ТЕХНОЛОГИЧНО УЧИЛИЩЕ “ЕЛЕКТРОННИ СИСТЕМИ”
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

ДИПЛОМНА РАБОТА

Тема: Разработка на 3д shooter игра с Unreal Engine 4

Дипломант: *Михаил Киров*

Научен ръководител: *Виктор Кетипов*

СОФИЯ
2020

Увод

Видео игрите са индустрия, която се заражда през 70-те години на 20-ти век. Първата игра, която се изчислява от компютър и се играе на някакъв вид екран е Nutting Associates, която излиза на пазара през 1971 г. Една година по-късно, Нолан Бушнел създава компания за разработка на видео игри, която кръщава Atari, давайки начало на една от най-успешните индустрии в наши дни. Отначало, игрите са с изключително проста двумерна графика, а най-ранните разработки дори са имали възможността да възпроизвеждат изключително лимитиран набор от цветове. С разрастването на т.нар. „аркадни“ игри и разработката на все по-мощни изчислителни машини, индустрията започва да произвежда все по-сложни и иновативни в дизайна и разработката си игри. Истинската иновация идва, когато компанията Sony пуска на пазара една от най-продаваните конзоли на всички времена - първият PlayStation. Конзолата играе ключова роля в развитието на игрите, тъй като дава достъп на масовия потребител до машина, която може да изчислява триизмерни изображения. Първоначално, 3D графиката е изключително проста, но изключително важна за развитието на видео игрите като нова форма на забавление. Процесорите започват да се развиват стремглаво напред и закона на Мур е в пълна сила. Започват да излизат нови техники за рендериране на обекти, както и програмируеми шейдъри, което позволява програмистите да задават начина на обработка на геометрията и текстурите в графичния процесор. Всичките тези нови развития в областта на компютърната графика, обаче са сложни за имплементация и тяхното имплементиране за дадена игра изисква много време и ресурси. Така започва създаването на различни игрови двигатели.

Всеки игрови двигател разполага с разработен модел за изчисляване на графиката за играта, както и с набор от инструменти за анимация, неинтерактивни сцени и изчисляване на физика. С развитието на игрите, започват да се появяват много различни жанрове.

От стратегически до приключенски игри, от игри със стрелба до различни симулации. Стрелковите игри се оказват едни от най-интересните за играчите. Голяма

част от игрите във времето имат стрелкови елементи. Още двуизмерните пиксел арт игри със роли имплементират подобни механики. В началото аркадните съревнователни игри с двама играчи (Street fighter) предлагат само битки с близък контакт и отбягват използването на проектили, поради по-сложната им имплементация и използване на повече ресурси. С развитието на възможностите на по-новите процесори и видеокарти тези игри се развиват експоненциално. Получават по-сложна графика и по-интересни и сложни механики.

Някои от игрите включват кооперативен елемент (Counter strike, Dual Dragons), други се фокусират върху съревнователна част.

Разглеждайки горепосочените факти, целите на дипломната работа стават кристално ясни. Играта, по задание, трябва да е триизмерна, което значи, че трябва да се използва някакъв вид „game engine“, тъй като програмирането на самата графика е трудоемко и използването на такъв софтуер позволява ресурсите да се насочат към разработването на самата игра, добавяне на изкуствен интелект, имплементиране на интересни функционалност без нужда от работа на ниско ниво. Механиките, които ще бъдат разработени са фундаментално необходимите за този жанр на игра (вземане, стреляне, презареждане и смяна на оръжия). Те са сходни във всяка игра от този жанр, което прави играта интуитивна за всеки потребител играл която и да е друга стрелкова игра.

Това, което прави тази стрелкова игра по-различна, е разработката на противников изкуствен интелект, който да се държи като истински играч. Заедно със кооперативния режим на играта, тази дипломна работа цели да създаде по-различна и интересна гледна точка на стрелковия жанр.

1. Първа глава - проучвателна част

1.1. История на игровите двигатели

ID Software разработват и пускат първата игра от поредицата DOOM през 1993 г. Тогава те създават нещо иновативно и невиждано до момента. Играта обещава иновации в областта на технологиите, геймплея, дистрибуцията и създаването на игри.Дори представят нов термин за това – “DOOM engine”, което е наименованието на технология, върху която е разработен най-новия играл софтуер на Id. Компанията обещава нов вид „отворена игра“ и с излизането си тя наистина става основата на нова индустрия за компютърни игри. Играта представлява двуизмерна стрелкова игра с графики симулиращи триизмерна игра. Създаването на нов вид софтуер, обаче, не е достатъчно постижение за Джон Кармак, главният програмист в id. Той също измисля начин на структуриране на компонентите на игрите, разделяйки ядрото на програмата от креативната част т.е. всичко, което населява света на играта. Може да се каже, че през 90те не компютрите са мястото, което е асоциативно свързвано с игри. Конзолите доминират пазара, докато изключително популярните за времето 8 и 16 битови компютри започват да виждат рязък дезинтерес от страна на потребителите. Персоналният компютър тепърва има да се доказва като платформа за дизайн на игри. С пускането на DOOM през 93-та, се стартира техническа революция. Играта показва наистина новаторски технология и дизайн, възползвайки се от VGA графика с цветова палитра от 256 цвята, онлайн игра и режим на игра, наименуван “death match”. DOOM обособява нова парадигма как един игрови двигател трябва да бъде структуриран, която е в основите на модерният дизайн на компютърни игри.

1.2. Обзор на различните технологии за разработка на видео игри

За разработката на видео игри, както се споменава в увода,най-често се използват вече готови игрови двигатели. Те биват няколко вида и всеки има различно предназначение, както и свои силни страни и слабости. Най-популярните за момента са Unity и Unreal Engine 4. Има и други, но те не са навлезли до такава степен в пазара или не предлагат инструменти, които са наистина иновативни. Примери са CryEngine и GameMaker. Cryengine, след години платен лиценз, реши наскоро да предлага и безплатен такъв, с цел примамването на т.нар. „независими разработчици на игри“. Това са малки по размер студия, които обикновено нямат голям бюджет или голям на брой персонал и разчитат на безплатни игрови двигатели като Unity и Unreal, с които могат да правят качествени продукти, дори и с нисък бюджет. С този ход, CryEngine се опитва да примами точно такава аудитория. За жалост, въпреки завидните графични способности на игровия двигател, той все още трябва да се обособи като надежден инструмент в арсенала на средностатистическия разработчик. GameMaker, от друга страна е на другия край на спектъра. Той е безплатен от доста време, но способностите му за рендериране на 3D графика не са от калибъра на останалите игрови двигатели на пазара, което затваря неговия пазар в 2D пространството. Той е много добър игрови двигател за разработка на двуизмерни игри, но не е нещо повече от това. Unity безспорно е най-популярният engine на пазара. Безплатния му лиценз е факт от немалко години, което означава, че през годините той успява да изгради общност от независими разработчици с невероятни размери. Способностите на този игрови двигател са обширни в двумерното, както и в триизмерното пространство.

Той е направен така, че да се използва изключително лесно дори и от най-неопитните в програмирането. Позволява на абсолютно всеки да направи красива игра с добри ефекти. Негови недостатък обаче е затвореният код. Unity не позволява на разработчика да види как работи функционалността, която ползва, което до известна степен я лимитира. Unity има голяма общност и форум, в който има решени много проблеми. Това обаче не заменя удобството на отворения код, защото с Unity дори

най-голямото студио за разработка на игри ще е безсилно при проблем със предоставената от игровия двигател функционалност. Използвания от Unity език за програмиране е C#. Той е удобен за употреба и предоставя garbage collector. Това улеснява работата значително. Въпреки това обаче той не е толкова бърз, колкото ползвания от Unreal Engine 4 език - C++. Бързодействието е важно за видео игрите, защото целим максимално бързо изчисление на игровата логика, физики и графики. Това е особено вярно за игрите в реално време като стрелковите игри.

Друг игрови двигател е Unreal Engine 4. Още със самото му обявяване , Epic Games (студиото, което го разработва), обявява, че той ще е безплатен за всички, като разработчиците ще трябва да плащат процент от печалбите си, но само ако те започнат да печелят суми от порядъка на няколко хиляди долара. Това решение популяризира игровия двигател още повече. Общността на разработчици на Unreal и преди обявяването на неговия безплатен лиценз бива от завидни пропорции, но след неговото пускане на пазара безплатно допринася за истински бум в броя на разработчиците, които използват Unreal. Общността от разработчици не е с големината на тази на Unity, но все пак нейният размер не трябва да се подценява. Подобно на Unity, Unreal също имат блог, в който публикуват интересни статии и често дори организират състезания, в които разработчиците могат да спечелят финансиране за проекта си. В YouTube канала на компанията също има уроци, които имат за цел да покажат дори на най-начинаещите как се използва програмата и различните й инструменти. Отново, както при Unity, и тук има огромна общност от програмисти, художници и дизайнери, които бързо могат да решат всеки проблем, стига да се намери някой, който да не е вече решен из огромните дълбини на форума. Главното предимство на Unreal е отвореният сорс код. Всеки разработчик може да види имплементацията на функциите, които ползва, което му позволява по-лесно дебъгване, оптимизиране и цялостно по-добре написан и разбираем код. Също така има огромна общност от хора, които развиват отворения код на игровия двигател. Всеки може да подобри, да оптимизира или да документира кода. Заради отворения код към него биват добавяни много повече функционалности и проблемите биват решени много по-бързо.

Unreal Engine използва C++ като език за програмиране на игровата логика, което го прави по-бърз от Unity. Също така Unreal разполага с визуално програмиране

(Blueprint), което се оказва особено удобно при бързо прототипиране на дадена функционалност.

Blueprint системата позволява да се програмира цяла игра, без да се напише и един ред код Като цяло Unreal, с неговата невероятна рендерна мощ, както и лесното му използване, го правят програма, която влиза в арсенала не само на нискобюджетните студия, но и в арсенала на едни от най-големите студия в индустрията. Специалните му render настройки му позволяват да извежда едни от най-реалистичните графики в индустрията. Поради лесното му използване и способността да се направи прилично изглеждаща игра с по-малко усилия, Unreal е игровия двигател, който стои зад разработката на тази дипломна работа.

1.3. Обзор на различните типове машинно самообучение

1.3.1. Надзиравано обучение

При самообучението с учител алгоритъмът изгражда математически модел на съвкупност от данни, която съдържа и входящите данни, и желаните изходящи резултати. Например, ако задачата е да се определи дали изображение съдържа даден предмет, обучаващите данни за самообучение с учител биха включвали изображения със и без съответния предмет (входящи данни), като всяко изображение има етикет (желан резултат), определящ дали съдържа предмета. В специални случаи входящите данни може да са налице само частично или да са ограничени до специфичен вид обратна връзка. Алгоритмите за смесено (отчасти надзиравано) индуктивно самообучение построяват математически модели от непълни обучаващи данни, в които за част от обучаващите примери не е зададен желаният резултат.

Алгоритмите за класификация и регресия спадат към самообучението с учител. Първите се използват, когато желаните резултати се свеждат до ограничен набор от стойности. За класифициращ алгоритъм, който филтрира електронна поща, входът би представлявал получено електронно писмо, а резултатът – името на папката, в която да

бъде поставено. За алгоритъм, който разпознава спам, резултатът би бил прогноза от вида „спам“ или „не спам“, представена с булевите стойности единица и нула. Резултатите на алгоритмите за регресионен анализ са непрекъснати величини, което означава, че могат да приемат произволна стойност в даден диапазон. Примери за непрекъснати величини са температурата, дължината или цената на даден предмет.

1.3.2. Ненадзиравано обучение

Алгоритмите за ненадзиравано самообучение приемат съвкупност от данни, съдържаща само входящи стойности, и намират структура в данните, например групи или клъстери. Тези алгоритми се обучават от тестови данни, които не са надписани, класифицирани или категоризирани. Вместо да реагират на обратна връзка, те разпознават общи характеристики в данните и реагират според присъствието или отсъствието на тези характеристики във всяка нова порция данни. Основното приложение на ненадзираваното самообучение е при оценка на плътността в статистиката, макар че то обхваща и други области, свързани с обобщаване и обясняване на признаци в данни.

1.3.3. Обучение с утвърждение

В контекста на изкуствения интелект, терминът "Обучение с утвърждение" означава група от методи за автоматично самообучение. Тези методи се отличават със способността си да функционират без необходимост от примерни решения на поставения проблем. Обучението протича като последователност от пробни действия, които постепенно водят до утвърждаване на добрите действия и избягване на неподходящите.

Проблеми от този тип често се представят като агент, разположен в някакво обкръжение, който взима решения в зависимост от състоянието на това обкръжение.

На свой ред обкръжението реагира с награда или наказание, в зависимост от това колко уместно е било избраното действие.

Резултатът от обучението е оптимална стратегия за действие във всяка ситуация. Стратегията е оптимална ако успява да максимизира сумата от всички награди получени по време на изпълнението си.

Обучението чрез утвърждение може да се разглежда като комбинация от три елемента:

- Множество S, включващо всички състояния на обкръжението които агентът разпознава
- Множество A, включващо всички действия, които агентът може да извърши
- Множество R, включващо наградата която агентът може да получи
- s - моментно състояние на играта
- a - моментно действие, предприето от агента в играта
- r - моментна награда, която получава агента

При дадено състояние $s \in S$ и $a \in A$, агентът получава награда $r \in \mathbb{R}$. Тази функционална зависимост може да бъде записана по следния начин:

$$S \times A \to \mathbb{R}$$

Целта на агента е да открие такава стратегия $\pi : S \to A$, която максимизира сумарната награда получени на всяка стъпка: $R = r_0 + r_1 + r_2 + \ldots + r_N$

Обикновено наградата в близките състояния на обкръжението е много по-важна от тази на далечните бъдещи състояния. Тази зависимост може да бъде отразена чрез въвеждането на коефициент гама:

$$R = r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots + \gamma^N r_N$$

1.4 Съществуващи решения

Съществуват малък на брой игри, използващи обучение с утвърждение за създаване на реалистични противникови герой. Reece A. Boyd сравнява обикновения изкуствен интелект чрез Behavior Tree и изкуствен интелект трениран чрез обучение с утвърждение. Той контролира противниковия герой чрез табличен Q-learning. За своя

проект обаче той ползва визуално Blueprint програмиране, което макар и лесно за създаване се оказва изключително трудно за разбиране от други програмисти и за дебъгване. Тази дипломна работа е вдъхновена от неговия проект, но разширява функционалността чрез добавяне на мултиплейър функционалност и писане на целия алгоритъм в програмен код. Това ще улесни разбираемостта на алгоритъма и неговото бързодействие и мащабируемост.

OpenAI е изследователска лаборатория, която проучва и създава библиотеки, нови алгоритми и хардуер свързан с машинното самообучение чрез утвърждение. Те създават библиотеката OpenAI gym, която предоставя различни среди за трениране агента, който се обучава. Предоставят интерфейс, чрез които подаваме действие и получаваме новото състояние на играта и награда за нашето действие. Чрез тази библиотека не е нужно да разработваме сами environment и да имплементираме система за раздаване на награди. Използва се в комбинация с библиотеката Tensorflow-agents, с която изграждаме агента, взаимодействащ с тази среда. Тази библиотека има две лимитации:

Първата е факта, че те предлагат среди от стари двуизмерни игри, чиито графики могат да бъдат пренебрегнати и под внимание да бъдат взети само параметри като скорост, позиция, въртящ момент и т.н. Много трудно е чрез тази библиотека да бъде създадена персонализирана среда за нашите нужди.

Втората лимитация е използвания език. Библиотеката е написана с цел да бъде използвана като Python module. Езикът, който ползва Unreal Engine 4 е C++, което означава, че за да го ползваме бихме се нуждали или да създадем bridge между Unreal и Python, или да извикваме нов процес от нашата игра, които да ни дава информация действието на противниковия герой. Това обаче би трябвало да стане на отделна нишка, за да не блокираме играта при вземане на решение. По-бавното вземане на решения няма да ни позволи да имаме реалистичен и забавен геймплей.

В тази дипломна работа изграждаме средата за агента чрез Unreal Engine 4, което ни позволява по-голяма гъвкавост и възможност за промяна, както и обезсмисля използването на подобни библиотеки.

1.5 Общи положения

Тази дипломна работа ползва Unreal Engine 4.23 както в прекомпилирана бинарна версия, така и в ръчно компилирана и на места модифицирана версия. Причината за използване на този игрови двигател се обсъжда в следващата глава. От тук нататък за предварително компилирания игрови двигател ще се нарича “игрови двигател”, “Unreal Engine 4”, “Unreal Engine 4.23”. Ръчно компилирания и модифициран игрови двигател ще се нарича “компилиран игрови двигател”. Проекта ще сменя между двата, защото първия разполага с debugging symbols, докато другия позволява компилацията на сървърната част на играта. Когато се използват пътища в проекта под “/” ще се разбира текущата директория, в която се намира проекта.

2. Втора глава - избор на средства

2.1. Избор на технология за разработка

С разглеждането на различните игрови двигатели става ясно, че този, който ще бъде използван за разработката ще бъде Unreal Engine 4. С невероятната си графична мощ и изключително лесни за използване инструменти, Unreal е идеалният продукт, който ще позволи постигането на естетически приятна игра, която също има добре изградена функционалност и здрави игрови механики в своята опора. Освен това игровия двигател предлага и вече изработени текстури, които са за свободно ползване от потребителя. Unreal също може да се компилира за широк диапазон от устройства. От PC до Xbox One, PS4, Android, iOS, както и много от модерните VR платформи. Гъвкавостта на програмата й позволява да изпълнява един и същ код на множество различни машини. Unreal също притежава способността да използва DirectX 12, което му позволява да бъде оптимално ефективен в работата си, тъй като, в сравнение с предшественика си, DirectX 12 е по-оптимизиран и позволява още по-директен контрол върху хардуера, което означава, че едни и същи обекти се изобразяват на екрана за по-кратко време, когато се използва DX12. В новите ъпдейти на софтуера, Epic Games също добавят изцяло нов инструмент за създаване на анимации по време на игра, както и подобрения върху изкуствения интелект на героите.

2.2. Избор на език за програмиране

Unreal Engine предлага средство, посредством което, програмистът може изцяло да изгради логиката на играта без да напише и един ред код. Това средство се нарича Blueprint. Blueprint е система, посредством която се създават функции и скриптове, използвайки визуални елементи, които по време на изпълнение се интерпретират и изпълняват от engine-а. Те могат също така да бъдат предварително транслирани до C++ код, който да бъде компилиран с останалата част от играта. Целта на Blueprint, което нататък ще бъде съкращавано като BP, е да мимикира структурата и поведението на езика за програмиране C++. Позволява дефинирането на структури, класове, наследственост между класове, променливи на класовете, както и дефинирането на цели функции. Всяка функция има някакъв вид начало. Някакъв елемент, който действа като начало и при някакво условие извиква последователното изпълнение на останалите компоненти на алгоритъма. Функцията се „чертае“, като се свързват различни функции помежду си и на някои от тях се подават променливите, които са нужни за правилното им изпълнение. Unreal, естествено, позволява и програмирането на езика C++, което от една страна е добър вариант, тъй като дава по-добър и прецизен контрол върху точната структура на класовете и тяхното взаимодействие. От друга страна, BP позволява постигането на същите резултати, но плюс в случая е факта, че програмистът не трябва да се тревожи върху лексиката на езика, а върху самия алгоритъм на играта. Недостатъкът на BP е факта, че при по-сложен алгоритъм визуалното програмиране, което ни предоставя е недостатъчно лесно за разбиране, промяна и дебъгване. Поради тази причина тази дипломна работа се стреми максимално да се придържа към използването на C++ за имплементиране на всички части от играта.

C++ е неспециализиран език за програмиране от високо ниво. Той е обектно-ориентиран език със статични типове. От 1990-те, C++ е един от най-популярните комерсиални езици за програмиране. C++ съдържа вградена в езика поддръжка на обектно-ориентирано програмиране. В C++ са добавени класове, множествено наследяване, виртуални функции, overloading, шаблони (templates),

обработка на изключения (exceptions) и вградени оператори за работа с динамична памет. Единствената част, в която има повече смисъл да бъде използван BP е потребителския интерфейс. Unreal Engine позволява много удобна разработка без използване на специалния език за разработка на потребителски интерфейс - Slate.

Програмирайки на C++ ние имаме възможността да представяме по-лесно взаимоотношенията и йерархията между класовете в играта.

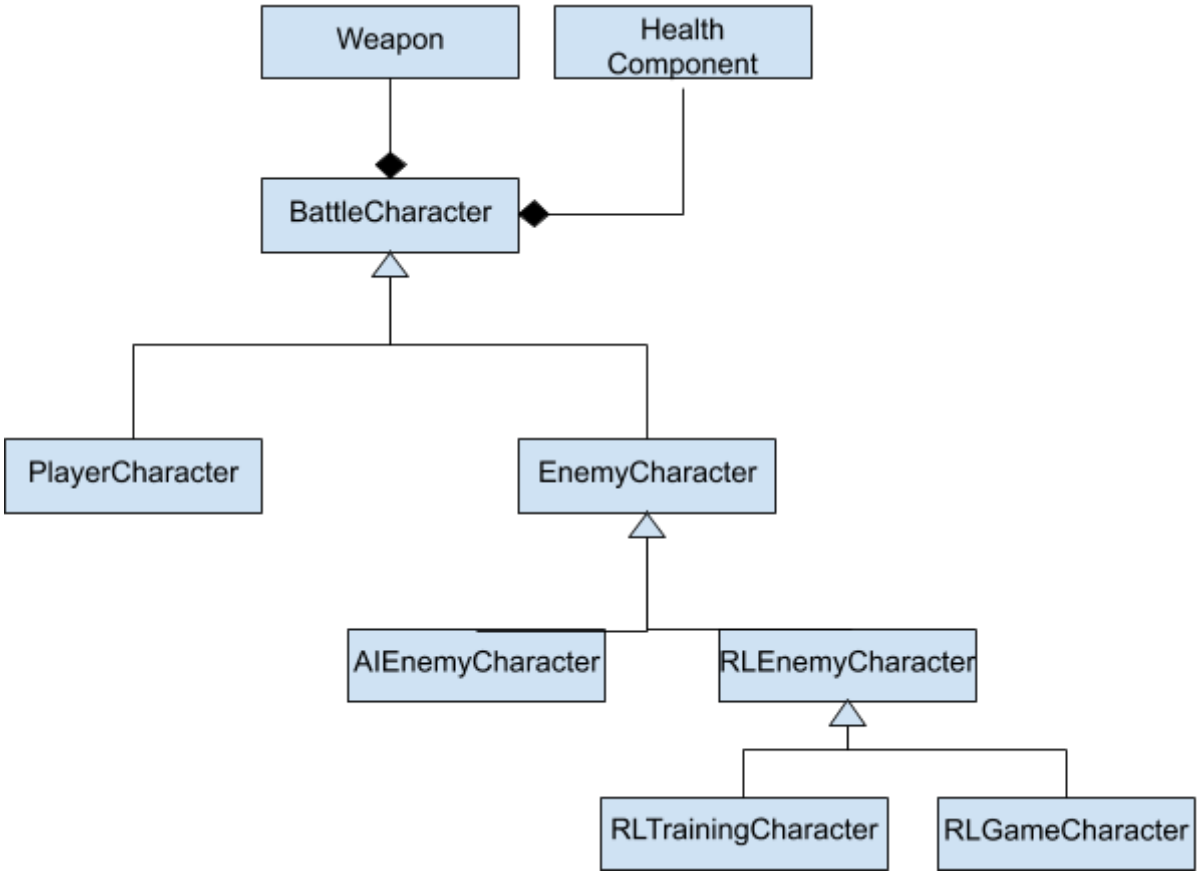
2.3. Избор на интегрирана среда за разработка

Unreal Engine 4 поддържа различни среди за разработка - Visual Studio Code, Visual Studio 2015, Visual Studio 2017, Visual Studio 2019 и CLion. Visual Studio Code не разполага с пълната функционалност на останалите среди за разработка без добавяне на голям брой плъгини. Той е по-скоро разширен текстов редактор отколкото среда за разработка. Visual Studio 2015 е спрян от поддръжка за версии на Unreal Engine 4.22 и по-високи. Clion има сравнително лоша интеграция, поради неофициалната му поддръжка на игровия двигател. Това оставя като избор само Visual Studio 2017 и 2019. От опит беше установено, че Visual Studio 2017 е по-надеждна и по-бързо зареждаща среда за разработка. Също така компанията Whole Tomato предлага плъгин към Visual Studio наречен “Visual Assist”, който предоставя по-добро индексиране, автоматично допълване на макроси използвани от Unreal, както и автоматично генериране на дефиниции на функция от нейната декларация. За това за разработката на тази дипломна работа бе използван Visual Studio 2017

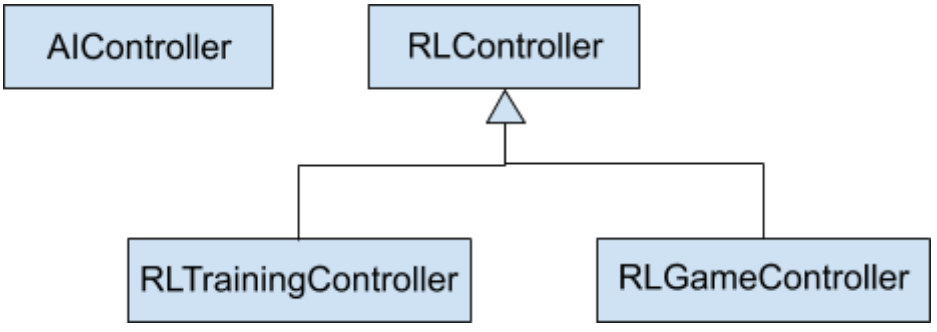
2.4. Схема на структурата на кода

В основата на йерархията е базовият клас за играч “BattleCharacter”. Той предоставя функционалност за взаимодействие с оръжия, умиране и други основни функции, споделяни от всеки играч, Композира различни главни функционалности, като базовия клас за оръжие и компонент за кръвта на героя. От него наследяват класовете “PlayerCharacter” и “EnemyCharacter”, които допълват функционалността на

базовия клас. Противниковия клас се разделя на “AIEnemyCharacter” и “RLEnemyCharacter”. Вторият е допълнително разделен на “RLTrainingCharacter” и “RLGameCharacter”:



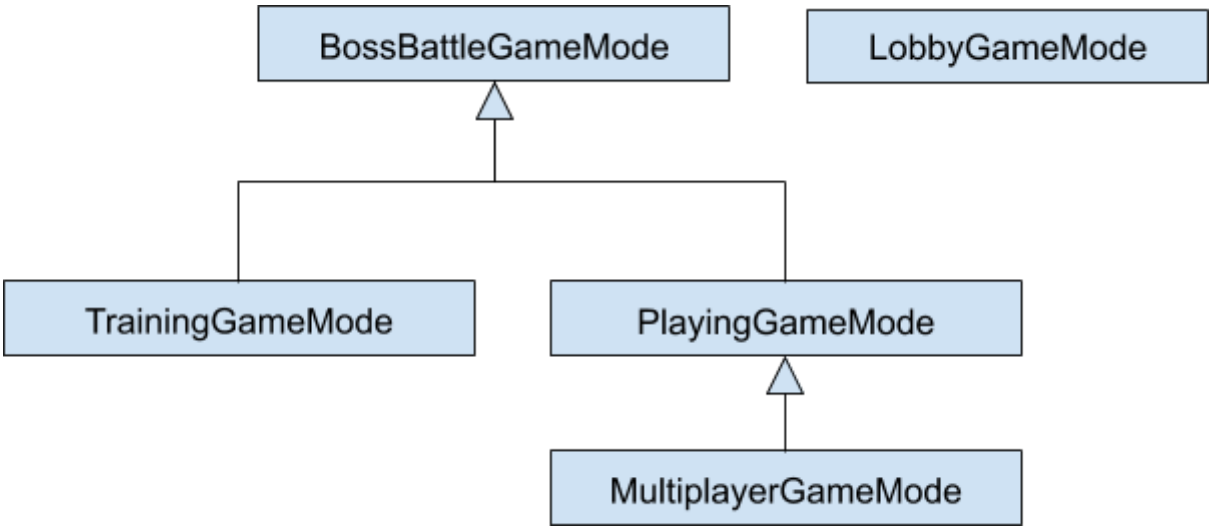
Всеки играч е контролиран от контролер клас. При тях също има йерархия. Има ‘EnemyAIController’, който управлява противниците, имплементирани чрез автомат на състоянията. Имплементирани са и базов клас “RLController”, който включва базовата функционалност за противник, използващ машинно самообучение с утвърждение. От него наследяват двата контролера “RLTrainingController”, който служи за трениране, и “RLGameController”, който ползва натренираната таблица за същинската игра.



Потребителският интерфейс е съставен от плоска йерархия от “HUD” и “Widget” класове.

За анимацията се грижи единствен C++ клас “CharacterAnimInstance”, което значи, че анимацията на всички играчи е еднаква.

Режимът на игра се диктува от “GameMode” класовете. Имплементиран е базов клас “BossBattleGameMode”, който декларира виртуални функции, които имат различна имплементация в различните наследяващи класове:



3. Трета глава - реализация

3.1. Разработка на различни видове оръжия

В основата на йерархията на Unreal стоят класовете **UObject** и **Actor**:

UObject е основния клас в reflection системата на Unreal. Чрез макроси Unreal Header Tool (съкратено UHT) обгражда C++ класовете и ги прави на класове, регистрирани в различните системи на игровия двигател. Unreal Engine знае за всеки клас, който наследява UObject. По конвенциите на Unreal, класовете, които наследяват UObject, но не наследяват Actor започват с буквата “U”.

Actor е клас, който стои на сцената на играта, независимо от това дали той може да бъде видян или не. Actor наследява UObject и класовете, които наследяват Actor по конвенция започват с буквата “A”.

Всички енумерации започват с буквата “E”, а помощните класове, които не са регистриране в UHT започват с буквата “F”.

Всеки Actor може да бъде синхронизиран така, че неговото движение в пространството на локалната машина да бъде отчетено и при другите играчи. Това става автоматично от Replication системата на Unreal Engine, като съответния Actor се маркира като “replicated”:

```
ABattleCharacter::ABattleCharacter()
{
	// Set this character to call Tick() every frame.  You can
turn this off to improve performance if you don't need it.
	PrimaryActorTick.bCanEverTick = true;

	DeathSoundComponent =
```

```
CreateDefaultSubobject<UAudioComponent>("DeathSoundAudioComponent"
);
	DeathSoundComponent->bAutoActivate = false;

	HealthComponent =
CreateDefaultSubobject<UHealthComponent>("HealthComponent");
	HealthComponent->OnDeath.AddDynamic(this,
&ABattleCharacter::Die);
	HealthComponent->SetIsReplicated(true);

	CharacterMovementComponent =
Cast<UCharacterMovementComponent>(GetCharacterMovement());
	if (validate(IsValid(CharacterMovementComponent)) == false)
return;

CharacterMovementComponent->GetNavAgentPropertiesRef().bCanCrouch
= true;
	CharacterMovementComponent->bCrouchMaintainsBaseLocation =
true;
	CharacterMovementComponent->SetIsReplicated(true);


	USkeletalMeshComponent* SkeletalMesh = GetMesh();
	if (validate(IsValid(SkeletalMesh)) == false) { return; }

}
```

Всеки един Actor има 3 основни метода: конструктор, Tick метод и BeginPlay метод. Конструктора задава начални стойности на всички нужни параметри, оказва дали метода Tick ще се вика или не. Не винаги е нужно даден Actor да вика Tick метода, затова е добра практика ръчно да се спира извикването му от игровия двигател. В конструктора се създават и компонентите на нашия Actor. В Unreal се използва композиране, поради природата на самата игра и нейните елементи. Един играч или противник има отделен компонент за движение, за звук, за управление на събития свързани с кръвта му, за анимациите и т.н.

За дебъгване бе създадено макрото “validate”. То служи за валидиране на дадено булево твърдение. Когато то е вярно функцията не прави нищо. Когато то е грешно функцията пише в логовете на коя линия, в коя функция и на кой обект е възникнал

проблема, а също така и кой е обекта, който извиква проблемната функцията. Грешките от “validate” макрото изглеждат по следния начин:

LogTemp: Error: caller: BP_BattleHUD_C_0 executor: BattleHUD at: DrawHUD:63 (IsValid(PlayerCharacter)) == false

Така разбираме точно къде е проблемът. Често се използва в комбинация с функцията “IsValid”. Тя се ползва само за класове, които наследяват UObject. Тя проверява даден указател против nullptr и проверява дали обекта, към който сочи указателя не е маркиран от Unreal Garbage collector за изтриване.

Фундаментална част от всяка стрелкова игра са различните видове оръжия. Разработена е система за създаване на различни оръжия, вземането и оставянето на оръжията, презареждане и съпътстващите го анимации, синхронизация между играчите. Всеки играч може да вземе оръжие от земята. Оръжията и проектилите са синхронизирани между всички играчи, което значи, че когато един играч вземе или стреля с оръжие, това стреляне ще бъде показано и на другите играчи.

```
void ABattleCharacter::PickGun(class AGun* NewGun)
{
	if (validate(IsValid(NewGun)) == false) { return; }
	if (validate(IsValid(CharacterMesh)) == false) { return; }
	if (validate(CharacterMesh->DoesSocketExist("GunSocket")) ==
false) { return; }

	Gun = NewGun;

	USkeletalMeshComponent* GunMesh =
Gun->FindComponentByClass<USkeletalMeshComponent>();
	if (validate(IsValid(GunMesh)) == false) return;


	Gun->OnPick();

	GunMesh->AttachToComponent(CharacterMesh,
FAttachmentTransformRules::SnapToTargetNotIncludingScale,
"GunSocket");
}
```

Вземането на оръжие става по следния начин. Валидира се съществуването на ново оръжие, герой и сокет на героя, за който да бъде закачено то. След което се вика функция, която променя състоянието на оръжието от свободно към захванато от героя. Всяко оръжие има 2 състояния: взето и оставено. При оставено състояние то се върти около Z оста си, за да привлече вниманието на играча. При взето състояние неговата ротация се нулира, за да може да бъде правилно прикрепено към съответния сокет.

```
void AGun::OnPick()
{
	if (HasAuthority() == false) return;
	SetActorRotation(FRotator(0, 0, 0));
	GunState = EGunState::Picked;
}
```

Накрая оръжието се прикрепя към сокета на скелета на героя, местоположението на който е предварително зададено. Вземането на оръжие може да стане само на сървърната част. Това се гарантира чрез функцията “HasAuthority”

```
void ABattleCharacter::DropGun()
{
	if (validate(IsValid(Gun)) == false) return;
Gun->DetachFromActor(FDetachmentTransformRules(EDetachmentRule::Ke
epWorld, EDetachmentRule::KeepWorld, EDetachmentRule::KeepWorld,
false));
	Gun->OnDrop();
	Gun = nullptr;
}
```

Хвърлянето на текущото оръжие става по подобен начин. След валидиране за съществуващо такова, оръжието се отделя от сокета на скелета на героя, вика се функция променяща състоянието на оръжието и текущото оръжие става nullptr

```
void AGun::OnDrop()
{
	if (HasAuthority() == false) return;
	SetActorRotation(FRotator(0, 0, 0));
	FVector ActorLocation = GetActorLocation();
	SetActorLocation(FVector(ActorLocation.X, ActorLocation.Y,
100));
	GunState = EGunState::Dropped;
}
```

За синхронизация на започването и спирането на стрелба на всички клиентски машини и за сървърна валидация за право на стрелба се използва клиент-сървър модела и RPC.

Клиент-сървър е модел, при който клиентът изпраща заявка, която показва желанието му да извърши дадено действие. Сървърът валидира параметрите на заявката и разрешава или отказва действието. Действието се изпълнява първо на сървърната инстанция и в последствие бива репликирано до всички клиенти. В Unreal това става чрез специални макроси:

```
	UFUNCTION(Server, Reliable, WithValidation)
	void ServerStartFiring();
```

```
        bool ServerStartFiring_Validate();
        void ServerStartFiring_Implementation();

        UFUNCTION(Server, Reliable, WithValidation)
        void ServerStopFiring();
        bool ServerStopFiring_Validate();
        void ServerStopFiring_Implementation();

        UFUNCTION(Server, Reliable, WithValidation)
        void ServerStartReloading();
        bool ServerStartReloading_Validate();
        void ServerStartReloading_Implementation();

        UFUNCTION(Server, Reliable, WithValidation,
BlueprintCallable)
        void ServerFinishReloading();
        bool ServerFinishReloading_Validate();
        void ServerFinishReloading_Implementation();

        UFUNCTION(Server, Reliable, WithValidation)
        void ServerInteractWithWeapon();
        bool ServerInteractWithWeapon_Validate();
        void ServerInteractWithWeapon_Implementation();
```

UFUNCTION макрото добавя код, който добавя функцията към Unreal reflection системата. То може да приема различни ключови думи. Ключовата дума “Server” оказва, че от който и клиент да бъде извикана функцията, тя ще бъде изпълнена само на сървъра. Всички функции, които започват със думата “Server” са обвиващи функции на тези, които не ползват тази дума. Единствената разлика, е че за тях е гарантирано да се изпълнят само и единствено на сървърната инстанция на играта. Ключовата дума “Reliable” оказва, че функцията със сигурност ще бъде изпълнена. Функциите могат да бъдат обозначени и като “Unreliable”, което означава, че те може да не бъдат изпълнени при натоварена мрежа. Ключовата дума “WithValidation” оказва, че преди да бъде изпълнена основната функция, ще бъде извикана функция за валидация. Ако

функцията за валидация завърши успешно, основната функция ще бъде изпълнена. В противен случай ще бъде пренебрегната.

```
UFUNCTION(NetMulticast, Reliable, WithValidation)
void MulticastStartReloading();
bool MulticastStartReloading_Validate();
void MulticastStartReloading_Implementation();

UFUNCTION(NetMulticast, Reliable, WithValidation)
void MulticastFinishReloading();
bool MulticastFinishReloading_Validate();
void MulticastFinishReloading_Implementation();
```

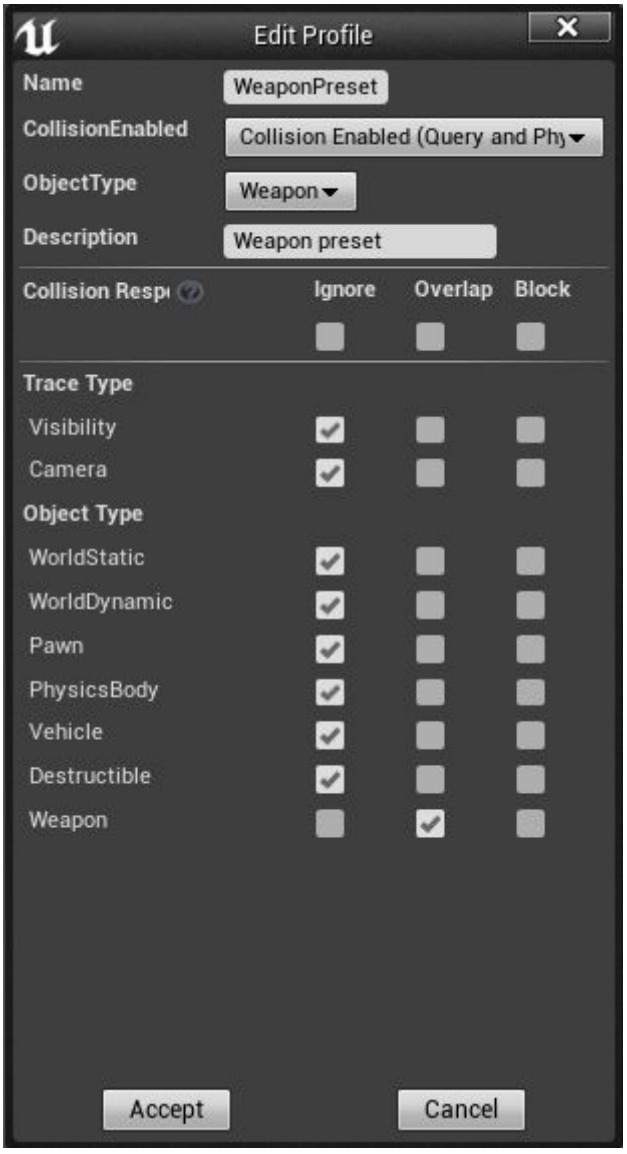
След като бъдат изпълнени на сървъра се викат функциите, които се изпълняват на всички клиентски машини. Ключовата дума “NetMulticast” оказва на Unreal, че функцията ще бъде извикана само веднъж от сървъра, но трябва да бъде изпълнена за всички клиенти.

Цялостната интеракция с оръжието е по-сложна:

```
void ABattleCharacter::InteractWithWeapon()
{
	AGun* OldGun = nullptr;
	if (IsValid(Gun)) {
		OldGun = Gun;
	}
	//drop the current gun if you have one
	if (IsValid(Gun)) {
		DropGun();
	}

	//find gun in a sphere around player with given distance
	UWorld* World = GetWorld();
	if (validate(IsValid(World)) == false) return;

	DrawDebugSphere(World, GetActorLocation(),
InteractionDistance, 50, FColor::Yellow, false, 1.0f);

	TArray<TEnumAsByte<EObjectTypeQuery>> ObjectTypes;
	ObjectTypes.Add(EObjectTypeQuery::ObjectTypeQuery7);
```

```
	TArray<AActor*> OverlappedActors;
	TArray<AActor*> ActorsToIgnore;

	UKismetSystemLibrary::SphereOverlapActors(World,
GetActorLocation(), InteractionDistance, ObjectTypes,
AGun::StaticClass(), ActorsToIgnore, OverlappedActors);
	if (validate(OverlappedActors.Num() != 0) == false) return;
	for (auto OverlappedActor : OverlappedActors) {

		AGun* NewGun = Cast<AGun>(OverlappedActor);
		if (validate(IsValid(NewGun)) == false) continue;
		if (NewGun == OldGun) continue;

		PickGun(NewGun);
		break;
	}

}
```

Първо се изхвърля текущото оръжие, ако има такова. След това се проверява в сфера около героя дали има оръжие от този тип и ако има такова то се взема. За оръжията бе създаден нов Object channel, както и специален “WeaponPreset”, който да реагира само при повикване от код.



Фиг. 3-1 collision preset за оръжията.

Презареждането на всяко оръжие работи по следния начин. При изчерпване на текущите амуниции или при натискане на бутона за презареждане (Фиг. 4-1). Във втория случай се създава събитие “Reload”. Това събитие бива свързано към функция, която да бъде изпълнена:

```
PlayerInputComponent->BindAction("Reload", IE_Pressed, this,
&APlayerCharacter::ServerStartReloading);
```

При изчерпване на амунициите логиката на играта вика директно функцията “ServerStartReloading”. Първо се извиква функцията на сървъра:

```
void ABattleCharacter::ServerStartReloading_Implementation()
{
	MulticastStartReloading();


	FTimerHandle TimerHandle;
	GetWorldTimerManager().SetTimer(TimerHandle, this,
&ABattleCharacter::UpdateGun, 2.16666, false);


	FTimerHandle TimerHandle2;
	GetWorldTimerManager().SetTimer(TimerHandle2, this,
&ABattleCharacter::ServerFinishReloading, 2.16666, false);

}
```

Тя създава на сървъра таймери за спиране на презареждането и за промяна на текущите амуниции на оръжието. Функциите “ServerStartReloading” и “ServerFinishReloading” се грижат за това анимацията за презареждане да бъде своевременно показана на всички клиенти. Промянта и контрола на амуниците се случва само на сървъра и от функцията “UpdateGun”:

```
void ABattleCharacter::UpdateGun()
{
	if (validate(IsValid(Gun)) == false) return;
	if (HasAuthority() == false) return;
	Gun->FinishReload();

}
```

Функцията “ServerStartReloading” освен това извиква и функцията за презареждане на всички клиентски машини:

```
void ABattleCharacter::MulticastStartReloading_Implementation()
{
	StartReloading();
}
```

Която започва анимацията за презареждане. Това е често срещан начин за синхронизиране на функция между всички крайни устройства. Самата функция за презареждане е сравнително проста:

```
void ABattleCharacter::StartReloading()
{
	if (bReloadingAllowed == false) { return; }
	if (validate(IsValid(Gun)) == false) { return; }

	bReloadingAllowed = false;
	CharacterAnimation->SetReloading(true);
}
```

След валидации тя забранява повторното си извикване за периода, в който се изпълнява, след което вика функция, която изпълнява анимацията за презареждане.

Функцията “ServerStartReloading” стартира анимацията, чака тя да завърши, след което вика сървърната функция за завършено презареждане. Тя работи аналогично на тази за стартиране на презареждането - извиква “Multicast” функцията, която от своя страна извиква същинската функция “FinishReloading” на всички клиентски машини:

```
void ABattleCharacter::FinishReloading()
{
	if (validate(IsValid(Gun)) == false) return;

	bReloadingAllowed = true;

	CharacterAnimation->SetReloading(false);
}
```

Логиката за презареждане на оръжието е изнесена в интерфейса на самото оръжие, тъй като промяната на количеството амуниции е свързано по-скоро с него, отколкото със самия герой. Разпределени са отговорностите на класовете така, че всеки клас да знае само са тези член-променливи, които трябва да принадлежат на него. Така получаваме по-добра енкапсулация на класовете и по-разбираем публичен интерфейс. Разбира се, тази функция може да бъде изпълнена само на сървъра, защото целия контрол на оръжията може да бъде управляван само от него:

```
void AGun::FinishReload()
```

```
{
	if (HasAuthority() == false) return;
	if (validate(WeaponData != nullptr) == false) return;
	if (WeaponData->bInfiniteAmmo) {
		CurrentClipAmmo = WeaponData->MaxClipAmmo;
	}
	else {
		int MissingAmmo = WeaponData->MaxClipAmmo -
CurrentClipAmmo;
		int AmmoToTransfer = FMath::Min(MissingAmmo,
CurrentAmmo);
		CurrentClipAmmo += AmmoToTransfer;
		CurrentAmmo -= AmmoToTransfer;
	}
	if (bTriggerPressed) {
		PullTrigger();
	}
}
```

Самите оръжия работят по сравнително прост начин. При натискане на контрола на стреляне се стартира таймер, който изпълнява функцията за стреляне. При отпускане на контрола този таймер се нулира.

```
	PlayerInputComponent->BindAction("Fire", IE_Pressed, this,
&APlayerCharacter::ServerStartFiring);
	PlayerInputComponent->BindAction("Fire", IE_Released, this,
&APlayerCharacter::ServerStopFiring);
```

Функцията “ServerStartFiring” се изпълнява на сървъра и извиква обикновената функция “StartFiring”. Тя от своя страна проверява за оръжие и при наличие на такова вика неговата функция за начало на стрелянето. Така можем да имаме виртуална функция за стреляне и различни видове оръжия, които я имплементират по различен начин:

```
void ABattleCharacter::StartFiring() {
	if (validate(IsValid(Gun)) == false) return;
	Gun->PullTrigger();
}
```

```
void AGun::PullTrigger() {
	if (validate(WeaponData != nullptr) == false) return;

	GetWorldTimerManager().SetTimer(
		FireTimerHandle,
		this,
		&AGun::Fire,
		1 / WeaponData->FireRate,
		true,
		WeaponData->FireDelay
	);
	bTriggerPressed = true;
}
```

Спирането на стрелянето работи по аналогичен начин:

```
void AGun::ReleaseTrigger() {
	bTriggerPressed = false;
	GetWorldTimerManager().ClearTimer(FireTimerHandle);
}
```

Основната вътрешна функция на всяко оръжие е функцията “Fire”. Тя проверява за наличие на амуниции. При наличие на такива изстрелва проектил, пуска анимацията за стрелба и намалява амуниците със съответния брой. При липса на амуниции тя започва презареждането на оръжието. Това е функцията, която контролира основната логика на всяко оръжие:

```
void AGun::Fire() {
	if (HasAuthority() == false) return;
	 if (validate(WeaponData != nullptr) == false) return;

	if (HasAmmo(WeaponData->BulletsPerShot)) {
		PlayFireSound();
		for (int i = 0; i < WeaponData->BulletsPerShot; i++) {
			SpawnProjectile();
		}

	    OnFire.Broadcast();
		CurrentClipAmmo -= WeaponData->BulletsPerShot;

		if (IsValid(WeaponData->FireAnimation.Get())) {
			UAnimSequence* FireAnimation =
```

```
Cast<UAnimSequence>(WeaponData->FireAnimation.Get());

                if (validate(IsValid(FireAnimation)) == false)
return;

SkeletalMeshComponent->PlayAnimation(FireAnimation, false);

            }
      }
      else {
            GetWorldTimerManager().ClearTimer(FireTimerHandle);
            //TODO: ServerStartReload
            StartReload();
      }
}
```

Функцията “SpawnProjectile” създава нов проектил, задава началните му параметри и началната му ротация. Тя може да бъде контролната ротация на героя(посоката в която гледа играча) или посоката напред на оръжието (предварително зададена). Към тази ротация се добавя случайно отместване в даден диапазон, който се задава при създаване на оръжието.

```
void AGun::SpawnProjectile() {
      if (HasAuthority() == false) return;
      if (validate(WeaponData != nullptr) == false) return;
      if (validate(IsValid(ProjectileSpawnTransform)) == false) {
return; }

      if (validate(IsValid(WeaponData->ProjectileTemplate.Get()))
== false) { return; }

      UWorld* World = GetWorld();
      if (validate(IsValid(World)) == false) { return; }

      ABattleCharacter* Parent =
Cast<ABattleCharacter>(GetAttachParentActor());
      if (validate(IsValid(Parent)) == false) { return; }
```

```
	FActorSpawnParameters SpawnParameters =
FActorSpawnParameters();
	SpawnParameters.Owner = Parent;
	SpawnParameters.Instigator = Parent->GetInstigator();
	SpawnParameters.SpawnCollisionHandlingOverride =
ESpawnActorCollisionHandlingMethod::AdjustIfPossibleButAlwaysSpawn
;

	FRotator ForwardRotator;
	switch (SpawnRotationMode) {
		case EProjectileSpawnRotation::ControlRotation:
			ForwardRotator = Parent->GetControlRotation();
			break;
		case EProjectileSpawnRotation::GunForwardVector:
			ForwardRotator =
ProjectileSpawnTransform->GetComponentRotation();
			break;
		default:
			validate(false);
	}

	const int RandomSeed = FMath::Rand();
	FRandomStream RandomStream(RandomSeed);

	FVector ForwardDirection = ForwardRotator.Vector();
	const float ConeHalfAngleRad =
FMath::DegreesToRadians(WeaponData->MaxSpreadAngle * 0.5); // we
want the half cone angle in radians
	FRotator ShootRotation =
RandomStream.VRandCone(ForwardDirection,
ConeHalfAngleRad).Rotation();

	FVector SpawnLocation =
ProjectileSpawnTransform->GetComponentLocation();



	AProjectile* Projectile = World->SpawnActor<AProjectile>(
		WeaponData->ProjectileTemplate.Get(),
		SpawnLocation,
		ShootRotation,
```

```
            SpawnParameters
        );

        validate(IsValid(Projectile));
}
```

За всичките параметри на оръжието се ползва DataTable. Това е таблица, която се пази като файл на файловата система. Тя се сериализира до csv или json формат. В нея се записва информацията за всички оръжия, като първичен ключ е името на оръжието. В кода тя се десериализира до клас, дефиниран от нас. Предоставени са функции за търсене на специфичен ред от таблицата. Това е алтернативата на ползването на бази данни в Unreal. Структурата на един ред, след като бъде десеризлизиран изглежда по следния начин:

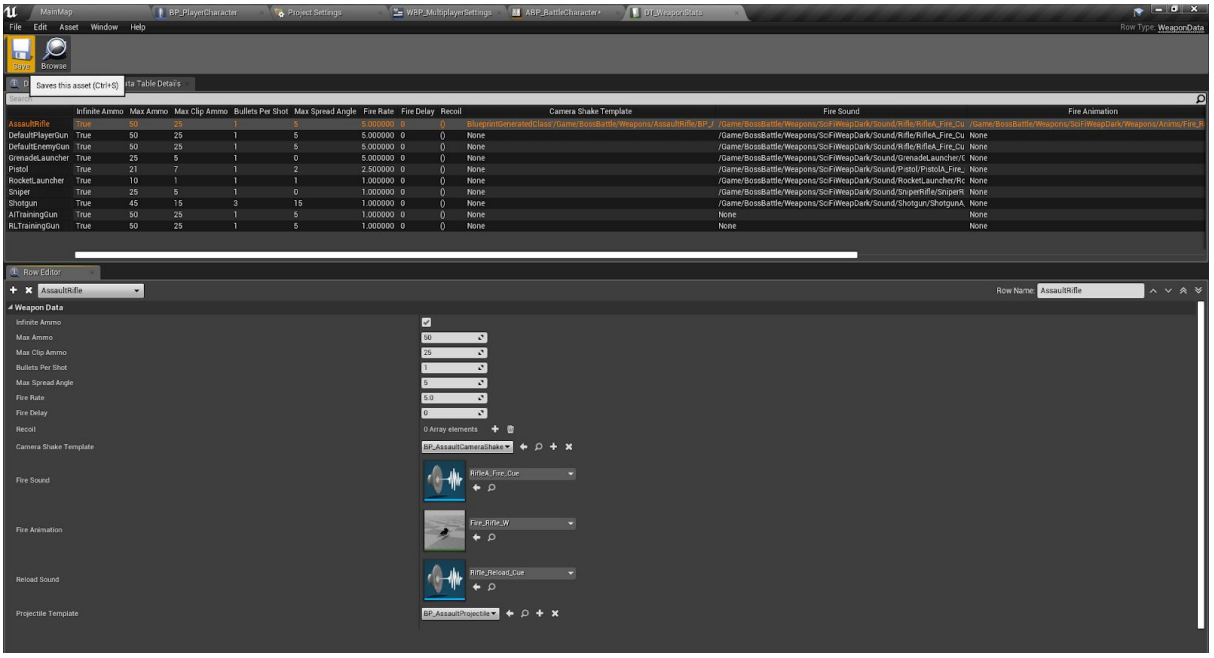
```
USTRUCT(BlueprintType)
struct FWeaponData : public FTableRowBase
{
	GENERATED_USTRUCT_BODY()
	// whether or not the gun has infinite ammo
	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite)
	bool bInfiniteAmmo;

	// max ammo outside the clip
	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite)
	int MaxAmmo;

	// max clip ammo
	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite)
	int MaxClipAmmo;

	//number of bullets fired per shot
	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite)
	int BulletsPerShot;

	//max angle the bullets can be spread in.
	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite)
	int MaxSpreadAngle;

	//number of times the weapon can fire(per second)
	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite)
	float FireRate;

	//delay in seconds that the weapon will wait before firing
	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite)
	int FireDelay;

	//array of predefined rotations added to the original bullet
rotation for every next shot.
	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	TArray<FRotator> Recoil;

	//camera shake class
	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite)
	TSubclassOf<class UCameraShake> CameraShakeTemplate;
```

```
	//sound to be played on weapon fire
	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite)
	TAssetPtr<class USoundCue> FireSound;

	//animation to be played on fire
	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite)
	TAssetPtr<class UAnimSequence> FireAnimation;

	//sound to be played on weapon reload
	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite)
	TAssetPtr<class USoundCue> ReloadSound;

	//type of projectile to use
	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite)
	TSubclassOf<class AProjectile> ProjectileTemplate;
};
```

Това са всички параметри, които могат да бъдат зададени на едно оръжие. Всяко оръжие има указател към тази таблица и намира своите параметри по следния начин:

```
	if (validate(IsValid(WeaponDataTable)) == false) return;
	FString ContextString(TEXT("GENERAL"));
	WeaponData =
WeaponDataTable->FindRow<FWeaponData>(WeaponName, ContextString);
	if (validate(WeaponData != nullptr) == false) return;
```

След това ги достъпва чрез член-променливата "WeaponData".



Фиг. 3-3 WeaponData Data Table

Таблицата се попълва и на оръжието се подава указател към нея. В таблицата трябва да има ред с име, което да съответства на името на оръжието.

Фиг. 3-4 Задаване на Weapon Data Table на оръжието

Файлът е именуван така, защото целим да следваме наръчник за стила и именуването на Michael Allar (виж използваната литература). Той не е официално написан от Unreal, но поради липсата на такъв този е най-доброто, което е предложено за публично ползване. Той предоставя конвенции за именуване на различни файлове, така че те да бъдат лесно разпознаваеми и разбрани. Този наръчник се е превърнал в индустриален стандарт и всички програмисти, дизайнери и хора, ползващи Unreal знаят за него и неговите означения.

3.2. Добавяне на анимации за различни действия на играчите и противниците

Всеки герой в играта има Skeletal Mesh Component, върху който могат да се ползват различни анимации. Имплементирани са такива за движение, клякане, презареждане и умиране. Класът “UAnimInstance” е базовият клас за Animation Blueprint. Създаден бе клас, който наследява от него и събира състоянието на играча на всеки “Tick” евент:

```
void UCharacterAnimInstance::NativeUpdateAnimation(float
DeltaTimeX) {
	Super::NativeUpdateAnimation(DeltaTimeX);
	if (validate(IsValid(CharacterPawn)) == false) { return; }
	FVector Velocity = CharacterPawn->GetVelocity();
	Speed = Velocity.Size();
	MovementDirection = CalculateDirection(Velocity,
CharacterPawn->GetControlRotation());
	UPawnMovementComponent* CharacterMovement =
CharacterPawn->GetMovementComponent();
	if (validate(IsValid(CharacterMovement)) == false) return;
	bInAir = CharacterMovement->IsFalling();
}
```

Състоянието при клякане и презареждане се задава чрез сеттър функция (setter), която е публична:

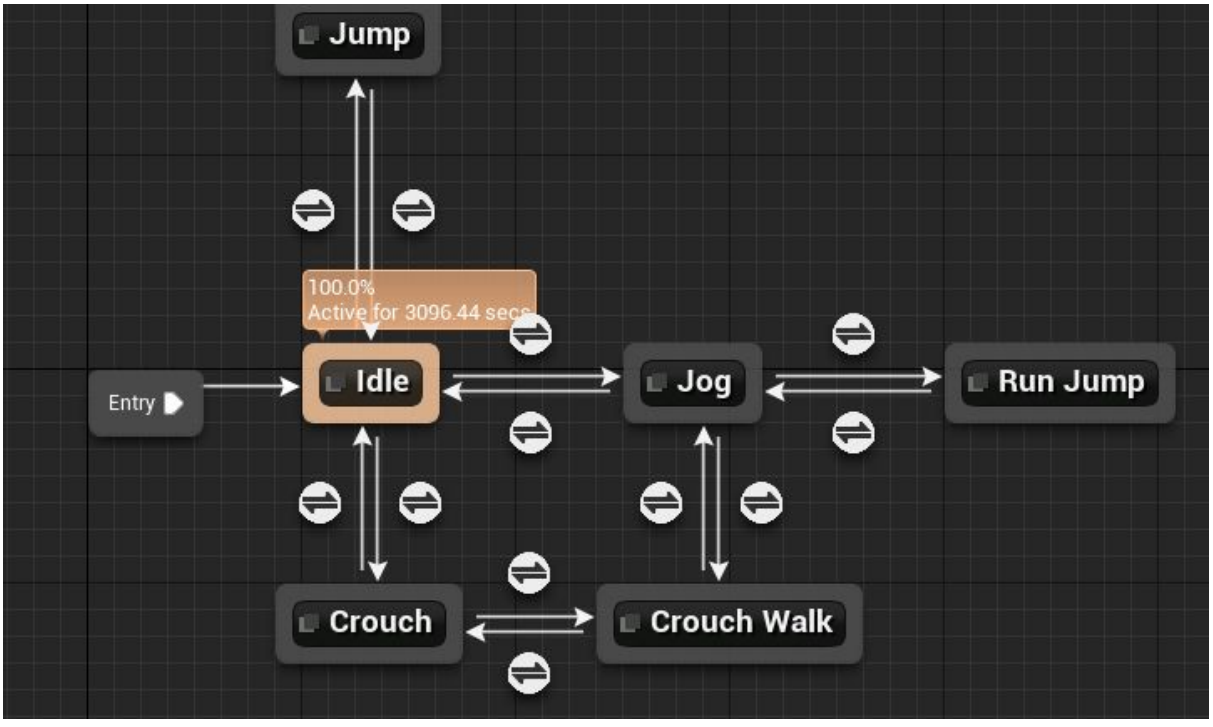
```
void UCharacterAnimInstance::SetCrouching(bool State)
{
	UE_LOG(LogTemp, Warning,
TEXT("UCharacterAnimInstance::SetCrouching"))
	bCrouching = State;
}


void UCharacterAnimInstance::SetReloading(bool State) {
	if (State) {
		PreviousState = AnimationState;
		AnimationState = EAnimationState::Reloading;
	}
```

```
        else {
                AnimationState = PreviousState;
        }
}
```

Състоянията на всеки герой са следните: Ходещ, презареждащ и мъртъв. Състоянието за презареждане е добавено за опростяване на кода, но всеки герой може да се движи и презарежда едновременно. Състоянията са представени чрез следната енумерация:

```
UENUM()
enum class EAnimationState : uint8 {
        Walking,
        Reloading,
        Dead
};
```

Blueprint наследява от този клас и ползва наследените променливи, за да изчисли финалната анимация. Основния state machine е Locomotion state machine. Той избира анимация на база на текущото състояние и условията за преминаване към друго състояние. Резултатната анимация от него се запазва в кеш наречен “LocomotionCache”:

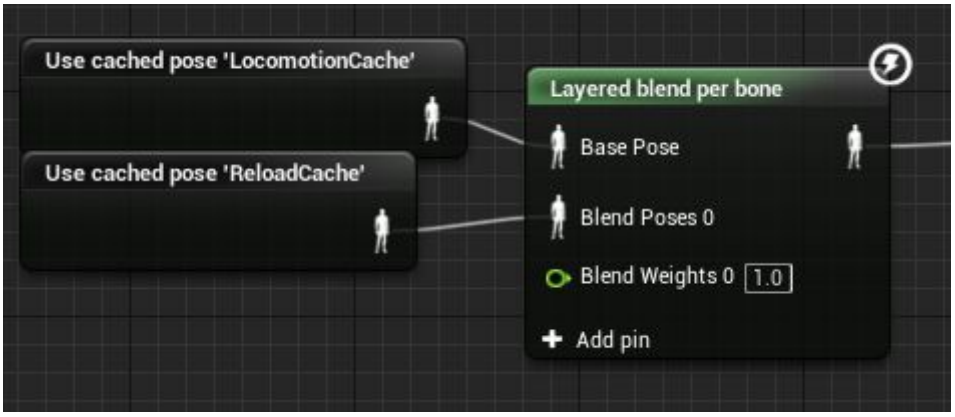


Фиг. 3-5 Релации между отделните състояния в Locomotion state machine

Кеш представлява запазена анимация (като променлива) която може да бъде използвана по-късно:

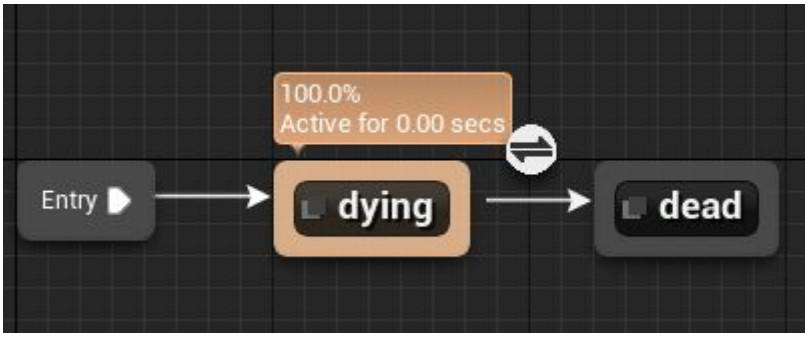
Фиг. 3-6. Кеш за презареждане

Към кеша “Locomotion” прилагаме кеша на анимацията за презареждане на оръжие. В нея обаче краката на героя не се движат. Това би било проблем, когато искаме да презареждаме в движение. Поради тази причина използваме функцията “Layered blend per bone”, която приема две анимации и съчетава движението в тях. Задаваме кост, която да служи като начална за изпълнение на втората анимация. Избираме кост на кръста на героя. Над нея се изпълнява анимацията за презареждане, под нея тази за движение. Крайния резултат е анимация, в която героя може едновременно да се движи и да презарежда.



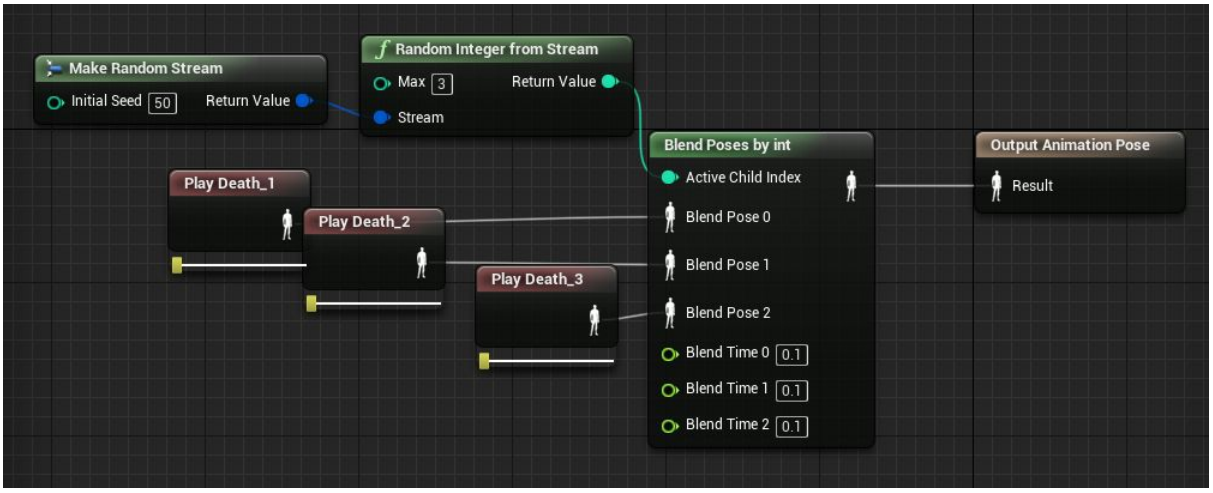
Фиг. 3-7. Комбиниране на две анимации

При липса на действиене не се нуждаем от изчисления на различни състояния. За това просто изпълняваме единична анимация.



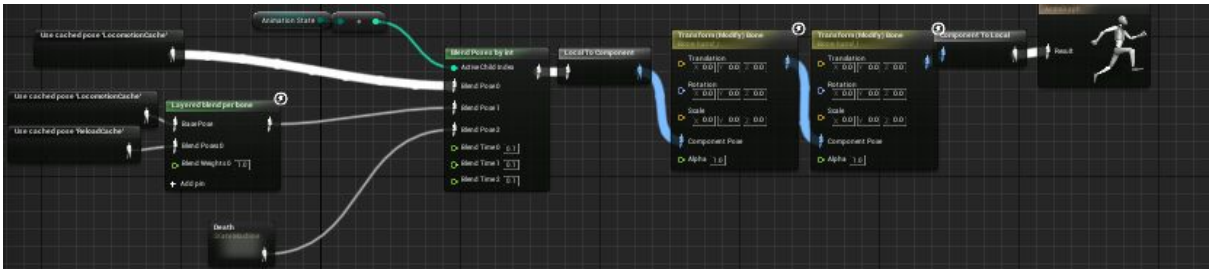
Фиг. 3-8. State machine при умрял герой

Когато един герой бъде убит, индексът на автомата на състоянията се сменя и се изпълнява единично анимацията при смърт. В края си се вика функция от кода, която индикира завършена анимация за умиране



Фиг. 3-9. Избор на анимация за умиране на героя.

Генерираме псевдо случайно число между 0 и 2 и избираме съответната анимация за умиране. Функцията “Blend Poses by int” сумира анимации, но в случая се използва единствено като агрегатор, който да върне избраната анимация.



Фиг. 3-10. Избиране на финална анимация.

Според променливата “Animation index”, чиято стойност се задава от наследения C++ клас функцията “Blend poses by int” агрегира анимациите изчислени от кешовете и автомати на състоянията и избира анимацията съответстваща на този индекс. Изходната анимация се подава като финална.

3.3. Добавяне на Multiplayer функционалност

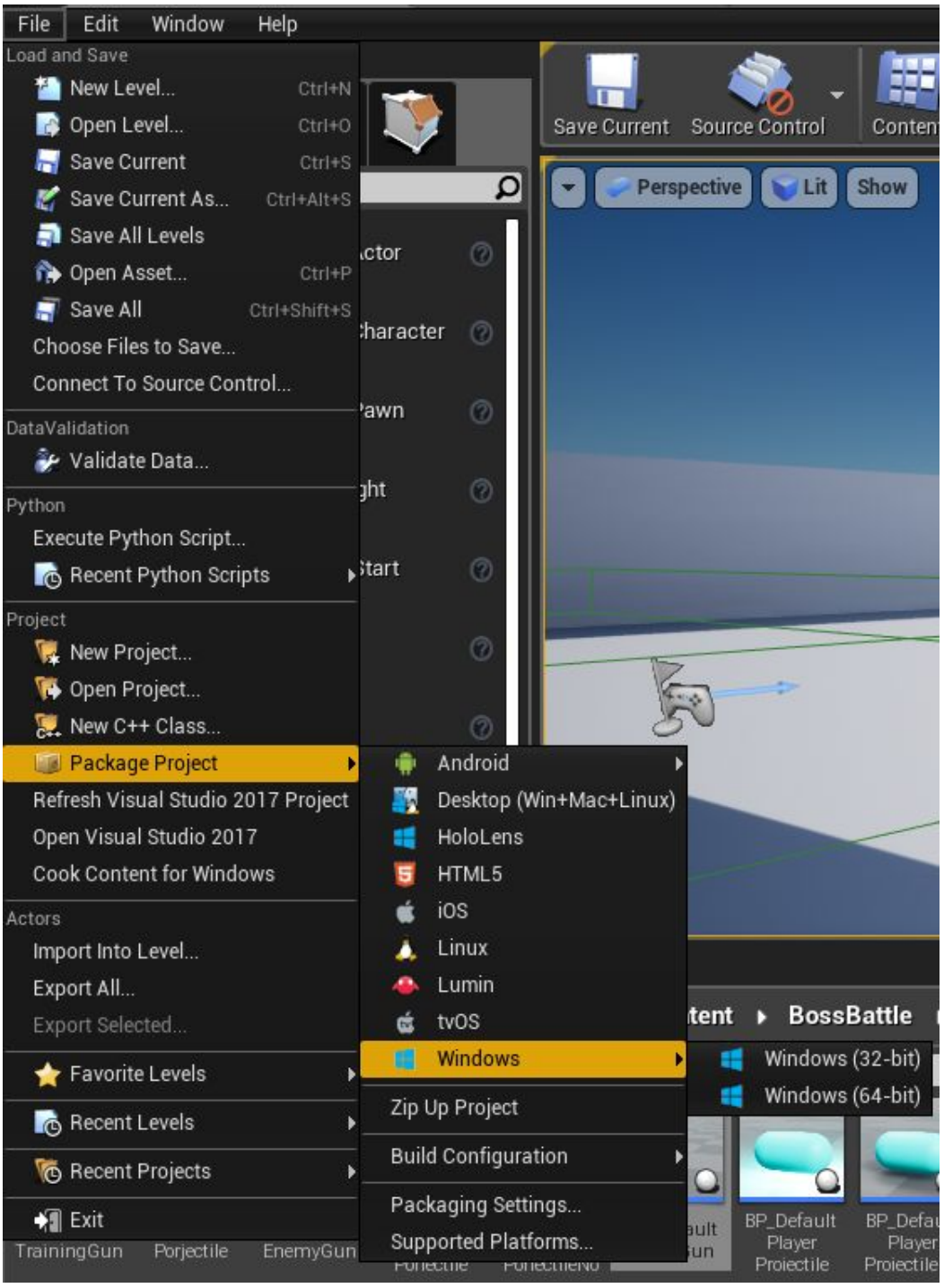
В центъра на играта е мултиплейър режима на игра. Играта е разработена чрез RPC и репликация на герои, оръжия и проектили от сцената.

RPC (Remote procedure call) е метод, при който дадена процедура или функция се извиква на една машина и се изпълнява на друга машина. Това е форма на интеракция между клиента и сървъра. Клиентът изпраща заявка за извършване на дадено действие, а сървърът го извършва. Това се нарича сървърно RPC.

В Unreal Engine 4 съществува и мултикастно RPC. То се извиква на сървъра и се изпълнява на всички клиентски машини. Ако мултикастно RPC се извика на клиентска машина то ще се изпълни само там, защото клиентите нямат изградена сесия помежду си, а само със сървъра.

За играта е разработен специален потребителски интерфейс, в който може да се избере дедикиран сървър, към който да се свържем. Той се разглежда в глава 3.6.

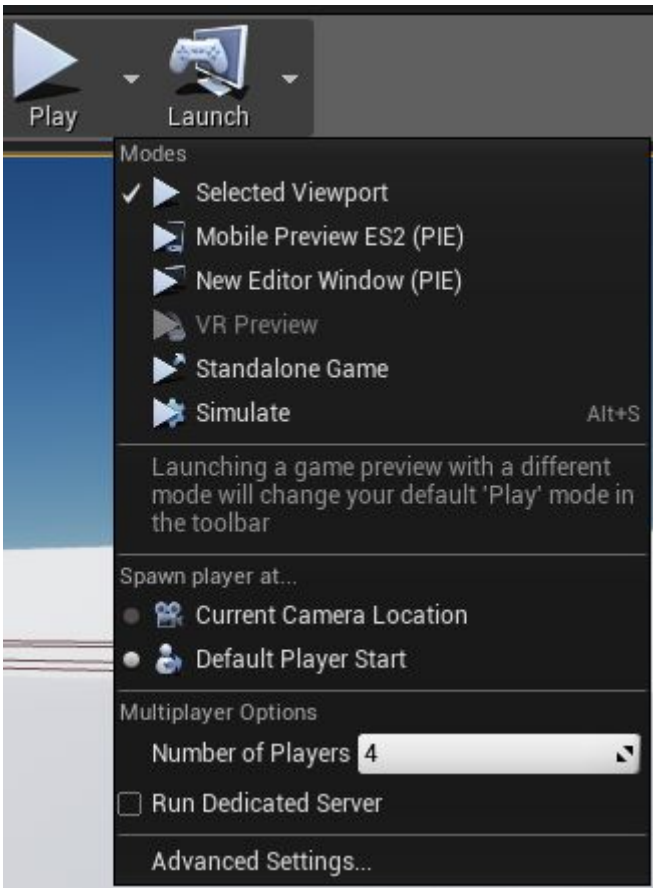
Всеки клиент може да избере менюто за свързване към такъв сървър, да въведе неговото IP и да се свърже. Сървърът работи на well-known порт 7777. При разработката на играта се пакетират както клиентски така и сървърен изпълним бинарен файл на играта. Пакетирането на играта става чрез менюто File > Package Project > Windows > Windows 64 bit:



Фиг. 3-11 Пакетиране на проект

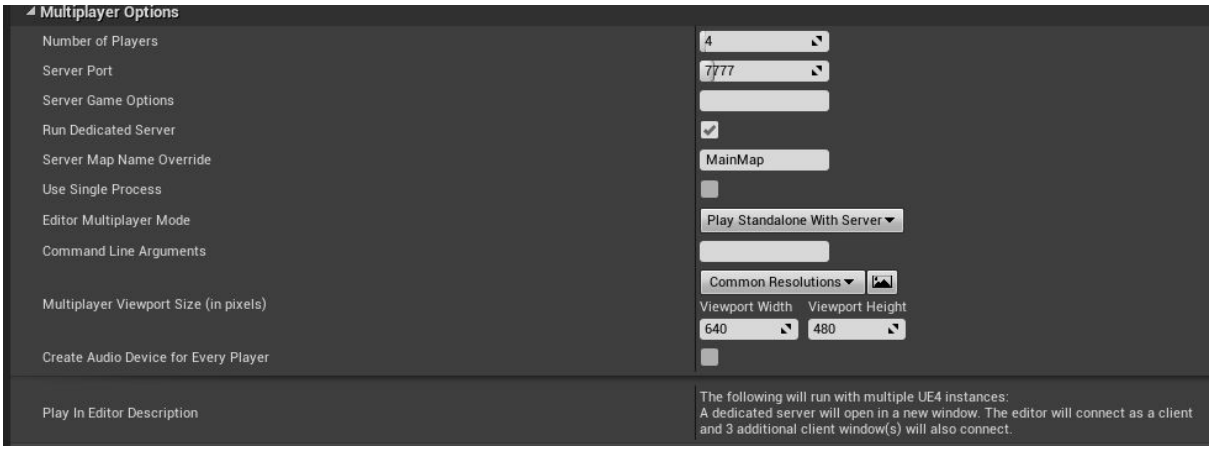
Също така се нуждаем от сървърен бинарен файл. Неговата компилация се обсъжда в глава 4.4.

Разработката на мултиплейър игра на една машина изисква по-специални настройки:



Фиг. 3-12 Базови настройки за симулиране на играта

Имаме възможност да пуснем симулация на играта, която сме разработили във визуализатора на игровия двигател, на телефон или на отделен прозорец на машината за разработка. Можем да изберем да ползваме dedicated server или един от хостовете да играе ролята на клиент и на сървър едновременно. Също така можем да изберем да симулираме играта с повече от един играч. Разширените настройки ни позволяват да контролираме други основни параметри свързани с мултиплейър частта на играта:



Фиг. 3-13 Мултиплейър настройки на симулацията

От настройките за мултиплейър задаваме брой играчи, с които искаме да пуснем симулацията. Ако настройката за единичен процес е включена, те могат да бъдат на една инстанция на играта. Така беше тествано коректната работа на кооперативния режим на играта. Ние не ползваме тази опция. За всеки клиент се създава отделен процес и отделен прозорец на играта. Това прави играта по-трудна за тестване на локалната машина, но ни доближава до реалната среда, в която ще бъде играна тя. Избираме опция, при която имаме отделен процес, който служи като dedicated server. Тогава е нужно и да окажем главното ниво, което да зареди той при стартиране - “MainMap”. Всеки от 4-те клиента ще има прозорец с размер 640x480. За порт на играта ползваме well-known port 7777. Това е destination port на нашия сървър. Когато клиент иска да установи сесия, той трябва да се свърже със съответния сокет.

Сокет е комбинация от IP адрес и порт. когато тестваме на локалната машина това ще бъде 127.0.0.1:7777.

3.4. Добавяне на кооперативен режим на играта

Всеки режим на игра в Unreal има само една инстанция и тя е на сървъра. Клиентите могат да викат функции от неговия публичен интерфейс, но нямат локална негова инстанция.

Разработени са няколко режима на играта: за един играч, кооперативен, за трениране на агента с изкуствен интелект (виж 3.4).

Основния игрови режим на играта е кооперативният. Изчакват се 4 играча да се свържат със един сървър на играта, след което противника се зарежда на сцената и същинската игра започва. Когато играчите убият противника всички те печелят и биват изпращани към менюто при победа. Когато един от играчите умре, той губи играта, сесията му се разрушава и се зарежда менюто за загуба

Разработено е лоби за събиране на играчите преди преместването им в съществената игра. Основните функции в лобито са “PreLogin”, която се извиква преди свързване на играч, “PostLogin”, която се извиква след свързване на играч и “LogOut” при излизане на играч.

Лобито следи текущия брой играчи и при достигане на максималния брой, картата на играта се сменя с игровата:

```
void ALobbyGameMode::PostLogin(class APlayerController*
PlayerController)
{
	Super::PostLogin(PlayerController);
	CurrentPlayers++;
	if (CurrentPlayers == MaxPlayersCount) {
		UWorld* World = GetWorld();
		if (validate(IsValid(World)) == false) return;

		if (validate(MainMapName.Len() > 0) == false) return;

		World->ServerTravel(MainMapName);
	}
```

```
}

void ALobbyGameMode::PreLogin(const FString& Options, const
FString& Address, const FUniqueNetIdRepl& UniqueId, FString&
ErrorMessage)
{
	Super::PreLogin(Options, Address, UniqueId, ErrorMessage);
	if (CurrentPlayers >= MaxPlayersCount) {
		ErrorMessage = "Cannot join";
	}
}

void ALobbyGameMode::Logout(AController* Exiting)
{
	Super::Logout(Exiting);
	CurrentPlayers--;
}
```

Не приемаме повече от максималния брой свързани играчи. При свързване на играч добавяме неговия контролер към списък, чрез който по-късно управляваме действията върху всички играчи.

Отново чрез DataTable е имплементиран spawner на противници:

```
void ABossBattleGameMode::SpawnEnemyWave()
{
	if (CurrentWaveIndex == WaveCount) {
		WinGame();
		return;
	}
	if (validate(IsValid(SpawnerLookupTable)) == false) { return;
}
	FString ContextString(TEXT("GENERAL"));
	FName WaveName = FName(*FString::FromInt(CurrentWaveIndex));
	FSpawnerTable* SpawnerLookupRow =
SpawnerLookupTable->FindRow<FSpawnerTable>(WaveName,
ContextString);
	if (validate(SpawnerLookupRow != nullptr) == false) { return;
}
	TArray<FSpawnerInfoArray> WaveInfo =
SpawnerLookupRow->SpawnerEnemyPlacement;
```

```
	TArray<AActor*> Spawners;
	UGameplayStatics::GetAllActorsOfClass(GetWorld(),
ASpawner::StaticClass(), Spawners);
	if (validate(WaveInfo.Num() >= Spawners.Num()) == false) {
return; }
	for (int i = 0; i < Spawners.Num(); i++) {
		ASpawner* Spawner = Cast<ASpawner>(Spawners[i]);
		if (validate(IsValid(Spawner)) == false) { continue; }
		FSpawnerInfoArray SpawnerEnemyPlacements = WaveInfo[i];

		if
(validate(SpawnerEnemyPlacements.SpawnerInfoArray.Num() != 0) ==
false) continue;
		for (FSpawnerInfo SpawnerInfo :
SpawnerEnemyPlacements.SpawnerInfoArray) {
			if (validate(IsValid(SpawnerInfo.EnemyAsset)) ==
false) { continue; }
			Spawner->SpawnEnemy(SpawnerInfo.EnemyAsset,
SpawnerInfo.EnemyCount);
		}
	}
}
```

При Победа на последната вълна противници печелим играта. В тази игра ползваме само две вълни с 2 противника - агент с изкуствен интелект и противник с ръчно написан алгоритъм. Кода, обаче, поддържа много вълни от много противници. Намираме съответния ’’asset” от “DataTable” и броя инстанции от него, които искаме да поставим на сцената. Имплементиран е клас, който извършва самото действие:

```
void ASpawner::SpawnEnemy(TSubclassOf<AAIEnemyCharacter>
EnemyTemplate, int Count) {
	if (validate(IsValid(BoxComponent))  == false) { return; }
	if (validate(IsValid(EnemyTemplate)) == false) { return; }

	UWorld* World = GetWorld();
	if (validate(IsValid(World)) == false) { return; }


	for (int i = 0; i < Count; i++) {
		//Get random point inside the bounding box
```

```
            FVector ActorLocation = GetActorLocation();
            FVector BoxExtent = BoxComponent->GetScaledBoxExtent();
            FTransform EnemySpawnPosition = FTransform(
                  UKismetMathLibrary::RandomPointInBoundingBox(
                        ActorLocation,
                        BoxExtent
                  )
            );

            //spawn enemy on the given point
            FActorSpawnParameters SpawnParameters;
            SpawnParameters.SpawnCollisionHandlingOverride =
ESpawnActorCollisionHandlingMethod::AlwaysSpawn;
            ACharacter* Enemy = World->SpawnActor<ACharacter>(
                  EnemyTemplate,
                  EnemySpawnPosition,
                  SpawnParameters);
      }

      ABossBattleGameMode* GameMode =
Cast<ABossBattleGameMode>(World->GetAuthGameMode());
      if (validate(IsValid(GameMode)) == false) { return; }

      GameMode->IncrementEnemyCounter(Count);}
```

Намираме случайна позиция в паралелепипед, чрез който ограничаваме мястото, на което може да бъде поставен противника. След това извикваме функцията за създаване на Actor от игровия двигател. Повтаряме действието за всеки противник. Накрая регистрираме в игровия режим наличието на съответния брой противници. При смърт на даден противник броя налични противници в режима на игра се намалява с единица и общия резултат на играчите се увеличава:

```
void AAIEnemyCharacter::Die() {
      Super::Die();

      if (HasAuthority()) {
            AController* EnemyController;
            EnemyController = GetController();
            if (validate(IsValid(EnemyController)) == false) {
return; }
```

```
            UWorld* World = GetWorld();
            if (validate(IsValid(World)) == false) { return; }

            EnemyController->StopMovement();
            EnemyController->Destroy();

            ABossBattleGameMode* BossBattleGameMode =
Cast<ABossBattleGameMode>(World->GetAuthGameMode());
            if (IsValid(BossBattleGameMode)) {
                  BossBattleGameMode->IncrementScore(Score);
                  BossBattleGameMode->DecrementEnemyCounter();
                  return;
            }

            ATrainingGameMode* TrainingGameMode =
Cast<ATrainingGameMode>(World->GetAuthGameMode());
            if (IsValid(TrainingGameMode)) {
                  //TODO: perhaps add death timer
                  TrainingGameMode->ResetCharacters(true);
                  return;
            }

      }

      SetActorEnableCollision(false);
      UCapsuleComponent* Capsule;
      Capsule = GetCapsuleComponent();
      if (validate(IsValid(Capsule)) == false) { return; }
      Capsule->SetEnableGravity(false);
}
```

При елиминиране на всички противници преминаваме към следващата вълна от противници.

```
void ABossBattleGameMode::IncrementScore(const int Amount)
{
      CurrentScore += Amount;
      UpdateHUDScore(CurrentScore);
}
void ABossBattleGameMode::UpdateHUDScore(int Score) {
      for (APlayerCharacterController* PlayerController :
PlayerControllers) {
```

```
            if (validate(IsValid(PlayerController)) == false) {
continue; }

            ABattleHUD* HUD =
Cast<ABattleHUD>(PlayerController->GetHUD());
            if (validate(IsValid(HUD)) == false) { continue; }

            UPlayerStatsWidget* PlayerStats =
HUD->GetPlayerStatsWidget();
            if (validate(IsValid(PlayerStats))) {
                  PlayerStats->SetScore(Score);
            }
      }
}
```

При убийство на противник резултатът се увеличава. Това увеличение се отразява от всички играчи, чрез техните контролери. На сървърната част записахме контролерите на играчите при тяхното свързване, след което можем да ги използваме, за да извикаме функция, която ще промени потребителския интерфейс на всеки играч. Също така намаляваме броя на противниците с единица. При унищожаване на всички противници от една вълна, преминаваме към следваща. При унищожаване на всички вълни печелим играта:

```
void ABossBattleGameMode::DecrementEnemyCounter()
{
      CurrentEnemies -= 1;
      if (AreAllEnemiesDead()) {
            CurrentWaveIndex++;
            //We win the game after all the enemy waves and the
boss wave
            if (CurrentWaveIndex == WaveCount) {
                  UE_LOG(LogTemp, Display, TEXT("You won the
game"));
                  WinGame();
            }
            else {
                  //TODO: add delay between waves
                  SpawnEnemyWave();
            }
      }
```

```
}
```

След смъртта на един играч, той губи играта, сесията му със сървъра бива разрушена и се показва екранът за загуба след определено време:

```
void APlayerCharacterController::OnLoseGame_Implementation() {
	ABattleHUD* HUD = Cast<ABattleHUD>(GetHUD());
	if (validate(IsValid(HUD)) == false) { return; }

	UPlayerStatsWidget* PlayerStatsWidget =
HUD->GetPlayerStatsWidget();

	if (validate(IsValid(PlayerStatsWidget))) {
		PlayerStatsWidget->SetLoseGame();
	}

	FTimerHandle RespawnTimerHandle; // not used anywhere
	GetWorldTimerManager().SetTimer(
		RespawnTimerHandle,
		this,
		&APlayerCharacterController::LoadLoseLevel,
		LoadEndLevelDelay
	);

}
```

Когато играчът или играчите убият всички противници, при всички тях едновременно сесията бива разрушена и се показва екранът за победа след определено време:

```
void AMultiplayerGameMode::WinGame()
{
		for (FConstPlayerControllerIterator Iterator =
GetWorld()->GetPlayerControllerIterator(); Iterator; ++Iterator) {
		APlayerCharacterController* Controller =
Cast<APlayerCharacterController>(*Iterator);
		if (validate(IsValid(Controller)) == false) return;

		Controller->OnWinGame();
	}

	FTimerHandle TimerHandle;
```

```
      GetWorldTimerManager().SetTimer(TimerHandle,this,
&AMultiplayerGameMode::LoadLobby, 3.0f, false);
}
```

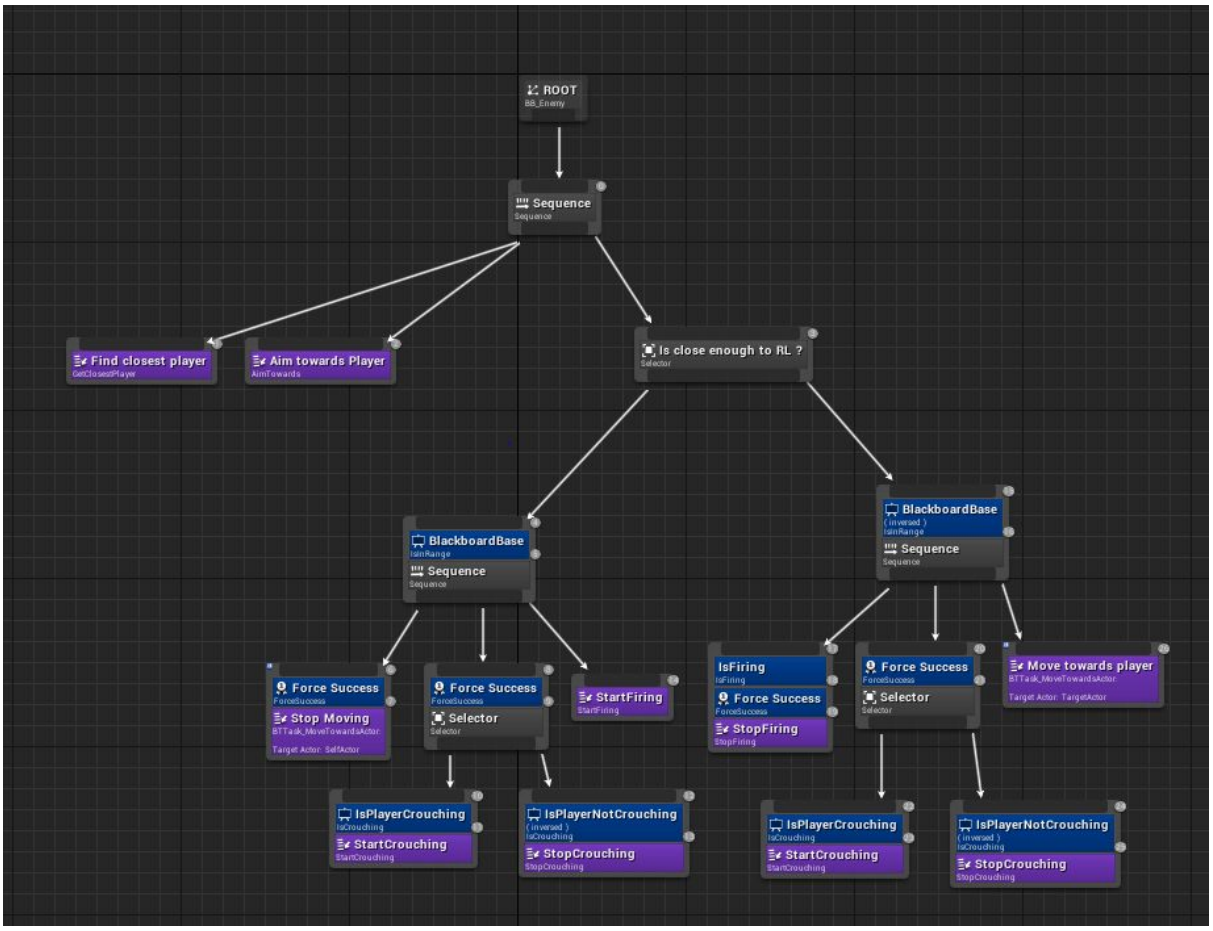
За всеки играч се извиква съответната функция за победа или загуба. Причините да имаме две почти аналогични функции са по-голямата описателност на кода и факта, че извикваните функции са RPC, които се викат от сървъра и се изпълняват на клиентската машина, което не позволява ползването на единична функция.

3.5. Добавяне на противник с изкуствен интелект.

Това, което прави тази дипломна работа иновативна, е имплементираният изкуствен интелект чрез самообучение с утвърждение.

Самообучението чрез утвърждение се състои от външна среда и агент, който извършва действия. Подаваме на агента състояние, което сме извлекли от външната среда. На база на това състояние агента предсказва кое е действието, което трябва да предприеме за да получи възможно най-голяма награда. Той извършва това действие и получава действителна награда. Наградата може да бъде положителна или отрицателна. Ние задаваме кои действия да бъдат наградени положително и кои отрицателно. Агента създава таблица, която показва за всяко състояние, при извършване на определено действие, каква ще бъде предсказаната награда. Извършва се действието с най-голяма предсказана награда за съответното състояние. След получаването на действителната награда таблицата се променя според разликата между предсказаната и получената награда.

Бяха разработени противници чрез дърво на състоянията (Behavior Tree) и чрез самообучение с утвърждение. Алгоритъмът на дървото на състоянията е следния:



Фиг. 3-14. Алгоритъм чрез дърво на състоянията.

Дървото на състоянията се изпълнява от горе надолу и от ляво надясно. състои се от различни визуални блокове. Всеки един от тези блокове е имплементиран чрез C++ код. Има различни видове блокове - за действия и контролни.

Основните блокове в дървото на състоянията са блоковете за действия. При изпълняване на такъв блок се изпълнява съответното имплементирано от него действие. Всеки един блок за действие може да завърши изпълнението си успешно или неуспешно.

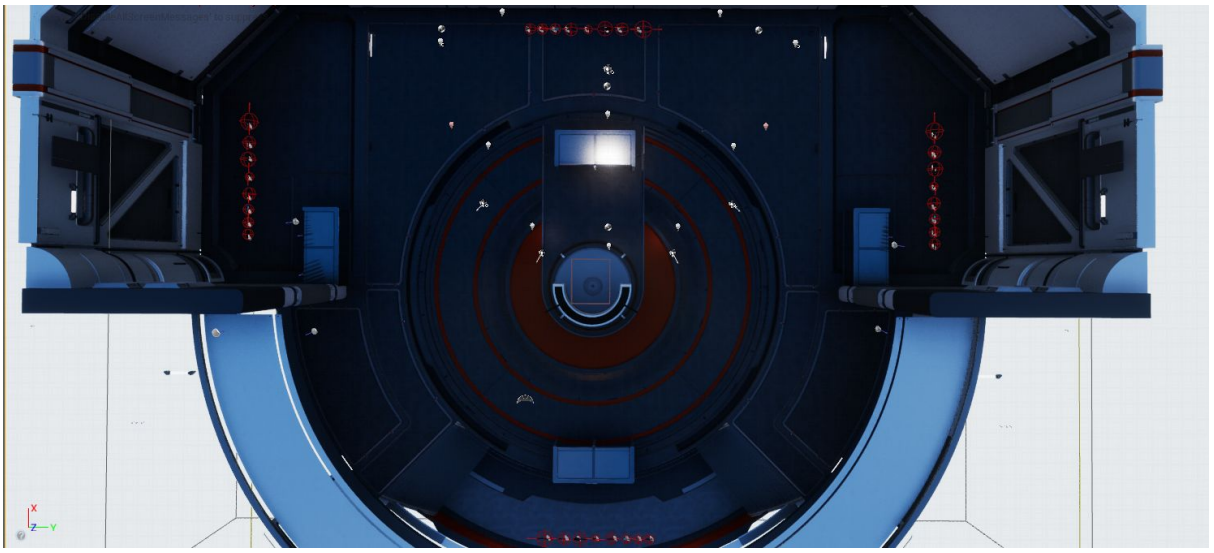
“Sequence” блокът е контролен блок, който задава последователно изпълнение на блокове с действия. При успешно изпълнение на един блок с действие преминаваме към следващия. При неуспешно изпълнение целия “Sequence” блок завършва действието си неуспешно и връща контрола на изпълнение на действията към по-горен клон.

“Selector” блокът е контролен блок, противоположен на “Sequence”. При успешно извършено действие на краен блок за действие той приключва успешно и връща контрола нагоре. При неуспешно изпълнение той избира следващия клон, докато не намери действие, което да се изпълни успешно.

Декораторите (изобразени в синьо над различните блокове) променят и контролират изпълнението на им. Могат да бъдат поставени както над блок за действие, така и над контролен блок. Те добавят условности, които да лимитират изпълнението на съответния блок. При изпълнено условие на декоратор блокът също бива изпълнен. При неизпълнено условие действието на целия блок не бива изпълнено. Също така може да бъде конфигурирано така, че целия клон да приключи изпълнението си (аборт). Декораторите също могат да променят резултата от блок с действие (например “Force Success”, който винаги връща успешно изпълнение).

Алгоритъмът на противниците с дърво на състоянията намира този, използващ обучение с утвърждение. след това се насочва към него. Проверява дали е в допустим обсег за стрелба. Ако условието е изпълнено започва да стреля. В противен случай спира да стреля (ако до момента е стрелял) и започва да ходи към него.

Средата за трениране на агента, ползващ самообучение чрез утвърждение се състои от една квадратна арена с препятствия. Агентът започва играта в центъра на картата и от всяка негова страна е заобиколен от по един противник, използващ традиционен behavior tree state machine (дървовиден автомат на състоянията). За по-интересен режим на играта са поставени и препятствия, които трябва да бъдат избягвани. Имплементирано е едно ниво с агент срещу противници с автомат на състоянията и едно ниво с тренираш се агент срещу вече тренирани агенти.



Фиг. 3-15 Среда за трениране на агента.

В четирите края на терена се намират оръжия, които могат да бъдат взети и ползвани от всички герои. Обучаваният агент започва играта от случайна позиция в червения квадрат в центъра на терена.

Имплементиран бе потребителски интерфейс, който показва в реално време основните параметри на играта (състояния, действия, коефициенти) и позволява променянето на коефициентите за трениране в реално време.

Фиг. 3-16 Потребителски интерфейс за контрол на машинното самообучение

Йерархията на контролерите на агентите, използващи машинно самообучение, се състои от базов клас, имплементирал всички общи функции и два класа, наследяващи от него, за трениране на агент и за ползване на тренирана таблица.

При започване на играта за един агент се поставя не сцената неговия герой. Вика се функцията “OnPossess” за неговия контролер.Тя динамично създава таблица с очакваните награди. Представлява двумерен масив от числа с десетична запетая, добавя възможните действия на агента и десериализира коефициентите за трениране:

```
void ARLTrainingController::OnPossess(APawn* InPawn)
{
	Super::OnPossess(InPawn);

	FString SaveSlotName = "Training";

	UTrainingSaveGame* LoadedGame = 
Cast<UTrainingSaveGame>(UGameplayStatics::LoadGameFromSlot(SaveSlo
tName, 0));
	if (validate(LoadedGame) != false) {
		TakingDamageReward = LoadedGame->TakingDamageReward;
		DealingDamageReward = LoadedGame->DealingDamageReward;
		KillReward = LoadedGame->KillReward;
		AliveReward = LoadedGame->AliveReward;
		DeathReward = LoadedGame->DeathReward;
		Epsilon = LoadedGame->Epsilon;
		LearningRate = LoadedGame->LearningRate;
		CriticalHealthRewardMultiplier = 
LoadedGame->CriticalHealthRewardMultiplier;
	}

}


void ARLController::OnPossess(APawn* InPawn)
{

	ARLEnemyCharacter* RLCharacter = 
Cast<ARLEnemyCharacter>(InPawn);
	if (validate(IsValid(RLCharacter)) == false) return;
	Actions.push_back(new MoveForwardAction(RLCharacter));
	Actions.push_back(new MoveBackwardAction(RLCharacter));
```

```
	Actions.push_back(new MoveLeftAction(RLCharacter));
	Actions.push_back(new MoveRightAction(RLCharacter));
	Actions.push_back(new FocusOnEnemyAction(RLCharacter));
	Actions.push_back(new ShootEnemyAction(RLCharacter));
	QTable = new float*[StateCount];
	for (int i = 0; i < StateCount; i++) {
		QTable[i] = new float[Actions.size()];
		for (int j = 0; j < Actions.size(); j++) {
			QTable[i][j] = 0.0f;
		}
	}
	bPossessed = true;
	DeserializeTable(QTable);
}
```

Десериализираме текущата таблица с научени стойности от файл. Добавяме Всички възможни действия, които героя може да предприеме. Имплементиран е клас за действие с абстрактен метод за изпълнение:

```
class BOSSBATTLE_API Action
{
public:

	Action(class ARLEnemyCharacter* CharacterToSet);

	virtual void Execute();

protected:

	class ARLEnemyCharacter* EnemyCharacter = nullptr;

};
```

Всяко действие наследява от него и имплементира виртуалния метод “Execute”. Добавят се в лист от действия и когато едно от тях трябва да се изпълни то става чрез достъпване по индекс и викане на метода “TakeAction”:

```
void ARLController::TakeAction(int ActionIndex)
{
	Actions[ActionIndex]->Execute();
}
```

Получаването на текущото състояние е базирано на това дали героя нанася щета на някой от останалите герои, дали върху него се нанася щета и дали е под критично ниво на кръвта (30%).

```
int ARLController::GetState()
{
	// taking current state
	//get dealing damage flag, the other two flags are calculated
on taking damage event
	UWorld* World = GetWorld();
	if (validate(IsValid(World)) == false) return -1;
	bool bDealingDamage = false;
	TArray<AActor*> OutActors;
	UGameplayStatics::GetAllActorsOfClass(World,
AAIEnemyCharacter::StaticClass(), OutActors);
	for (auto Actor : OutActors) {
		AAIEnemyCharacter* AICharacter =
Cast<AAIEnemyCharacter>(Actor);
		if (AICharacter->IsTakingDamage()) {
			bDealingDamage = true;
		}
	}
	ARLEnemyCharacter* RLCharacter =
Cast<ARLEnemyCharacter>(GetPawn());
	if (validate(IsValid(RLCharacter)) == false) return -1;

	int CurrentState = (bDealingDamage * 4) +
RLCharacter->IsTakingDamage() * 2 +
RLCharacter->IsOnCriticalHealth(); // binary flags to integer

	return CurrentState;
}
```

Избора на най-добро действие се свежда до взимане на най-голямата стойност на очаквана награда от реда на таблицата, съответстващ на това състояние:

```
void ARLController::GetBestAction(const int CurrentState, int&
OutCurrentActionIndex, float& OutCurrentActionValue)
{
	//get current action values for this state
	//get the action value that gives us the highest reward from
our array
```

```
	OutCurrentActionValue = QTable[CurrentState][0];
	OutCurrentActionIndex = 0;
	//find max value
	for (int i = 0; i < Actions.size(); i++) {
		if (QTable[CurrentState][i] > OutCurrentActionValue) {
			OutCurrentActionValue = QTable[CurrentState][i];
			OutCurrentActionIndex = i;
		}
	}

}
```

Сериализацията и десериализацията на таблицата с очакваните награди става посредством писане и четене във файл чрез класовете на Unreal “FPaths” и “FFileHelper”:

```
void ARLController::SerializeTable(float** Table)
{
		FString FilePath = 
FPaths::ConvertRelativePathToFull(FPaths::ProjectSavedDir()) + 
TEXT("/table.csv");

	FString FileContent;
	for (int i = 0; i < StateCount; i++) {
		for (int j = 0; j < Actions.size() - 1; j++) {

FileContent.Append(FString::SanitizeFloat(Table[i][j]));
			FileContent.Append(",");
		}

FileContent.Append(FString::SanitizeFloat(Table[i][Actions.size()
- 1]));
		FileContent.Append("\n");

	}
	//TODO: maybe tweak flags
	bool bSuccess = FFileHelper::SaveStringToFile(FileContent, 
*FilePath, FFileHelper::EEncodingOptions::AutoDetect, 
&IFileManager::Get());
	if (validate(bSuccess) == false) return;
```

```
}
```

Когато агента умре се вика метода му “EndPlay”, който записва текущите параметри от тренирането за следващия агент, сериализира досегашната таблица и изчиства заделената от нас динамична памет. Налага се ръчно да изчистваме паметта, само когато работим с динамична памет и класове, които не наследяват от класа UObject. За вторите се грижи имплементирания в Unreal Engine Garbage collector.

```
void ARLTrainingController::EndPlay(const EEndPlayReason::Type
EndPlayReason)
{
	SerializeTable(QTable);

	ShowTable(QTable);

	UTrainingSaveGame* SaveGameInstance =
Cast<UTrainingSaveGame>(UGameplayStatics::CreateSaveGameObject(UTr
ainingSaveGame::StaticClass()));
	if (validate(IsValid(SaveGameInstance)) == false) return;

	// Set data on the savegame object.
	SaveGameInstance->TakingDamageReward = TakingDamageReward;
	SaveGameInstance->DealingDamageReward = DealingDamageReward;
	SaveGameInstance->KillReward = KillReward;
	SaveGameInstance->DeathReward = DeathReward;
	SaveGameInstance->Epsilon = Epsilon;
	SaveGameInstance->LearningRate = LearningRate;
	SaveGameInstance->AliveReward = AliveReward;
	SaveGameInstance->CriticalHealthRewardMultiplier =
CriticalHealthRewardMultiplier;

	FString SaveSlotName = "Training";
	// Save the data immediately.
	if (UGameplayStatics::SaveGameToSlot(SaveGameInstance,
SaveSlotName, 0))
	{
		UE_LOG(LogTemp, Warning,
```

```
TEXT("UGameplayStatics::SaveGameToSlot(SaveGameInstance,
SaveSlotName, 0)"))
                  // Save succeeded.
      }
      else {
            UE_LOG(LogTemp, Error,
TEXT("UGameplayStatics::SaveGameToSlot(SaveGameInstance,
SaveSlotName, 0) == false"))

      }
      Super::EndPlay(EndPlayReason);
}


void ARLController::EndPlay(const EEndPlayReason::Type
EndPlayReason)
{

      UE_LOG(LogTemp, Warning, TEXT("ARLController::EndPlay"))


      for (int i = 0; i < StateCount; i++) {
            delete[] QTable[i];
      }
      delete[] QTable;

      for (int i = 0; i < Actions.size(); i++) {
            delete Actions[i];
      }

      Super::EndPlay(EndPlayReason);
}
```

Контролера за трениране добавя функционалност за показване на текущата таблица в логовете:

```
void ARLTrainingController::ShowTable(float** Table)
{
      UE_LOG(LogTemp, Warning, TEXT("QTable:\n==========\n"));

      for (int i = 0; i < StateCount; i++)
      {
```

```
            for (int j = 0; j < Actions.size(); j++) {

                  UE_LOG(LogTemp, Warning, TEXT("State: %d, Action:
%f\n"), i, Table[i][j])
            }

      }
      UE_LOG(LogTemp, Warning, TEXT("==========\n"));}
```

Също така в него е дефинирана функцията за награждаване на агента, базирано на действието, което е предприел. Разработчика може да дефинира стойностите на наградите, които получава агента чрез потребителския интерфейс. Те също имат стойности по подразбиране. При убийство агента получава награда +1, при нанасяне на щета + 0.5, при поемане на щета -0.5 и при смърт -1.

```
float ARLTrainingController::GetReward()
{

      UWorld* World = GetWorld();
      if (validate(IsValid(World)) == false) return 0.0f;

      bool bHittingAICharacter = false;
      bool bKilledAICharacter = false;
      bDealingDamage = false;


      TArray<AActor*> OutActors;
      UGameplayStatics::GetAllActorsOfClass(World,
AAIEnemyCharacter::StaticClass(), OutActors);
      for (AActor* Actor : OutActors) {
            AAIEnemyCharacter* AICharacter =
Cast<AAIEnemyCharacter>(Actor);
            if (validate(IsValid(AICharacter)) == false) return
0.0f;

            if (AICharacter->IsTakingDamage()) {
                  bHittingAICharacter = true;
                  bDealingDamage = true;
            }
            if (AICharacter->IsDead()) {
                  bKilledAICharacter = true;
```

```
		}
	}


	bDealingDamage = bHittingAICharacter;

	ARLEnemyCharacter* RLCharacter =
Cast<ARLEnemyCharacter>(GetPawn());
	if (validate(IsValid(RLCharacter)) == false) return 0.0f;

	float Reward = 0.0f;
	if (bKilledAICharacter) {
		Reward += KillReward;
	}
	if (RLCharacter->IsDead()) {
		Reward += DeathReward;
	}
	if (RLCharacter->IsTakingDamage()) {
		Reward += TakingDamageReward;
	}
	if (bHittingAICharacter) {
		Reward += DealingDamageReward;
	}

	if (RLCharacter->IsOnCriticalHealth()) {
		Reward += (TimeAlive * AliveReward *
CriticalHealthRewardMultiplier);
	}
	return Reward;
}
```

В този контролер се развива и основното действие по тренирането. На всеки кадър се извиква функцията “Tick”. Тя получава наградата и състоянието за текущия кадър. Избира най-доброто действие според текущото състояние. Има шанс да избере случайно действие, което да изпълни. Това провокира у героя изследователско поведение. Шансът може да бъде ръчно дефиниран, а ако не е той бива изчислен според текущия епизод. Епизода е число, което се променя на определен брой кадри. При по-малък епизод има по-голям шанс за случайно действие и обратно. Така в

началото на играта агента е по-вероятно да разузнава повече и по-малко вероятно към края на тренирането. Базирано на текущата награда се променя стойността на очакваната награда при предишното действие и състояние. Промяната се изчислява според разликата между очакваната и действителната получена награда. Коефициентът “learning rate” указва колко бързо да се променя стойността, а коефициентът “discount factor” обозначава колко значими да бъдат наградите, получени в бъдещето в сравнение с тези в настоящето. Накрая се извършва избраното от алгоритъма най-оптимално действие.

```
void ARLTrainingController::Tick(float DeltaTime)
{
	Super::Tick(DeltaTime);

	UpdateStepAndEpisode();

	// update the previous state and action to the current state
and action before updating them
	PreviousStateIndex = CurrentStateIndex;
	PreviousActionIndex = CurrentActionIndex;

	// get the reward received this step
	CurrentReward = GetReward();

	CurrentStateIndex = GetState();
	if (validate(CurrentStateIndex >= 0) == false) return;

	// choosing current action based on the current state. We
either take a random action or the best action from the q-table
for this step
	// if we don't set a custom epsilon value the random action
chance is determined by the episode number
	bRandomAction = FMath::IsNearlyZero(Epsilon) ?
FMath::RandRange(0.0f, 1.0f) < (1 / Episode) :
FMath::RandRange(0.0f, 1.0f) < Epsilon;

	if (bRandomAction == true) {

		CurrentActionIndex = FMath::RandRange(0, 5);

	}
```

```
	else {

		GetBestAction(CurrentStateIndex, CurrentActionIndex,
CurrentActionValue);
		if (validate(CurrentActionIndex >= 0 &&
CurrentActionIndex < Actions.size()) == false) return;

	}

	//find the previous action value
	if (validate(PreviousActionIndex >= 0 && PreviousActionIndex
< Actions.size()) == false) return;
	float PreviousActionValue =
QTable[PreviousStateIndex][PreviousActionIndex];

	//update the previous action value
	PreviousActionValue = PreviousActionValue + LearningRate *
(CurrentReward + (DiscountFactor * CurrentActionValue) -
PreviousActionValue);

	QTable[PreviousStateIndex][PreviousActionIndex] =
PreviousActionValue;

	ShowTable(QTable);

	//taking the current action
	TakeAction(CurrentActionIndex);
}
```

3.6. Добавяне на Потребителски интерфейс (Меню, Настройки, Крайно меню)

При включване на играта се зарежда отделно ниво за главното меню на играта. То има собствен HUD, който зарежда основния Widget на менюто и управлява показването и скриването на различните Widget инстанции на екрана. Също така е

добавен PlayerController, който задава настройката за показване на курсура на екрана. Главното меню изглежда по следния начин:

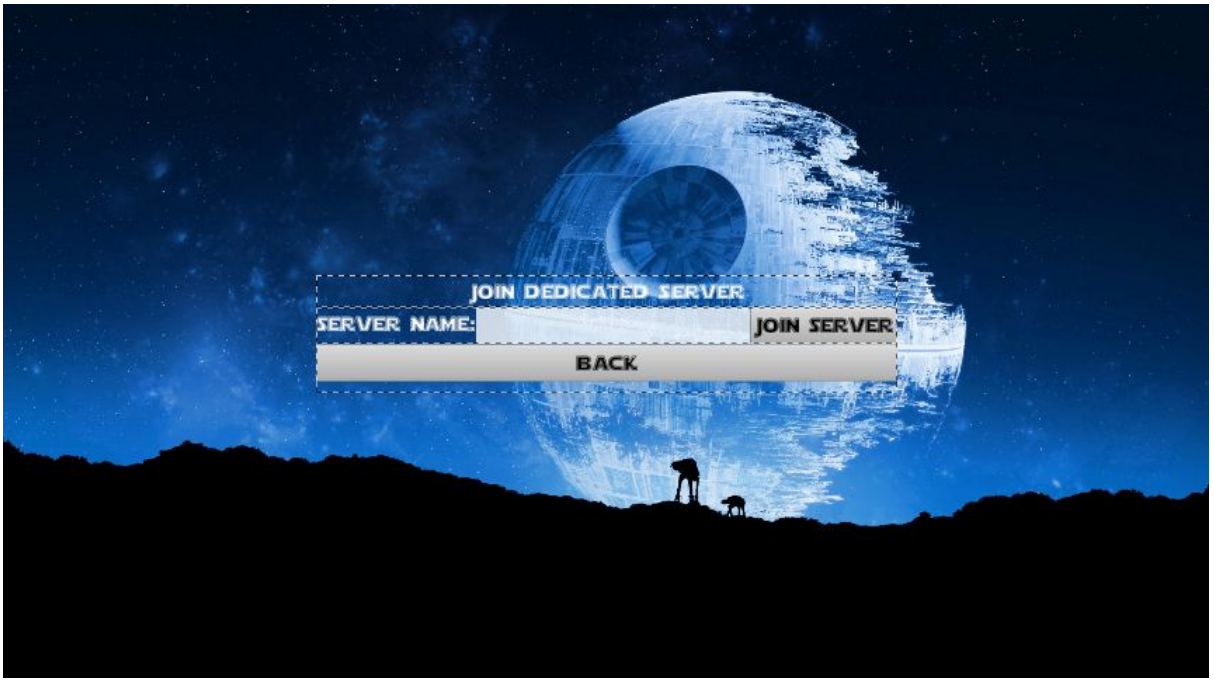
Фиг 3-17. Главно меню на играта.

При избиране на опция “Play” се извежда меню за избор на режим на играта. Съществува единичен и кооперативен режим. При избор на единичния се зарежда локално сцената за игра. При избор на кооперативен се прави опит за свързване към дедикиран сървър:

```
void UGamemodeSelectionWidget::LoadMultiplayer()
{
	UWorld* World = GetWorld();
	if (validate(IsValid(World)) == false) { return; }

	SetInputModeGameOnly();

	APlayerController* PlayerController =
World->GetFirstPlayerController();
	if (validate(IsValid(PlayerController)) == false) return;

	//multiplayer joins local server @ 127.0.0.1
	PlayerController->ClientTravel(ServerAddress,
ETravelType::TRAVEL_Absolute);
}
```

При разработката на играта, сървърът се намира на локалната машина съответно IP адресът, който ползваме, е “127.0.0.1”. При успешна връзка със сървъра, той ни премества към главната сцена на играта.

Менюто “Training” зарежда на локалната машина сцената, използвана за трениране на изкуствения интелект. Играчът е само наблюдател на действието в нея. Тя представлява битка между противник, който се учи да играе играта, и такъв, за когото вече е имплементиран чрез код алгоритъм за действията в нея.



Фиг 3-18. Меню за свързване с дедикиран сървър

Менюто “Multiplayer” съдържа поле за въвеждане на IP адреса на дедикирания сървър и логиката за свързване с него. При натискане на бутона “Join Server” се извиква функция аналогична на “`LoadMultiplayer`”. При натискане на бутона “Back” се връщаме към главното меню

Фиг. 3-19. Меню за графични настройки

Менюто за графични настройки позволява на потребителя да избира между 5 опции относно качество на графиките, post-processing, anti-aliasing, качество на сенките и честота на кадрите. Зад всяка една от опциите стои списък от команди, списък с имена на опциите и индекс на текущата избрана опция. Двата списъка споделят общ индекс за достъпване. При натискане на стрелките наляво или надясно индексът съответно се намалява или увеличава.

В началото при зареждане на настройките се десериализират текущите им стойности, а ако няма такива се задават такива по подразбиране:

```
void USettingsWidget::NativeConstruct() {
	Super::NativeConstruct();

	FString SaveSlotName = "Settings";

	USettingsSaveGame* LoadedGame =
Cast<USettingsSaveGame>(UGameplayStatics::LoadGameFromSlot(SaveSlo
tName, 0));
	if (validate(LoadedGame) != false) {

		FPSIndex = LoadedGame->FPSIndex;
		AAIndex = LoadedGame->AAIndex;
		PPIndex = LoadedGame->PPIndex;
```

```
            GraphicalIndex = LoadedGame->GraphicalIndex;
            ShadowIndex = LoadedGame->ShadownIndex;

            // The operation was successful, so LoadedGame now
contains the data we saved earlier.
      }
      else {
            //set initial values
            FPSIndex = 2;
            AAIndex = 2;
            PPIndex = 2;
            GraphicalIndex = 2;
            ShadowIndex = 2;

            SaveSettings();
      }
```

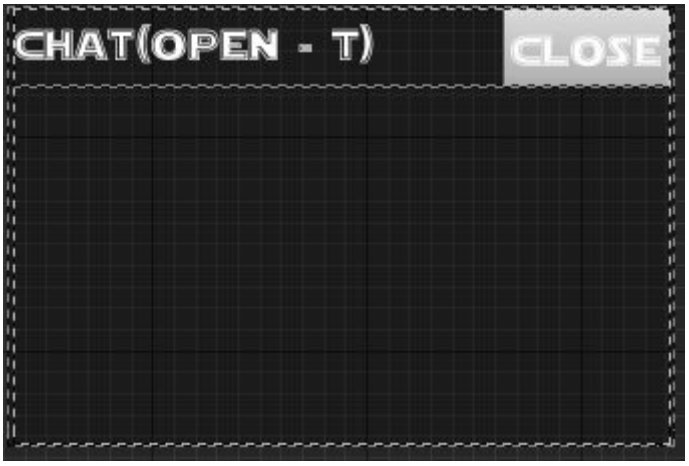
При натискане на бутона за запис, командите, чиито индекси са избрани, биват изпълнени и се връщаме към главното меню:

```
void USettingsWidget::SaveSettings()
{
      USettingsSaveGame* SaveGameInstance =
Cast<USettingsSaveGame>(UGameplayStatics::CreateSaveGameObject(USe
ttingsSaveGame::StaticClass()));
      if (validate(IsValid(SaveGameInstance)) == false) return;

      // Set data on the savegame object.
      SaveGameInstance->FPSIndex = FPSIndex;
      SaveGameInstance->AAIndex = AAIndex;
      SaveGameInstance->PPIndex = PPIndex;
      SaveGameInstance->GraphicalIndex = GraphicalIndex;
      SaveGameInstance->ShadownIndex = ShadowIndex;


      FString SaveSlotName = "Settings";
      // Save the data immediately.
      if (UGameplayStatics::SaveGameToSlot(SaveGameInstance,
SaveSlotName, 0))
      {
            UE_LOG(LogTemp, Warning,
```

```
TEXT("UGameplayStatics::SaveGameToSlot(SaveGameInstance,
SaveSlotName, 0)"))
                // Save succeeded.
     }
     else {
          UE_LOG(LogTemp, Error,
TEXT("UGameplayStatics::SaveGameToSlot(SaveGameInstance,
SaveSlotName, 0) == false"))

     }
}
```

3.7. Чат

Играчите имат възможността да комуникират помежду си чрез чат. За целта разработваме Widget за чата. Той се състои от поле за въвеждане, поле, в което се показват съобщенията с вграден скрол и бутон за затваряне на чата. Той изглежда по следния начин:



Фиг. 3-20 Потребителски интерфейс на отворен чат

Също така беше разработена анимация за затваряне на чата при натискане на бутона за затваряне. Крайното състояние на затворения чат изглежда така:

Фиг. 3-21. Потребителски интерфейс на затворен чат

Играчът може да отвори чата чрез контролата “OpenChat”(Виж 4.3 контроли)

```
PlayerInputComponent->BindAction("OpenChat", IE_Pressed, this,
&APlayerCharacter::OpenChat);

void APlayerCharacter::OpenChat() {
      if (validate(IsValid(ChatWidget)) == false) return;
```

```
	ChatWidget->PlayAnimation(ChatWidget->GetChatAnimation(),
0.0f, 1, EUMGSequencePlayMode::Reverse, 1.0f);

	APlayerController* PlayerController =
Cast<APlayerController>(GetController());
	if (validate(IsValid(PlayerController))) {
		PlayerController->bShowMouseCursor = true;

UWidgetBlueprintLibrary::SetInputMode_UIOnly(PlayerController,
ChatWidget, false);
	}
}
```

При отваряне на чата се изпълнява същата анимация като при затваряне на чата, но инвертирана. Показва се курсорът на мишката и влизаме в режим на обработване на събития единствено върху потребителския интерфейс.

```
CloseButton->OnClicked.AddDynamic(this, &UChatWidget::Close);

void UChatWidget::Close() {
	PlayAnimation(OpenCloseAnimation, 0.0f, 1,
EUMGSequencePlayMode::Forward, 1.0f);

	APlayerController* PlayerController = GetOwningPlayer();
	if (validate(IsValid(PlayerController))) {

UWidgetBlueprintLibrary::SetInputMode_GameOnly(PlayerController);
		PlayerController->bShowMouseCursor = false;
	}
}
```

При натискане на бутона за затваряне на чата се извиква функцията “Close”. Тя изпълнява анимацията за затваряне, скрива курсора на мишката и променя режима на обработка на натиснати контроли към такъв, при който те се обработват само в контекста на играта. В Unreal има ясно разделение между събития свързани с играта и

тези свързани с потребителския интерфейс. При задаване на обработка на събитията от контроли като събития в потребителския интерфейс, те ще бъдат игнорирани в играта. Обратното твърдение също е вярно.

```
void UChatWidget::HandleOnTextCommitted(const FText& Text,
ETextCommit::Type CommitMethod) {

	UE_LOG(LogTemp, Warning,
TEXT("UChatWidget::HandleOnTextCommitted(const FText& Text,
ETextCommit::Type CommitMethod)"))
	if (CommitMethod == ETextCommit::OnEnter &&
!Text.EqualTo(FText::FromString(""))) {
		UE_LOG(LogTemp, Warning, TEXT("CommitMethod ==
ETextCommit::OnEnter && !Text.EqualTo(FText::FromString(""))"))
		ABattlePlayerState* PlayerState =
Cast<ABattlePlayerState>(GetOwningPlayerState());
		if (validate(IsValid(PlayerState))) {
			PlayerState->ServerSendMessage(Text,
FText::FromString(PlayerState->GetPlayerName()));

		}
	}
}
```

След въвеждане на текст в полето за въвеждане на текст на чата натискаме бутона Enter. При натискането му чата извиква “PlayerState” класа на играча, чрез който извиква сървърно RPC със съобщението. PlayerState е клас, който съдържа информация за играча, която не е пряко свързана с игровия елемент на играта. В него се съдържа например името на играча.

```
	UFUNCTION(Server, Reliable, WithValidation)
	void ServerSendMessage(const FText& Message, const FText&
Sender);
	bool ServerSendMessage_Validate(const FText& Message, const
FText& Sender);
	void ServerSendMessage_Implementation(const FText& Message,
const FText& Sender);

	UFUNCTION(NetMulticast, Reliable, WithValidation)
	void MulticastSendMessage(const FText& Message, const FText&
Sender);
```

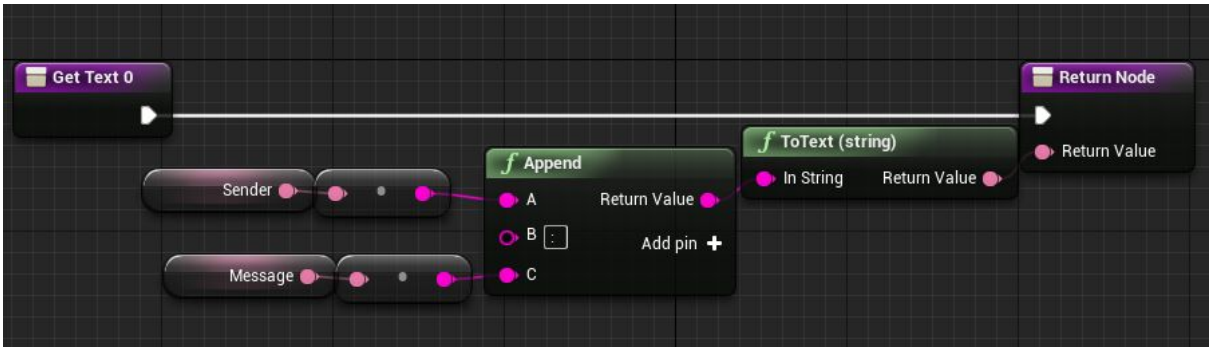
```
	bool MulticastSendMessage_Validate(const FText& Message,
const FText& Sender);
	void MulticastSendMessage_Implementation(const FText&
Message, const FText& Sender);
```

При изпращане на съобщението се извиква функцията “ServerSendMessage”. Тя се извиква на на клиента, но се изпълнява на сървъра. Сървъра не пази лист от всички съобщения. Той просто разпространява съобщението чрез мултикаст RPC. Така пестим памет на сървъра, но ако играч влезе в играта след изпращане на дадено съобщение, той няма да го получи:

```
void ABattlePlayerState::ServerSendMessage_Implementation(const
FText& Message, const FText& Sender) {
	MulticastSendMessage(Message, Sender);
}

void ABattlePlayerState::MulticastSendMessage_Implementation(const
FText& Message, const FText& Sender) {

	UWorld* World = GetWorld();
	if (validate(IsValid(World)) == false) return;

	APlayerController* PlayerController =
World->GetFirstPlayerController();
	if (validate(IsValid(PlayerController)) == false) return;

	ABattleHUD* HUD =
Cast<ABattleHUD>(PlayerController->GetHUD());
	if (validate(IsValid(HUD)) == false) return;

	UChatWidget* ChatWidget = HUD->GetChatWidget();
	if (validate(IsValid(ChatWidget)) == false) return;

	ChatWidget->AddMessageWidget(Message, Sender);

}

void UChatWidget::AddMessageWidget(const FText& Message, const
```

```
FText& Sender) {
	UWorld* World = GetWorld();
	if (validate(IsValid(World)) == false) return;


	if(validate(IsValid(ScrollBox)) == false) return;

	UMessageWidget* MessageWidget =
Cast<UMessageWidget>(CreateWidget(World, MessageTemplate));
	if (validate(IsValid(MessageWidget)) == false) return;

	MessageWidget->SetMessage(Message, Sender);

	ScrollBox->AddChild(MessageWidget);
}
```

Добавянето на съобщение в “ChatWidget” представлява създаване на нов Widget за съобщение, задаване на текст на съобщението и подател и добавянето на новосъздадения Widget към скрол полето. След като стойностите са зададени от C++, те биват прочетени и визуализирани чрез Blueprint



Фиг. 3-22 Логика за показване на текст в потребителския интерфейс

4. Четвърта глава - ръководство на потребителя

4.1. Инсталация

- Отидете в секция “releases” на репото на играта в Github: https://github.com/mikirov/Elsys-Thesis-Work-2019-2020/releases
- Изтеглете последната версия на WindowsNoEditor.rar
- Разархивирайте WindowsNoEditor.rar
- Навигирайте до WindowsNoEditor\BossBattle\Binaries\Win64
- Създайте препратка към файла “BossBattleServer.exe”
- Отворете настройките “Properties” на препратката и добавете накрая на “Target” опцията “-log”.
- Пуснете сървъра на играта чрез файла “BossBattleServer.exe - Shortcut”
- Пуснете клиентска инстанция на играта чрез файла “BossBattle.exe”

4.2. Минимални системни изисквания и препоръчителен хардуер

4.2.1. Ubuntu

4.2.1.1. препоръчителен хардуер

Processor	Quad-core Intel or AMD processor, 2.5 GHz or faster
RAM	8 GB

Graphics card	NVIDIA GeForce 470 GTX or AMD Radeon 6870 HD series card or higher OpenGL 4.1+ compatible graphics card

4.2.1.2. минимални софтуерни изисквания

Operating system	Any reasonably new Linux distro (CentOS 6.x being the oldest tested)
Linux Kernel version	Kernel 2.6.32 or newer
Additional Dependencies	Glibc 2.12.2 or newer

4.2.2. Windows

4.2.2.1. препоръчителен хардуер

Processor	Quad-core Intel or AMD processor, 2.5 GHz or faster
RAM	8 GB
Graphics card	NVIDIA GeForce 470 GTX or AMD Radeon 6870 HD series card or

	higher DirectX 11 or DirectX 12 compatible graphics card

4.2.2.2. минимални софтуерни изисквания

Operating system	Windows 7 or higher 64-bit
DirectX Runtime	DirectX End-User Runtimes (June 2010)

4.2.3. MAC OS

4.2.3.1. препоръчителен хардуер

Processor	Quad-core Intel or AMD processor, 2.5 GHz or faster
RAM	8 GB
Graphics card	NVIDIA GeForce 470 GTX or AMD Radeon 6870 HD series card or higher OpenGL 4.1+ compatible graphics card

4.2.3.2. минимални софтуерни изисквания

Operating system	Mac OS X 10.10.5 or higher

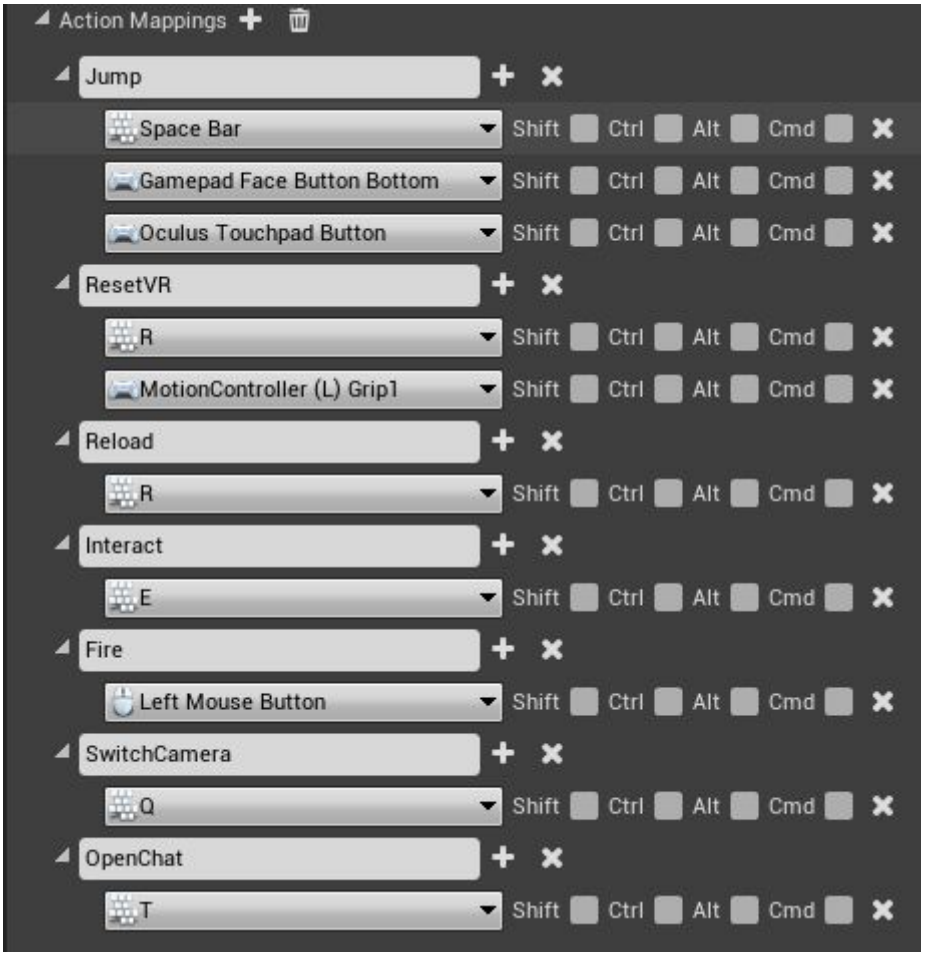
4.2.4. Изисквания към средата за разработка

- Visual Studio 2017 v15.6 or later (recommended)
- Visual Studio 2019

4.2.5. Конфигурация на тестерната машина

Operating system	Microsoft Windows 10 Pro
RAM	8 GB
Processor	Intel(R) Core(TM) i5-7600 CPU @ 3.50GHz, 4 Cores
Graphics card	Nvidia Geforce GTX 1060 3gb
Engine version	4.23

4.3. Контроли



Фиг. 4-1 бутони за контрол на играча

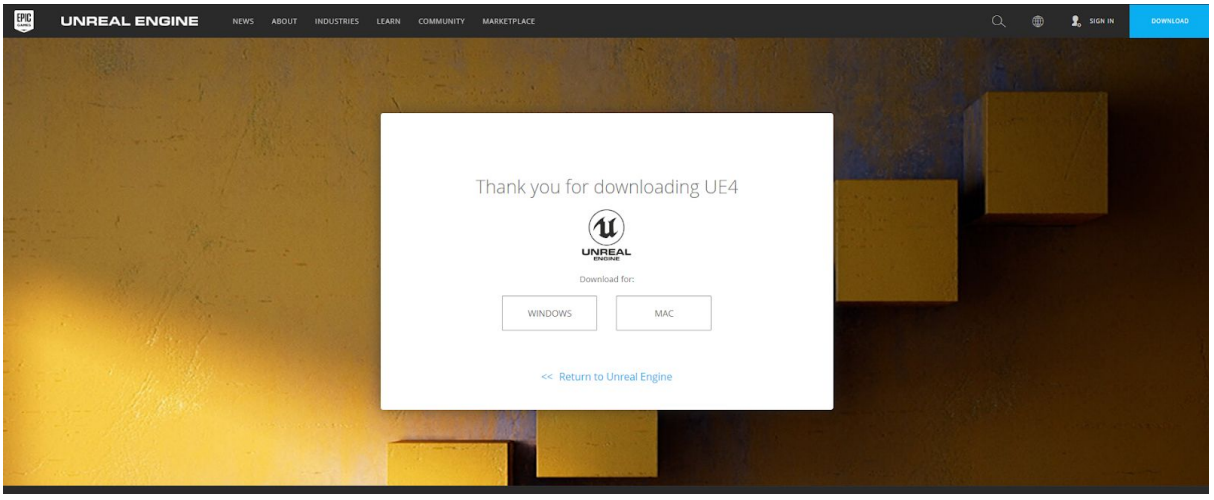
Фиг. 4-2 Аналогови контроли на играча

Показани са различните действия, които играча може да извърши в играча(Фиг 4-1 и 4-2). На всяко едно от тях съответстват различни бутони от клавиатура и контролер. Аналоговите контроли имат така наречен “Scale”. Той представлява коефициент, с който е умножена стойността. Така например задаваме посока на движение напред-назад, използвайки само една функция вместо две.

Изисквания за отваряне на проекта и неговото редактиране

4.3.1. Изтегляне на игровия двигател от официалния сайт на Unreal

Можем да го открием на следния URL: www.unrealengine.com/en-US/download

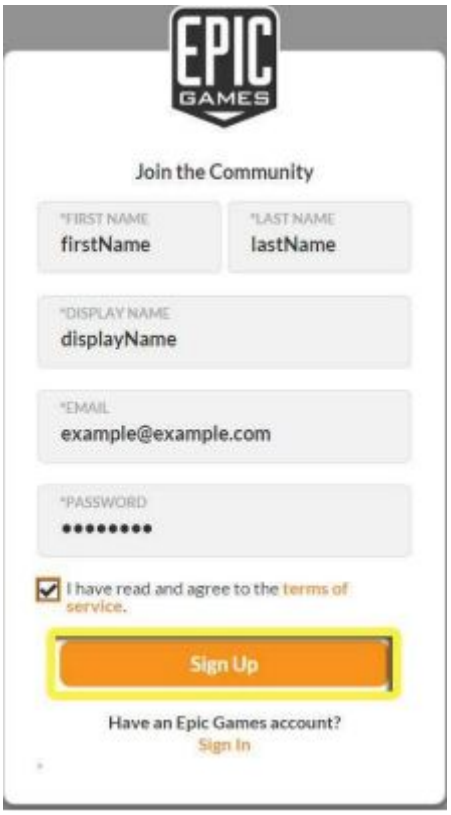


Фиг. 4-3 Страница за изтегляне на Unreal Engine 4

След това избираме съответната операционна система и изтегляме инсталатора.

4.3.2. Инсталация на Epic Games Launcher и създаване на профил в Epic

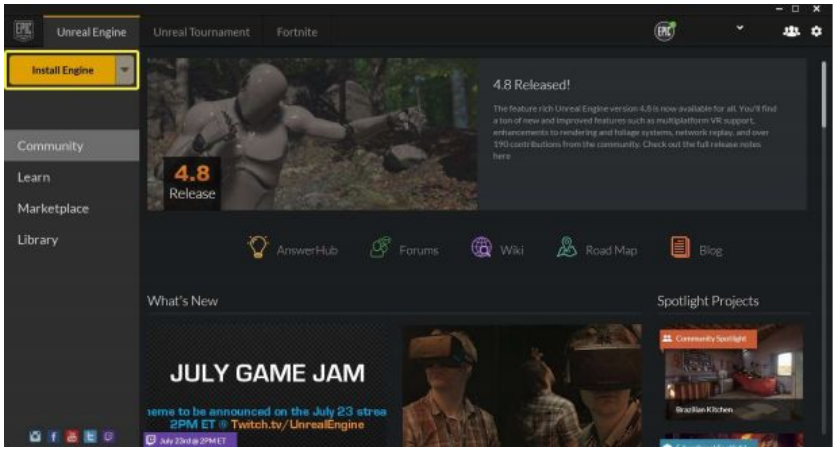
Следваме стъпките на инсталатора и регистрираме акаунт в Epic Games, ако вече нямаме такъв.



Фиг. 4-4 Създаване на акаунт в Epic Games

4.3.3. Инсталация на версия на Unreal Engine 4.

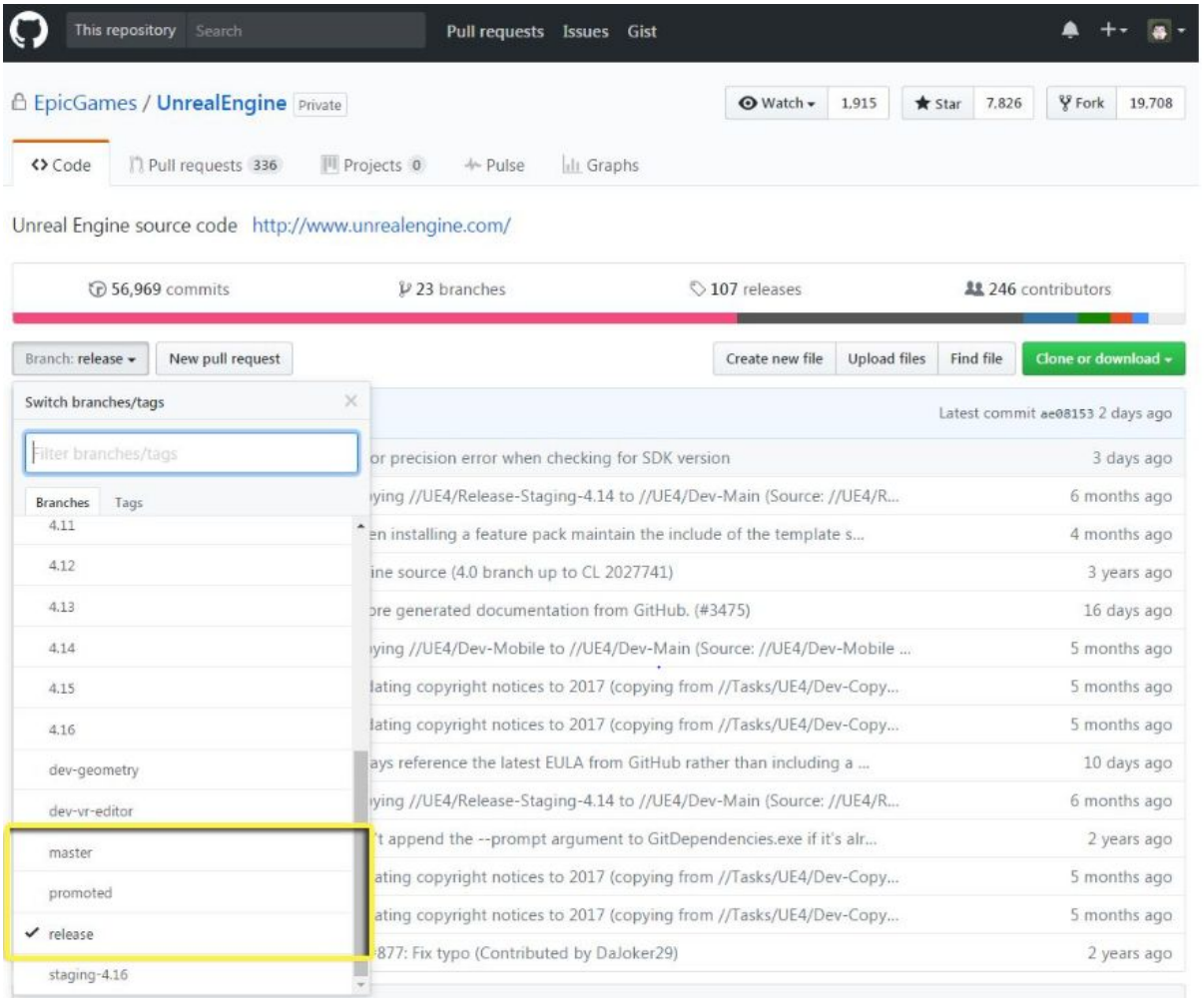
Този проект използва версия 4.23, но версиите са backwards compatible, което означава, че проекта може да бъде отворен с всяка версия над 4.23. Проекта няма да работи с версии под 4.23 поради различния формат на сериализация.



Фиг. 4-5 Меню за изтегляне на Unreal Engine 4

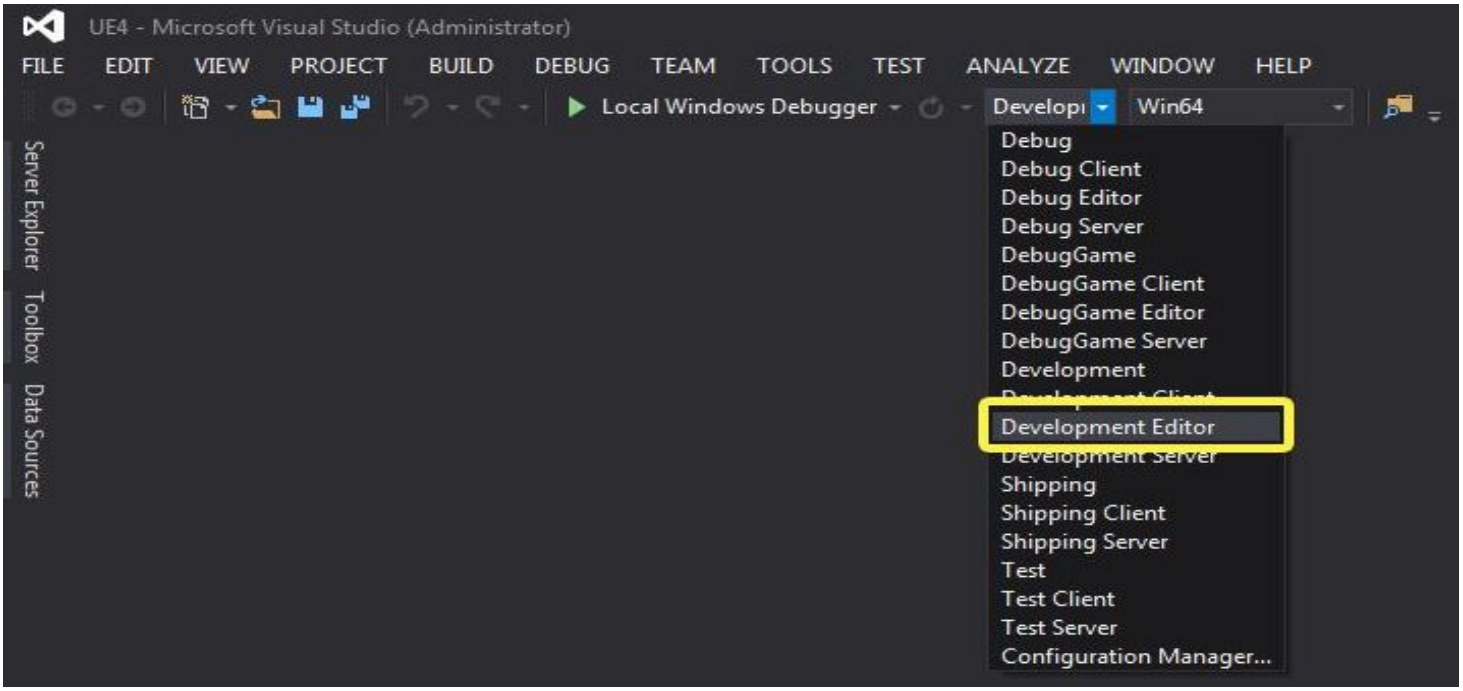
4.3.4. Компилиране на сървърната част

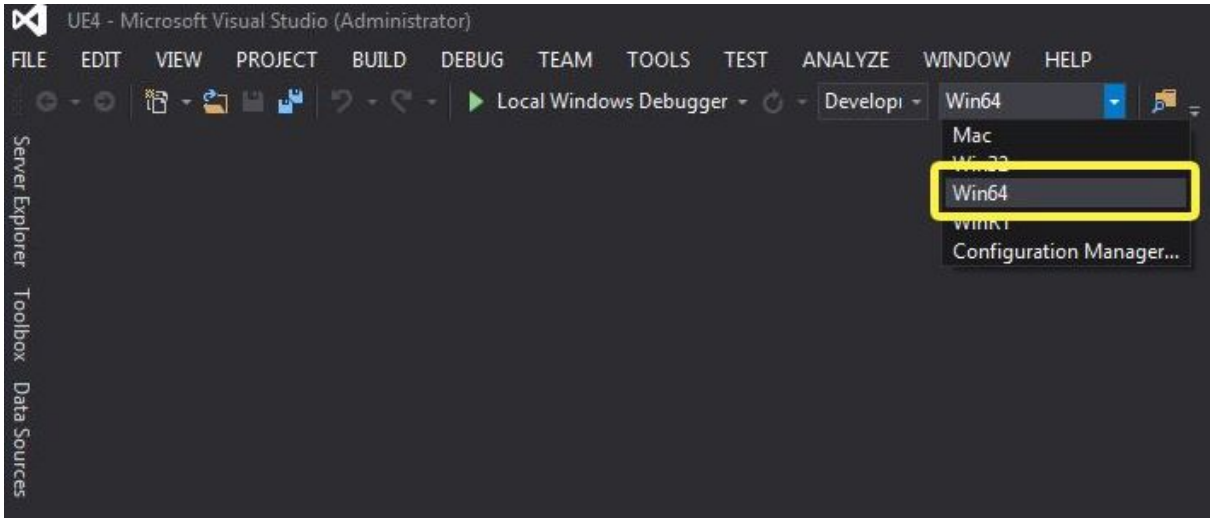
За съжаление бинарния игрови двигател на Unreal Engine 4 не позволява компилация на dedicated server. Поради тази причина трябва сами да клонираме репото на игровия двигател и да го компилираме. Първо Трябва да се свържат Epic Games акаунта и GitHub акаунта. За да го клонираме трябва първо да бъдем одобрени от Epic Games и да бъдем приети в тяхната организация в GitHub. След това можем да разгледаме тяхното репо и изберем master branch



Фиг. 4-6. Гитхъб хранилище с официалния сорс код на игровия двигател

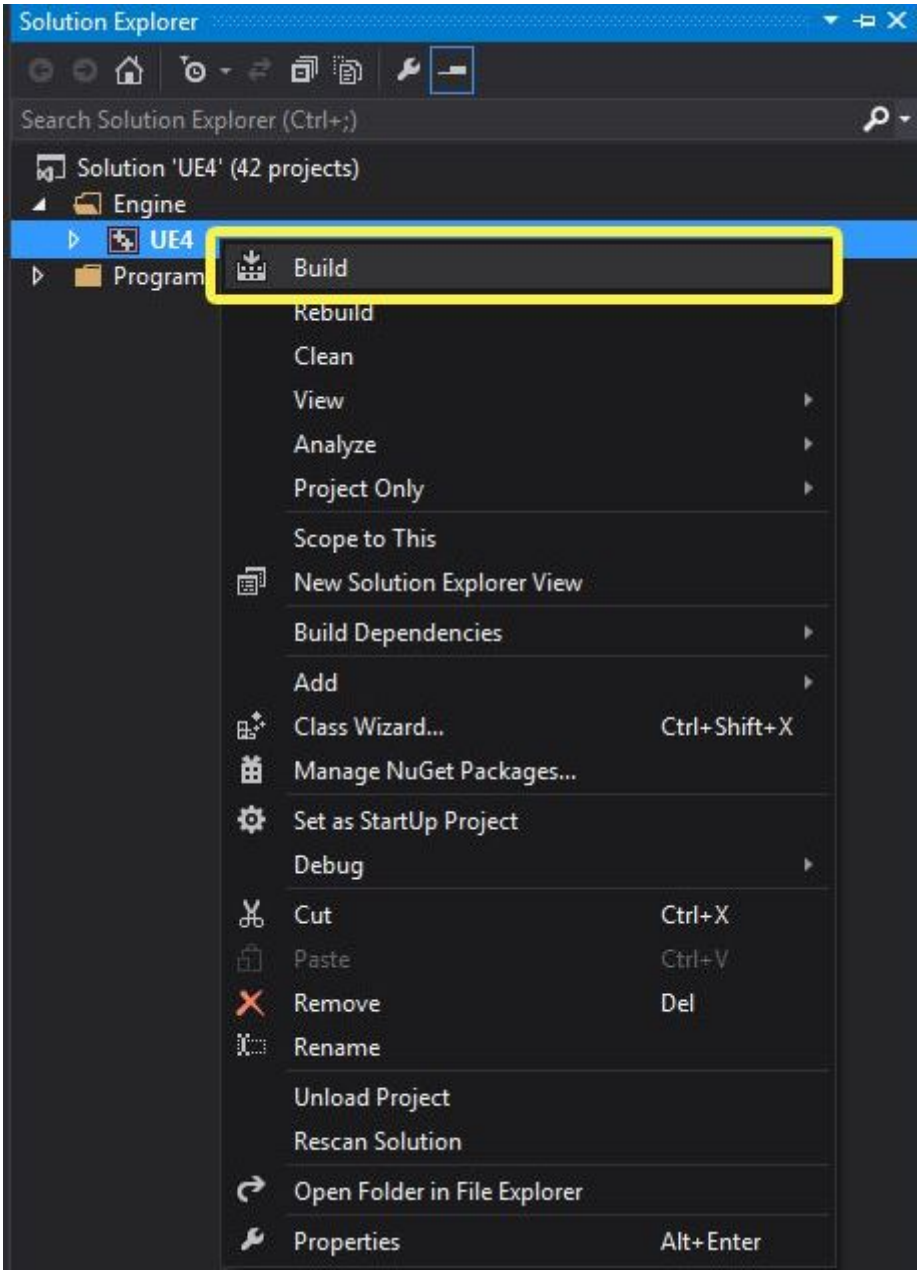
След като имаме достъп и сме клонирали игровия двигател пускаме скриптовете GenerateProjectFiles.bat и Setup.bat, които теглят всички нужни dependencies. Отваряме проекта от файла UE4.sln със Visual Studio, избираме опция за компилация Development Editor за Win64, десен клик върху проекта и натискаме build





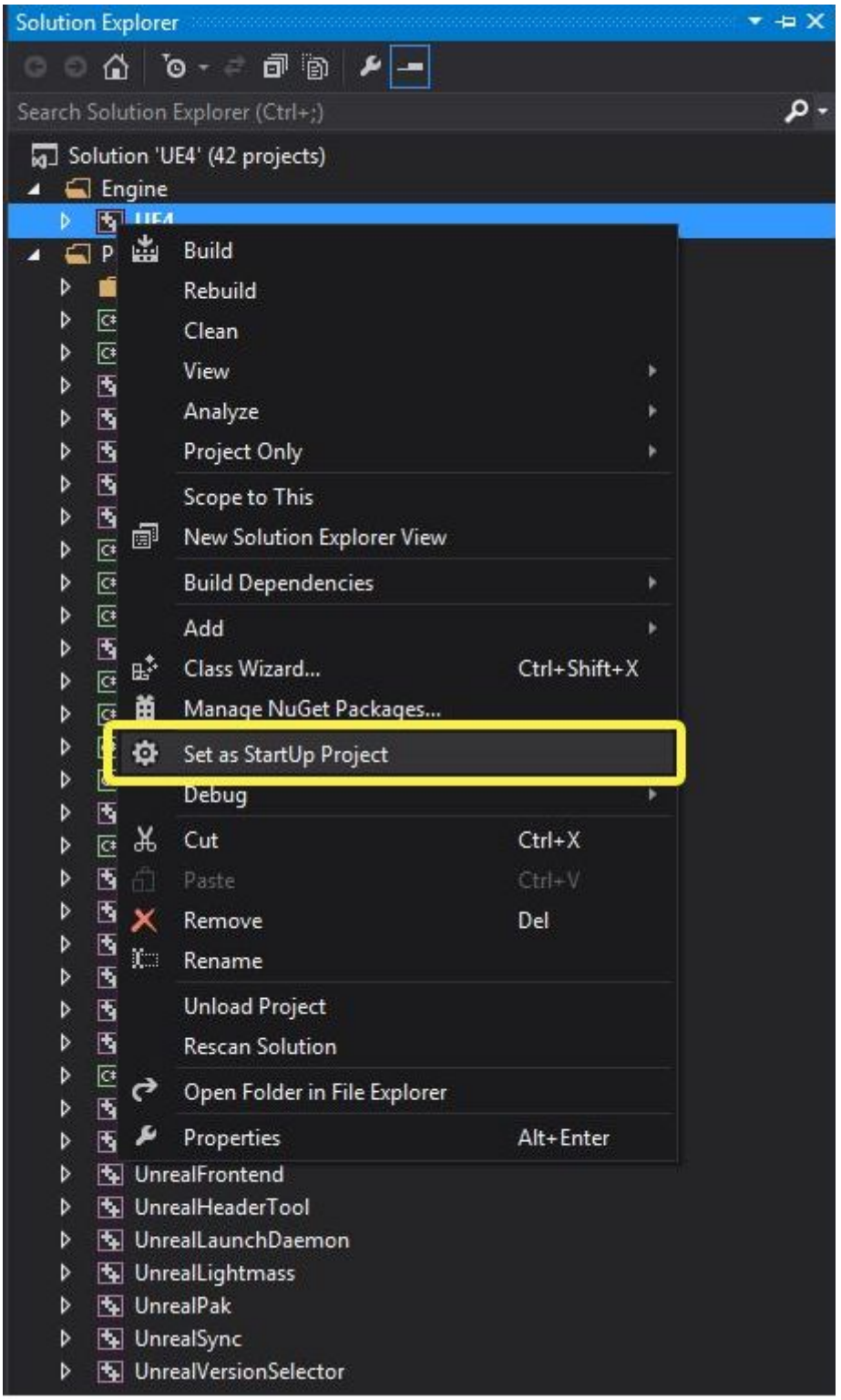
Фиг 4-7, 4-8. Избиране на платформа за компилация

Накрая задаваме проекта като начален проект при отваряне на Visual Studio



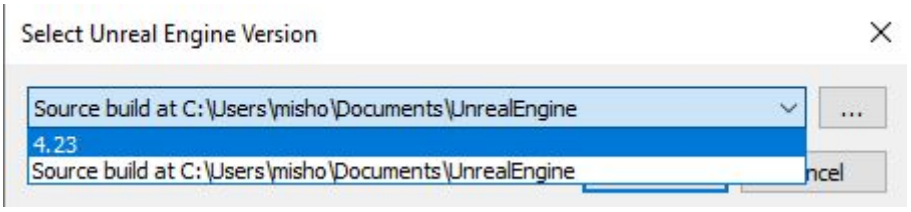
Фиг. 4-9 Компилация на проекта

След като имаме компилиран игрови двигател можем да сменим ползвания от нашия проект двигател от изтегления към компилирания.



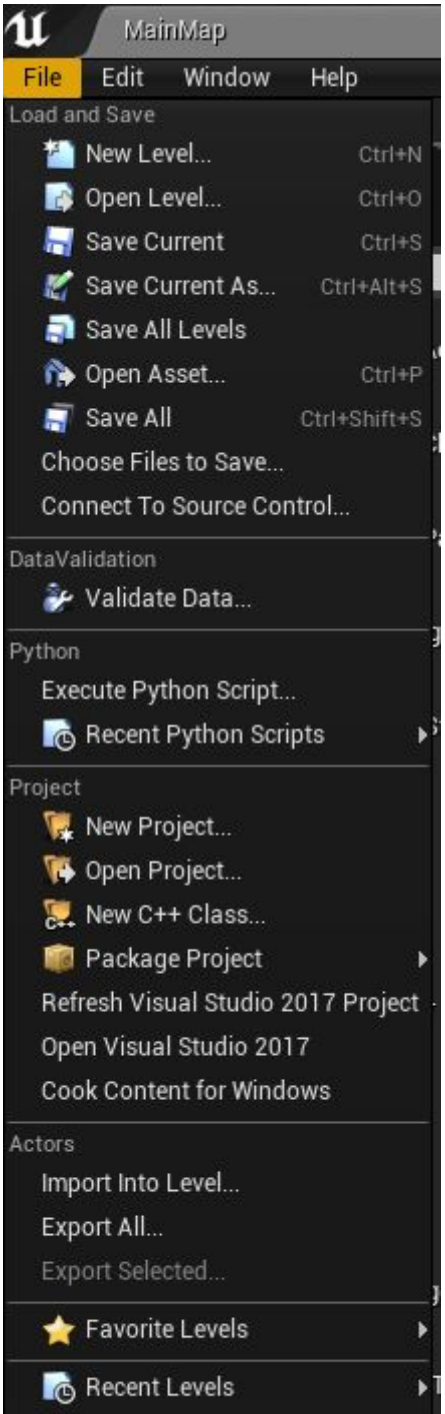
Фиг. 4-10. Задаване като начален проект.

Натискаме десен бутон върху файла “BossBattle.uproject” и избираме менюто “Switch Unreal Engine version ...”

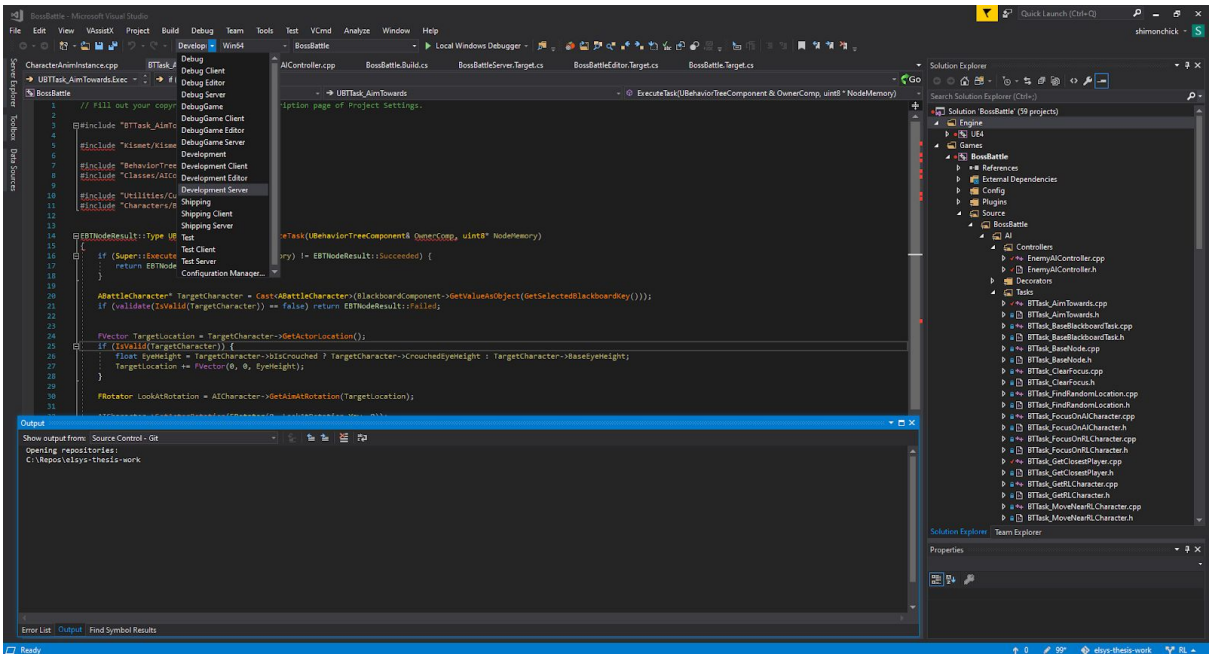


Фиг. 4-11 промяна на използвания от проекта игрови двигател

От падащото меню избираме компилирания игрови двигател. Това ще компилира наново нашия проект и ще го отвори. След като Unreal Editor е отворен, отваряме и кода на проекта от менюто File > Open Visual Studio:



След това избираме опцията “Development Server” и компилираме проекта.



Фиг 4-13 Меню за избиране на режим за компилация

След компилацията бинарния файл на сървъра се намира в \Binaries\Win64\BossBattleServer.exe

Заключение

Всички задачи, поставени в заданието са изпълнени. Това в никакъв случай не означава, че проектът е в напълно завършен вид.

Машинното самообучение чрез утвърждение е много капризно към средата, с която агента взаимодейства. Тя може да бъде променена, за да бъдат получени нови интересни поведения от него.

Разработена бе кооперативна стрелкова игра, в която няколко играча играят срещу противник с изкуствен интелект. Имплементиран бе потребителски интерфейс за управление на тренирането на агента и показване на всичката релевантна информация за процеса на самообучение.

Бъдещото развитие също би включвало добавяне на произволен брой действия, състояния на играта и награди, които да могат да бъдат контролирани от разработчика.

Библиография

- Michael Allar, UE4 Style Guide, https://github.com/Allar/ue4-style-guide, 2017
- Dedicated Server Build for Windows, https://wiki.unrealengine.com/Dedicated_Server_Guide_(Windows_%26_Linux)#Section_3_launching_and_joining_the_dedicated_server
- Unreal Engine API Reference, https://docs.unrealengine.com/en-US/API/index.html
- Unreal Engine Documentation, https://docs.unrealengine.com/en-US/index.html
- Cedric 'eXi' Neukirchen, UE4 Networking Compendium, http://cedric-neukirchen.net/Downloads/Compendium/UE4_Network_Compendium_by_Cedric_eXi_Neukirchen.pdf
- Slate, Simple C++ Chat System, https://wiki.unrealengine.com/Slate,_Simple_C%2B%2B_Chat_System
- Reece A.Boyd, https://pdfs.semanticscholar.org/37f2/e2cb6c2e0e8547aee457a3e1beae6ee7f441.pdf, 2017
- Unreal Coding Standard, https://docs.unrealengine.com/en-US/Programming/Development/CodingStandard/index.html
- Lex Fridman, Deep reinforcement learning, https://www.dropbox.com/s/z4bfr9nifopijdj/deep_reinforcement_learning_2018.pdf?dl=0 , 2018

Приложение 1 - Използвани съкращения

1. UE4 - Unreal Engine 4
2. RL - Reinforcement Learning
3. AI - Artificial intelligence
4. BP - Blueprint
5. UHT - Unreal Header Tool
6. Behavior Tree - дървовиден автомат на състоянията
7. State machine - автомат на състоянията
8. DataTable - таблица във формат csv/json със четими стойности
9. Asset - файл, използван от логиката на играта

Приложение 2 - Сорс код + CD

Github хранилище: https://github.com/mikirov/Elsys-Thesis-Work-2019-2020

CD:

Съдържание