ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

ДИПЛОМНА РАБОТА

Тема: 2D Top-Down RPG на Unity

Дипломант: *Кирил Чингаров*

Научен ръководител: *Гергана Гергьовска*

СОФИЯ
2021

Увод

Историята на индустрията за разработка на видео игри може да бъде проследена до края на 50-те години на миналия век, когато студенти и преподаватели по компютърни науки в университетите започнали да разработват игри с цел симулации. Но индустрията набира популярност през 70-те години, когато се появяват първото поколение домашни конзоли като Atari 2600 и Intellivision, които позволяват на потребителя да играе на своя телевизор у дома.

За начало на златния век на индустрията се счита 1978 година. По това време в Америка са се установили аркадните игри като “Galaga”, “Donkey Kong” и “Space Invaders”. “Space Invaders” успява да промени коренно индустрията, като добавя няколко нови идеи: играта се регулира от система с животи вместо традиционно използвания таймер или резултат, като получаваш допълнителни животи при събирането на определен брой точки. За първи път са въведени таблиците с най-високите резултати, записани в машината. С тези нови подобрения “Space Invaders” става голяма сензация.

Не след дълго на пазара излизат конзоли за дома, които имат достатъчно мощни изчислителни системи, за да позволят появата на игри с 3D графика, което поражда нуждата от така наречените игрови двигатели. Този тип софтуер позволява лесната обработка на графиките на игрите отделно, както и отделната разработка на “game engine” и съдържанието на играта. Така започват да се развиват различни видове жанри от игри - от стратегически до приключенски, от 2D графики до 3D графики, от ролеви игри до игри със стрелба.

С това се стига до целите на тази дипломна работа - да се разработи 2D Top-Down RPG игра на игровия двигател Unity. За разработката ще се

използва игрови двигател, за да се избегне програмиране на ниско ниво и да се фокусират ресурсите върху създаването на съдържанието на играта - играч, противници, различни нива и система за сражение между играча и противниците. Това са основите на RPG (Role-playing game) жанра или така наречените ролеви игри. Играта също трябва да поддържа локална база данни за информация за играта и играча, както и запазване на прогреса.

1. Първа глава - Проучване на различни игрови двигатели и съществуващи решения

1.1. Игрови двигател

Игрови двигател, или “game engine”, е софтуер проектиран и разработен с цел улеснено разработване на видео игри от всякакви жанрове. Така разработчиците успяват да изградят игра от начало до край за мобилни устройства, конзоли и персонални компютри без да навлизат в програмирането на ниско ниво.

История на игровите двигатели

Преди появата на игровите двигатели видео игрите обикновено са били проектирани като една програма, която използва оптимално наличния хардуер. Един от най-големите проблеми тогава е било управлението на паметта. Поради скоростното развиване на технологиите по това време, по-голяма част от кода е невъзможно да се използва за други проекти. Така повечето компании започват да разработват собствени игрови двигатели, които да преизползват за повечето си проекти.

Една от първите игри, направени чрез игрови двигател, е играта “Doom” от “id Software”. За създателя на така наречения игрови двигател “id Tech 1” или “Doom engine” се счита Джон Кармак. Въпреки ограниченията, поставени от двигателя върху изграждането на света на играта (ограничен наклон на стените на стаята), това представлявало революция в света на видео игрите.

По-късно компанията продължава практиката да разработва игровите двигатели отделно от съдържанието на играта (моделите, нивата, историята и др.). Тази практика става стандарт в индустрията и много

компании започват да разработват собствени двигатели. Дори започват да продават игровите двигатели под лиценз, което се доказва като сигурен поток от приходи за компанията, печелейки им от десетки хиляди до милиони долари.

Компоненти на игровия двигател

Модерните игрови двигатели са едни от най-сложните програми и приложения, нуждаещи се от много фино настроени системи, които да си взаимодействат помежду си. В зависимост от нуждите на компаниите, които ги разработват, и вида на игрите, които се създават с тях, игровите двигатели имат много и различни компоненти. Но във всички двигатели можем да намерим следните три компонента - “Rendering engine”, “Audio engine” и “Physics engine”.

“Rendering engine” е двигателя, който се грижи за визуализацията на играта от отделни модели и околна среда до крайната картина, която се появява на екрана на потребителя. Този двигател се концентрира само върху тази задача - рендерирането на сцената. Нужните пресмятания могат да бъдат извършени от централния процесор или от видео картата. “Rendering engine” най-често се изгражда с помощта на няколко приложни програмни интерфейси (по-нататък наричани API), като едни от най-известните и най-използваните са “Direct3D”, “OpenGL“ и “Vulkan”. Тези интерфейси дават софтуерна абстракция на графичния процесор на видео картата. Съществуват и библиотеки от ниско ниво като “DirectX”, които се използват за визуализацията върху екрана на съответната платформа. Те предоставят достъп до компютърната периферия (мишка, клавиатура или контролери), звукови карти, мрежови карти и т.н. Преди навлизането на ускореното рендериране на 3D графики, игровите двигателите са използвали отделен софтуер, който макар и да е произвеждал реалистични картини, отнемал изключително много време.

“Audio engine” е частта от игровия двигателя, която се грижи за звука и музиката на играта. Като минимално изискване към този компонент е да може да зарежда, декомпресира и пуска музика и звуци. По-напредналите звукови двигатели могат да симулират най-различни ефекти в зависимост от това какво се случва в играта. Изчисленията се извършват в централния процесор, но могат и да се извършват в звуковите карти.

“Physics engine” се грижи за симулацията на законите на физиката. Той предоставя съвкупност от функции за симулация на физични сили и сблъсъци между различните обекти в играта по време на изпълнение на програмата.

1.2. Обзор на различните игрови двигатели

С развитието на игралната индустрия на пазара се появяват голям брой игрови двигатели, всеки от които имат своите предимства и недостатъци. Повечето компании продават своите двигатели и си подсигуряват постоянен поток от приходи чрез тези лицензи. Някои от лицензите позволяват на начинаещите разработчици на игри да изпробват различни методи, без да заделят пари за игровите двигатели. По-известните такива двигатели са “Unity”, “Unreal Engine”, “Godot” и “GameMaker Studio”.

“Unity” е създаден от компанията “Unity Technologies” и пусната на пазара през 2005 като игрови двигател, разработен специфично за операционната система Mac OS X. Но до края на 2018 година двигателя е разширен и разработен да работи с повече от 25 платформи. Ограниченията на “Unity” са малки, защото с него могат да се направят 2D и 3D игри, виртуална реалност или добавена реалност, както и симулации и др. Дори компании извън игралната индустрия го използват за филми, архитектура, автомобили и т.н. В момента “Unity” има два вида лицензи -

безплатен и платен. Безплатният лиценз е за лична употреба или за по-малки компании, които не печелят повече от 100 000 долара годишно. Платения лиценз се базира върху месечен абонамент за използването на продукта, като таксата се изчислява на базата на приходите на компанията.

“Unreal Engine”, разработен от компанията “Epic Games”, е показан за първи път в играта “Unreal” от 1998 година. Първоначално създаден специфично за стрелкови игри от първо лице, игровия двигател вече може да се използва за разновидности от игри. Последната му версия “Unreal Engine 4” е пусната на пазара през 2014, а следващата версия “Unreal Engine 5” е планирана за края на 2021 година. Лицензът на този игрови двигател позволява за безплатното му използване, но при печалба от продукта, компанията изисква 5% от приходите.

“Godot” е игрови двигател проектиран за разработката на 2D и 3D игри на различни платформи. Той е безплатен софтуер, който може да бъде променен според нуждите на потребителя, ползвайки MIT лицензa. Разработен от аржентинците Хуан Линиетски и Ариел Манзур, “Godot” може да бъде използван на няколко операционни ситеми като Linux, macOS и Microsoft Windows и може да създава игри за персонални компютри, мобилни устройства и уеб апликации.

“GameMaker Studio” (първоначално “Animo” до 2011 година) са серии от игрови двигатели разработени от Марк Овермарс през 1999 година и след 2007 година разработван от “YoYo Games”. Последната итерация на този игрови двигател е “GameMaker Studio 2” създаден през 2017 година. Поддържа разработването на игри от няколко жанра, като използва визуален или скриптов програмен език GameMaker Language. Първоначално е създаден за начинаещите, но в последните си версии е адаптиран и за по-напредналите потребители.

1.3. Съществуващи реализации

На пазара съществуват сравнително голям брой игри, характеризиращи се с перспективата Top-Down (играта се гледа отгоре). Едни от най-успешните игри на пазара използват тази перспектива и така наречения пиксел арт. Повечето от тях използват специфичен за компанията разработчик игрови двигател, който не е обществено достъпен. Някои от тези игри са сериите “Pokemon”, “The Legend of Zelda” и “Dragon Quest”.

“Pokemon” е Top-Down RPG, която е една от най-популярните серии от игри в света, създаден от Сатоши Таджири през 1995 година. Франчайзът се развива около въображаемите същества наричани покемон, които хората хващат и обучават да се сражават помежду си чрез ходове. Серията от игри е първоначално разработена от компанията “Game Freak” за конзолата на “Nintendo” Game Boy през 1996 година. Скоро след това става огромен хит по цял свят и бързо се развива в други медии. Игрите стават вторите най-продавани по целия свят (назад от “Super Mario” франчайза на “Nintendo”), продавайки над 368 милиона копия и един милиард сваляния на мобилни устройства. На екрана се появяват анимационни серии с над 1 000 епизода в над 169 страни из целия свят.

“The Legend of Zelda” е един от световно известните франчайзи на “Nintendo”, с които се прославят из цял свят. Създадена от японските дизайнери на игри Шигеру Миямото и Такаши Тезука, играта за първи път излиза на пазара на 21 февруари 1986 година. Играта може да се разглежда като комбинация от приключенския жанр и екшън ролева игра. Франчайза се фокусира върху различните превъплъщения на Link (младо момче, готово да се жертва в името на добрата кауза), Zelda (принцеса, в която се прераждa богинята Hylia) и Ganon (краля на демоните, решен да покори

света). “Nintendo” продължава да създава нови игри от тези серии и вече има 19 игри за всичките конзоли направени от Nintendo през годините.

“Dragon Quest” (“Dragon Warrior” в Северна Америка до 2005 година) е японска ролева игра създадена от Юджи Хорий. Играта се публикува от компанията “Square Enix” (преди “Enix”) за конзолите Nintendo DS и Nintendo 3DS, а компанията “Nintendo” - за конзолата Nintendo Switch. Първата игра от серията е публикувана през 1986 година и към момента има 11 главни игри и няколко отделни игри. Тази игра има един от най-големите приноси към жанра на ролевите игри. Главната идея на играта е да си герой, който спасява света от злото. Обикновено този герой е придружен от група компаниони, готови да му помогнат. Това е играта, която за първи път въвежда система за сражение на база на ходове (подобно на шах и други подобни игри) и случайните срещи с противниците.

2. Втора глава - избор на технологии за разработка и структура на кода и базата данни

2.1. Избор на технология за разработка

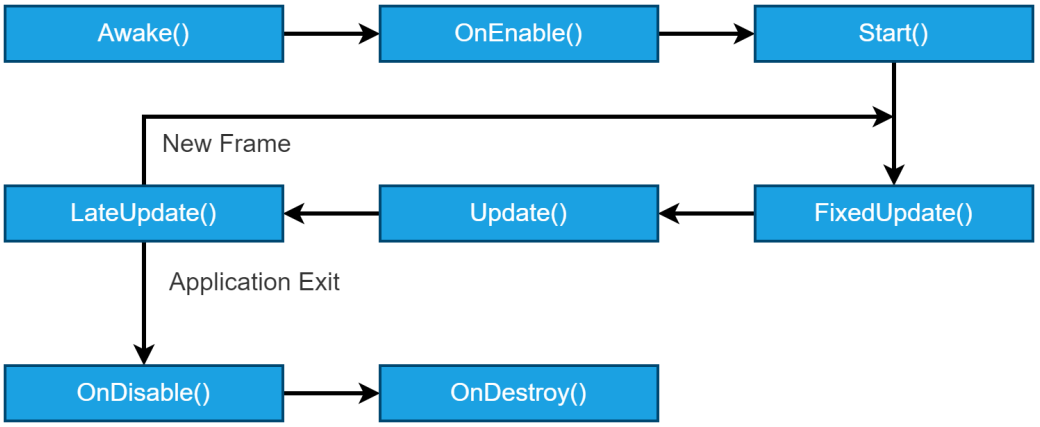
След направения оглед какво е игрови двигател и на някои от по-известните такива, достъпни до нас, за разработка на този дипломен проект ще се използва двигателя Unity. Въпреки по-слабите си възможности за графична мощ спрямо конкурента си Unreal Engine 4, Unity предлага възможност за създаване на игри за многобройно платформи. От гледна точка на двуизмерната перспектива Unity предлага възможност за използване на собствени спрайтове (изображения, анимации и т.н.) както и рендер на двуизмерния свят. При нужда от спрайтове и в случай, че разработчика иска да се фокусира изцяло върху функционалността на играта, Unity предлага решение на проблема - Unity Asset Store. Този магазин предлага най-различни елементи за една игра - от 2D изображения и 3D модели до различни инструменти за ускорена работа по дизайна на нивата и играта като цяло. Unity също е популярен игрови двигател, поради което има изключително добра поддръжка и голямо наличие на форуми в случай на затруднение или проблеми по време на разработката.

В Unity всички обекти се съхраняват в така наречената сцена. В нея се създава всичко необходимо за нейното пълноценно използване. Всеки обект в сцената се състои от компоненти. Игровия двигател поддържа най-различни компоненти за използване по време на работа или възможността да се създават собствени компоненти чрез скриптове. Чрез

компонентите в различни обекти игровия двигател управлява обектите по време на работа. Има голям брой компоненти, като някои от тях са главни и са най-често използвани:

- Transform - този компонент автоматично се придава на всеки един обект, когато се създаде. В себе си той съдържа позицията, ориентацията и размера на обекта в пространството на света. Чрез този компонент може да се достъпят обектите родители и техните компоненти.
- Rigidbody - Когато обект съдържа в себе си този компонент, той се поставя под управлението на Physics engine частта на игровия двигател. Така се симулират много и различни физични закони, необходими за движението на обекта в пространството.
- Collider - Дори с компонента Rigidbody обекта не може да се сблъска с друг обект. За тази цел се използва така наречния компонент Collider. Той придава на обекта форма, която участва в изчисленията на Physics engine частта на двигателя. Така обектите могат да се сблъскват и да не минават един през друг.
- Renderer - Това е компонента, който показва обекта на екрана на потребителя. Той е част от Rendering engine частта на игровия двигател. Съществуват няколко различни вида Render компоненти, в зависимост от това дали работим с двумерна или тримерна графика.
- Camera - Този компонент се дава на обекта, който играе ролята на камера. Чрез него можем да настроим начина, по който виждаме сцената, както и позицията и ориентацията от която я гледаме.

За да създадем свои собствени компоненти, които да се използват заедно с другите компоненти, трябва да се напишат собствени скриптове.



Фиг. 2.1 Жизнения цикъл на игровия двигател Unity

Всички скриптове, които се използват директно от игровия двигател, наследяват класа MonoBehaviour. В себе си той съдържа най-голямата част от функционалността на игровия двигател, която може да се изрази в няколко функции, които се извикват по време на работа на приложението.

Във фигура 2.1 е показана опростена схема на жизнения цикъл на игровия двигател. В него се съдържат много повече функционалности, но ние ще разгледаме следните осем функции:

- Awake() - това е първата функция, която се извиква, когато се създава инстанцията на скриптовия обект. Тази инстанция се създава при началото на играта(сцената) и/или активирането на различните обекти.
- OnEnable() - Тази функция наподобява функцията Awake(). Тя се извиква след Awake() и върши подобна работа. Единствената разлика между двете е реда на изпълнение.
- Start() - Функцията се извиква на първия кадър, на който обектът и скриптовия компонент са активни. Извиква се преди Update() функциите.
- Update() - Тази функция се извиква на всеки кадър, на който обекта и скриптовия компонент са активни.

- FixedUpdate() - Подобна на Update() функцията, FixedUpdate() се извиква постоянно. Единствената разлика между двете функции е, че Update() се извиква отново само когато се смени кадъра, а FixedUpdate() се извиква на определени времеви периоди(по подразбиране е 0.02 секунди - 50 пъти в секундата), за да може да се направят добри физични изчисления.
- LateUpdate() - Тази функция се извиква след останалите Update() функции. Това е полезно, за да се подреди реда на изпълнение на скриптовете.
- OnDisable() - Тази функция се извиква, когато обект е деактивиран или разрушен. Тя е подходяща за разчистване на зависимости към други скриптове и други.
- OnDestroy() - Това е функция, която се извиква, когато играта приключи или се смени сегашната сцена с друга. Тази функция се извиква само на обекти, които са активни в края на играта или промяната на сцената.

Поради начина, по който е конструиран игровия двигател Unity, реда на извикване на всяка една функция е на случаен принцип - не се знае дали Start() метода на камерата ще се извиква винаги преди Start() метода на друг обект или наобратно.

2.2. Избор на език за програмиране

За разработка на игри игровия двигател Unity предлага собствен API написан на C#. В предишни версии Unity е поддържал обектно-ориентирания език Boo и собствена версия на JavaScript наричан UnityScript. Но с последната версия на Unity - Unity 5 тези два езика са спрени от поддръжка и е отдадено по-голямо внимание и значение на C#.

C# е обектно-ориентиран език за програмиране създаден от Microsoft като част от тяхната .NET инициатива и по-късно одобрен като международен стандарт от ECMA (европейска организация, която определя стандартите в информационните и комуникационните системи) през 2002 година и от ISO (Международната организация по стандартите) през 2003 година. Последната версия на C# - версия 9.0 е пусната през 2020 година във версия 5.0 на .NET и е включена във версията 16.8 на средата за разработка Visual Studio 2019.

2.3. Избор на среда за разработка

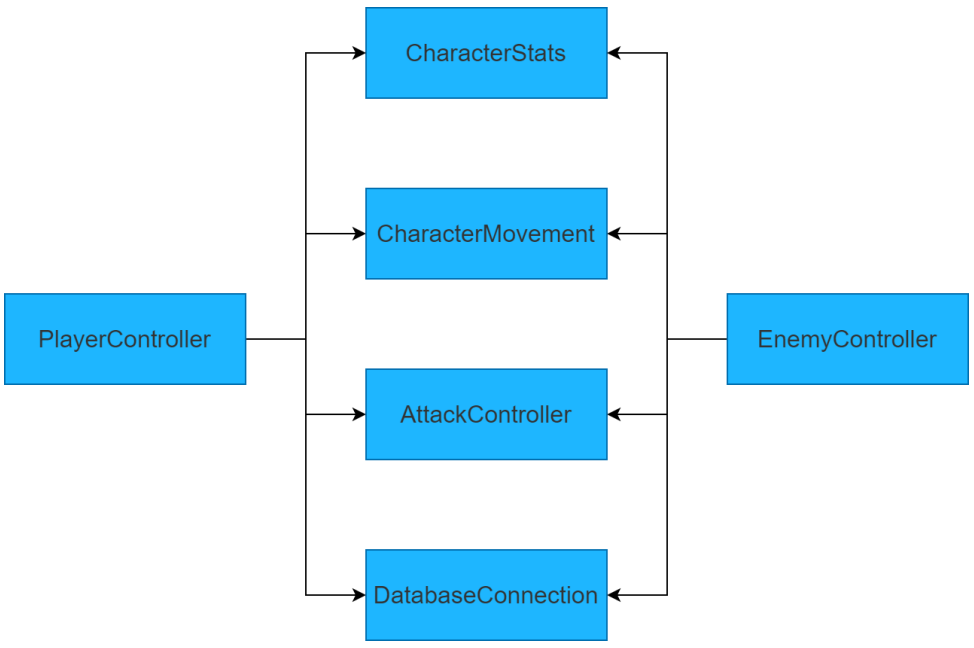
Още с инсталация на игровия двигател Unity предлага вградена среда за разработка - Visual Studio. Това IDE(среда за разработка) е достатъчно задоволително от страна на потребностите за разработка на малки игри. Но в този дипломен проект се използва IDE на компанията JetBrains наречено Rider. През годините компанията се е доказала, че може да създава изключителни добре среди за разработка като IntelliJ, CLion, PhpStorm, PyCharm и други. Rider предлага разработване на приложения на .NET, ASP.NET, .NET Core, Xamarin или Unity на операционните системи Windows, Mac OS и Linux. Rider има интегрирани няколко полезни функционалности - интелигентен редактор на код, анализ на кода, декомпилатор, навигатор и търсачка, рефакториране на кода. Тези функционалности подпомагат за по-бързата и по-качествената разработка на код и продукта като цяло.

2.4. Структура на кода

Поради структурата и начина на работа на Unity различните части на проекта трябва да се разглеждат като различни структури. Първата такава структура е тази на играча и противниците.

Както е показано в диаграмата на фигура 2.1, играча се управлява от класа PlayerController, а противниците се управляват от класа EnemyController. И двата класа съдържат в себе си като променливи класовете CharacterStats, CharacterMovement, AttackController и DatabaseConnection. Макар и големите прилики между двата класа те имат различни методи, а тези, които се срещат и в двата класа, имат различни имплементации и функционалности.

Следващата структура е тази на графичния потребителски интерфейс(UI). Той е съставен от няколко компонента, които се използват от различни класове и обекти по време на игра. Менюто, което се появява, когато потребителя слага играта на пауза, има сравнително проста имплементация - при натискане на Esc клавиша, играта замръзва и се

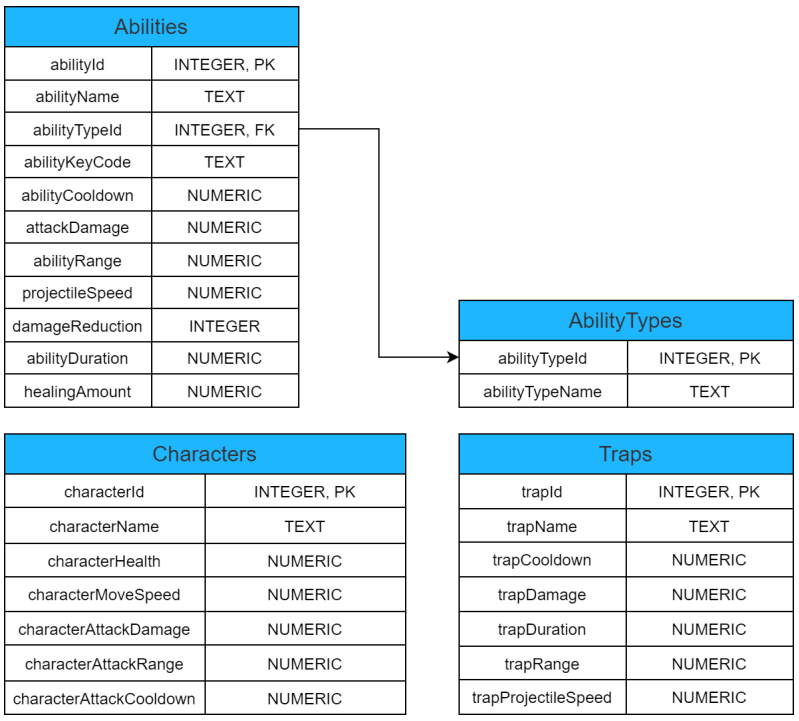


Фиг. 2.2 Структура на кода за играча и противника

появява менюто. Менюто имплементира и няколко функционалности за бутоните си, като запазване на прогреса на играча, които се разглеждат в трета глава.

2.5. Структура на базата данни

Според изискванията на тази дипломна работа, играта трябва да поддържа локална база данни, в която да се съхранява информацията за състоянието на играта и играча, както и други данни необходими за играта. За целта се използва SQLite. Тъй като не е нужна база данни с огромни размери от информация, избираме SQLite пред MySQL, защото то предлага по-малко функционалности, но използва по-малко ресурси на операционната система.



Фиг. 2.3 Структура на базата данни

Не е нужна много сложна схема от таблици, за това базата данни съдържа 4 таблици - Abilities, AbilitiesType, Characters и Traps(фигура 2.3). Най-простата таблица е таблицата AbilitiesType. Тя съдържа в себе си две

колони за категоризиране на четирите вида умения - AbilityTypeId и AbilityTypeName. Първата колона е първичният ключ на таблицата и е целочислено число, което не може да се повтаря, а втората - наименованието на вида умение.

Следващата таблица е Abilities. В нея се съхранява цялата информация (стойности) необходима за реализацията на уменията на играча. Състои се от 11 колони, което я прави най-голямата таблица в базата данни на този дипломен проект. Първите 2 колони са идентификатори на умението - първичния ключ abilityId и името на умението AbilityName. Третата колона abilityTypeId на таблицата е външен ключ, който сочи към колоната abilityTypeId на таблицата AbilitiesTypes. Тази колона е целочислено число, което отговаря на съответния запис в таблицата AbilitiesTypes. Колоната AbilityKeyCode е текстова колона, в която се записва кода на клавиша, който играчът трябва да натисне, за да активира умението. Колоната abilityCooldown показва колко време е необходимо да мине, преди играча да може да използва това умение отново. Останалите колони, в зависимост от това какъв е вида на умението, се използват различно:

- Умение за къси разстояния - използват се колоните attackDamage и attackRange;
- Умение за дълги разстояния - използват се колоните attackDamage, attackRange и projectileSpeed;
- Умение за защита на играча - използват се колоните damageReduction и abilityDuration;
- Умение за лекуване на играча - използват се колоните healingAmount.

Таблицата Characters отговаря за информацията, нужна за създаването на героите на играча и противниците в играта. Тя съдържа в себе си 7 колони:

- characterId - първичен ключ, неповторимо целочислено число;
- characterName - името на герой;
- characterHealth - “живота” на героя;
- characterMoveSpeed - бързината, с която се движи героя;
- characterAttackDamage - размера на “щетата” героя нанася на противника си;
- characterAttackRange - разстоянието на атаката на героя;
- characterAttackCooldown - времето, което трябва да измине, преди героя може да атакува отново.

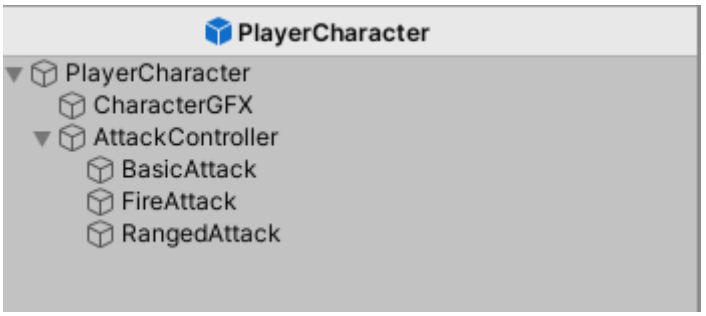
Последната таблица в базата данни на този дипломен проект е таблицата Traps. В нея се съхраняват данните за препятствията, които играчът трябва да избягва по време на игра. Тя също съдържа 7 колони:trapCooldown - време, което трябва да измине за да се активира препятствието отново;

- trapId - първичен ключ, неповторимо целочислено число;
- trapName - името на препятствието;
- trapDamage - “щетата”, която препятствието нанася на играча;
- trapDuration - време, през което препятствието е активно;
- trapRange - разстояние, на което работи препятствието;
- trapProjectileSpeed - скоростта на обекта, който се изстрелва от някои препятствия.

3. Трета глава - Разработка на двуизмерна ролева игра с преспектива от горе надолу

3.1. Разработка на играча

Играчът е тази част от играта, която се управлява изцяло от потребителя. Затова се отделя най-голямо време върху нея, за да може всички контроли да бъдат плавни. Компонентите нужни за тази цел се разделят на пет главни части - фиг. 2.2, стр. 14. Самият игрови обект в редактора на игровия двигател има следната структура от игрови обекти:



Фиг. 3.1 Структура на обекта на играча

Играчът се контролира от един главен обект наречен PlayerController, който се намира в игровия обект PlayerCharacter. Той се грижи за всички функционалности, от които се нуждае и използва играча.

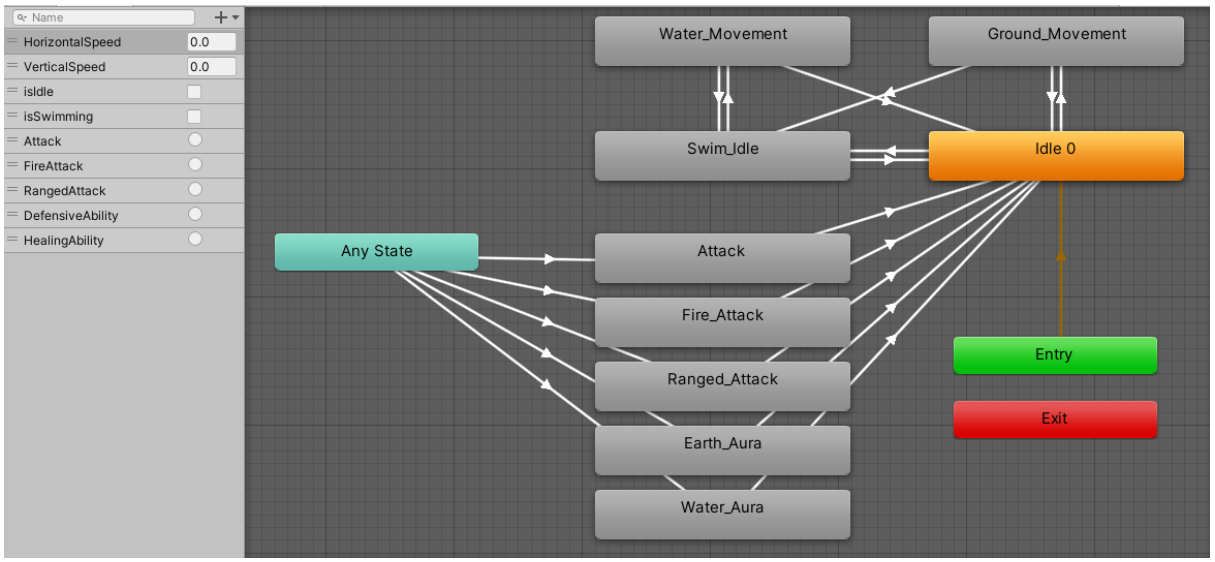
```
public class PlayerController : MonoBehaviour
{
    private CharacterMovement characterMovement;
    private PlayerDatabaseConn DBConn;
    private CharacterStats characterStats;
    private PlayerAttackController playerAttackController;
}
```

В себе си той съдържа голям брой атрибути нужни за неговата функционалност, но може да бъде опростен до тези четири - characterMovement, DBConn, characterStats, playerAttackController.

Първия атрибут от класа PlayerController, който се разглежда, е characterMovement, съдържащ се в същия игрови обект - PlayerCharacter. Чрез него се управлява движението на героя в пространството на играта.

```
public class CharacterMovement : MonoBehaviour
{
    private Rigidbody2D rb;
    private CharacterAnimationController characterGFX;
}
```

Обектът е сравнително просто конструиран, като има два атрибута, чрез които се управлява движението на героя - Rigidbody2D rb и CharacterAnimationController characterGFX. Първият атрибут е от тип Rigidbody2D (подобен е на типа Rigidbody, но работи само в двумерното пространство на играта) и чрез него се управлява игровия обект. А другият атрибут от тип CharacterAnimationController се грижи за анимацията, която трябва да се използва по време на движението на героя. Той контролира компонента Animator[1.1] в игровия обект CharacterGFX.



Фиг. 3.2 Анимации на играча и връзките за тяхната промяна

На фигура 3.2 са показани анимациите на играча и връзките между тях. Промяната на анимацията, която се изпълнява в момента, се осъществява чрез така наречените параметри. При промяна на някой

параметър, сегашната анимация преглежда своите връзки с другите анимации и в зависимост от това дали съществува такава връзка, започва друга анимация. Например, ако сегашната анимация, която се използва, е анимацията Idle и се променят параметрите HorizontalSpeed, VerticalSpeed и isIdle, анимацията се измества в GroundMovement и в зависимост от стойностите на първите два параметъра се избира анимация за ходене по земя(наляво, надясно, нагоре или надолу).

Класът CharacterMovement съдържа няколко функции, но за разбирането на начина на работа на движението на героя са нужни няколко функции:

```
public void SetCharacterVelocity(Vector2 direction){
    rb.velocity = direction;
}

public void SetCharacterDirection(Direction direction){
    characterGFX.ChangeDirection(direction);
}

public void SetCharacterSwimming(bool isSwimming){
    characterGFX.CharacterSwim(isSwimming);
}
```

Първата функция е SetCharacterVelocity(), която получава като параметър променливата direction тип Vector2[1.2]. Тази функция дава посоката и силата на движение на атрибута rb и “physics engine” частта на игровия двигател пресмята и извършва останалата работа нужна за движението на играча.

Следващите две функции се грижат за анимацията на движение на играча. Първата функция SetCharacterDirection() приема променливата direction от тип Direction[1.3] и в зависимост от нейната стойност се използва различна анимация. Втората функция SetCharacterSwimming() получава булевата променлива isSwimming, която показва дали играчът се намира във вода, при което се използват различни анимации.

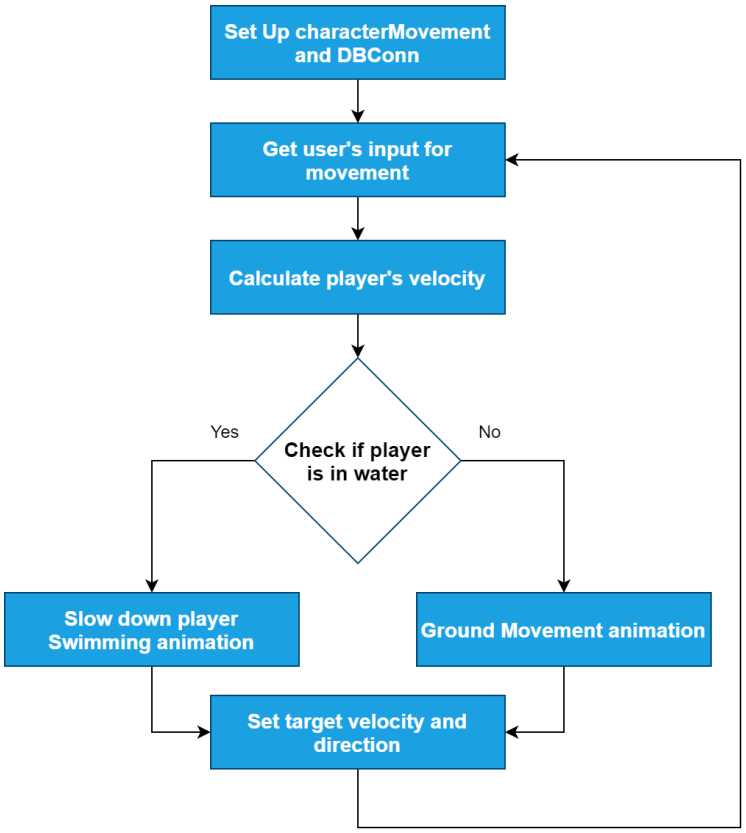
Вторият атрибут на класа PlayerController е DBConn от тип PlayerDatabaseConn. Той осъществява връзката с базата данни, в която се записва цялата информация нужна за управление на състоянието на играча. За да се осъществи движението на играча, необходима е стойността characterMoveSpeed от таблицата Characters да се вземе в началото на играта във функцията Awake().

Третият атрибут на класа е characterStats от типа CharacterStats. В него се съдържат всичките стойности на играча. Използва се, за да се избегне постоянното свързване с базата данни, което е тежък процес. В началото се взима цялата необходима информация от таблиците.

```
public class CharacterStats{
      private float health;
      private float maxHealth;
      private float moveSpeed;
      private float attackDamage;
      private float attackRange;
      private float attackCooldown;

      public CharacterStats(PlayerDatabaseConn dbConn){
                  health = dbConn.GETPlayerHealth();
                  maxHealth = m_Health;
                  moveSpeed = dbConn.GETPlayerMoveSpeed();
                  attackDamage = dbConn.GETPlayerAttackDamage();
                  attackRange = dbConn.GETPlayerAttackRange();
                  attackCooldown = dbConn.GETPlayerAttackCooldown();
      }
}
```

Последният атрибут на класа PlayerController е playerAttackContoller от типа PlayerAttackController. Това е компонент на игровия обект AttackController(фигура 3.2), който контролира цялата система за сражение между играча и противниците.

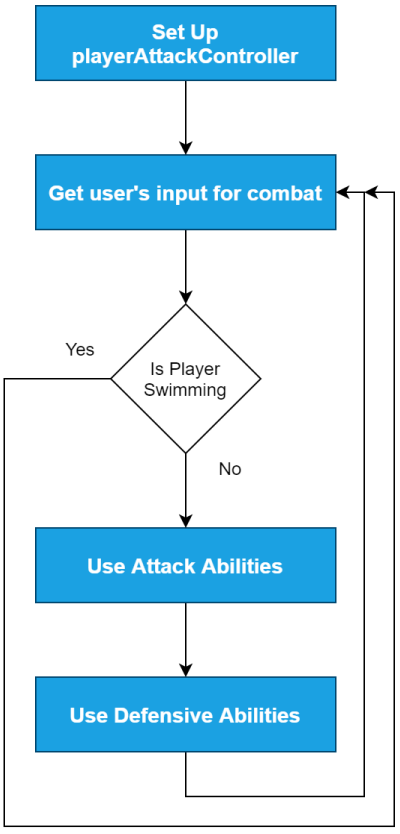


Фиг. 3.3 Блокова схема на движението на играча

За осъществяването на първото изискване на дипломната работа са необходими атрибутите DBConn, characterStats, characterMovemet. В блоковата схема на фигура 3.3 е показано как работи целия процес за движението на играча. Първо се настройват DBConn и characterMovement - осъществява се връзка с базата данни, от която се вземат всички стойности на играча и се записват в characterStats за по-нататъшно използване в другите функционалности на играча. След това се настройват атрибутите на characterMovement - rb и characterAnimationController. Те се вземат съответно от игровия обект PlayerCharacter и CharacterGFX. Втората стъпка е да се вземат входящите данни от потребителя за движението на играча - посоката на движението. В зависимост от това се изчислява новата скорост и посока на играча за съответния кадър. След това се проверява дали играча се намира във вода - ако е вярно, неговата скорост се намалява

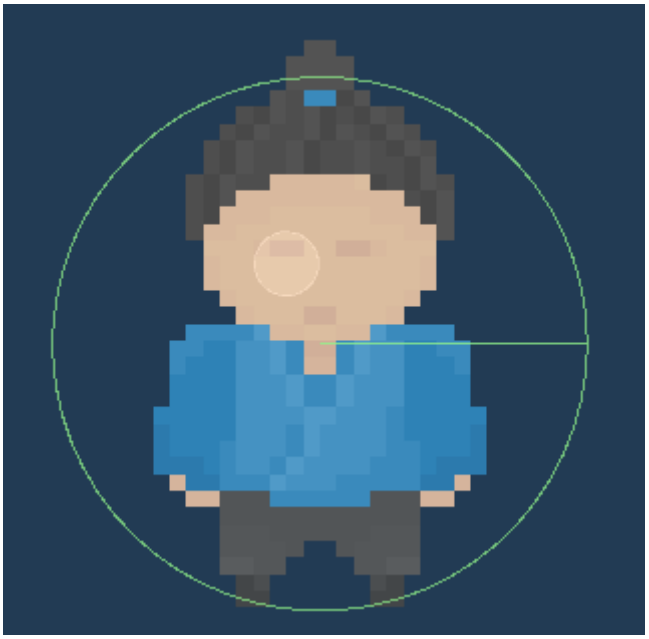
с 30% от оригиналната скорост и се използват анимациите за движение във вода, в противен случай скоростта остава същата и се използват анимациите за движение по земя. Накрая тази скорост и посока се слагат на играча и се осъществява движение по вода и земя.

Най-сложната част на устройството играча е системата за сражаване с неговите противници. За нея е отделен атрибута playerAttackController. Чрез него се управлява цялата логика на атаките на играча. На фигура 3.4 е показано опростено как работи атрибута. Първо се настройва с необходимата информация от базата данни. После се влиза в цикъла на FixedUpdate() функцията. В нея се проверява дали играча се намира във вода и ако ли не - се използват първо атакуващите умения и после тези за защита.



Фиг. 3.4 Блокова схема на системата за сражения на играча

Сега може да се навлезе по-подробно в начина на работа на уменията. Атакуващите умения се разделят на две - умения за късо разстояние и за дълго разстояние. Уменията за къси разстояния се осъществяват чрез кръгов Collider. Този Collider е от тип Trigger - за разлика от обикновения Collider, който спира обекти, когато се сблъскват, Trigger Collider пропуска обектите през себе си, но регистрира сблъсъка. Така се създава "зоната", в която, ако се намират противници, те ще поемат щета. Този Collider компонент се намира в отделен игрови обект - BasicAttack. Към този обект се прибавя скриптът AttackHitBox. Неговата



Фиг. 3.5 Обсег на действие на основната атака на играча

работа е на всеки кадър да провери дали има противник в обсега на атаката и го записва. Като се използва атаката, щета се нанася само на тези опоненти, които се намират в обсега на съответната атака.

Атаката за дълги разстояния работи по малко по-различен начин. При нейното използване се инстанцира обект, който играе ролята на "снаряд". Той съдържа в себе си компонента Collider и скрипта ProjectileHitBox. При създаването на обекта, той получава скорост в

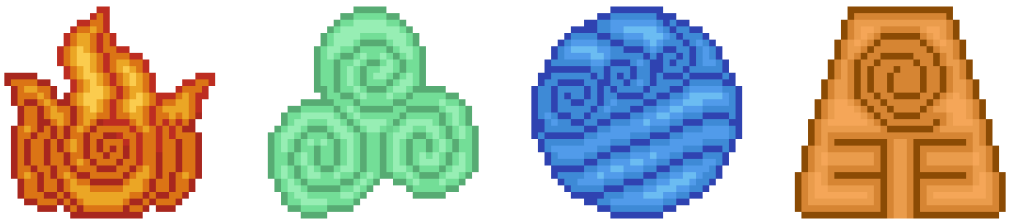
зависимост от стойностите взети от базата данни и посока, която сочи към курсора на екрана на потребителя. При сблъсък с противник скриптът нанася щета на него. Обектът се разрушава след като измине определено разстояние от началната точка.

Другите умения, които са на разположение на играча, са уменията за защита. Първото умение намаля размера на щетите, които играча получава, за определено време. При използването на умението булевата променлива isActive се променя на true. Чрез функцията `Invoke()` след определено време (`abilityDuration`) се извиква функция, която я променя обратно на false:

```
Invoke(“DisableDefensiveAbility”), abilityDuration);
```

Последното умение на играча е за увеличаването на “живота” му. При използването на умението към променливата за “живот” се добавя определена стойност, записана в базата данни. Това умение е важно за оцеляването на играча.

Така се осъществява второто изискване на дипломната работа - четири различни функционални умения. Трето изискване за подобряването на ефективността на тези умения става чрез игрови обекти. Те съдържат в себе си компонента Collider от тип Trigger и скрипт, който при сблъсък с играча подобрява различните му умения в зависимост от това, за кое умение се отнася обекта(фигура 3.6).

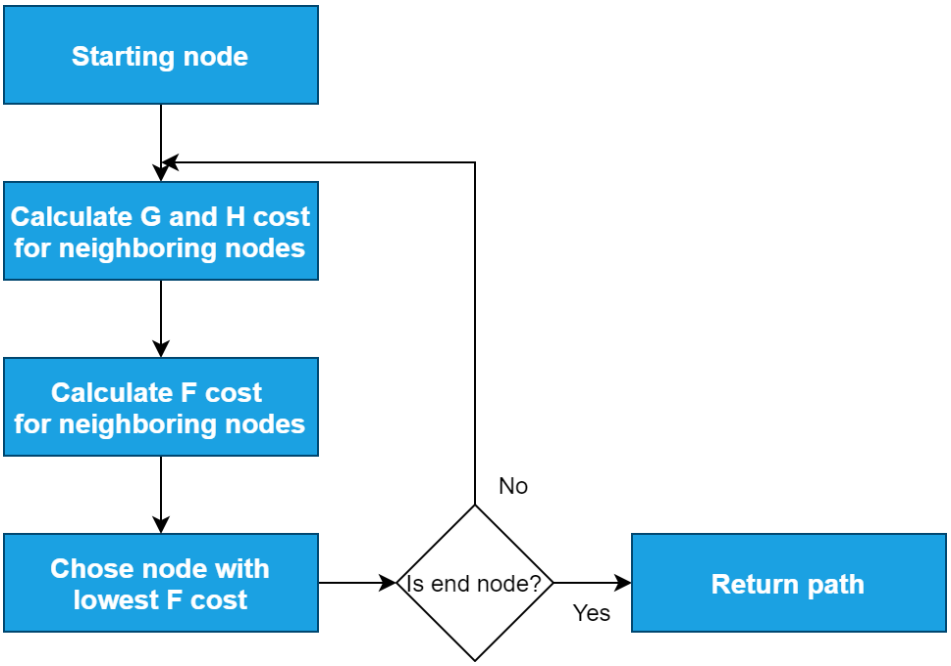


Фиг 3.6 Различни видове обекти за подобряване на уменията на играча

3.2. Разработка на противници

Следващото изискване на дипломната работа е разработката на три различни противници, с които играча да може да се сражава. Тяхното устройство е подобно на игровия обект на играча, но съдържа в себе си много по-голяма и сложна функционалност.

За разлика от играча, управляван от потребителя, който избира посоката, противниците изцяло се контролират от скрипта EnemyController. За да може противника да се движи в пространството на играта, той трябва да знае къде се намира играча. Най-лесното решение на този проблем е да се зададе посоката, в която се намира играча, и противникът да се движи към нея. Но възниква проблема, ако играчът се намира зад някакво препятствие, което блокира пътя на противника и той не може да го заобиколи. За тази цел се използва алгоритъмът A*. Това е един от най-известните алгоритми за намиране на път в пространството, като най-често се използва в видео игри и онлайн карти.



Фиг. 3.7 Логическият цикъл на A* алгоритъма

За да работи алгоритъмът, пространството се разделя на секции с определен размер наречени “nodes”. След това се избират начален и финален node. Алгоритъмът започва от началния node и на всяка итерация на цикъла пресмята така наречените G cost и H cost параметри на съседните nodes, като се игнорират тези, през които противникът не може да премине - nodes препятствия. G cost параметърът представлява “разхода” за придвижване от началния node до този, в който се намира конкретната итерация, а H cost параметъра показва подобен “разход”, но от конкретния node до крайната дестинация. С тези две стойности се изчислява така наречената F cost, което просто представлява сбора на предишните два параметъра. След тези изчисления, алгоритъмът избира node с най-малката стойност на параметъра F cost. Ако избраният node не е крайният, цикълът се пресмята отново за съседните клетки на новият node.

За осъществяването на движението на противниците в проекта се използва библиотека, в която алгоритъмът е вече имплементиран да работи с “physics engine” частта на игровия двигател Unity. Тази библиотека е изключително развита и осъществяването на движението може да се осъществи, без да се напише нито един ред код. Но за по-голям контрол върху движението на противника, ще бъдат използвани само някои компоненти.

```
public class EnemyController : MonoBehaviour{
    private CharacterMovement characterMovement;
    private EnemyDatabaseConn DBConn;
    private CharacterStats characterStats;

    private Path aiPath;
    private Seeker seeker;
}
```

Подобно на PlayerController в игровия обект на играча, EnemyController съдържа в себе си трите атрибута characterMovement, DBConn (единствената разлика с PlayerDatabaseConn е, че взима

информация само за противника) и characterStats и те работят по същия начин като в PlayerController. Само че в този случай, атрибута characterMovement не се управлява ръчно от потребителя, а от A* алгоритъма. За тази цел са необходими атрибутите aiPath от тип Path и seeker от тип Seeker. За да работи алгоритъмът, трябва да се създаде игрови обект, който да раздели пространството на nodes. Той трябва да съдържа компонента Pathfinder от библиотеката. Чрез него се определят препятствията по пътя на противника към играча и, за да могат противниците да избягват сблъсък един с друг, те също се причисляват към препятствията. Но тъй като противниците се движат, тази графа от nodes трябва да се актуализира постоянно:

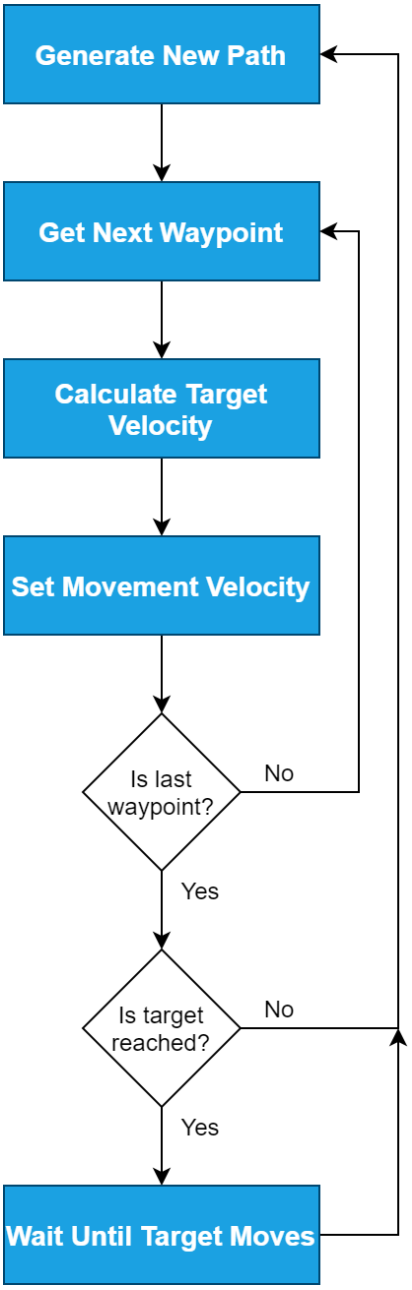
```
private void UpdateAStarGrids(){
    AstarPath.active.Scan();
}
```

След като се създаде графата от nodes за алгоритъма, може да се започне генерирането на пътят до целта. Тази работа се върши от атрибута seeker. Той генерира променлива от тип Path, която се записва в атрибута aiPath на класа:

```
seeker.StartPath(currentPosition, targetPosition, ONPathComplete);
```

На функцията StartPath() се подават три параметъра:

- currentPosition - началната позиция, от която стартира пътя, генериран от алгоритъма, като това винаги е сегашната позиция на противника;
- targetPosition - крайната дестинация (позицията на играча), към която трябва да сочи пътят;
- ONPathComplete - функцията, която се извиква при генериране на пътя. Подава ѝ се като параметър генерираният път. В този случай тя просто придава стойността на атрибута aiPath за ползване от останалите функции на класа.



Фиг. 3.8 Блокова схема на движението на противниците

Вече може да се използва този атрибут за да се придвижва противниковия герой. За целта, както е имплементирано при играча, ще се използва скрипта characterMovement, който използва Rigidbody2D компонента на противниковия игрови обект. Атрибута aiPath съдържа в себе си масив от променливи от тип Vector3[1.2]. Те представляват точки, през които минава генерирания път. След това се влиза в цикъл, в който

през всяка итерация се взима всяка следваща точка, през която трябва да мине противника. Изчислява се посоката на движение за новата точка и се слага на противника. Накрая се проверява дали сегашната точка е последната в масива. Ако е последната точка, то тогава се генерира нов път и цикълът се повтаря. Така се осъществява движението на противниците.

След това трябва да се създадат системата за сражение на противниците. За щастие може да се преизползва същата система като за играча. Единствената разлика е, че вместо атаките да се контролират от потребителя или някой друг, атаките ще се активират, когато играчът се намира в обсега на атаката на противника и атаката може да се използва отново. Всички противници имат една атака, но това, което ги различава един от друг, е начина на изпълнение на тази атака. Има три различни опонента в зависимост от техния вид атака. Първият вид противник има нормална атака, за която неговата позиция се “замразява” - той не може да се придвижва по време на анимацията на атаката. Това позволява на играча малко време да избегне сблъсъка и да предотврати поемането на щета. Вторият вид имат по-слаба атака, но могат да се движат по време на нейната анимация, което може да затрудни играча. Накрая последния тип имат една силна атака, след което игровия обект на противника се разрушава. Този вид опонент придава усещането, че играчът трябва да мисли бързо, в противен случай може лесно да загуби играта.

Фиг. 3.9 Различните видове противници спрямо тяхната атака

3.3. Разработка на препятствията

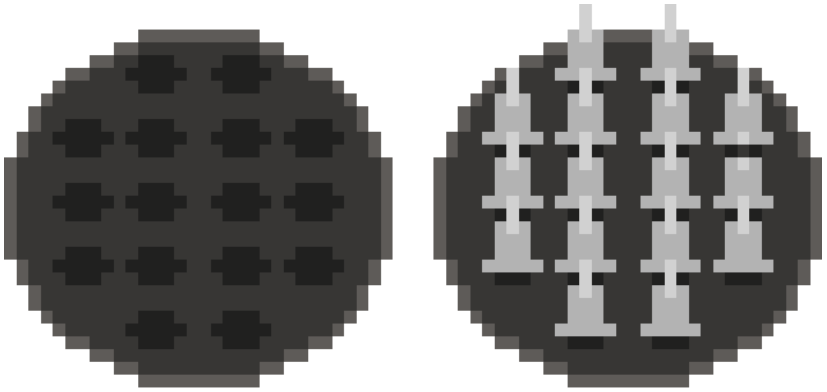
Следващото изискване към разработката на дипломната работа е да се създадат игрови обекти, играещи ролята на препятствия, които да затрудняват и разнообразяват играта. За целта ще се създадат два капана, които да нанасят щета на играча, когато се сблъска с тях, и дори на противниците, ако се примамят добре към съответния капан.

Първият капан, който е имплементиран в дипломната работа, са класическите за RPG жанра шипове. Начинът на работа на този капан е сравнително прост - когато шиповете са изправени (дясната част на фигура 3.10), капанът нанася щета или върху играча, или върху противника. А когато шиповете са скрити, играчът и неговите противници могат да минат през капана без последствия.

```
private void Awake()
{
    sprites = Resources.LoadAll<Sprite>("Sprites/Traps/Spikes");
    cooldown = new TrapsDatabaseConn("Spikes").GETTrapCooldown();
    duration = new TrapsDatabaseConn("Spikes").GETTrapDuration();
    damage = new TrapsDatabaseConn("Spikes").GETTrapDamage();
    GetComponent<Collider2D>().enabled = false;

    Invoke("ActivateSpikes", timeToActivate);
}
```

За осъществяването на този капан са необходими двете изображения на шиповете - когато са деактивирани и активирани. Към игровия обект на шиповете се добавят компонента Collider2D от тип Trigger и скриптът Spikes, а като обекти деца се добавят изображенията на шиповете, защото може да съдържат няколко броя капани в себе си. В Awake() функцията на скриптът Spikes първо се зареждат необходимите изображения - функцията `Resources.LoadAll<Sprite>()` зарежда всички файлове от тип Sprite[1.4], които се намира в директорията, показана от параметъра на функцията(`Sprites/Traps/Spikes`). След това се зарежда информацията от



Фиг. 3.10 Капани шипове

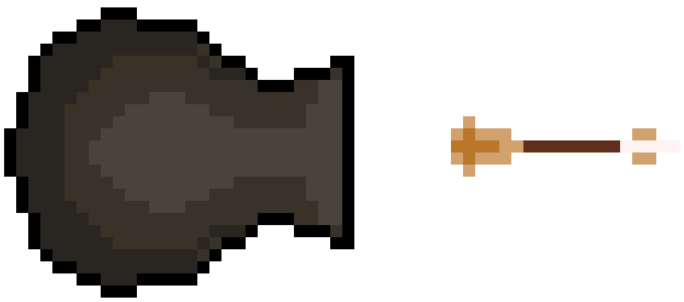
базата данни, нужна за функционалността на съответния капан. Тъй като първоначално капанът не е активен, Collider компонента на игровия обект се деактивира. Чрез функцията `Invoke("ActivateSpikes", timeToActivate)` се извиква функцията `ActivateSpikes()` след определено време, посочено от параметъра `timeToActivate`. Тя активира Collider компонента и променя изображението за всички шипове, които се намират в обектите деца на игровия обект. В края на функцията пак се използва `Invoke("DisableSpikes", duration)`, която извиква функцията `DisableSpikes()`, която извършва обратното действие на `ActivateSpikes()` и накрая се извиква наново `Invoke("ActivateSpikes", timeToActivate)`, което затваря цикъла на действие на шиповете капан.

Другият капан, който е осъществен в дипломния проект, е също друг класически за RPG жанра: оръдие. Работи на прост принцип - през определен интервал оръдието изстрелва снаряд. Този снаряд работи на същия принцип като умението на играча за атака на дълги разстояния. Игровия обект на оръдието съдържа в себе си скрипта Cannon, който го управлява, и Collider компонент, за да може играчът да не преминава през капана. Игровият обект съдържа в себе си обект дете, което ще служи като стартова позиция на снаряда. За целта използваме функция подобна на `Invoke() - InvokeRepeating("Shoot", startTime, trapCooldown)`. Тя

работи по същия начин, но се повтаря през определено време (`trapCooldown`). Функцията `Shoot()` инстанцира обекта снаряд (`projectile`), като му дава начална позиция и ориентация.

```
private void Shoot()
{
    Instantiate(projectile, firePoint.position,
firePoint.rotation);
}
```

Този игрови обект на снаряда съдържа компонентите Collider2D от тип Trigger и Rigidbody2D и скрипта Arrow. Този скрипт съдържа само имплементацията на двете функции `Awake()` и `Update()`. В първата функция се взима необходимата информация от базата данни. В зависимост от нея се изчислява скоростта, с която да се движи снаряда, и се прилага. Във втората функция се проверява изминалото разстояние на снаряда и при изминато определено разстояние (обсег) игровия обект се разрушава. При сблъсък на снаряда с обект се проверява дали е игровия обект на играча и тогава се нанася щета върху него.



Фиг. 3.11 Капан оръдие и снаряд

3.4. Разработка на потребителския интерфейс

С цялата тази функционалност имплементирана в дипломният проект, потребителят няма да може да я използва лесно. За тази цел се създава потребителския интерфейс на играта.

Фиг. 3.12 Главно меню

Първото нещо, което потребителя трябва да види при стартиране на играта е главното меню. В себе си то съдържа пет бутона, които изпълняват различна функционалност. Първия бутон Play зарежда нова игра за потребителя. Вторият бутон Levels зарежда менюто за потребителски избор на нива по шаблони. Това изискване и функционалност на дипломната работа се разглежда в точка 3.5, но по подразбиране се избират нива Forest 1, Forest 2, Forest 3. Третият бутон Load Save също стартира играта за потребителя, но зарежда последно запазеният прогрес на играча - тази функционалност се разглежда в точка 3.5. Четвъртия бутон Player Guide препраща към менюто, където се представя накратко информацията за играта, бутоните и клавишите, с които потребителя управлява героя. Последният бутон Quit затваря и прекратява работата на приложението.

При стартиране на играта, независимо дали това става посредством Play бутона или Load Save бутона, потребителският интерфейс на играча се изгражда от три части - лента, която да следи “живота” на играча, четири икони на уменията на играча, които следят дали са готови за използване и меню за пауза, което по подразбиране е скрито.

Фиг. 3.13 Потребителски интерфейс на играча

Фиг. 3.14 Лента за “живот” на играча

Така наречената лента за “живот” на играча е един игрови обект, който се състои от три обекта деца - Fill, Border, Heart. Обектите Fill, Border и Heart само визуализират изображенията чрез Renderer компонента, като Fill обекта е просто червен правоъгълник, който показва стойността на текущия живот на играча. По-важно е устройството на игровия обект HealthBar. В себе си той съдържа компонента Slider[1.5] и

скрипта HealthBar, който го контролира. Скриптът HealthBar е устроен по прост начин:

```
public class HealthBar : MonoBehaviour
{
    public Slider healthBar;

    public void SetHealth(float health)
    {
        healthBar.value = health;
    }

    public void SetMaxHealth(float maxHealth)
    {
        healthBar.maxValue = maxHealth;
        healthBar.value = maxHealth;
    }
}
```

Първоначално класът съдържа публичния атрибут healthBar от тип Slider. Той е публичен, защото не е задължително компонента Slider да се намира на същият игрови обект както скрипта, а този скрипт ще бъде преизползван в други обекти. След това чрез двете функции `SetHealth()` и `SetMaxHealth()` се управляват стойностите на компонента Slider. Първата функция задава нова стойност на компонента, който автоматично се грижи за промяната на елемента, който той управлява. Тя се извиква при всяка промяна на стойността на "живота" на играча в PlayerController. Втората функция задава двата лимита на тази стойност, но тъй като "живота" на играча постоянно е положително число, а когато е по-малко от 0, играта приключва, минималната стойност остава по подразбиране 0. Максималната стойност се взима от параметъра и се прилага, но също така се задава и на сегашната стойност, за да се запълни докрай лентата. Тя се извиква при създаването на игровия обект на играча във функцията Awake() на PlayerController класа.

Същият игрови обект може да бъде преизползван за лентата на “живота” на противниците. Единствената разлика е, че тя ще се движи заедно с противниковия обект. Като се направи игровия обект на лентата да бъде дете на игровия обект на противника, тя ще се придвижва с него.

Фиг. 3.15 Лента на “живот” на противника

Другата част на потребителския интерфейс на играча са символите, които показват дали уменията на играча са готови за използване. Работят на същия принцип като лентата за “живот” на играча - игровия им обект съдържа компонента Slider[1.5] и скриптът AbilitiesUICooldownController, който го контролира. Появява се проблема, че при лентата за живот, PlayerController подновява стойността на Slider компонента, когато има промяна в стойността на “живота” на играча, скриптът AbilitiesUICooldownController трябва да променя стойността на компонента плавно от 0 до 1. Това може да се изпълни чрез цикъл във функция, но игровият двигател ще се опита да го приключи в рамките на

един кадър. За целта ще се използва специален вид функция - `Coroutine`. Това е функция, която може да паузира своята екзекуция и да прехвърли управлението обратно на игровия двигател и на следващия кадър да продължи изпълнението си.

```
public IEnumerator CooldownFill()
{
    fillImage.fillAmount = 0f;
    float startCooldownTime = Time.time;
    float endCooldownTime = Time.time + cooldown;

    while (Time.time < endCooldownTime)
    {
        float fillAmount = MapFloat(Time.time, startCooldownTime,
endCooldownTime, 0f, 1f);
        fillImage.fillAmount = fillAmount;
        yield return null;
    }
}
```

`CooldownFill()` е функцията, която изчислява плавно стойността на Slider. Първоначално се задават стойности на променливите за начало и край на време на запълване на символа. След това се влиза в цикъл докато не премине определен момент във времето(`endCooldownTime`). Изчислява се текущата стойност за Slider компонента чрез функцията `MapFloat()`, която връща стойност от един числови интервал отнесена към друг числови интервал, и получената стойност се задава. Чрез `yield return null` функцията `CooldownFill()` прехвърля управлението на кадъра обратно на игровия двигател и на следващия кадър го получава обратно, за да продължи цикъла. За да започне своето изпълнение, функцията трябва да бъде извикана от PlayerController класа като се използва специална функция на игровия двигател - `StartCoroutine("CooldownFill")`. Тя приема като параметър името на функцията, която да изпълнява, като е задължително да връща тип `IEnumerator`.

Следващата и последна част на потребителския интерфейс на играча по време на игра е менюто за пауза. Нормално то е скрито, за да не пречи на играча. То се показва чрез натискане на бутона Esc на клавиатурата и после се скрива чрез натискането на същия бутон. При появата на менюто

Фиг. 3.16 Меню за пауза

играта се паузира чрез промяна на времевата скала - `Time.timeScale = 0f`. В менюто се съдържат пет бутона - Resume, Restart, Save Progress, Quit To Main Menu и Save And Quit. Първият бутон възобновява играта на потребителя като задава нормалната стойност на времевата скала и менюто за пауза се скрива - извършва същата работа като повторното натискане на бутона Esc на клавиатурата. Вторият бутон стартира сегашното ниво наново, като премахва прогреса на потребителя направен в него. Третият бутон запазва сегашния прогрес на потребителя. Тази функционалност е обяснена в точка 3.5. Четвъртият бутон връща потребителя в главното меню на играта, без да запазва никакъв прогрес. Последния бутон обединява функционалностите на третия и четвъртия бутон, като първо запазва прогреса и после препраща потребителя към главното меню на играта.

3.5. Разработка на системата за запазване на прогрес

Едно от последните изисквания на дипломната работа е потребителят да има възможност да запазва прогреса си в играта. За осъществяването на такава система е необходимо да се създаде обект, който да се прехвърля през сцените. Този обект ще съдържа в себе си цялата информация за прогреса на потребителя в играта. Информацията се пази в клас от тип PlayerData. В този клас се пази пътят до файла на последното запазено ниво, стойността на "живота" на играча, информация за уменията на играча, нивата, от които се състои сегашната игра, и индексът на следващото ниво.

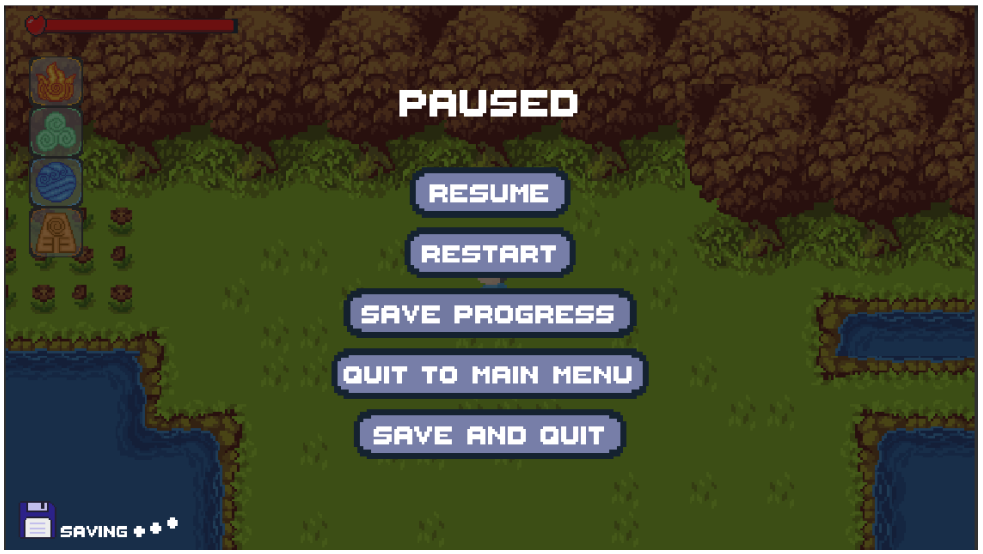
```
[System.Serializable]
public class PlayerData
{
    string levelPath;
    float health;
    float fireCooldown;
    float fireDamage;
    float windCooldown;
    float windDamage;
    float earthCooldown;
    float earthDamageReduction;
    float waterCooldown;
    float waterHealingAmount;

    string[] levels;
    int nextLevel;
}
```

Когато играта се стартира, първото нещо, което потребителят вижда, е главното меню. В същото време играта създава игрови обект GameStateController със скрипт GameStateController. Скриптът съдържа в себе си същите атрибути като класа PlayerData, но и допълнителен атрибут

`instance` - сегашната инстанция на класа. Чрез него се достъпва атрибутите на класа, за да се избегне загуба на данни при смяна на нивата.

Необходимо е създаването на клас SaveSystem, който да записва информацията от PlayerData в бинарен файл и да извлича тази информация от файла. Функцията за запазване на информацията първо форматира в бинарен формат данните от инстанцията на GameStateController и после ги записва във файла `player.bin`, който се намира `Application.persistentDataPath`[1.6]. Функцията за извличане на информацията от файлът първо проверява за съществуването на този конкретен файл. При неговото съществуване функцията извлича данните от файла и ги записва в инстанцията на GameStateController. Така Load Save бутонът на главното меню(фиг. 3.12, стр. 34) използва функцията за зареждане на прогреса на играча на класа SaveSystem. Функцията за запазване на прогреса се използва от менюто за пауза. Когато се извиква функцията символ и анимация се появяват в долния ляв ъгъл на екрана, което индикира запазването на прогреса на играча. Когато анимацията приключи и символът изчезне, тогава прогресът е записан във файла `player.bin`.



Фиг. 3.17 Запазване на прогрес в менюто за пауза

Чрез игровия обект GameStateController се осъществява изискването на дипломната работа потребителя да зарежда избрани от него/нея нива по шаблони. За целта се използва атрибута `levels` на GameStateController. В главното меню на играта след натискане на бутона Levels потребителят е препратен към ново меню, в което може да настрои последователността на нивата на играта. Със стрелките от лявата и дясната страна на първите три

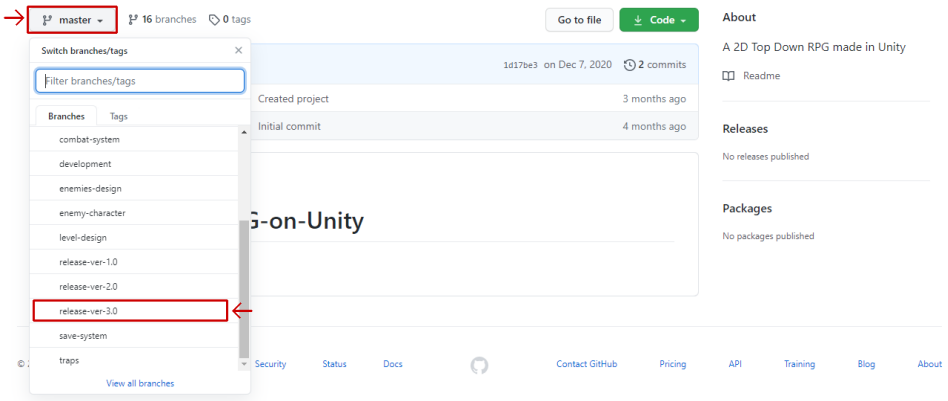
Фиг. 3.18 Меню за избор на последователността на нивата

кутии се избират съответно първото, второто и третото ниво след входящото ниво на играта. Потребителят може да избира измежду четири шаблона - Forest 1, Forest 2, Forest 3, Forest 4, като тези нива могат да се повтарят(по подразбиране нивата са Forest 1, Forest 2, Forest 3). Бутонът Back връща потребителя обратно в главното меню, а неговия избор се запазва. При натискане на бутона Play в това меню или в главното меню избора на играча се записва в атрибута `levels` на GameStateController. При зареждане на следващото ниво играта взима името на новата сцена от атрибута `levels` на инстанцията на класа.

4. Четвърта глава - ръководство на потребителя

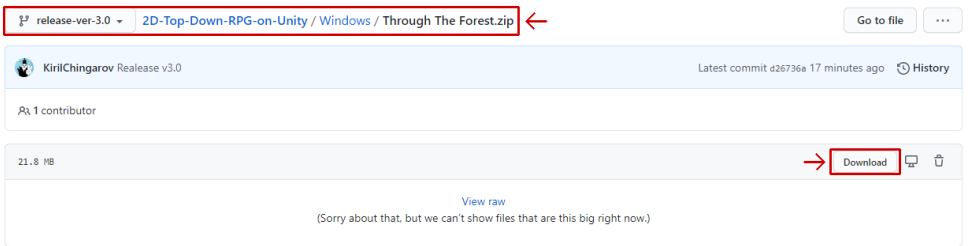
4.1. Инсталация на продукта за употреба

- Отворете линкът към хранилището на дипломния проект: https://github.com/KirilChingarov/2D-Top-Down-RPG-on-Unity
- Навигирайте до последната версия на играта:



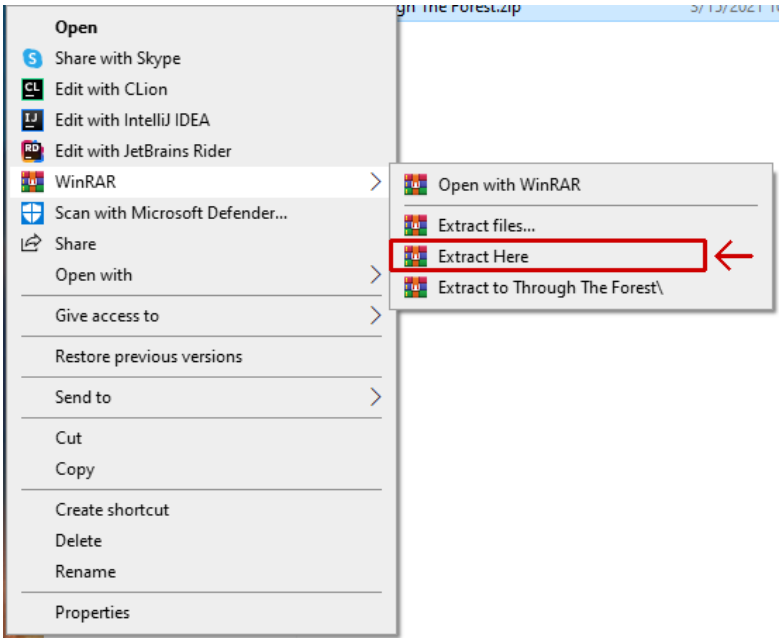
Фиг. 4.1 Главна страница на хранилището на дипломната работа

- Навигирайте до Through the Forest.zip файла и го свалете чрез бутона Download



Фиг. 4.2 Сваляне на Through the Forest.zip файл

- След като свалянето приключи, като използвате програма като Winrar или 7-zip, разархивирайте файла в папка по желание



Фиг. 4.3 Разархивиране на Through the Forest.zip

- Накрая стартирате приложението чрез изпълнимия файл “Through the Forest.exe” за операционната система Windows или “Through the Forest.app” за операционната система MacOS

4.2. Системни изисквания и препоръчителен хардуер

Windows:

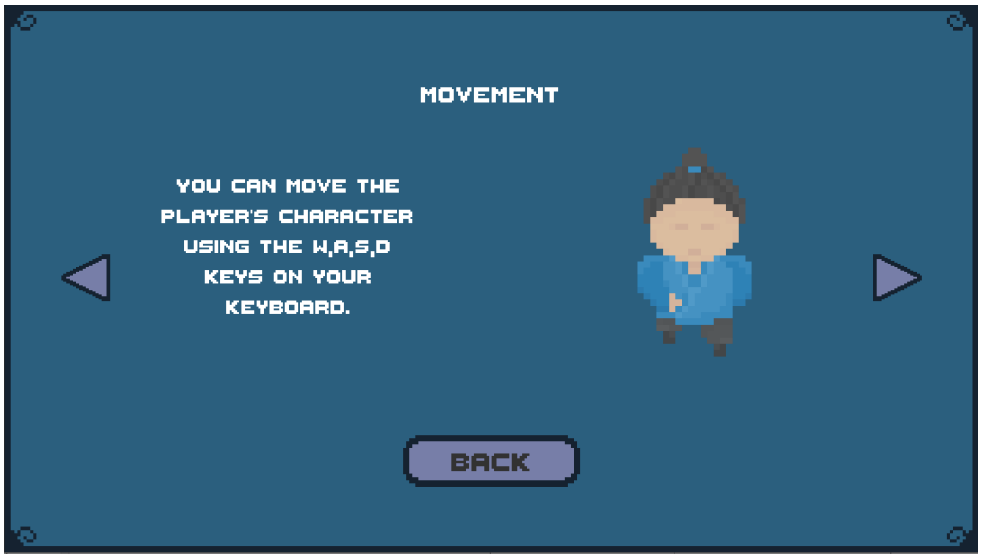
Operation System	Windows 7(SP1+) Windows 10
CPU	x86, x64 architecture with SSE2 instruction set support
RAM	8 GB
GPU	DX10, DX11, DX12 capable

MacOS:

Operation System	Sierra 10.12+
CPU	x64 architecture with SSE2.
RAM	8 GB
GPU	Metal capable Intel and AMD GPUs

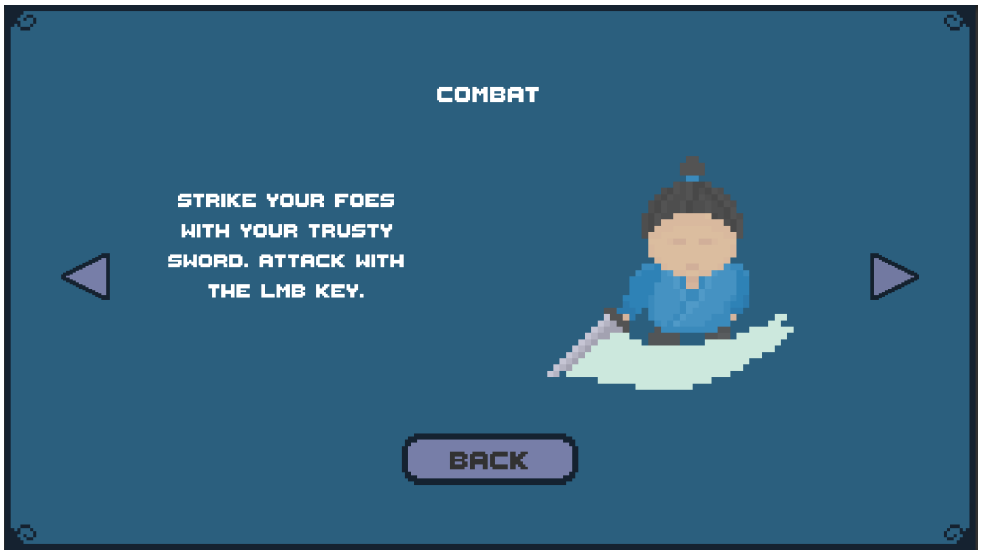
4.3. Контроли за управление на играча, ръководство за играча

Контролите за управление на играча са опростено показани в Player's Guide секцията на главното меню на играта.



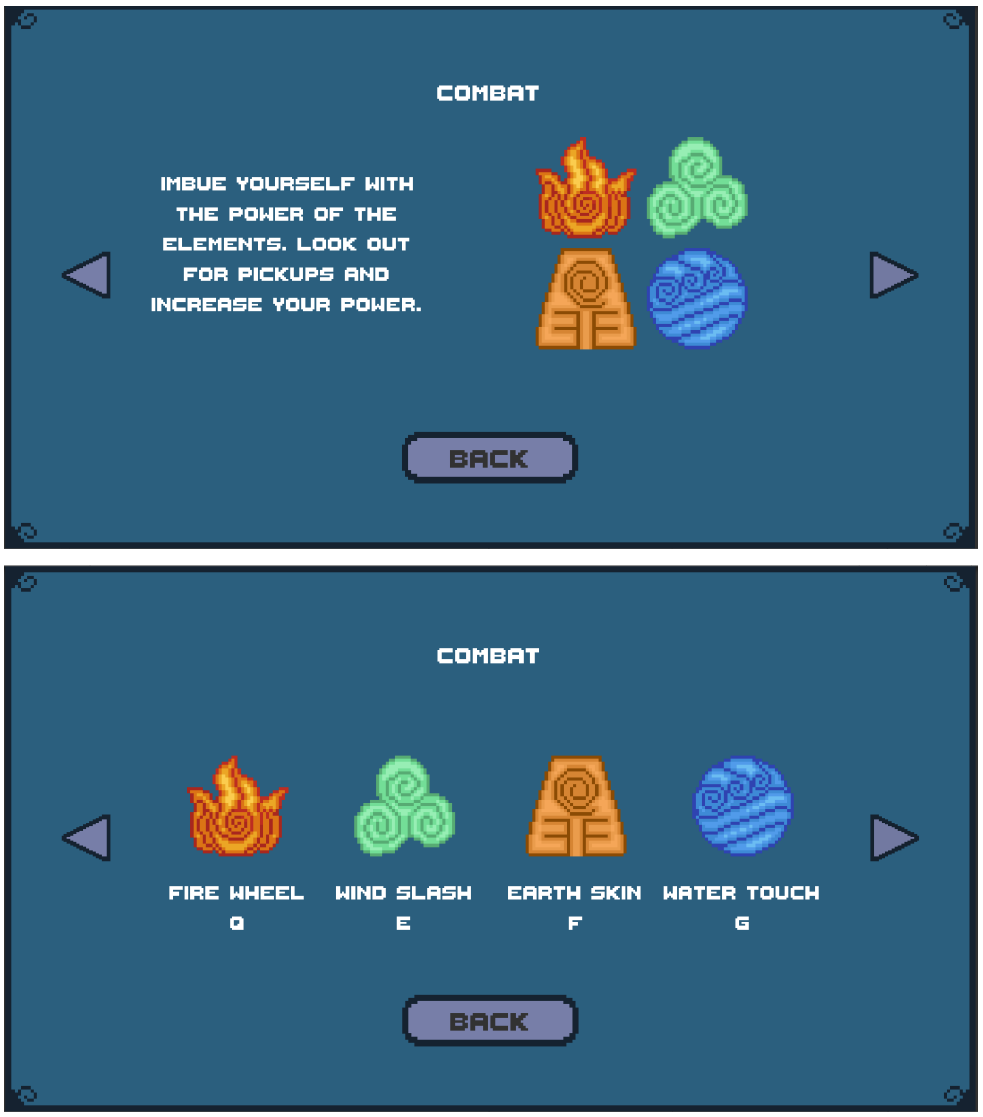
Фиг. 4.4 Първа страница на Player's Guide

В първата страница е показано как потребителя може да движи играча през пространството на играта - чрез използването на клавишите W, A, S, D на клавиатурата.



Фиг. 4.5 Втора страница на Player's Guide

Във втората страница показва основната атака на играча, която потребителят може да използва като натисне левият бутон на мишката.

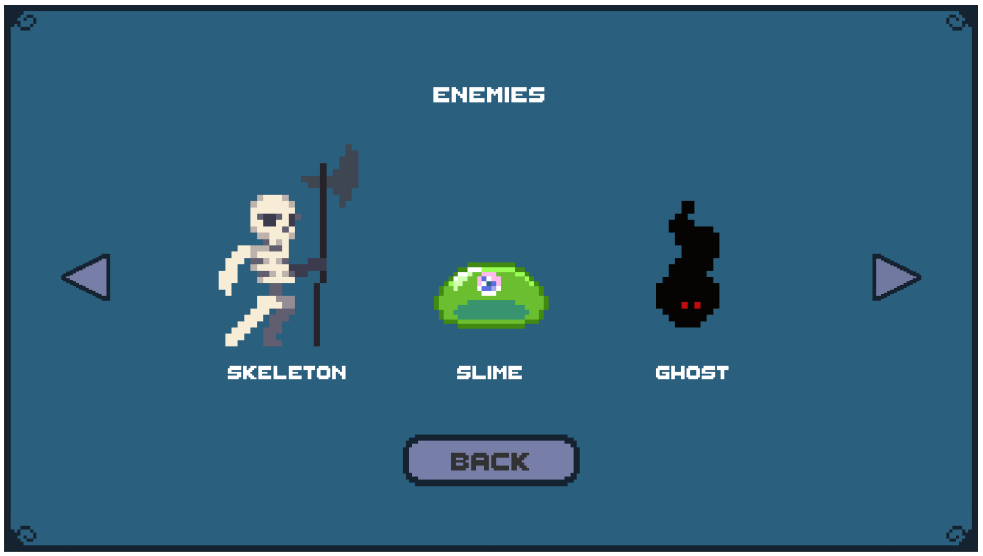


Фиг. 4.6 Трета и четвърта страница на Player's Guide

Трета и четвърта страница показват уменията, които са на разположение на играча:

- Fire Wheel - умение за атака на близки разстояния. Огнен кръг, който нанася от умерена до висока щета върху противниците на играча. Потребителят активира умението чрез натискане на клавиша Q на клавиатурата.

- Wind Slash - умение за атака на дълги разстояния. Вятър, който изминава определено разстояние (обсег) и нанася умерена щета на всички противници, през които премине. Потребителят активира умението чрез натискане на клавиша E на клавиатурата и го насочва чрез курсора на екрана.
- Earth Skin - умение за защита. Намаля размерите на щетите, които играчът поема, за определен период от време. Потребителят активира умението чрез натискане на клавиша F на клавиатурата.
- Water Touch - умение за “лекуване”. Възстановява определена стойност на “живота” на играча. Потребителят активира умението чрез натискане на клавиша G на клавиатурата



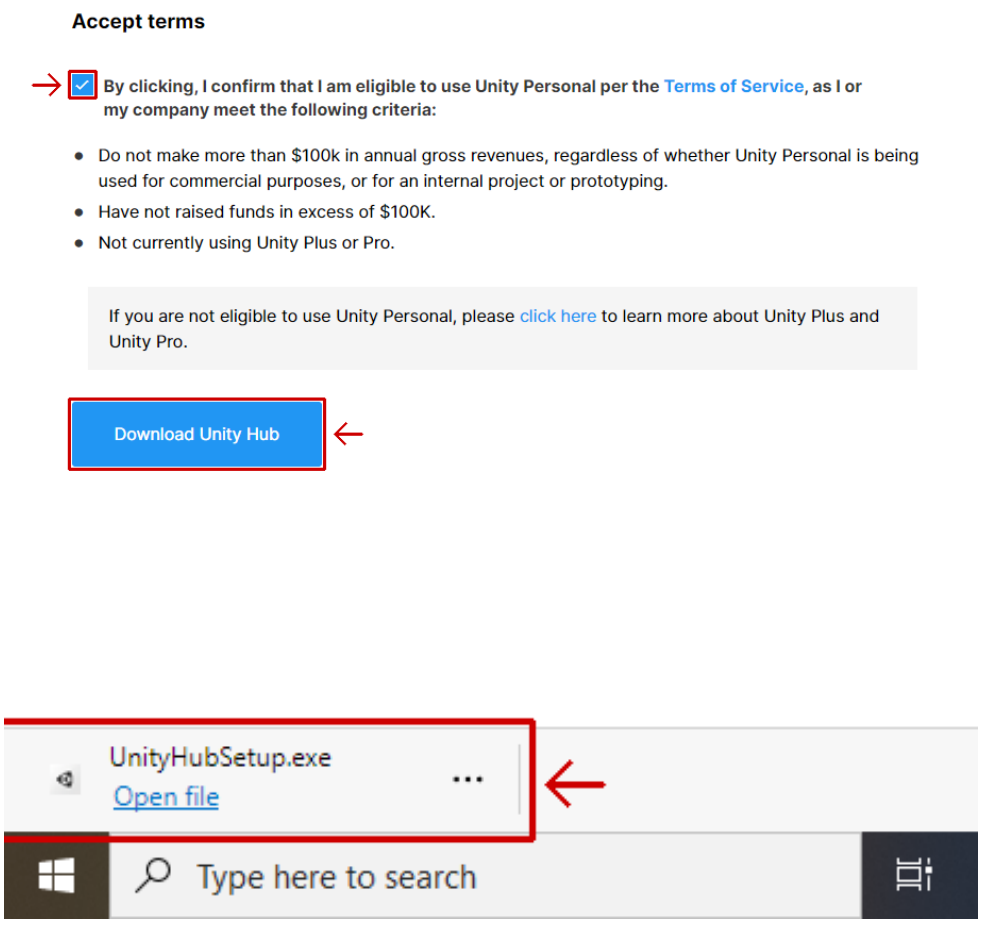
Фиг. 4.7 Последната страница на Player’s Gudie

Последната страница на Player's Guide секцията показва различните видове противници за играча.

4.4. Инсталация на проекта за лична разработка

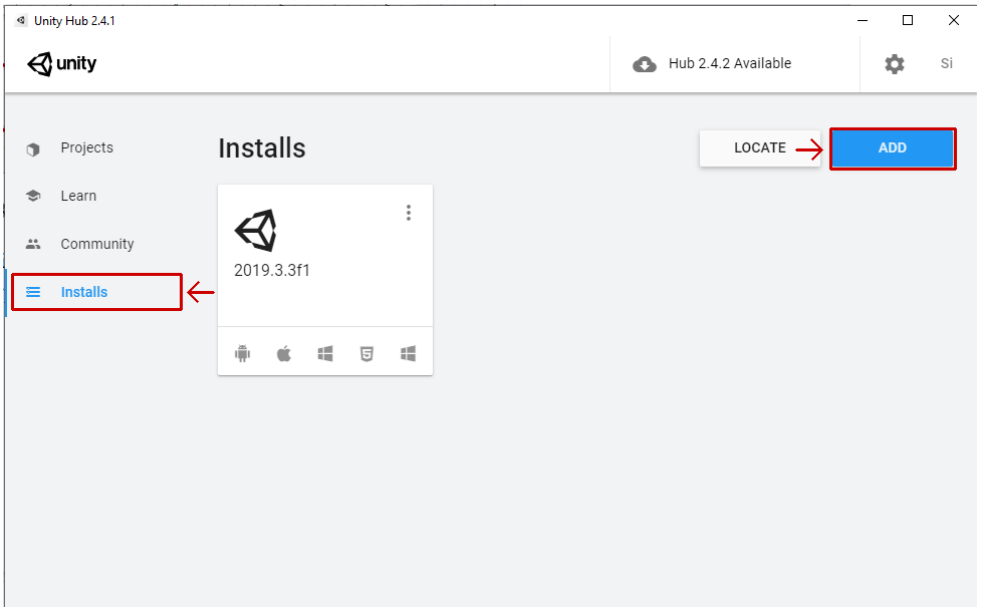
За инсталация и настройване на проекта за лична разработка трябва да се изпълнят следните стъпки:

- Навигирайте до официалния сайт за инсталиране на Unity Hub: https://store.unity.com/download
- Инсталирайте Unity Hub



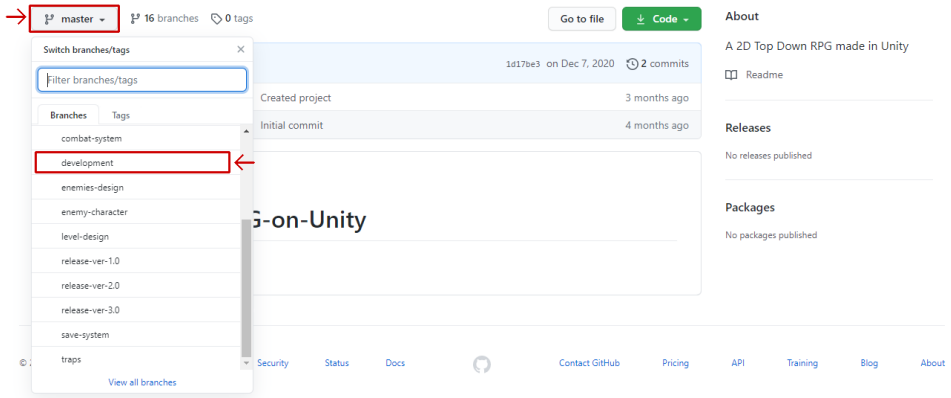
Фиг. 4.8 Инсталиране на Unity Hub

- Стартирайте Unity Hub и инсталирайте игровия двигател Unity, версия 2019.3.3f1



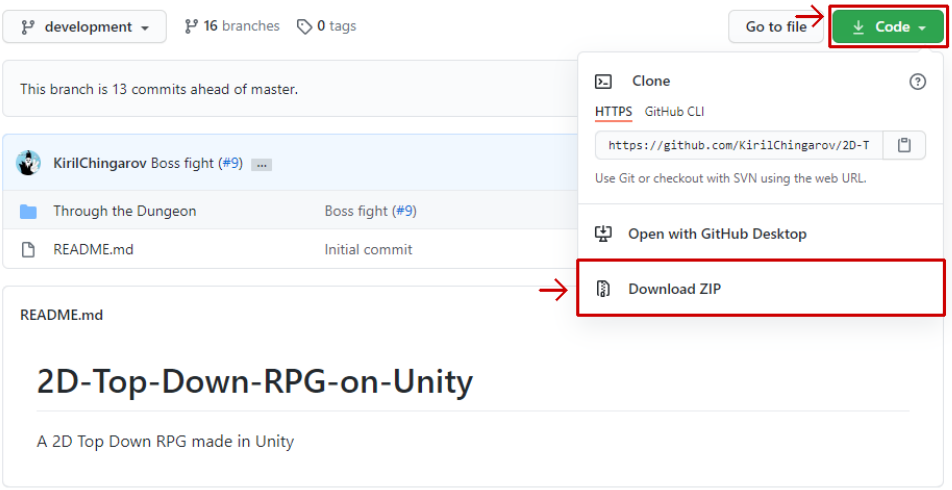
Фиг. 4.9 Инсталация на игровия двигател Unity

- Навигирайте до хранилището на дипломния проект: https://github.com/KirilChingarov/2D-Top-Down-RPG-on-Unity
- Навигирайте до development бранш на проекта



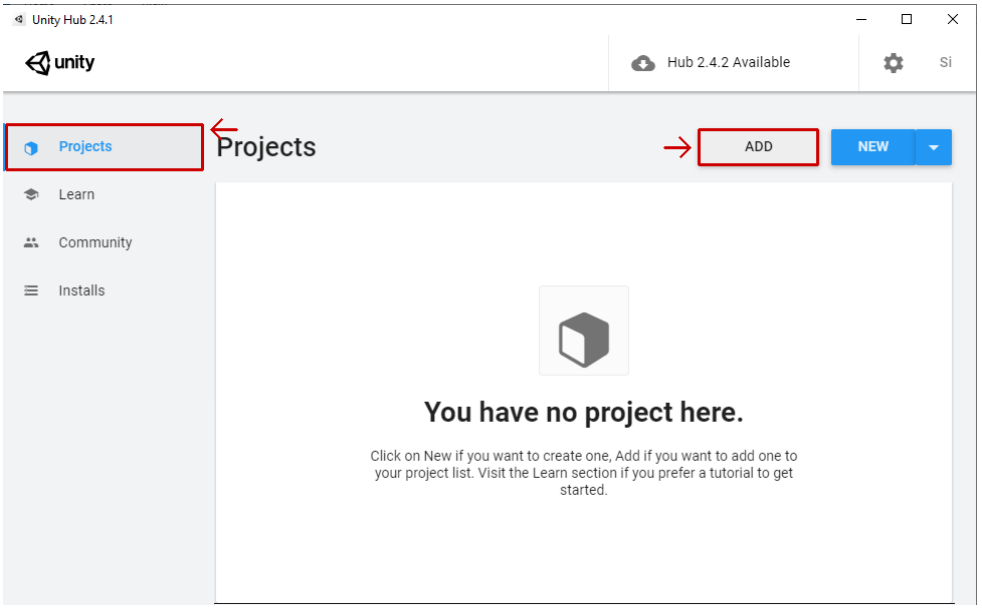
Фиг. 4.10 Навигиране до development бранша на дипломния проект

- Свалете архивирания .zip файл на проекта



Фиг. 4.11 Сваляне на дипломния проект

- След разархивирането на файла добавете проекта в Unity Hub



Фиг. 4.12 Стартиране на проекта

- Стартирайте проекта

Заключение

В поставеното време за разработка на дипломната работа е успешно създадена основа за развитие на малка ролева игра. Имплементиран е играч, който може се движи в пространството на играта и да се сражава с противници. От своя страна противниците сами се придвижват в пространството на играта и се сражават с играча. Създадени са обекти, които да разнообразяват играта на потребителя, и потребителски интерфейс за улеснено използване на продукта.

За да може една игра да бъде цялостна е нужен екип от хора - артисти, програмисти, дизайнери, композитори, писатели и др., които да работят върху различните елементи на играта. Със съвместна работа може да се създаде една приятна игра, която може да се играе многократно.

Бъдещето развитие на проекта включва добавянето на повече нива с по-разнообразни планове, по-добър дизайн на потребителския интерфейс, на логиката на противниците и повече функционалности на играча. За да може потребителят да се потопи изцяло в играта, възможно е добавянето на по-добри изображения, анимации, ефекти, да се добави музика и звукови ефекти, както и история на играта.

Библиография

- Unity User Manual, docs.unity3d.com/2019.3/Documentation/Manual/UnityManual.html, 2019
- Rizwan Asif, SQLite and Unity: How to do it right, medium.com/@rizasif92/sqlite-and-unity-how-to-do-it-right-31991712190, 17 Юли 2018
- Brackeys, Tutorials, www.youtube.com/channel/UCYbK_tjZ2OrIZFBvU6CCMiA
- A* Pathfinding Project, https://arongranberg.com/astar/front
- Unity Technologies, Unity official website, unity.com
- Epic Games, Unreal Engine 4 official website, www.unrealengine.com/en-US/
- Godot, Godot official website, https://godotengine.org/
- YoYo Games, GameMaker Studio official website, https://www.yoyogames.com/gamemaker
- JetBrains, Rider official website, https://www.jetbrains.com/rider/
- Development of The Legend of Zelda Series, https://zelda.gamepedia.com/Development_of_The_Legend_of_Zelda
- Pokemon Wiki, https://pokemon.fandom.com/wiki/Pok%C3%A9mon_Wiki
- Dragon Quest Wiki, https://dragonquest.fandom.com/wiki/Dragon_Quest_Wiki
- Използвани изображения и анимации:
    - Ansimuz, Tiny RPG - Forest, assetstore.unity.com/packages/2d/characters/tiny-rpg-forest-114685, 4 Април 2018

- Tiny Worlds, Free Pixel Font - Thaleah, https://assetstore.unity.com/packages/2d/fonts/free-pixel-font-thaleah-140059, 5 Април 2019
- Szadi Art., Rogue Fantasy Castle, https://assetstore.unity.com/packages/2d/environments/rogue-fantasy-castle-164725, 15 Март 2020
- Jesse M, Skeleton Sprite Pack, jesse-m.itch.io/skeleton-pack
- Pixelnauta, Slime Pixel 32x32, pixelnauta.itch.io/slime-pixel-32x32
- Kronovi, Undead Executioner, darkpixel-kronovi.itch.io/undead-executioner

Приложение 1 - използвани съкращения и термини

1. Animator - компонент, който се грижи за анимацията на обект като променя картината, която се показва от Render компонента. Управлява се чрез параметри от различен тип - Float, Int, Bool и Trigger.
2. Vector - клас, който описва векторите в математиката и линейната алгебра - може да се възприема като посока в пространството.
3. Direction - enum, чрез който се показват посоката на движението на играча. Съдържа в себе си пет стойности - Up, Down, Right, Left и Idle,
4. Sprite - клас, който съдържа в себе си информацията за дадено изображение. Render engine частта на игровия двигател използва този клас за визуализацията на играта.
5. Slider - компонент, който съдържа в себе си стойност, която може да бъде изменена между минимална и максимална стойност. В зависимост от тази стойност, компонента може да промени визуализацията(промяна на дължината на обекта в определена посока) на определен елемент от потребителския интерфейс.
6. Application.persistentDataPath - директория, в която може да се записват файлове и информация, която да се използват за многократно пускане на приложението. В зависимост от операционната система на машината, persistentDataPath се намира на различни места(Windows: %userprofile%\AppData\LocalLow\<companyname>\<productname>).

Приложение 2 - сорс код и CD

Линк към хранилището на дипломния проект:
https://github.com/KirilChingarov/2D-Top-Down-RPG-on-Unity/tree/master

CD:

Съдържание