Razvoj ponovno iskoristivih komponenti u programskom alatu Godot

Sitarić, Matej

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Matej Sitarić

RAZVOJ PONOVNO ISKORISTIVIH KOMPONENTI U PROGRAMSKOM ALATU GODOT

DIPLOMSKI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Matej Sitarić

Studij: Informacijsko i programsko inženjerstvo

RAZVOJ PONOVNO ISKORISTIVIH KOMPONENTI U PROGRAMSKOM ALATU GODOT

DIPLOMSKI RAD

Mentor/Mentorica:

Doc. dr. sc. Mladen Konecki

Varaždin, studeni 2024.

Matej Sitarić

Izjava o izvornosti

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

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

_____

Sažetak

Ovaj rad se bavi razvojem ponovno iskoristivih komponenti na primjeru alata Godot. Ideja je da ulažući više vremena u dizajn sustava (igre u slučaju ovog rada) se može znatno olakšati budući razvoj i skalabilnost. Također ako se pokreće novi projekt mogu se iskoristiti komponente iz prijašnjih projekata ako su napravljene na modularan način. Naravno nije nužno da se cijela komponenta iskoristi, moguće je i njen „kostur“ iskoristiti, ali to i dalje zahtjeva planiranje. Prikazat će se na koji način se može planirati razvoj i kako primijeniti plan konkretno na primjeru više videoigara. Na samome kraju će se zaključiti koliko se komponenata uspjelo prenijeti iz projekta u projekt i kako su se ponovno iskoristile unutar jednog samog projekta te koje su pozitivne i negativne strane ovakvog pristupa razvoju.

Ključne riječi: Godot, iskoristive komponente, planiranje, videoigre

Sadržaj

1. Uvod

Ovaj rad bude pokušao predstaviti način razvoja koda/sustava kroz planiranje i primjenu napravljenog plana na konkretnim primjerima.

Planovi koji budu prikazani unutar ovog rada su jako fleksibilni i ovisi od osobe do osobe kako i na koji način želi implementirati pojedini sustav i posložiti elemente. Cilj planiranja je unaprijed sastaviti popis svega što bude bilo potrebno za lakši i modularni razvoj. Bez planiranja se može jednostavno dogoditi da se prilikom razvoja novog elementa krene mijenjati ostatak sustava kako bi bolje odgovarao novom elementu i time trošiti vrijeme i resurse na problem koji se mogao unaprije riješiti dobro sastavljenim planom.

Planovi će se razvijati iterativno što znači da će se prvo krenuti od najapstraktnijih stvari i zatim ponovno krenuti od početka sastavljenog plana da se svaki pojedini dio bolje razradi. Ovaj proces se bude ponavljao sve dok ne budem dovoljno zadovoljni priloženim planom nakon čega će biti razvijen kod unutar programskog alata Godot.

Primjeri su razvijeni u programskom alatu Godot primarno jer je jednostavniji za korištenje od Unity-a (za apsolutne početnike), ima zanimljiv sustav scena i nodeova koji će biti objašnjen u kasnijem poglavlju i također ima vlastiti programski jezik koji jako liči na Python programski jezik koji je poznat po svojoj pristupačnosti novim programerima.

Cijeli proces planiranja i izrada videoigre bude napravljen 3 puta kako bi se na različite načine demonstrirao cijeli proces, ali i da se prikaže kako razvoj modularnih sustava može biti vrlo koristan ne samo unutar jednog projekta, nego više njih.

Na kraju će se rezimirati svi zaključci navedenog načina razvoja, koje su pozitivne i negativne strane cjielog procesa te eventualno kako bi se moglo poboljšati sve zajedno.

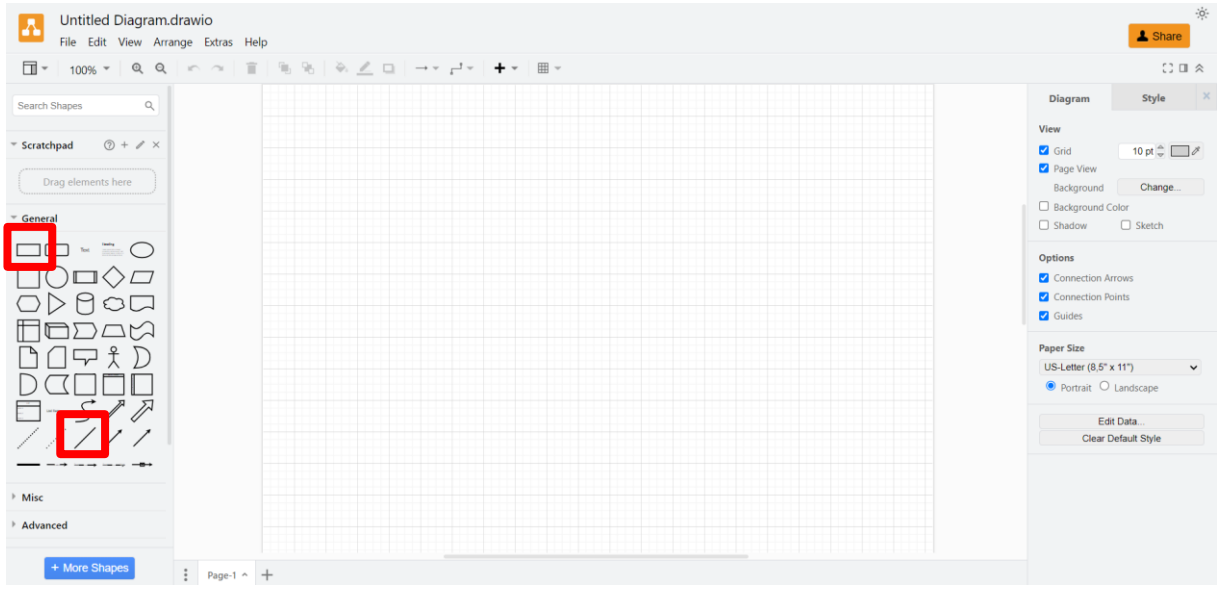
2. Alati

Prije samog početka rada treba odabrati koji će se alati koristiti u ovom radu. U nastavku poglavlja sljedi popis korištenih alata i opis za što se koriste. Naravno nije strogo definirano da se moraju koristiti navedeni alati, ali se u svrhu izrade ovog rada koristili navedeni alati zbog osobne preference.

2.1. Draw.io

Draw.io je online alat za izradu grafova. Vrlo je koristan za stadij planiranja projekta i vrlo jednostavno se može uređivati, bojati, izmjenjivati i sve ostale stvari koje mogu biti korisne za lakšu organizaciju projekta. Nije nužno koristiti Draw.io, moguće je sve napraviti pomoću olovke i papira, no Draw.io omogućava jednostavno uređivanje i dodavanje elemenata što će i biti potrebno zbog iterativne metode koju planiram koristiti tijekom planiranja pojedinog projekta.

Primarno se budu koristili pravokutnici i linije unutar Draw.io-a kako ne bi dodatno vizualno zakomplicirao sliku. Na slici se može vidjeti kako izgleda alat sa označenim elementima koji će se koristiti.



Slika 1. Draw.io alat

2.2. Godot

Godot program je više-platformski program otvorenog koda (eng. *open source*) za razvoj 2D i 3D videoigara kroz zajedničko sučelje. Ono pridonosi značajnu količinu zajedničkih alata kako bi se korisnik mogao fokusirati na izradu videoigara bez ponovnog stvaranja nečega što postoji. Videoigre se mogu jednim klikom izvesti na više platformi, uključujući platforme za stolna računala (Linux, macOS, Windows), mobilne platforme (Android, iOS), te također i web platforme i konzole [1].

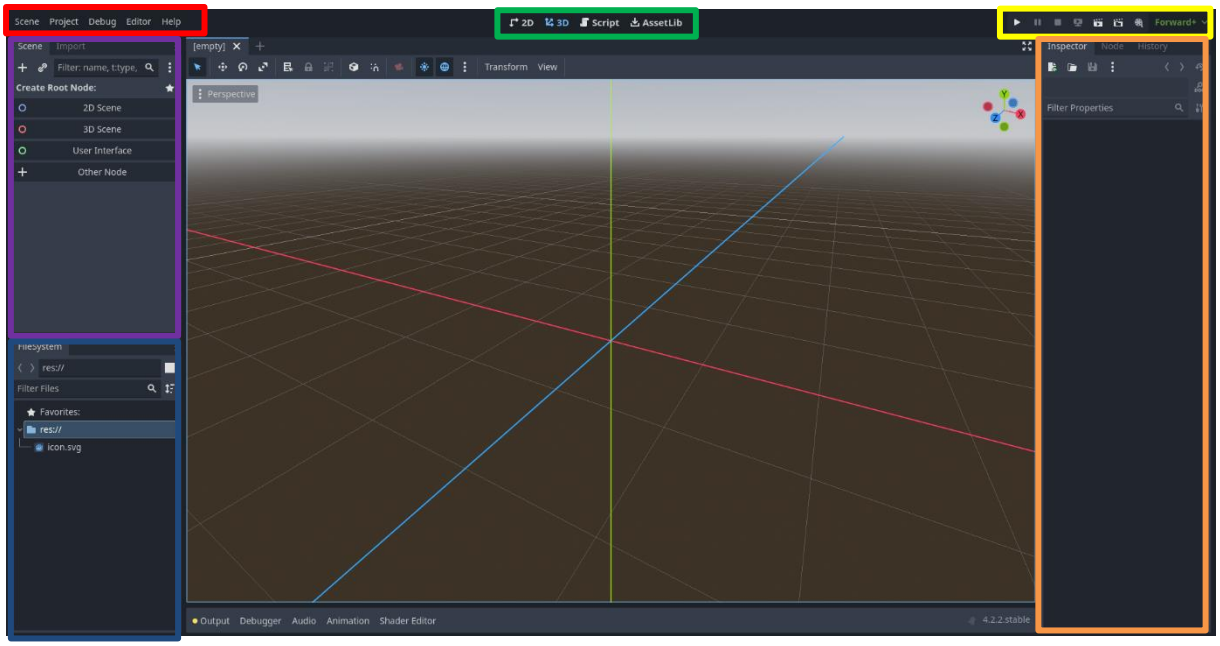
Za razliku od drugih programa za razvoj igara, Godot je baziran na 4 glavna koncepta: node, scene, drvo scena i signali. Nodeovi su najmanji element s kojime se gradi videoigra. Oni se međusobno kombiniraju kako bi se izgradila scena. Scene se međusobno kombiniraju i ugnježđuju u drvo scena. Signali se koriste kako bi nodeovi mogli reagirati na događaje drugih nodeova ili drugih scena [1].

Ukratko opisani koncepti budu bili mnogo jasniji prilikom razvoja igara u ovom radu.

Jezik koji koristi Godot se zove GDScript. To je Godot specifičan programski jezik koji je dobro integriran sa programom i jednostavnom sintaksom koja jako naliku Python jeziku. Godot još podržava C# jezik kao alternativu GDScriptu zbog njegove široke uporabe u industriji razvoja videoigara [1].

Za lakše razumjevanje ovog rada, bude objašnjeno gdje se što nalazi unutar editora kod praznog projekta.

Na samom vrhu se nalazi traka sa različitim izbornicima (Scena, Projekt, Debug, Editor, Pomoć) (crvena boja), nakon kojih slijedi odabir ekrana (2D, 3D, Script editor, biblioteka asseta) (zelena boja) i na kraju gumbovi za pokretanje i testiranje igre (žuta boja). Na lijevoj strani se nalazi prozor koji prikazuje trenutno odabranu scenu i sve njene nodeove (ljubičasta boja). Ispod njega se nalazi prozor sa datotekama (plava boja). Na sredini ekrana je prostor za prikaz same igre i ispod njega razni dodatni prozori (debug, animacijski prozor i drugi). I s desne strane je inspector prozor koji prikazuje detalje o nodeu koji je u fokusu (narančasta boja). Zajedno s njime je Node prozor koji sadrži informacije o signalima nodea u fokusu.



Slika 2. Godot editor

Ovo su osnovne informacije programa i kako on funkcionira. Za detaljniji opise i objašnjenja preporučujem službenu dokumentaciju koja je jako detaljno objašnjena i sadrži puno praktičnih primjera.

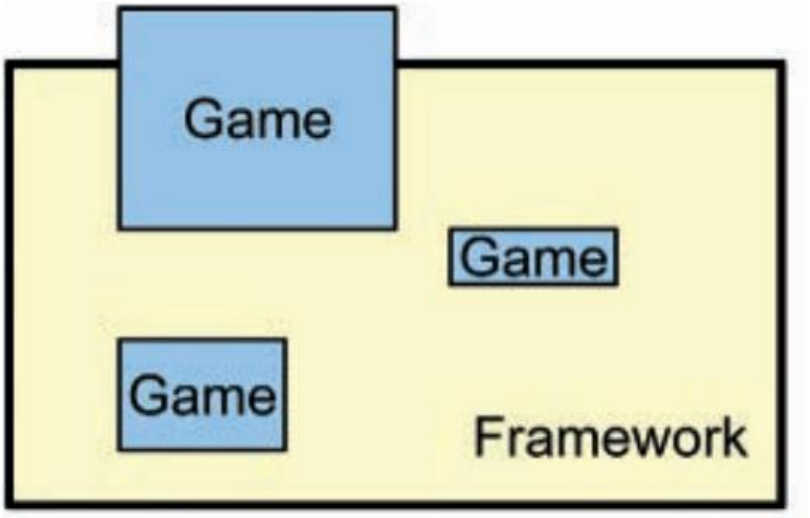
3. Pisanje ponovno iskoristivog koda

Prije pisanja samog koda, potrebno je definirati određene stvari kako bi bilo jasnije na koji način kasnije pisati/razvijati kod.

3.1. Arhitektura

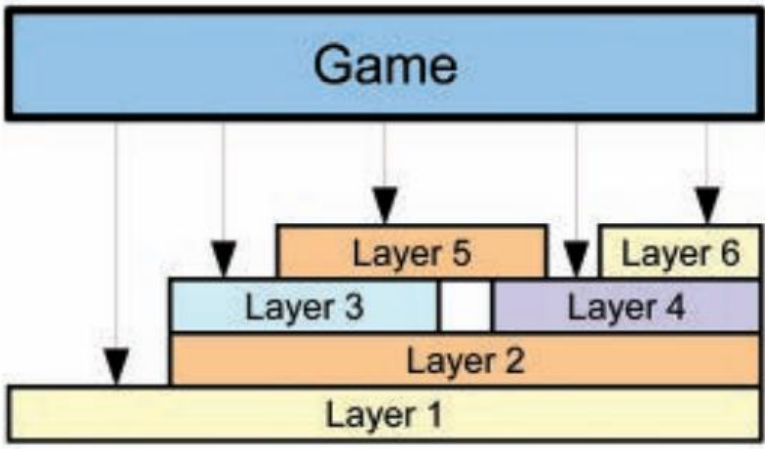
Postoji više različitih arhitektura koje se mogu koristiti prilikom razvoja koda: framework arhitektura, slojna arhitektura i toolkit arhitektura.

Framework arhitektura nije pogodna za korištenje prilikom razvoja igara zato što je jako restriktivna. Ona omogućuje kreiranje programa sa malo mogućnosti dodavanja vlastitih funkcionalnosti, što za ravoj videoigre nije nimalo optimalno [6].



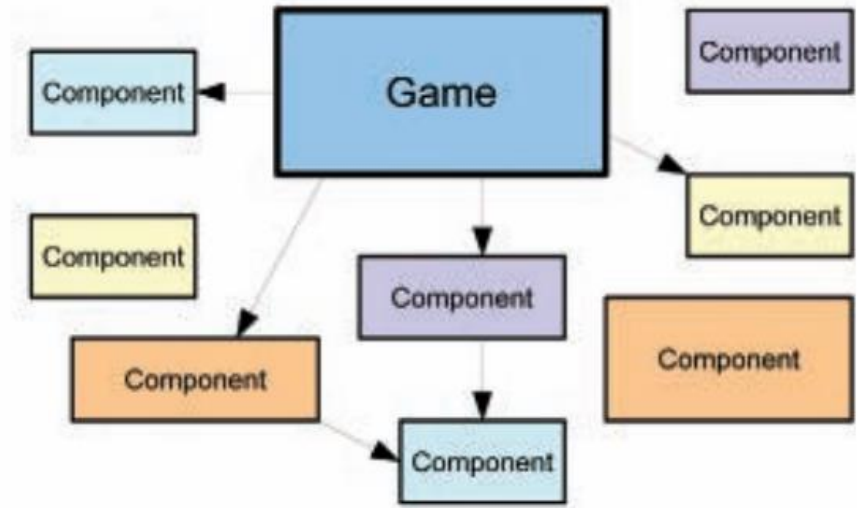
Slika 3. Framework arhitektura [6]

Slojna arhitektura je fleksibilnija od framework arhitekture. Ona gradi svoje fonkcionalnosti sloj po sloj gdje se svaki sljedeći sloj povezuje na prethodni. Može biti korisna pri razvoju videoigara, pogotovo ako se zna pravilno koristiti, no nije modularna. A modularnost je ono što se u ovom radu traži [6].



Slika 5. Slojna arhitektura [6]

Time se dolazi do toolkit arhitekture. Ona je najfleksibilnija arhitektura od 3 navedene jer definira manje, dobro definirane module ili funkcije koje imaju malo ovisnosti jedna o drugoj. To omogućuje korisnicima da koriste bilo koji modul koji im je potreban za potrebe njihove igre. Također korisnici mogu ponovno iskorištavati i modificirati module kako god žele s ciljem proširivanja njihove funkcionalnosti [6].



Slika 4. Toolkit arhitektura [6]

To sve zajedno čini toolkit arhitekturu najoptimalnijom za korištenje prilikom razvoja videoigrica.

3.2. Moduli

Sami moduli u toolkit arhitekturi su pisani objektno orijentirani. To znači da je svaki modul svoja klasa sa vlastitim varijablama i funkcijama. Moduli predstavljaju kompoziciju, a ne nasljeđivanje koje bi se trebalo izbjegavati ako nema potrebe za njom. Naravno nasljeđivanje može biti korisno ako na primjer imamo više različitih neprijatelja i svi trebaju imati iste module,

nasljeđivanje može pomoći kako ne bi trebali stalno dodavati iste module na svakog novog neprijatelja. Iako nasljeđivanje može biti korisno, bolje je koristiti kompoziciju zbog modularnosti koju ono donosi.

U samom Godot programu se može primijetiti da su nodeovi zapravo moduli s kojima se gradi svaka scena. Samim time može se zaključiti da je Godot građen sa kompozicijom kao svojim temeljem i time je intuitivan za korištenje i modularan.

Jako bitna karakteristika modula je da rade samo jednu stvar. To znači da ako modul upravlja životnim bodovima, onda on ne sadrži ikakav kod koji se odnosi na igračevu snagu, već bi to trebao onda biti svoj vlastiti modul. Ukratko, svaki modul treba imati samo jednu jedinu svrhu i po mogućnosti ne ovisiti o drugim modulima.

3.3. Clean Code

Zadnja stvar koja je bitno prilikom pisanja bilo kakvog koda, a to je prakticirati „Clean code“.

Jedna od stvari koju treba raditi je pisati imena klasa, funkcija i varijabli tako da su opisne. Odnosno da se čitanjem samog imena može razumjeti na što se odnosi klasa/funkcija/varijabla.

Zatim je potrebno komentirati vlastiti kod kako bi bilo lakše kasnije shvatiti što smo točno mislili dok smo pisali dani kod, što bi trebalo možda popraviti, da drugi razumiju što radi napisani kod itd. Zvuči jednostavno, ali svi znamo da se programeri to nerado pišu, iako je vrlo korisno dugoročno. Zato je najbolje zakomentirati funkciju čim se napiše i radi ono što treba dok je još sve, tako rečeno, “svježe”.

Isto tako je jako dobro ako pisane funkcije nema puno koda u sebi. mnogo je lakše snalaženje unutar funkcije koja ima 30 do 50 linija koda nego li u onoj koja ima 200. Naravno prvo je uvijek dobro napisati cijelu funkciju i tek kada radi što želimo provjeriti da li je čitljiva. Ako nije onda ju je potrebno rastaviti na više manjih funkcija.

Iako Clean code nije tema rada, smatram da je vrlo važan aspekt razvoja koda za bilo koju svrhu i htio sam na kratko obratiti pozornost na tu temu. Više detalja o Clean codeu se može pronaći na [2].

3.4. Metode rada

Prije samog početka planiranja potrebno je znati na koje načine bi trebalo posložiti sustav videoigre kako bi bio što više ponovno iskoristiv unutar iste igre, ali i van nje. U tu svrhu se bude primarno koristila svojstva nasljeđivanja i kompozicije.

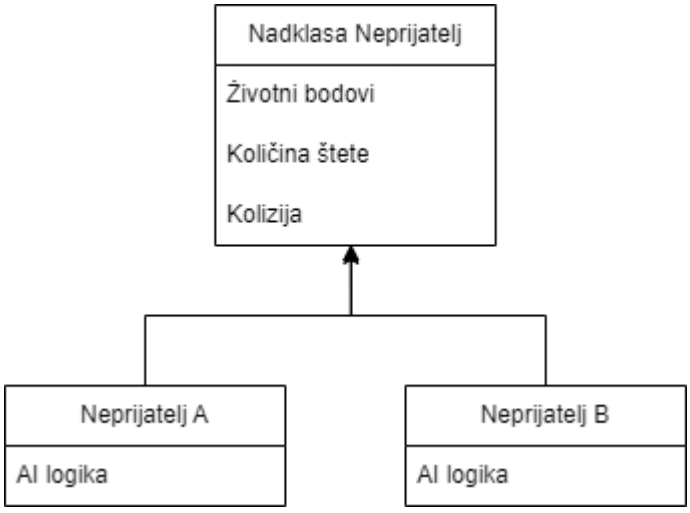
3.4.1. Nasljeđivanje

Nasljeđivanje je mehanizam zasnivanja objekta ili klase na bazi drugog objekta ili klase, zadržavajući sličnu implementaciju [3]. To znači da ako više klasa ima ista svojstva i/ili funkcije, ne treba svaka definirati svoje već ih može naslijediti od iste nadklase.

Time dobivamo sve iste parametre na jednom mjestu i ne ponavlja se kod na više lokacija u sustavu. Vrlo korisno svojstvo koje treba imati na umu prilikom dizajniranja sustava.

Na primjer, ako u igri imamo više vrsta neprijatelja, onda svi oni trebaju imati neke iste karakteristike poput životnih bodova, količinu štete koju nanose igraču, itd. Također, svaki neprijatelj treba imati neku svoju internu logiku koja je unikatna za njega i tu logiku ne stavljamo unutar nadklase neprijatelja, nego u samu klasu neprijatelja.

Slika 6 prikazuje primjer kako bi izgledao dizajn u slučaju da imamo neprijatelja A i B. Oba imaju neke životne bodove, koliziju, rade neku količinu štete, ali svaki ima svoju logiku ponašanja u igri jer isti neprijatelji nisu toliko zabavni u igrama.



Slika 6. Primjer nasljeđivanja

Iako se nasljeđivanje čini super, ne možemo ga koristiti za sve, odnosno nije najpametnije koristiti samo nasljeđivanje za sve jer nam ono može vrlo brzo zakomplicirati cijeli sustav i onda može doći do toga da više niti sami ne znamo gdje se što nalazi i kako je povezano. Zbog toga se koristi kompozicija.

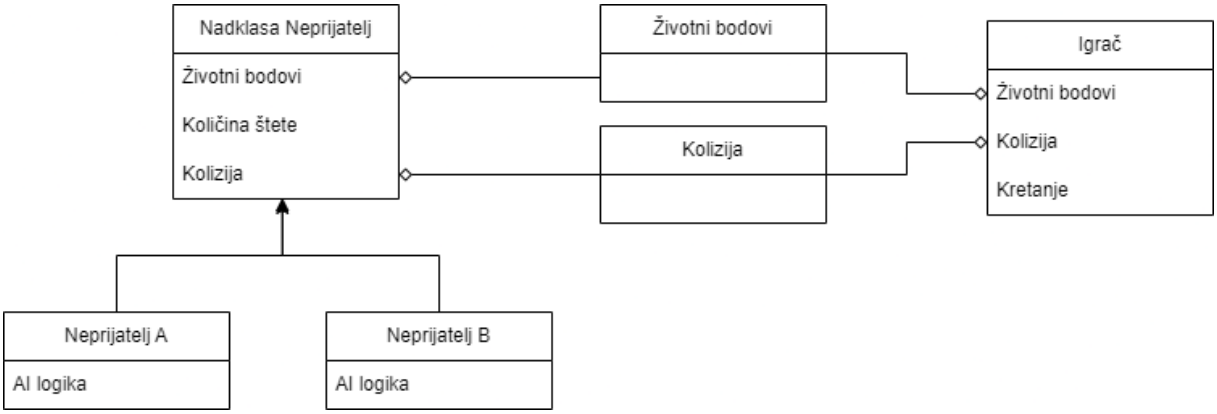
3.4.2.Kompozicija

Kompozicija je tehnika dizajna u objektno-orjentiranom programiranju za implementiranje „ima“ vezu između objekata [4]. Ukratko, kompozicija definira objekte koji opisuju jednu funkcionalnost i mogu se pridružiti više drugih objekata i klasa. Također kroz kompoziciju će se primarno razvijati ponovno iskoristive komponente u ovom radu jer se kroz kompoziciju najlakše može ponovno iskoristiti kod unutar iste igre ali i između različitih igara.

U Godotu, svi objekti nekog tipa (poput Sprite2D) jesu sami po sebi kompozicija jer nisu ovisni o nijednoj drugoj klasi i mogu se pridružiti više različitih klasa koje ih upotrebljavaju.

Naravno, u radu se bude više fokusiralo na razvoj vlastitih kompozicijskih objekata poput sustava za životne bodove i računanje bodova.

Sljedeći primjer (Slika 7) prikazuje kako se može primjer iz nasljeđivanja prenamjeniti da korisit kompozicijske elemente koji se mogu ponovno iskoristiti i za druge dijelove igre, kao na primjer igrača.



Slika 7. Primjer kompozicije

Životni bodovi i kolizija su izvučeni u vlastite kompozicijske klase koje onda može koristiti nadklasa neprijatelj i igrač. Također, klasa životni bodovi se može iskoristiti za svaki objekt koji treba imati životne bodove i nije potrebno ponovno pisati ili kopirati cijelu logiku u svaku klasu gdje ih želimo koristiti. Isto tako je moguće kopirati klasu životni bodovi u drugu igru pošto je, u ovom slučaju, to mehanika svake igre i nema potrebe da prilikom razvoja druge igre ponovno pišemo istu logiku.

3.4.3.Uzorci dizajna

Uzorci dizajna su tipična rješenja za česte probleme u dizajnu softvera. Svaki uzorak je kao plan koji se može prilagoditi rješavanju specifičnog problema dizajna u kodu [5].

Uzorci dizajna se često zamjenjuju sa algoritmima zbog zajedničkog koncepta opisivanja tipičnih rješenja nekog postojećeg problema. Dok algoritam definira točan set akcija koje se trebaju odraditi za dobivanje željnog rezultata, uzorci su više apstraktniji opis rješenja. Kod isto uzorka dizajna u dva različita programa može biti drugačiji [5].

Time zaključujemo da su uzorci dizajna idealni za ponovno iskorištavanje i jednostavniju implementaciju određenih sistema unutar same igre. Pretežito se koriste kako bi neki problem riješili na efektivan način kako bi ga mogli i jednostavno proširiti u slučaju da želimo proširiti sistem sa novim elementima.

Zbog velike količine različitih uzoraka, neće se svi opisivati u ovom radu, već se preporučuje pregled online izvora u literaturi pod brojem 6 gdje su detaljno opisani uzorci te sadrže primjer implementacije za lakše razumijevanje.

Prilikom korištenja novog uzorka dizajna, bude ukratko objašnjen unutar rada kako bi se bolje razumjela implementacija sistema.

4. Starter Projekt

Kako bi maksimalno i jednostavno iskoristili svi moduli razvijeni u ovom radu, napraviti će se Starter projekt. To je projekt koji bude sadržavao sve općenite stvari zajedno sa modulima i služiti će kao odskočna daska za svaki sljedeći projekt. Recimo da se kreira modul za upravljanje životnim bodovima. Čim modul bude testiran i provjereno radi, bude se kopirao u Starter projekt kako bi bio odmah dostupan prilikom izrade idućeg projekta.

Prilikom kretanja u izradu sljedećeg projekta se neće kroz korisničko sučelje Godota kreirao novi projekt, već se bude kopirao cijela mapa Starter Projekt, promijeniti će se ime mape i nakon toga učitat će se projekt u Godotu. Na ovaj način krećemo novi projekt sa unaprijed kreiranim modulima i odmah se može krenuti u izradu igre i mehanika posebnih za novu igru. I ako se definiraju novi moduli u drugoj igri, onda se oni nakon testiranja samo kopiraju u starter kako bi bili dostupni u idućem projekt.

Modul iz Starter projekta možda ne bude savršeno odgovarao potrebama novog projekta. U toj situaciji se može izmijeniti kod modula u novom projektu kako bi bolje odgovarao situaciji. Proširivanjem postojećeg modula se mogu dodati nove funkcionalnosti vezane za mijenjani modul i napravljeno proširenje se može dodati u Starter projekt kako bi se on još više obogatio i omogućio više izbora kod pokretanja novog projekta.

U slučaju da postoji modul, a nije potreban ili pravi nepotrebnu „gužvu“ u projektu, uvijek se može obrisati. Bitno je jedino da postoji kopija u Starter projektu kako bi bio dostupan za budući projekt koji ga bude možda koristio.

Ako postoji velika količina modula, dobra je ideja razvrstati ih tematski kako bi se lakše mogli snalaziti po datotekama projekta.

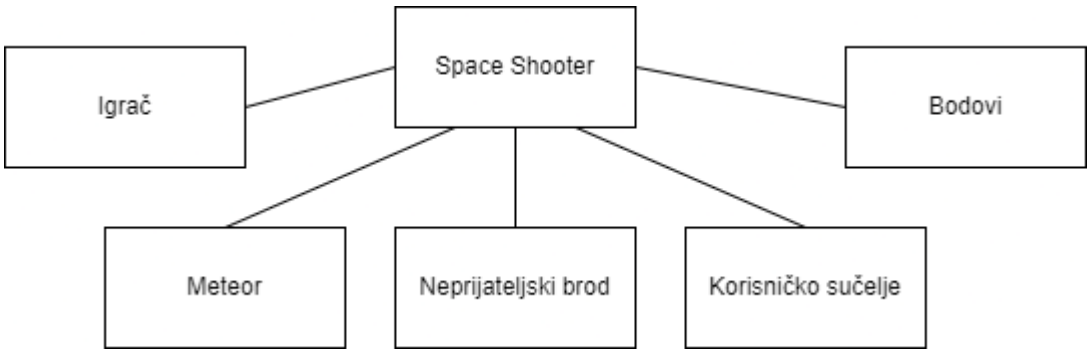
5. Prva igra: Space shooter

5.1. Planiranje

Tema Space Shooter igre je dobivena jer je relativno jednostavan žanr, jasni su elementi igre i super primjer kao uvod u temu rada.

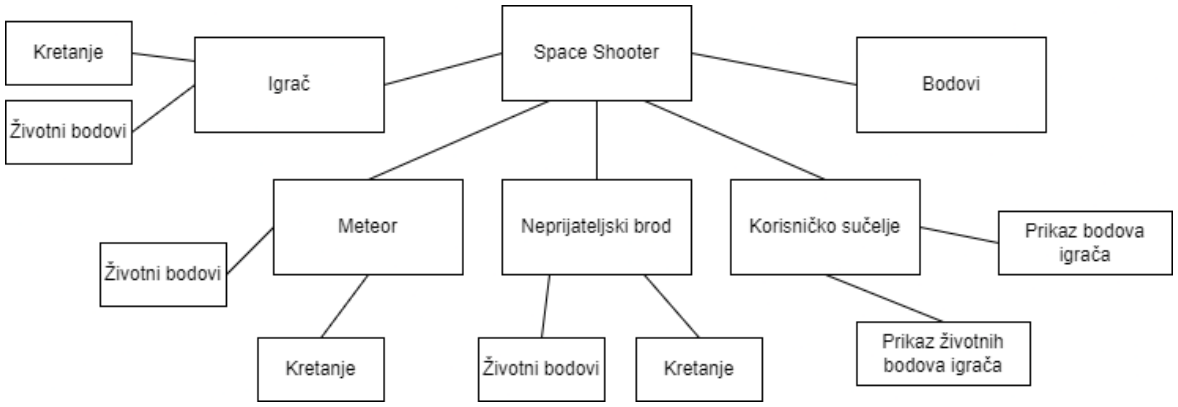
Za početak planiranja je potrebno odlučiti što će sve igra sadržavati. U ovom stadiju je dobro biti što apstraktniji, bez nekih velikih detalja kako bi imali dobru viziju do čega želimo doći, ali i da znamo količinu posla koju bi htjeli napraviti. Nije potrebno pretjerivati sa količinom stvari koju bi dodali, jer ako razvijamo igru na modularan način, onda će biti vrlo jednostavno dodati stvari na osnovu koju napravimo.

Kao i svaka igra u Space Shooter žanru, trebat ćemo imati igrača, protivnika, brojanje bodova i još nekih dodatnih stvari, na primjer dodatne prepreke kao recimo meteor. Nije potrebno odmah na početku imati potpuni dijagram svih željenih elemenata, dovoljno je imati osnovne elemente, te se jednostavno mogu dodati novi elementi. Sljedeća slika prikazuje dijagram s informacijama u ovom odlomku koji se mogao napraviti i na papiru, no napravljeno je u draw.io alatu radi preglednosti.



Slika 8. Prva iteracija elemenata prve igre

Kako je prva, najapstraktnija iteracija gotova, možemo krenuti u drugu iteraciju. Ona će se sastojati od detaljnije razrade svakog elementa koji se definirao. Detaljnija razrada se sastoji od mehanika koje se odnose na pojedini element. To znači da će igrač imati mehaniku kretanja pomoću tipkovnice, trebat će imati sustav za upravljanje životnim bodovima, naravno i neke općenite stvari poput vizuala i kolizije, ali te generalne stvari nije potrebno pisati jer ih skoro svi elementi trebaju imati. Stoga je bolje staviti fokus na one mehanike koje su najbitnije za element koji se trenutno detaljizira. Sljedeća slika prikazuje kako bi graf mogao izgledati nakon definiranja mehanika za svaki element.

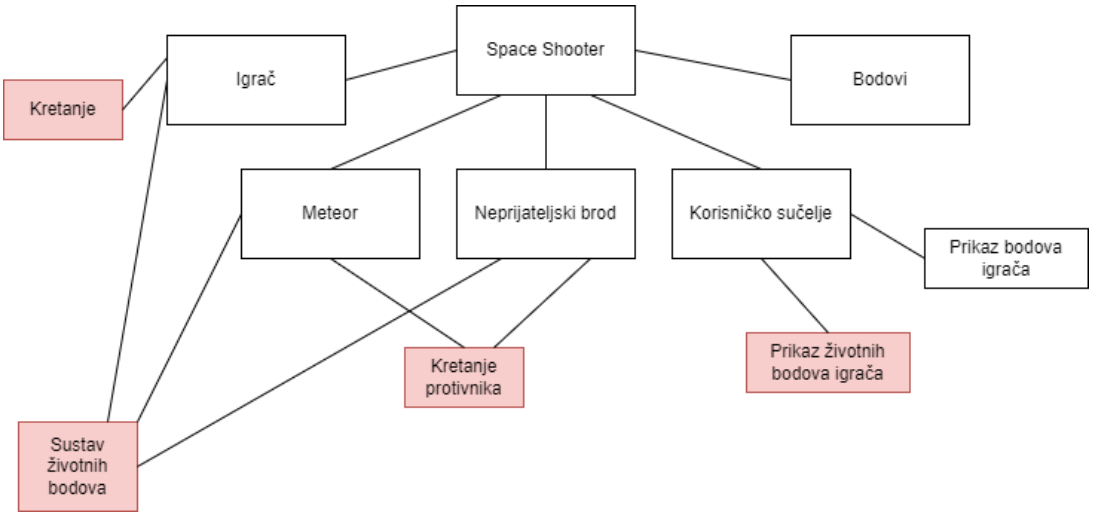


Slika 9. Druga iteracija elemenata prve igre

Na slici druge iteracije mogu se primijetiti elementi koje su mehanike iste za elemente. Iste mehanike se budu grupirale, označile bojom i povezale sa elementima koje ih budu koristile i time su se dobile mehanike koje treba implementirati pomoću metode kompozicije. Tako nastaju ponovno iskoristive komponente koje će koristiti svaki element koji ga treba i može se koristiti vrlo jednostavno između različitih projekata.

Svaka nova mehanika koja se implementira metodom kompozicije se kopira u starter projekt, ako već nije tamo, kako bi imali sve komponente na jednom mjestu.

Sljedeći dijagram prikazuje grupirane iste mehanike.



Slika 10. Treća iteracija elemenata prve igre

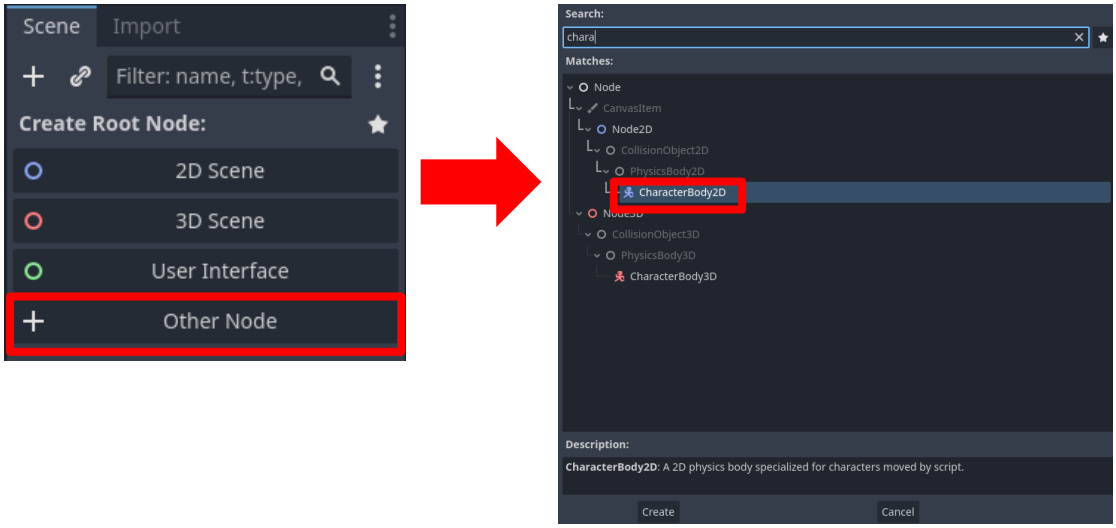
Za zajedničku mehaniku životni bodovi će se kreirati sustav životnih bodova koji će biti modul za životne bodove. Jedino ako smo 100% sigurni da ne trebamo negdje drugdje ponovno iskoristiti mehaniku onda je ne moramo pisati na modularan način. Također, ako se kasnije predomislimo, možemo je izvući u vlastiti modul kako bi se mogla ponovno iskoristiti. U ovom radu će se većinu mehanika pisati na modularan način.

5.2. Razvoj igre

Prvi dio igre koji će se krenuti razvijati se igrač i ostalo će se graditi oko njega kako bi se moglo sve i testirati.

5.2.1. Igrač

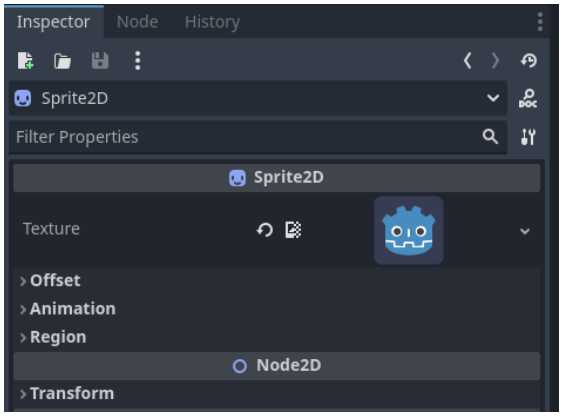
Kako bi kreirali igrača, trebamo napraviti novu scenu odabirom znaka + iznad srednjeg prozora. Zatim na lijevom gornjem prozoru odaberemo vrstu baznog nodea. Mogli bi odabrati „2D Node“, ali postoji mnogo praktičniji node koji se zove „CharacterBody2D“. Kako bi njega odabrali, kliknemo na odabir „Other“ i na listi potražimo željeni node i dvokliknemo ga.



Slika 11. Odabit „CharacterBody2D“ kao bazni node

Svi ostali nodeovi budu bili dodani kao djeca baznog node. Time program zna da se svi ti drugi nodeovi odnose na igrača. Odabrani bazni node ima upozorenje, on zahtjeva 2 nodea bez kojih nemože raditi a to su „Sprite2D“ i „CollisionShape2D“. Njih možemo dodati klikom na znak + na vrhu prozora gdje se vidi bazni node. U otvorenom prozoru potražimo nodeove koji su potrebni i dodaju se.

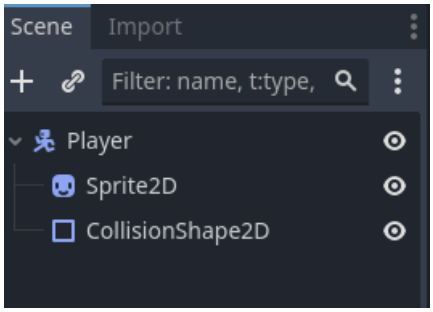
Za početak nije nužno potrebno imati sve vizuale savršene, bolje je fokusirati se na funkcionalnost igre i na kraju dodati sve završne detalje poput vizuala, zvučnih efekata, glazbe, menija, itd. Zato se bude dodala godot ikona (može se pronaći u prozoru ispod prozora s nodeovima) kao vizual u node „Sprite2D“.



Slika 12. Inspektor opcije nodea „Sprite2D“

Za node „CollisionShape2D“ je potrebno odabrati oblik kolizije. Nakon dodavanja kolizije, oblik se može prilagoditi u samom vizualnom pregledu scene na središnjem prozoru.

Igrač će imati skriptu pridruženu na bazni modul koja će sadržavati funkcije igrača specifične za ovu igru i neće se opisivati u sklopu ovog rada.

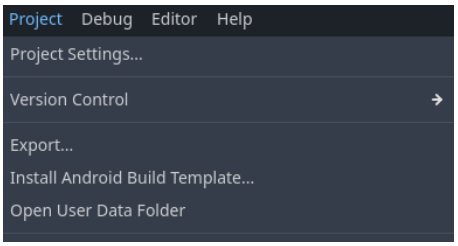


Slika 13. Scena igrača sa osnovnim nodeovima

5.2.2. Modul za kretanje igrača

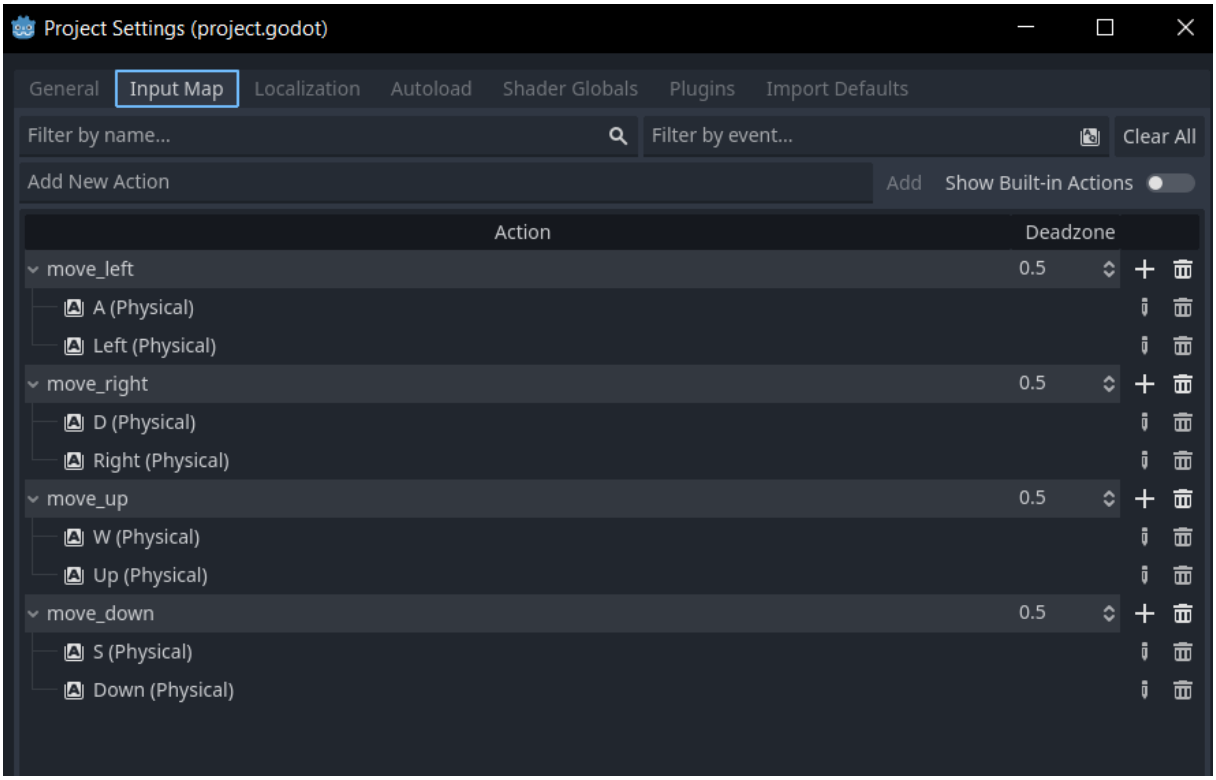
Za sada su dodani samo moduli (nodeovi) koji već postoje u samom Godot programu. Dobra je praksa uvijek provjeriti postoji li već gotovo rješenje za potrebe projekta prije kreiranja vlastitog modula. No u većini slučajeva se bude kreirao vlastiti modul.

Prvi modul koji će se kreirati u ovom radu je modul za 360° kretanje igrača. U Godot programu unutar postavki projekta postoji sekcija koja se odnosi na mapiranje tipki kako bi se lakše referencirale unutar koda. Klikom na „Project“, zatim „Project settings“ u gornjem lijevom uglu ekrana se otvaraju postavke projekta i unutar postavki se odabire kartica „Input map“. Ovdje se nalaze sve mapirane tipke.



Slika 14. Otvaranje „Project Settings“ prozora

Kako bi se mapirale tipke, pri vrhu kartice u polje treba unjeti naziv koji će se odnositi na unesene tipke. Za svrhe rada će se koristiti imena „move_up“, „move_down“, „move_left“ i „move_right“. Kako bi se dodala tipka za svaku kreiranu sekciju, kod svake sekcije treba kliknuti na znak + i u skočnom prozoru odabrati tipku koju želimo dodati u tu sekciju. Naravno nismo limitirani na samo jednu tipku i odmah se budu dodale strelice, tipke „w“, „a“, „s“ i d, te ulazi sa kontrolera.



Slika 15. Izgled mapiranih tipki

Kako je ovo dosta univerzalna stvar u videoigrama, odmah će se ista stvar napraviti i na Starter projektu.

Sada se može krenuti razvijati skripta za kretanje igrača. Skripta prvo treba referencu za bazni node igrača kako bi se kretao. Ta definirana varijabla ispred sebe ima ključnu riječ „@export“ kako bi bila prikazana u inspektor prozor. Time možemo jednostavno povući bazni node igrača na mjesto varijable u inspektoru i odmah imamo referencu na igrača. Također će se na isti način u inspektor prikazati varijabla za brzinu igrača kako bi se brzina lakše mjenjala tijekom testiranja.

```
class_name WASD_Movement

@export var player : Node2D
@export var movement_speed : int
```

Kretanje igrača se treba događati cijelo vrijeme, odnosno svaki frame igre, i zato se nadalje kod piše u finkciji „_process(delta)“ jer se onda izvodi svaki frame igre. Prvo je potrebno znati smjer kretanja igrača, što se može jednostavno definirati sa sljedećom naredbom:

```
var direction = Input.get_vector("move_left", "move_right", "move_up", "move_down")
```

Ova naredba uzima normalizirani vektor pritisnutih tipki koje su maloprije bile mapirane. Zatim se promijeni akceleracija igrača na smjer * brzina nakon čega mora ići naredba „move_and_slide()“ koja primjenjuje definiranu akceleraciju.

```
func _process(_delta):
    var direction = Input.get_vector("move_left", "move_right", "move_up", "move_down")
    player.velocity = direction * movement_speed
    player.move_and_slide()
```

Time je gotova skripta. Izgleda jednostavno, no treba je testirati. Skripte se mogu pridružiti nodeovima, ali node može imati samo jednu skriptu na sebi. Mogli bi pridružiti skriptu baznom nodeu igrača, ali time nebi mogli dodati igraču skriptu koja obrađuje ostale stvari poput detekcije kolizije i drugo. Sama skripta može poslužiti kao node, ali mora imati definirano ime klase, odnosno mora biti klasa. Radi toga je dodano „class_name WASDMovement“ pri vrhu skripte što omogućuje dodavanje skripte kao nodea putem odabira znaka + u prozoru pregleda nodeova i pretraživanje prema imenu klase. Nakon dodavanja nodea se može u inspektoru nodea dodati bazni node igrača, definirati brzinu njegovu i može se testirati.

Skripta radi kako je i zamišljeno da radi, te se sada skripta i scena mogu kopirati u Starter projekt za buduću upotrebu.

Slika 16. Modul kretanja igrača je dodan u uzbor nodeova

5.2.3. Modul za životne bodove

Modul za životne bodove se sastoji od 5 funkcija i nekoliko varijabli. Varijabla „starting_health“ se odnosi na početne životne bodove koji će se upisivati kroz inspektor. „max_health“ je varijabla koja će na početku preuzeti vrijednost „starting_health“ varijable, ali će se mijenjati tijekom igre. „current_health“ varijabla prati trenutne životne bodove.

Osim rečenih varijabli, budu se definirali i signali koji će biti okinuti kada se povećavaju i smanjuju životni bodovi kako bi se na njih moglo spojiti korisničko sučelje za prikaz životnih bodova i pravovremeno ažurirati, te signal kada se smanje životni bodovi na 0.

```
class_name HealthPoints
@export var startingHealth : int
var maxHealth : int
var currentHealth : int
signal death
signal health_added
signal health_subtracted
```

Nadalje, kako bi na početku „max_health“ i „current_health“ preuzeli vrijednost iz „starting_health“ varijable, treba sljedeći kod upisati u „_ready“ funkciju koja se izvršava jednom pri instanciranju modula.

```
func _ready():
    maxHealth = startingHealth
    currentHealth = startingHealth
```

Zatim slijede 4 funkcije za upravljanje životnim bodovima:

1) Funkcija za povećavanje maksimalnih životnih bodova

```
func increaseMaxHP(amount : int):
    max_health += amount
    heal(amount)
```

2) Funkcija za smanjivanje maksimalnih životnih bodova

```
func decreaseMaxHP(amount : int):
    if (max_health - amount) < current_health:
        takeDamage(amount)
    if (max_health - amount) < 1:
            max_health = 1
    else: max_health -= amount
```

3) Funkcija za povećanje trenutnih životnih bodova, odnosno eng. *healing* (ne veće od maksimalnih životnih bodova)

```
func heal(amount : int):
    if (current_health + amount) > max_health:
            current_health = max_health
    else:
            current_health += amount
    for i in range(1, amount):
            health_added.emit()
```

4) Funkcija za smanjivanje životnih bodova, odnosno za primanje štete (ne manje od 0).

```
func takeDamage(amount : int):
    if (current_health - amount) <= 0:
            current_health = 0 death.emit()
    else:
            current_health -= amount
    for i in range(0, amount):
            health_subtracted.emit()
```

Napisanu skriptu ponovno definiramo kao klasu te je stavimo na igrača. Testiranje možemo odraditi privremeno tako da napravimo skriptu za testiranje, pridružimo je jednom nodeu i na pritisak neke tipke se odazove određena funkcija i ispiše se rezultat funkcije u konzolu.

Funkcija radi kako treba te će se kopirati u Starter projekt. Ako dođe do nekih izmjena u skripti, onda je potrebno ažurirati i skriptu u Starter projektu.

5.2.4.Modul za slučajne brojeve

Prije kreiranja modula za meteor, bude se kreirala globalna skripta za odabir slučajnih brojeva. Skripta slučajnih brojeva bude vlastita skripta tako da svi elementi mogu, ako žele, pristupiti i dobiti slučajni broj bez posebnog instanciranja objekta za slučajne brojeve i pisanje funkcije. U ovoj skripti će se također definirati i funkcija za nasumičan smjer i ostale nasumične

funkcije po potrebi. Također će to omogućiti da svi slučajni brojevi imaju isti seed koji definira bazu na kojoj se računa redoslijed nasumičnih brojeva. Ova funkcionalnost omogućuje kasniju funkcionalno unošenja vlastitom seeda i dobivanja isti slijed slučajnih događaja. Možda za ovu igru to nije potrebno, ali je zanimljiv dodatak.

```
var rng : RandomNumberGenerator

func _ready():
    rng = RandomNumberGenerator.new()

func setSeed(seed:int):
    rng.seed = seed

func getRandomFloatNumber(from:float, to:float) -> float:
    return rng.randf_range(from, to)

func getRandomIntNumber(from:int, to:int) -> int:
    return rng.randi_range(from, to)

func getRandomDirection(min_x:float, max_x:float, min_y:float,
max_y:float) -> Vector2:
    var x
    var y
    if min_x == max_x:
        x = min_x
    else:
        x = getRandomFloatNumber(min_x, max_x)
    if min_y == max_y:
        y = min_y
    else:
        y = getRandomFloatNumber(min_y,max_y)
    var direction = Vector2(x, y).normalized()
    return direction

func getRandomElement(pool):
    if pool.size() == 0:
        return null
    var random_index = rng.randi() % pool.size()
        return pool[random_index]
```

Globalne skripte su skripte koje se učitaju jednom, pri pokretanju igre i uvijek su dostupne svim elementima igre. Imaju samo jednu jedinu instancu i zato omogućuju konzistentnost kroz cijelu igru kao što je opisana funkcionalnost seeda. Zbog konzistentnosti i dostupnosti svim skriptama, globalne skripte se ne definiraju kao klase (iako se mogu tako definirati) jer se ne pridružuju elementima igre, već postoje kako bi se koristile po potrebi.

5.2.5.Modul za kretanje neprijatelja

Prije kreiranja neprijatelja, bude se kreirao modul za njegovo kretanje. Kako bi modul bio što univerzalniji, osim kretanja neprijatelja bude obavljao i njegovu rotaciju. Primarno se dodaje zbog meteora, koji će imati istu skriptu kao i neprijatelj, ali postoji i mogućnost kreiranja neprijatelja koji se rotira i nasumično puca. U ovom radu se takav neprijatelj neće napraviti kako se ne bi previše komplicirala igra, ali postoji mogućnost za njegovo dodavanje što je najbitnije.

Za ovaj modul se potrebne varijable koje definiraju smjer kretanja, brzinu kretanja, smjer rotacije i brzina rotacije. Također je potrebna i referenca na bazni node.

```
class_name EnemyMovement
@export var enemy : Node2D
var direction : Vector2
var movement_speed : int
var rotation_speed : int
var rotation_direction : int
```

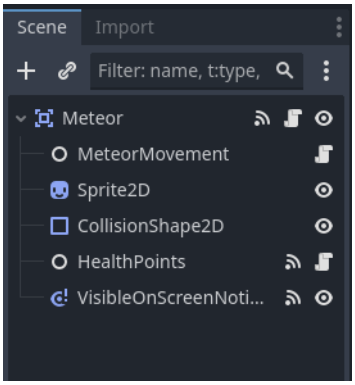
Samo kretanje će se obavljati u funkciji „_process(delta)“ koja pomiče bazni node i rotira ga.

```
func _process(delta):
   enemy.position += direction * movement_speed * delta
   enemy.rotation += rotation_direction * rotation_speed * delta
```

5.2.6.Meteor

Kako bi odmah upotrjebili kreirani modul za kretanje neprijatelja, kreirat ćemo neprijatelja, odnosno prepreku u ovom slučaju.

Meteor će se sastojati od baznog nodea „Area2D“ zato što on detektira ako je objekt ušao u njegov prostor i može obavljati akcije ovisno o tome. Ovaj node zahtjeva koliziju koja se također pridruži, te se također pridruži node za vizualni prikaz meteora i naravno kreirani modul za kretanje te modul za životne bodove.



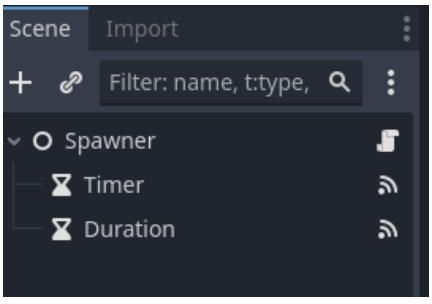
Slika 17. Scena meteora

Neće se prikazati cijeli kod jer on nije toliko bitan za temu rada. Ukratko, kada igrač uđe u prostor meteora, radi se šteta igraču i meteor se miče iz scene. Također ako laser (kasnije opisan) uđe u prostor, smanjuju se životni bodovi meteora i dodaju se bodovi. Ako meteor izađe van ekrana, miče se iz scene.

5.2.7.Modul za pojavu neprijatelja (eng. spawner)

Trenutno imamo jednu vrstu neprijatelja, odnosno prepreku – meteor. Kako bi se meteor pojavljivao u sceni, imat će kontrolu koja će definirati sve njegove specifikacije poput gdje ga stvoriti, brzinu kretanje, smjer kretanja, itd.

Modul za pojavu neprijatelja će biti vlastita scena sa 2 nodea tipa „Timer“ od kojih jedan prati koliko dugo će se neprijatelji pojavljivati, a drugi u kojem razmaku se pojavljuju neprijatelji.



Slika 18. Scena modula za pojavu neprijatelja

Skripta modula sadrži varijable za sve potrebne parametre te ih bude prikazao u inspektoru kako prilikom ponovnog korištenja ne bi ulazili u kod i tražili brojeve koje bi mijenjali.

```
class_name Spawner
```

```
@export var spawn_delay : Vector2
@export var duration : int
@export var is_first:bool
@export var is_boss:bool
@export var min_max_x_percentage: Array[Vector2i]
@export var min_max_y_percentage: Array[Vector2i]
@export var random_rotation_direction : Array [int]
@export var random_direction : Vector4
@export var min_max_rotation_speed : Vector2i
@export var min_max_movement_speed : Vector2i
@export var enemy : PackedScene
@export var next_wawe:Spawner
@export var spawn_timer:Timer
@export var duration_timer:Timer
var child_movement_node
```

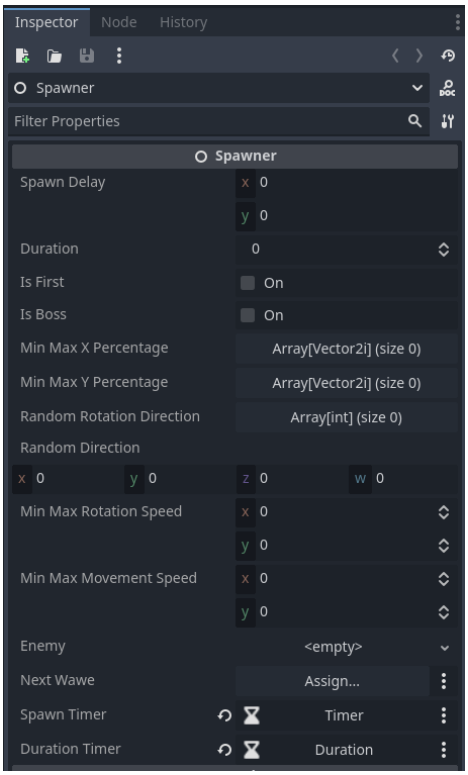
Varijable koje su tipa „Vector2“ su takve jer uvijek za njih treba biti definirana neka minimalna i maksimalna vrijednost, odnosno trebaju brojevi biti u parovima. U inspektoru piše da su to x i y, no koristiti će se za raspon vrijednosti.

Također minimalni i maksimalni raspon za poziciju igrača treba biti upisan kao postotak kako bi se neprijatelji pojavljivali na konzistentnim pozicijama u slučaju mijenjanja rezolucije prozora igre. Pozicije za pojavu neprijatelja su unutar polja kako bi mogli definirati više različitih sekcija gdje bi htjeli da se pojavljuje neprijatelj.

One vrijednosti koje ne želimo ili ne trebamo koristiti, mogu se ostaviti prazne ili na vrijednosti 0.

Varijabla „is_first“ određuje hoće li se krenuti neprijatelji stVarati odmah pri učitavanju u scenu. Pri završetku trenutnog vala neprijatelja, pokreće se sljedeći koji se definiran varijablom „next_wawe“.

Ako val treba stvoriti samo jednog neprijatelja (u ovom slučaju zadnjeg neprijatelja igre), treba se označiti varijabla „is_boss“.



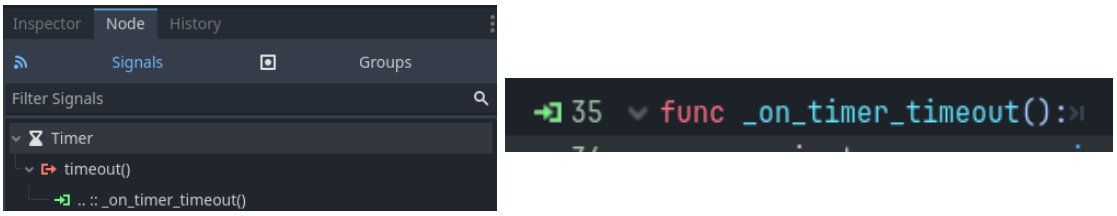
Slika 19. Izgled parametara u inspektoru

Prilikom inicijalizacije, neke varijable se trebaju pridružiti pripadajućim nodeovima i ako je definirani val prvi, pokrene se.

```
func _ready():
    spawn_timer.wait_time = Random.getRandomFloatNumber(spawn_delay.x, spawn_delay.y)
    duration_timer.wait_time = duration
    if is_first:
        spawn_timer.autostart = true
        spawn_timer.start()
        duration_timer.start()
    if is_boss:
        spawn_timer.one_shot = true
```

Kao što je već rečeno, modul sadrži node „Timer“ koji definira u kojem se intervalu pojavljuju neprijatelji. Taj node ima signal „on_timeout“ na kojeg je potrebno spojiti skriptu. To se napravi tako da odaberemo „Timer“ node u prozoru za pregled scene, na prozoru gdje se nalazi inspektor se nalazi i prozor Node u kojemu je popis svih signala koje node može imati. U tom prozoru se dvoklikne „on_timeout“ signal čime se otvara prozor na kojemu trebamo odabrati skriptu koja će se spojiti na signal. Odabirom na skriptu otvara se prozor sa otvorenom

skriptom i generiranom funkcijom koja je spojena na signal. Znamo da je funkcija spojena na signal jer se može vidjeti zelena ikona kraj funkcije.



Slika 20. Popis signala u prozoru Node (lijevo) i funkcija spojena na signal u kodu (desno)

Ova funkcija treba kreirati novu instancu, definirati varijable instance i na kraju dodati instancu u scenu.

Dodane su dvije funkcije, jedna za odabir nasumične pozicije i druga za odabir ostalih nasumičnih varijabli, kojima je svrha preglednost koda.

```
func _on_timer_timeout():
   var instance = enemy.instantiate()
   for child in instance.get_children():
         if child.name.ends_with("Movement"):
               child_movement_node = child
               break
   setRandomPosition(instance)
   setOtherValues()
   add_child(instance)


func setRandomPosition(instance):
   if min_max_x_percentage.size() == 0:
      instance.position.x      =      Random.getRandomIntNumber(-50,
get_viewport().size.x+50)
   else:
      var pool : Array[int]
      for i in min_max_x_percentage:
         pool.append(Random.getRandomIntNumber(i.x                *
   get_viewport().size.x / 100, i.y * get_viewport().size.x / 100))
      instance.position.x = Random.getRandomElement(pool)
   if min_max_y_percentage.size() == 0:
         instance.position.y = -50
   else:
         var pool : Array[int]
         for i in min_max_y_percentage:
```

```
                pool.append(Random.getRandomIntNumber(i.x * get_viewport().size.y / 100, i.y * get_viewport().size.y / 100))
                instance.position.y = Random.getRandomElement(pool)

    func setOtherValues(instance, direction):
        var direction = Random.getRandomDirection(random_direction.x, random_direction.y, random_direction.z, random_direction.w)
        if child_movement_node != null:
            child_movement_node.direction = direction
        if min_max_rotation_speed.x != 0 && min_max_rotation_speed.y != 0:
            child_movement_node.rotation_speed = Random.getRandomIntNumber(min_max_rotation_speed.x, min_max_rotation_speed.y)
            if random_rotation_direction.size() != 0:
                child_movement_node.rotation_direction = Random.getRandomElement(random_rotation_direction)
                child_movement_node.movement_speed = -Random.getRandomIntNumber(min_max_movement_speed.x, min_max_movement_speed.y)
```

Kako bi se moglo testirati sve do sada napravljeno, kreiramo novu scenu nazvanu „Level“. U nju stavimo scenu igrača i modul za pojavu neprijatelja. Definiramo sve njihove potrebne parametre i pokrenemo igru.

Primijetimo da možemo pomicati igrača i da se pojavljuju meteori sa nasumičnim smjerom, brzinom i vrtnjom. Također rade štetu igraču pri čemu nestaju, no ta je logika unutar dodatne skripte specifične za meteor.

5.2.8.Modul za kretanje lasera

Igrač treba moći uništiti meteore i neprijatelje kako bi povećao bodove. Za tu svrhu će pucati lasere koji će uništavati protivnike i time povećavati bodove igraču.

Modul za kretanje lasera će imati varijable za brzinu kretanja, varijablu za bazni node, provjeru da li je preokrenut ( ), te varijablu za smjer.

```
class_name LaserMovement
@export var laser : Node2D
```

```
@export var inverted:bool
@export var movement_speed:int
var direction
```

Uz njih će u funkciji „_ready()“ provjeravati da li je laser preokrenut i time definirati njegov smjer. Ako je definiran smjer lasera (na baznom nodeu), onda će se kretati tim smjerom.

```
func _ready():
laser = get_parent()
   if inverted:
         direction = Vector2(0,1)
   else:
         direction = Vector2(0,-1)
   if laser.direction != null:
         direction = laser.direction
```

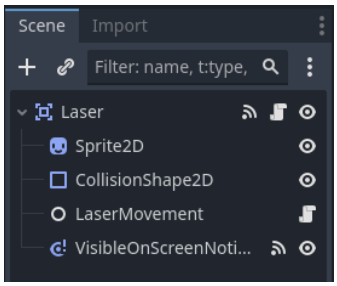
Isto kao i modul za kretanje neprijatelja, laser će imati funkciju „_process(delta)“ u koja će pomicati laser.

```
func _process(delta):
   laser.position += direction * movement_speed * delta
```

5.2.9. Laser

Laser će, kao i meteor, imati modul za kretanje i skriptu specifičnu za laser te će biti vlastita scena. Biti će, kao i meteor, nova scena sa baznim modulom „Area2D“ i sve potrebne nodeove koje on zahtjeva.

Skripta specifična za laser sadrži varijablu koja definira koliko štete radi te funkciju spojenu na signal „on_area_entered()“ koja detektira ulaz drugog objekta i radi mu štetu.



Slika 21. Scena lasera

Potrebno je dodati skripti za igrača da može pucati lasere pritiskom na željenu tipku. Još je bolje definirati novu akciju, kao što je bilo definirano za kretanje i u tu akciju definirati

tipku. Ako sada testiramo igru, imamo igrača koji se može kretati i pucati te meteore koji se mogu uništavati i raditi štetu igraču.

5.2.10. Modul za bodove

Kako se trenutno mogu meteori uništavati, trebaju i dati bodove igraču. Za upravljanje bodova će se kreirati jednostavan modul za bodove koji će biti globalna skripta koja prati bodove igre i resetira se prilikom ponovnog pokretanja igre.

Globalne skripte kojima mogu pristupiti svi elementi igre spadaju u uzorak dizajna Singleton. Singleton je uzorak dizajna čija je karakteristika jedna instanca kroz cijeli program. To znači da mu svi elementi mogu pristupiti i da će podaci pohranjeni unutra biti isti za sve elemente (posljedica jedne instance). Treba biti na oprezu da se ne koristi u pretjeranim količinama rečeni uzorak, zato što je to loša praksa pri pisanju koda i treba se koristiti samo ako je potrebno.

Za te potrebe modula je definirana varijabla za bodove, funkcija za resetiranje bodova i funkcija za dodavanje bodova. Kada se bodovi promjene, poslati će se signal kako bi se korisničko sučelje spojilo na njega i pravovremeno ažuriralo i prikazalo aktualne bodove.

```
var score : int
signal score_changed
func _ready():
    score = 0

func add(amount:int):
    score += amount
    score_changed.emit()

func reset():
    score = 0
    score_changed.emit()
```

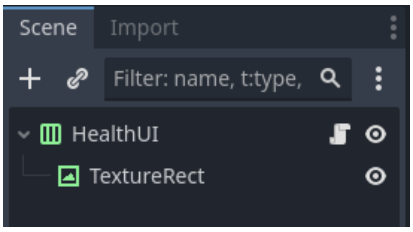
Kao i skriptu za nasumične brojeve, ova skripta se također stavi u automatsko učitavanje projekta kako bi svi elementi mogli pristupiti skripti.

5.2.11. Modul za prikaz životnih bodova u korisničkom sučelju

Kako bi životni bodovi bili prikazani igraču, trebaju biti u korisničkom sučelju. Za te svrhe će se kreirati modul jer neke videoigre prikazuju životne bodove na način da postoji broj vizuala koliko igrač ima životnih bodova.

Kreirat će se nova scene čiji će bazni node biti „HBoxAlign“. On jednoliko raspoređuje svu svoju djecu horizontalno sa željenim razmakom, što ga čini idealnim za prikaz bodova. Zatim će se dodat node „VisualRect“ umjesto „Sprite2D“ zato što „Sprite2D“ ne spada u „Control“ tip nodeova koji se koriste za korisničko sučelje. Jednostavno konfiguriramo napravljen node sa željenim vuzualom i zatim kreiramo skriptu za bazni node.

Baznom nodeu će se dodati skripta koja se treba spojiti na signale životnih bodova igrača kako bi ažurirala prikazane životne bodove. Osim toga na samom početku treba prikazati sve životne bodove sa kojima igrač započinje igru.

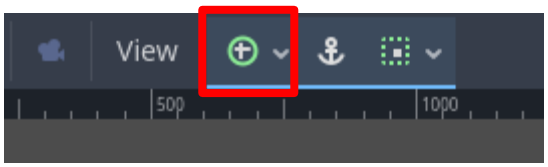


Slika 22. Scena modula životnih bodova u korisničkom sučelju

Spajanje na signal se može napraviti kroz kod pomoću naredbe „connect“. Koristi se u slučajevima kada se ne može kroz korisničko sučelje Godota spojiti na signal.

```
@export var health_visual : TextureRect

func setup(player:Node2D):
    var health_points_node = player.get_node_or_null("HealthPoints")
    for i in range(1, health_points_node.current_health):
        add_child(health_visual.duplicate())
    health_points_node.connect("health_added", _on_health_added)
    health_points_node.connect("health_subtracted",
_on_health_subtracted)

func _on_health_added():
    add_child(health_visual)

func _on_health_subtracted():
    get_child(0).queue_free()
```

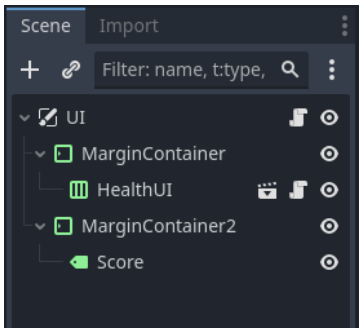
Svi nodeova koji su tipa „control“ mogu imati definirano sidro. Sidro određuje gdje će biti prikazano u slučaju da se promijeni rezolucija igre i time bude konzistentno.



Slika 23. Definiranje sidra u Godotu

5.2.12. Korisničko sučelje

Sljedeće će se kreirati scena sa baznim nodeom „CanvasLayout“ koji označava prikaz sve svoje djece na korisničkom sučelju. Ova scena će služiti za sve elemente korisničkog sučelja specifični za ovu igru, iako se može kreirati od njega modul za jednostavno korisničko



Slika 24. Scena korisničkog sučelja

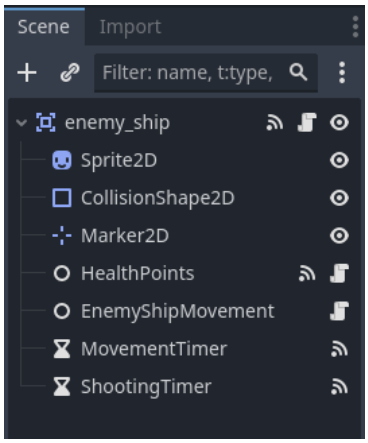
sučelje. No onda će ono ovisiti o drugim modulima (modul ovog poglavlja i modul za bodove) i treba pripaziti oko mijenjanja ovisnosti.

Trenutno korisničko sučelje na početku poziva „setup(player) funkciju modula prethodnog poglavlja kako bi ga inicijalizirala. Zatim inicijalizira prikaz bodova, te se spaja na signal globalne skripte bodova kako bi ažuriralo bodove na korisničkom sučelju prilikom povećanja bodova.

5.2.13. Neprijateljski brod

Neprijateljski brod će, kao i meteor, biti svoja scena sa baznim nodeom „Area2D“ i svim potrebnim nodeovima. Uz njih će imati skriptu specifičnu za brod koja provjerava da li je brod

pogođen, stvara laser itd. Mora isto tako imati i node za životne bodove kako bi mogao biti uništen.



Slika 25. Scena neprijateljskog broda

5.2.14. Modul kretanja neprijateljskog broda

Zamisao kretnje protivničkog broda je ta da nakon nekog vremena kretanja stane i krene pucati. Zbog toga ne može koristiti identičnu skriptu kretanja, ali je može dopuniti svojom funkcionalnošću. Stoga će se iskoristiti svojstvo nasljeđivanja kako bi se proširilo modul za kretanje neprijatelja.

Proširenjem će se dodati varijabla za node „Timer“ koji će definirati kada neprijateljski brod treba pucati. U scenu neprijateljskog broda će se dodati još jedan „Timer“ node koji će nakon isteka brzinu kretanja broda staviti na 0 i započeti pucanje.

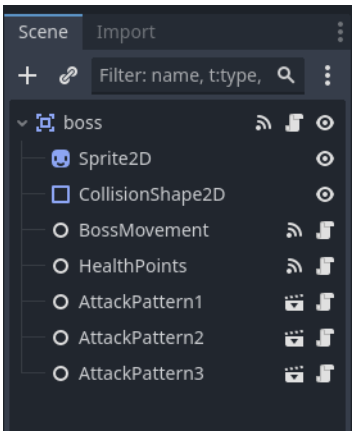
```
extends EnemyMovement
class_name EnemyShipMovement
@export var shooting_timer : Timer
func _on_movement_timer_timeout():
   movement_speed = 0
   shooting_timer.start()
```

Nakon dodavanja modula neprijateljskom brodu, može se testirati pomoću spawnera definiranog u ranijem poglavlju sa svim specificiranim parametrima.

5.2.15. Boss bitka

Glavni, zadnji neprijatelj igre (za ove svrhe će se zvati eng. *boss*) bude iskoristio najkompliciraniju skriptu u ovoj igri, a to je skripta za stvaranje neprijatelja. On bude funkcionirao na načina da ima nekoliko napada gdje stvara protivnike i prepreke koje igrač može izbjegavati ili uništiti za dodatne bodove kako bi mogao raditi štetu samom neprijatelju.

Boss ima vlastitu skriptu na baznom nodeu koja sadrži polje spawnera, tako da se mogu dodati po želji i jednostavno modificirati kroz inspektor. Boss će birati nasumični element polja i pokrenuti ga, te kada istekne njegovo vrijeme (ima definirano koliko traje) bude ponovno odabrao nasumični spawner, odnosno napad na igrača.



Slika 26. Scena bossa

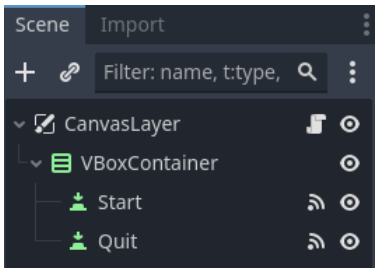
Također će imati pridruženu skriptu za neprijateljski brod, jer će se također zaustaviti nakon kratkog kretanja.

Time se vrlo brzo kreirao boss pomoću modula za stvaranje neprijatelja, te se mogu jako jednostavno modificirati njegovi napadi.

5.2.16. Modul za meni

Ovaj vrlo jednostavni modul je napravljen kao nova scena sa baznim nodeom „CanvasLayer“. U njega je dodan node „VerticalBoxAlign“ sa 2 nodea djece koji su gumbovi. Jedan gumb je za pokretanje igre, a drugi za gašenje igre, te je centrirano na sredinu ekrana i dodani su tekstovi na gumbove.

Meni je vrlo jednostavan tako da se mogu dodati vlastiti vizuali i željene opcije. Iako je modul vrlo jednostavan, može se lako proširiti i funkcionalno ima minimalnu količinu stvari da bude meni igre.



Slika 27. Scena modula za meni

Prilikom pritiska Start gumba, scena se mijenja na „level“ i pritiskom na Quit gumba izlazi se iz igre. Funkcije su spojene na signale gumbova „on_pressed“. Također je potrebno i staviti početnu scenu projekta na meni modul.

```
func _on_start_pressed():
    get_tree().change_scene_to_file("res://Scenes/level.tscn")
func _on_quit_pressed():
    get_tree().quit()
```

5.3. Dodavanje vizuala, zvukova, glazbe

Nakon svih razvijenih elemenata igre i provjere radi li sve kako je zamišljeno, vrijeme je za dodavanje završnih dijelova igre. Vizuali, zvukovi i glazba su preuzeti s online izvora i neće se detaljno prikazati niti objašnjavati kako je to postignuto, jer je cilj rada prikazati kako razvijati modularne dijelove igre, a ne kako napraviti kompletnu igru do završetka.

Jedina stvar koja je ostala za napraviti je provjeriti jesu li svi željeni moduli kopirani u Starter projekt. Moduli koji su kopirani u starter projekt su sljedeći: modul za kretanje igrača, modul za životne bodove, modul za slučajne brojeve, modul za kretanje neprijatelja, modul za pojavu neprijatelja, modul za kretanje lasera, modul za bodove, modul za prikaz životnih bodova u korisničkom sučelju i modul za meni.

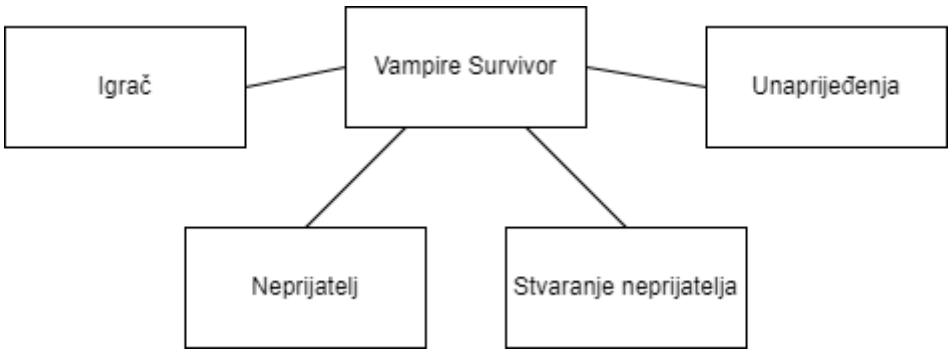
Uz navedene module kopirane su i scene igrača i lasera bez njihovih skripti, pošto su te skripte specifične za ovu igru. Razlog kopiranja scena je taj da smanjimo repeticiju kreiranja iste scene u novoj igri, već postoji odskočna daska i odmah se može krenuti razvoj specifičnih elemenata igre.

6. Druga igra: Vampire Survivors

Za početak rada igre, kopiran je Starter projekt u kojemu se nalaze sve skripte koje smo odlučili ponovno iskoristiti iz prve igre. U skripte, neke scene su također ponovno kopirane poput igrača, spawnera i lasera. Scene su se kopirale jer će se i ubuduće sastojati od istih nodeova i skripti koje se mogu modificirati.

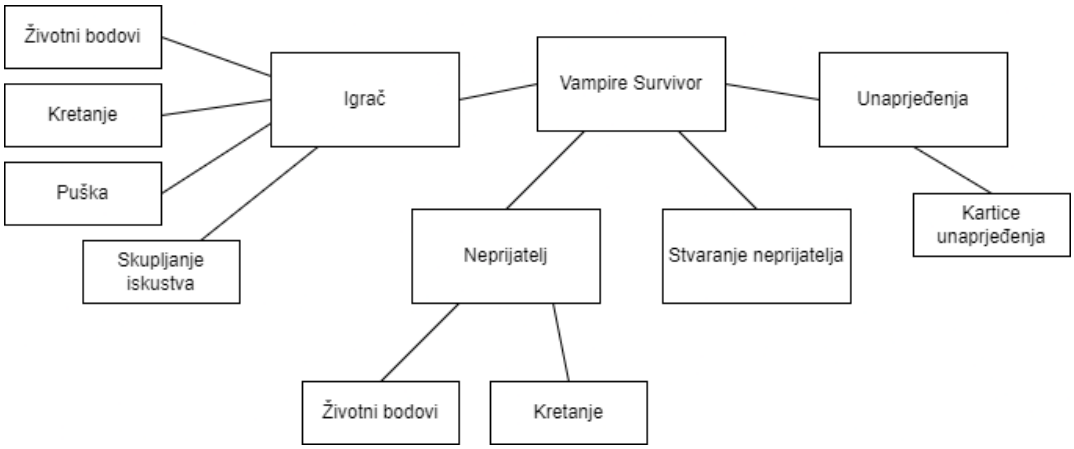
6.1. Planiranje

Vampire Survivor stil igre, u najapstraktnijem razmišljanju, sadrži igrača, protivnike koji se kreću prema igraču, upravljač za valove neprijatelja, unaprjeđenja, objekte koji su na putu neprijatelju i igraču i početno oružje kojim će se igrač boriti s neprijateljima (u ovom slučaju to je puška).



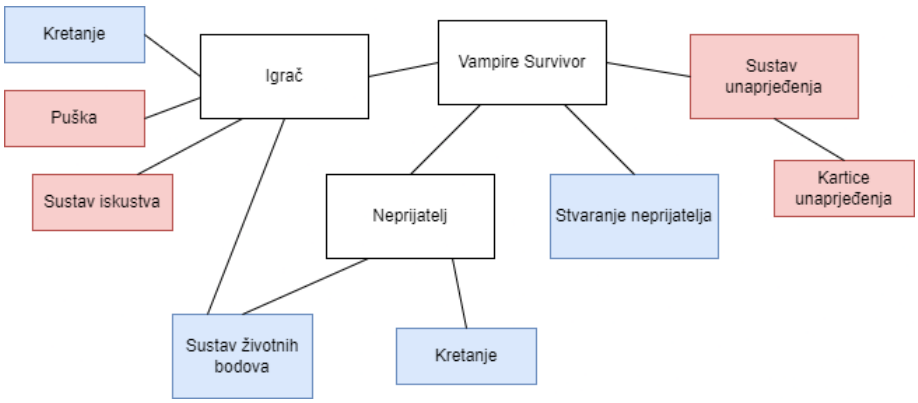
Slika 28. Prva iteracija elemenata druge igre

Sljedeći korak je definiranje mehanika koje će pojedini element koristiti. Igrač će koristiti životne bodove, imat će domet za skupljanje iskustva, treba imati pušku i kretati se. Ostale mehanike za pojedini element se mogu vidjeti sa slike ispod.



Slika 30. Druga iteracija elemenata druge igre

I za kraj, sve se iste mehanike spoje i definiraju se moduli koji će se razviti. Rezultat se može vidjeti sa slike ispod (bojom su označeni moduli). Mogu se uočiti moduli koji su isti kao i u prošloj igri poput sustava za životne bodove. Takvi moduli su označeni plavom bojom, dok su novi moduli označeni crvenom bojom.



Slika 29. Treća iteracija elemenata druge igre

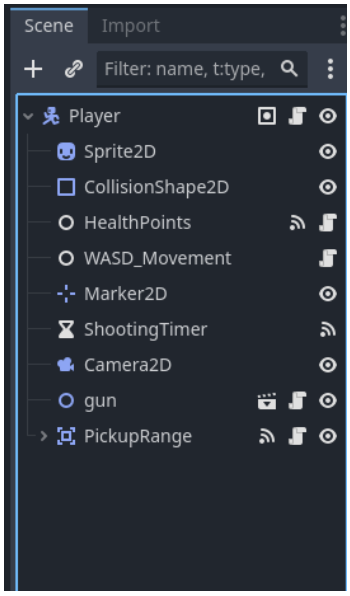
Iako moduli već postoje, prema potrebi se mogu i modificirati što će se i raditi za ovu igru. Naravno ako se može koristiti modul bez modifikacije, to bolje za nas kao kreatore videoigre. No nekad je bolje modificirati kako bi modul bolje odgovarao slučaju videoigre.

6.2. Razvoj

6.2.1. Modifikacija igrača

Pošto je scena igrača u Starter projektu, krenuti će se od njega. Trenutno, scena u sebi sadrži bazni node „CharacterBody2D“, koliziju za bazni node, „Sprite2D“ node, „Marker2D“ node, node za kretanje i node za životne bodove. Pri otvaranju javlja grešku da ne postoji „player.gd“ skripta koja se nije kopirala u Starter projekt jer je ta skripta bila specifična za prošlu igru. Tako da će se za ovu ponovno kreirati prazna skripta za igrača.

U ovom stilu igre kamera je stalno na igraču, on puca na neprijatelje automatski i može skupljati objektu koji povećavaju njegovo iskustvo (eng. *Experience*). Time će se dodati nodeovi koji će omogućiti igraču navedene stvari. Slika prikazuje kako izgleda scena igrača nakon razvoja svih modula igre, te će kasnije biti opisana scena „gun“ nodea sa slike.



Slika 31. Izgled scene igrača sa svim modulima

Skripta samog igrača, osim za skupljanje signala kada ostane bez životnih bodova i ostalo, bude imao varijable koje referenciraju njegove module poput životnih bodova i kretanje kako bi kasnije mogao sustav za unaprjeđenja mogao jednostavnije upravljati vrijednostima igrača poput maksimalni životni bodovi ili brzina kretanja.

Sami moduli za životne bodove i kretanje igrača su ostali nepromijenjeni. Igrač je dodan u grupu nazvanu „player“ kako bi drugi moduli po potrebi mogli imati lakši pristup igraču ako trebaju.

6.2.2. Modificiran modul za pojavu neprijatelja (eng. *spawner*)

Spawner modul je modificiran zato što u ovoj igri ne želim toliko slučajnih vrijednosti. Naravno taj dio koda se mogao ostaviti i jednostavno sve ostaviti na 0 tijekom definiranja valova neprijatelja, ali zbog čišćeg koda sam odlučio prilagoditi skriptu igri (što se i treba napraviti, nije moguće da sve bude univerzalno).

Kako već postoji logika, treba samo prilagoditi što se želi koristiti i obrisati što se ne koristi. Stoga će se obrisati varijable za odabir nasumičnog položaja, kretnje, brzine i smjera. Umjesto njih će se dodati varijable za nasumičnu količinu neprijatelja i brzina neprijatelja.

Također su dodane dvije pomoćne varijable, od kojih jedna referencira igrača, a druga definira udaljenost od igrača.

```
@export var spawn_delay : float
@export var duration : int
@export var enemy_amount:Vector2i
@export var enemy_movement_speed : int
@export var is_first:bool
@export var is_boss:bool
@export var enemy : PackedScene
@export var next_wawe:Spawner
@export var spawn_timer:Timer
@export var duration_timer:Timer
var child_movement_node
var player
var spawn_offset
```

Može se lako uočiti da je veliki dio varijabli ostao isti, što se postiglo kopiranjem cijelog koda u Starter projekt i nije bilo potrebno ponovno razmišljati koje su sve vrijednosti potrebne.

Sljedeća je „_ready()“ funkcija koja inicijalizira sam spawner i neke njegove varijable. Dobar dio funkcije je isti. Dodan je kod koji definira igrača te definira udaljenost od igrača.

```
func _ready():
    spawn_timer.wait_time = spawn_delay
    duration_timer.wait_time = duration
    if is_first:
          spawn_timer.autostart = true
          spawn_timer.start()
          duration_timer.start()
    if is_boss:
          spawn_timer.one_shot = true
    player = get_tree().get_first_node_in_group("player")
    spawn_offset = get_viewport().size.x/2 + 200
```

Nadalje slijedi kod za kreiranje neprijatelja. Kod koji je ovdje dodan odabire količinu neprijatelja koji se treba stvoriti i za svakog neprijatelja slijedi kod napisan u Starter projektu za kreiranje neprijatelja. Razlika je što druga pomoćna funkcija „setOtherValues()“ je zamijenjena sa linijom koja određuje brzinu neprijatelja. To je napravljeno jer nema puno dodatnih vrijednosti koje se mijenjaju i nema potrebe da jedna linija bude u svojoj funkciji.

Što se tiče funkcije za postavljanje pozicije, ona je u potpunosti izmijenjena tako da računa nasumični smjer i definira poziciju gdje treba kreirati neprijatelja.

```
func _on_timer_timeout():
    var number_of_enemies = Random.getRandomIntNumber(enemy_amount.x, enemy_amount.y)
    for i in range(0, number_of_enemies):
        var instance = enemy.instantiate()
        for child in instance.get_children():
            if child.name.ends_with("Movement"):
                child_movement_node = child
                break
        setRandomPosition(instance)
        child_movement_node.movement_speed = enemy_movement_speed
        add_child(instance)

func setRandomPosition(instance):
    var direction = Random.getRandomDirection(-1, 1, -1, 1)
    instance.position = player.position + direction * spawn_offset
```

Na kraju postoji još i funkcija koja pri završetku vala pokreće sljedeći ako je definiran koja nije promijenjena.

6.2.3.Modificiran modul za kretanje lasera

Kretanje lasera je bilo malo neobično definirano u prvoj igri, stoga se ovdje modificirala skripta kako bi bila univerzalnija. U potpunosti se uklonila „_ready()“ funkcija i „_process(delta)“ funkcija pomiče laser prema definiranom smjeru.

```
class_name LaserMovement
@export var laser : Node2D
var movement_speed:int
var direction
func _process(delta):
    laser.position += direction * movement_speed * delta
```

6.2.4. Laser

Laser se sastoji od istih nodeova, zbog čega i je kopiran u Starter projekt, te ima malo izmijenjenu svoju skriptu koja prima smjer i brzinu kretanja i prosljeđuje ih modulu za kretanje lasera unutar „_ready()“ funkcije.

Ako detektira ulazak tijela, ako nije statično onda mu radi štetu. Statična tijela će biti sve prepreke u ovoj igri poput drveća kako ne bi bila prazna mapa. Opis implementacije se nalazi u kasnijem poglavlju.

6.2.5.Modificiran modul za kretanje neprijatelja

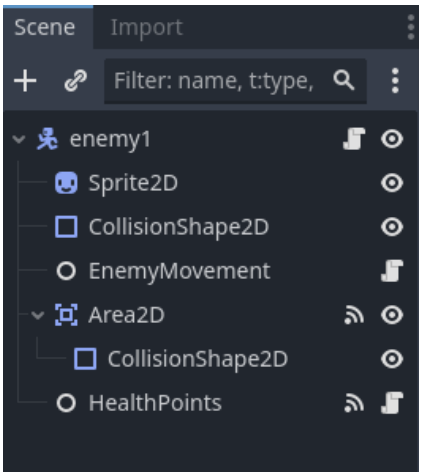
Neprijatelji se kreću drugačije u ovoj igri za razliku od prošle. U ovoj kretanje nije nasumično, već se uvijek kreću prema igraču. Zato je njihov smjer kretanja uvijek prema igraču na koga imaju referencu. Osim toga sadrže varijablu za brzinu kretanja.

```
class_name EnemyMovement
@export var enemy : Node2D
@export var movement_speed : int
var player var direction : Vector2
func _ready():
   player = get_tree().get_first_node_in_group("player")

func _physics_process(_delta):
   direction = (player.position - enemy.position).normalized()
   enemy.velocity = direction * movement_speed
enemy.move_and_slide()
```

6.2.6. Neprijatelj

Kako u ovoj igri želim da neprijatelj može, tako rečeno, zapeti na statična tijela igre, bazni node je isti kao i igračev „CharacterBody2D“. Ovaj node ne služi samo za kretanje igraća, već za kretanje svih tijela koja trebaju imati interakciju sa statičnim i dinamičnim tijelima igre. Iz tog razloga se u skripti za kretanje neprijatelja mijenja brzina kretanja, a ne pozicija.

Na slici se može vidjeti izgled kompletne scene neprijatelja.



Slika 32. Scena neprijatelja

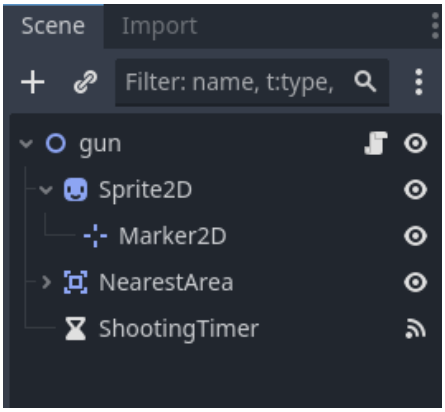
Na baznom nodeu je skripta neprijatelja koja kao i igračeva skripta, ima varijable koje su reference na nodeove poput modula za kretanje neprijatelja kako bi drugi elementi lakše pristupili tim podacima.

Osim tih varijabli sadrži i potrebne funkcije za primanje štete, što napravi kada ostane bez životnih bodova i drugo.

6.2.7. Modul oružja igrača

Kao što je viđeno na sceni igrača, on sadrži pušku na sebi koja je napravljena kao posebna scena. Ona traži najbližeg neprijatelja od igrača, cilja prema njemu i zatim puca lasere.

Kako bi ono moglo pronaći najbližeg neprijatelja, treba imati node „Area2D“ koji je preko cijelog ekrana. Time će detektirati sve neprijatelje u području, pronaći najbližeg i ciljati prema njemu te pucati. Uz node „Sprite2D“ treba imati i „Maker2D“ koji definira iz koje pozicije puca. Ta pozicija će biti dijete „Sprite2D“ jer će se on okretati prema neprijatelju. Još mu je i potreban „Timer“ node koji definira brzinu pucanja.



Slika 33. Scena puške

Od varijabli sadrži reference na bitne nodeove kao „Marker2D“, varijable za vrijednosti poput količine štete lasera i pomoćne varijable.

```
@export var gun_visual:Sprite2D
@export var grab_area:Area2D
@export var shooting_marker:Marker2D
@export var laser:PackedScene
@export var laser_movement_speed:int
@export var laser_damage:int
var current_enemy
var can_shoot = false
```

Na samom početku definira veličinu prostora detekcije tako da bude jednaka veličini ekrana. Time bez obzira na rezoluciju, uvijek cilja na najbližeg neprijatelja na ekranu.

```
func _ready():
    grab_area.get_child(0).shape.size = get_viewport().size
```

Zatim dohvaća sve elemente u polju za detekciju. Ako ih nema, prestaje daljnje izvođenje. Ako ih ima onda računa udaljenosti svakog neprijatelja, pamti najmanju udaljenost i neprijatelja s najmanjom udaljenosti. Nakon svih izračuna, postavlja smjer gledanja na pronađenog neprijatelja.

```
func _process(_delta):
    var available_items = grab_area.get_overlapping_bodies()
    if !available_items:
        can_shoot = false
        return
    can_shoot = true
    var min_distance
    for enemy in available_items:
        var distance =
enemy.global_position.distance_to(global_position)
        if min_distance == null || distance < min_distance:
            min_distance = distance
            current_enemy = enemy
    gun_visual.look_at(current_enemy.position)
```

I naravno, kada istekne vrijeme za pucanje, ako može pucati instancira laser, definira njegovu poziciju, smjer, brzinu i štetu te ga pridodaje sceni.

```
func _on_shooting_timer_timeout():
   if can_shoot:
         var instance = laser.instantiate()
         var direction = (global_position - current_enemy.global_position).normalized()
         instance.global_position = shooting_marker.global_position
         instance.direction = direction
         instance.movement_speed = -laser_movement_speed
         instance.damage = laser_damage
         get_parent().add_sibling(instance)
```

6.2.8. Modul za iskustvo (eng. *Experience*)

Modul za iskustvo služi kako bi prilikom prelaska na višu razinu igrač mogao dobiti nasumično unaprjeđenje (objašnjeno u kasnijem poglavljiu).

Ovaj modul koristi uzorak dizajna Singleton, što znači da je kao i modul za bodove samo skripta dodana u autoload projekta i svi joj elementi mogu pristupiti.

Osim što prati trenutne životne bodove, na početku ih resetira i ima funkciju koja ih povećava. Kada prijeđe preko granice za višu razinu, povećava granicu sljedeće razine, resetira trenutno iskustvo i dodaje ako postoji količina iskustva preko granice prošle razine. Na kraju u funkciji signalizira da se dogodilo unaprjeđenje kako bi bilo koji element mogao pratiti unaprjeđenja igrača i po potrebi odradio određene funkcije.

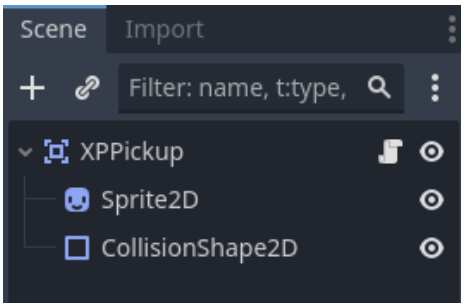
```
var amount : int
var next_level_up = 10
signal xp_changed
signal level_up

func _ready():
   amount = 0

func add(add_amount:int):
   amount += add_amount
   xp_changed.emit()
   if amount >= next_level_up:
      var over_xp = amount - next_level_up
      next_level_up += next_level_up * 0.2
      level_up.emit()
      reset()
      add(over_xp)
```

```
func reset():
    amount = 0
    xp_changed.emit()
```

6.2.9.Objekt za iskustvo

U ovom tipu igre, neprijatelji na smrti ostave objekt iskustva koje igrač može pokupiti i s vremenom prijeći na višu razinu.

Objekt je vrlo jednostavan, ima bazni node „Area2D“, pripadajuću koliziju, „Sprite2D“ node i skriptu na baznom nodeu koja kad detektira da je objekt ušao u područje skupljanja koje se nalazi na igraču, kreće se kretati prema igraču. Kada se nalazi jako blizu igrača, onda povećava količinu iskustva globalne skripte i uništava se.



Slika 34. Scena objekta iskustva

6.2.10. Modul za prikaz iskustva u korisničkom sučelju

Igrač treba znati koliko iskustva ima, zbog čega će se oni prikazati u korisničkom sučelju kao traka koja će se ispunjavati kako se iskustvo skuplja.

Za te svrhe će se kreirati nova scena sa baznim nodeom „ProgressBar“, definirat će se da se nalazi na sredini gornjeg djela ekrana i neka visina da bude vidljiva.

Zatim će se dodati skripta na njega koja na početku definira maksimalnu vrijednost onom koja je potrebna igraču za višu razinu iz modula za iskustvo. Onda definira širinu tako da bude jednaka širini ekrana i spaja svoje 2 funkcije na signale modula za iskustvo.

```
func _ready():
    max_value = Experience.next_level_up
    size.x = get_viewport().size.x
    Experience.connect("xp_changed", _on_xp_changed)
    Experience.connect("level_up", _on_level_up)
func _on_xp_changed():
    value = Experience.amount
```

```
func _on_level_up():
    max_value = Experience.next_level_up
```

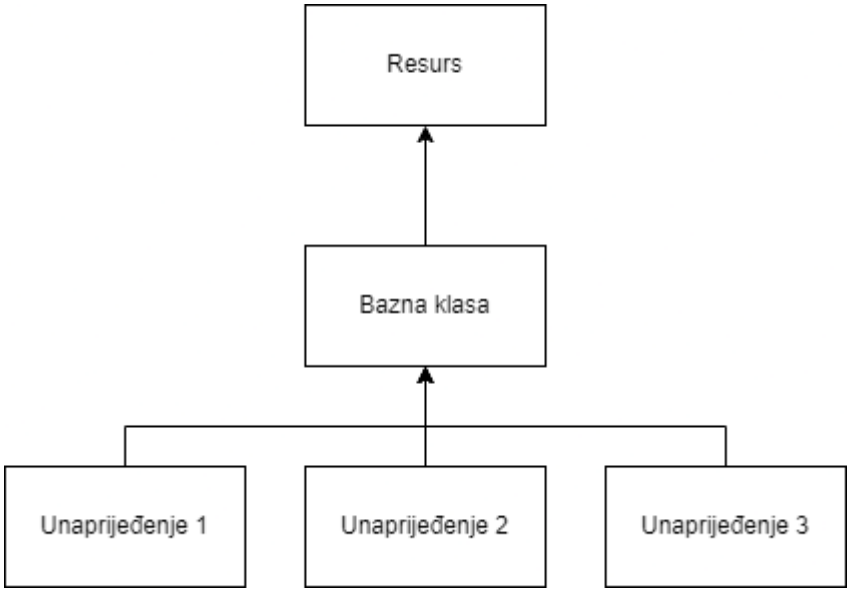
6.2.11. Sustav za unaprjeđenja

Za sustav unaprjeđenja će se iskoristiti uzorak dizajna Strategy.

Strategy je uzorak dizajna koji omogućuje definiranje obitelji algoritama, odvajanje svakog u vlastitu klasu, i kreiranje njihovih objekata međusobno izmjenjivima [5].

Za ovaj slučaj znači da definiramo baznu klasu za sva unaprjeđenja koja definira zajedničke funkcije i varijable. Zatim će se kreirati klase za svako unaprjeđenje posebno koje će naslijediti baznu klasu i implementirati funkcije kako žele te dodati varijable ako im trebaju. Time pri pozivanju unaprjeđenja, definiramo da trebamo tip bazne klase i uvijek se poziva ista funkcija, a sama instanca koja se primjenjuje je objekt neke od naslijeđenih klasa.

Za jasniji prikaz kako je sustav zamišljen je napravljen sljedeći dijagram koji prikazuje opisano. Korisničko sučelje će sadržavati polje unaprjeđenja koje su tipa bazne klase, a mogu biti bilo koje naslijeđene klase (svojstvo nasljeđivanja).



Slika 35. Primjer implementacije sustava unaprijeđenja pomoću uzorka dizajna strategije

Bazna klasa naslijediti resurs. Resurs u Godotu je tip podataka koji se pohranjuje na disku u JSON, CSV ili bilo kojem drugom obliku. Resursi se rade iz definirane klase, jedna klasa može imati više resursa sa drugačijim vrijednostima. Također ako se svi resursi za unaprjeđenja spremaju na jedno mjestom, lako je izmjenjivati vrijednosti bez traženja gdje je u projektu to definirano.

Kod bazne klase će definirati varijable za ikonu unaprjeđenja i za opis unaprjeđenja, te funkciju za pridruživanje unaprjeđenja.

```
extends Resource
class_name BaseUpgrade
@export var texture:Texture2D
@export var description:String

func on_equip(_player):
   pass
```

Nakon definiranja bazne klase, može se kreirati prva skripta unaprjeđenja. Kao primjer kreirat će se unaprjeđenje brzine igrača. Ona će naslijediti baznu klasu, dodati varijablu za količinu promjene i u funkciji će povećati brzinu igrača za navedenu količinu.

```
extends BaseUpgrade
class_name MovementSpeedUpgrade
@export var amount:int

func on_equip(player):
   player.movement.movement_speed_bonus += amount
```

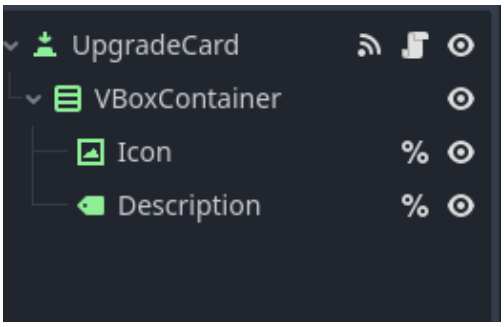
Prvo unaprjeđenje je definirano. Sada je preostalo napraviti resurs od kreirane skripte. To je radi jednostavno desnim klikom na prozoru sa datotekama, zatim opcija „New“ i odabrati „Resource“. U skočnom prozoru treba upisati ime resursa koji je u skripti nakon „class_name“ i dvoklikom na kreirani resurs se u inspektoru on otvorio i mogu se definirati varijable. Za sada će se definirati privremeni opis, dati generična ikona i postaviti vrijednost.

Sada kada je definiran proces kreiranje unaprjeđenja, lako se mogu kreirati dodatna unaprjeđenja bez izvođenje nečega drugačijeg za svako novo unaprijeđenije.

6.2.12. Modul kartica unaprjeđenja u korisničkom sučelju

Ideja je da igrač skuplja iskustvo i pri prelasku na višu razinu dobiva odabir između 3 različite kartice unaprjeđenja.

Kreirat će se nova scena sa baznim nodeon tipa „Button“, pridružit čemu se „VBoxContainer“ kako bi ikona i opis bili jedan ispod drugog. Zatim će se tom nodeu dodati



Slika 36. Scena modula kartice unaprjeđenja

„TextureRect“ i „Label“ node za ikonu unaprjeđenja i opis. Ta 2 nodea su definirana kao unikatni i promijenilo se njihovo ime kako bi se lakše pristupilo njima kroz kod.

Baznom nodeu je dodana skripta koja ima varijablu resursa unaprjeđenja, „_ready()“ funkciju koja postavlja vrijednosti ikone i opisa, te kod primjene unaprjeđenja na pritisak gumba nakon čega šalje signal da je unaprjeđenje odabrano kako bi se mogao odabir maknuti s ekrana nakon odabira.

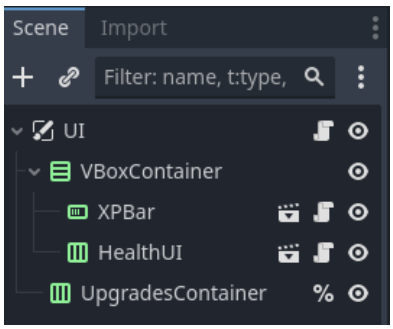
```
var upgrade_resource:BaseUpgrade
signal upgrade_chosen
func _ready():
    %Icon.texture = upgrade_resource.texture
    %Description.text = upgrade_resource.description
func _on_pressed():
    upgrade_resource.on_equip(get_tree().get_first_node_in_group("p
layer"))
    upgrade_chosen.emit()
```

6.2.13. Korisničko sučelje

Korisničko sučelje, kao i u prošloj igri, ima bazni node „CanvasLayer“. On u sebi sadrži „HBoxContainer“ node, modul za unaprijeđenja u korisničkom sučelju, modul za prikaz iskustva u korisničkom sučelju i modul za prikaz životnih bodova u korisničkom sučelju koje je isto kao i u prošloj igri.

Bazni node ima pridruženu skriptu koja upravlja prikazom unaprjeđenja. Sadrži polje mogućih unaprjeđenja i scenu kartice unaprjeđenja.

Zatim se na početku spaja na signal povećanja razine modula za iskustvo. Prilikom povećanja igračeve razine nasumično bira 3 različita unaprjeđenja, kreira novi objekt kartice,



Slika 37. Scena korisničkog sučelja druge igre

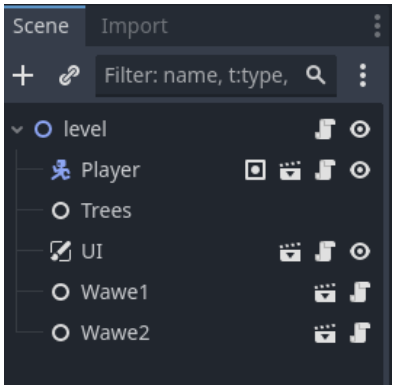
pridruži joj resurs te daje instancu „HBoxContainer“ nodeu koji ih uredno raspoređuje na ekranu. Nakon prikaza kartica zaustavlja igru koja se nastavlja prilikom odabira unaprjeđenja.

6.2.14. Nivo (eng. *Level*)

Scena u kojoj će se odvijati igra treba imati igrača, korisničko sučelje, konfigurirane spawnere koji kreiraju neprijatelje i drveća da nije prazan prostor.

Drveća će se kreirati na nasumičnoj poziciji između -5000 i 5000 X i Y osi. Igrač i neprijatelji mogu zapeti za drveća i igrač ne može pucati kroz drvo. Stoga se drvo sastoji od baznog nodea „StaticBody2D“ i „Sprite2D“ nodea kao njegovo dijete. Ako se drvo kreira na igraču, jednostavno će se maknuti iz scene.

Skripta za kreiranje drveća se nalazi na samom baznom nodeu nivoa koji je „Node2D“.



Slika 38. Scena nivoa druge igre

6.3. Dodavanje vizuala, zvukova, glazbe, dodavanje modula u Starter projekt

Zadnja stvar koja je preostala je uljepšati igru, kreirati nove neprijatelje i definirati dodatna unaprjeđenja. Unaprjeđenja ne moraju samo mijenjati razne vrijednosti (poput brzine kretanja ili količine štete lasera), već se može i kodirati nova funkcionalnost koja se može dodati igraču poput duplih metaka.

Neki moduli su promijenjeni u odnosu na to kakvi su u Starter projektu. U ovim slučajevima treba odlučiti da li je promjena poboljšala modul i treba li zamijeniti postojeći modul u Starter projektu ili se samo dodati kao alternativa.

Odlučio sam da će se modul za kretanje lasera izmijeniti s novim jer je fleksibilniji. Modul za pojavu neprijatelja se kopirati, ali će postojati i prošla verzija i drugi moduli koji su izmjenjeni za ovu igru se neće mijenjati u Starter projektu.

Novi moduli koji će se dodati u projekt jesu: modul za iskustvo, modul za prikaz iskustva u korisničkom sučelju, bazna skripta sustava za unaprjeđenja i modul kartica unaprjeđenja zajedno sa svojom skriptom. Sustav na unaprjeđenje je jako koristan i kako sada postoje „temelji“ sustava, jednostavno se on može implementirati i u prošloj igri.

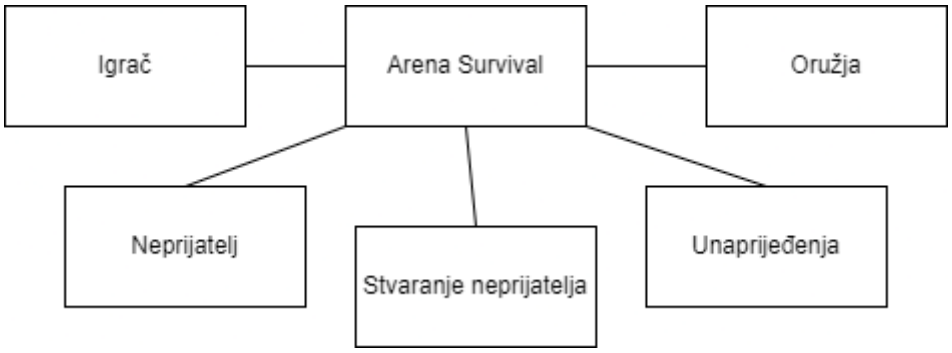
7. Treća igra: Arena survival

Kao i na početku prošle igre, kopirat će se Starter projekt, preimenovati se mapa i uvesti u Godot.

U ovoj igri će igrač na početku moći odabrati početno oružje, zatim će preživljavati valove neprijatelja i nakon svakog vala će moći odabrati unaprjeđenje po želji. Dio elemenata već postoji, ali dio toga se treba i prenamijeniti kako bi funkcioniralo za ovaj tip igre. Također će se razviti nekoliko novih modula.

7.1. Planiranje

Elementi igre će biti slični prošloj, postojat će igrač, neprijatelji, iskustvo, pojava neprijatelja i unaprjeđenja. Novi element igre će biti oružja.



Slika 39. Prva iteracija elemenata treće igre

Mehanike su većinom iste/slične, no biti će potrebno modificirati pojavu neprijatelja što bi trebalo biti jednostavno. Element oružja će biti razvije kao vlastiti sustav, odnosno modul.

Slika 40. Druga iteracija treće igre

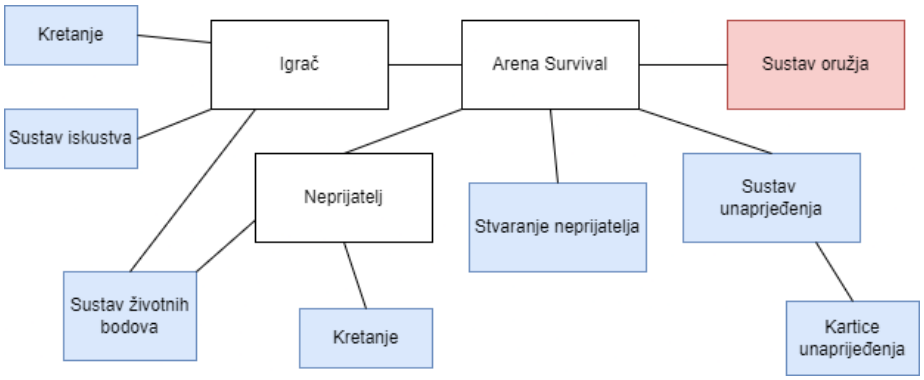
I za kraj će se spojiti svi isti elementi i označiti moduli koji se trebaju razviti/primijeniti.

7.2. Razvoj

7.2.1. Sustav oružja

U ovoj igri će igrač na početku moći odabrati početno oružje. Zatim će se krenuti pojavljivati neprijatelji sa vlastitim nasumičnim oružjima koja će igrač moći dobiti na početku igre ako porazi cijeli val neprijatelja.

Slično kao i u prošloj igri, napraviti će se skripta iz koje će se kreirati resursi za oružja.

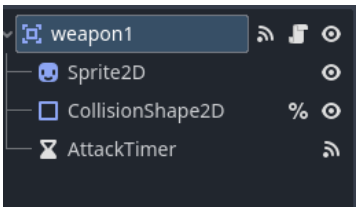


Slika 41. Treća iteracija treće igre

U ovom slučaju će sva oružja isto funkcionirati, pa će se napraviti samo jedna skripta sa potrebnim varijablama i iz nje više različitih resursa od kojih će svako predstavljati drugo oružje.

```
extends Resource

class_name BaseWeapon

@export var texture:Texture2D
@export var description:String
@export var is_melee:bool
@export var damage:int

signal weapon_picked_up
```

Zatim se kreira scena kojoj će spawner dati odgovarajući resurs.



Slika 42. Scena oružja

Na samom početku oružje učitava sve vrijednosti iz resursa i pomiče sprite kako bi se pravilno rotirao. Zaustavlja vlastiti proces zbog toga što se želi da igrač na početku pokupi oružje i tek onda krene ono odrađivati što treba.

```
func _ready():
    sprite.texture = weapon_resource.texture
    damage = weapon_resource.damage
    description = weapon_resource.description
    pickup_collision.shape.radius = sprite.texture.get_width() / 2.0
    sprite.offset.x = sprite.texture.get_width() / 2.0
    pickup_collision.position = sprite.offset

    if !weapon_resource.is_melee:
        shooting_position = Marker2D.new()
        shooting_position.position =
Vector2(sprite.texture.get_width(),0)
        add_child(shooting_position)

    set_process(false)
```

Pošto igrač i neprijatelj mogu imati isto oružje, oružju se prosljedi koga ono treba ciljati (igrača ili neprijatelja u ovom slučaju). Ako treba ciljati igrača prosljedi se objekti igrača, inače će ciljati na poziciju miša jer igrač cilja prema mišu.

```
func _process(_delta):
    if target_object is String:
        target_position = get_global_mouse_position()
        look_at(get_global_mouse_position())
    else:
        look_at(target_object.position)
        target_position = target_object.position
```

Ovo je najosnovnija logika koja se treba proširiti sa puštanjem pravilne animacije i zvuka tijekom napada.

Ima logiku za skupljanje koje će biti aktivno jedino na početna oružja koja igrač treba pokupiti.

```
func _on_body_entered(_body):
    weapon_resource.weapon_picked_up.emit(self)
    pickup_collision.set_deferred("disabled", true)
    disconnect("body_entered", _on_body_entered)
```

Nakon toga je opisana funkcija napada s daljine koja stvara laser. Funkcija je iz razloga što će se aktivirati kada igrač klikne tipku napada. Instanciranje lasera je isto kao i u drugim igrama sa minimalnim promjenama oko logike. Sam laser je isti kao što je i do sada.

```
func attack():
    if target_position != null:
        can_attack = false
        attack_timer.start()
        var instance = laser.instantiate()
        instance.global_position = shooting_position.global_position
        instance.movement_speed = laser_movement_speed
        instance.direction = (target_position - global_position).normalized()
        instance.collision_mask = collision_mask
        instance.damage = damage
        get_parent().add_sibling(instance)
```

I na kraju kad istekne vrijeme koje broji čekanje između napada, igrač ponovno može napasti.

7.2.2.Modificiran modul za pojavu neprijatelja (eng. *spawner*)

I u ovoj igri je primijenjen modul za pojavu neprijatelja. Osim postavljanje željene brzine neprijatelja, ideja je definiranje s kojim oružjem bi se neprijatelj mogao pojaviti. Radi toga je napravljeno polje oružja koje neprijatelj može imati. Zatim nova stvar je i pozicija od kuda se neprijatelji stvaraju. Ideja je da postoji nekoliko točaka iz kojih se neprijatelj može stvoriti i cijeli al neprijatelja se stvara iz jedne od danih točaka.

Kako se neprijatelji nebi stvarali jedan na drugome, napravljeno je polje koje detektira kada neprijatelj odlazi iz polja i kada ode, stvori se novi i tako sve dok se ne dođe do željene količine neprijatelja.

```
@export var duration : int
```

```
@export var enemy_amount:Vector2i
@export var enemy_movement_speed : int
@export var enemy : PackedScene
@export var enemy_spawn_points: Array[Marker2D]
@export var enemy_weapon_pool : Array[BaseWeapon]
@export var next_wawe:SpawnerV3
@export var duration_timer:Timer
@export var detection_area:Area2D

var number_of_enemies
```

Na početku se postavlja vrijeme trajanja vala, odabire nasumični broj neprijatelja i nasumična točka s koje će se stvarati.

```
func _ready():
    duration_timer.wait_time = duration
    number_of_enemies   =   Random.getRandomIntNumber(enemy_amount.x,
enemy_amount.y)
    setRandomSpawnerPosition()

func setRandomSpawnerPosition():
    var spawn_point = Random.getRandomElement(enemy_spawn_points)
    detection_area.global_position = spawn_point.global_position
```

Pri započinjanju vala, odabire se oružje neprijatelja i krenu se stvarati. Zbog čekanja na izlaska neprijatelja iz polja (kako se nebi stvarali jedan u drugome) potrebno je funkcije pozivati na drugačiji način.

```
func start_wawe():
    duration_timer.start()
    var enemy_weapon = Random.getRandomElement(enemy_weapon_pool)
    for i in range(0, number_of_enemies):
        var instance = enemy.instantiate()

        instance.weapon_resource = enemy_weapon
        instance.movement_speed = enemy_movement_speed
        instance.global_position = detection_area.global_position
        if i == 0:
            get_tree().current_scene.call_deferred("add_child",
instance)
```

```
                else:
                        await detection_area.body_exited
                        get_tree().current_scene.call_deferred("add_child",
instance)
```

I naravno pri završetku vala, pokreće se sljedeći val neprijatelja.

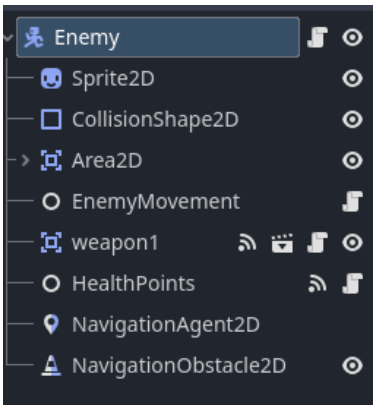
7.2.3.Modul neprijatelja

Neprijatelj, iako ima isti modul za kretanje i upravaljanja životnim bodovima, funkcionira malo drugačije. Specifičan je za ovu igru te se zato neće objašnjavati kod, ali na razini igre će se konstantno ponovno iskorištavati jer će svi neprijatelji biti isti sa drugačijim oružjima.

Neprijatelj ima dodane nodeove za navigaciju. To znači da se kreće najkraćim putem do igrača zaobilazeći prepreke. Prepreke jesu on sam, kako bi svi stajali u liniji tijekom pucanja primjerice, umjesto da su svi jedni na drugima.

Osim toga, ako imaju oružje na daljinu, stane na udaljenosti od igrača i ako mu se igrač krene približavati, pokuša „pobjeći“ od njega. Isto tako ako igrač krene bježati, prati ga dok cilja na njega. U slučaju da drži oružje na blizinu, onda pokušava bez obzira na sve doći što bliže igraču kako bi ga pokušao ozljediti.

Time svime neprijatelj dobiva jednostavnu, ali dublju logiku nego li prošle igre koja se mogla razviti zahvaljući manjem razmišljanju oko stvari koje već postoje kod drugih igara.



Slika 43. Scena neprijatelja treće igre

7.2.4.Menađer početnog oružja

Ovo nije modul, već objekt specifičan za ovu igru. On je prazan node koji odabire koja će se oružja stvoriti i na kojim (označenim) pozicijama i prilikom uzimanja jednog od oružja, makne ostala i pokreće val neprijatelja.

7.2.5. Sustav otkrivenih oružja

Razvijen je i sustav za otkrivena oružja. On funkcionira tako da se definiraju sva oružja koja se mogu dobiti i ima popis otkrivenih i neotkrivenih oružja. Kada igrač pobjedi val neprijatelja čije oružje ne zna, onda se ono dodaje u popis otkrivenih oružja i zapiše se taj popis u datoteku.

Skripta ovog sustava je stavljena pod automatsko učitavanje kako bi joj mogle pristupiti druge skripte koje javljaju otkrivanje novog oružja i spremanje podataka.

```
var all_weapons:Array[BaseWeapon]
var discovered_weapons:Array[BaseWeapon]
var dir_name = "res://Resources/Weapons/"

func _ready():
    var dir = DirAccess.open(dir_name)
    dir.list_dir_begin()
    var file = dir.get_next()
    while file != "":
        if !dir.current_is_dir():
            all_weapons.append(load(dir_name + file))
        file = dir.get_next()

func save_data():
    var discovered_weapon_paths = []
    for weapon in discovered_weapons:
        discovered_weapon_paths.append(weapon.resource_path)

    var JSON = JSON.stringify(discovered_weapon_paths)
    var save_file = FileAccess.open("user://discovered_weapons.save", FileAccess.WRITE)
    save_file.store_string(JSON)
    save_file.close()

func load_data():
    if not FileAccess.file_exists("user://discovered_weapons.save"):
        return

    discovered_weapons = []
    var save_file = FileAccess.open("user://discovered_weapons.save", FileAccess.READ)
```

```
var JSON_string = save_file.get_line()
var JSON = JSON.new()
JSON.parse(JSON_string)
var data = JSON.get_data()

for d in data:
	var resource = load(d)
	discovered_weapons.append(resource)
```

7.3. Dodavanje vizuala, zvukova, glazbe, dodavanje modula u Starter projekt

Preostalo je kreirati nova oružja sa vlastitim izgledom i uljepšati cijelu igru.

Novo za Starter projekt može biti pojavljivanje neprijatelja uz promjenu njegovog naziva jer funkcionira dosta drugačije nego li u prošlim igrama.

Sustav oružja se može dodati u Starter projekt, ali bi ga trebalo još doraditi kako bi svako oružje moglo imati svoj efekt, čime bi cijeli sustav trebalo proširiti na način kako i unaprijeđenja funkcioniraju.

Mnogo toga se ponovno iskoristilo za ovu igru zbog čega se dobila mogućnost fokusa na sustav oružja i dodatno razviti sustav otkrivanja oružja. Osim toga se dobilo dodatno vrijeme za dovršavanje igre.

8. Zaključak

Tijekom razvoja igara, bio je veliki stres na količinu posla, pogotovo za prvu igru.

Razvoj prve igre je potrajao oko 2 tjedna, a velika količina vremena je utrošena na razvoj svih mehanika i testiranje svega napisanog. Nakon toga je utrošeno nešto vremena na dovršavanje same igre kako bi bila bolje prezentirana i gotovo. Dosta vremena je utrošeno na njen ravoj, ali u retrospektivi dobro utrošeno vrijeme s obzirom na razvoj drugih igara.

Razvoj druge igre je bio mnogo glađi, jer je mnogo toga već postojalo u Starter prijektu od prošle igre. Zbog toga sam se fokusirao više na razvoj novih sustava (unaprijeđenja) i specifičnih stvari za igru. Vrijeme razvoja svih funkcionalnih dijelova je trajao samo 3 dana, a cijele igre nešto više od tjedan dana. Već se ovdje vidi direktan utjecaj razvoj ponovno iskoristivih komponenti.

Razvoj treće igre je bio još brži, jer je razvoj funkcionalnih dijelova trajao samo 2 dana. Nakon toga su se dovršavala igra i sveukupno je trajalo tjedan dana.

Svime time se može vrlo lako zaključiti da je razvoj ponovno iskoristivih komponenti vrlo koristan, iako dulje traje. Potrebno je više vremena da bi se kreirali svi sustavi i moduli kako bi bili ponovno iskoristivi, potrebno je i utrošiti dio vremena na planiranje kako će pojedini modul izgledati, ali rezultat koji se dobije se itekako isplati. Brzina kojom se mogu razviti dodatni dijelovi igre pomoću modula je nevjerojatna. Također se prilikom razvoja druge igre može lakše fokusirati na nove module i mehanike igre umjesto ponovno pisati i razvijati dijelove koji su praktički identični prošlim igrama.

9. Popis literature

[1] “Godot Docs – 4.3 branch — Godot Engine (stable) documentation in English.” Accessed: July 18, 2024. [Online]. Available: https://docs.godotengine.org/en/stable/

[2] 262588213843476, “Summary of ‘Clean code’ by Robert C. Martin,” Gist. Accessed: July 19, 2024 [Online]. Available: https://gist.github.com/wojteklu/73c6914cc446146b8b533c0988cf8d29

[3] “Nasljeđivanje (programiranje),” *Wikipedia*. May 30, 2021. Accessed: July 19, 2024 [Online]. Available: https://bs.wikipedia.org/w/index.php?title=Naslje%C4%91ivanje_(programiranje)&oldid=3319288

[4] “Composition vs Inheritance | DigitalOcean.” Accessed: July 19, 2024. [Online]. Available: https://www.digitalocean.com/community/tutorials/composition-vs-inheritance

[5] “Design Patterns.” Accessed: July 20, 2024. [Online]. Available: https://refactoring.guru/design-patterns

[6] "Reusable Code" Accessed: July 20, 2024. [Online]. Avaliable: https://homes.cs.aau.dk/~bt/GameProgrammingE09/ReusableCode.pdf

10. Popis slika