3. ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

После определенных в структурном проектировании функциональных блоков необходимо более подробно описать реализацию данных модулей. 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 модули:

- 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

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.

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

2.1 Классы игровых сущностей

2.1.1 Бот

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 **CapsuleComponent**.

Первый 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, **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 анимацию. Как уже было сказано выше, основной разрабатываемой частью был именно искусственный интеллект, а не графическая и звуковая части. 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 и установке галочки напротив опции добавления начального контента. Помимо анимаций персонажа, в начальном контенте 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. Позволяет не падать в пропасть в начале игры. Также он позволяет обрабатывать стрельбу в некоторых случаях. Представляет собой обыкновенную капсулу с настраиваемыми размерами и некоторыми другими свойствами. Обработка коллизии происходит просто – сперва проверяются настройки объектов, связанные с игнорированием коллизии, далее уже 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 в них, 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. **MaxHealth** – переменная, которая отвечает 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 в процессе игры или изменена. Также была необходимость добавить точку для начала стрельбы. Далее в описании стрельбы будет подробнее описана ее функция. Типом компонента является **StaticMesh**, а названием является **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.

По причине отсутствия символьного представления в языке blueprint, функционал разработанных классов, будет максимально описан именно в этом разделе.

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, 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 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.

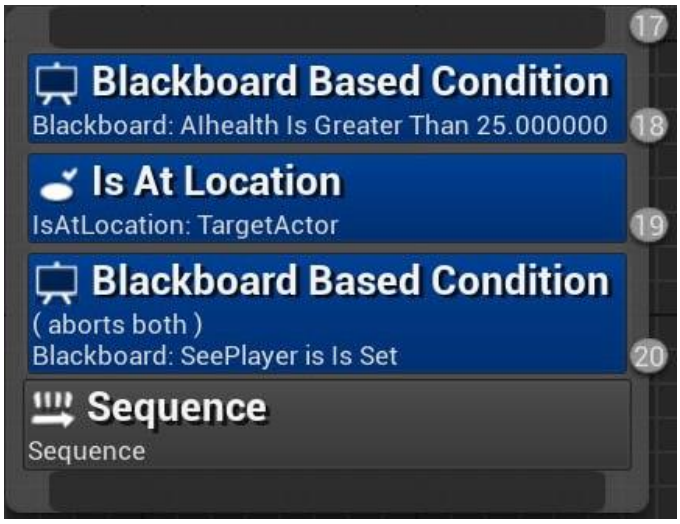

Рисунок 2.1 – Порядок проверки несколькими декораторами

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 **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.

2.1.2 Дрон

2.1.3 Автоматическая турель

2.2 Вспомогательные классы

2.2 Описание структуры и взаимодействия классов