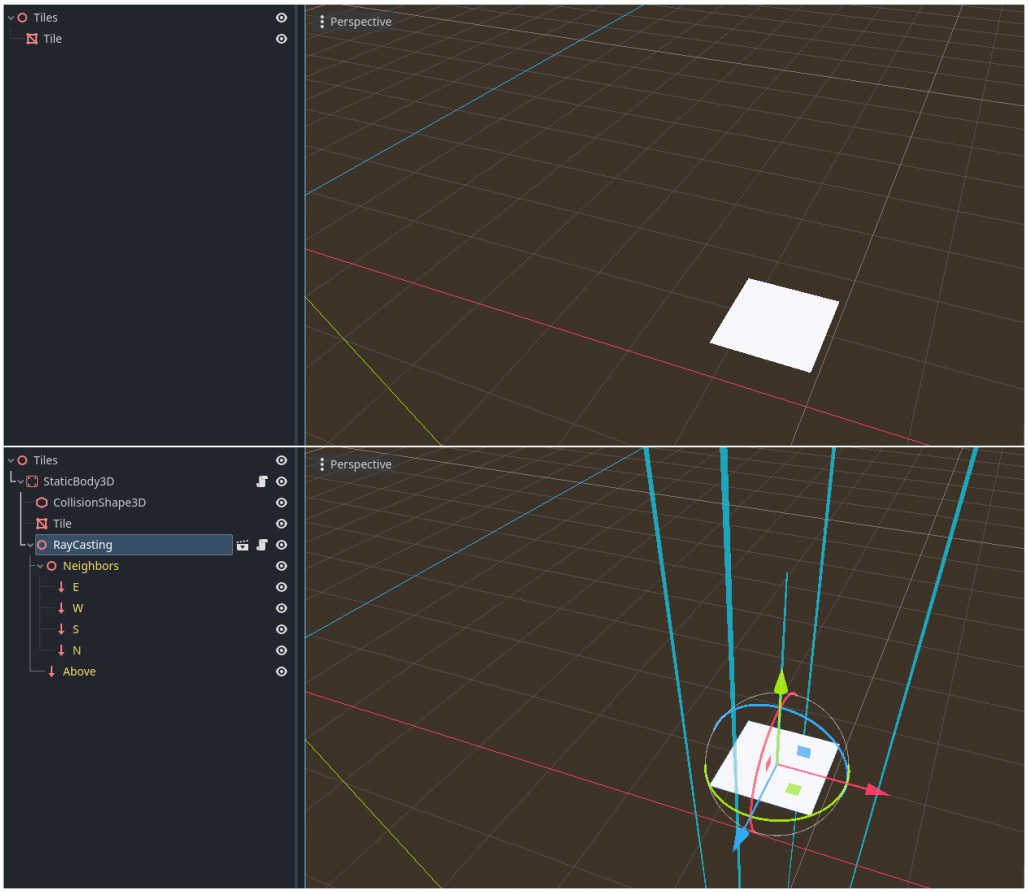
Slika 19 - Sve četiri mape na kojima se odvijaju nivou.

Ključna stvar koja se dogodi prilikom instanciranja objekta arena, je transformacija koja se vrši nad svim nevidljivim poljima. Naime trenutno nevidljiva polja su običan *MeshInstance3D* objekt nad kojim ne postoji nikakva skripta. Ta kolekcija objekta se ovom transformacijom pretvara u *StaticBody3D* koji nad sobom posjeduju skriptu *Tile.cs*. Kod koji čini ovu transformaciju je sljedeći:

```
public static void ConvertTilesIntoStaticBodies(Node3D tilesObj)
{
    var script = ResourceLoader.Load<RefCounted>(TileSrc);
    var tilesObjects = tilesObj.GetChildren().As<MeshInstance3D>();
    foreach (MeshInstance3D tileObj in tilesObjects)
    {
        tileObj.CreateTrimeshCollision();
        var staticBody = tileObj.GetChild(0) as StaticBody3D;
        staticBody.Position = tileObj.Position;
        tileObj.Position = Vector3.Zero;
        tileObj.Name = "Tile";
        tileObj.RemoveChild(staticBody);
        tilesObj.RemoveChild(tileObj);
        staticBody.AddChild(tileObj);
```

```
            var instanceID = staticBody.GetInstanceId();
            staticBody.SetScript(script);
            var staticBodyTile = GodotObject.InstanceFromId(instanceID) as
Tile;
             staticBodyTile.ConfigureTile();
             staticBodyTile.SetProcess(true);

            tilesObj.AddChild(staticBodyTile);
        }
    }
```

Parametar metode je čvor u kojem se nalaze nevidljiva polja. Zatim se za svaki član parametra prolazi kroz petlju gdje se nad svakim elementom radi proces transformacije. *StaticBody3D* se može dobiti preko *MeshInstance3D* koristeći već ugrađene Godot formule. Zatim s metodom *RemoveChild*, koja ne briše elemente, se od spoje postojeće veze, te se u konačnici zapravo zamijene. ovo zapravo znači da je u početku petlje *MeshInstance3D* imao objekt *StaticBody3D*, ali u konačnici stvari su se zamijenile.



Slika 20 - Stanje scene prije i poslije transformacije polja.

Prilikom postavljanja skripte na novo kreiranom objektu i pristupanju tom objektu kao C# klasi njega je potrebno eksplicitno instancirati, što je vidljivo na objektu *staticBodyTile*. Razlog ovome je što C# nije zapravo skriptni jezik i s toga kompajler ne bi mogao prepoznati što treba raditi. Nakon što je objekt instanciran kao *Tile*, na njemu se pokrene konfiguracijska metoda *ConfigureTile*, postavi se proces, te se ovaj novi objekt doda u mjesto gdje su originalno bila nevidljiva polja.

ConfigureTile je isto ključna metoda jer se unutar nje na samom polju dodaje nova scena. Ta nova scena zapravo predstavlja pet zraka svjetlosti (engl. *RayCast*). Najosnovnije rečeno zrake svjetlosti prolaze kroz trodimenzionalni prostor i u slučaju da udare u nekakvu površinu, o tome mogu obavijestiti svog pozivatelja. U ovom slučaju tih pet zraka svjetlosti služi za moguće promatranje susjednih polja na četiri različite strane svijeta te zadnja koja gleda koji objekt se nalazi nad tim poljem. Izgled scene prije i poslije ove cjelokupne transformacije je vidljiv na slici 20.

Razlog zašto se ove mnogobrojne transformacije rade u kodu, a ne unutar samog uređivača scene je jednostavno pitanje praktičnosti. Kroz ovaj pristup izuzetno je jednostavno kreirati nove mape i zatim ih učitati. Sve što je potrebno je kopirati objekt terena, a zatim kopirati sva nevidljiva polja u jedan čvor. Obrnuto bi svako polje ručno trebalo modificirati, a pošto u radu za svaki nivo postoji minimalno sto, a ponekad i preko dvjesto polja, taj proces bi bio izuzetno nepraktičan.

5.3.4.Kontroler protivnika (engl. *Enemy Controller*)

Kontroler protivnika predstavlja generički čvor unutar Godota, koji nad sobom ima skriptu da kontrolira protivničke likove. Kao što je bilo viđeno unutar poglavlja za nivoe, tako ovaj kontroler ima tri provjere, koje su je li forsirano kretanje, je li prva faza i je li druga faza. Ovisno o tome je li zadovoljava jedan od ta tri uvjeta u pravodobnim situacijama, onda će ući unutar logike pojedinog uvjeta. Forsirano kretanje je praktički identično forsiranom kretanju kod kontrolera igrača pa će o tome nešto više biti rečeno kasnije, a sad će više pozornosti biti posvećeno radnjama unutar prve i druge faze.

5.3.4.1. Prva faza protivnika

Prva faza protivnika započinje s upitom je li prva faza moguća. Način na koji se ova provjera dogodi je da se prođe kroz kolekciju protivnika, i ukoliko se ijedan protivnik može kretati tada nastupa prva faza. Ulaskom unutar prve faze ulazi se u metodu *FirstAct*, koja ima sljedeću strukturu:

```
public void FirstAct(double delta)
{
```

```
    switch (Stage)
    {
        case EnemyStage.ChoosePawn:
            ChoosePawn();
            break;
        case EnemyStage.ChoseNearestEnemy:
            ChoseEnemy();
           break;
       case EnemyStage.MovePawn:
           MovePawn();
           break;
        default:
            ChoosePawn();
            break;
    }
}
```

Ključna stvar kod protivničkog kontrolera prilikom obje faze je interno stanje *Stage*. To je varijabla, tipa *EnemyStage* (tipa *Enum*) koja se može naći u sljedećih sedam stanja: *ChoosePawn*, *ChoseNearestEnemy*, *MovePawn*, *ChosePawnToAttack* i *AttackPawn*. Tada ovisno o stanju te varijable ulazi se u različite procese logike samih protivnika. Na sličan način funkcionira i igračev kontroler, ali o njemu će se pričati kasnije.

Ciklus prve faze protivnika je praktički poredan unutar *switch* uvjeta prikazanog maloprije. Početno stanje *Stage* varijable je uvijek *ChoosePawn*. U ovom stanju kontroler traži prvog lika koji se nalazi unutar prvog akta, postavlja ga kao trenutnog lika te promjeni stanje *Stage* varijable u *ChoseNearestEnemy*.

```
public void ChoseEnemy()
{
    Arena.Reset();
    Tile currentPawnTile = CurrentPawn.GetTile();
    Arena.LinkTiles(
        currentPawnTile,
        CurrentPawn.JumpHeight,
        EnemyPawns);
    Arena.MarkReachableTiles(currentPawnTile, CurrentPawn.MoveRadius);
    var distance = Math.Clamp(
        CurrentPawn.AttackRadius - 1,
        1,
        CurrentPawn.AttackRadius);
```

```
        Tile to = Arena.GetNearestNeighborTileToPawn(
            distance,
            CurrentPawn,
            PlayerPawns);
        CurrentPawn.PathStack = Arena.GeneratePathStack(to);
        TacticsCamera.Target = to;
       Stage = EnemyStage.MovePawn;
    }
```

Sa stanjem *ChoseNearestEnemy* protivnik ulazi u metodu *ChoseEnemy*, koja je prikazana u kodu iznad. S *Arena.Reset()* sva svojstva nevidljivih polja se restartiraju. Svojstva koja pojedina polja posjeduje su tri zastavice a to su, je li korijen, je li moguće doći do, je li moguće napasti te jedan cjeloviti broj, udaljenost. Prema tome da se ove vrijednosti ne restartiraju definitivno bi se došlo do krivog izračuna. U idućem koraku se dohvaća polje trenutnog lika te se prema tom polju povezuje mapa.

Osnovna ideja povezivanja mape je da se prvo uzme jedno polje, kojeg se postavi kao trenutno, te se zatim uzmu njegovi susjedi po visini. Ukoliko je na susjede moguće doći, tada se za susjedno polje kao korijen uzima trenutno polje i postavlja se udaljenost od jedan. Kad se prođe kroz sve ove susjede, tada se susjedno polje uzima kao trenutna polje te se proces ponavlja.

Prilikom ponavljanja procesa ignoriraju se polja koja su već bila, a udaljenost će uvijek biti za jedan viša od prethodnog kruga. Na ovaj način se izračun jako brzo proširi i uključi sve polja na koje je moguće doći. Kod za ovaj algoritam je sljedeći:

```
    public void LinkTiles<[MustBeVariant] T>(
        Tile root,
        float height,
        Godot.Collections.Array<T> allies = null) where T : APawn
    {
        var tiles = new Godot.Collections.Array<Tile> { root };
        while (tiles.Count > 0)
        {
            Tile currentTile = tiles[0];
            tiles.RemoveAt(0);
            var neighbors = currentTile.GetNeighbors(height).Values;
            foreach (Tile neighbor in neighbors)
            {
                var neighborRootState = neighbor.Root is null
```

```
                    && neighbor != root;
                var pawnsOccupy = neighbor.IsTaken();
                var shouldLinkTiles = neighborRootState
                    &&!pawnsOccupy;

                if (shouldLinkTiles)
                {
                    neighbor.Root = currentTile;
                    neighbor.Distance = currentTile.Distance + 1;
                    tiles.Add(neighbor);
                }
            }
        }
    }
```

Nakon što su sva polja povezana, sva polja dostupna protivniku se oboje transparentnom-žutom bojom. Ovaj proces je poprilično jednostavan. Potrebno je proći kroz sve nevidljiva polja i ukoliko je njihova novo zadana udaljenost manja ili jednaka udaljenosti koju lik može prijeći i ako polje nije zauzeto, bit će obojano.

Nakon ovog koraka određuje se udaljenost na koju će protivnik pokušati doći do igračevog lika. Kod protivnika koje koriste "hladna" (engl. *melee*) oružja ovo će uvijek biti jedan, ali kod protivnika koje imaju nešto veći domet ovo će biti njihov domet manje jedan. Ovo je učinjeno iz razloga balansiranja, jer ponekad kad bi se stvorilo više kostur-strijelaca, samo bi jedan kostur-strijelac uspio doći u poziciju da napadne protivnika.

Korak nakon je određivanje polja do kojeg će protivnik pokušati doći. Logika algoritma je da prođe kroz igračeve likove te ukoliko je taktika protivnika grubijan onda se uzimaju svi igračevi likovi, ali ukoliko je taktika protivnika snajper ciljeva (engl. *objective sniper*) onda se samo totemi uzimaju u obzir. Tada se za svakog lika koji je uzet u obzir gledaju polja koja su udaljene za udaljenost koja je originalno poslana. Najbliže polje za tog igračevog lika tada postaje prvo polje s najmanjom udaljenosti.

Ovo najmanje polje se zatim usporedi s prethodno najmanje udaljenim poljem, i ukoliko je manje od prethodnog novo polje postaje najmanje. U rijetkim slučajevima u ovom prvom krugu ne će biti dostupno niti jedno polje jer su primjerice svi igrači okruženi kosturima/provalijama. Nakon toga se ovaj algoritam od ranije rekurzivno pokuša ponovo, ali ovaj put za udaljenost za jedan većom od prethodne. Iako se rekurzija nikad ne bi trebala ponoviti previše, ograničena je na tri puta maksimalno (uključujući prvi put). Ovaj algoritam kroz kod je izveden na sljedeći način:

```
    private const int maxIterationOfGetNearestNeighborTileToPawn = 3;
    public Tile GetNearestNeighborTileToPawn(
        int distance,
        APawn pawn,
        Godot.Collections.Array<PlayerPawn> pawns,
        int currentIteration = 1)
    {
        Tile nearestTile = null;
        if (pawn.PawnStrategy == PawnStrategy.ObjectiveSniper)
        {
            pawns = GetOnlyTotems(pawns);
        }
        foreach (PlayerPawn _pawn in pawns)
        {
            var currentPawnTile = _pawn.GetTile();
            var tiles = currentPawnTile.GetNeighbors(pawn.JumpHeight);
            nearestTile = SpecificDistanceAwayTiles(distance, nearestTile,
tiles, pawn);
        }

        while (nearestTile is object
            && !nearestTile.Reachable)
        {
            nearestTile = nearestTile.Root;
        }

        if (nearestTile is object)
        {
            return nearestTile;
        }

        if (nearestTile is null
            && ++currentIteration <=
maxIterationOfGetNearestNeighborTileToPawn)
        {
            return GetNearestNeighborTileToPawn(
                ++distance,
                pawn,
                pawns,
                currentIteration: currentIteration);
        }
```

```
        return pawn.GetTile();
    }
```

Nakon što je određeno polje do kojeg se protivnik želi pomaknuti, generira se put, to jest „*PathStack“* do tog polja. Generacija puta se može gledati kao gradnja stoga. Na nultu, to jest početnu poziciju, se stavi lokacija polja destinacije. Zatim se na nulto mjesto doda korijen ovog polja, a staro polje bude pomaknuta za jedno mjesto unaprijed. Ovaj proces se ponavlja sve dok postoji korijensko polje na prethodno. Ovaj algoritam je implementiran kroz iduću metodu:

```
public Godot.Collections.Array<Vector3> GeneratePathStack(Tile to)
{
    var pathStack = new Godot.Collections.Array<Vector3>();
    while (to is object)
    {
        pathStack.Insert(0, to.GlobalTransform.Origin);
        to = to.Root;
    }
    return pathStack;
}
```

ChoosePawnToAttack metoda funkcionira pomalo slično funkcionalnosti za odabiru mete. Prvo se sva nevidljiva polja restartiraju kako bi se osigurala da neke prethodne kalkulacije ne utječu na nove. Zatim se ponovno sve nevidljiva polja povežu, ali ovaj put za napad. Nakon toga se prikažu polja koje trenutni protivnik može napasti na način da budu obojana transparentnom-crvenom bojom. Poslije ovoga protivnik nasumično bira kojeg će igračeva lika napasti prateći svoju strategiju. Ovo proces možda izgleda kao neefikasan s obzirom da je ranije u algoritmu odlučeno pomaknuti se na određeno mjesto da napadne ili bude bliži svojoj meti, no ovu odluku sam donio jer mi se igra tako činila zabavnijom. I dalje će u većini slučajeva kao meta biti odabrano ono što je u ranijim koracima već odabrao, ali će opet donijeti i trenutke kada primjerice kostur-strijelac pokuša napasti neku metu koja mu je udaljeno samo jedno ili dva polja.

Jedanput kada je meta odabrana, unutar trenutnog lika se ne zapisuju neka svojstva tom liku kojeg napada, već se puni vrijednost *AttackingTowards* s *KeyValuePair*, gdje je ključ udaljenost koju napada, a vrijednost je strana svijeta koja je napadnuta. Ovakav princip je dosta fleksibilan i omogućuje laku pripremu za drugu protivničku fazu. Naime nakon ove metode trenutni lik će morati čekati svoju drugu fazu, a u međuvremenu je moguće da će je

igrač pomaknuti. Ukoliko je igrač pomakne i dalje je potrebno da ovaj protivnički lik učini svoj napad, ako je originalno namjeravao napasti, ali je potrebno da napadne polje relativno s obzirom gdje je originalno bio. Također u slučaju da originalno nije mogla napasti nešto korisno, dovoljno je staviti ključ s vrijednošću nula. Kasnijom provjerom će se slučajevi s ključem nula ignorirati. Na samom kraju trenutni *Stage* se postavlja na *AttackPawn* i ponovno se nevidljiva polja restartiraju kako ne bi ostale obojene crvenom bojom nakon poteza protivnika. Metoda za ovu logiku kao i za dohvata vrijednosti *AttackingTowards* je iduća:

```
    public void ChoosePawnToAttack()
    {
        Arena.Reset();
        Godot.Collections.Array<EnemyPawn> emptyArray = null;
        var currentTile = CurrentPawn.GetTile();
        Arena.LinkTilesForAttack(currentTile, CurrentPawn.AttackRadius,
             emptyArray);
        Arena.MarkAttackableTiles(currentTile, CurrentPawn.AttackRadius);
        AttackablePawn = Arena.GetRandomPawnToAttack(
            PlayerPawns,
            CurrentPawn.PawnStrategy);
        CurrentPawn.AttackingTowards = GetDirectionToWhichShouldAttack();
        if (AttackablePawn != null)
        {
            TacticsCamera.Target = AttackablePawn;
        }
        Stage = EnemyStage.AttackPawn;
        Arena.Reset();
    }

    private KeyValuePair<int, WorldSide> GetDirectionToWhichShouldAttack()
    {
        if (AttackablePawn is null)
        {
            return new KeyValuePair<int, WorldSide>(0, WorldSide.North);
        }
        Vector3 attackDirectionRounded = AttackablePawn.Position.Rounded()
            -CurrentPawn.Position.Rounded();
        int distance = 0;
        var worldSide =
CurrentPawn.GetSideOfWorldBasedOnVector(attackDirectionRounded);
        if (worldSide.EqualsAnyOf(WorldSide.North, WorldSide.South))
        {
```

```
            distance = (int)Math.Round(Math.Abs(AttackablePawn.Position.Z -
CurrentPawn.Position.Z));
        }
        else
        {
            distance = (int)Math.Round(Math.Abs(AttackablePawn.Position.X -
CurrentPawn.Position.X));
        }
        var attackableTile = AttackablePawn.GetTile();
        CurrentPawn.LookAtDirection(attackableTile.GlobalTransform.Origin -
CurrentPawn.GetTile().GlobalTransform.Origin);
        var attackingTowards = new KeyValuePair<int, WorldSide>(distance,
worldSide);
        return attackingTowards;
    }
```

Prethodno nabrojani koraci bili su za jednog lika. Kroz prvu fazu svi likovi prođu kroz ovaj proces, i pošto se unutar metode *MovePawn*, zastavica koja dozvoljava kretanje se postavi na laž, više neće biti nijedan lik kojoj je dozvoljena prva faza i kao takva prva faza protivnika završava. Na samom kraju *Stage* od protivničkog kontrolera će biti unutar stanja *AttackPawn*.

5.3.4.2. Druga faza protivnika

Druga faza protivnika je dosta jednostavnija od prve faze, te se praktički sastoji od samo dvije funkcionalnosti. Druga faza protivnika započinje nakon što je igrač gotov sa svojim potezom. Ova faza traje sve dok postoji bar jedan protivnik čija zastavica da može napasti, to jest *CanAttack* je istinita. Logika druge faze je zapisana kroz idući kod:

```
    public void SecondAct(double delta)
    {
        ChoosePawnThenPrepareAttack();
        if (Stage == EnemyStage.AttackPawn)
        {
            Attack();
        }
    }
```

Metoda *ChosePawnThenPrepareAttack* samo izabere trenutnu jedincu, uzimajući u obzir zastavu *CanAttack* te postavlja trenutni *Stage* na *AttackPawn*. Ključna metoda druge faze je *Attack*. Sama metoda je također jednostavna te se može podijeliti na dva dijela. U

prvom dijelu se dohvaća polje uzimajući u obzir varijablu *AttackingTowards* koja je bila postavljana u prvoj fazi.

Polje se pronađe na način da se krene od polja na kojem lik trenutno stoji, i onda se dohvaćaju susjedna polja prema strani svijeta, sve dok se ne dođe do udaljenosti gdje je lik početno htio napasti. Nakon toga lik odrađuje svoju akciju nad tim poljem. Kako sve te akcije funkcioniraju će biti objašnjeno unutar cjeline za likove, dok je zasada bitno da se na kraju te metode tom liku zastava *CanAttack* postavi na laž. Algoritam *CanAttack*, kao i njegova popratna metoda za pretragu polja, je vidljiv u idućem bloku koda:

```
    public void Attack()
    {
        if (CurrentPawn.AttackingTowards.Key == 0)
        {
            CurrentPawn.CanAttack = false;
            return;
        }
        var distance = CurrentPawn.AttackingTowards.Key;
        var side = CurrentPawn.AttackingTowards.Value;
        var attackingTile = CurrentPawn.GetTile();
        attackingTile = GetTileBasedOnDistanceAndSide(distance, side,
attackingTile);
        if (attackingTile is object)
        {
            CurrentPawn.DoCharacterActionOnTile(AllActiveUnits,
attackingTile);
        }
    }

    private Tile GetTileBasedOnDistanceAndSide(
        int distance,
        WorldSide worldSide,
        Tile startingTile)
    {
        for (int i = 0; i < distance; i++)
        {
            startingTile = startingTile.GetNeighborAtWorldSide(worldSide);
            if (startingTile is null)
            {
                return null;
            }
```

```
        }
        return startingTile;
    }
```

Kada svi protivnici prođu kroz ovaj algoritam, svima će zastava *CanAttack* biti laž. S ovim završava druga faza protivnika, nakon koje slijedi funkcionalnost za određivanja kraja poteza, koja je bila spomenuta ranije.

5.3.5.Kontroler igrača (engl. *Player Controller*)

Kontroler igrača je zadužen za upravljanjem igračevim likovima. Po tome je dosta sličan kontroleru protivnika, ali glavna razlika njih dvoje je u tome što logika za rad s igračevim likovima ne dolazi automatski, već se generira vanjskim, to jest igračevim podražajima.

Igračeva faza poteza dolazi nakon prve protivničke faze, te traje sve dok se bar jedan igračev lik može kretati ili napadati, osim totema. Totemi iako se vode kao igračevi likovi su ignorirani za proces logike. Tako igrač ne može jednostavno izabrati totema i napast s njime ili ga pak pomaknuti. Još jedna bitna stavka totema je da su u potpunosti nepomični, što znači da ih ni igračevi napadi ne mogu pomaknuti.

Slično kao i kod protivničke metode *Act*, unutar kontrolera igrača postoji glavna metoda istog naziva, ali nešto drukčije implementacije. Kod za igračevu metodu *Act* je sljedeći:

```
    public void Act(double delta)
    {
        if (CurrentPawn is null)
        {
            Stage = PlayerStage.SelectPawn;
        }
        var visibilityBasedOnStage = VisibilityBasedOnStage();
        UIControl.SetVisibilityOfActionsMenu(
            visibilityBasedOnStage,
            CurrentPawn);
        switch (Stage)
        {
            case PlayerStage.SelectPawn:
                SelectPawn();
                break;
            case PlayerStage.DisplayAvailableActionsForPawn:
                DisplayAvailableActionsForPawn();
                break;
```

```
            case PlayerStage.DisplayAvailableMovements:
                DisplayAvailableMovements();
                break;
            case PlayerStage.SelectNewLocation:
                SelectNewLocation();
                break;
            case PlayerStage.MovePawn:
                MovePawn();
                break;
            case PlayerStage.DisplayAttackableTargets:
                DisplayAttackableTargets();
                break;
            case PlayerStage.SelectTileToAttack:
                SelectTileToAttack();
                break;
            case PlayerStage.AttackTile:
                AttackTile(delta);
                break;
        }
    }
```

Slično kao i kod protivničkog kontrolera, metoda ovisi o *Stage* globalnoj varijabli. Ovo je i dalje *Enum* varijabla, samo što je ona tipa *PlayerStage*, a ne *EnemyStage*. U sebi ima osam stanja, a to su: *SelectPawn*, *DisplayAvailableActionsForPawn*, *DisplayAvailableMovements*, *SelectNewLocation*, *MovePawn*, *DisplayAttackableTargets*, *SelectTileToAttack* i *AttackTile*. Ova stanja također imaju dodijeljene cjelovite brojčane vrijednosti u sebi, od nula do sedam, poredano ovim redoslijedom kojim su napisane. Ovi cjeloviti brojevi se koriste za dio logike kontrolera pa su zato navedeni.

Act metoda se početno može raščlaniti na dva dijela. Prvi dio je zadužen za prikazivanje igračevog sučelja, dok se drugi dio dotiče samih funkcionalnosti unutar *switch* naredbe. Pod prikazivanje igračevog sučelja se misli na prikaz gumbi u donjem desnom dijelom ekrana, koji omogućuju igraču da obavi svoj potez. Prvotno ako još nijedan lik nije odabran, to jest ako se kontroler nalazi u stanju *SelectPawn*, taj prozor ne će biti prikazan jer bi bio besmislen.

Također taj prozor uopće ne će biti prikazan ukoliko se kontroler nalazi unutar stanja *AttackTile*. Razlog ovome je više estetski, jer prilikom igračeva napada često dolazi do toga da likovi budu forsirano pomaknuti pa se sa sakrivanjem ovog dijela sučelja to lakše vidi. Ukoliko igračevo sučelje nije u potpunosti skriveno, tada se još proslijedi i trenutna jedinica, te

ovisno o tome može li napadati i može li se kretati, se gumbi za to onemoguće. Izgled igračevog sučelja je vidljiv na slici 21.

Slika 21 - Igračevo sučelje. Obratiti pozornost na donji desni kut.

Ulaskom u *switch* naredbu sa stanjem *SelectPawn* ulazi se u istoimenu metodu. Za razliku od protivničkog kontrolera gdje bi trenutni lik bio odabran automatski, kod igrača lik se bira klikom miša. U osnovi kod cijelo vrijeme promatra nad čim igrač lebdi s mišem. Ono polje nad kojim igrač lebdi će biti specijalno označeno. Ukoliko igrač klikne na polje, a ne na samog lika, igra će mu i dalje uspješno vratiti tog lika kao da ga je ručno odabrao.

U trenutku kad igrač pošalje signal prihvati (engl. *ui_accept*), igra će provjeriti je li to uistinu igračev lik, a ne protivnikov te je li odabrani lik posjeduje dozvoljene radnje (kretnja ili napad). Jedna stvar koja se ovdje sad prvi put pojavljuje je da metoda lebdenja s mišem, može uzimati elemente s različitog sloja. U ovom projektu likovi se nalaze na drugom sloju, dok su nevidljiva polja na prvom. Logika dohvaćanja trenutnog lika, kao i pripadajuća logika lebdenja s mišem je vidljiva u idućem kodu:

```
public void SelectPawn()
{
    CurrentPawn = AuxSelectPawn();
    if (CurrentPawn is null)
    {
        return;
```

```
        }
        if (Input.IsActionJustPressed("ui_accept")
            && CurrentPawn.CanAct()
            && PlayerPawns.Contains(CurrentPawn))
        {
            TacticsCamera.Target = CurrentPawn;
            Stage = PlayerStage.DisplayAvailableActionsForPawn;
        }
    }

    private PlayerPawn AuxSelectPawn()
    {
        PlayerPawn pawn = GetMouseOverObject(2) as PlayerPawn;
        Tile tile = pawn is null
            ? GetMouseOverObject(1) as Tile
            : pawn.GetTile();
        Arena.MarkHoverTile(tile);
        if(tile.GetObjectAbove() is PlayerPawn playerPawn)
        {
            return playerPawn;
        }
        if (pawn is object)
        {
            return pawn;
        }
        else
        {
            return null;
        }
    }
```

Sa odabirom jedinice, ulazi se u drugi dio kontrolera, a to je *DisplayAvailableActionsForPawn*. Ovo stanje se može smatrati kao neko neutralno stanje, te je njegova funkcija pokazati igraču koji lik je trenutno odabran, te da se sva svojstva nevidljivih polja restartiraju. Ovo je učinjeno zato što igrač ne mora učiniti sve dostupne akcije s jednim likom prije nego li prijeđe na idućeg, za razliku od protivničkih likova.

Naprimjer igraču je moguće prvo pomaknuti se s jednim likom, zatim odabrati drugog pa naposljetku odabrati trećeg i s njime učiniti samo napad. Nakon svih ovih akcije, svi likovi

će i dalje imati neke dostupne akcije, a upravo ovaj dio koda omogućuje da jedan lik nema utjecaja na drugog. Kod za ovaj dio je jednostavan i on poprima sljedeći oblik:

```
public void DisplayAvailableActionsForPawn()
{
    Arena.Reset();
    var currentPawnTile = CurrentPawn.GetTile();
    Arena.MarkHoverTile(currentPawnTile);
}
```

Preostalih 6 funkcionalnosti unutar *switch* naredbe se mogu podijeliti u dvije kategorije, a to su kategorija za kretanje i kategorija za napadanje. Klikom na gumb *Move*, kontrolerova varijabla *Stage* poprima vrijednost *SelectNewLocation* te igrač ulazi u istoimenu metodu. Ovo se smatra prvim korakom kategorije za kretanje.

Sama metoda je po svojoj logici dosta slična protivničkom dijelu kontrolera *ChooseEnemy*. Prvo se restartiraju svojstva svih polja, zatim se od trenutnog polja lika povežu sva ostala polja. Polja koja se nalaze u dosegu igrača su zatim obojana žutom bojom, a stanje varijable *Stage* se prebacuje u *SelectNewLocation*. Logika ovog dijela kontrolera je vidljiva u idućem isječku koda:

```
public void DisplayAvailableMovements()
{
    Arena.Reset();
    TacticsCamera.Target = CurrentPawn;
    var currentTile = CurrentPawn.GetTile();
    Arena.LinkTiles(currentTile, CurrentPawn.JumpHeight, PlayerPawns);
    Arena.MarkReachableTiles(currentTile, CurrentPawn.MoveRadius);
    Stage = PlayerStage.SelectNewLocation;
}
```

Sa dolaskom u stanje *SelectNewLocation*, ulazi se u drugi korak kontrolerove kategorija za kretnju. Slično kao i prilikom odabira samog lika, ovaj dio koda kontinuirano prati poziciju igračeva miša i dohvaća polje nad kojim igrač lebdi.

Za razliku od trenutka kad se bira lik, ovaj put se uzima u obzir samo prvi sloj igre, tamo gdje se polja nalaze. Ukoliko igrač klikne s mišem na neko polje do kojeg se igrač može pomaknuti tada se generira putanja do tog polja, a kontroler ulazi unutar stanja *MovePawn*. Logika ovog dijela kontrolera je vidljiva u idućem isječku koda:

```
public void SelectNewLocation()
```

```
{
    Tile tile = GetMouseOverObject(1) as Tile;
    Arena.MarkHoverTile(tile);
    if (Input.IsActionJustPressed("ui_accept")
        && tile is object
        && tile.Reachable)
    {
        CurrentPawn.PathStack = Arena.GeneratePathStack(tile);
        TacticsCamera.Target = tile;
        Stage = PlayerStage.MovePawn;
    }
}
```

Zadnji dio kategorije kretanja unutar kontrolera se ispunjava ulaskom *Stage* u stanje. Samo kretanje se obavlja unutar procesa lika, a unutar kontrolera se samo čeka kada će sveukupni put trenutnog lika isprazniti. S završetkom putanje, trenutnom liku se automatski onemogućuje daljnje kretanje.

Ukoliko lik nakon ovoga ima i mogućnost napada, lik ostaje odabran, a stanje se prebacuje unutar *DisplayAvailableActionsForPawn* gdje će gumb za kretanje biti onemogućen. Ukoliko je lik već bio izvršio napad, tada završavaju sva njegova moguća stanja te se stanje kontrolera šalje nazad unutar *ChoosePawn*. Isječak koda za posljednji dio kretanja unutar kontrolera se nalazi u idućoj metodi:

```
public void MovePawn()
{
    if (CurrentPawn.PathStack.Count == 0){
        CurrentPawn.CanMove = false;
        SetCurrentPawnState();
    }
}
```

Druga kategorija dostupna igraču je napadanje. Igrač može napasti s pojedinim likom, ako već ranije u potezu nije napao s tim likom, to jest ako mu je zastavica *CanAttack* i dalje istinita. Prvi dio za kategoriju napada nastupa kada igrač klikne na gumb *Attack*. Klikom na taj gumb kontroler igrača postavlja *Stage* na *DisplayAttackableTargets* te se ulazi u istoimenu metodu. Unutar te metode se dohvaća polje nad kojim igrač stoji, te se spajaju polja u cilju napada. Polja koja su igraču dostupna oboja ju se u crveno, a kontroler ulazi u stanje *SelectTileToAttack*.

Unutar tog stanja radi se praktički ista operacija kao i prilikom kretanja lika, samo što se odabrano polje bilježi pod varijablom *AttackableTile*. Biranjem polja za napasti ulazi se u zadnju kategoriju napada, a to je stanje *AttackTile*. Unutar ovog stanja zove se ista metoda koju i protivnici koriste za napad. To je metoda koja se nalazi unutar logike samog lika, a za kontroler je najbitnije da na samom kraju ta metoda postavi zastavicu *CanAttack* na laž. Nakon ovoga dijela kao i prilikom završetka kretanje, *Stage* kontrolera se postavlja na odgovarajuću vrijednost, ovisno je li igraču dostupna još ijedna akcija, odnosno je li se igrač već prethodno bio kretao s ovim likom. Cjelokupni isječak koda za igračevo napadanje je sljedeći:

```
public void DisplayAttackableTargets()
{
    Arena.Reset();
    TacticsCamera.Target = CurrentPawn;
    Godot.Collections.Array<PlayerPawn> emptyArray = null;
    var currentTile = CurrentPawn.GetTile();
    Arena.LinkTilesForAttack(
        currentTile,
        CurrentPawn.AttackRadius,
        emptyArray);
    Arena.MarkAttackableTiles(currentTile, CurrentPawn.AttackRadius);
    Stage = PlayerStage.SelectTileToAttack;
}

public void SelectTileToAttack()
{
    Tile tile = AuxSelectTile();
    if (tile is object)
    {
        AttackableTile = tile;
    }
    else
    {
        AttackableTile = null;
    }

    if (Input.IsActionJustPressed("ui_accept")
        && tile is object
        && tile.Attackable)
    {
        TacticsCamera.Target = AttackableTile;
```

```
            Stage = PlayerStage.AttackTile;
            Arena.Reset();
        }
    }
    public void AttackTile()
    {
        CurrentPawn.DoCharacterActionOnTile(
            AllActiveUnits,
            AttackableTile);
        TacticsCamera.Target = CurrentPawn;
        SetCurrentPawnState();
    }
```

5.3.6. Forsirano kretanje likova unutar kontrolera

Forsirano kretanje likova je stanje koje potencijalno nastaje kada igrač za vrijeme svoje faze učini napad. Obe vrste likove, odnosno i igračevi i protivnički mogu ući u ovo stanje, a način na koji pojedini lik može biti gurnut je već ranije bilo opisano u radu. Pošto oba kontrolera imaju isti način na koji su im likovi gurnuti, dovoljno je objasniti guranje samo kod jednog kontrolera.

Kao što oba kontrolera za odvijanje svoje faze imaju *switch* naredbu, čije grananje ovisi o *Enum* varijabli *Stage*, tako i forsirano kretanje ima svoju *Enum* varijablu *ForceCalculation*. Varijabla se sastoji od tri stanja, a to su: *ForceFree*, *ForceBeingCalculated* i *ForceBeingApplied*. Osnovno stanje, kada nad likom nije potrebno izvršiti nikakvu silu je *ForceFree*.

Ako se ponovno pogleda *TurnHandler* unutar nivoa vidljivo je da provjera forsiranog kretanja dolazi prije provjere bilo koje faze igrača ili protivnika. Prvo se provjera vrši za protivnika, a zatim za igrača. Kako bi neki lik ušao u ovo stanje potrebno je da zastavica nad njegovom varijablom *shouldBeForciblyMoved* bude istinita. Zastava se postavlja istinitom tijekom igračeva napada, ukoliko je potrebno. Jedanput postavljena varijabla *forceCalculation* se postavlja na *ForceBeingCalculated*, te kontroler ulazi u metodu *DoForcedMovement*, koja je po svojoj funkciji slična *Act* metodama.

DoForcedMovement se sastoji od jednog uvjeta. Ukoliko je stanje *ForceBeingCalculated*, on će jedanput izračunati potrebnu silu koju treba primijeniti nad likom, dok će inače uvijek ulaziti unutar metode za primjenu same sile, *ApplyForce*.

U metodi za računanje sile, *CalculateForce*, generira se put kojim će lik biti gurnut. Za ovaj dio koriste se i apsolutne lokacije samih objekata, odnosno u ovom slučaju likova, za

razliku od prijašnjih načina gdje su se koristile samo polja i zrake svjetlosti. Razlog ovome je zato što lik može upasti u provaliju, odnosno izvan same mape. U početku ove metode se izračuna apsolutna pozicija gdje bi se pojedini lik trebao gurnuti. Ukoliko se ispod te lokacije ne nalazi polje, riječ je o poziciji izvan mape, te se onda ta lokacija uzima kao put samog lika. Ukoliko postoji neko polje, onda se put generira do tog polja slično kao i kod normalnog kretanja. Na samom kraju ove metode *forceCalculation* se postavlja na *ForceBeingApplied*.

Ukoliko je stanje *ForceBeingApplied*, kontroler će cijelo vrijeme ulaziti unutar *ApplyForce* metode. Slično kao i kod kretanja, sami kontroler ne će pomicati lika, već će se lik kretati unutar svog procesa, o čemu će nešto više biti rečeno kasnije. *ApplyForce* zapravo cijelo vrijeme samo provjerava je li putanja forsiranog gibanja gotova. Kad putanja završi restartira se stanje trenutnog lika, odnosno zastavica *shouldBeForciblyMoved* se postavlja na laž, a *forceCalculation* ponovno prelazi u stanje *ForceFree*. Još jedna važna stvar za napomenuti je da ukoliko je *forceCalculation* u bilo kojem trenutku u stanju *ForceBeingApplied*, kontroler ne će vršiti neke druge provjere, već će automatski ući unutar metode *DoForcedMovement*. Isječak koda za funkcionalnost guranja je:

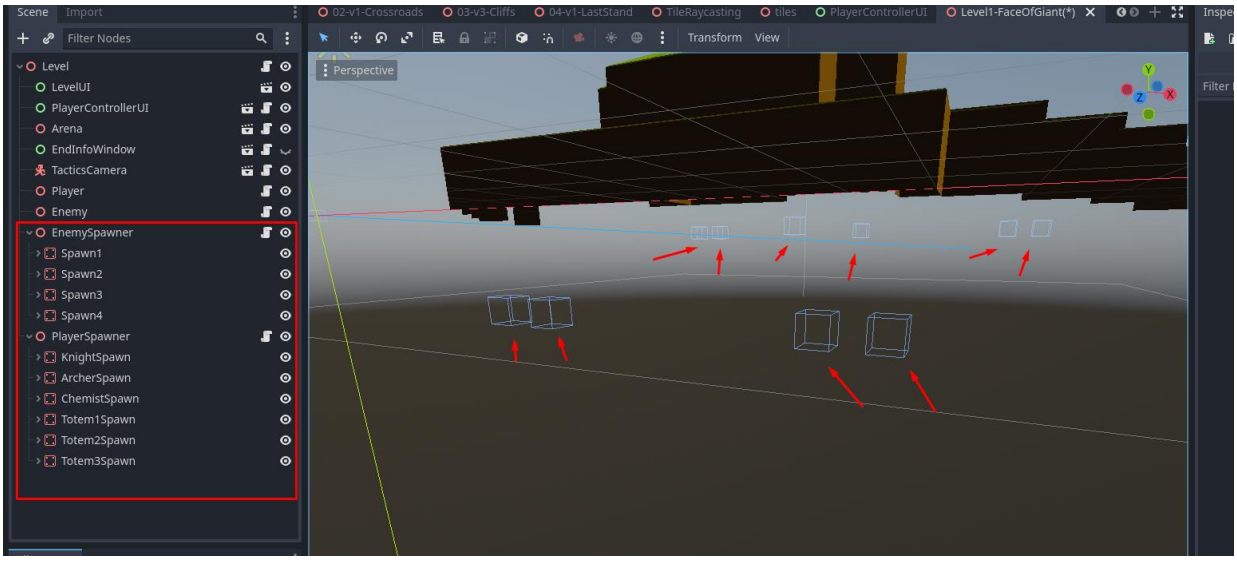
```
public bool ShouldApplyForce()
{
    if (_forceCalculation == ForceCalculation.ForceBeingApplied
        && CurrentPawn is object)
    {
        ApplyForce();
        return true;
    }
    foreach (var pawn in EnemyPawns)
    {
        if (pawn.shouldBeForciblyMoved)
        {
            _forceCalculation = ForceCalculation.ForceBeingCalculated;
            CurrentPawn = pawn;
            return true;
        }
    }
    return false;
}

public void DoForcedMovement(double delta)
{
    if (_forceCalculation == ForceCalculation.ForceBeingCalculated)
```

```
    {
        CalculateForce();
        ApplyForce();
    }
    else
    {
        ApplyForce();
    }
}

private void CalculateForce()
{
    var location = (CurrentPawn.GlobalPosition
        + CurrentPawn.directionOfForcedMovement).Rounded();
    Tile to = Arena.GetTileAtLocation(location);
    if (to is null)
    {
        CurrentPawn.PathStack.Add(location);
    }
    else
    {
        var currentPawnTile = CurrentPawn.GetTile();
        CurrentPawn.PathStack = Arena.GenerateSimplePathStack(
            currentPawnTile,
            to);
        TacticsCamera.Target = to;
    }
    CurrentPawn.MoveDirection = Vector3.Zero;
    _forceCalculation = ForceCalculation.ForceBeingApplied;
}

private void ApplyForce()
{
    if (CurrentPawn.PathStack.Count == 0)
    {
        _forceCalculation = ForceCalculation.ForceFree;
        CurrentPawn.shouldBeForciblyMoved = false;
    }
}
```

5.3.7.Stvaratelj likova (engl. *spawner*)

Kao što je bilo spomenuto ranije, likovi nisu standardni dio neke scene, već se oni po potrebi instanciraju unutar svojih kontrolera. Likovi se jedanput stvore za igrača, prije početka prve runde, dok se za protivnike stvaraju tada, ali i po završetku svakog poteza. Razlog za inicijalno stvaranje svih likova je pretežito bio zbog buga koji bi mi nastao UI elementima likova, ukoliko bi jedinicu stvorio odmah s nivoom. Proces kreiranja likova igrača i protivnika je iznimno sličan, stoga će se detaljno opisati stvaratelj protivnika, dok će se za stvaratelja igračevih likova samo spomenuti ključne razlike.

Protivnička radnja stvaranja likova prije prvog poteza kao i nakon svakog poteza je jednaka. Jedan od ključnih elemenata stvaranja likova su elementi samog stvaratelja na nivou. Glavni element je sam po sebi najobičniji čvor, ali on u sebi ima različite čvorove *StaticBody3D*. Ti čvorovi se nalaze ispod mape te se ovisno o njihovoj širini i dužini stvaraju likove. Kod igračevog čvora za stvaranja također je bitan i redoslijed čvorova, jer ovisno o redoslijedu će se stvoriti različiti likovi. Razlog zašto se ovi čvorovi nalaze ispod mape je radi jednostavnijeg rukovanja s kolizijom, a i visinom. Naime ukoliko se točke stvaranje postave tamo gdje bi se likovi točno trebali stvoriti, tada lika nije moguće stvoriti jer se tu već nalazi neko tijelo. Postojala su još neka druga rješenja za ovaj problem, kao što je primjerice stavljanje čvorova u različite slojeve, ali sam radi praktičnosti ovo riješio tako da mi se čvorovi jednostavno se nalaze ispod nivoa. Na slici 22 je moguće vidjeti lokaciju čvorova za stvaranja unutar prvog nivoa.



Slika 22 - Čvorovi za stvaranje unutar prvog nivoa

Prilikom stvaranja likova, bit će stvoreno onoliko likova koliko ima čvorova unutar samog stvaratelja. Glavna metoda za stvaranje likova se može podijeliti u dva dijela, a to je dio koji stvara samu instancu lika te dio koji računa gdje se taj lik mora nalaziti.

Samo instanciranje lika je jednostavan proces u kojem se instancira *PackedScene* kao objekt protivnika te mu se zatim popune osnovna svojstva kao što su ime, strategija i klasa. Svojstva imena i strategije svoju nasumičnost uzimaju iz konstanta koje se nalaze unutar klase, dok se svojstvo klase jedinice uzima iz konfiguracije trenutnog nivoa.

Lokacija mjesta gdje će lik biti stvoren dobije se pomoću zrake svjetlosti. Sama zraka započinje na lokaciji koja je po dužini i širini ista kao točka, ali je zato njena visina jako puno iznad nivoa, dok se njezin cilj nalazi točno unutar točke stvaranje. Udarom zrake svjetlosti u neki objekt se dobije potpuno precizna lokacija na koju treba stvoriti lika. Ovaj pristup osim što rješava prethodni problem kolizije, također omogućuje da bez obzira koliko različitih visina ima unutar nivoa, da je uvijek moguće liku precizno generirati visinu na kojoj bi se trebao stvoriti.

Unutar najniže metode za stvaranje likova, likovi zapravo ne će biti stvoreni, a razlog tome je zato što još uvijek ne će pripadati nikakvom čvoru unutar samog nivoa. Čvoru ne će pripadati zbog toga što oba kontrolera moraju znati za novonastale likove. Pošto likovi još uvijek nisu stvoreni, nije im moguće odmah zadati svojstvo lokacije pa se zbog ovog u najnižoj metodi vraća rječnik gdje je ključ sami lik, a vrijednost lokacija lika. Izgled koda koji stvara protivničkog lika je idući:

```
public Dictionary<EnemyPawn, Vector3> SpawnEnemies()
{
    var enemyPawns = new Dictionary<EnemyPawn, Vector3>();
    var allowedEnemies = LevelManager.GetCurrentLevelInformation()
        .AllowedEnemies;
    for (int i = 0; i < Points.Count; i++)
    {
        var enemyPawn = GetEnemyNode(allowedEnemies) as EnemyPawn;
        var pointTranslation = new Godot.Vector3
        (
            Points[i].GlobalPosition.X,
            Points[i].GlobalPosition.Y,
            Points[i].GlobalPosition.Z
        );
        var spaceState = GetWorld3D().DirectSpaceState;
        var query = PhysicsRayQueryParameters3D.Create(
            pointTranslation + Vector3.Up * 100, pointTranslation);
        var result = spaceState.IntersectRay(query);
```

```
            var tilePosY = result["position"].AsVector3().Y;
            pointTranslation.Y = tilePosY;

            enemyPawn.Visible = true;
            enemyPawns.Add(enemyPawn, pointTranslation);
        }
        return enemyPawns;
    }

    private Node GetEnemyNode(
        Godot.Collections.Array<PawnClass> allowedEnemies)
    {
        var node = EnemyScene.Instantiate();
        var enemyPawn = node as EnemyPawn;
        enemyPawn.PawnClass = allowedEnemies.GetRandom();
        enemyPawn.PawnStrategy = possibleStrategies.GetRandom();
        enemyPawn.PawnName = possibleNames.GetRandom();
        return node;
    }
```

Razlika ovog dijela i igračevog stvaranja je praktički samo unutar generiranja čvor objekta koji je kod igračevog kontrolera *PlayerPawn*. Njegova svojstva nisu nasumična, već se ovisno o indeksu unutar petlje generira lik s određenim imenom i klasom.

Jedanput kad je rječnik dohvaćen unutar kontrolera, likovi se stvarno instanciraju unutar scene te se likovima proslijedi lokacija na kojoj treba biti. Ovi likovi se zatim umetnu u odgovarajuće kolekcije samog kontrolera. Također sa svakim stvaranjem likova unutar jednog kontrolera, drugi se kontroler obavijesti, kako bi i on mogao pravilno spremiti novonastale likove unutar svog kontrolera.

Osim spremanje u kolekcija ono što oba kontrolera rade jest spajanje na sami objekt kao promatrač. Naime na svakom liku je ručno kreiran uzorak dizajna *Observer*, gdje su likovi subjekti, dok su oba kontrolera promatrači. To je kreirano tako da se prilikom brisanje lika, do kojeg dolazi prilikom njegove smrti, ne bi došlo do *null* referenci unutar kontrolerovih kolekcija. Način na koji protivnički kontroler instancira likove na scenu, te način na koji igračev kontroler počne promatrati iste je prikazan u idućem isječku koda.

```
//unutar protivničkog kontrolera
public Godot.Collections.Array<EnemyPawn> SpawnEnemies()
{
```

```
    var actualPawns = new Godot.Collections.Array<EnemyPawn>();
    var pawns = EnemySpawner.SpawnEnemies();
    foreach (var pawn in pawns)
    {
        AddChild(pawn.Key);
        pawn.Key.GlobalPosition = pawn.Value;
        pawn.Key.Attach(this);
        actualPawns.Add(pawn.Key);
    }
    EnemyPawns.AddRangeAs(actualPawns);
    AllActiveUnits.AddRangeAs(actualPawns);
    return actualPawns;
}

//unutar igračevog kontrolera
public void NotifyAboutNewEnemies(Array<EnemyPawn> enemies)
{
    foreach (var enemyPawn in enemies)
    {
        enemyPawn.Attach(this);
        AllActiveUnits.Add(enemyPawn);
    }
}
```

5.3.8. Likovi

Likovi su uz kontroler najkompleksniji dio cjelokupne igre. Likovi igrača su *PlayerPawn*, dok su likovi protivnika *EnemyPawn*, ali oboje nasljeđuju apstraktnu klasu *APawn*. Upravo se unutar te apstraktne klase nalazi većina logike, pošto obje vrste likove dijele većinu svoje logike. U sljedećim pod-poglavljima ne će se prolaziti kroz sve dijelove samih likove, već će se obraditi samo najbitniji dijelovi koji su već bili spomenuti, a to kretanje i napadanje likova.

5.3.8.1. Kretanja likova

Dosad je kretanje likova bilo spomenuto dva puta. Jedanput kad se svojevoljno želi pomaknuti na određenu lokaciju, a jedanput kad je forsiran pomaknuti se na neko polje. Uz ove dvije vrste kretanja, tehnički postoji i treća, a to je malo mikro kretanja, kojemu je cilj da se lik centrira nad svojim poljem.

Samo kretanje se nalazi unutar procesa lika, što znači da će za svaki novo renderiranu sličicu, igra pokušati pomaknuti lika. Ukoliko je na liku generiran nekakav put, tada lik ulazi u normalno kretanje, ili u forsirano ako mu je zastavica *shouldBeForciblyMoved* postavljena na istinu. Ukoliko je put lika prazan, tada se zove samo centriranje lika, koje u većini slučajeva ne će morati ništa napraviti. Ova kratka osnovna petlja je zapisana u sljedećem obliku:

```
public void ApplyMovement(double delta)
{
    if (PathStack.Count != 0)
    {
        if (shouldBeForciblyMoved == true)
            ForciblyMovePawn(delta);
        else
            FollowThePath(delta);
    }
    else
    {
        AdjustToCenter();
    }
}
```

MoveDirection je varijabla lika, oblika *Vector3*, koja ukazuje prema gdje se lik kreće. U stanje kad se lik ne miče, to jest kad se samo centrira to je nul vektor, odnosno vektor čije su sve osi nula. Ulaskom u metodu *FollowThePath(delta)* ta varijabla kao svoju vrijednost uzima razliku između prvog člana kolekcije puta, odnosno *pathStacka* i svoje lokacije te s time počinje logika svojevoljnog kretanja.

Nakon što je generirana vrijednost *MoveDirection*, lik se okreće prema toj točki. Pošto je lik tipa *CharacterBody3D* unutar Godota na njemu postoji ugrađena implementacija metode za kretanje *MoveAndSlide*, koja pokreće samo tijelo. Ova metoda će se koristiti za pokretanje samog lika, a ovisi o argumentu *Velocity* koji je isto ugrađen unutar objekata tipa *CharacterBody3D*. Taj objektni atribut se puni s umnoškom dvije varijable, a to su *velocity* i *speed*.

Varijabla *speed* će uvijek imati konstantnu vrijednosti, osim ako se radi o skoku. U slučaju skoka prolazi kroz malu formulu gdje mu brzina ovisi o tome koliko visoko treba skočiti, ali najčešće ova brzina bude nešto manja od obične.

Varijablu *velocity* početno se napuni s normaliziranim vektorom *MoveDirectio*n. Varijabla se zatim promijeni samo ukoliko bi objekt trebao doći, to jest pasti na neku nižu

razinu. U tom slučaju se interno kreira varijable gravitacije koja se zbroji s originalnom varijablom *velocity*.

Ovakav proces za kretanje za jedan član kolekcije puta će se ponavljati sve dok tijelo lika kojeg se pomiče ne bude u određenom radijusu lokacije kojoj želi doći. Kad je lik dovoljno blizu lokacije, briše se ta varijabla kolekcije puta te se neke druge varijable postave na zadane vrijednosti. Ako nakon brisanja sami put nije prazan, ovaj cijeli proces će se ponoviti, sve dok put nije prazan. Kad put postane prazan kretanje lika završava. Kod za prethodni algoritam je sljedeći:

```
public void FollowThePath(double delta)
{
    if (!CanMove)
    {
        return;
    }
    if (MoveDirection == Vector3.Zero)
    {
        MoveDirection = PathStack.FirstOrDefault()
            - GlobalTransform.Origin;
    }
    if (MoveDirection.Length() > 0.5)
    {
        LookAtDirection(MoveDirection);
        Vector3 velocity = MoveDirection.Normalized();
        float currentSpeed = Speed;
        if (MoveDirection.Y > MinHeightToJump)
        {
            currentSpeed = Godot.Mathf.Clamp(
                Math.Abs(MoveDirection.Y) * 2.3f, 3f, Godot.Mathf.Inf);
            IsJumping = true;
        }
        else if (MoveDirection.Y < -MinHeightToJump)
        {
            Gravity += Vector3.Down * (float)delta * GravityStrength;
            var distanceWithoutY = Utils.VectorDistanceWithoutY(
                PathStack.FirstOrDefault(),
                GlobalTransform.Origin);
            if(distanceWithoutY <= 0.2)
            {
                velocity += Gravity;
```

```
            }
            else
            {
                velocity = Utils.VectorRemoveY(
                    MoveDirection).Normalized() + Gravity;
            }
        }
        this.Velocity = velocity * currentSpeed;
        MoveAndSlide();
        var distanceToPoint = GlobalTransform.Origin.DistanceTo(
            PathStack.FirstOrDefault());
        if ( distanceToPoint >= 0.2)
        {
            return;
        }
    }
    if (PathStack.Count > 0)
    {
        PathStack.RemoveAt(0);
    }
    MoveDirection = Vector3.Zero;
    IsJumping = false;
    Gravity = Vector3.Zero;
}
```

Forsirano kretanje je zapravo samo jedan manji podskup normalnog kretanja. Ovo kretanje nema neku svoju logiku, već je sva logika građena samo od dijelova normalnog kretanja. Dijelovi koji u forsiranom kretanju fale naspram normalnog kretanje je okretanje prema smjeru gdje je lik gurnut kao i logika za računanja dijelova oko skoka. Razlog za okretanje je da se ne dobije dojam da je lik svojevoljno pomaknut u nekom smjeru, dok razlog za skok je jednostavno da lika nije moguće gurnuti u stanje skoka, jer bi se već zabio u zid.

Centriranje lika je jednostavan proces u kojem se lik postavlja na centar polja na kojem stoji. Za početak se uzima središnja točka polja na kojem lik stoji i nad njim se oduzima središnja točka samog sebe. Razlici ovog se zatim pridodaje gravitacija kako bi se osiguralo da tijelo stvarno stoji na nekoj plohi. Ova konačna varijabla se zatim množi s određenim cijelim brojevima kako bi jedinica došla do središta. Logika za centriranje kroz kod je iduća:

```
public void AdjustToCenter()
{
```

```
    var tile = GetTile();
    var tilePoint = tile.GlobalTransform.Origin;
    MoveDirection = tilePoint - GlobalTransform.Origin + Gravity;
    this.Velocity = MoveDirection * Speed * 4;
    MoveAndSlide();
}
```

5.3.8.2. Napadi likova

Dosad su napadi likova bili spomenuti kod kontrolera i ono što je u tom trenutku bilo spomenuto je da će lik nakon napada postaviti svoju zastavici *CanAttack* u laž. Metoda *DoCharacterActionOnTile* koja u sebi prima sve aktivne likove kao i polje nad kojim se izvršava akcija je jednaka za sve likove i za oba kontrolera. Po svojoj strukturi nalikuje pomalo na *Act* metode unutar kontrolera, na način da se ovisno o klasi trenutnog lika kreiraju različiti napadi. Glavni kod za akcije pojedinih likova je idući:

```
public void DoCharacterActionOnTile(
    Array<APawn> allActiveUnits,
    Tile attackableTile)
{
    LookAtDirection(attackableTile.GlobalTransform.Origin
        - GlobalTransform.Origin);
    switch (PawnClass)
    {
        case PawnClass.Knight:
        case PawnClass.Archer:
        case PawnClass.Cleric:
            NormalPlayerUnitAttack(allActiveUnits, attackableTile);
            break;
        case PawnClass.Chemist:
            ChemistAttack(allActiveUnits, attackableTile);
            break;
        case PawnClass.SkeletonWarrior:
        case PawnClass.SkeletonArcher:
        case PawnClass.SkeletonHero:
            NormalNPCAttack(allActiveUnits, attackableTile);
            break;
        case PawnClass.SkeletonBomber:
            BomberAttack(allActiveUnits);
            break;
```

```
            case PawnClass.SkeletonMedic:
                MedicAction(allActiveUnits);
                break;
        }
        this.SoundPawnAttack.Play();
        this.CanAttack = false;
    }
```

Iz koda je vidljivo da postoje pet glavnih akcija, a to su normalni napadi igračevih likova, alkemistov napad, normalni napad protivničkih likova, napad kostur-bombaša te liječenje kostur-doktora. Protivnički napadi su jednostavniji od igračevih jer u sebi nemaju elemente guranja.

Prva stvar kod protivničkog algoritma je da provjeri provjeri je li se nalazi u prisustvu kostur-heroja. Ako je tada će šteta koju čini biti uvećana za jedan. Nakon toga traži postoji li lik na zadanom polju i ukoliko postoji radi direktnu štetu nad njim. Iz idućeg isječka koda je vidljiva ova logika:

```
    private void NormalNPCAttack(
        Array<APawn> allActiveUnits,
        Tile attackableTile)
    {
        var additionalDamage = GetSkeletonDmgBuffs();
        var targetPawn = attackableTile.GetObjectAbove() as APawn;
        if (targetPawn is object)
        {
            DealDirectDamageAndRemoveIfDead(
                targetPawn,
                AttackPower + additionalDamage);
        }
    }
```

Kostur-doktor i kostur-bombaš funkcioniraju nešto drugačije jer oni nemaju normalne napade. Kostur-doktor prolazi kroz sve likove, i ukoliko je neka lik protivnik, a ima manje zdravlja nego li je to maksimalno moguće, tada će mu obnoviti jedno zdravlje. S

Kostur-bombaš s druge strane za vrijeme svoje akcije zapravo ubije sam sebe, što aktivira njegovu posmrtnu akciju gdje radi ogromnu štetu svim susjednim poljima. Kostur-bombašev napad ima sljedeći izgled unutar koda:

```
    private void BomberDeathAftermath(APawn bomberPawn)
```

```
    {
        var explosionTile = bomberPawn.GetTile();
        DoIndirectCrossAttack(explosionTile, damage: 999);
    }

    private void DoIndirectCrossAttack(Tile attackableTile, int damage = 0)
    {
        foreach (var worldSide in allWorldSides)
        {
            var worldSideTile = attackableTile.GetNeighborAtWorldSide(
                worldSide);
            if (worldSideTile is null)
            {
                continue;
            }
            if (worldSideTile.GetObjectAbove() is APawn worldSidePawn)
            {
                switch (PawnClass)
                {
                    case PawnClass.SkeletonBomber:
                        DealDirectDamageAndRemoveIfDead(worldSidePawn,
damage);
                        break;
                    case PawnClass.Chemist:
                        DoTheRepeatingCongaLineAttack(worldSidePawn,
worldSide);
                        break;
                }
            }
        }
    }
```

Ono što se i kod kostur-bombaša, ali i normalnih kostura može vidjeti je da svoju štetu rade kroz metodu *DealDirectDamageAndRemoveIfDead*. Ovo je zapravo jedina metoda u kojoj se likovima oduzima zdravlje, a nju također koriste i igračevi likovi. Glavna stvar ove metode je da ukoliko lik ima nula ili manje zdravlja, da obavijesti svoje slušatelje (slušatelji su bili postavljeni tijekom stvaranja likova), a zatim da deinstancira objekt sa scene. Razlog zašto je potrebno obavijestiti svoje slušatelje je da oni na vrijeme mogu maknuti sve reference sa lika koji će ubrzo biti deinstanciran, da izbjegnu referenciranje na prazan objekt. Unutar ovih

referenci pripadaju kolekcije gdje se nalaze svi likovi ili likovi pojedinog kontrolera, kao i varijabla *CurrentPawn* koju posjeduje svaki kontroler.

Igračevi likovi viteza i strijelca imaju skoro pa identičnu logiku za napadanje kao i normalni protivnički likovi, ali prije zvanja metode *DealDirectDamageAndRemoveIfDead*, oni zovu metodu *DoIndirectAttacks*. Unutar te metode dohvaća se strana svijeta na koju se zadana lik gura te se gurnuti lik i strana svijeta prosljeđuje metodi *DoTheRepeatingCongaLineAttack*.

Metode *DoTheRepeatingCongaLineAttack* je glavna metoda koja je namijenjena guranju likova. Svoju logiku započinje da dohvati polje na koje bi trenutni lik trebao biti gurnut, uzimajući u obzir stranu svijeta. Ukoliko navedeno polje ne postoji, to znači da je lik gurnut u provaliju.

Ukoliko je gurnut u provaliju tom liku se postavlja zastavica *shouldBeForciblyMoved* na istinu, a smjer forsiranog kretanja je jednak razlici vektora između pozicije gdje bi se polje nalazilo da postoji i trenutne pozicije samog lika. Ukoliko se lik pokušaje gurati na polje koje postoji, ali je previsoko, lik će primiti štetu guranja, ali metoda će završiti jer ovo znači da je lik udario u zid. Na samom kraju vrši se provjera je li postoji lik na polju na kojeg se trenutni lik gura.

Ukoliko ne postoji tada se trenutnom liku postavljaju parametri forsiranog kretanja, to jest zastavica *shouldBeForciblyMoved* se postavlja na istinu, a vektor guranja je jednak razlici pozicije polja na koju je trenutni lik gurnut i poziciji samog lika. No ako na polju gdje se gura lik postoji neki drugi lik, započeti će rekurzija ove metode.

Rekurzija će trajati tako dugo, dok neki od prethodnih uvjeta nije ispunjen. Po izlasku iz rekurzije, i lik koji je gurao i lik koji je bio gurnut će primiti štetu guranja. Algoritam ove metode primijenjen u kodu je vidljiv u idućem isječku:

```
private void DoTheRepeatingCongaLineAttack(
    APawn targetPawn,
    WorldSide sideWherePawnIsGettingPushed)
{
    var targetPawnTile = targetPawn.GetTile();
    var tileWherePawnIsGettingPushed = PawnTile
        .GetNeighborAtWorldSide(sideWherePawnIsGettingPushed);
    if (tileWherePawnIsGettingPushed is null)
    {
        var positionOfOutOfBounds = targetPawnTile
            .GetNeighborPositionAtWorldSide(
                sideWherePawnIsGettingPushed);
        var forcedFallDirection = (positionOfOutOfBounds
```

```
                - targetPawn.Position).Rounded();
            targetPawn.shouldBeForciblyMoved = true;
            targetPawn.directionOfForcedMovement = forcedFallDirection;
            return;
        }
        if (Math.Abs(tileWherePawnIsGettingPushed.Position.Y -
targetPawnTile.Position.Y) >= WallHeightToGetDamaged)
        {
            DealDirectDamageAndRemoveIfDead(targetPawn, PushDamage);
            return;
        }

        var potentialPawn = tileWherePawnIsGettingPushed.GetObjectAbove()
as APawn;
        if (potentialPawn is null)
        {
            var forcedMovementDirection =(
                  tileWherePawnIsGettingPushed.Position
                  - targetPawnTile.Position).Rounded();
            targetPawn.shouldBeForciblyMoved = true;
            targetPawn.directionOfForcedMovement = forcedMovementDirection;
        }
        else
        {
            DoTheRepeatingCongaLineAttack(
                potentialPawn,
                sideWherePawnIsGettingPushed);
            DealDirectDamageAndRemoveIfDead(potentialPawn, PushDamage);
            DealDirectDamageAndRemoveIfDead(targetPawn, PushDamage);
        }
    }
```

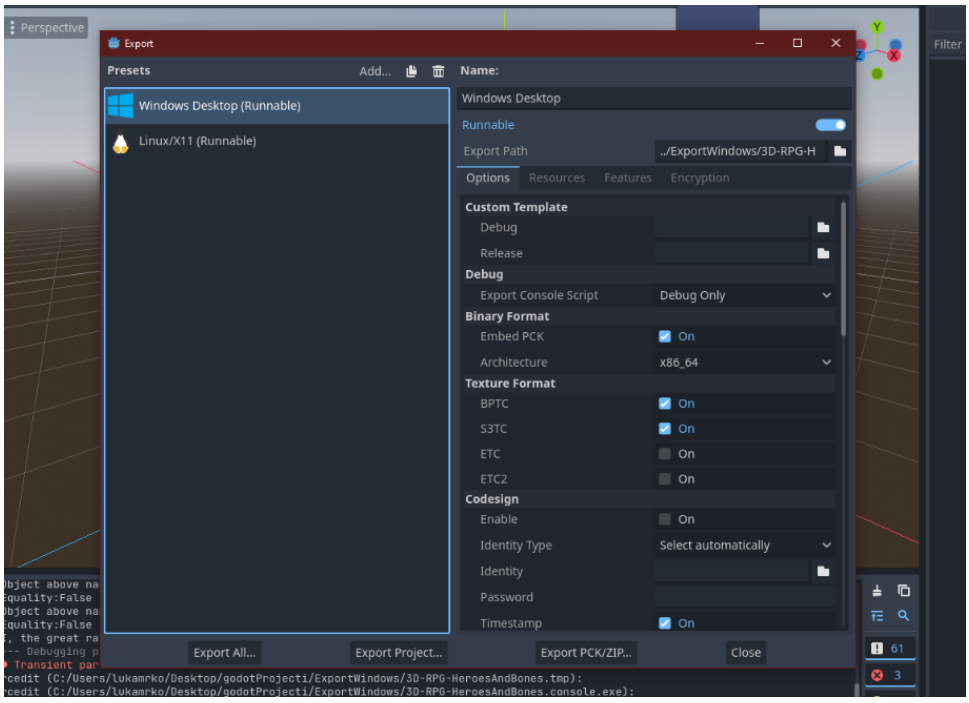
Jedini lik čiji napad još nije opisan je alkemist. Njegov napad se može podijeliti na dva dijela, a to je križasti napad s kojim gurne protivnike, te centralno polje nad kojim napravi direktnu štetu. Njegov križasti napad je isti kao i kod kostura-bombaša, samo što umjesto direktne štete, on na svakom polju zove *DoTheRepeatingCongaLineAttack* metodu da gurne protivnike. Jedanput kad je s petljom prošao kroz sve strane svijeta, tada samo učini direktnu štetu na polju koje je originalno naciljao.

5.4. Izvoz igre (engl. *game export* i prikaz same igre

Jedanput kad je projekt gotov, potrebno je omogućiti drugima da igraju igru. U teoriji je moguće samo kopirati cijeli projekt i reći drugoj strani da instaliraju Godot, ali to nije nimalo praktičan izbor. Optimalni način za omogućiti drugima da igraju ovaj projekt, je da se napravi izvoz same igre. Izvoz igre omogućuje da se igra spremi na disk i zatim pokrene jednim klikom, bez potrebe za nekim dodatnim instalacijama.

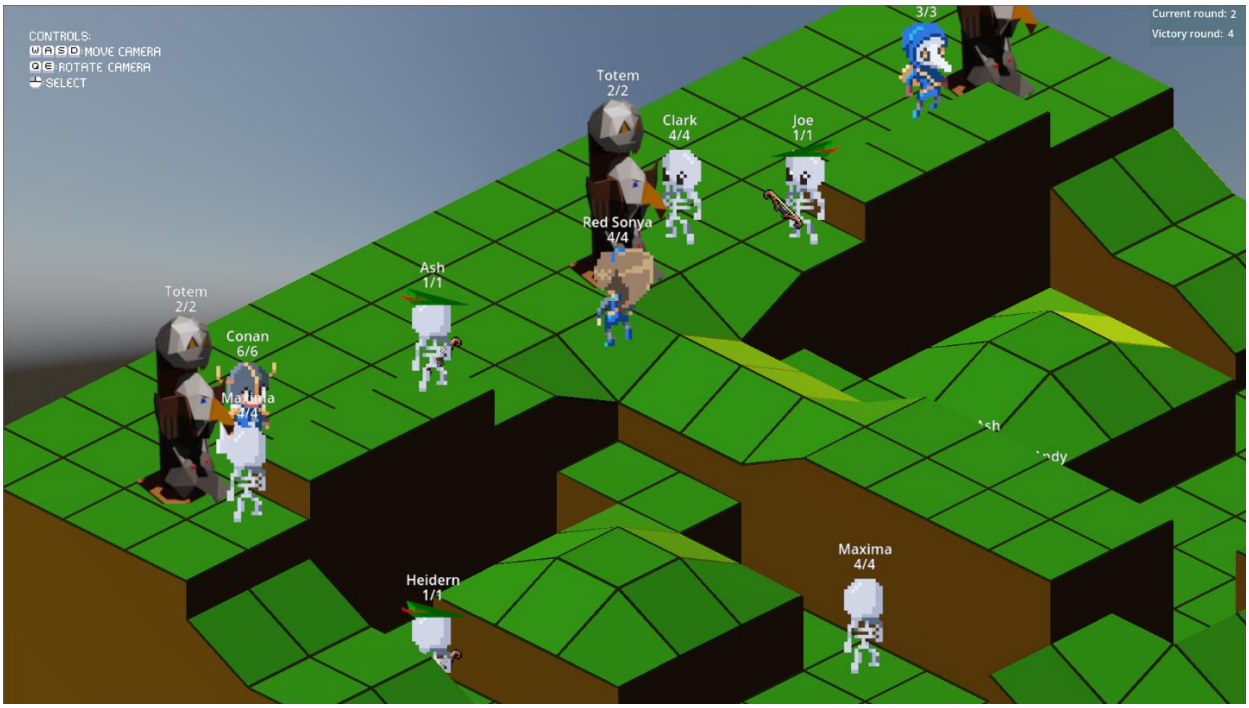
Upravo se s ovim izvozom može ciljati više platforma, što su u ovom slučaju Windows i Linux. Jedina stvar u kodu koju je potrebno napraviti kako bi izvoz bio uspješan za obje platforme, je način na koji se radi s eksternim datotekama. Naime unutar Windowsa putanja između različitih mapa se definira kosom crtom prema lijevo, odnosno s '\\', dok se unutar Linuxa definira s kosom crtom prema desno, odnosno s '/'.

Jedanput kad je rad s eksternim datotekama, što je u ovom projektu rad s konfiguracijskim datotekama, ispravno podešen, onda se izvoz projekta može napraviti praktički samo s jednim klikom. Sve što je potrebno je otići u prozor *Project* → *Export* i onda odabrati platformu za izvoz. Nakon što je to učinjeno otvara se prozor gdje se nude različite platforme za izvoz. Odabirom platforme se ponude još neke opcije, ali za potrebe ovog rada je dovoljno samo odabrati mapu u kojoj se želi obaviti izvoz. Na slici 23 je moguće vidjeti prozor za izvoz projekta.



Slika 23 - Prikaz prozora za izvoz projekta

Jedanput izvezena za igrati igru potrebno je samo otvoriti izvršnu datoteku aplikacije. Ako se pak želi igrati na nekom drugom računalu potrebno je samo kopirati mapu gdje je projekt izvezen. Izvoz je osim jednostavnog načina za igranje također optimizirao veličinu same igre i malo je poboljšao performanse. Na slici 24 se može vidjeti jedan prikaz same igre.



Slika 24 - Prikaz same igre

6. Zaključak

Cilj ovog projekta je bila izrada strateške videoigre na poteze unutar Godota, za platforme Windows i Linux, što smatram da je uspješno izvedeno. Istražene su osnovne funkcionalnosti Godota, što je ponajprije uključivao rad sa scenama, implementacija različitih mehanike igre, rad s različitim resursima kao što su mape, slike i teksture i slično. Sama igra, iako podosta gruba, je u potpunosti funkcionalna i igriva. Sastoji se od četiri nivoa, gdje svaki nivo nudi nove izazove u obliku novih protivnika. Korisnik kroz sučelje vrši interakciju sa svijetom te je u svakom trenutku točno jasno što se događa. Za projekt je prikazana i integracija Godot-a s programskim jezikom C# i korištenje nekih naprednih programskih tehnika kao što su uzorci dizajna, nasljeđivanja i rekurzije.

Kroz rad sam imao svakakvih iskustava. Implementacija bilo kakve mehanike bi dala jako dobar osjećaj, ali kroz cjelokupni razvoj nailazio sam na dosta frustrirajućih bugova. Iako je dio njih nastao mojom krivicom, nezanemariva količina ih je nastalo jednostavno čudnim ponašanjem samog engine. Također iako implementacija s C\# postoji po meni na njoj još treba podosta proraditi. S debugiranjem sam imao dosta problema, a nemogućnost spajanja Visuala Studia za razvoj igre je ogromni minus.

S druge pak strane Godot ima dosta potencijala. Nove značajke se stalno dodavaju, a postojeći bugovi se popravljaju. Za izradu neke jako jednostavne, a pogotovo dvodimenzionalne igre, Godot je vjerojatno jedan od najboljih alata na tržištu. Same igre se dobro pokreću, a njihova veličina je minimalna.

Znao sam da razvoj videoigara nije lagan proces, ali sam tek kroz ovaj projekt shvatio koliko je to zahtjevno. Moje poštovanje kreatorima videoigara je znatno poraslo, a pogotovo samostalnim kreatorima. Kroz projekt sam naučio mnogo toga, a najdraže mi je bio rad s zrakama svjetlosti kao i praktična primjena uzoraka dizajna. Kad bih projekt išao ponovno raditi, princip mapa bi mi vjerojatno ostao isti, ali bih nastojao pojednostaviti logiku kontrolera i jedinica, koje smatram da su u svom trenutnom stanju prekompleksni. Ne mislim da ću nastaviti samostalno razvijati videoigre, te ako se ponovno okušam u nekom projektu to će biti ili unutar nekog tima ili neka jednostavnija 2D igra.

Popis literature

[1] Demos, Ramaureirac, Godot tactical rpg, 2023. adresa:

https://github.com/ramaureirac/godot-tactical-rpg.

[2] J. Gregory, Game Engine Architecture. A K Peters, 2009.

[3] Godot, Godot main page, 2023. adresa: https://godotengine.org/

[4] L. Dawe, „Godot Game Engine Is Now Open Source," GamingOnLinux, 2014. adresa:

https://www.gamingonlinux.com/articles/godot-game-engine-is-now-open-source.3096/.

[5] J. Linietsky, „Godot history in images!" Godot engine blog, 2014. adresa:

https ://godotengine.org/article/godot-history-images/.

[6] J. Linietsky, „Godot Engine reaches 1.0, first stable release," Godot engine blog, 2014.

adresa: https://godotengine.org/article/godot-engine-reaches-1-0/.

[7] J. Linietsky, „Godot Engine reaches 2.0 stable," Godot engine blog, 2016. adresa:

https://godotengine.org/article/godot-engine-reaches-2-0-stable/.

[8] J. Linietsky, „Godot 3.0 is out and ready for the big leagues," Godot engine blog, 2018.

adresa: https://godotengine.org/article/godot-3-0-released/.

[9] J. Linietsky, „Godot 4 released," Godot engine blog, 2023. adresa:

https://gamefromscratch.com/godot-4-released/.

[10] Google, Godot trends, Kreirano 10.5.2023., 2023. adresa:

https://trends.google.com/trends/explore?date=2014-01-01\%202023-05-10&q=godot\%20engine&hl=hr

[11] Google, Godot trends, Kreirano 10.5.2023., 2023. adresa:

https://trends.google.com/trends/explore?date=2014-01-01\%202023-05-10&q=godot\%20engine,unreal\%20engine,unity&hl=hr

[12] Similar web engine trends, Kreirano 11.5.2023., 2023. adresa:

https://pro.similarweb.com/#/digitalsuite/websiteanalysis/traffic-engagement/*/999/3m/?webSource=Total&selectedWidgetTab=Visits&key=unrealengine.com,unity.com,godotengine.org.

[13] Into the Breach - Gameplay screenshots, Kreirano 14.5.2023., 2023. adresa:

https://subsetgames.com/itb.html#screenshot1

[14] Stealthix, Pixel bow pack, Kreirano 14.5.2023., 2023. adresa:

https://stealthix.itch.io/pixel-bow-pack

[15] E. Gamma, R. Helm, R. Johnson i J. M. Vlissides Design Patterns: Elements of Reusable Object-Oriented Software, illustrated edition. addison-wesley, 1994.

Popis slika