**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.

**2.1 Классы игровых сущностей**

**2.1.1 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, будет описаны основные исполняемые задачи. Как бот выглядит внешне, показано на рисунке 2.1.



Рисунок 2.1 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,SkeletalMeshи CapsuleComponent.

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. 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 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 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, уcтановленное 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 MovementComponentсобытия 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 Shootingпризван 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: HitActorи 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 2.1 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.

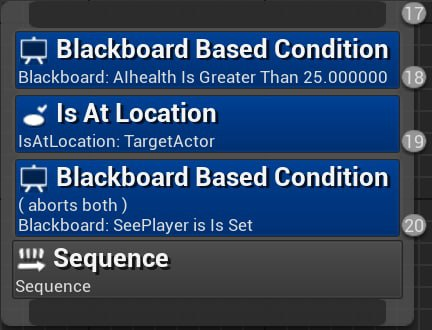


Рисунок 2.1 – Порядок проверки несколькими декораторами

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 EnvironmentQuerySystem (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.

**2.1.2 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 2.1.1, 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 2.2.

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.



Рисунок 2.2 Внешность дрона

Стрельба потребовала добавления точек начала стрельбы. Для точек был выбран компонент **Arrow**. Представляет собой обычную стрелку с направлением по оси Х. 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.

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.

**2.1.3 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. Основной задачей турели является атака любого объекта, способного перемещаться. Это может быть как игрок, так и дрон или бот. Его внешность показана на рисунке 2.3.

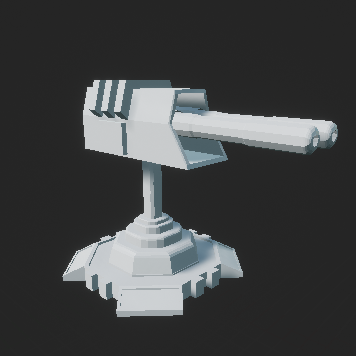


Рисунок 2.3 – Внешность автоматической турели

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.

Класс турели **Turret** наследуется от класса **actor** и не имеет встроенных компонентов. Для визуального восприятия была взята модель стандартной турели. Модель состоит из трех частей – основание, обертка и стволы оружия. Иерархия не важна с точки зрения разработки, но имеет смысл закрепить стволы за обертку, а обертку за основание. Количество стволов не принципиально, но было решено поставить два.

Также, помимо компонентов, необходимо было добавить переменные. Одна отвечает за состояние и говорит, разрушена или отключена турель – **Active**. За состоянием уровня здоровья помогает следить переменная вещественного типа **Health**. Ее значение выставляется в начале игры турели – при загрузке соответствующего уровня, на котором установлена турель. **FallRotation** – переменная вспомогательная, представляет собой вектор из трех вещественных чисел. Необходима только в конце, при разрушении турели. Переменная вещественного типа **LactTimeFired** помогает стрелять с заданной частотой. **TurretTarget** хранит актера, которого турель считает целью. Это, как уже писалось выше, может быть игрок, бот или дрон.

Также существует ряд констант, которые упрощают разработку и отладку. Первая константа – **SenseRange**. Она хранит радиус, который турель может сканировать на предмет нахождения противника. **RotateSpeed** отвечает за скорость поворота обертки и стволов соответственно. Важный параметр, так как от скорости поворота зависит то, как быстро турель сможет менять цели. **FireRate** помогает задавать частоту стрельбы. Она у турели самая быстрая в игре. Урон турели записан в переменную **Damage**. Это урон от одного ствола турели. Значит, урон, наносимый за выстрел, вдвое больше, так как стволов у турели на данный момент два. Все константы нельзя изменять в игре, но можно считывать при необходимости.

Далее стоит описать функции, реализованные в турели: **FireTurret**, **FindTarget**, **RotateTurret**, **FireFrom**. А также события, запускающие эти функции в графе событий: **Tick**, **BeginPlay**, **Fire**, **AnyDamage**.

**FireTurret** – функция, которая закреплена на событии **Tick**. Так как это событие запускается почти каждый кадр, то выполнение функции будет тоже частым. В функции проверяется два условия. Первое условие завязано на времени, частоте стрельбы. Так как событие вызывается почти каждый кадр, привязка ко времени была необходима. Привязка реализовывалась таким же образом, как и в дроне – создавалась локальная переменная вещественного типа, в которую записывалось время последней стрельбы. Если интервал между временем последней стрельбы и текущим временем достаточен, стрельба разрешается и начинается проверка следующего условия – наведения на цель. В случае прохождения проверок, турель может стрелять и запускается событие **Fire**, которое будет описано ниже в этом же пункте.

**FindTarget** необходимая функция для нахождения противников в радиусе, который задается одной из констант. Основой является встроенная функция **SphereOverlapActors**. Данная функция выдает все экземпляры классов в радиусе, указываемом в параметрах. Тип объектов также задается входными параметрами. В данном случае выбраны объекта типа **Pawn**. Далее запускается цикл, в котором ищется актер, который находится в зоне видимости. Тут стоит отметить, что итогом может являться нахождение только одной цели. Если в зоне видимости находится несколько актеров, выбирается ближайший из них, что соответствует приоритету атаки. Чем ближе противник, тем опаснее для турели он – повышается вероятность попадания по турели.

**RotateTurret** – функция, которая помогает задавать радиус поворота турели. Основная задача – поворот к цели, но ниже, при описании разрушения турели, будет рассмотрено ее второе предназначение. Поворот происходит плавнее при большем количестве тиков, выделенных турели, поэтому турели выдано значение близкое к единице, что соответствует большему приоритету. В случае, если выдать меньший приоритет, поворот станет дерганным. Также стоит учитывать скорость поворота. Основной функцией, используемой тут, является функция **RInterp**, которая интерполирует значения одной переменной к другой со временем и с заданной скоростью. Тут же сделаны ограничение при повороте, чтобы турель не могла застрять в себе же при повороте. Она не может поворачиваться по оси Y ниже 45 градусов и выше 75 градусов.

**FireFrom** позволяет стрелять лазером при необходимости из одного из стволов. При это будет проверятся лучом, попадает ли турель по цели. В случае промаха нет необходимости наносить урон несуществующей цели. В случае же попадания, надо отправить объекту, в который попали, информацию об нанесенном уроне. Это достигается путем использования функции **ApplyDamage**.

**Fire** – основное событие, которое вызывает стрельбу турели. При этом будет произведено два выстрела – один из левого и один из правого ствола с некоторой задержкой. Тут также идет проверка на время последнего выстрела, как и в уже описанной функции **FireTurret**.

**Tick** – событие, вызывающееся почти каждый кадр. К этому событию прикреплены функции **FindTarget**, **RotateTurret**, **FireTurret**, которые были описаны выше. Помимо этого проверяется, не разрушена ли турель. В случае, если разрушена, начинается уничтожение турели. Если быть точнее – деактивации. Турель переходит в неактивный режим и перестает стрелять по противникам. Тут же используется функция **RotateTurret**, которая показывает визуально игроку, что турель неактивна. Это показывается медленным сворачиванием турели – стволы опускаются к основанию турели.

**BeginPlay** – событие, которое выполняется лишь раз, в начале игры или загрузки уровня. К этому событию закреплена лишь настройка начального уровня здоровья.

**AnyDamage** – событие, необходимое для получения урона турелью. Тут обновляется уровень здоровья. Вызов проверок на разрушение проводится в событии **Tick**.

Турель, как уже было расписано, не является сложным искусственным интеллектом. Она является лишь частью окружения с базовыми навыками – сканированием в радиусе и стрельбой. Однако стоит отметить, что турель является уже полностью завершенной и ее дальнейшая доработка необязательна.

**2.2 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 2.4 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 Х) и в зависимости от расстояния до точки.

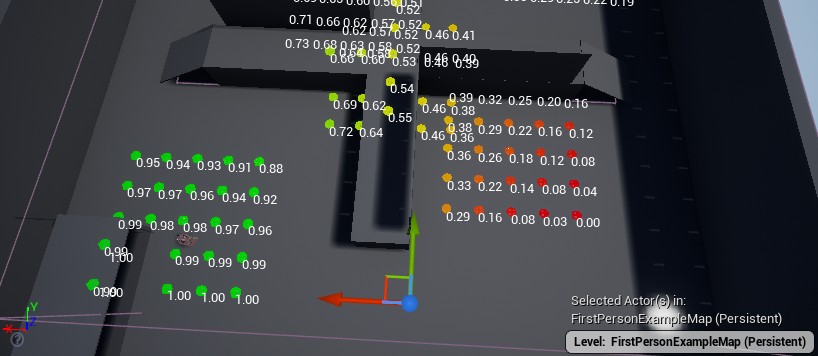


Рисунок 2.4 – Пример работы EQS

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 2.5 показан пример его использования. Зеленой областью закрашена территория, по которой теоретически может передвигаться бот. Он, **NavMesh**, имеет множество настроек, рассмотрение которых не будет приведено в данной записке по дипломному проектированию по причине их большого количества и незначительности при работе над данной игрой. 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 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.

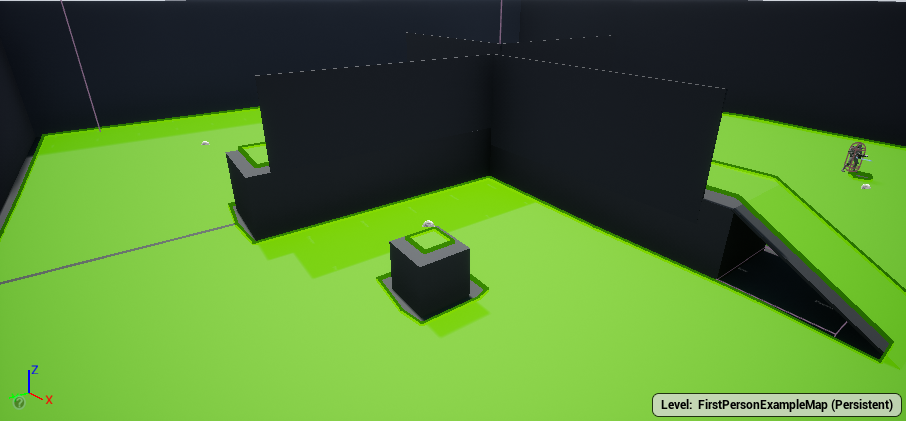


Рисунок 2.5 – NavMesh для бота

**NavMesh** **3D** – 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 2.6 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 2.7 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.

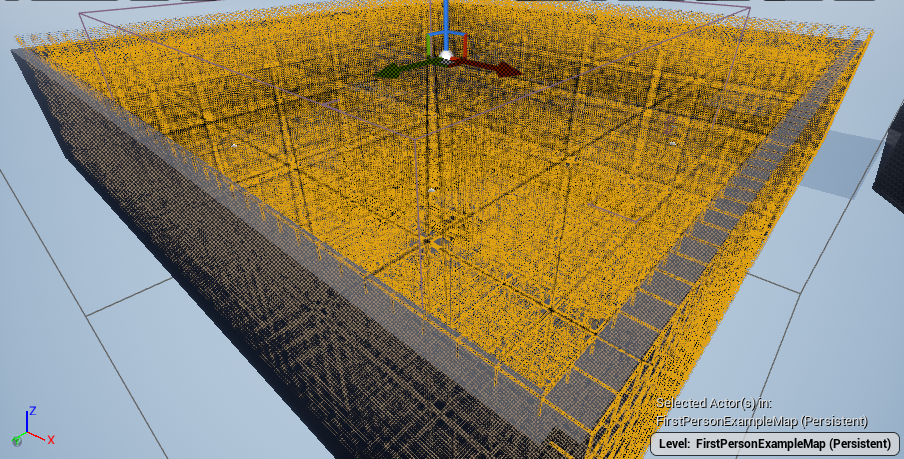


Рисунок 2.6 – Навигационная сета для дрона



Рисунок 2.7 – Построение путей у 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.

**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 2.8 показан типичный пример создания системы запроса к среде.

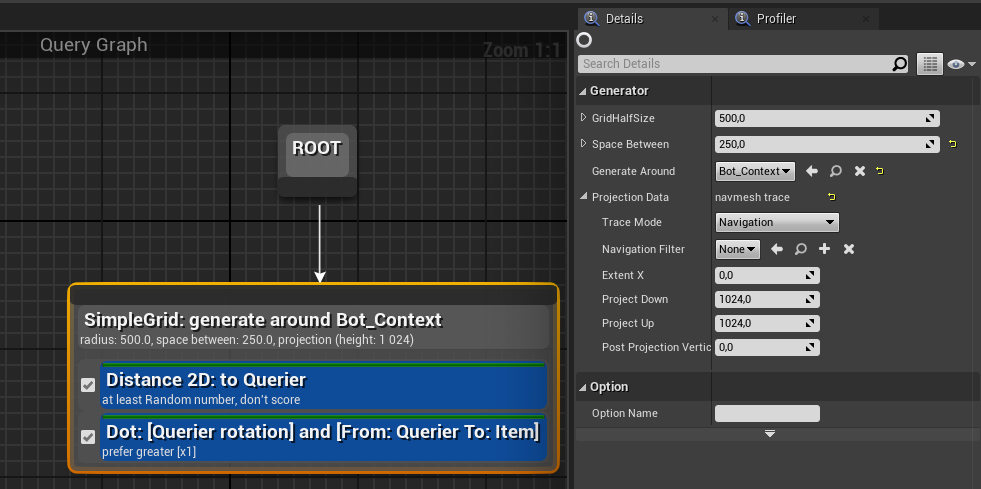


Рисунок 2.8 – Запрос к среде

На рисунке 2.8 показан запрос к среде для патрулирования ботом территории. На нем видно, что граф похож на дерево поведений – также есть корен, 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 рисунке 2.4. На том же рисунке 2.4 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.