Универзитет у Београду

Електротехнички факултет



Имплементација пуцачке 2D видео игре у

Godot развојном оквиру

Дипломски рад

|  |  |
| --- | --- |
| Ментор: | Кандидат: |
| Проф. др Марко Мишић | Влада Бељански 2017/415 |

Београд, септембар 2024.

Садржај

[Садржај 2](#_Toc177739092)

[1. Увод 3](#_Toc177739093)

[2. Опис проблема 4](#_Toc177739094)

[2.1. Преглед жанра 4](#_Toc177739095)

[2.2. Технички изазови и функционалне спецификације 4](#_Toc177739096)

[3. Опис коришћених технологија 6](#_Toc177739097)

[3.1. *Godot engine* 6](#_Toc177739098)

[3.1.1. Чворови и сцене 6](#_Toc177739099)

[3.1.2. Node2D и наслеђивање 6](#_Toc177739100)

[3.1.3. Сигнали 6](#_Toc177739101)

[3.1.4. GDScript 7](#_Toc177739102)

[3.2. Развојно окружење 7](#_Toc177739103)

[3.3. Контрола верзија 7](#_Toc177739104)

[4. Општа подешавања 8](#_Toc177739105)

[5. Имплементација *HealthBox* и *DamageBox* система 10](#_Toc177739106)

[5.1. HealthBox 10](#_Toc177739107)

[5.2. DamageBox 10](#_Toc177739108)

[6. Имплементација пројектила 12](#_Toc177739109)

[6.1. ProjectileBase 12](#_Toc177739110)

[6.2. ProjectileEnemySine 13](#_Toc177739111)

[6.3. ProjectileEnemyTarget 14](#_Toc177739112)

[7. Имплементација пуцања код непријатеља 15](#_Toc177739113)

[8. Имплементација непријатеља 16](#_Toc177739114)

[8.1. EnemyBase 16](#_Toc177739115)

[8.2. EnemyWide 17](#_Toc177739116)

[8.3. EnemyKamikaze 17](#_Toc177739117)

[9. Генерисање непријатеља 19](#_Toc177739118)

[9.1. SpawnerBase 19](#_Toc177739119)

[9.2. SpawnerPath 20](#_Toc177739120)

[9.2.1. EnemyPathFollow2D 21](#_Toc177739121)

[9.3. SpawnerPosition 21](#_Toc177739122)

[10. Имплементација играча 22](#_Toc177739123)

[10.1. Имплементација кретања 22](#_Toc177739124)

[10.2. Имплементација пуцања 23](#_Toc177739125)

[10.3. Имплементација живота 24](#_Toc177739126)

1. Увод

За видео игре се може рећи да су најмлађи уметнички медијум. Самим тим су медијум који још увек није довољно истражен. Популарност видео игара је огромна и то је тренд који се наставља, на овај вид забаве, или уметности, опредељује се све већи број људи у свету. Са обзиром на то да је њихов утицај огроман важно је истражити и разумети их, а да би се то урадило потребно је пронаћи најбоља техничка решења њиховог стварања. Ово је важно јер се идеја најбоље изражава само уз разумевање медијума који се користи, уз сва његова ограничења и предности.

Овај рад се бави имплементацијом пуцачке 2D видео игре у *Godot* развојном оквиру. Циљ овог документа је да прикаже комплетан развој ове видео игре, од описа жанра и почетних захтева, преко избора технологије, до саме реализације и изазова који су се јавили током реализације.

У почетку, рад пружа преглед самог жанра и сагледава постојећа решења, односно игара које припадају описаном жанру. Затим, рад уводи читаоца у основне концепте и алате коришћене током развоја игре у Godot оквиру. У наставку се детаљно описују специфичности имплеметације, односно са конкретна решења проблема који су се јавили током развоја овог типа игре користећи изабрану технологију.

Имплементација игре обухвата креирање играча, непријатеља, пројектила и образаца кретања, као и интеракције између тих елемената. Поред тога, имплементиран је и стандардни мени игре. Систем је дизајниран тако да омогућава лако додавање нових функционалности, као што су различити типови непријатеља и пројектила, уз модуларни приступ који подржава лаке измене.

Завшни делови рада посвећени су приказу коначне функционалности игре кроз детаљно корисничко упутство које објашњава рад система. На крају, закључак нуди кратку рекапитулацију целокупног процеса развоја, са освртом на потенцијалне надоградње и могућности унапређења система.

1. Опис проблема

У овом поглављу биће дат преглед жанра, техничких изазова, као и кратка функционална спецификација.

* 1. Преглед жанра

Игре из жанра *vertically scrolling shoot ‘em up*, односно пуцачке игре са вертикалним скроловањем, карактеришу непријатељи који се, углавном, појављују на врху екрана и крећу се према доле, док играч контролише кретање свог лика у четири смера. Главни циљ играча је да избегава непријатељске пројектиле, уништи што већи број непријатеља испаљујући своје пројектиле и преживи до краја нивоа.

Овај тип игре обично садржи таласе непријатеља са различитим обрасцима кретања и различитим врстама пројектила. Пројектили и интеракције између играча и непријатеља чине кључну механику ове игре. Игра може да укључује и систем бодовања који награђује играча за ефикасност у уништењу непријатеља.

Главне карактеристике жанра укључују:

* Генерисање таласа непријатеља са различитим обрасцима кретања.
* Кретање играча у сва четири смера и испаљивање пројектила како би се уништили непријатељи.
* Фокус на избегавању непријатељских пројектила и награђивање играча кроз ефикасно уништавање непријатеља.
  1. Примери игара које припадају жанру

Игре које ће овде бити описане су из различитих временских периода и разликују се по неким уникатним механикама, међутим све садрже основне одлике жанра.

* + 1. Raiden

*Raiden* је класична вертикално скролујућа аркадна игра из 1990. године. Игра је доживела огроман успех и портована је за личне рачунаре и кућне конзоле тог времена. Направљено је неколико наставака.

Игру је одликовала велика тежина која се повећавала сваким следећим нивоом којих има осам. Када би се играла у два играча постојала је посебна механика која је омогућавала напајање специјалних пројектила пуцањем у летелицу другог играча. Када би се прешли свих осам нивоа игра би почињала испочетка али са повећаном тежином.

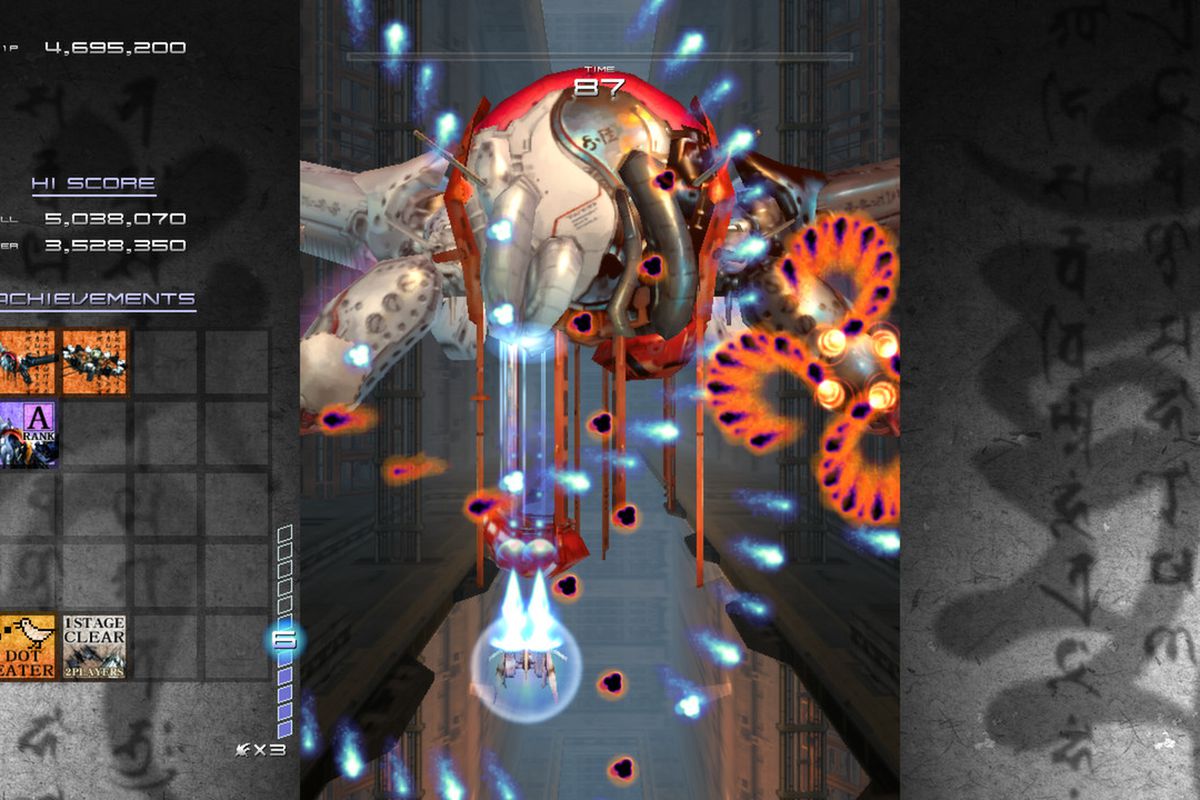


Слика 2.2.1.1. Приказ сцене из игре *Raiden*

* + 1. Ikaruga

Ова игра је оригинално била аркадна и изашла је 2001. године. Због своје уникатне механике игра је у почетку била непопуларна, играчи су очекивали стандарну пуцачку игру, међутим добили су нешто што није захтевало толико брзе рефлексе и имало је елементе *puzzle* игара. Касније су критичари ову игру сврставали у најбоље игре свих времена.

Главна механика ове игре је прилично оригинална, наиме играч може да мења своје стање тако што бира један од два поларитета, црни или бели. Сви непријатељи и пројектили су такође или црни или бели. Играч апсорбује пројектиле своје тренутне боје и чува њихову снагу за активацију специјалног напада. Пошто играчеви пројектили са њим такође мењају поларитет, ако непријатеље погађају пројектили супротног поларитета тада они примају двоструку штету.



Слика 2.2.2.1. Приказ сцене из игре *Ikaruga*

* + 1. ZeroRanger

*ZeroRanger* је игра која је изашла 2018. године за *Mirosoft Windows*. Примарно је пуцачка игра са вертикалним скроловањем, међутим постоје и сегменти хоризонталним скроловањем. Игра је брзо постала веома цењена, садржи бројне референце на старије игре овог жанра и бројне мета моменте. Има будистичку тематику и естетику.

Игра је врло приступачна за људе који се нису сусретали са жанром, тежина расте постепено док играч учи концепте и механике. Може се изабрати једна од две летелице, после сваког већег сегмента играч бира један од два нова понуђена оружја. Касније се добија могућност трансформације механичко одело, на енглеском *mech*. Специфичан је начин прогресије, после сваког покушаја бодови играча се сакупљају и после довољног броја откључавају играчу један додатни покушај од места где је изгубио. Оваквих додатних покушаја се може сакупити девет и то важи до краја игрице.



Слика 2.2.3.1. Приказ сцене из игре *ZeroRanger*

* 1. Технички изазови и функционалне спецификације

Током развоја игре кључни изазов је био направити систем који је модуларан, проширив и лако подесив, чиме се омогућава лако додавање нових непријатеља, пројектила или чак механика, уз једноставно итеративно тестирање нивоа или нових елемената.

Био је важан дизајн флексибилног система који омогућава креирање различитих типова непријатеља са специфичним обрасцима кретања. Систем је морао подржавати лако подешавање времена појављивања и путања непријатеља.

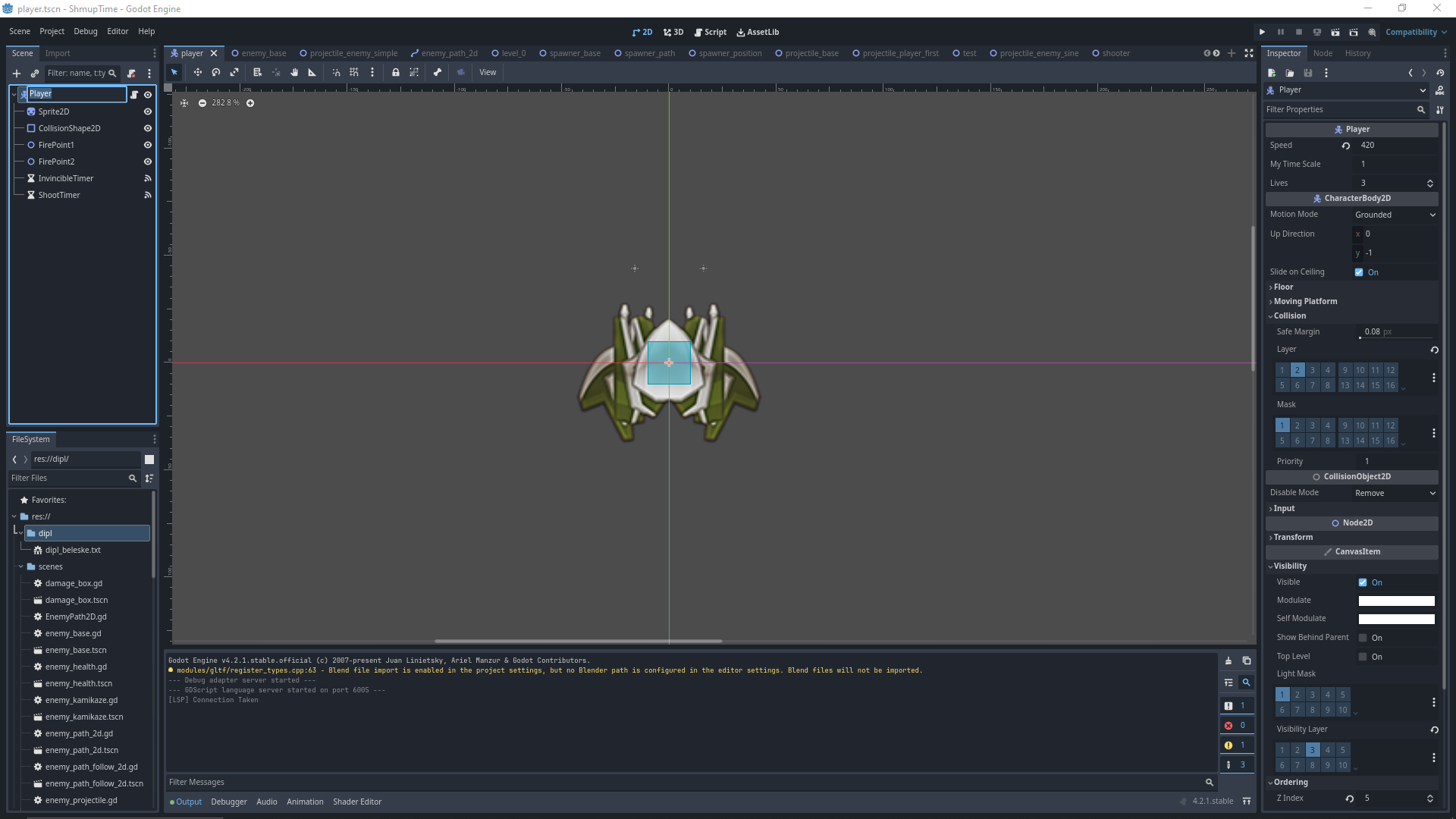
Поред тога, модуларност система је омогућила лако додавање непријатеља, пројектила и механика, при чему сваки елемент функционише независно, али у јасној интеракцији са другим елементима. Овај приступ, осим што је олакшао развој, омогућио је правилно функционисање интеракција и механика након што су тестиране, јер је једном тестирани систем функционисао исправно у свим деловима игре, без потребе за поновним тестирањем сваки пут када се додају нови елементи.

1. Опис коришћених технологија

Током развоја ове 2D пуцачке игре, главна платформа био је *Godot Engine* који је пружио потребне алате за имплементацију механика игре. Осим тога коришћени су и додатни алати за подршку развоја.

* 1. *Godot engine*

*Godot engine* био је централна технологија у развоју ове игре. То је *open-source* развојни окрвир који подржава развој 2D и 3D игара. У оквиру овог одељка детаљније ће бити објашњени кључни елементи који су коришћени у развоју игре.



Слика 3.1.1. Приказ окружења за *Godot Engine*

* + 1. Чворови и сцене

У оквиру *Godot* основни градивни блокови су чворови( *nodes* ). Чворови су вишенаменски објекти који могу да обављају разне функције, као што су контрола кретања, анимација, обрада уноса итд. Сви чворови организовани су у сцене( *scenes* ), при чему свака сцена може да садржи више чворова или чак друге сцене као под-сцене. Сцена је такође наследник одређеног чвора у објектно оријентисаном смислу, што омогућава хијерархијску структуру и поновну употребу кода у систему.

* + 1. Node2D и наслеђивање

Већина чворова у 2D игри наслеђује чвор ***Node2D***, који обезбеђује основне функције као што су позиционирање, ротација и скалирање у 2D простору. Такође омогућава контролу редоследа приказивања при рендеровању.

Чворови који наслеђују ***Node2D*** обухватају различите типове елемената као што су ***Sprite2D***, који се користи за приказивање слика и спрајтова, и ***CharacterBody2D***, који омогућава контролу физике и колизије објеката у игри. Ова структура наслеђивања омогућава велику флексибилност у дизајну игара, јер сваки нови елемент може наследити постојеће функционалности и прилагодити их према потребама игре.

* + 1. Сигнали

**Сигнал** је механизам који омогућава објектима да комуницирају међусобно на ефикасан и разматљив начин. Сигнали раде по принципу емитовања и повезивања, омогућавајући да један објекат обавести друге објекте о неким догађајима без директне зависности или позивања њихових функција. Ово помаже да се смањи повезаност између објеката и омогућава већу флексибилност у дизајну.

* + 1. GDScript

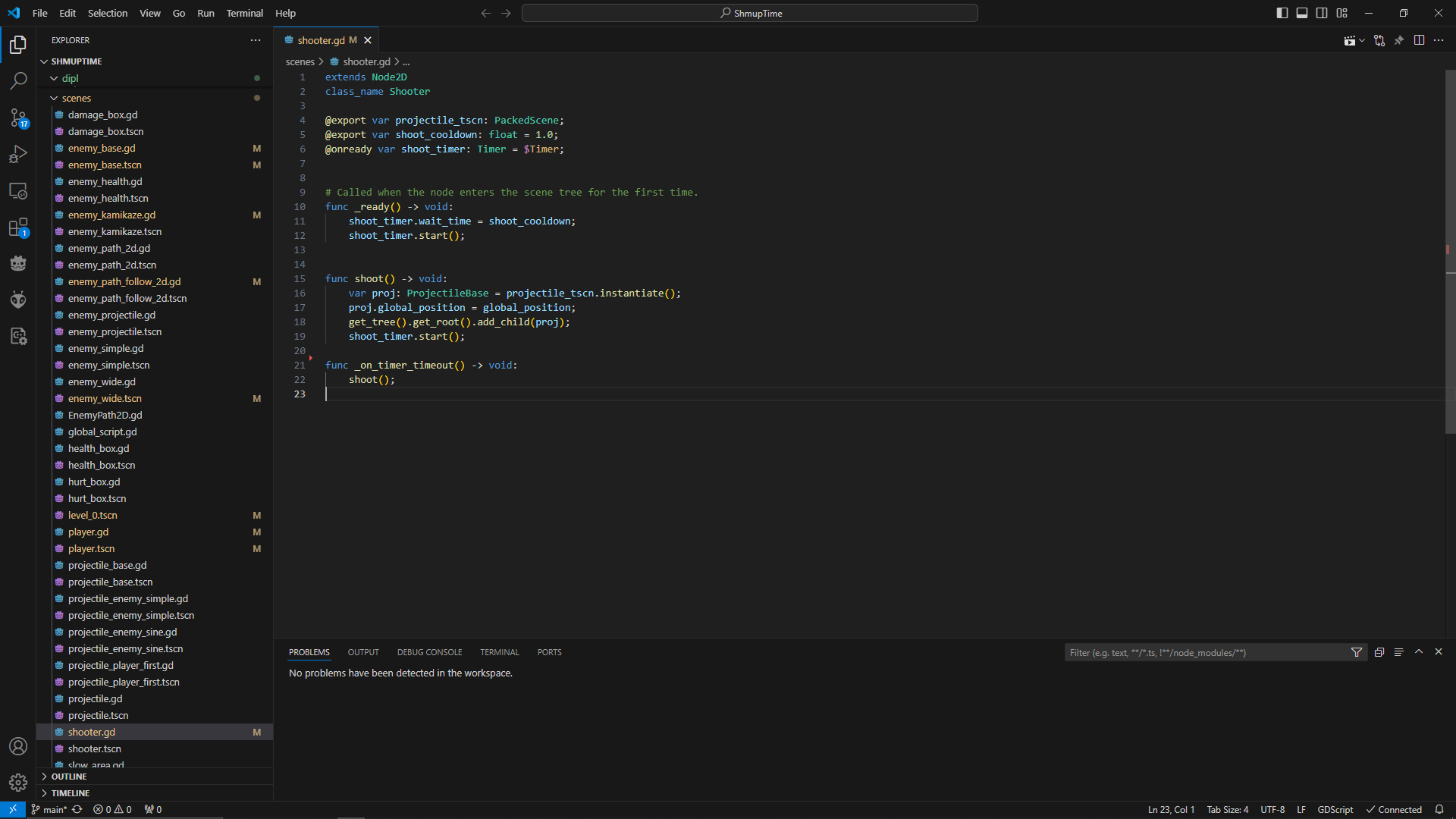
***GDScript*** је програмски језик високог нивоа, објектно оријентисан, императиван и постепено типизиран, направљен за *Godot*. Користи синтаксу засновану на увлачењу, сличну језицима као што је *Python*. Његов циљ је да буде оптимизован за и тесно интегрисан са *Godot Engine*-ом, омогућавајући велику флексибилност при креирању садржаја и интеграцији.

Током развоја ове игре, ***GDScript*** је коришћен за програмирање логике кретања играча, управљање непријатељима, интеракције са окружењем, као и за имплементацију корисничког интерфејса и других елемената игре. Због своје брзине и флексибилности, GDScript је омогућио брзе итерације и тестирање нових функција.

Скрипте, тј. фајлови у којима је написан код, могу постојати самостално или се могу доделити чворовима или сценама и тако управљати њиховим ресурсима. Такође у скрипти је могуће навести да је нека променљива **експортована** помоћу кључне речи @export што омогућава њено директно подешавање из самог едитора.

* 1. Развојно окружење

Током развоја игре, коришћен је ***Godot*-ов уграђени едитор** за већину послова, укључујући креирање сцена, управљање чворовима и подешавање пројекта. За писање скрипти, коришћено је интегрисано развојно окружење (IDE) **Visual Studio Code**, које је омогућило лако писање *GDScript* кода. Поред тога, коришћени су и плугини за синтаксну проверу како би се осигурало да је код чист и без грешака.



Слика 3.2.1. Приказ окружења за *VS Code*

* 1. Контрола верзија

За управљање верзијама на пројекту коришћен је **Git** заједно са **GitHub** платформом, уз коришћење **GitHub Desktop** алата. Овај систем за контролу верзија омогућио је да се сви кораци у развоју прате и да се врати на претходне верзије у случају потребе. **GitHub Desktop** је поједноставио процес контроле верзија, а **GitHub** платформа је омогућила чување резервних копија пројекта.

1. Општа подешавања

Током развоја, коришћена је *autoload* скрипта названа ***GlobalScript***, која служи за складиштење глобалних података и управљање функционалностима које се користе у различитим деловима игре. Ово омогућава да важни елементи буду доступни свим сценама без потребе за експлицитним прослеђивањем референци. *Autoload* скрипте се понашају као *singleton* објекти.

**Колизијске маске и слојеви** су важан аспект управљања интеракцијама између објеката у игри.

* **Колизијски слојеви** дефинишу на којим слојевима се објекат налази, односно где тај објекат може бити детектован од других објеката.
* **Колизијске маске** одређују са којим другим слојевима објекат може ступити у интеракцију. На пример, објекат може бити на слоју "ENEMIES", али његова колизијска маска може бити подешена тако да реагује само на "PLAYER" и "WALLS", игноришући друге непријатеље.

У овом пројекту, колизијске маске су подешене на следећи начин:

* WALLS – слој 1, за све елементе који представљају препреке у игри.
* **PLAYER** – слој 2, за играча.
* ENEMIES – слој 3, за непријатеље.
* HEALTHBOX – слој 4, за објекте који се односе на здравље.

Ова подела омогућава јасну организацију интеракција између различитих елемената игре.

Улазни подаци (inputs) су такође конфигурисани тако да омогуће интуитивно управљање:

* Кретање је подешено на **WASD** тастере.
* Пуцање је додељено тастеру **K**.

Сва подешавања су статички типизирана, што значи да се типови података морају дефинисати унапред. Ово не само да омогућава бољу контролу и прецизну проверу грешака током развоја, већ и убрзава извршавање кода, јер компајлер зна који типови података се користе и може их оптимизовати.

Подешавања дисплеја су следећа:

* Резолуција: **1920x1080** пиксела.
* Режим: **Windowed**.
* Stretch mode: **Canvas\_items**.
* Stretch aspect: **Expand**.

1. Имплементација *HealthBox* и *DamageBox* система

Циљ је био направити модуларни спрегнути ситем који симулира прављење штете и смањивање животних поена ентитета. У оваквој имплементацији се лако могу додати функционалности одређеним ентитетима тако што се у хијерархију њихове сцене само дода ***HealthBox***, ако је циљ да ентитет има животне поене, односно дода се ***DamageBox***, ако је циљ да ентитет или област ствара штету.

* 1. HealthBox

Ова сцена наслеђује чвор ***Area2D***, што значи да представља регион 2D простора који је дефинисан децом чворовима који су типа ***CollisionShape2D***, такође има могућност да детектује када други објекти типа ***CollisionShape2D*** улазе или излазе из тог региона.

Садржи две променљиве, max\_health и curr\_health, прва представља максималне, а друга тренутне животне поене. Прва променљива је **експортована** да би се олакшало дизајнирање ентитета који садрже ову сцену. Има дефинисан **сигнал** који се зове health\_deplted. Поседује функцију за смањивање животних поена и када ти животни поени дођу до нуле или испод нуле емитује се наведени сигнал.

* 1. DamageBox

Ова сцена такође наслеђује чвор ***Area2D***. Садржи две експортоване променљиве, прву, should\_hurt\_player, која назначава да ли се штета прави играчу или непријатељима, и другу, damage, која садржи вредност штете у виду реалног броја.

У функцији \_ready(), коју садржи сваки чвор и која се позива када се чвор и његова деца учитају у стабло сцена, на основу променљиве should\_hurt\_player подешавају се вредности колизионих маска саме сцене ***DamageBox***.

func \_ready() -> void:

    if should\_hurt\_player:

        set\_collision\_mask\_value(2, 1);

        set\_collision\_mask\_value(4, 0);

    else:

        set\_collision\_mask\_value(4, 1);

        set\_collision\_mask\_value(2, 0);

***Area2D*** садржи већ уграђене **сигнале** за детекцију других ***Area2D*** и ***PhysicsBody2D***, па ћемо лако повезати повратне функције \_on\_area\_entered и \_on\_body\_entered са та два сигнала помоћу едитора.



Слика 5.2.1. Повезивани сигнали са функцијама у едитору

Прва функција служиће за детекцију објеката типа ***HealthBox***, а друга за детекију самог играча, јер играч заправо наслеђује ***CharacterBody2D***. У обе функције позива се одговарајућа функција самог објекта који је детектован за умањивање животних поена. У обе се такође емитује **сигнал** по имену damage\_collided који касније могу користити родитељи ове сцене.

func \_on\_body\_entered(body:Node2D) -> void:

    if body is Player:

        var player: Player = body as Player;

        player.get\_hit();

    damage\_collided.emit();

func \_on\_area\_entered(area:Area2D) -> void:

    if area is HealthBox:

        var health\_box: HealthBox = area as HealthBox;

        health\_box.reduce\_health(damage);

    damage\_collided.emit();

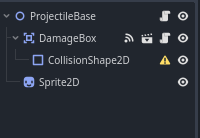
1. Имплементација пројектила

У игри, пројектили су кључни елементи који омогућавају играчу и непријатељима да наносе штету другим објектима. Систем пројектила је дизајниран тако да буде флексибилан и лако проширив, омогућавајући различите типове пројектила са специфичним брзинама, путањама и количинама штете. За омогућавање проширивости изабран је полиморфизам објектно оријентисане парадигме.

* 1. ProjectileBase

Сцена ***ProjectileBase*** наслеђује чвор ***Node2D***. Као децу садржи ***DamageBox***, који садржи ***CollisionShape2D***, и садржи ***Sprite2D***. У скрипти има дефинисане променљиве speed, која је типа float, direction, која је типа Vector2 и референцу на свој ***DamageBox*** подназивом damage\_box. Референца се дохвата помоћу кључне речи ***@ready*** испред имена променљиве, која указује да ће променљива бити иницијализована тек након што је чвор учитан у сцену, и оператора ***$*** након којег је наведено име чвора који припада сцени. То у коду изгледа овако:

@onready var damage\_box: DamageBox = $DamageBox;



Слика 6.1.1. Хијерархија чворова за сцену *ProjectileBase*

Функција \_physics\_proces прима параметар delta, који предствља време протекло од прошлог извршавања фунцкије и наслеђена је из Node2D. Ова функција се извршава у тачно дефинисаним интервалима од шездесет пута у секунди и користи се за физичке симулације. У овој сцени наведена функција је надјачана. У њој се само позива функција move\_projectile којој прослеђује параметар delta и коју наследници ове сцене треба да надјачају. Функција move\_projectile рачуна кретање пројектила на једноставан начин тако што на позицију додаје производ брзине, нормализованог вектора кретања и временског интервала. То се ради у случају да пројектил нема превише специфичан начин кретања, у том случају довољно је подесити му вектор и брзину.

func move\_projectile(delta: float) -> void:

    position += direction.normalized() \* speed \* delta;

У иницијализацији сцене, дакле у функцији \_ready, функција \_on\_damage\_collided се повезује на **сигнал** damage\_collided којег емитује damage\_box. У тој функцији позива се queue\_free која уклања сцену из стабла сцена, односно уклања објекат из игре.

При дизајнирању пројектила потребно је урадити следеће:

* Наследити сцену ***ProjectileBase*** и уклонити скрипту која је закачена, то се ради јер ће бити закачена скрипта родитељске сцене.
* Потребно је закачити нову скрипту и написати да она наслеђује ***ProjectileBase***
* Изабрати колозиони облик и подесити му величину.
* Убацити одговарајући визуелни спрајт.
* У општем случају подесити вектор и брзину кретања
* У зависности од тога да ли пројектил припада играчу или непријатљу потребно је подесити damage\_box на начин који је раније описан.
* По потреби додати специфичну логику у скрипту.

У наставку се могу видети неколико примера пројектила.

* 1. ProjectileEnemySine

Поред уобичајеног кретања помоћу вектора кретања и брзине, овај пројектил осцилира, односно има додатну компоненту у кретању која је нормална на његов вектор кретања, и та компонента брзине осцилира синусоидално.

func move\_projectile(delta: float) -> void:

    position += direction.normalized() \* speed \* delta;

    time\_passed += delta;

    var perpendicular\_direction: Vector2 = Vector2(-direction.y, direction.x);

    var cosine\_offset: float = amplitude \* cos(time\_passed \* frequency \* PI \* 2) \* frequency;

    position += perpendicular\_direction.normalized() \* cosine\_offset;

* 1. ProjectileEnemyTarget

Овај пројектил дохвата позицију играча и поставља свој вектор кретања ка њему.

1. Имплементација пуцања код непријатеља

Пуцање је генерализовано помоћу сцене која се зове ***Shooter***. Ова сцена наслеђује чвор ***Node2D***. У себи садржи чвор који је типа ***Timer***. Кључна променљива је projectile\_scene, којa је типа ***PackedScene***, тај тип представља апстракцију серијализоване сцене и може му се доделити сцена као вредност. Садржи функцију shoot() која је одговорна за логику пуцања.Од променљивих такође садржи shoot\_cooldown, која представља интервал између два пуцања и референцу на ***Timer*** под називом shoot\_timer који емитује **сигнал** на који је потписана функција \_on\_timer\_timeout().

Када чвор уђе у сцену, у функцији \_ready се подешава време чекања тајмера и он се стартује. Након истека тајмера, сигнал \_on\_timer\_timeout покреће функцију \_on\_timer\_timeout која позива shoot, која инстанцира нови пројектил типа ***ProjectileBase*** и поставља га на позицију пуцача. Након тога, тајмер се поново покреће, чиме се омогућава континуирано пуцање у интервалима.

func shoot() -> void:

    var proj: ProjectileBase = projectile\_tscn.instantiate();

    proj.global\_position = global\_position;

    get\_tree().get\_root().add\_child(proj);

    shoot\_timer.start();

func \_on\_timer\_timeout() -> void:

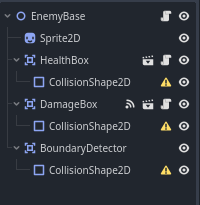
    shoot();

1. Имплементација непријатеља

Сви непријатељи биће изведени из сцене ***EnemyBase***. Ово ће олакшати брзо прављење различитих непријатеља.

* 1. EnemyBase

Ова сцена као своје чворове садржи ***Sprite2D***, ***HealthBox*** и ***DamageBox***. Сцена у скрипти садржи променљиву speed, која представља брзину самог непријатеља и дефинише **сигнал** enemy\_died.



Слика 8.1.1. Хијерархија чворова за сцену *EnemyBase*

***DamageBox*** служи да нанесе штету играчу при додиру. Такође је заслужан за брисање непријатеља који изађу ван оквира нивоа и ударе у баријере чије су колизиони слојеви подешени на *WALLS*.

***HealthBox*** има већ познату улогу. Сцена у функцији \_ready повезује своју функцију death са **сигналом** health\_deplted помоћу референце health\_box.

func \_ready() -> void:

    enemy\_health.health\_deplted.connect(death);

У функцији за смрт сцена емитује свој **сигнал** под називом *enemy\_died*, који касније могу користити сцене које садрже непријатеље, и позива queue\_free како би се сцена уклонила.

func death() -> void:

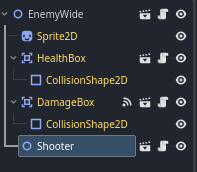
    enemy\_died.emit();

    queue\_free();

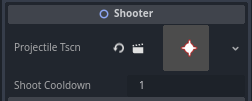
Сада ћемо видети неколико типова непријатеља.

* 1. EnemyWide

Овај непријтељ има могућност пуцања, тако да му је додата подсцена типа ***Shooter***. Тој подсцени су подешени потребни параметри и изабран је пројектил који ће се испаљивати. На слици 8.2.1 жутом бојом су обележени наслеђени чворови и сцене.



Слика 8.2.1. Хијерархија чворова за сцену EnemyWide



Слика 8.2.2. Подешавања за подсцену Shooter

Потребно је и подесити број животних поена, колозионе облике тамо где се они траже и саму брзину непријатеља. Као што се да приметити, ни у основној сцени ни у овој не користи се брзина нити је дефинисано кретање. То је урађено зато што ће се одређени тип непријатеља кретати само по предефинисаним путањама. О томе ће бити речи у следећем поглављу.

* 1. EnemyKamikaze

Наведени непријатељ нема могућност пуцања, већ је његов циљ да физички удари играча и тако му нанесе штету. Он зато има дефинисан начин кретања по некој логици. Наиме, он прво дохвати позицију играча из глобалне скрипте и искористи је да дефинише свој вектор кретања. Такође, када направи штету мора да се уклони из сцене, тиме симулира самоуништење.

extends EnemyBase

class\_name EnemyKamikaze

var target\_direction: Vector2;

@onready var damage\_box: DamageBox = $DamageBox;

func \_ready() -> void:

    super.\_ready();

    damage\_box.damage\_collided.connect(death);

    var target: Vector2 = GlobalScript.player.position;

    target\_direction = (target - position).normalized();

func \_physics\_process(delta: float) -> void:

    position += target\_direction \* speed \* delta;

1. Генерисање непријатеља

Непријатеље је потребно лако генерисати, при чему мора бити дефинисана локације и време њиховог стварања. Такође мора се створити путања за непријатеље који немају своју способност кретања.

* 1. SpawnerBase

Ово ће бити основна сцена за стварање непријатеља. Сцена наслеђује чвор ***Node2D*** и садржи један ***Timer*** који ће се звати ***SpawnTimer***. На **сигнал** timeout наведеног тајмера повезана је функција \_on\_spawn\_timer\_timeout. Сцена садржи следеће променљиве:

* enemies\_to\_spawn – број непријатеља који ће се створити
* enemy\_tscn – сцена која представља тип непријатеља, типа ***PackedScene***
* spawn\_cooldown – време између генерисања два непријатеља
* spawn\_start\_time – време које је потребно да прође од учитавања саме сцене до почетка стварања непријатеља
* spawn\_timer – референца на већ поменути чвор ***SpawnTimer***

Функција spawn у себи садржи само наредбу pass, та наредба не ради ништа, поента је да наведену функцијиу изведене сцене надјачају. У функцији \_ready тајмеру се постави време чекања помоћу променљиве spawn\_start\_time и тајмер се покрене.

func \_ready() -> void:

    spawn\_timer.wait\_time = spawn\_start\_time;

    spawn\_timer.start();

Кључна функција ове сцене је \_on\_spawn\_timer\_timeout. У њој се позива функција spawn и обезбеђује се да је то урађено за подешени број непријатеља, такође се обезбеђује поновно покретање тајмера, овог пута се чека онолико времена колико је записано у променљивој spawn\_cooldown.

func \_on\_spawn\_timer\_timeout() -> void:

    if enemies\_to\_spawn <= 0:

        return;

    spawn();

    enemies\_to\_spawn -= 1;

    if enemies\_to\_spawn > 0:

        spawn\_timer.wait\_time = spawn\_cooldown;

        spawn\_timer.start();

* 1. SpawnerPath

Ова сцена наслеђује SpawnerBase сцену. Када се инстанцира, на пример у нивоу, потревно је као сина има чвор Path2D. Path2D садржи Curve2D, односно путању коју могу да прате инстанце чвора PathFollow2D, или сцене које наслђују поменути чвор. Curve2D описује Безјеову криву у 2D простору која се лако може конструисати у едитору. PathFollow2D мора бити дете чвора Path2D и служи за кретање по датој кривој помоћу променљивих као што су progress или progress\_ratio. У наставку је дат комплетан код ове сцене:

extends SpawnerBase

@onready var path: Path2D = $Path2D;

@export var enemy\_follow\_tscn: PackedScene;

func \_ready() -> void:

    super.\_ready();

    if path == null:

        return;

func spawn() -> void:

    var new\_enemy: EnemyBase = enemy\_tscn.instantiate();

    var enemy\_path\_follow: EnemyPathFollow2D = enemy\_follow\_tscn.instantiate();

    enemy\_path\_follow.enemy = new\_enemy;

    enemy\_path\_follow.add\_child(new\_enemy);

    path.add\_child(enemy\_path\_follow);

Као што је и описано, у хијерархији чворова дете ове сцене мора бити PathFollow2D, а дете тог чвора ће бити непријатељ.

* + 1. EnemyPathFollow2D

Ова сцена наслеђује PathFollow2D и служи да се управља кретањем непријатеља по задатој путањи. Садржи промељиву која је референца на непријатеља и помера га помоћу уграђених функција set\_progress и get\_progress. У случају да непријатељ умре и емитује сигнал о својој смрти овај чвор се такође уклања. Следи сегмент кода за логоки померања:

func \_physics\_process(delta: float) -> void:

    set\_progress(get\_progress() + enemy.speed \* delta);

    if get\_progress\_ratio() == 1:

        queue\_free();

* 1. SpawnerPosition

За овај генератор непријатеља се очекује да када се инстанцира има подчворове који су типа Marker2D. У питању је веома једноставн чвор, исти је као Node2D само је јасније обележен у едитору оквира Godot. Дохвата све маркере и ставља их у низ, заправо ће генерисати онолико непријатеља колко има маркера. Код је прилично једноставан и дат је у наставку:

extends SpawnerBase

var marker\_nodes: Array[Marker2D];

var enemy\_index: int = 0;

func \_ready() -> void:

    super.\_ready();

    for child in get\_children():

        if child is Marker2D:

            marker\_nodes.append(child)

    enemies\_to\_spawn = marker\_nodes.size();

func spawn() -> void:

    var curr\_marker: Marker2D = marker\_nodes[marker\_nodes.size() - enemies\_to\_spawn];

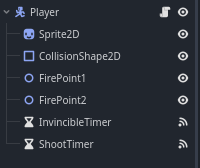
    var new\_enemy: EnemyBase = enemy\_tscn.instantiate();

    new\_enemy.position = curr\_marker.position;

    get\_tree().get\_root().add\_child(new\_enemy);

1. Имплементација играча

Сцена ***Player*** наслеђује чвор ***CharacterBody2D***. ***CharacterBody2D*** је специјализован чвор за физичка тела која контролише корисник. На њу не утиче физика, али она утиче на друга физичка тела. Обично се користи да пружи напредни *API* за померање објеката са детекцијом и резолуцијом колозије са зидовима и нагибима помоћу функције move\_and\_slide, поред тога поседује и генералну детекцију колизије која се постиже функцијом move\_and\_collide. Пошто је сам играч јединствен имплементација се није фокусирала на модуларност, иако играч поседује сличне елементе као неријатељи ипак се превише разликује од њих. Следи изглед стабла чворова за играча:



Слика 10.1. Хијерархија чворова за сцену Player

* 1. Имплементација кретања

Играч има променљиву speed која је типа float. Кретање се одвија у наслеђеној функцији \_physics\_process. На основу притиснутих дугмића тастатуре изгради се вектор по имену move\_vector. Он се даље користи заједно са брзином играча и веменским интервалом delta за израчунавање и додљивање вредности наслеђеној променљивој velocity, која је типа ***Vector2*** и наслеђена из ***CharacterBody2D***. Када се подеси наведена променљива довољно је само позвати функцију move\_and\_slidе. Та функција помера играча користећи velocity и регулише колизије.

func \_physics\_process(delta: float) -> void:

    var direction\_x: float = 0;

    var direction\_y: float = 0;

    var move\_vector: Vector2;

    if Input.is\_action\_pressed("move\_right"):

        direction\_x += 1;

    if Input.is\_action\_pressed("move\_left"):

        direction\_x -= 1;

    if Input.is\_action\_pressed("move\_down"):

        direction\_y += 1;

    if Input.is\_action\_pressed("move\_up"):

        direction\_y -= 1;

    move\_vector = Vector2(direction\_x, direction\_y);

    move\_vector = move\_vector.normalized();

    velocity = move\_vector \* speed \* my\_time\_scale;

    move\_and\_slide()

* 1. Имплементација пуцања

***Player*** садржи два чвора ***FirePoint1*** и ***FirePoint2*** који су типа ***Node2D*** и служе за лакше подешавање места генерисања пројектила. ***Player*** такође поседује ***ShootTimer*** који служи за одређивање интервала између два стварања пројектила. Пошто је кориснику омогућено да може да држи дугме за пуцање притиснуто константно ако жели уведена је променљива која се зове can\_shoot која говори да ли тренутно може да се пуца, ова променљива се користи у тандему са поменутим тајмером. ***Player*** има и референцу на сцену пројектила којег треба да испали.

Користи се наслеђена функција \_process која је јако слична функцији \_physics\_process, прима параметар delta, односно време протекло од прошлог циклуса, наслеђена је из чвора ***Node2D*** и извршава се периодично. Разлика је у томе што нема фиксну периоду већ зависи од рендеровања, стога се ова функција не користи за прецизна физичка израчунавања већ за све остале ствари.

У сваком фрејму се постави питање да ли је притиснуто дугме за пуцање и да ли играч тренутно сме да пуца. У случају да су оба услова тачна позива се функција shoot.

func \_process(delta: float) -> void:

    if Input.is\_action\_pressed("shoot") && can\_shoot:

        shoot();

Услов can\_shoot се поставља на тачну вредност у функцији \_on\_shoot\_timer\_timeout која је повезана на **сигнал** timeout тајмера ***ShootTimer***, на нетачну вредност се поставља после пуцања у самој функцији shoot.

func \_on\_shoot\_timer\_timeout() -> void:

    can\_shoot = true;

Генерисање пројектила је слично ономе у сцени ***Shooter***. Инстанцира се пројектила на основу сцене која је додељена играчу, постави му се позиција на основу референце на ***FirePoint1*** или ***FirePoint2*** и пројектил се дода у корен стабла чворова.

func shoot() -> void:

    var proj\_1: ProjectileBase = PROJECTILE.instantiate();

    get\_tree().get\_root().add\_child(proj\_1);

    proj\_1.global\_position = firing\_point\_1.global\_position;

    var proj\_2: ProjectileBase = PROJECTILE.instantiate();

    get\_tree().get\_root().add\_child(proj\_2);

    proj\_2.global\_position = firing\_point\_2.global\_position;

    can\_shoot = false;

    shoot\_timer.start();

* 1. Имплементација живота

***Player*** се разликује од непријатеља по томе што нема животне поене, већ одређени број живота. Живот се губи при контакту са инстанцом сцене ***DamageBox***. Из ових разлога играч не поседује сцену ***HealthBox***. Поента је да играч буде доста опрезан и прецизан при избегавању.

Променљиве које се користе у овом механизму су lives, која означава број живота и целобројна је, индикатор is\_invincible који је типа *bool* и који означава да ли је ***Player*** тренутно неповредив, и invincible\_timer, што је референца на ***InvincibilityTimer*** на чији је **сигнал** timeout повезана функција \_on\_invincibility\_timer\_timeout. Такође је дефинисан **сигнал** player\_death.

Функција get\_hit се позива од стране инстанце типа ***DamageBox*** при колизији као што је приказано у приложеном коду у поглављу 5.2. Ако је играч неповредив излази се из функције. У супротном број живота се смањује за један, ако је број живота постао нула емитује се сигнал player\_death. Играч постаје неповредив и активира се тајмер invincible\_timer.

func get\_hit() -> void:

    if is\_invincible:

        return;

    lives = lives - 1;

    if lives == 0:

        player\_death.emit();

    is\_invincible = true;

    invincible\_timer.start();

У функцији \_on\_invincibility\_timer\_timeout променљива is\_invincible се поставља на нетачну вредност.

func \_on\_invincible\_timer\_timeout() -> void:

    is\_invincible = false;