ВВЕДЕНИЕ

Вместе с появлением первых компьютеров люди начали задумываться о создании программ, способных «мыслить». Почти сразу создается отдельная область науки, которая и изучает такие программы. Такую науку и технологию назвали искусственный интеллект. Под искусственным интеллектом в наше время понимается свойство интеллектуальных систем выполнять какие-либо творческие функции, которые обычно считаются свойственными лишь человеку. Так или иначе, «интеллект» в данном выражении стоит понимать не серьезнее, чем метафору. Не стоит этого делать потому, что пусть вычислительные возможности интеллекта и намного превышают человеческие, а также растут с развитием технологий, однако он, искусственный интеллект, не способен понять смысл того, что он делает.

После создания первых компьютеров, появились и первые программы, играющие в игры с человеком. Игровой искусственный интеллект часто отличается от традиционного. Также стоит отметить, что такой интеллект включает и другие алгоритмы, например алгоритмы из теории управления, робототехники, компьютерной графики и информатики в целом. Причинами отличия традиционного искусственного интеллекта является непроходимость персонажей, управляемых искусственным интеллектом, для игрока как в плане сложности прохождения самой игры, так и в плане системных требований, поэтому его принято упрощать. Примером необходимости упрощения являются шахматы. Сыграть в ничью с ботом в них уже является довольно сложным испытанием, не говоря о победе. Интеллект в начале игры просчитывает все ходы и после очередного хода игрока будет просчитывать их заново, находя более выгодные ходы для себя, не оставляя играющему шанса на победу. В новых компьютерных играх ситуация схожа. При создании интеллекта стоит учитывать тот факт, что, если не давать шанса на победу, игрок с высокой долей вероятности не захочет продолжать прохождение.

Игровой искусственный принято разделять на две группы: развлекательный и хороший. Хороший нацелен на победу над противником. Примером как раз может служить программа, играющая в шахматы или шашки. Целью игры с таким интеллектом может служить тренировка или нежелание играть с реальными людьми. Развлекательный, как можно догадаться, преследует довольно простую цель – развлечь противника. Такая категории используется в большинстве современных игр. В пример можно привести такие игры как «Doom», «Star Wars: Battlefront II», где боты не представляют опасности игроку, но успешно справляются со своей задачей развлекать. Для создания такого интеллекта применяются различные методы. Основная задача, однако, остается неизменной – сделать персонажа,

управляемого искусственным интеллектом максимально реалистичным, логичным для играющего, но не слишком сложным.

У игрока не должно создаваться ощущение, что против него играет не реальный человек, а машина, которая пользуется дополнительной информацией, возможно недоступной самому игроку. Желательно минимизировать обращения к информации об окружении такого типа, полагаясь прежде всего на то, что бот условно может слышать и видеть вокруг себя. Это достигается имитированием зрения, слуха, зачастую уже встроенными в ботов, а также дополнительными датчиками, которые разработчик может написать сам.

Целью дипломного проекта является разработка развлекательного игрового искусственного интеллекта, способного подстраиваться под изменения в окружении, условия игры. Данный искусственный интеллект будет относительно универсальным, что означает независимость от игры, в которую будет внедрен, при незначительной настройке параметров контролируемых персонажей. Базовыми персонажами для игры будут являться три управляемых персонажа с логикой различной сложности.

1 ОБЗОР ЛИТЕРАТУРЫ

1.1 Обзор существующих аналогов

На сегодняшний день существует огромное количество игр, в которых в том или ином виде представлен развлекательный игровой искусственный интеллект. Сильно различаются лишь подходы к реализации развлекательной задачи. Ниже представлены три примера, наиболее похожие на разработанный искусственный интеллект. Первые два примера схожи по цели, но имеют различную реализацию в игре. Третий пример будет показывать то, каким создают искусственный интеллект в многопользовательских играх с сюжетной компанией на сегодняшний день.

1.1.1 DooM 1993

Игра Doom была создана в 1993 году компанией id Software. Она является одной из самых значимых игр в истории индустрии. Именно она определила вектор развития игр от первого лица. В ней игрок поочередно исследует уровни-комнаты разной сложности, решает загадки для дальнейшего прохождения, может находить тайники и, конечно, сражается с противниками. Углубляться в устройство уровней не имеет смысла, так как это не является темой данного дипломного проекта и не связано с искусственным интеллектом напрямую. Единственное, что стоит отметить – противники не могут пользоваться окружением. Это обусловлено несколькими причинами, основной является недостаточная вычислительная мощность техники того времени.

Стоит указать возможные действия со стороны игрока, связанные с взаимодействием с противниками. Так как игрок может пользоваться оружием не только дальнего боя, но и ближнего, то и противникам такую возможность необходимо было добавить. Это повлияло на принцип работы интеллекта некоторых противников.

Предшественником данной игры был Wolfenstein 3D, но искусственный интеллект был значительно ослаблен по сравнению с ним. Например, пропала возможность противников позвать на помощь из другой комнаты или попытаться зайти игроку за спину для атаки. Ослаблен он был и по причине увеличения динамичности игры, что потребовало дополнительной мощности.

Основными недостатками, за которые, однако не стоит винить саму игру, являются, как и написано выше, невозможность использования окружения, отсутствие кооперации противников для достижения цели. Также примитивность данного искусственного интеллекта не предполагает какую-либо очередность и приоритезацию атаки противников, что не оставит игроку шанса на победу при встрече с большим количеством врагов. Более подробную информацию про искусственный интеллект в описываемой игре

можно получить в источнике [1]. В данном источнике представлено видео с более детальным пояснением интеллекта и частичным разбором кода игры.

1.1.2 DooM 2016

Данная мультиплатформенная игра по той же вселенной, что и DooM 1993 года, разработана компанией id Software совместно со студией Certain Affinity и издана Bethesda Softworks. Вышла 13 мая 2016 года на Windows, Xbox One и PlayStation 4.

В данной версии игры разработчики старались учесть все недочеты прошлых игр по этой вселенной. Так как игра является шутером от первого лица, разработчики приняли решение, что игра должна быть динамичнее, чем обычные игры. Динамичность игры достигалась за счет особого подхода к прохождению игры. Игрок для более интересного и относительно простого прохождения не должен был стоять на месте, всегда двигаться. Иначе противники начинали точнее стрелять, и, разумеется, приближаться к стоящему игроку и атаковать уже в ближнем бою. Тут стоит отметить, что, в отличие от DooM 1993 года, противникам добавили ограничение на одновременную стрельбу, что одновременно и упрощало игру и делало ее более приятной для прохождения. Как отмечалось ранее, искусственный интеллект, если не ограничивать его, станет непроходимым препятствием и вряд ли будет интересен при прохождении. Более подробно про искусственный интеллект в данной игре можно узнать из источника [2].

Так же, хоть это и не напрямую относится к основной логике интеллекта, было значительно увеличено количество анимации за счет динамического скелета персонажей. Это делает игру более живой с точки зрения наполнения. Это косвенно, но помогает улучшить противников и восприятие игры в целом.

1.1.3 Star wars Battlefront II

Существует несколько игр с таким названием, однако речь пойдет про игру, разработанную в 2017 году. Расчет в данной игре сделан на многопользовательский режим игры. В нем существует несколько видов персонажей, за которых может сыграть игрок или которыми может управлять искусственный интеллект. Специализации каждого из них уникальны, так как каждый класс заточен под особую задачу. Стоит отметить, что искусственный интеллект в данной игре опять же создан прежде всего для развлечения игроков, а не победы над ними. Искусственный интеллект, как говорят разработчики, придает сражениям на планетах ощущение масштабности за счет увеличения количества бойцов, одновременно присутствующих на одной карте.

Игрок, встречающий таких ботов, управляемых искусственным интеллектом, может почувствовать себя более существенным на поле боя,

что так или иначе повышает заинтересованность играющего. Внедрение таких ботов позволило веселиться без необходимости соперничества с другими людьми. Как пишет разработчик: «Визуальные скрипты позволяют ставить перед ИИ следующие цели: идти, защищать, атаковать, взаимодействовать, использовать, искать и уничтожить, а также следовать. В конечном итоге мозг ИИ ищет правильные «клавиши» – стрельба, смещение, рыскание, наклон, прыжок и так далее – для каждого кадра. Это, как если бы робот играл на контроллере и нажимал физические кнопки – только на концептуальном уровне. В случае подключения к игровому движку этот инструмент становится крайне универсальным и может использоваться почти в любой игре…».

В случае данного искусственного интеллекта, который старается симулировать самого игрока, а не просто развлекать своим присутствием, сильно увеличивается вовлеченность реальных игроков. В первую очередь потому, что в игре с такой динамичностью потребуется либо большой опыт игры, либо очень большие старания, чтобы просто различить реального человека и бота, управляющего персонажем. Это, как и говорилось выше, повышает ощущение массовости при боях и ощущение того, что игрок является «героем».

Разумеется, в игре присутствует и однопользовательский режим, где используются боты с таким искусственным интеллектом. Их использование в сюжетной компании никак не отягощает прохождение самой игры, ведь ощущение живости в совокупности с использованием отличной графики окружения, использованием достаточного количества классов персонажей в игре, приводит к тому, что игрок захочет играть в игру дольше. Это и является целью искусственного интеллекта в конечном счете – увеличивать шанс возвращения людей в игру. Более детально с данным игровым интеллектом в источнике, написанном разработчиками [3].

1.2 Unreal Engine 4

1.2.1 Игровой движок

Unreal Engine – игровой движок [4], который разрабатывается компанией Epic Games. Впервые был выпущен в 1998 году и изначальным предназначением которого являлось создание игр от первого лица. После разработки дальнейших версий, стал применяться для создания игр различного жанра.

Благодаря данному базовому программному обеспечению и большому количеству людей, создающих игры на нем, разработчик может при относительно малых затратах во времени на изучение всех аспектов разработки начать создавать игру. При этом не обязательно знать многие вещи связанные с созданием окружения, звуковых эффектов и графического содержимого игры. Разработку также облегчает наличие свободно

распространяемых материалов для создания игры. В данном дипломном проекте они также будут использоваться в связи с недостаточным временем, выделенным на разработку, а значит и на изучение необходимой литературы. Однако вещи, напрямую связанные с разработкой искусственного интеллекта или созданием того, с чем может взаимодействовать персонаж, управляемый им, будут по возможности создаваться без использования готовых решений.

1.2.2 Система визуальных сценариев Blueprint

Система визуальных сценариев Blueprint в Unreal Engine 4 – это полноценная система сценариев игрового процесса, основанная на концепции использования интерфейса, где за основу взяты узлы для создания элементов игрового процесса из Unreal Editor. Как и многие распространенные языки сценариев, он используется для определения объектно-ориентированных классов или объектов в движке.

Данный подход позволяет разработчику использовать все инструменты, обычно доступные лишь программистам. Стоит отметить, что количество возможных ошибок, которые может допустить разработчик при создании проекта, сводится к минимуму. Это достигается потому, что следить за тем, что происходит в логике задачи легче, нежели в привычных языках программирования.

Существует несколько типов blueprint, создание каждого из которых преследует различные цели. Далее будут кратко описаны типы: blueprint class, data-only blueprint, level blueprint, blueprint interface, blueprint macro library и blueprint utilities.

Blueprint class, для которого обычно используется сокращение до blueprint. В общем случае его создание преследует цель добавления функционала для уже существующих классов в игровом процессе. Создаются обычно визуально, что упоминалось ранее, а не путем ввода кода. Они определяют новый класс или тип Actor для последующего размещения на сцене как экземпляра, который будет вести себя, как и другие экземпляры типа Actor, за тем лишь исключением, что добавленная логика скорее всего добавляет функционал, расширяющий его возможности на карте.

Data-only blueprint является классом blueprint, содержащим только код, опять же оформленный в виде графов узлов, необходимые переменные и компоненты, унаследованные от родителя. Данный тип позволяет только настраивать и изменять уже существующее в чертеже, но не добавлять новые элементы.

Level blueprint действует как глобальный график событий уровня. Все уровни в проекте имеют свой план уровня, который создается по умолчанию. Его можно редактировать, но новые level blueprint не получится создать через интерфейс редактора. Все события, которые имеют отношение к уровню или экземплярам объектов, типа Actor, будут использоваться для запуска последовательностей действий в виде вызовов функций или операций по

управлению потоком. Такие чертежи предоставляют механизмы для управления потоковой передачи уровней в Sequencer и имеют привязки событий к экземплярам классов типа Actor, которые размещены на уровне.

Blueprint interface представляет собой набор одной или нескольких функций без реализации, которые можно добавлять в другие чертежи. Это похоже на идею интерфейсов в общем программировании, которая позволяет различным типам объектов взаимодействовать между собой. При этом стоит учитывать, что добавление новых компонентов, переменных или изменение графов невозможно в интерфейсах.

Blueprint macro library является библиотекой, где можно создавать часто используемые функционал. Они могут использоваться в других чертежах. Подробно описывать данный тип чертежей не имеет смысла из-за чрезвычайной схожести с библиотеками в языках программирования.

Blueprint utility или сокращенно blutility, используется только для редактора. А конкретно – выполнения задач редактором или простого расширения функционала того же редактора.

Так как не представляется возможным показать все основные узлы на чертежах, ниже будет кратко описан принцип программирования используя blueprint. Это будет показано на примере оператора условного перехода. Его логика проста для понимания и отлично подходит для объяснения основных принципов.

У данного узла есть 2 входа, один из которых является исполняемым, а второй используется для выбора задействования выходов, типом данного входа является Boolean. Как и в других языках программирования, тип Boolean может иметь два значения, правда или ложь. При этом, для тестирования или, в случае использования других узлов, можно выставлять константные значения, посылаемые на входы узлов. Исполняемый вход используется для построения самой логики.

Все так или иначе сводится к последовательному исполнению кода, что означает, что исполняемые выходы узлов можно соединять с исполняемыми входами других узлов, с помощью чего и строятся функции и иные конструкции в blueprint. Разумеется, стоит учитывать, что пусть читабельность написанного таким образом кода значительно, чем у кода написанного, например, на языке C++, который будет описан в пункте 1.2.3, в некоторых случаях количество ведущих в узлы переменных может достигать большого количества. Что стоит учитывать при создании функций, оптимизируя количество соединяющих линий.

Рассмотрение всех типов переменных в данном дипломном проекте не будет производится в связи с тем, что количество использованных встроенных типов данных слишком велико и не сможет быть описано в пояснительной записке. Ознакомится с документацией к blueprint можно в источнике [5].

1.2.3 Язык программирования C++ в Unreal Engine 4

В разрабатываемом проекте данный способ разработки почти не использовался, но изучался как альтернативный способ создания игровых персонажей, написания необходимых функций для чертежей и прочего. Главной причиной неиспользования является возможность появления ошибок, которые не смогут быть решены автоматически. Но стоит отметить, что после точного определения на какой версии игрового движка будет создаваться приложение, стоит рассмотреть возможность использовать некоторый бесплатный контент, написанный на C++.

Примером такого контента в разрабатываемом проекте является плагин для нахождения пути в пространстве. Он используется в первую очередь для повышения интеллекта на данный момент единственного представленного в игре летающего персонажа – дрона. Плагин был разработан сторонним разработчиком и предоставлен для использования в библиотеке приложения компании Epic Games на бесплатной основе. Написан данный плагин с помощью средств разработки на языке C++. Написание некоторых функций и плагинов полностью на основе чертежей может составить значительные трудности как в плане отладки, так и трудностей, связанных со временем разработки.

Стоит отметить, что программирование на C++ в Unreal Engine 4 имеет свои особенности. К сожалению, в связи с огромным количеством особенностей и, как следствие, невозможностью описать их, а также по причине малого опыта разработки, описание в предоставленной пояснительной записке приводится не будет по этим причинам. Для ознакомления с основами, ключевыми понятиями и особенностями разработки на данном языке в Unreal Engine 4 можно в источнике [6].

1.2.4 BSP-геометрия

Geometry brushes один из удобнейших инструментов для проектирования уровней внутри Unreal Engine. Напрямую с разработкой искусственного интеллекта они не связаны, но для возможности интеллекта корректно пользоваться окружением и для упрощения проектирования большого количества игровых локаций разработчиком, знание BSP-геометрии необходимо. Основной задачей такой геометрии является прежде всего создание именно начального уровня, его геометрии. Данный инструмент больше подходит для создания объектов, способствующих ускоренному тестированию механик игры. Прототип уровня может претерпевать множество правок, а изменение уровня после создания готовой модели затруднено или вовсе невозможно без дополнительных расширений.

Создание кистей происходит простым перетягиванием на сцену необходимого примитива. Существует 6 исходных примитивов, которыми

при разработке данного дипломного проекта пришлось ограничиваться для создания прототипов уровней.

Самым простым и часто используемым примитивом является куб. У него есть 6 главных настроек, связанных с его основной геометрией. Среди них размеры по осям X, Y, Z, толщина стенок, которая работает только при включенной опции Hollow, сама опция Hollow, позволяющая создавать полости в кубе и настройка Tessellated, которая позволяет разделять стороны куба на треугольники или квадраты в зависимости от выбранной опции.

Вторым примитивом является конус. У него есть такие настройки, как высота по оси Z. Функция Hollow, кратко описанная выше. Cap Z – высота внутренней области, работает при включенной опции Hollow. Inner и Outer Radius, радиусы основания конуса, при этом Inner radius будет учтен опять же только при включенной функции Hollow. Sides – количество сторон конуса, так как конус не может быть идеально гладким из-за невозможности отрисовки. Align to side – настройки выравнивания объекта.

У цилиндра присутствуют настройки высоты, внутреннего и внешнего радиуса, количество сторон, причина такой настройки коротко объяснена в описании конуса. Так же присутствует функция Hollow и Align to side тоже описанные выше.

Также существует три типа примитивов лестниц различных по настройкам и, соответственно, по применению. Первый тип примитива лестниц – простая прямая лестница. Настройки данного примитива ограничены и представляют собой набор констант, определяющих размеры ступенек, их количество и сколько добавлять к первой ступени. Константы, которые позволяют редактирование ступеней – глубина, высота и ширина одной ступени.

Существует также изогнутая лестница, в ней можно изменять радиус внутреннего цилиндра, вокруг которого образуется лестница. Помимо описанных настроек для прямой лестницы, присутствует характерные только изогнутой лестнице настройки: угол поворота лестницы и обратное вращение лестницы.

Последним типом лестниц является спиральная лестница, помимо настроек, существующих у предыдущих видов, присутствует четыре дополнительные настройки, такие как толщина ступени, возможность плавного спуска, возможность установки гладкой поверхности под ступенями и количество ступеней на полный оборот лестницы.

Последним рассматриваемым примитивом является сфера. Настройка данного примитива проста, так как из встроенных опций есть всего две: радиус и управление количеством сторон самой сферы.

При это всем не получится объединять несколько примитивов для создания одного объекта. При создании геометрии, перед перетягиванием на сцену необходимого примитива, можно указать тип кисти – добавление или вычитание. При добавлении он добавится на сцену, а при вычитании, он будет вычитать из других объектов пересекаемый с ним объем подобно

булевой операции. Помимо выбора типа кисти можно устанавливать приоритетность, что может помочь при создании более сложных объектов.

После создания примитива перетаскиванием его на сцену, появляется возможность изменять его, перетягивая грани, точки или стороны примитива в стороны. Это позволяет сосредоточится на создании проработанного окружения, а не на точности постановки объекта на уровне. После того, как нужные объекты из BSP-геометрии расставлены на уровне, есть возможность создать из них статические объекты. Это становится необходимым, если, например, есть необходимость использовать такой же объект несколько раз в игре или при большом количестве геометрии. Большое количество таких примитивов может привести к уменьшению производительности из-за расчета процессором самой геометрии на карте. Поэтому конвертация в готовые модели необходима по завершению прототипирования уровня.

1.2.5 Искусственный интеллект в Unreal Engine 4

При создании игр часто приходится писать искусственный интеллект для нее. В Unreal Engine 4 присутствуют встроенные классы, функции, макросы и прочий функционал, в целом облегчающий написание интеллекта для игры. Краткое описание того, что необходимо знать для создания и последующей разработки искусственного интеллекта с помощью средств Unreal Engine 4 будет представлено ниже.

Первое, что стоит описать является AIController. Это нефизический Actor, который может контролировать персонажа. С его помощью можно передавать информацию самому персонажу. Если проводить аналогию, то контроллер это голова персонажа, контролируемого искусственным интеллектом. В контроллере принято писать логику, отвечающую за нахождение чего-либо вокруг, такую как зрение, слух и прочие чувства, которые разработчик сочтет необходимым добавить для улучшенного восприятия игроком интеллекта. Подробная настройка контроллера не приводится по причине ее ситуативности при разработке. Функции, которые могут помочь в написании искусственного интеллекта могут использоваться в контроллере, но это не приветствуется для написания относительно продуманных персонажей. При дальнейшей разработке их использование в контроллере может замедлить разработку и увеличить сложность самого алгоритма.

При создании качественных персонажей, контролируемых искусственным интеллектом, принято использовать Behavior tree. Если контроллер можно считать за голову, то это своего рода мозг. Программирование интеллекта в нем упрощено за счет еще лучшего разделения поведения на простые задачи. В дереве поведения бывает четыре типа узлов, по-другому их называют нодами. Первые два – задачи и композиты.

При создании Behavior tree в центре экрана уже будет добавлен композит Root. Его нельзя переопределить, удалить или изменить. Это главный корень исполняемой логики, стартовая точка дерева поведений. Единственное, что необходимо сделать для начала выполнения написанного алгоритма – выбрать необходимое дерево поведений и запустить его из контроллера. Обычно это делается при начале игры или сразу после получения контроля над персонажем. Описание всех встроенных узлов можно более подробно изучить в официальном источнике [7]. Далее будет предоставлено лишь их краткое описание.

Задачи – узлы, за которыми закреплена какая-либо логика, от простого ожидания некоторого времени на месте до цепочек исполняемых задач. Само дерево поведения требует использования композитов. Оно, дерево, может состоять из огромного множества ветвей, иначе называемых поведениями. В корне, над задачами в ветвях, есть корень, композит. Композит в общем случае выбирает задачу, которая будет исполнена. У главного корня в дереве есть только один выход, от которого может идти сколь угодно большое количество ветвей. Задачи являются конечной точкой, после них нельзя закрепить композитов или других задач ниже. Стоит помнить, что исполнение будет начинаться слева направо, это значит, что в дереве поведений учитывается расположение задач и композитов. Очередность выполнения задач для удобства пишется в правом верхнем углу задач и композитов. На рисунке 1.1 показан простой пример создания дерева поведения и очередности выполнения узлов в нем.

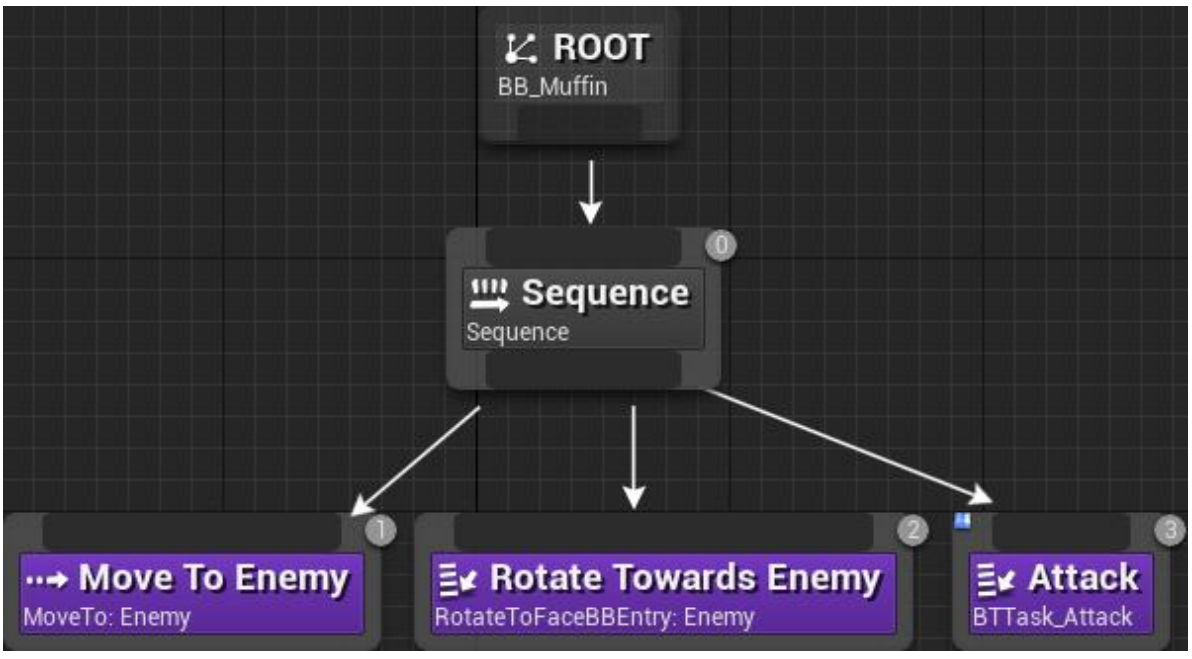

Рисунок 1.1 – Очередность исполнения задач в Behavior tree

В стандартных деревьях поведений есть уже созданные задачи, которые позволят написать легковесный искусственный интеллект, но в

большинстве случаев разработчик будет вынужден писать собственные задачи, что и позволяет делать Unreal Engine. Разумеется, для корректного использования деревьев поведений, требуется какой-либо ресурс для хранения переменных, используемых в задачах. Для этого используется Blackboard. Он прикрепляется непосредственно к дереву поведений. Его использование не обязательно, но может значительно повысить качество разрабатываемого искусственного интеллекта.

Помимо задач и композитов есть еще службы (сервисы), которые можно прикреплять к задачам или композитам. Они также могут использовать и изменять переменные, хранящиеся в blackboard при необходимости. За ними может закрепляться разная логика. Это может быть простое событие при начале выполнения задачи, например перед патрулированием это может быть выбор следующей точки, к которой необходимо пройти, или длящаяся на протяжении всего выполнения ветви для отслеживания или выполнения какой-либо задачи параллельно. Примером параллельной задачи могут служить дальние атаки противников при определенных условиях, устанавливаемые разработчиком.

Последними рассматриваемыми узлами в деревьях поведений являются декораторы. Они также присоединяются к задачам или композитам. Их можно ассоциировать с операторами ветвления. При возвращении декоратором значения true, дальнейшее выполнение ветви продолжается, иначе ветвь блокируется и идет выбор ветвей правее. Совместное использование четырех узлов позволяет составлять задачи для игрового искусственного интеллекта. Искусственный интеллект, написанный с их использованием, сможет быть легко дополнен без существенного изменения уже существующей логики.

2. СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

После того как определены требования к функциональности разрабатываемой системы, ее следует разделить на функциональные блоки. Данный подход облегчит понимание системы, устранит проблемы в архитектуре проекта и позволит создать гибкую и масштабируемую систему для работы в будущем. Но стоит отметить, что деление на функциональные блоки довольно условно в связи с особенностями разрабатываемого проекта.

Всего было выделено восемь блоков: блок приема и обработки зрительной информации, блок приема и обработки звуковой информации, блок приема и обработки информации о получении урона, блок предсказаний, блок принятия решений, блок выполнения задач, блок коммуникации между объектами и блок приоритезации данных органов чувств.

2.1 Блок приема и обработки зрительной информации

Блок приема и обработки зрительной информации предназначен, как можно понять из названия, прежде всего для имитации зрения персонажей, контролируемых искусственным интеллектом. Из-за относительной сложности построенной системы, обработка данной информации происходит не четко выделенными функциями, а множеством связанных функций, использованных в разных частях искусственного интеллекта. При этом используется относительно простая в реализации логика для приема самой информации.

Основную часть работы по приему зрительной информации делают встроенные функции и компоненты, специально предназначенные и оптимизированные для приема такого рода информации. После добавления необходимого компонента необходимо настроить данный компонент на поиск необходимых персонажей. Далее появится необходимое событие на появление в зоне видимости объектов типа actor. Для корректной работы необходима фильтрация данного массива объектов и их приоритезация.

2.2 Блок приема и обработки звуковой информации

Блок приема и обработки звуковой информации предназначен уже в основном для имитации слуха персонажей. В данном случае, система, отвечающая за слух, имеет больший потенциал для использования. Он позволяет всем необходимым персонажам не только условно слышать окружение, но и косвенно взаимодействовать с другими объектами на карте. Опять же, вся логика, отвечающая за слух искусственного интеллекта, находится не в одном месте, а распределена по разработанной системе.

Логика приема и последующей обработки звуковой информации строится на принципе разделения и тэгирования информации. Так, например,

издавая какой-либо звук, можно добавить дополнительную информацию: тэг, громкость, тип и прочее. Это позволяет передавать не только примитивную информацию о звуке, такую как координаты, но и информацию об источнике и способе создания и прочем.

Основную часть работы выполняют встроенные компоненты и функции, но для более корректной и слаженной работы требуется написание дополнительных алгоритмов. От них будет зависеть, сможет ли персонаж, управляемый искусственным интеллектом получать более подробную информацию об источнике звука.

При этом стоит различать звуки, которые может издавать персонаж и звуки, которые слышит игрок при прохождении. Это разные как по целям, так и по функционированию. На основе дополнительной информации можно строить дополнительную логику, совершенствуя интеллект персонажа. Об этом будет более подробно изложено в пункте 2.7 при описании блока коммуникации.

На рисунке 2.1 можно немного детальнее понять, как работают компоненты для блоков, описанных в пункте 2.1 и пункте 2.2. На рисунке показано, как используются описанные компоненты.

Для условного органа слуха создается сфера, она традиционно обозначена желтым цветом (цвет не имеет значения и может быть изменен). Всё, что произойдет в сфере, может быть услышано и обработано ботом. При этом существует дополнительная сфера, она не представлена на рисунке в силу большого радиуса и того, что не задействована при разработке. Зеленым показаны границы конуса, используемого как зрение. Все, что попадает в него, также может быть обработано ботом. При это так же важно, что будет именно зрительный контакт, что не будет каких-либо объектов между наблюдателем и наблюдаемым. Для сферы, использованной для слуха это значения не имеет – между наблюдателем и наблюдаемым объектом может быть сколь угодно большое количество объектов.

Разумеется, для правдоподобной системы стоит воспользоваться более сложной логикой. Учитывать при этом тот факт, что звук не может быть услышан за стеной. Но для данного проекта использование встроенного компонента и отсутствие более сложной логики нецелесообразно. Причиной является более сложная система обработки, что повлекло бы дополнительное время разработки. В будущем, при дальнейшей разработке игры планируется добавление такой системы.

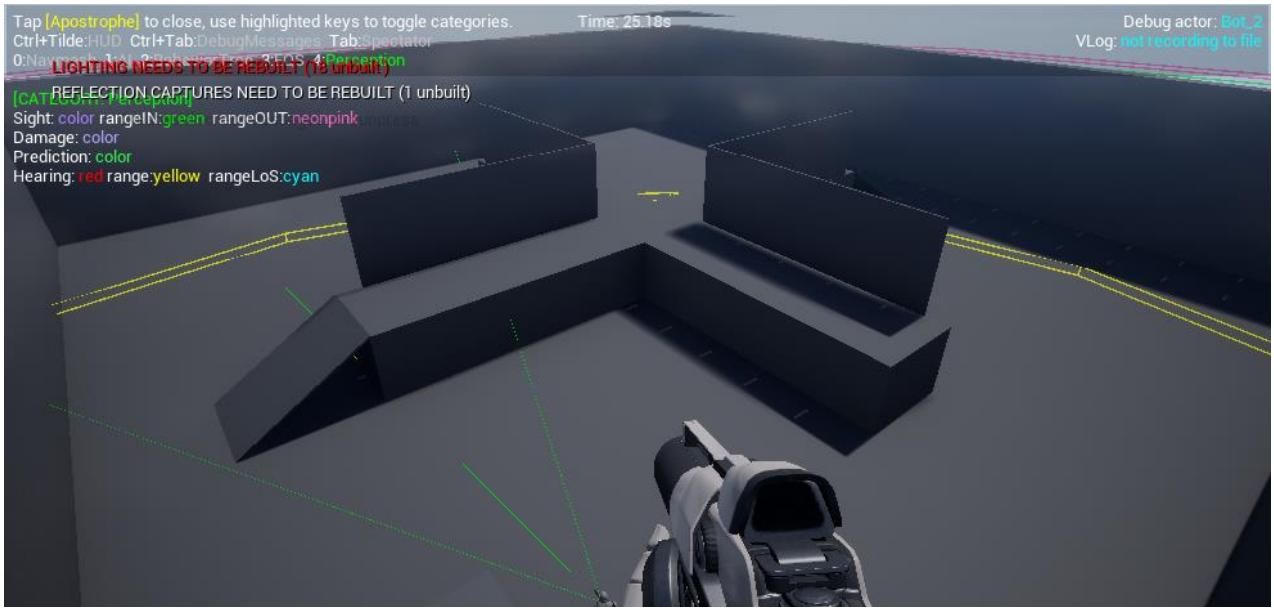

Рисунок 2.1 – Органы чувств искусственного интеллекта

2.3 Блок приема и обработки информации о получении урона

Блок приема и обработки информации и получении урона тоже является одним из встроенных способностей искусственного интеллекта в Unreal Engine 4. К сожалению, данное чувство у персонажей в игре не было реализовано в полном объеме по причине свой ситуативности в использовании. Была использована только базовая информация при получении урона: кем был нанесен, с какого направления, величина урона, нормали попадания.

При создании более качественного интеллекта принято писать собственную систему получения урона для повышения эффективности и многофункциональности. Для использования компонента, отвечающего за обработку получения урона, необходимо добавить его к самому персонажу типа actor во внутреннем редакторе персонажа. Далее, как и в случае предыдущих чувств, появится возможность использовать соответствующее событие на получение урона в графе событий персонажа.

2.4 Блок предсказаний

Данный блок отвечает прежде всего за предсказание действий персонажа при движении. Логика проста для понимания – при выходе персонажа из зоны видимости, за которым наблюдает и/или следует искусственный интеллект, делается предсказание. Предсказание представляет собой расчет возможной точки пребывания наблюдаемого объекта через некоторое время, которое указывается прежде всего в функции напрямую.

Данные, разумеется, можно было взять иным способом, используя данные о местоположении персонажа, запросив их после некоторого промежутка времени. Однако это в некоторых случаях было бы некорректно. Как только персонаж скрывается за условной стеной, может произойти изменение вектора направления движения, чего не может знать персонаж, управляемый искусственным интеллектом.

Уже говорилось ранее, что стоит ограничивать количество таких запросов. Основной причиной желательной минимизации запросов таких данных служит возможная неправдоподобность действий интеллекта при их использовании, что может не понравится пользователю. Добавление такого компонента также не составляет труда – из того же списка доступных при добавлении чувств, необходимо лишь выбрать данный компонент. После добавления появится соответствующий обработчик события. Логика, созданная при помощи данного компонента совместно с использованием blackboard для хранения данных, позволяет добавить интеллекту некоторое подобие памяти.

2.5 Блок принятия решений

Блок принятия решений является одним из основных в системе. Он обрабатывает всю информацию, полученную от так называемых органов чувств персонажа. В некоторых случаях используется дополнительное обращение за данными напрямую к объектам окружения для минимизации хранимой информации. В некоторых случаях хранение данных нецелесообразно так как нет определенных событий для записи их в хранилище. Данный блок по своей сути объединяет всю логику обработки органов чувств и после использования блока приоритезации дает разрешение на выполнение необходимой задачи.

Описываемый блок является максимально распределенным и опять же не может быть описан в группе функций, предназначенных только для реализации этого блока. Для корректной работы он использует не только блок приоритезации, но и блок обработки и выполнения принятых решений. Причинами тому являются, в некоторых исключительных ситуациях, невозможность выполнения поставленных искусственному интеллекту задач. Вспомогательный блок обработки и выполнения не совсем корректно считать блоком обработки ошибок, однако они схожи по выполняемым функциям.

Тут же стоит упомянуть и блок коммуникации, который тоже влияет на принятие решений. При коммуникации между собой, противники получают дополнительную информацию. В исключительных случаях, требуется непосредственный опрос окружения на предмет необходимой информации. Одним из примеров является опрос команды для получения информации о возможном местонахождении игрока.

2.6 Блок выполнения задач

Блок выполнения задач в свою очередь выполняет задачи, поставленные блоком принятия решений. Задачи выбираются не напрямую, а выставлением необходимых переменных в необходимые значения. Например, для того, чтобы начать или продолжить патрулирование, персонаж не должен видеть игрока, или иной объект, угрожающий ему. Это одно из условий выбора данной задачи. При невозможности выполнения какой-либо задачи необходимо установить соответствующие значения переменных и пересмотреть необходимость выполнять поставленную задачу.

Опять в пример можно привести патрулирование. При нахождении игрока, что является целью патрулирования, противник, контролируемый искусственным интеллектом, перестает патрулировать и идет выбор новой приоритетной задачи. Примерами таких задач может быть попытка убежать в укрытие или попытка атаки игрового персонажа, управляемого пользователем. В силу сложности реализации четко разделенных блоков в искусственном интеллекте, блок выполнения задач тоже распределен и не может быть выделен как отдельная группа функций, созданных исключительно для выбранных целей.

2.7 Блок коммуникации между объектами

Блок коммуникации прежде всего является частью обработки звуковой информации. Как уже было кратко описано выше, в описании блока приема и обработки звуковой информации, блок коммуникации использует дополнительную информацию о звуках вокруг. Так как информация о звуке тэгирована, а значит содержит какое-либо закодированное сообщение, то необходима также и особая обработка такой информации. Для этих целей был выделен отдельный блок обработки.

Основной информацией, которую может получить искусственный интеллект, является причина появления звука. Причина включает в себя как источник, так и неоговоренный приоритет обработки информации. Например, при выстреле игрок издает звук, который может уловить противник. Если он его уловит, он может получить примерное местоположение игрока и пропустит патрулирование, решив исследовать область выстрела.

Причиной приостановки задачи патрулирования может быть также условное общение между персонажами, управляемыми искусственным интеллектом. В данном случае персонажи издают звуки с особым тэгом с заданными интервалом. Также есть некоторый приоритет у разных видов персонажей. Так, например, летающий искусственный интеллект, дрон, может посылать информацию персонажам, но при получении одновременно информации от другого идентичного персонажа и дрона, приоритетнее будет информация персонажа, а не дрона. Это связано прежде всего с едва

заметной попыткой сделать общение более живым. Для более интеллектуальных противников приоритетной информацией будет информация, полученная от более интеллектуальных противников. Но за неимением такой, персонаж выберет информацию дрона и выполнит соответствующую задачу.

2.8 Блок приоритезации данных органов чувств

Блок приоритезации является относительно небольшим в разрабатываемой системе, но также он является значимой ее частью. Он тесно связан с блоком принятия решений и в некоторых случаях они могут выполнять максимально схожие по целям задачи. Но их разграничение необходимо прежде всего для обработки исключительных ситуаций.

Он так же связан с блоком коммуникации и блоком обработки и выполнения принятых решений, но связан косвенно, через блок принятия решений. Ярко выраженным примером приоритезации является отдельная ветвь в дереве поведений. Данная ветвь выполняет приоритетную задачу осмотра локации после выстрелов, совершенным игроком, или иных действий, которые персонаж может услышать. Она выполняется после приема информации от блока коммуникации, обработки в блоке принятия решений и опроса возможных персонажей в радиусе.

Остальная часть приоритезации сильно распределена по системе в силу невозможности выделения ее в отдельный блок или подобную функцию. Без данного блока было бы невозможно существование задач, требующих незамедлительной обработки. Приоритеты в каждого объекта на карте или, вернее сказать, уровне, заданы неявно. Они привязаны непосредственно к типу объекта, отдельной переменной или иного способа предусмотрено не было по причине отказа от дополнительной загруженности переменными.

3. ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

После запуска игры, пользователь попадает в главное меню игры. В меню он сможет перейти в настройки игры для изменения графики или звука. Также сможет вернуться в игру, загрузив сохранение, выйти из игры при желании или продолжить игру, если временно приостановил ее по разным причинам. При этом стоит отметить, что загрузка игры предполагает заранее записанные сохраненные игры.

На данный момент в игре не представлена возможность играть по сети интернет с другими пользователями. После дальнейшей разработки и добавлении данного функционала в игру, у пользователя появится дополнительное подменю для поиска и выбора пользователей для совместной игры.

Основной целью данного дипломного проекта является разработка искусственного интеллекта. Остальные части проекта будут рассмотрены довольно кратко, лишь для лучшего понимания структуры проекта, либо не будут описаны вовсе.

После определения в структурном проектировании блоков для создания искусственного интеллекта, необходимо более подробно описать их функционирование. Рассмотрим более подробно функционал искусственного интеллекта. С этой целью необходимо провести анализ основных блоков программы и их зависимостей. Также стоит рассмотреть назначение всех методов и их переменных.

В разработанной игре у искусственного интеллекта выделены следующие блоки:

- Блок приема и обработки зрительной информации
- Блок приема и обработки звуковой информации
- Блок приема и обработки информации о получении урона
- Блок предсказаний
- Блок принятия решений
- Блок выполнения задач
- Блок коммуникации между объектами
- Блок приоритезации органов чувств

Главной целью будет являться подробный разбор разработанных игровых сущностей, управляемых искусственным интеллектом и использованных для упрощенной работы с ними вспомогательных классов. Вспомогательные классы относительно легковесны, но помогают увеличить качество принятых искусственным интеллектом решений.

Так как блоки, выделенные при системном проектировании сильно распределены, то их рассмотрение по пунктам в данном разделе не представлено. Для полного понимания разработанной системы, стоит рассмотреть отдельно взятые разработанные сущности и вспомогательные функции. Как было описано в системном проектировании, в игре присутствует три персонажа, контролируемых искусственным интеллектом:

бот, дрон и автоматическая турель. Далее будет представлено подробное описание каждой разработанной сущности.

3.1 Классы игровых сущностей

3.1.1 Бот

Основным разрабатываемым персонажем, который будет контролироваться искусственным интеллектом, является бот. Данный персонаж в будущем получит дополнительные возможности, задачи, которые можно будет выполнять. Добавление не составит труда благодаря дереву поведения, которое будет подробно описано в данном разделе. После рассмотрения всей закрепленной за ботом логики, будут описаны основные исполняемые задачи. Как бот выглядит внешне, показано на рисунке 3.1.

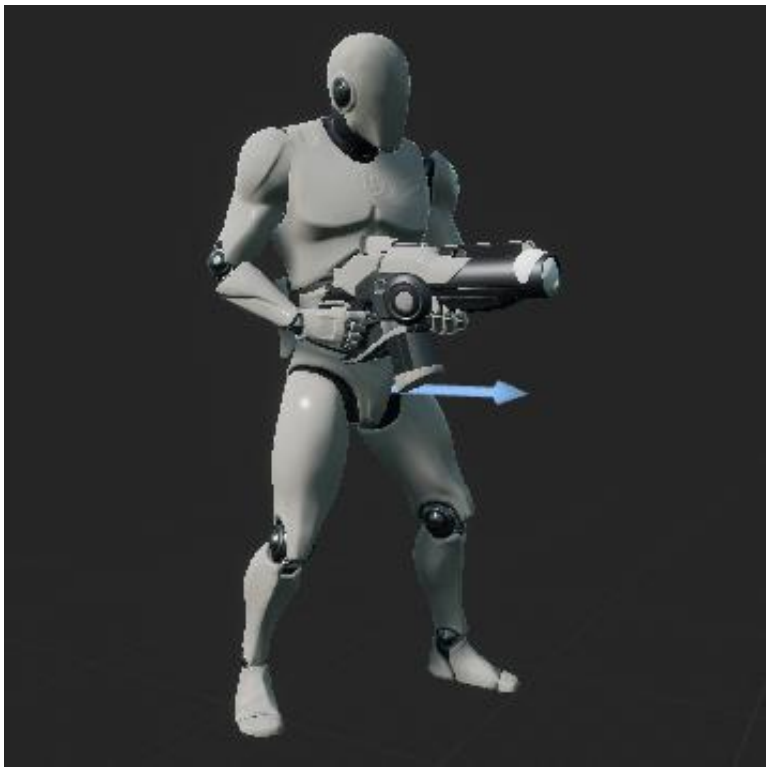

Рисунок 3.1 Внешность бота

Класс `Bot` унаследован от класса `Character` уже встроенного в игровой движок Unreal Engine 4. Класс персонажа был выбран по причине встроенного в него необходимого функционала и компонентов, а именно: компоненты `CharacterMovement`, `SkeletalMesh` и `Capsule Component`.

Компонент `CharacterMovement` необходим для управления движением персонажа и корректной работы анимаций. Он обладает настраиваемыми свойствами для изменения максимальной скорости ходьбы,

полета, плавания, для выбора контроллера искусственного интеллекта, запускаемого для управления данным персонажем и множество других параметров, изменение которых влияет на передвижения и принимаемые решения. Изменение данных параметров будет рассмотрено ниже в этом же пункте.

Второй компонент, `SkeletalMesh`, необходим для использования скелета персонажа, который в свою очередь нужен для работы анимаций и корректной работы физики персонажа после смерти. После его выбора также можно выбрать класс ответственный за анимации. Как уже было сказано выше, основной разрабатываемой частью был именно искусственный интеллект, а не графическая и звуковая части игры. По этой причине работа с анимациями всех созданных персонажей в записке будет опущена. Стоит лишь отметить, что в случае бота и персонажа, управляемого пользователем, анимации были взяты из начального набора разработки Unreal Engine 4. Они добавляются автоматически при создании проекта.

Помимо анимаций персонажа, в начальном наборе присутствует большое количество материалов для создания карт, несколько игровых объектов, уровень для обзора представленных объектов и материалов, а также несколько систем частиц.

`CapsuleComponent` нужен для работы с упрощенной коллизией, которая позволяет обрабатывать события столкновения с какими-либо объектами на карте. Также он позволяет обрабатывать стрельбу в некоторых случаях. Представляет собой обыкновенную капсулу с настраиваемыми размерами и некоторыми другими параметрами.

Обработка коллизии происходит просто – сперва проверяются настройки объектов, связанные с игнорированием коллизии, далее проверяется граничная точка объекта и если эта точка находится в радиусе капсулы, то запускается соответствующая логика. Логика взаимодействия и обработки коллизии уже реализована в игровом движке и не нуждается в повторной разработке.

Помимо встроенных компонентов и переменных, возникла необходимость добавить еще несколько переменных. `FireDelay` – переменная вещественного типа, отвечает за скорость стрельбы. Может изменяться во время игры при необходимости. Для условного баланса значения намного выше, чем у иных персонажей. Это необходимо, поскольку точность стрельбы данного персонажа довольно высока, а урон у оружия, используемого им, тоже высок по сравнению с оружием у других противников.

`CanFire` – переменная типа bool, изменяется либо сразу после стрельбы, либо после определенного времени после стрельбы, отвечает за возможность стрельбы.

`MaxHealth` – переменная, отвечающая за здоровье персонажа, используется лишь однажды – при начале игры для установки текущего

уровня здоровья. Имеет вещественный тип и может меняться в случае необходимости. Но ее изменение в процессе игры не будет влиять на бота по причине неиспользования после начала игры.

`CurrentHealth` нужна для контроля текущего состояния здоровья бота. Ее изменение в процессе игры путем, не предусмотренным игрой, а именно, прямым изменением значения, тоже не повлияет ни на что. Ее изменение привязано к событию получения урона. Переменная вещественного `Damage` используется для возможного добавления к урону оружия и не может быть просмотрена или изменена в процессе игры. Также была необходимость добавить точку для начала стрельбы. Далее в описании стрельбы будет подробнее описана ее функция. Типом компонента данной точки является `StaticMesh`, а названием `FireStart`. Отображение в игре и во вьюпорте отключено за ненадобностью.

В данном классе отсутствуют функции в привычном понимании помимо уже встроенного скрипта создания. Во многих случаях, обработка происходит не в отдельных функциях, а в графе событий. События, они же ивенты, запускают закрепленную за ними логику и, соответственно возможные функции. Разработчики могут создавать собственные ивенты, благодаря чему сильно увеличивается читабельность и уменьшается потребление процессорного времени, затрачиваемого на обработку событий.

У бота присутствует три логические части, такие как сам blueprint бота, его контроллер и дерево поведений. Они будут описываться ниже в этом же порядке.

В чертеже бота были созданы функции и события, которые описывают дополнительные действия персонажа или помогают в их создании. Некоторые действия персонажа уже были реализованы в самом движке, но требовали дополнительной доработки. Их описание будет представлено при описании дерева поведений. Далее будут описаны функции и ивенты, реализованные непосредственно в чертеже бота.

Событие `FireAtTarget` отвечает за обработку события стрельбы по выбранной цели. Цель должна быть заранее выбрана и помещена в необходимую переменную, как ссылка на объект типа персонажа, управляемого пользователем. В случае если она не установлена, обработка прекращается. Тут же идет проверка на возможность стрельбы – бот не может стрелять чаще чем значение, установленное в переменной вещественного типа с названием `FireDelay`. Если эти два условия выполняются, а значит, разрешение на стрельбу получено, бот начинает стрельбу. На данный момент стрельба происходит следующим образом: берутся координаты точки, прикрепленной к компоненте оружия, используемого ботом. Далее берутся координаты цели, берется случайное их изменение для симуляции реальной работы оружия. Производится проверка на наличие между этими двумя точками игрока или иной цели. Если цель найдена на пути – осуществляется попытка нанесения урона объекту. Также

независимо от того, есть ли цель на пути или нет, издается два звука. Один – слышимый игроком звук, звук стрельбы. Второй – технический, для возможной коммуникации между игровыми объектами.

Событие `FireDelaying` является вспомогательным ивентом для работы по изменению переменной `canFire`. Логика работы события проста – по истечению заданного промежутка времени, устанавливается флаг возможности стрельбы по цели. После установки значения и в случае необходимости, бот может продолжить или начать стрельбу.

Событие `AnyDamage` является основным ивентом для обработки получения урона. На данный момент в нем используется только переменная `CurrentHealth`. В начале выполнения закрепленной за данным событием логики, данная переменная уменьшается на значение получаемого урона и начинается проверка, должен ли объект уничтожиться путем сравнения с нулем `CurrentHealth`. Если проверка пройдена и объект подлежит уничтожению, запускается ивент `Death`, подробно который будет описан ниже.

Ивент `Death` запускает логику на уничтожение экземпляра класса. Тут стоит отметить, что для корректной обработки деактивации персонажа следует отключить контроллер управления ботом. Далее для более живой обработки смерти данного игрового персонажа, следует также незамедлительно отключить коллизию, чтобы не мешать проходу других противников и игрока в том числе. Также стоит включить симуляцию физики скелета персонажа. Все изменения происходят сразу же после установки необходимых флагов во встроенных используемых функциях.

Отдельного упоминания заслуживает встроенный ивент на обработку начала игры персонажем - `EventBeginPlay`. За данным событием закрепляется логика, которая должна быть выполнена лишь один раз – при начале игры. Например у бота это задание начального уровня здоровья и установка симуляции физики встроенных компонентов.

Помимо функций, а точнее событий в классе `Bot`, следует описать логику, закрепленную за контроллером, который контролирует действия персонажа. Под действиями понимается не только вызов описанных событий, но и встроенные в `MovementComponent` события на движения персонажа. Из всего функционала в компоненте движения, был использован только бег.

Среди разработанных функций, в контроллере присутствует только одно основное и одно дополнительное событие. Основное событие называется `OnTargetPerceptionUpdated` и оно позволяет обрабатывать информацию сразу всех органов чувств, получаемых от органов чувств, таких как зрение, слух или иные. На данные момент присутствует обработка полученной информации только о персонаже пользователя. После получения данных от органа чувств, необходимо разделить ее для обработки соответствующим обработчиком-функцией. Разбитие данной информации

происходит благодаря встроенной функции ветвления SwitchOnString. После разделения, исполняемые выходы соединяются с необходимыми обработчиками. Далее обработчики будут более подробно описаны.

Как уже было описано в обзоре литературы, использование органов чувств возможно благодаря встроенному в Unreal Engine 4 компоненту AIPerception. В нем можно добавлять как уже существующие органы чувств, так и создать для него новый кастомный орган.

Первый обработчик – функция HearingSense. По названию можно догадаться, что данная функция обрабатывает зрительную информацию. Закрепленная в ней логика проста – так как источники информации уже отфильтрованы, то при заходе в данную функцию остается только обновить данные, которые хранятся в blackboard. Соответственно, обновляются переменные в нем, отвечающие за исследуемую локацию и булева переменная, говорящая об нахождении игрока в зоне слышимости. Далее, после описания функций контроллера, будет разобрано дерево поведения бота и его blackboard. На данный момент их описание будет опущено.

Второй обработчик более легковесен. Это обработчик блока предсказаний. Он лишь выдает информацию в виде координат игрока через некоторое время. Это понадобилось для преследования игрока после потери его из виду.

Следующий обозреваемый обработчик – SightSense. Он наиболее интересен для рассмотрения. В нем используется уже изменение состояния. Под этим стоит понимать тот факт, что AIPerception вызывает ивент для обработки не только в случае обнаружения в зоне видимости игрока, но и при потере игрока из нее же. При обнаружении необходимо выставить значение целевого актера, сфокусировать зрение на нем и выставить такие переменные, которые отвечают за обнаружение. В случае же потери необходимо поставить таймер и попытаться предсказать действия игрока после его скрытия. Для корректной обработки тут и было применено вспомогательное событие – LooseTarget. Оно необходимо для установки нужного значения в переменную seePlayer в экземпляре класса бота.

После описания контроллера и класса бота, стоит рассмотреть наиболее важную часть искусственного интеллекта – дерево поведения. В дереве поведения пишется большая часть искусственного интеллекта для игр. В некоторых случаях, даже разбиение на классы и функции, методы в них и множественное наследование не позволяет создавать качественный интеллект для ботов. Для этого в Unreal Engine 4 предусмотрены деревья поведений. Они уже были кратко разобраны в обзоре литературы и упомянуты в системном проектировании. Сейчас стоит более подробно описать их работу на примере разработанного персонажа.

Сперва стоит кратко упомянуть про blackboard. Blackboard это своего рода хранилище данных класса. Своего рода область полей класса. В нем можно создать переменные всех типов, представленных в проекте, а значит

можно хранить любые данные, которые необходимы для корректной работы искусственного интеллекта. Обычно blackboard закрепляется за деревом поведений, которое в свою очередь закреплено за персонажем.

У прототипа бота в хранилище созданы такие переменные, как TargetLocation, TargetActor, AIhealth, SeePlayer, HearPlayer. SeePlayer и HearPlayer имеют тип Boolean, AIhealth имеет тип float, TargetLocation имеет тип вектор и хранит данные о координатах обследуемой локации, TargetActor имеет тип actor и хранит информацию о цели. Все они используются для построения дерева поведений и некоторые из них используются в сервисах, декораторах и узлах дерева.

В прототипе представлено на данный момент пять ветвей поведений. Под ветвями в данном случае подразумевается пять задач с различными целями. У каждой ветви присутствуют условия выполнения. При этом исполнение ветви может зависеть не только от значений переменных, но и от результата исполнения предыдущей ветви. Так, к примеру, в процессе выполнения задач, закрепленных за композитом sequence, одна из закрепленных задач, которых может быть много, может сработать с результатом false. Это будет означать выход из данного композита и невыполнение идущих правее задач.

Перед описанием дерева поведений стоит описать все сервисы, декораторы и задачи, используемые в дереве поведений бота. В дереве присутствует два сервиса, три задачи и декораторы, необходимые для реализации искусственного интеллекта.

Первым описываемым сервисом является WasHeard. Он помогает лучше обрабатывать информацию от органа чувств, отвечающего за слух. Устанавливает значение HearPlayer, которое отвечает как раз таки за то, был услышан игрок или нет. Особой сложной логики в нем не было необходимости создавать, однако без данного сервиса корректная обработка данной информации затруднена. Сервис работает от события ReceiveTickAI, который вызывает закрепленную логику с некоторым интервалом, закрепленным за деревом поведений. ReceiveTickAI по сути работает при выделении ему времени обработки. Это одна из особенностей работы с чертежами в Unreal Engine 4. Все функции так или иначе выполняются в одном потоке, а значит должна быть какая-то приоритезация выполнений событий. Многопоточность возможна в данном игровом движке, но в дипломном проекте не используется из-за своей специфичности и отсутствия необходимости.

Сервис Shooting призван облегчить работу со стрельбой заданного персонажа. Он также работает с ReveiveTickAI, который позволяет отслеживать расстояние до цели. При нахождении цели в заданном радиусе, на который рассчитано оружие, закрепленное за ним, ему выдается разрешение на стрельбу по персонажу, заданному в хранилище. Стрельба уже была описана выше, при описании класса Bot. Алгоритм в данном

сервисе берет актера, на котором сфокусирован с помощью узла GetFocusActor и пытается провести до него луч. Луч шириной с шар. Значение ширины особого значения не имеет, но стоит учитывать размер снаряда оружия. Луч создается с использованием SphereTraceByChannel. Данная функция выполняет простую проверку по каналу, есть ли на его пути какой-либо объект. Возвращает не только булеву переменную, но и всю информацию, которая касается характера столкновения с лучом, примерно также, как при LineTraceByChannel.

Подробное описание всех выходов данного узла не целесообразно. Для разработки были интересны лишь два выхода: HitActor и Distance. При этом выход Distance был удален на конечной стадии разработки, так как использовался для отладки и балансировки способностей бота. В случае, если на пути не находится никаких непростреливаемых объектов и в случае попадания данной сферы в игрока, бот получает возможность стрельбы.

После рассмотрения сервисов, необходимо рассмотреть подробнее работу с декораторами. Большинство декораторов, используемых в реализации дерева поведения бота в данном дипломном проекте уже реализованы в Unreal Engine 4 – BlackboardBasedCondition. Он является, как говорилось ранее при описании декораторов в обзоре литературы, подобием условных операторов. Проверяется необходимое значение в хранилище blackboard. При этом можно установить несколько настроек работы. Первой самой значимой является наблюдаемая переменная.

Наиболее ярким примером является проверка, находится ли игрок в зоне видимости. За это отвечает переменная SeePlayer. При ложном значении – бот продолжает патрулирование или идет исследовать локацию после стрельбы или иного взаимодействия игрока. Это работа с булевой переменной. С другими типами данных работа немного отличается. При работе с вещественным типом данных можно указать значение, с которым необходимо сравнивать. В любых декораторах можно выставить также и инверсию проверки. Если изначально проверяется выставление значения true, то после изменения данной настройки, можно проверять ложное значение. С вещественными и иными типами данная настройка имеет несколько

Еще одним встроенным декоратором является IsAtLocation декоратор. По названию можно догадаться, что он проверяет, находится ли персонаж в необходимой точек. При этом можно указать не только координаты, но и переменные типа actor. Это позволит не писать дополнительную логику нахождения координат, с которыми сравнивать координаты контролируемого персонажа. Он также подразумевает дополнительную настройку. Позволяет выставить радиус, заходя в который, декоратор выдаст разрешение на выполнение задачи или ветви.

Стоит также упомянуть про возможность использования нескольких декораторов на одном узле. Такая необходимость возникла и в дипломном проекте. Применена была на ветви выполнения уклонения от атак игрока.

Данная ветвь, как и все остальные будут описаны после рассмотрения задач, декораторов и сервисов. При применении нескольких декораторов также учитывается их порядок. На рисунке 3.2 показано применение сразу трех встроенных декораторах на композите Sequence. Как видно на рисунке, на декораторах, как и на композите, присутствует номер. Этот номер – номер выполнения при выполнении в дереве поведений. В случае, если верхний декоратор не позволяет пройти дальше, нижние даже не срабатывают.

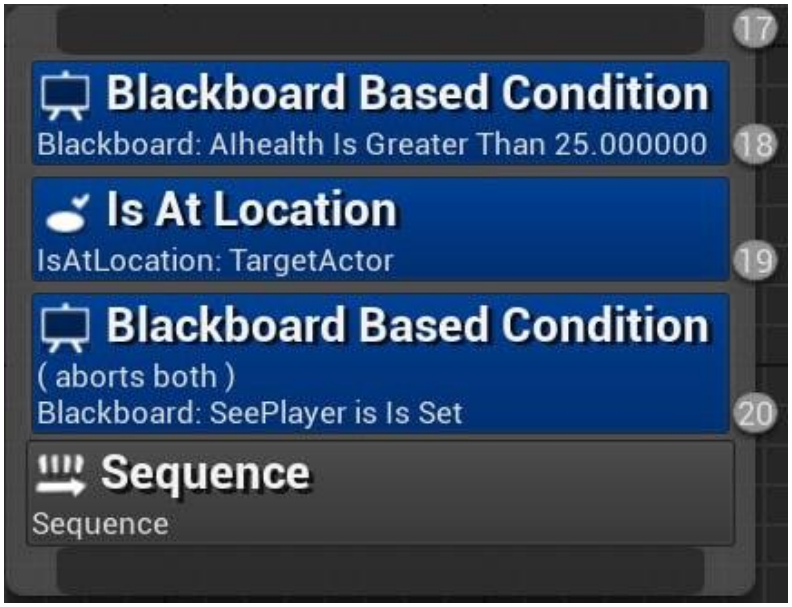

Рисунок 3.2 – Порядок проверки несколькими декораторами

Задачи при реализации бота было решено использовать по возможности встроенные. Одна из них – `MoveTo`. Эта задача позволяет персонажу бегать. Скорость указывается через компонент `CharacterMovement` в переменной `MaxWalkSpeed`. Для использования функции `MoveTo` и схожих по назначению функций, необходимо использование на карте дополнительных объектов.

Одним из этих объектов является `NavMeshBoundsVolume`. Данный объект представляет собой навигационную сетку, используется она для просчета путей на карте. В простейшем случае, создает на земле область, по которой может идти искусственный интеллект. При расположении объекта на уровне, растягивается по всей области, где необходимо пройти ботам. В случае необходимости, возможно модифицирование навигационной сетки – изменение цены пути. При этом создается отдельный объект для модификации и указывается стоимость пути в данной области. Это может использоваться для приоритезации областей, куда может идти персонаж. Также можно указать нулевое значение, которое, по сути, будет запрещать прохождение по области.

Стоит также упомянуть про NavLinkProxy. Он позволяет ботам строить пути даже в случае, когда нет пути, построенного используя только NavMeshBoundsVolume. При этом стоит понимать, что во многих случаях использования NavLinkProxy, необходимо писать соответствующий обработчик для их использования. Примером такого случая может служить прыжок для преодоления препятствий. В этом случае необходимо указать две точки, одна будет служить началом прыжка, а вторая местом приземления. При попадании персонажа в область первой точки, вызывается событие для прыжка. При реализации дипломного проекта данная система была изучена, но ее использование показалось нецелесообразным. Единственным местом, где можно было применить NavLinkProxy были прыжки, но в ходе продумывания проекта, было решено отказаться от них из-за сложности реализации, что потребовало бы большего времени на создание проекта в целом.

Второй задачей является RunEQSQuery. Если задача MoveTo по своей сути не меняется, кроме заданной точки куда двигаться или объекта, то данная задача выполняет другую функцию – функцию поиска подходящих координат для перемещения. В настройках данного узла необходимо выделить несколько блоков: блок выбора Environment Query System (EQS), блок настроек выбора точек и выбор переменной для записи.

Блок выбора EQS позволяет выбирать необходимый заранее созданный объект EQS. Обычно в данном объекте не более одного или двух встроенных узлов. В

При описании данного узла необходимо разделять назначение отдельно взятых объектов EQS объектов. Из-за ограничений, связанных с объемом отдельно взятых разделов, рассмотрение системы EQS будет приведено в этом пункте.

EQS представляет собой систему запросов к среде. Она является одной из функций системы искусственного интеллекта, которая используется для сбора данных среды. Запрос EQS можно вызвать из дерева поведения и использовать для принятия решений о дальнейших действиях на основе результатов тестов. Запросы в основном состоят из генераторов (используемых для создания местоположений или актеров, которые будут протестированы и взвешены) и контекстов (которые используются в качестве точки отсчета для любых тестов или генераторов). Запросы EQS можно использовать для того, чтобы инструктировать ИИ-персонажей найти наилучшее возможное место, обеспечивающее прямую видимость игроку для атаки, ближайший пункт сбора аптечек или боеприпасов или ближайшую точку укрытия.

Далее будет рассказано более подробно о используемых системах EQS. Patrol_EQS используется для нахождения точки на карте, куда должен пройти персонаж, контролируемый искусственным интеллектом. При этом используются уже заранее расставленные на карте объекты типа Bot_POI,

которые по своей сути являются объектами класса actor с минимальными добавлениями в функционале. Вокруг них будет генерироваться множество точек, среди которых бот сможет выбирать необходимую для продвижения. Выбирается точка по ее весу, приоритету в заданный момент времени.

В случае, если точек с одним весом несколько, возможно выбрать одну из них, путем случайного выбора. В этом случае задается вероятность выбора точек 5 или 25 процентов. Расставление точек приоритета не составляет особого труда – нужный экземпляр объекта выставляется на карте без дополнительных настроек.

`Avoidance_EQS` прежде всего используется после нахождения игрока. Ее целью является заставить действовать бота более живо. Это достигается при помощи попыток уворота от атак противника. При этом учитывается возможность стрельбы после прохода к выбранной точке. Координаты выбираются после такой же настройки весов. Веса настраиваются не только после проверки удаленности от игрока, но и после проверки угла поворота того же игрока (куда смотрит игрок). Чем дальше будет находится персонаж, контролируемый искусственным интеллектом, от условного луча направления. Данная система не обязательна для реализации, однако добавляет боту сложности, что в современных шутерах приветствуется игроками.

Последняя система запросов, которую необходимо рассмотреть – `Retreat_EQS`. Она используется в дереве поведения для поиска подходящего укрытия. Задача довольно проста как в реализации, так и в понимании – найти укрытие для бота, чтобы он мог уклоняться от атак условного противника. Для поиска требуется создавать точки вокруг бота на определенном удалении и просматривать, сможет ли игрок атаковать, если бот будет находится в заданной точке. Это условие проверяется функцией `HasLineOfSight`, которая проверяет наличие объектов между двумя точками путем испускания луча. Данная система также не является обязательной, но опять же используется для большего ощущения живости игровых персонажей.

После описания всех составных частей остается рассмотреть логику всего дерева поведений. А конкретно всех ветвей в дереве поведений, которые используют написанные задачи, декораторы и сервисы. Рассмотрение будет проводится не по порядку выполнения при запуске игры, а в случайном порядке. Причиной является случайность, непредсказуемость последовательности выполнения ветвей.

Первая рассматриваемая ветвь в прототипе бота выполняет исследовательскую задачу. Она позволяет боту исследовать местность после того, как будет услышан звук от игрока или иного противника, например выстрел, взрыв, открытие двери или иные звуки. В ней используется композит sequence, на котором закреплено два декоратора. Один проверяет, находится ли игрок в зоне видимости, второй проверяет услышан ли игрок и

является своего рода триггером, так как сбрасывается сразу после начала выполнения задачи. К композиту закреплена единственная задача – MoveTo. Она уже была описана выше. Использует в данном случае переменную TargetLocation. Сервис WasHeard вспомогательный и не является важной частью логики в ветви.

Вторая ветвь основная для данного персонажа – ветвь патрулирования. Также использует композит sequence, но уже с использованием декоратора, проверяющего виден ли игрок для бота. Если не виден – задача на патрулирование может начать выполняться. Для реализации потребовалось использовать две задачи. Первая запускает систему запросов для поиска необходимой точки, к которой необходимо пройти боту. Вторая – MoveTo, которая позволяет боту двигаться к нужной точке, простраивая к ней путь.

Третья ветвь нужна для следования к игроку, в случае если он находится в хоне видимости, но на большом расстоянии от бота для стрельбы. В этом случае выставляется настройка в узле дерева поведений MoveTo, которая отвечает за допустимый радиус вхождения. Состоит из двух задач MoveTo. Первая будет выполняться в случае нахождения игрока в зоне видимости, а вторая отвечает за преследование – завершает ветвь в целом.

Четвертая ветвь не обязательна, но добавляет живучести боту и интереса при прохождении. Она отвечает за уворачивание от атак игрока. Основную задачу выполняет уже описанная выше система опроса среды – Avoidance_EQS. При этом должно выполняться условие, что у бота количество здоровья выше четверти его изначального здоровья и также нахождение игрока в зоне видимости. После нахождения точки, запускается узел MoveTo.

Последняя ветвь также не обязательна для реализации. При малом количестве здоровья, учитывая, что остальные ветви не работают, бот начинает отступать в зоны, где игрок не сможет открывать по ним огонь. Далее находится там некоторое время, которое в будущем будет зависеть от выставленного параметра храбрости персонажа.

Созданный бот представляет собой персонажа, основной задачей которого по сути является охрана территории, заданной точками интереса. В процессе патрулирования, бот может обнаружить противника и начать атаковать. При нахождении рядом с противником, бот будет стараться уворачиваться от атак. А при малом количестве здоровья скрываться, убегать от игрока, стараясь остаться целым. В будущем, как и говорилось ранее, данный персонаж будет дорабатываться и получит новые задачи для возможного исполнения на картах.

Дерево поведения из-за важности и невозможности вставить непосредственно в записку было решено разместить на отдельном плакате, данный плакат представлен в приложении Г[x]. При описании ветвей-задач, как и говорилось ранее, сохранялась последовательность такая же, как и в

дереве поведений. Слева направо были описаны задачи, возможные действия бота на данный момент. На дереве поведений так же указаны и номера у задач, сервисов, декораторов и композитах, благодаря чему можно проследить порядок их исполнения.

3.1.2 Дрон

Дрон представляет собой обыкновенного летающего персонажа, который менее развит, чем бот, и не будет в будущем сильно улучшаться. Причиной является его относительная завершенность. Основными выполняемыми им задачами являются лишь патрулирование, коммуникация и атака в крайнем случае для закрепления противника в нужной точке. Закрепление нужно для того, чтобы боты, описанные в пункте 3.1.1, пришли на помощь. Урон у данного персонажа меньше, чем у бота. Это связано с тем, что персонаж летает и меньше по размерам, что усложняет стрельбу по нему. Если оставить такой же урон, как и у бота, игрок будет испытывать значительные трудности при прохождении. Как дрон выглядит внешне, показано на рисунке 3.3.

Данный класс `Drone` наследуется также от класса `Character` и имеет те же компоненты, описанные в боте. Стоит лишь упомянуть измененные настройки, позволяющие персонажу летать. Для полета необходимо поставить соответствующую настройку в поле `CanFly` в компоненте `CharacterMovement`, убрать гравитацию у компонента `CapsuleComponent`, который является по сути корнем, к которому прикрепляются все остальные компоненты. Он, `CapsuleComponent`, так же отвечает за упрощенную коллизию. Также для баланса в игре изменена скорость по умолчанию. Конечно, существует разные скорости дрона. Дрон будет лететь медленнее при патрулировании, чем в случае нахождения игрока.

Для реалистичности был добавлен компонент `SpotLight`. Он используется для освещения пути и не является обязательным. При смерти персонажа свет будет отключаться, что символизирует отключение логики в нем.

Стрельба потребовала добавления точек начала стрельбы. Для точек был выбран компонент `Arrow`. Представляет собой обычную стрелку с направлением по оси X. Ее направление не имеет значения на данный момент. Используются только координаты компонентов в мире.

Помимо компонентов также возникла необходимость добавить некоторые переменные. Некоторые из них используются лишь дроном и не будут просматриваться вне персонажа другими классами.

Рисунок 3.3 Внешность дрона

Переменная `MaxHealth` – переменная вещественного типа, которая служит для установки начального уровня здоровья. Подобная переменная уже была использована в боте и описана в соответствующем пункте.

Переменная `Health` также была использована в боте. В дроне выполняет ту же функцию. Нужна для хранения информации о количестве здоровья дрона. Используется при обработке получения урона при решении уничтожать дрон или нет.

Переменная `FireRange` в общем случае должна быть привязана и использоваться применительно к тому или иному оружию. Но в дроне используется так как оружие условно встроено. Представляет собой переменную вещественного типа. Используется при выполнении некоторых задач, связанных с приближением к игроку и последующей атаке.

Переменная вещественного типа `Damage` также по-хорошему должна быть закреплена за оружием, переносимым персонажем, но той же причине была закреплена за дроном. Урон, стоит отметить, должен быть в разы ниже, чем у бота. Причины были указаны в начале данного пункта.

Основной задачей, которую выполняет дрон, является патрулирование. Стоит отметить, что патрулирование в отличие от бота происходит по четко заданным точкам. Отклонение от них при патрулировании исключено. Более подробно патрулирование будет описано после рассмотрения функций и переменных, используемых в реализации.

Переменная `PatrolPoints` – массив точек, условно представляющих собой точки для патрулирования. Именно по ним летает дрон, сканируя

местность. Их количество может быть неограниченно. Это задается самим разработчиком, а точнее дизайнером уровня, который должен прорабатывать баланс каждого уровня, его детали в окружении и прочее. Массив инициализируется в начале игры, а точнее в начале игры дрона. Более подробно будет описано ниже при рассмотрении событий и функций, используемых в классе `Drone`.

Переменная `PointsTag` помогает разграничивать точки патрулирования дронов и в принципе различные объекты на карте. Так, каждый объект, актер, может иметь неограниченное количество тэгов. Для получения всех точек в патрулировании, используется встроенная функция, в которую и посылается необходимый тэг. Это переменная используется при инициализации переменной `PatrolPoints` в начале игры.

Переменная `FireRate`, как следует из названия, отвечает за скорость стрельбы персонажа. Опять же должна быть закреплена за оружием. Для баланса стоит учитывать тот факт, что дрона сложно уничтожить, а значит не стоит добавлять не только большой урон, но и не стоит добавлять высокую скорость стрельбы.

Переменная `LastTimeFired` – вспомогательная переменная. Отвечает прежде всего за контроль стрельбы. В ней записывается, условно говоря время последней стрельбы. Изменяется и в начале игры и при стрельбе дрона. Данная переменная позволяет немного уменьшить нагрузку при обработке действия персонажа.

Переменная `Deactivated` служит для проверки уничтожен дрон или нет. Так же дрон может быть деактивирован игроком до прихода в комнату, поэтому может выставляться при создании персонажа на карте. Если он деактивирован, значит в скором времени саморазрушится.

Переменная `NavMeshTag` используется для упрощения работы с навигацией и будет описан дальше. Стоит лишь упомянуть, что принцип работы схож с переменной `PointsTag`.

Переменная `TargetPointIndex` является вспомогательной переменной и отвечает за то, какая точка в патрулировании будет следующей для посещения. Представляет собой обыкновенную переменную целочисленного типа. Ее размерность – единственное, что ограничивает количество точек патрулирования.

Переменная `CurrentPath` – массив точек, путь дрона от точки к точке, который он должен преодолеть. Нахождение пути стоит более подробно рассмотреть позже, при описании реализованных задач в дереве поведений.

Переменная `Mesh3D` – переменная, содержащая ссылку на объект на уровне, позволяющий ориентироваться в пространстве дрону. Ее также стоит более подробно рассмотреть при описании используемого плагина. Описание плагина будет приведено ниже, в этом же пункте после описания всех переменных класса `Drone`.

NavMesh3D не является переменной у дрона или его компонентом. Это класс, созданный сторонним разработчиком для ориентирования в пространстве. Он позволяет строить пути персонажам, особенно упрощает работу при создании летающих персонажей. Как уже было сказано в боте, в Unreal Engine 4 существует встроенный функционал, позволяющий строить пути для искусственного интеллекта. Но он подходит лишь для наземных персонажей, тех кто не будет летать или ездить.

Для летающих персонажей в разных играх создаются особые условия в игре. Можно ограничить персонажа тем, чтобы он сканировал сам окружение, путем испускания лучей в нужные направления, но это не позволит искать пути до объектов за препятствиями. Для этих целей, для облета препятствий, обычно пишется особый функционал.

Данный плагин добавляет возможность строить пути летающим и не только персонажам, путем создания условной трехмерной сетки. В сетке, разумеется, есть кубы, которые реагируют на пересечение с другими объектами. В случае пересечения, куб помечается. При запросе на построение пути такие кубы фильтруются. Они, так как пересекаются с чем-либо, не дадут перелететь персонажу. В случае пересечения с точкой назначения, куб помечается как конечная точка, а точка нахождения персонажа помечается как начальная. Далее задействуется алгоритм поиска пути.

При нахождении пути, выдается набор точек, центров кубов, по которым можно пролететь к конечной точке. Подробнее про алгоритм поиска пути в пространстве в записке писаться не будет по причине того, что данная тема довольно сложная для описания и соответственно, описание выйдет объемным, что не позволительно.

У дрона реализованы такие функции, как Death, FireFromPoint и вспомогательная функция GetWorldLocationFromArray.

Функция GetWorldLocationFromArray нужна лишь для инициализации точек патрулирования. Так как нельзя сразу взять координаты нужных объектов-точек, используется простой цикл для заполнения массива. Данная функция закреплена на событии начала игры, которое вызовется лишь раз. Помимо точек патрулирования стоит упомянуть инициализацию сетки навигации, уровня здоровья, переменной LastTimeFired и настройки режима движения персонажа.

Функция смерти также как и у бота закреплена на событии получения урона и вызывается только при отсутствии здоровья персонажа. Также выключается коллизия компонентов, включается обработка физики и, как было сказано выше при описании компонентов и переменных, выключается свет компонента условной лампочки дрона.

Функция FireFromPoint закреплена на созданном событии Fire и отвечает за стрельбу из начальных точек, которые представлены уже описанными выше компонентами. Событие Fire лишь запускает

последовательность выстрела из левой и правой точек стрельбы с некоторой задержкой между ними. Задержка равна половине значения, установленного в `FireRate`.

Также, как и у бота, логика персонажа разделена на три части. Выше был описан класс, в котором реализуются основные действия, такие как стрельба, смерть и прочее, а также переменные, необходимые для них и для дерева поведений (задач в нем).

Второй немаловажной частью дрона является его контроллер. Также как и у бота, в контроллере добавлен компонента `AIPerception`, который позволяет добавлять органы чувств дрону. В нем добавлено три основных органа чувств: зрение, слух и орган для получения информации о уроне. Работа органов чувств принципиально ничем не отличается от работы органов чувств в боте, описанном выше.

В контроллере не было необходимости создавать переменные, помимо локальных. Локальные же нужны были лишь для упрощения написания кода, а именно для уменьшения линий связи между узлами. Это значительно упрощает разбор кода и его последующую оптимизацию, как уже было указано в обзоре литературы.

Основная часть дрона именно зрение, остальные части алгоритма, связанные с другими органами чувств рассмотрены не будут по причине схожести с частями в боте, который уже был рассмотрен. Для зрения также не было необходимости усложнять логику, поэтому вся функция на данный момент не является сложной и закреплена в одной функции – `SightSense`.

Так как до вхождения в эту функцию отфильтрована информация об источнике, что означает вхождение лишь при виде противника, то проверяется лишь информация, вышел из вида или вошел в поле зрения дрона заданный противник. В том и другом случае выставляются значения для соответствующих переменных в хранилище данных бота для дерева поведения, которое будет рассмотрено позже в этом разделе.

Как уже было сказано, дрон в реализации проще. Это означает, что дерево поведения также меньше, что логично, так как задач, которые может выполнять дрон меньше, чем у бота. У прототипа дрона на данный момент присутствует две основные задачи-ветви – патрулирование, атака, преследование.

Патрулирование похоже на задачу бота, но сильно отличается реализация. Для перемещения дрона необходимо строить пути не в заранее просчитанной плоскости, а в трехмерном пространстве. Это значительно усложнило разработку, так как не было известно, как решать такого рода проблемы. Было решено использовать плагин стороннего разработчика, позволяющего искать пути в пространстве. Основная функция, которую добавляет этот плагин – `FindPath`. Она как раз и ищет путь. Возвращает массив координат, представляющих собой центры кубов.

Для работы плагина и соответственно функции FindPath необходимо, как и в случае с ботом, разместить на уровне навигационную сетку, которую также добавляет плагин и растянуть ее, путем выставления настроек высоты, ширины и длины. Также можно указать размер кубов. Размеры выставляются разработчиком и связаны только с тремя условиями – размер дрона или других летающих персонажей, точность и дискретность сетки навигации, а также скорости обработки и составления пути.

Стоит упомянуть, что весь просчет путей происходит при запуске игры и не будет сильно влиять на производительность игры после настройки. Рассмотрение ветвей в дереве поведения также следует начать после описания используемых задачи сервисов.

Сервис FocusOnPlayer создан с целью фокусироваться на игроке в начале выполнения соответствующей ветви и сбрасывать фокус в конце выполнения. Также для уменьшения количества сервисов было встроено увеличение и уменьшение скорости полета дрона. В начале выполнения оно становится выше, в конце сбрасывается в значение по умолчанию.

Сервис DroneShooting заставляет дрона стрелять по цели во время выполнения задачи. Условием выполнения является пролет дрона на определенном расстоянии от игрока.

AbleToFly сервис нужен для проверки и возможного создания пути для дрона в конце выполнения ветви с задачами. В конце в зависимости от того, проложен ли путь, выставляется переменная PatrolSkip в хранилище.

Ветвь патрулирования несколько отличается от аналогичной ветви в боте. Так как использование MoveTo и EQS затруднено при реализации бота, то было принято решение написать аналогичные задачи – FlyTo и FindNextPoint для пролета к точкам. FindNextPoint по сути простраивает путь от местоположения бота к заданной точке с использованием функции FindPath. FlyTo лишь запускает поэтапное выполнение функции MoveToActorOrLocation, которая встроена в движок.

Помимо данных двух задач в ветви патрулирования, существует между ними еще одна задача, которая не является обязательной, но позволяет минимизировать размер массива координат и также сделать движение дрона немного плавнее – OptimisePath. Она оптимизирует путь, записанный в массив координат путем удаления лишних точек. Реализовано довольно просто – идя с конца массива, просматривается, имеет ли последняя точка, которая проверяется, прямую связь. Если имеет, все точки между ними удаляются.

Последняя задача, созданная для дерева поведения дрона – FindReachablePointNearPlayer. Она ищет точку, к которой дрон может пролететь и которая находится рядом с игроком. При этом точка

выбирается случайно и с нее должна быть возможность стрелять, атаковать противника.

Как уже было сказано, у дрона всего три выполняемые ветви в дереве поведения. Рассмотрение ветвей также, как и при рассмотрении ветвей в боте, стоит сделать по порядку их расположения в дереве.

Первая ветвь дает боту возможность пролететь в сторону игрока или месту, где он был недавно. Тут используется сервис `AbleToFly`, закрепленный на композите Sequence. Также к нему прикреплен декоратор, проверяющий необходимость выполнения этой ветви.

На композит закреплено три задачи – поиск точки рядом с игроком, оптимизация пути и собственно задача пролета к заданной точке. По своей сути, выполняется данная ветвь по триггеру в blackboard – `PatrolSkip`.

Ветвь патрулирования схожа по реализации с ботом. У бота имелась задача, запускающая опрос среды и проход к точке, у дрона ситуация та же, за исключением того, что были реализованы собственные задачи и добавлена задача оптимизации пути пролета. Данная ветвь также выполняется в случае, если дрон не видит игрока или иного противника.

Последняя третья ветвь выполняется при атаке противника дроном. На композите Sequence закреплено три задачи – поиск точки рядом с игроком, оптимизация пути и пролет к точке. Так же закреплено на композите два сервиса – `FocusOnPlayer` и `DroneShooting`. При этом стоит помнить, что фокусировка сбрасывается только при потере противника из виду.

Дрон более прост в реализации, чем бот, и имеет меньший потенциал для развития. Но стоит отметить, что он позволяет увеличить вовлеченность игрока в процесс прохождения игры.

Также при реализации дрона было изучено много материала, который был полезен в разработке других объектов. Основными задачами, как уже было описано выше, является патрулирование и атака противника. Третья задача лишь ответвление задачи атаки, но она должна была быть отделена из-за невозможности совмещения в одну и большого отличия по цели выполнения.

Если дерево поведения бота было выделено в отдельный плакат, то дерево поведения дрона возможно было вставить в саму записку и показано на рисунке 3.4. Задача по оптимизации, как и было сказано выше, была выделена в отдельную задачу поскольку ее использование не всегда оправдано.

При последующей разработке возможно появится необходимость в отсутствии оптимизации данной функцией пути полета дрона. Ветви-задачи, описанные выше, расположены на дереве поведения в том же порядке, как и при описании в записке. Порядок исполнения узлов, как и в дереве поведения бота также указан на данном рисунке для полного понимания логики поведения дрона.

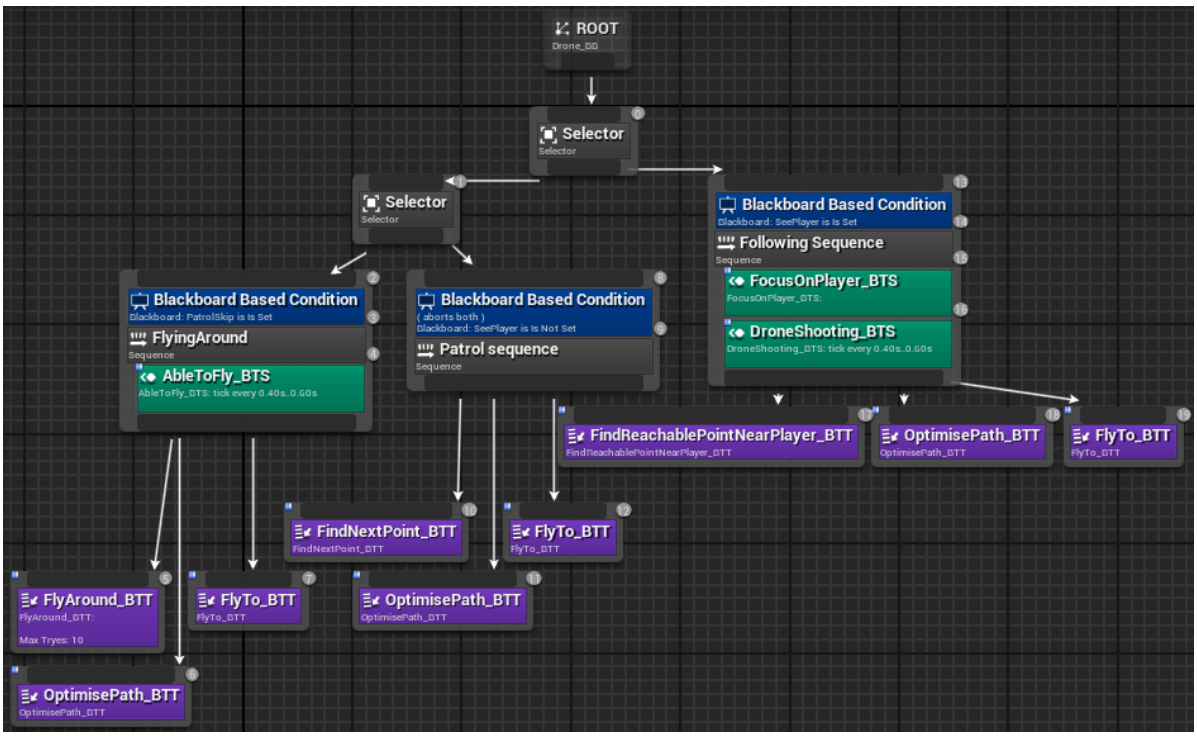

Рисунок 3.4 – Дерево поведения дрона

3.1.3 Автоматическая турель

Автоматическая турель по своей сути не является персонажем или чем-то наделенным искусственным интеллектом в привычном понимании. У нее нет задач для патрулирования, у нее нет коммуникации с другими объектами, и она не способна сканировать окружение. Также у турели нет органов чувств в привычном понимании. Турель может лишь брать все объекты класса actor, которые выбраны как предполагаемые цели. Дальнейший выбор основывается лишь на дистанции до цели. Как если бы турель была настроена на защиту территории ото всех, независимо от иных параметров.

Основной задачей турели является атака любого объекта, способного перемещаться. Это может быть как игрок, так и дрон или бот. Его внешность показана на рисунке 3.5. Внешность в данном случае не важна. Она необходима лишь для визуального отделения и выделения на карте для удобства тестирования и игры в целом.

Для такого типа объекта не возникло необходимости писать дерево поведений и добавлять контроллер. Это бы было бессмысленно, потому как задача, которую турель исполняет всего одна. Также она, турель, была первым объектом, разрабатываемым в дипломном проекте. Ее разработка была необходима в основном для закрепления изученного материала по Unreal Engine 4.

Рисунок 3.5 – Внешность автоматической турели

Сперва стоит описать функции, которые будут прикреплены к событиям для выполнения. Тут же будет кратко описана работа с чертежами в Unreal Engine 4 для лучшего понимания.

Первой рассматриваемой функцией является `FindTarget`. Для лучшего понимания, она будет изображена на рисунке 3.6. Как упоминалось при описании турели в функциональном проектировании, у турели отсутствуют встроенные органы чувств. Он может ли получать информацию о сущностях типа actor, находящихся в радиусе. Получение данной информации происходит после вызовы функции `SphereOverlapActors`. Задается центр сферы, ее радиус, объекты, которые необходимо искать и опционально можно настроить список сущностей, которые стоит исключать. Последний параметр не использовался. Далее происходит поиск сущностей, которые ближе всего расположены к турели и находятся также в зоне видимости, чтобы в последствии атаковать.

На чертежах белыми линиями показан порядок выполнения функций. Узел с названием `FindTarget` является начальным, что и видно на рисунке. Это своего рода начало выполнения функции. Прочие цветные линии — это переменные связи переменных, которые могут быть как локальными, так и глобальными. Под локальной переменной в данном случае стоит подразумевается то, что они объявлены разработчиком. Локальные же наоборот нельзя записать в память, они исчезают сразу после того, как были использованы функциями или иными узлами. К сожалению, для демонстрации кода в записке, было необходимо уменьшить расстояние между узлами, что сильно ухудшило читаемость кода.

Тут стоит кратко рассказать, что в основном, для создания игровых персонажей используются классы типа `Pawn` или `Character`, который

наследуется от класса `Pawn`. По этой причине вход `ObjectTypes` в функции `SphereOverlapActors` использует как маску лишь классы типа `Pawn`. По этой же причине турель атакует всех – ботов, дронов, игрока.

Далее, после сбора информации обо всех сущностях в радиусе, начинается фильтрация по двум признакам. Первый, как уже было сказано – возможность турели стрелять по цели. Второй – радиус атаки. У турели есть как радиус видимости, при котором он начнет сводить башню к цели, так и радиус атаки, при нахождении персонажей в котором турель начнет атаковать цель. Опять же, для демонстрации пришлось уменьшать расстояние между узлами.

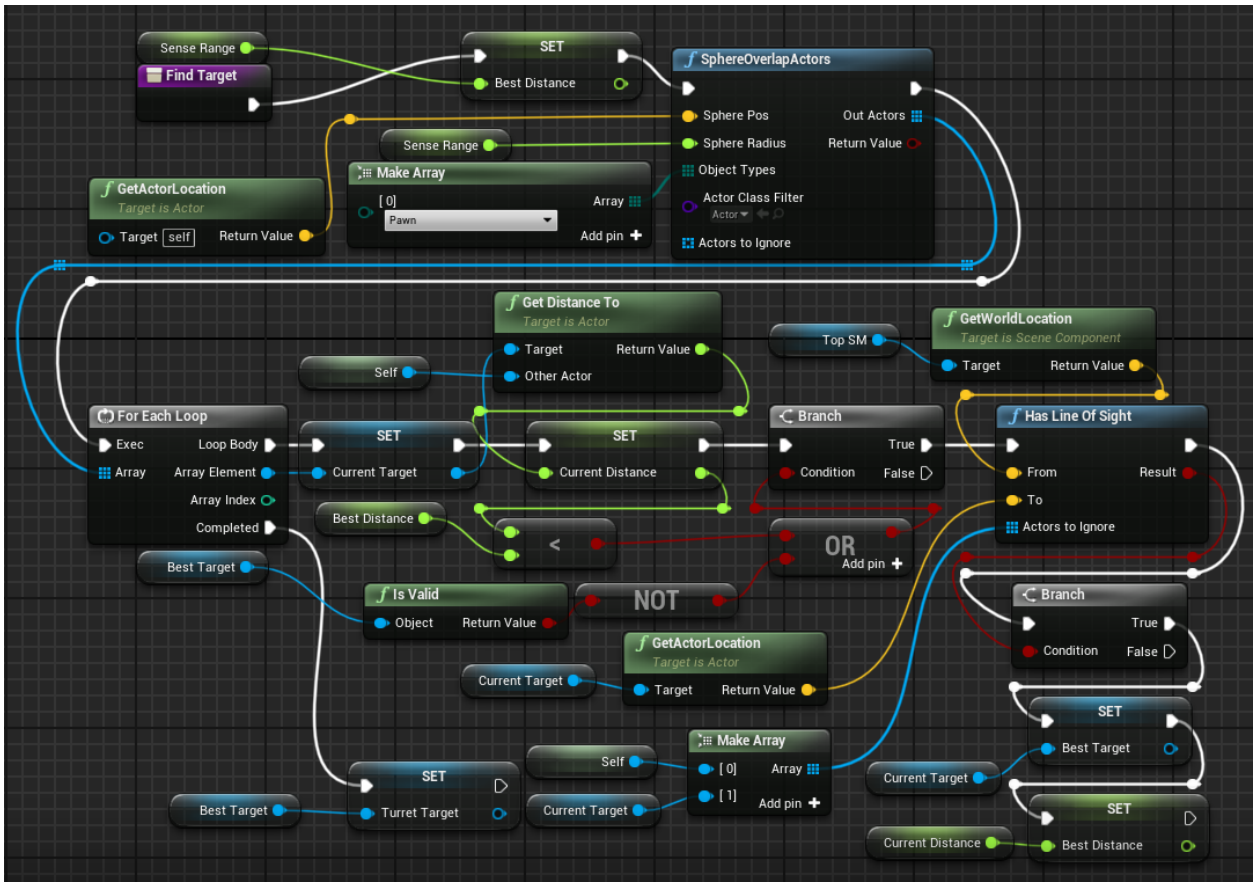

Рисунок 3.6 – Функция `FindTarget`

Следующая функция `RotateTurret` была необходима для вращения главной части турели – башни. Так как для данной сущности не предусмотрено никаких анимаций, что является также и избыточным, то возникла необходимость контролировать повороты башни вручную. На рисунке 3.7 показана реализация данной функции.

Логика довольно проста – башня поворачивается каждый тик в направлении текущей цели. Скорость поворота и ее плавность контролируется переменными `RotateSpeed` и `DeltaSeconds`. `DeltaSeconds` контролируется параметром класса, уже описанным в

функциональном проектировании – `TickInterval`. Реализация основана на одной функции – `RinterpToConstant`, которая плавно изменяет значение одной переменной со временем. Значение, которое должно плавно изменяться, берется как результат работы функции `FindLookAtRotation`. Угол поворота высчитывается между поворотом башни и целью.

Необходимо также учесть, что турель может вращаться только в определенных границах. Границы выставляются соответственно логике – турель не должна застревать в своих же текстурах, в своей модели. Для этого, при помощи функции `ClampAngle`**,** был ограничен поворот башни по оси Y. Значения ограничений приведены на рисунке 3.7 в параметрах функции.

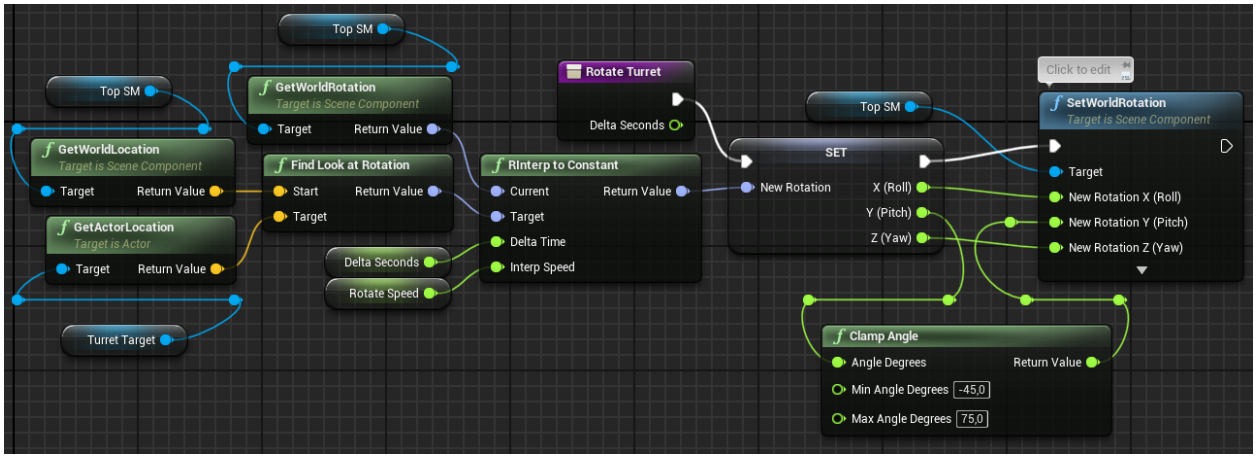

Рисунок 3.7 – Функция `RotateTurret`

Функция `FireFrom` отвечает за стрельбу турели. Существует множество способов реализации стрельбы. Основной используемый способ – создание отдельного класса снаряда. В него заносится начальная скорость, наносимый урон и иные эффекты, например анимация попадания или создание системы частиц в месте попадания.

В данном проекте было принято решение сделать стрельбу турели при помощи моментального получения урона. Это достигается при использовании функции `LineTrace` и ей подобных. Данная функция в общем случае пускает луч или несколько лучей. Далее проверяется пересечения лучей с объектами на карте по маске, которая задается опять же в параметрах функции. Тут также привязана и работа с лазером. Лазер был реализован, но не является важной частью проекта, он служит лишь для визуального выделения стрельбы.

Как и во всех функциях, существует начальный узел, он выделяется фиолетовым цветом и соответствующим названием. Данная функция вызывается после наведения на игрока или иную цель, а значит можно сразу начинать стрельбу, без последующей фильтрации полученной информации – не важно, попала турель по цели или нет, так как нанесение урона тем же стенам должно предусматривать обработку ей же.

После генерации луча, берется информация об итоге работы функции `LineTrace` и в зависимости от исхода, отображается лазер и генерируется звук стрельбы. Стоит отметить, что звук генерируется не только обычный, который может слышать игрок, но и технический, при помощи `MakeNoise` для возможного информирования других персонажей.

Так как лазер должен заканчиваться только при попадании в цель или, в случае если не было попадания, заканчиваться после определенной длины, то в конце присутствует небольшое ветвление кода. Ветвление происходит потому, что рисоваться лазер должен либо до какой-то определенной точки, выдаваемой в `TraceEnd`, либо до цели, координаты попадания в этом случае будут указаны в `HitLocation`. Как и в случае других функций, код будет представлен на рисунке 3.8.

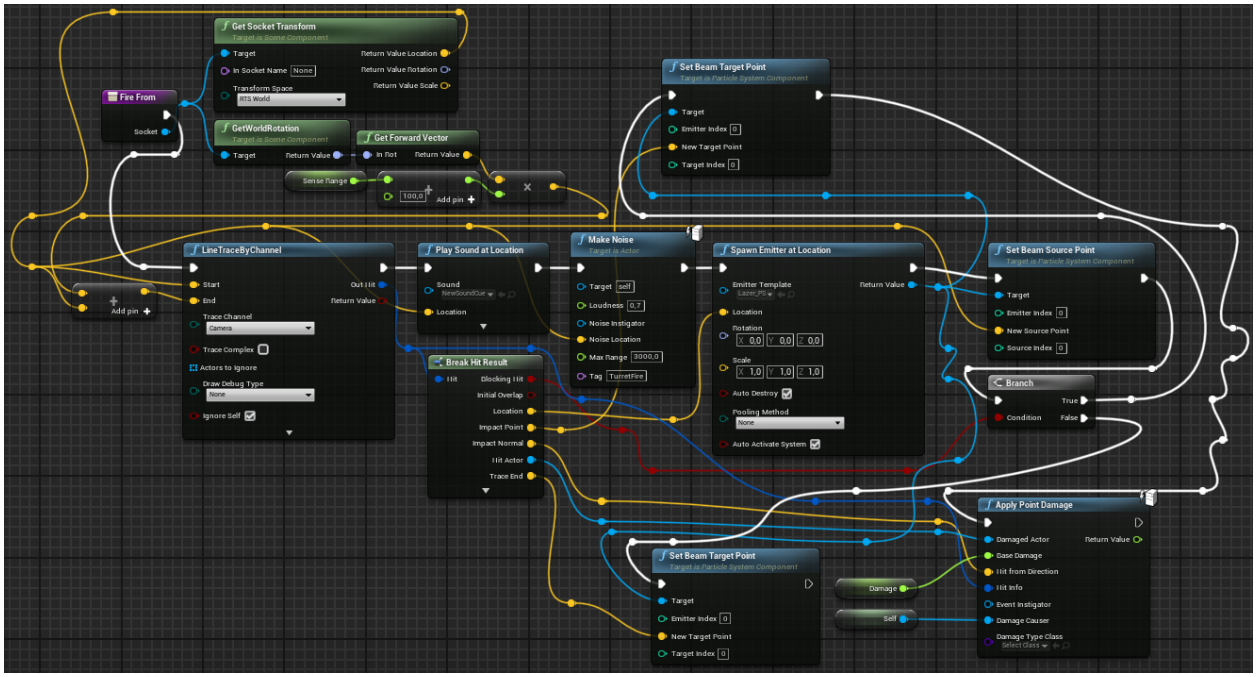

Рисунок 3.8 – Функция `FireFrom`

Последняя функция – `FireTurret`. Она отвечает за сам запуск стрельбы. Так как стрельба может производиться не чаще чем указано `FireRate`, то тут происходит проверка по времени. Также проходит проверка на то, наведено ли орудие на цель. Берется вращение башни и угол поворота, оставшийся до полного наведения на цель. В случае совпадения углов, стрельба разрешается. Код представлен на рисунке 3.9.

Отдельно стоит упомянуть лазер, так как он используется и в оружии, и в дроне с турелью. Внешне он выглядит, как и обычный лазер. Он не имеет коллизии и является системой частиц. Для создания необходимо указать глобальные координаты для размещения и две точки – начала лазера и его

конца. Его время жизни контролируется в настройках самой системы частиц. Значение времени жизни было выставлено минимальное.

Система частиц уничтожается по окончанию отображения лазера, что позволяет в некотором роде экономить память и прочие ресурсы компьютера. Цвет лазера также является изменяемым и может меняться в зависимости от нужд разработчика. В будущем планируется добавление кооперативного режима и значительное расширение возможностей игрока. В том числе кастомизации. Так же цвет можно использовать для определения того, кем был произведен выстрел, какой персонаж или команда.

Лазер также можно использовать и в окружении, но на данный момент это не представлено в проекте из-за отсутствия надобности в построении более сложного окружения.

При разработке лазера также был использован отдельный материал, но он является довольно легковесным, так как по своей сути является обыкновенным прямоугольником, который и растягивается по всей длине лазера.

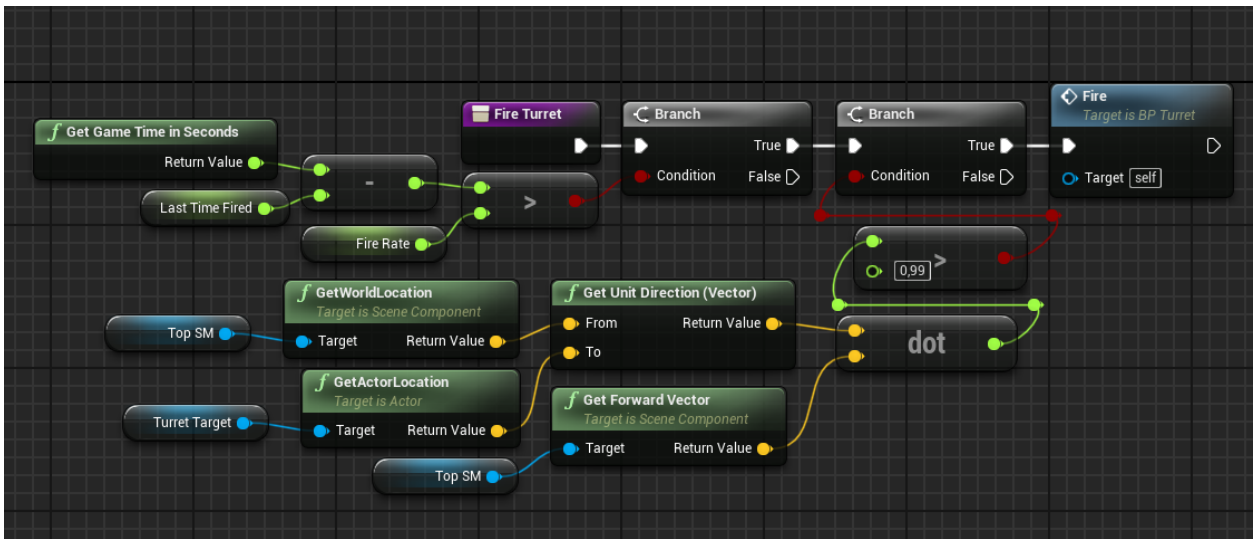

Рисунок 3.9 – Функция `FireTurret`

Далее стоит описать события, которые вызывают данные функции и иные функции, которые уже устроены в движок. Самым важным ивентом является `Tick`. За данным событием закреплена вся основная логика и данное событие вызывается каждый выделенный тик для турели. Чертеж данного события представлен на рисунке 3.10.

Данное событие встроено в каждый класс, наследуемый от actor. В каждом наследуемом классе есть возможность переопределять данное событие. При построении более сложной системы, как например при создании оружия для использования персонажами, переопределение данных методов необходимо в первую очередь чтобы менять тип стрельбы.

Разумеется, переопределять можно не только события, но и функции. Событий, которые можно использовать, намного больше. И все ивенты можно переопределять. Также можно вызывать закрепленную логику класса, от которого наследуется нужный класс, до или после логики, реализованной разработчиком.

Первым действием, закрепленным за данным событием, является проверка, была ли разрушена турель или была ли она деактивирована иным способом. В случае, если она еще активна, вызывается функция поиска цели, проверяется, валидна ли данная переменная для цели и далее происходит вращение башни турели в сторону цели. Далее вызывается функция стрельбы, в которой сперва проверяется возможность стрельбы, а далее вызывается уже и сама стрельба, если она разрешена.

В случае, если турель была уничтожена или деактивирована извне, необходимо каким-либо образом показать игроку, что турель разрушена или не активна. На данный момент это достигается путем поворота башни к основанию. Часть функционала была взята из функции поворота башни `RotateTurret`.

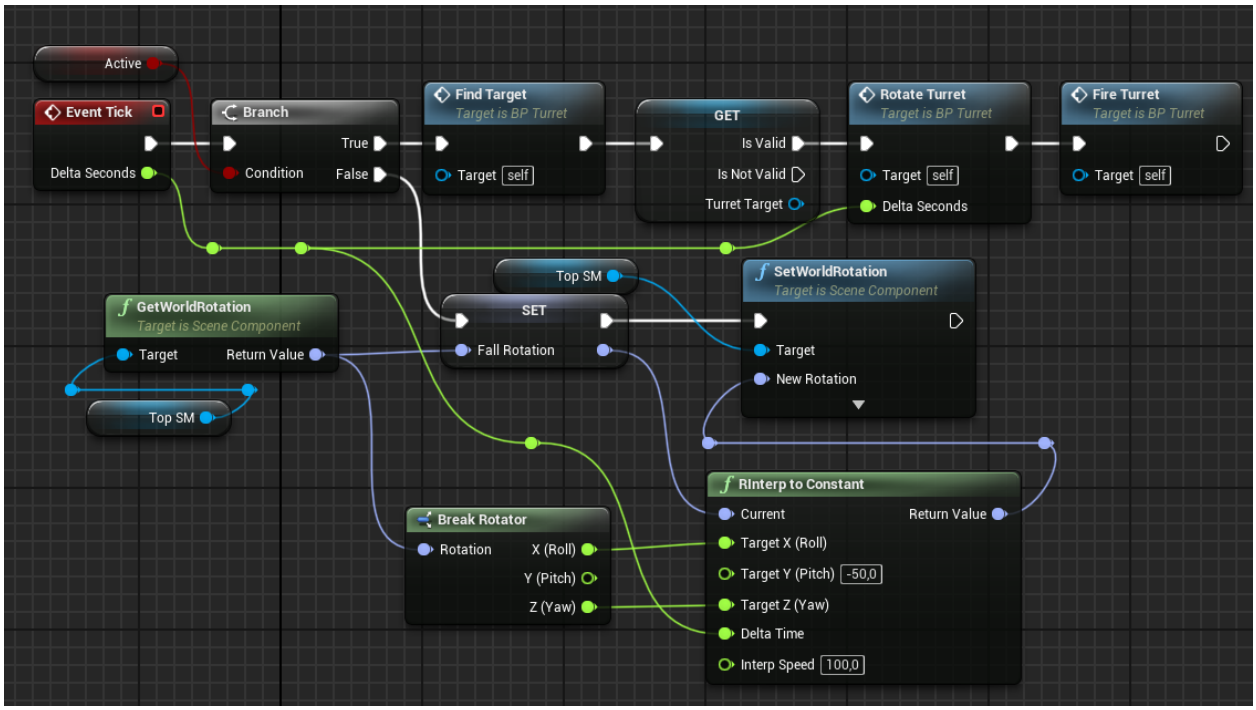

Рисунок 3.10 – Событие `Tick`

Самым небольшим событием является `BeginPlay`. Он также встроен в класс благодаря наследованию от класса Actor. Данное событие было крайне полезным при отладке турели, однако в конце вся настройка персонажа была перенесена. Данное событие использовалось в первую очередь для балансировки скорости стрельбы, при изучении функций по типу `LineTraceByChannel` и `SphereOverlapActors`. Осталась лишь

настройка начального уровня здоровья. На рисунке 3.11 показана логика, связанная с данным событием.

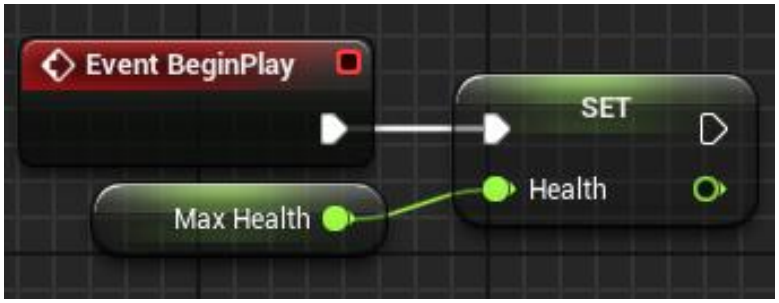

Рисунок 3.11 – Событие `BeginPlay`

Событие `AnyDamage` позволяет отлавливать получение урона турелью. Обработка получения урона опять же довольно проста с точки зрения реализации. При начале выполнения необходимо условно нанести урон. Так как событие хранит информацию о типе урона, кем был он нанесен и, разумеется, о количестве наносимого урона, то информация берется из него. Из текущего значения здоровья вычитается значение наносимого урона и сохраняется сразу же в ту же переменную. Далее при помощи простого сравнения проверяется, должна ли уничтожаться турель. В случае если в этом нет необходимости, ничего не происходит и выход False у функции ни с чем не соединяется. Так как анимации получения урона опять же не предусмотрена для турели. Уничтожение турели или точнее ее деактивация происходит в уже описанном событии `Tick`. На рисунке 3.12 показано использование данного события.

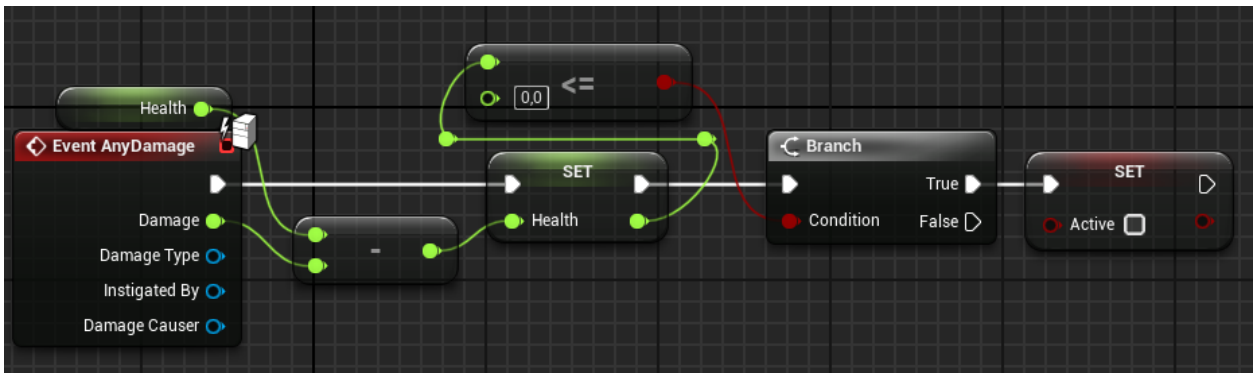

Рисунок 3.12 – Событие `AnyDamage`

Последнее рассматриваемое событие – `Fire`. Оно вызывается для стрельбы по цели. При стрельбе также необходимо запомнить время последнего выстрела, что и контролируется переменной `LastTimeFired`. Далее поочередно вызывается стрельба из двух орудий турели. Вызывается она с некоторой задержкой, равной половине частоты стрельбы. Данная

задержка необходима, чтобы корректно обрабатывать попадания и наносить урон.

После установки значения в переменную `LastTimeFired`, запускается узел последовательности, в котором присутствует два исполняемых выхода. Один из них направляется на вход enter цикла while, второй направляется на вход reset того же цикла. К выходу цикла была закреплена сама стрельба из двух орудий. При реализации было решено разделить стрельбу таким образом, чтобы минимизировать повторяющиеся части кода. Для этого и была создана функция `FireFrom`. Она принимает компонент, из которого необходимо производить стрельбу по цели. Использование события показано на рисунку 3.13.

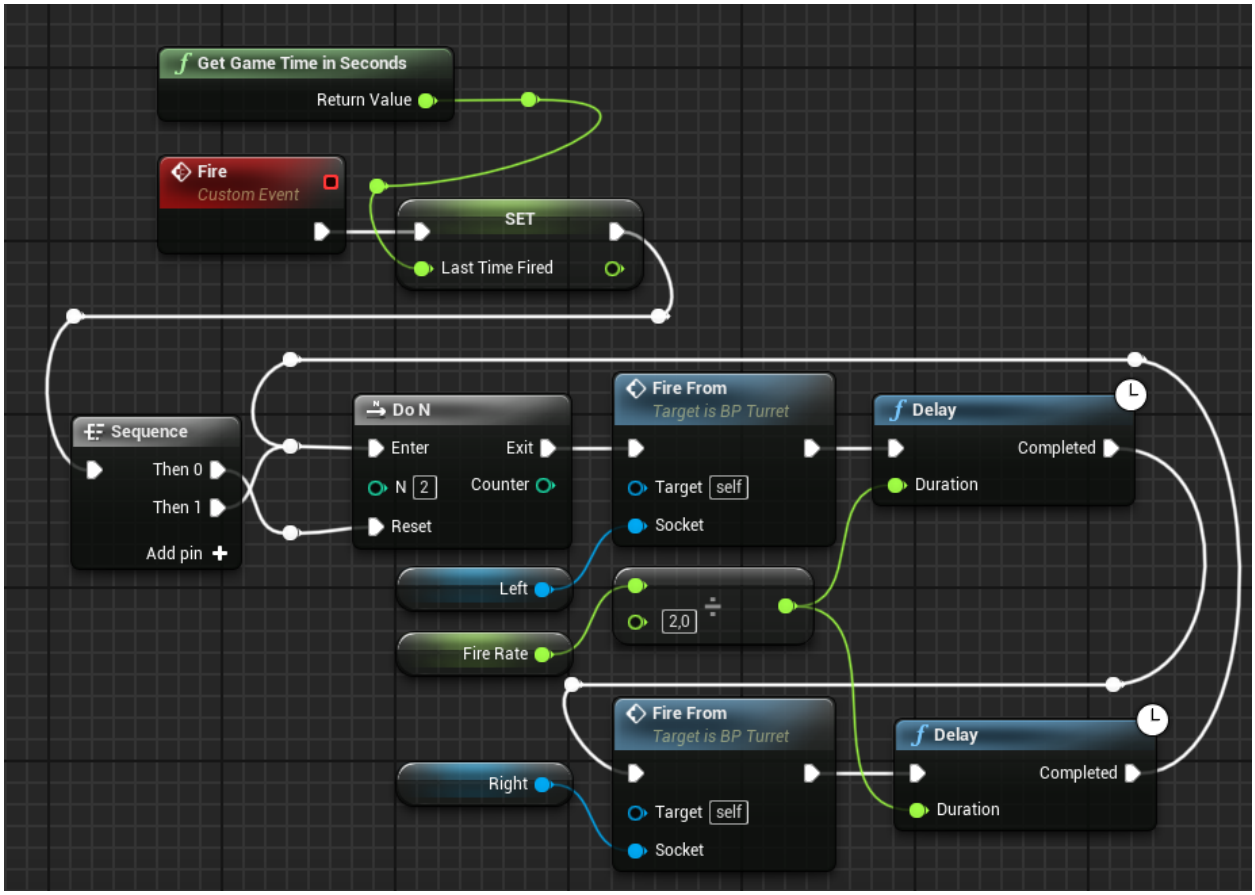

Рисунок 3.13 – Событие `Fire`

Разумеется, при желании, в будущем возможна модификация турели. А именно добавление или наоборот удаление стволов в ней. Данный персонаж хоть и является завершенным в плане логики, но может улучшаться в будущей доработке проекта.

Одной из доработок может быть балансировка возможностей стрельбы турели. В некоторых играх, особенно в стратегиях реального времени, часто используются турели, которые контролируют территорию не всю вокруг себя с радиусом обзора в триста шестьдесят градусов, а лишь часть, например

девяноста градусов. Такая балансировка, возможно, понадобится при доработке проекта.

Одно из возможных применений турели в будущей доработке проекта является добавление класса персонажей типа инженеров. Он находился на момент написания турели в проработке.

Одной из возможностей данного типа персонажа являлась бы установка временной переносной турели в области. Также у игрока появлялась бы возможность устанавливать турели и чинить их при необходимости. К сожалению, из-за малого времени, выделенного на разработку программного дипломного проекта, эту наработку было необходимо отложить.

3.2 Вспомогательные классы

`TestingPawn` – специализированный тип класса Pawn для проведения тестов системы запросов к среде, позволяющий проверить, что на самом деле делают разработанные запросы к среде. Точный состав запроса среды будет определять размер и форму создаваемого объекта, он всегда будет представлен в виде цветных сфер. Сферы на цветовой шкале от зеленого до красного указывают на некоторый уровень совпадения для различных тестов, которые выполнялись при опросе среды. Синие сферы указывают на ошибку или тест логического типа, вернувший false, например тест Trace. На рисунке 3.14 показан пример работы с `TestingPawn`. На нем показана работа опроса среды при патрулировании ботом местности, которая была описана при рассмотрении задач дерева поведения в боте. Как видим, у каждой сферы есть значение. Оно варьируется от нуля до единицы. Также может быть синим, что не показано – в данном тесте может быть только нулевое значение, точки не отбрасываются. Чем больше условный вес точки, тем больше вероятность того, что бот будет идти к данной точке. Стоит напомнить, что при патрулировании бот будет выбирать точку по направлению текущего движения (показано красной стрелкой, ось X) и в зависимости от расстояния до точки.

Данный класс не является частью игры, так как нужен только при тестировании запросов к среде. Внутренняя логика изменяется для каждого запроса в отдельности. Однако данные классы типа `TestingPawn` было решено оставить как на диаграмме классов, так и описание их в данной записке.

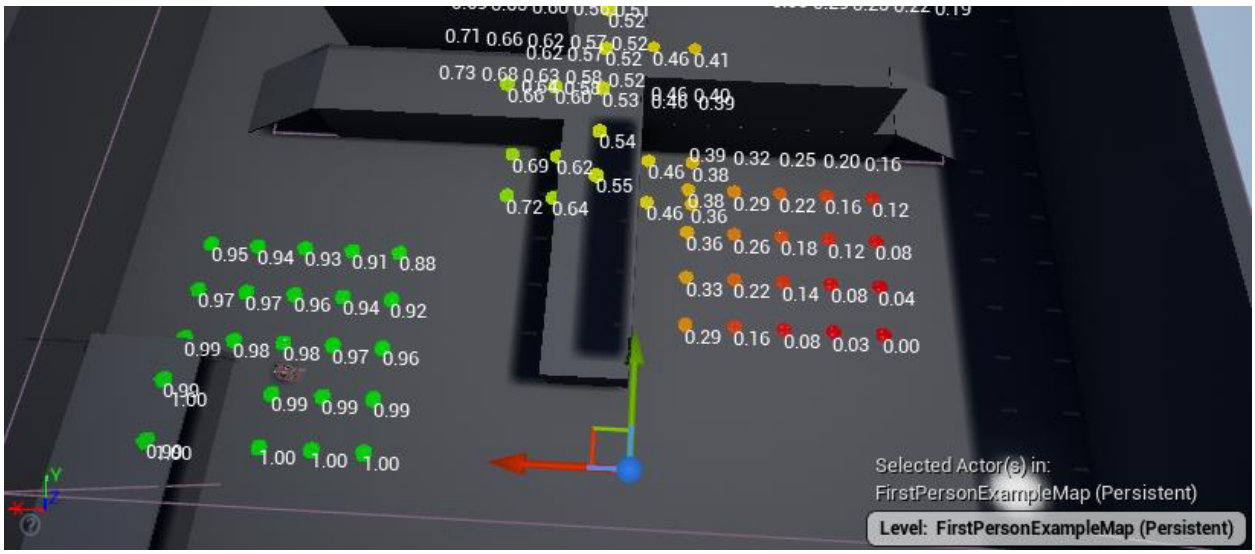

Рисунок 3.14 – Пример работы EQS

Данный класс был крайне полезен при разработке бота, так как показывал корректность работы запросов к среде. Также благодаря ему можно было сбалансировать персонажа. К сожалению, работа с этим классом при разработке дрона значительно усложнилась, из-за чего было принято решение не использовать запросы к среде как таковые у дрона и, соответственно `TestingPawn`. В автоматической турели из-за отсутствия работы с окружением необходимости в этом классе тоже не возникло.

`NavMesh` – крайне полезный класс, который помогает строить пути передвижения для персонажей, двигающихся по земле. Для летающих сущностей он не подходит, что уже было сказано раньше при описании дрона. На рисунке 3.15 показан пример его использования. Зеленой областью закрашена территория, по которой теоретически может передвигаться бот. Он, `NavMesh`, имеет множество настроек, рассмотрение всех параметров было опущено в данной записке по дипломному проектированию по причине их большого количества и незначительности при работе над данной игрой.

Из наиболее интересных параметров всё же стоит выделить расстояние до границ. Данный параметр контролирует то, какой необходимо взять отступ от стен или ям. На рисунке 3.15 область на кубе при изменении данного параметра показывает довольно хорошо этот отступ. Отступ было решено увеличить по сравнению со значением отступа по умолчанию во избежание нерешаемых проблем со стрельбой. Если персонаж стоит вплотную к стене может возникнуть ситуация, при которой он будет стрелять из стены.

Второй параметр, который может быть косвенно связан с одной из настроек самого персонажа, является – размер ступенек, на который может ступать бот. Так как навигационная сетка представляет собой условно плоскость, то она не может содержать каких-либо резких выпадов вверх или

вниз. Она остается под углом, но не в виде ступенек. Этот параметр и контролирует угол наклона данной сетки.

На рисунке 3.15, если вместо наклонной плиты поставить лестницу и настроить в ней слишком большую высоту ступенек, навигационная сетка не позволит строить пути через лестницу. Произойдет обрыв сетки. Косвенно связан с настройкой персонажа потому, что у бота, как и у игрока, есть схожий по назначению параметр в классе `Character`. Он также позволяет игроку или боту настраивать высоту ступеней, по которым они смогут ходить. Стоит помнить об этом при настройки навигационной сетки и персонажей.

Если возникают ситуации при которых нет связи с некоторыми поверхностями, возможно использование `NavLinkProxy`. Он помогает соединять некоторые области, которые не соединены сами по себе изначально. Например, на том же рисунке 3.15 показано, что бот может находиться на кубе. Но в случае необходимости можно настроить `NavLinkProxy` и событие в боте. Тогда можно, например, дать боту возможность запрыгнуть на данный куб. В данном проекте это не было использовано из-за недостаточного времени, выделенного на разработку. `NavLinkProxy` был рассмотрен при описании бота.

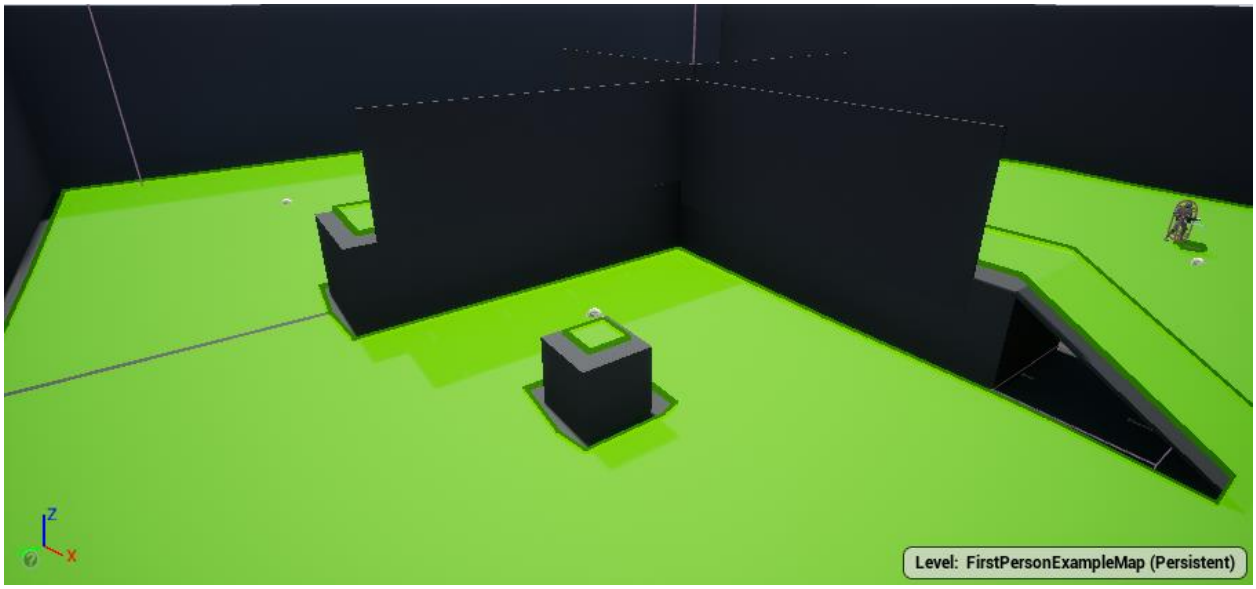

Рисунок 3.15 – NavMesh для бота

`NavMesh3D` – своего рода замена `NavMesh`, которая может использоваться для летающих персонажей. Он также позволяет строить пути в пространстве. Но если его аналог не позволяет строить пути в воздухе, то данный класс, добавленный как плагин, позволяет это сделать. Он был использован при реализации дрона. Так как при патрулировании и иных задачах для поднятия уровня интеллекта персонажа потребовалось использовать более умное построение путей, чем с использованием

LineTrace. К тому же, без его помощи нельзя строить пути, если дрон отвлекся на игрока и улетел с пути патрулирования. В данном случае дрон должен будет строить путь до ближайшей точки патрулирования, заданной разработчиком.

Если в случае с NavMesh была плоскость, то в данном классе, как уже было сказано при описании дрона, используется множество кубов. На рисунке 3.16 показано, как он используется и отображается на уровне. Как уже было сказано, путь строиться по точкам центрам кубов. Получается, дрон движется как бы по кубам, их центрам. На рисунке 3.17 показан пример построения пути. Сетка была выключена, так как скриншот делался при тестировании, запущенной игре. Сетка может быть включена при игре, если есть необходимость провести тестирование. Также ее можно не отключать, если, например, игра представляет собой симулятор полетов в космосе. Там сетка может представлять собой некоторого рода сегментацию космоса и помогать рассчитывать ходы. Также сетку можно настроить, а именно настроить толщину линий, цвет. Иные параметры были упомянуты в описании дрона.

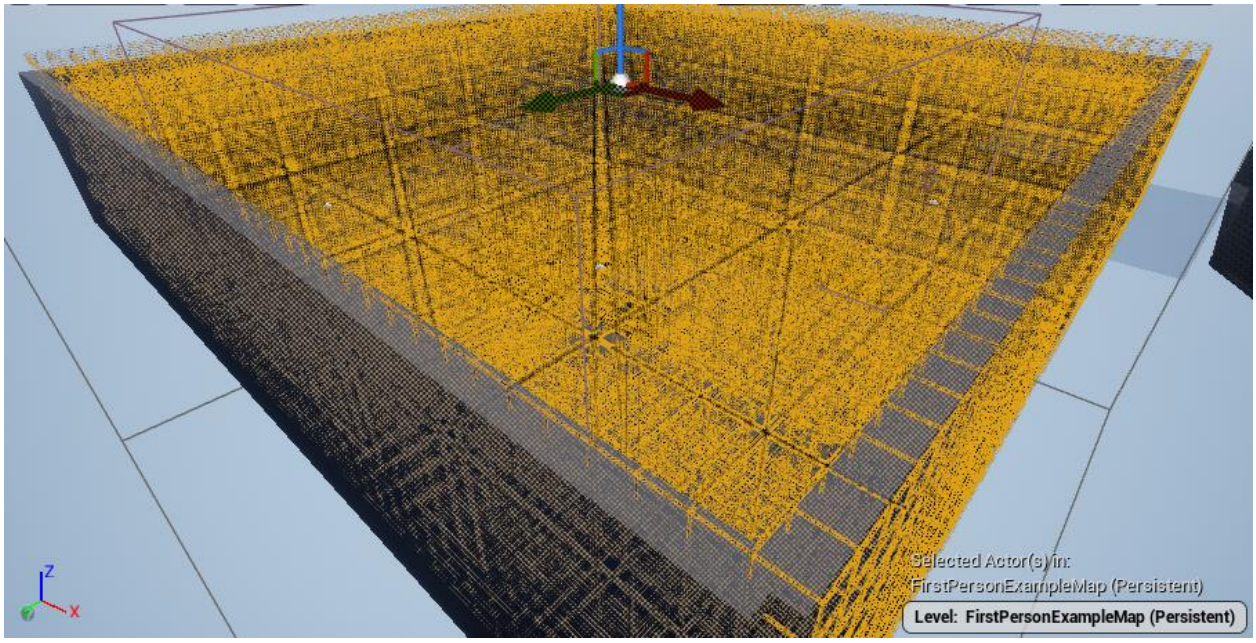

Рисунок 3.16 – Навигационная сета для дрона

HasLineOfSight – небольшая вспомогательная функция, позволяющая узнавать без дополнительной настройки, имеют ли две точки препятствия между собой. Необходимо лишь указать две точки.

GetPlayerWorldLocation также является небольшой вспомогательной функцией. Она в свою очередь помогает в получении координат игрока. На данный момент не имеет входных пинов, так как в игре нет мультиплеера. На выходе выдает глобальные координаты игрока.

Рисунок 3.17 – Построение путей у дрона

`EQS System` – система запросов к среде. Уже была описана при рассмотрении `TestingPawn` и дрона. Подробное описание каждого запроса, написанного в данном дипломном проекте, приводится не будет, так как они не являются чем-то сложным в реализации. Будет приведено только краткое описание работы с системой запросов в целом.

`EQS System` необходима для поиска объектов, сканирования окружения в целом. При этом не обязательно центром сканирования должен быть тот, кому информация будет предоставляться. Центром может быть любой объект на сцене. Например, для бота в система патрулирования для центров сканирования были использованы обычные актеры с измененным названием – `PointOfInterest`. Никакой логики в данном классе нет. Их создание было необходимо лишь для отделения их от другие актеров. На рисунке 3.18 показан типичный пример создания системы запроса к среде.

На рисунке 3.18 показан запрос к среде для патрулирования ботом территории. На нем видно, что граф похож на дерево поведений – также есть корень `root`, из которого идет стрелка в другие узлы. На данном примере и было решено показать работу системы запросов.

Справа в панели Details есть множество настроек. Первая настройка, которую было решено оставить по умолчанию, – `GridHalfSize`. Она как понятно по названию, контролирует площадь, на которой будут располагаться сферы, уже показанные выше. На том же рисунке и показаны точки интереса в патрулировании бота. Вторая настрокай – `SpaceBetween`, отвечает за разделение сфер, на какой удаленности друг от друга они будут располагаться. Третьим параметром задается кто будет центром этой площади.

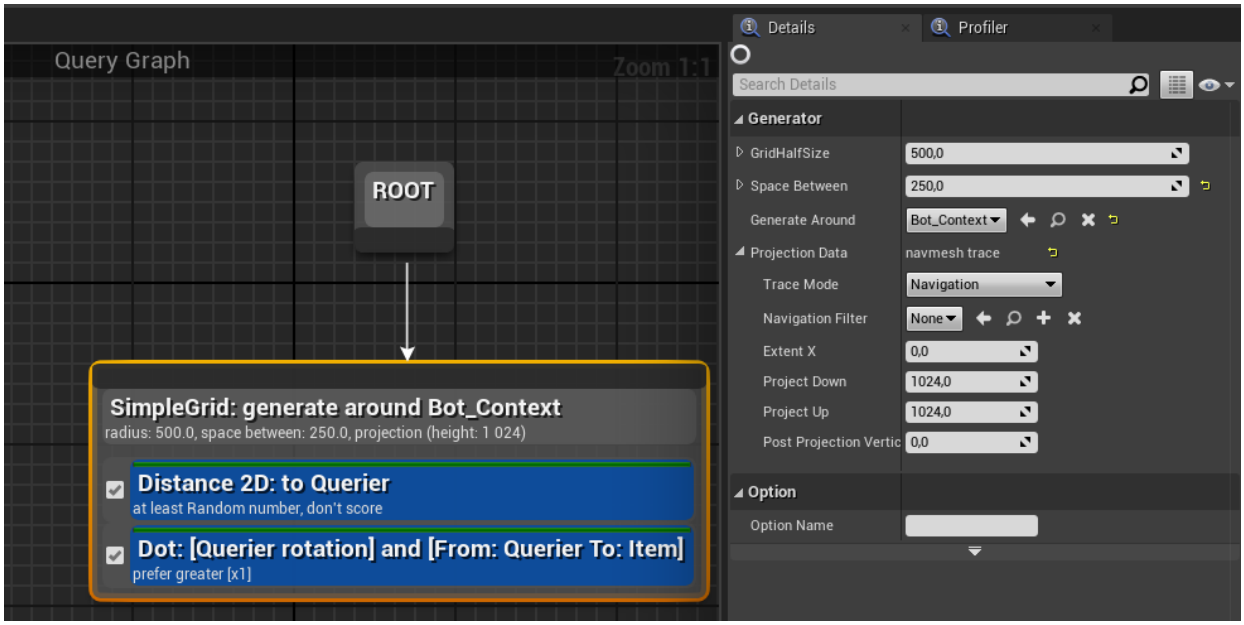

Рисунок 3.18 – Запрос к среде

На рисунке 3.18 показан запрос к среде для патрулирования ботом территории. На нем видно, что граф похож на дерево поведений – также есть корень `root`, из которого идет стрелка в другие узлы. На данном примере и было решено показать работу системы запросов.

Справа в панели Details есть множество настроек. Первая настройка, которую было решено оставить по умолчанию, – `GridHalfSize`. Она как понятно по названию, контролирует площадь, на которой будут располагаться сферы, уже показанные выше. На том же рисунке и показаны точки интереса в патрулировании бота. Вторая настрокай – `SpaceBetween`, отвечает за разделение сфер, на какой удаленности друг от друга они будут располагаться. Третьим параметром задается кто будет центром этой площади.

Тут стоит упомянуть, что сетка, создаваемая системой запросов, может быть не только квадратом. Сетка может быть конусовидной, торообразной, квадратной или круглой. В патрулировании была выбрана квадратная форма сетки.

Помимо генерации сфер с площадью разной формы, необходимо также задавать некоторые тесты. Тест представляет собой условную функцию фильтрации сфер. Тест может как сразу отсекать ненужные сферы, что означает, что целью теста является только фильтрация. Также тест может сохранять условный весь сферы. Как было сказано выше, сферы имеют цвет – синий и от красного к зеленому. Синий означает, что не был пройден тест или был сбой, а цвета от красного к зеленому показывают, насколько сфера, а точнее точка, которую она представляет, подходит для использования.

Тесты могут использовать значения дистанции, точек, тэгов или имен объектов рядом, путей, линий и прочего. При этом используя значения тест в любом случае можно настроить либо на фильтрацию, либо на сохранение значения. При фильтрации также можно указать рамки, границы значений сфер, которые стоит оставлять. Границы фильтра могут быть заданы минимальным значением, максимальным значением или максимальным и минимальным значением одновременно.

Более подробный обзор узлов системы запросов к среде и их настроек приводится не будет. Все узлы и настройки можно найти в официальном источнике, посвященному системе запросов к среде [8].

`BlackBoard`, как уже упоминалось, является своего рода хранилищем данных, которые использует тот или иной персонаж. Он не является обязательным атрибутом искусственного интеллекта. Вместо него, разумеется, можно написать собственный класс хранилища, использовать сами классы, контролируемые искусственным интеллектом. Но использование `blackBoard` лучше оптимизировано и к тому же он уже реализован. Единственным недостатком, если это возможно отнести к ним, является невозможность добавления ключей-переменных для типов данных отличных от классических, таких как float, Boolean, integer, rotator, string, actor и иные.

Данный вспомогательный класс, а точнее наследованные от него классы, были использованы при реализации бота и дрона. У автоматической турели не было необходимости создавать дерево поведения, а соответственно и хранилище данных.

4. РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

4.1 Алгоритм поиска координат и пути полета дрона

Алгоритм осуществляет поиск координат над противником для полета к ним и последующего преследования противника.

1. Привести класс контролируемой сущности к классу дрона;
2. Обновить переменную дрона;
3. Взять координат противника;
4. Установить начальную точку для проверки координат;
5. Сгенерировать случайный вектор направления;
6. Домножить вектор направления на значение удаленности необходимой точки;
7. Создать и установить конечную точку для проверки координат;
8. Создать и установить игнорируемых актеров;
9. Проверить нахождение точки в зоне видимости для возможной последующей стрельбы;
10. Если найденная точка не находится в зоне видимости, перейти к шагу 4;
11. Установить используемую сетку навигации дрона для нахождения пути;
12. Установить начальную точку пути;
13. Установить конечную точку пути;
14. Установить тип объектов препятствий;
15. Найти путь к заданной точке;
16. Если путь не найден, перейти к шагу 19;
17. Обновить путь полета дрона;
18. Завершить данную задачу с установленным флагом успеха;
19. Установить используемую сетку навигации дрона для нахождения пути;
20. Установить начальную точку пути с выставленными координатами дрона;
21. Установить конечную точку пути с выставленными координатами противника, измененными по оси Z на необходимое значение;
22. Найти путь к противнику;
23. Если путь не найден, перейти к шагу 26;
24. Обновить путь полета дрона;
25. Завершить данную задачу с установленным флагом успеха;
26. Установить используемое хранилище данных;
27. Установить необходимое значение переменной `PatrolSkip` в хранилище данных;
28. Завершить задачу со сброшенным флагом успеха выполнения;

4.2 Алгоритм выбора точки над игроком для преследования

1. Установить максимальное количество попыток поиска пути;
2. Привести класс контролируемой сущности к классу дрона;
3. Установить переменную дрона;
4. Установить необходимую сетку навигации дрона;
5. Установить координаты дрона как точку начала полета;
6. Получить случайный вектор направления для поиска случайной точки;
7. Сложить с вектором глобальных координат дрона;
8. Полученный вектор координат использовать как точку конца полета при нахождении пути;
9. Назначить типы объектов препятствий;
10. Запустить поиск пути;
11. Если путь не найден, перейти к шагу 16;
12. Обновить путь полета дрона;
13. Установить необходимое хранилище данных дрона;
14. Записать конечную точку полета в переменную `TargetLocation` в хранилище дрона;
15. Завершить задачу с установленным флагом успеха завершения задачи;
16. Уменьшить счетчик попыток поиска пути;
17. Если остались попытки, вернуться к шагу 4;
18. Завершить поиск пути со сброшенным флагом успеха завершения;

4.3 Алгоритм полета к заданной точке

1. Сбросить текущий индекс полета;
2. Привести класс контролируемой сущности к классу дрона;
3. Обновить переменную текущего пути полета;
4. Привести класс контроллера текущего дрона к классу `DroneAI`;
5. Сравнить текущий индекс полета с индексом последней точки;
6. Если остались точки в пути, перейти к шагу 11;
7. Взять координаты дрона;
8. Считать координаты точки, к которой должен был долететь дрон;
9. Если схожи, завершить выполнение задачи с установленным флагом успеха выполнения;
10. Если не схожи, завершить выполнение задачи со сброшенным флагом успеха выполнения;
11. Установить необходимый контроллер искусственного интеллекта;
12. Установить конечную локальную точку полета;
13. Запустить функцию `MoveToLocationOrActor` с установленными параметрами;

14. Дождаться исполнения выхода функции с названием `OnMoveFinished`;

15. Инкрементировать количество использованных попыток;

16. Перейти к шагу 5;

4.4 Алгоритм инициализации дронов

1. Установить режим движения в режим полета;
2. Определить, деактивирован ли дрон;
3. Если не деактивирован, перейти к шагу 15;
4. Взять контроллер, контролирующий данного дрона;
5. Открепить контроллер искусственного интеллекта;
6. Собрать все прикрепленные к модели компоненты;
7. Установить симуляцию физики;
8. Установить работу гравитации;
9. Выключить обработку столкновений на работу только физики объекта;
10. Установить оставшееся время жизни компонента;
11. Считать следующий компонент для настройки;
12. Перейти к шагу 7;
13. Установить время жизни дрона на минимальное разрешенное;
14. Смерть персонажа, конец алгоритма;
15. Установить уровень текущего здоровья на максимальное допустимое дрона;
16. Используя заранее заданные имена собрать все точки патрулирования;
17. Используя функцию `GetWorldLocationFromArray` собрать координаты точек патрулирования в массив;
18. Считать текущее время игры;
19. Установить текущее время в переменную `LastTimeFired`;
20. Найти навигационную стеку используя заранее прикрепленное название;
21. Сохранить навигационную сетку в переменную `Mesh3D`;

4.5 Алгоритм стрельбы дрона

1. Считать текущее время в игре;
2. Отнять от текущего времени в игре время последней стрельбы;
3. Сравнить результат вычитания с частотой стрельбы, заданной в переменной вещественного типа `FireRate`;
4. Если прошедшее время не превышает значение частоты стрельбы, то завершить выполнение;
5. Начать стрельбу из правого орудия;

6. Узнать точные глобальные координаты орудия;

7. Установить координаты в параметры начала `LineTraceByChannel`;

8. Взять глобальное вращение орудия;

9. Из глобального вращения взять вектор направления;

10. Удлинить вектор направления на значение радиуса стрельбы;

11. Используя полученный вектор и координаты игрока, высчитать конечную точку луча;

12. Установить локальную переменной с информацией о луче;

13. Установить локацию для издаваемого игрового звука;

14. Запустить игровой звук;

15. Установить примерные координаты для функции `MakeNoise`;

16. Запустить функцию `MakeNoise`;

17. Установить точку для системы частиц;

18. Запустить исполнение системы частиц;

19. Установить начальную точку лазера для последующей отрисовки;

20. Если луч не задевает препятствие, перейти к шагу 22;

21. Установить конечную точку для отрисовки лазера;

22. Если луч не задевает противника, завершить алгоритм;

23. Нанести урон объекту на пути луча;

24. Установить задержку в половину частоты стрельбы;

25. Если стрельба из левого орудия уже была совершена, завершить алгоритм;

26. Начать стрельбу из левого орудия;

27. Установить используемый в функции стрельбы компонент на левое орудие;

28. Перейти к шагу 6;

4.6 Алгоритм обработки зрения бота

1. Установить закрепленное за ботом хранилище данных;

2. Взять обработанную информацию от органа чувств;

3. Установить необходимое значение в `SeePlayer`;

4. Если флаг успеха видимости установлен, перейти к шагу 10;

5. Установить необходимые параметры таймера к ивенту;

6. Установить значение в `LooseTimer`;

7. Установить запрашиваемого актера;

8. Установить актера, действия которого требуется предсказать;

9. Запросить предсказание действий актера;

10. Выбрать необходимое хранилище данных;

11. Обновить в хранилище данных успешно увиденного актера;

12. Установить фокус бота на успешно увиденного актера;

13. Очистить значение таймера;

14. Забрать контролируемого ботом актера;
15. Привести класс данного актера к классу бота;
16. Выбрать хранилище данных бота;
17. Обновить уровень здоровья бота;

4.7 Алгоритм стрельбы бота

1. Проверить наличие фокусировки на противнике;
2. Проверить возможность стрельбы;
3. Если оба условия удовлетворены, продолжить стрельбу;
4. Взять глобальные координаты прикрепленного оружия;
5. Взять примерные координаты противника;
6. Проверить возможность стрельбы по цели;
7. Установить точки издаваемых звуков;
8. Издать игровые звуки;
9. Запустить выполнение функции `MakeNoise`;
10. Создать систему частиц на месте выстрела;
11. Установить начальную и конечную точку отрисовки лазера;
12. Установить параметр наносимого урона;
13. Установить причину получения урона для функции `ApplyPointDamage`;
14. При попадании в противника или иного актера нанести урон;
15. Запустить таймер для события обработки скорости стрельбы;
16. На событии закрепить изменение соответствующей переменной;
17. Сбросить флаг возможности стрелять у бота;

5 ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ

Процесс тестирования является наиважнейшим этапом разработки любой системы, так как во время написания кода практически не предоставляется возможным рассмотреть все случаи и, соответственно, предусмотреть ошибки или недоработки в программе. Тестирование также призвано исследовать программный продукт для получения его уровня или по-другому качества. Помимо всего прочего, при доработке проекта возможно появление дополнительных ошибок в уже проверенном и протестированном коде.

Существует два основных этапа тестирования программы. Первый – поэтапное тестирование каждого модуля при непосредственном написании кода. Второй – поэтапное тестирование уже полностью разработанной системы.

Оба этапа равноценно важны и необходимы. Их нельзя исключать при разработке. Без первого этапа тестирования, например, невозможно построить устойчивую систему, которая будет сама минимизировать сбои программы при выходе из строя иных модулей. К тому же, отладка всей системы без данного этапа становится непосильной задачей, так как на такое тестирование в данном случае уйдет большое количество времени, что, в свою очередь, может оказаться непозволительно при создании некоторых видов программных продуктов. Без второго этапа, в свою очередь, невозможна работа программы в целом. Возможны многочисленные сбои приложений без тестирования всей системы. В основном такие сбои будут происходить из-за неправильной системы связывания модулей.

Так как разработка всей игры с нуля занимает в некоторых случаях не один год, дипломный проект был разделен на две части. Основной разрабатываемой частью данного дипломного проекта был искусственный интеллект, а не игра в целом. Поэтому в записке не будет приводится тестирование модулей, разработанных сторонним разработчиком. Исключение будет составлять встроенный плагин, используемый для построения путей в дроне. Цели и задачи плагина, весь его функционал и прочее были описаны в системном и функциональном проектировании.

Стоит также упомянуть что в Unreal Engine 4 нет встроенных систем тестирования. Причиной тому является ситуативность тестов. Порой невозможно создать унифицированные модули тестирования. Существует отдельный неофициальный плагин, разработанный сторонними разработчиками, однако он также не предоставляет необходимого функционала.

По этой причине было принято решение проводить все тестирование вручную, используя встроенные методы отладки, точки остановок, визуальные методы тестирования на ошибки. Во множестве случаев функции отображают некоторую информацию тестирования в виде сообщений, объектов на сцене разного цвета и форм.

В данном разделе, также как и в функциональном проектировании, описание и тестирование разработанных модулей будет разделено на несколько частей. Каждая из частей будет символизировать конкретного игрового персонажа.

5.1 Автоматическая турель

Первый игровой персонаж – автоматическая турель. Она проста в тестировании по причине простой логики и относительно небольшого количества функций и событий.

Сразу после запуска игры и прогрузки сцены автоматическая турель готова к работе и сразу имеет возможность стрелять по противникам. По этой причине в процессе разработке был протестирован и отлажен механизм установки начальных настроек дрона, а именно его вращения, здоровья, правильного отображения на сцене и прочие.

В процессе игры турель может получать урон от разных источников при помощи события `AnyDamage`. Оно помогает обрабатывать получение урона с возможностью дополнительной обработки типа урона. В случае получения урона также стоит обработать возможное уничтожение автоматической турели.

Также в процессе игры турель может неоднократно изменять свое положение, а именно вращение башни с закрепленными орудиями. Необходимо было протестировать не только корректное вращение орудий с башней, но и корректную стрельбу после этого. Помимо стрельбы стоило проверить и вращение по оси Y. Это было необходимо было протестировать не только на предмет ошибок, но и недоработок, для балансировки и последующего ограничения вращения башни по данной оси.

Как было сказано ранее при описании данного персонажа, органов чувств данный класс не имеет. Получение информации о противниках вокруг довольно примитивно. При получении информации он использует сферу, при попадании в которую противник фиксируется как потенциальная цель. Из всех целей, а их может быть несколько, должна корректно выбираться цель. Критерием выбора в первую очередь является возможность стрельбы по противнику. Далее, после отбрасывания целей, по которым стрельба невозможна, идет приоритезация целей по критерию удаленности от турели. Чем ближе противник, тем выше шанс того, что он не успеет скрыться за каким-либо препятствием для стрельбы. Соответственно приоритетной целью является ближайшая цель.

Последней тестируемой частью турели является ее уничтожение. При уничтожении башня вместе с прикрепленными орудиями должны визуально выделяться, напоминая тем самым то, что турель уже уничтожена и не способна вести стрельбу. Это достигается в первую очередь поворотом башни максимально вниз и незамедлительное отключение стрельбы. Помимо

поворота, для удобства пользователя, отключается столкновение турели с иными объектами.

5.2 Дрон

Следующим тестируемым персонажем является дрон. У дрона присутствует множество самописных задач, которые не были представлены в Unreal Engine 4. Рассмотрение и тестирование всех задач и сервисов заняло большую часть времени разработки данного персонажа. В дроне также был использован сторонний плагин для навигационной сетки в трехмерном пространстве. Данный плагин не тестировался, так как не являлся разрабатываемой частью.

Основными тестируемыми задачами были: полет до заданных координат, поиск и обновление пути для полета, оптимизирование пути полета.

При полете до заданной точки дрон использует заранее простроенный маршрут на сетке навигации. Были использованы все три оси сетки координат для более плавного полета и увеличения возможностей. При тестировании данной задачи необходимо было отрисовывать путь, заданный точками в массиве и вручную регулировать корректность построения пути.

Поиск и обновление пути для полета являются самыми важными задачами, так как включают как обработку информации от органов чувств, так и построение и, соответственно, обновление уже построенного пути. На тестирование данных задач также ушло большая часть времени, отведенного на тестирование.

Оптимизация пути также потребовала отрисовки изначального и оптимизированного пути разными цветами. Оптимизация в данном случае была необходима для минимизации используемой памяти, а также для более плавного полета дрона.

Органы чувств дрона были протестированы с использованием встроенных методов визуализации. Попадание в разные области и выполнение определенных задач игроком вызывало те или иные ветви поведения дрона, что и говорила о корректности работы различных органов чувств.

Само дерево поведения было протестировано лишь визуально, так как встроенных методов тестирования не предусмотрено, а при визуальном тестировании на предмет ошибок можно не заметить некоторые незначительные с первого взгляда недоработки.

5.3 Бот

Бот является наиболее сложным персонажем для тестирования так как имеет класс анимаций внутри и имеет множество систем запросов к среде. Каждый запрос к среде необходимо протестировать и сбалансировать для

более приятного геймплея. Помимо запросов к среде также было написано несколько сервисов, которые также подлежали тестированию. Дерево поведения у бота более развито, чем у дрона, что также повлияло на сложность тестирования.

Сперва стоит начать с самого низкого уровня условной абстракции – класса самого бота. В нем было необходимо протестировать стрельбу, корректную работу анимаций и корректное уничтожение при получении большого количества урона.

Хоть в игре в целом и присутствует система оружия с различной скоростью стрельбы, уроном и весом, но она не была интегрирована в бота. Причиной тому является то, что изначальной целью проекта являлось создание полностью независимого искусственного интеллекта и, соответственно, игровых персонажей для него. Изначально корректность стрельбы тестировалась при отключенном передвижении персонажа дабы исключить возможное влияние дерева поведения. В конце, после проверки работы дерева поведений, стрельба была опять протестирована на предмет ошибок.

Обработка смерти подробно уже была описана в функциональном проектировании. Стоит лишь добавить, что для увеличения скорости тестирования количество здоровья данного персонажа был понижен втрое.

Работа анимаций была протестирована при наблюдении со стороны в режиме полета. Анимация стрельбы и передвижения были уже добавлены в начальном пакете при создании проекта, но их тестирование было необходимо после добавления некоторых задач в дереве поведения.

После описания тестирования самого бота стоит описать тестирование конкретно дерева поведения и контроллера персонажа. Для более корректного тестирования был использован класс `TestingPawn`. Наследуемые от него разработанные классы помогли визуализировать и предсказывать поведение бота в игровых ситуациях по выбору точек передвижения.

Всего было разработано три системы запроса к среде: для уклонения, для патрулирования и для отступления. Все они уже были разобраны в функциональном проектировании при описании бота. Работа с системами запросов к среде также была описана в функциональном проектировании в разделе вспомогательных классов. Так как дерево поведения создавалось поэтапно, то отладка и тестирование данных систем не составило труда.

Запрос к среде для патрулирования тестировался следующим образом. Сперва использовался класс тестовой пустышки, или по-другому `TestingPawn`. После этого отдельно запускались задачи выбора точек в дереве поведения для более точного тестирования и боту требовалось пройти к заданной точке.

Для тестирования запроса к среде для уклонения потребовалось использовать помимо тестовой пустышки самого игрового персонажа.

Причиной этому являлось использование вектора направления игрока. Также использовалась информация о дистанции до игрока и окружение. Этот запрос к среде необходимо было тестировать в динамике игры.

Последний запрос к среде тестировался также с использованием персонажа, контролируемого игроком. При малом количестве здоровья бот старается избегать встреч с игроком и ограничиваться малыми очередями при стрельбе. Тут также используется дистанция до игрока.

5.4 Итоги тестирования игровых персонажей

Как говорилось ранее, автоматической системы тестирования в Unreal Engine 4 не представлено. Все тестирование производилось вручную, с использованием графических меток и проверкой на визуальное соответствие написанной логики с поведением персонажей. Стоит упомянуть что игровой искусственный интеллект практически невозможно протестировать полностью. Использование новых карт в игре может привести к лишь относительно предсказуемому поведению игровых персонажей. В большинстве случаев разработкой карт, а также тестированием и последующей доработкой, и исправлением ошибок занимаются специально обученные люди, которые способны выявить часто встречающиеся ошибки быстрее.

На данном этапе игровые персонажи были протестированы на единственной имеющейся карте игры. Все тесты были пройдены успешно, что означает возможность использования в различных проектах как дополнение, часть игрового мира.

6 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

Хотя целью дипломного проекта и является написание игрового искусственного интеллекта развлекательного типа, написание интерфейса для пользователей и персонажа для управления игроками было необходимо. Это является необходимостью потому, что для более подробного описания продукта необходимо показать возможности игровых персонажей, контролируемых искусственным интеллектом. Лучшим способом показать их возможности является написание базового персонажа и создание простого окружения с возможностью взаимодействия. В данном разделе будет описано руководство пользователя, а именно интерфейс игры, возможности пользователя в игре и сам процесс установки игры.

6.1 Пользовательский интерфейс

При запуске игры пользователь попадает в главное меню игры. На данный момент в мнею у пользователя выбор незначителен. Представлена возможность начать игру, начав прохождение с начального уровня. Есть возможность изменить некоторые настройки, включая настройки графики и звука. Также есть возможность узнать контактную информацию разработчиков, которых на данный момент два. В случае необходимости пользователь может также незамедлительно выйти из игры. На рисунке 6.1 представлено главное меню игры.

Рисунок 6.1 – Главное меню игры

При нажатии на кнопку начала игры «Play» игрок попадает на начальный уровень для ознакомления с игрой, управлением. Данный уровень полностью безопасен для игрового персонажа, что дает возможность пользователю ознакомиться с управлением. Управление существенно не изменено по сравнению с другими играми шутерами от первого лица.

При нажатии на кнопку изменения настроек игры «Options» пользователь попадает в подменю для изменения настроек графики и звука. Все настройки изменяются встроенными функциями движка, что упростило реализацию изменения настроек.

Если пользователь захочет узнать состав разработчиков, он может использовать кнопку «Credits». В данном подменю будет расположена вся необходимая информация, такая как почта разработчиков для связи и то, что именно они разрабатывали в проекте.

Нажатие же последней кнопки «Quit» закроет игру и связанные с ней процессы.

У игрока есть возможность подбирать различное оружие, взаимодействовать с кнопками и дверьми в процессе игры. Для лучшего восприятия игры, был разработан пользовательский интерфейс. Он призван помогать пользователю освоить управление и напоминать о некоторых возможных функциях игрового персонажа. Некоторый интерфейс является всплывающим, как например интерфейс возникающий при попытке взять оружие, располагающееся на карте.

Более подробное рассмотрение возможностей игрового персонажа и интерфейс для пользователей будет описан в записке по дипломному проекту у Кухарева Льва Михайловича, так как разработкой данного персонажа занимался он. Главное меню, меню настроек графики и звука, прочий пользовательский интерфейс также разрабатывался им, но будет кратко описан также и в данном разделе.

При попытке поднять оружие любого типа, над ним будет предоставлена информация о возможном взаимодействии и о типе оружия. При использовании же появляется дополнительная информация об оставшемся боекомплекте и типе оружия. Данный интерфейс реализован с помощью виджетов.

Так как у каждого игрового персонажа есть определенное количество здоровья, в том числе и у игрока, то также есть виджет предоставляющий пользователю информацию о текущем уровне здоровья.

6.2 Системные требования

При разработке игры было решено ориентироваться на разработку под операционную систему Windows 10. Причиной этому является в первую очередь огромное количество потенциальных пользователей игры. Второй причиной является наличие опыта разработки игры под данную

операционную систему. Стоит отметить, что при малейших правках проект может быть пересобран для операционных систем Linux по причине схожести управления персонажем, а также по причине того, что вся разработка велась исключительно средствами BluePrint.

Данный проект все еще на этапе разработки, по причине отсутствия в команде разработчиков опытных дизайнеров уровней, графических редакторов и разработчиков, работающих со звуком. По этой причине проект не требователен к оперативной памяти, 2ГБ оперативной памяти хватит для прохождения игры на максимально возможных графических и иных настройках игры. На диске игра будет занимать не более 2ГБ памяти.

По той же причине, проект не является требовательным к графическим картам. Графические карты, начиная с NVIDIA GeForce GTX 650 для корректной обработки и плавной игры будет хватать пользователю. Особых требований к процессору также нет. Центральные процессоры, произведенные с начала 2010 года, будут удовлетворять все требования игры.

Все описанные выше требования являются минимальными. Повышение характеристик безусловно повысит плавность игры и скорость вычислений, но не даст каких-либо значительных преимуществ в игровом плане.

6.3 Установка программного продукта

Установка игры как таковая отсутствует. Запуск игры осуществляется при помощи запуска исполняемого файла, заранее созданного встроенными средствами Unreal Engine 4. Само создание данного исполняемого файла занимает длительное время, так как происходит перекомпиляция всего проекта, запекание света и прочие необходимые операции над используемым контентом.

Вместе с исполняемым файлом создаются различные бинарные файлы для использования во время игры, встраиваются динамические и статические библиотеки в папку игры. Их удаление приведет к некорректной работе игры. Сама папка с исполняемым файлом непосредственно занимает не более 1ГБ памяти, остальная память может понадобится для хранения возможных сохранений игры, временных файлов и прочего.

При создании готового приложения была использована функция для разработчиков, что означает дополнительные возможности в игре, а именно просмотр информации для тестирования. Пользователю предоставляется возможность использовать различные консольные команды. Эта функция была необходима потому как цель данного дипломного проекта не разработка полноценной игры, а лишь ее части – искусственного интеллекта для последующего использования в различных проектах. Данная функция облегчит обозревание игровых персонажей, контролируемых искусственным интеллектом, и игры в целом.

7 ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ КОМПЬЮТЕРНОЙ ИГРЫ НА БАЗЕ UNREAL ENGINE 4

7.1 Характеристика разработанного программного средства

Созданный дипломный проект представляет собой игру, созданную на базе движка Unreal Engine 4. Основной разработанной частью являлся искусственный интеллект, который способен подстраиваться под действия игроков. Персонаж был создан таким образом, чтобы показать возможный функционал искусственного интеллекта. Карты и, соответственно, все, что может на ней делать игрок, были созданы для той же цели.

На данный момент в игре присутствует три типа искусственного интеллекта: робот, дрон и автоматическая турель. Они указаны в порядке усложнения реализации и, соответственно, повышения уровня качества интеллекта.

Аналоги уже были разобраны в разделе обзора литературы. Но стоит опять же отметить, что каждый искусственный интеллект разрабатывается для своей цели, которые, в свою очередь, зависят от того, для кого игра будет предоставляться и как она должна будет выглядеть.

Робот на основе собираемой информации об окружении и, в частности, игроке способен выбирать приоритетную задачу для немедленного исполнения. Примером задачи может служить передвижение в более выгодную позицию для относительно безопасного наблюдения над игроком, для ведения стрельбы или простое уклонение от возможных атак самого противника. При этом может даваться и последующая задача для выполнения, когда после передвижения необходимо, например, исследовать территорию для нахождения игрока в ней.

Дрон более прост в реализации, менее развит, но оказывает большую поддержку роботам. При патрулировании, заметив противника, которым является игрок, он может отослать сигнал ближайшим дронам и роботам о нахождении игрока в области. Так же он может атаковать персонажа, но не способен услышать его действия (стрельба, взаимодействие с окружением). Данный персонаж потребовал наибольших усилий при разработке. Были созданы задачи, сервисы и особые функции обработчики, которые заменяют те же задачи, которые существую у бота по умолчанию. Для реализации некоторых задач было прочитано и изучено большое количество источников информации.

Автоматическая турель еще проще в реализации, но опять же менее развит. Она не может слышать персонажа, как и дрон, также не может отсылать сигнал. Атаковать будет любую ближайшую движущуюся цель противоположной команды в настраиваемом радиусе.

Основными задачами созданных противников является развлечение игрока, а не победа над ним, что может понравится многим людям. В

будущем планируется добавить кооперативный режим для совместной игры нескольких людей на одном уровне, увеличение количества вооружения, а также улучшение графической и звуковой составляющей игры. Также в планах увеличение возможных взаимодействий игрока с окружением.

Целевой аудиторией данного программного продукта являются прежде всего подростки. Возрастные ограничения определить сложно в связи с большой вероятностью доработки программного продукта. На данный момент, в игре не представлены какие-либо сцены, позволяющие хотя бы примерно оценить минимальные возрастные ограничения. Основные трудности могут возникнуть лишь с установкой и прохождением. По этим причинам, стоит поставить минимальное возрастное ограничение в двенадцать лет. С этого возраста человек может относительно разумно распоряжаться денежными средствами, необходимыми для покупки данного программного продукта, а также корректно установить игру.

7.2 Расчет инвестиций в разработку программного средства

7.2.1 Расчет зарплат на основную заработную плату разработчиков

В расчет затрат на создание программного продукта входит количество людей, причастных к разработке программного продукта, сложности и объема их работы. Затраты на основную заработную плату рассчитаны по формуле:

$$З_о = К_{пр} \sum_{i=0}^{n} З_{чi} \cdot t_i \qquad (7.1)$$

где $К_{пр}$ – коэффициент премий;

n – категория исполнителей, которые заняты разработкой;

$З_{чi}$ – часовая заработная плата исполнителя i-ой категории, р.;

$Т_i$ – трудоемкость работ исполнителя i-ой категории, ч.

Разработкой искусственного интеллекта для игры будут заниматься инженер-системный программист-геймдизайнер и инженер-тестировщик. Основными задачами программиста-геймдизайнера являются создание самого искусственного интеллекта, кратко описанного выше, а также карты и написание простой логики для управления персонажа. Задачей тестировщика будет выявление ошибок и балансировка способностей искусственного интеллекта.

Заработная плата выбрана в соответствии со средними показателями по стране для Junior разработчика. Часовая заработная плата исполнителей высчитывается путем деления месячной заработной платы на количество рабочих часов в месяце. За количество рабочих часов в месяце принято 168

часов. Размер премии будет равен 25% от основной заработной платы. Затраты на основную заработную плату показаны в таблице 7.1.

Таблица 7.1 – Затраты на основную заработную плату

Категория исполнителя	Месячная заработная плата, р.	Часовая заработная плата, р.	Трудоемкость работ, ч.	Итого, р.
инженер-системный программист-геймдизайнер	2360	14.04	168	2360
Инженер-тестировщик	2065	12.29	168	2065
Итого				4425
Премия				1106
Всего затрат на основную заработную плату				5531

7.2.2 Расчет затрат на дополнительную заработную плату разработчиков

Формула, использованная для расчета затрат на дополнительную заработную плату:

$$З_д = \frac{З_о \cdot Н_д}{100} \quad (7.2)$$

где $Н_д$ – норматив дополнительной заработной платы.

7.2.3 Расчет отчислений на социальные нужды

Размер отчислений на социальные нужды определяется ставкой отчислений, которая, в соответствии с действующим законодательством по состоянию на 10.04.2022, составляет 34,6%. Размер отчислений можно рассчитать по формуле:

$$Р_{соц} = \frac{(З_о + Н_д) \cdot Н_{соц}}{100} \quad (7.3)$$

где $Н_{соц}$ – ставка отчислений на социальные нужды.

7.2.4 Расчет прочих расходов

Прочие расходы рассчитываются с учетом норматива прочих расходов. Приняв это значение равным 35 %, рассчитаем прочие расходы по формуле:

$$Р_{пр} = \frac{З_о \cdot Н_{пр}}{100} \quad (7.4)$$

где $Н_{пр}$ – норматив прочих расходов.

7.2.5 Расчет расходов на реализацию

Для расчета расходов на реализацию необходимо знать норматив расходов на него. Он взят за 4%. Формула, которая была использована для расчета:

$$Р_р = \frac{З_о \cdot Н_р}{100} \quad (7.5)$$

где $Н_р$ – норматив расходов на реализацию.

7.2.6 Расчет общей суммы затрат

Определим общую сумму затрат как сумму ранее вычисленных расходов: на основную заработную плату, дополнительную заработную плату, отчислений на социальные нужды, расходы на реализацию и прочие расходы. Значение определяется по формуле:

$$З_р = З_о + З_д + Р_{соц} + Р_{пр} + Р_р \quad (7.6)$$

Найдем величину затрат на разработку программного средства в таблице 7.2, используя формулы, указанные выше.

Таблица 7.2 – Затраты на разработку

Название статьи затрат	Расчеты по формуле	Значение, р.
Основная заработная плата разработчиков	См. таблицу 7.1	5531.25
Дополнительная заработная плата разработчиков	$З_д = \frac{5531.25 \cdot 15}{100}$	829.69
Отчисление на социальные нужды	$Р_{соц} = \frac{(5531.25 + 829.7) \cdot 34.6}{100}$	2200.88

Продолжение таблицы 7.2

Прочие расходы	$Р_{пр} = \frac{5531.25 \cdot 35}{100}$	1935.94
Расходы на реализацию	$Р_{р} = \frac{5531.25 \cdot 4}{100}$	221.25
Общая сумма затрат на разработку и реализацию	$З_{р}$ = 5531.25+ 829.69 + 2200.88 + 1935.94 + 221.25	10719

7.3 Расчет экономического эффекта от реализации программного средства на рынке

Для расчета экономического эффекта организации-разработчика программного средства, а именно чистой прибыли, необходимо знать такие параметры как объем продаж, цену реализации и затраты на разработку.

Соответственно необходимо создать обоснование возможного объема продаж, количества проданных лицензий программного средства, купленного пользователями. Так как аналогов данного продукта на рынке довольно большое количество, то за объем продаж было взято 5000 копий лицензий разработанной игры.

Цена продукта была определена с учетом цен на аналогичные продукты, которые представлены для покупки на рынке, а также после опроса возможной аудитории. Отпускную цену было решено принять за 15 р.

Для расчета прироста чистой прибыли, необходимо учесть налог на добавленную стоимость. Расчет данного налога можно определить по формуле:

$$НДС = \frac{Ц_{отп} \cdot N \cdot Н_{д.с}}{100\% + Н_{д.с}} \quad (7.7)$$

где N – количество копий(лицензий) программного продукта, реализуемое за год, шт.; $Ц_{отп}$ – отпускная цена копии программного средства, р.; $Н_{д.с}$ – ставка налога на добавленную стоимость, %.

Ставка налога на добавленную стоимость по состоянию на 10.04.2022 в соответствии с действующим законодательством составляет 20%. Используя данное значение, посчитаем НДС:

$$НДС = \frac{15 \cdot 5000 \cdot 20\%}{100\% + 20\%} = 12500 \text{ р.} \quad (7.8)$$

Зная налог на добавленную стоимость, можно рассчитать сам прирост чистой прибыли, которую получит разработчик от продажи программного продукта. Это можно сделать по формуле:

$$\Delta П_{ч}^{р} = (Ц_{отп} \cdot N - НДС) \cdot Р_{пр} \cdot \left(1 - \frac{Н_{п}}{100}\right) \quad (7.9)$$

где N – количество копий(лицензий) программного продукта, реализуемое за год, шт.; $Ц_{отп}$ – отпускная цена копии программного средства, р.; НДС – сумма налога на добавленную стоимость, р.; $Н_{п}$ – ставка налога на прибыль, %; $Р_{пр}$ – рентабельность продаж копий.

Ставка налога на прибыль согласно действующему законодательству, по состоянию на 10.04.2022 является 18%. Рентабельность продаж копий была взята на уровне 30%. Зная ставку налога и рентабельность продаж копий (лицензий), был рассчитан прирост чистой прибыли для разработчика:

$$\Delta П_{ч}^{р} = (15 \cdot 5000 - 12500) \cdot 0.3 \cdot \left(1 - \frac{18}{100}\right) = 15375р. \quad (7.10)$$

7.4 Расчет показателей экономической эффективности разработки и реализации программного средства на рынке

Для оценки экономической эффективности разработки и реализации программного продукта на рынке необходимо учитывать то, сколько было затрачено на разработку данного программного продукта, а также то, сколько было получено чистой прибыли за год.

Так как сумма затрат на разработку меньше суммы годового экономического эффекта, что означает, что инвестиции окупятся мене чем через один год, то оценить экономическую эффективность инвестиций можно при помощи расчета рентабельности инвестиций (Return on Investment, ROI). Формула, использованная для расчета ROI:

$$ROI = \frac{\Delta П_{ч}^{р} - З_{р}}{З_{р}} \cdot 100\% \quad (7.11)$$

где $\Delta П_{ч}^{р}$ – прирост чистой прибыли, полученной от реализации программного средства на рынке информационных технологий, р.; $З_{р}$ – затраты на разработку и реализацию программного средства, р.

$$ROI = \frac{15375 - 10719}{10719} \cdot 100\% = 43.44\% \quad (7.12)$$

7.5 Вывод об экономической эффективности

После всех расчетов технико-экономического обоснования, среди которых расчет инвестиций, необходимых для разработки программного

продукта, расчет экономического эффекта от реализации конечного продукта на рынке, расчет рентабельности инвестиций.

Общая сумма затрат на разработку и реализацию составила 10719 рублей. Отпускная цена была принята за 15 рублей. Прирост чистой прибыли за год, с учетом, что будет продано 5000 копий, составит 15375 рублей. Рентабельность инвестиций составит 43.44% за год.

Учитывая полученные в результате расчетов данные, можно сделать вывод о целесообразности разработки данного продукта, а также об оправданности инвестиций в него. Однако, как отмечалось ранее, данных результатов можно добиться при продаже 5000 копий, что является риском, потому что всегда существует вероятность того, что продукт может остаться незамеченным на рынке. Стоит так же отметить, что при последующей поддержке и развитии, проект может не только просто пройти отметку в 5000 приобретенных лицензий, но и получить иные способы получения заработка за него.

ЗАКЛЮЧЕНИЕ

За время работы над дипломным проектом был разработан и реализован алгоритм игрового искусственного интеллекта развлекательного типа. В данном проекте могут быть заинтересованы подростки. Возрастные ограничения определить сложно в связи с большой вероятностью доработки программного продукта. На данный момент, в игре не представлены какие-либо сцены, позволяющие хотя бы примерно оценить минимальные возрастные ограничения. Основные трудности могут возникнуть лишь с установкой и прохождением.

Проект разработан с использованием известных, надежных и активно развивающихся технологий. Для реализации алгоритмов искусственного интеллекта были использованы такие технологии, как C++, Unreal Engine 4 и BluePrints. C++ использовался для реализации функциональности плагина сторонним разработчиком. Плагин используется для пространственной ориентации дрона. Unreal Engine 4 был использован как игровой движок, так как разработка игры с использованием стандартных графических библиотек, таких как OpenGL, занимает слишком много инженерного времени и человеческих ресурсов, что является нецелесообразным. BluePrints были использованы для обеспечения обратной совместимости проекта при переходе на новые версии игрового движка.

На данный момент в игре присутствует три типа искусственного интеллекта: робот, дрон и автоматическая турель. Они указаны в порядке усложнения реализации и, соответственно, повышения уровня качества интеллекта.

Аналоги уже были разобраны в разделе обзора литературы. Но стоит опять же отметить, что каждый искусственный интеллект разрабатывается для своей цели, которые, в свою очередь, зависят от того, для кого игра будет предоставляться и как она должна будет выглядеть.

Робот на основе собираемой информации об окружении и, в частности, игроке способен выбирать приоритетную задачу для немедленного исполнения. Примером задачи может служить передвижение в более выгодную позицию для относительно безопасного наблюдения над игроком, для ведения стрельбы или простое уклонение от возможных атак самого противника. При этом может даваться и последующая задача для выполнения, когда после передвижения необходимо, например, исследовать территорию для нахождения игрока в ней.

Дрон более прост в реализации, менее развит, но оказывает большую поддержку роботам. При патрулировании, заметив противника, которым является игрок, он может отослать сигнал ближайшим дронам и роботам о нахождении игрока в области. Так же он может атаковать персонажа, но не способен услышать его действия (стрельба, взаимодействие с окружением). Данный персонаж потребовал наибольших усилий при разработке. Были созданы задачи, сервисы и особые функции обработчики, которые заменяют

те же задачи, которые существую у бота по умолчанию. Для реализации некоторых задач было прочитано и изучено большое количество источников информации.

Автоматическая турель еще проще в реализации, но опять же менее развит. Она не может слышать персонажа, как и дрон, также не может отсылать сигнал. Атаковать будет любую ближайшую движущуюся цель противоположной команды в настраиваемом радиусе.

Система позволяет уменьшить затраты компаний на выделенных отдельно для развертывания и настройки кластера специалистах, а также на поддержке кластера, так как система предоставляет заказчику готовую инфраструктуру в помесячную аренду.

Проведя расчет экономической эффективности, можно сделать вывод, что проектирование и разработка данного алгоритма является целесообразной, принесут выгоду как компании-разработчику, так и компании-заказчику.

Проект был разработан в соответствии с поставленными задачами, вся функциональность была реализована в полном объеме.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Искусственный интеллект в DooM 1993 [Электронный ресурс]. – Электронные данные. – Режим доступа: https://youtu.be/f3O9P9x1eCE – Дата доступа: 28.03.2022

[2] Искусственный интеллект в DooM 2016 [Электронный ресурс]. – Электронные данные. – Режим доступа: https://youtu.be/2KQNpQD8Ayo – Дата доступа: 28.03.2022

[3] Искусственный интеллект в Star wars: battlefront II [Электронный ресурс]. – Электронные данные. – Режим доступа: https://www.ea.com/ru-ru/games/starwars/battlefront/star-wars-battlefront-2/news/ai-article-deep-dive – Дата доступа: 29.03.2022

[4] Игровой движок [Электронный ресурс]. – Электронные данные. – Режим доступа: https://club.dns-shop.ru/blog/t-64-videoigryi/34701-chto-takoe-igrovoi-dvijok – Дата доступа: 29.03.2022

[5] Руководство по работе с blueprint [Электронный ресурс]. – Электронные данные. – Режим доступа: https://docs.unrealengine.com/4.27/en-US/Programming AndScripting/Blueprints – Дата доступа: 29.03.2022

[6] Основы C++ в Unreal Engine 4 [Электронный ресурс]. – Электронные данные. – Режим доступа:https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/IntroductionToCPP – Дата доступа: 30.03.2022

[7] Описание узлов для behavior tree [Электронный ресурс]. – Электронные данные. – Режим доступа: https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/BehaviorTreeNodeReference – Дата доступа: 30.03.2022

[8] Описание узлов для EQS [Электронный ресурс]. – Электронные данные. – Режим доступа: https://docs.unrealengine.com/4.27/en-US/Interactive Experiences/ArtificialIntelligence/EQS – Дата доступа: 26.04.2022