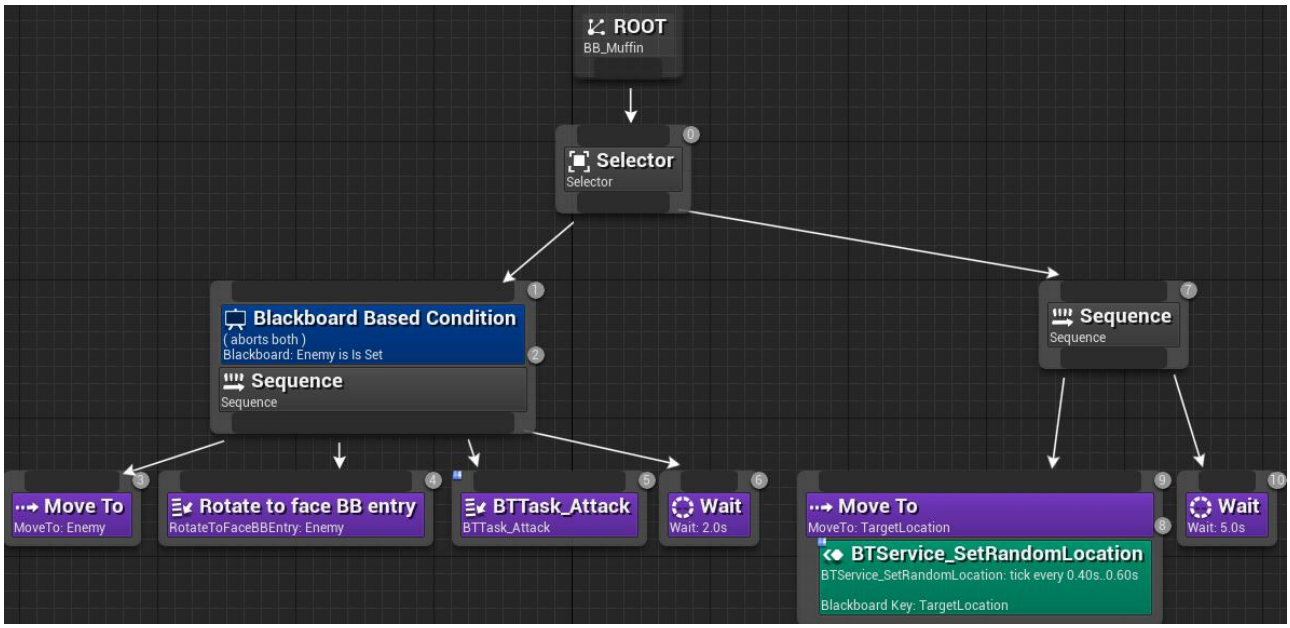
Туториал по Unreal Engine. Часть 9: Искусственный интеллект



В индустрии видеоигр искусственным интеллектом (Artificial Intelligence, AI) обычно называют процесс принятия решений не управляемыми игроком персонажами. Он может быть простым: враг видит игрока и атакует. Или же более сложными, например, управляемый ИИ противник в стратегии реального времени.

В Unreal Engine создавать ИИ можно с помощью *деревьев поведения*. Дерево поведения (behavior tree) — это система определения *поведения*, используемого ИИ. Например, у него может быть поведение боя или бега. Можно создать дерево поведения, при котором ИИ будет драться с игроком, если его здоровье выше. Если оно ниже 50%, то он будет убегать.

В этом туториале вы научитесь следующему:

- Создавать ИИ-сущность, которая может управлять элементом Pawn
- Создавать и использовать деревья поведения и blackboard
- Использовать AI Perception, чтобы дать Pawn зрение
- Создавать поведения, чтобы Pawn мог ходить и атаковать врагов

Приступаем к работе

Скачайте https://koenig-media.raywenderlich.com/uploads/2017/12/MuffinWarStarter.zip и распакуйте её. Перейдите в папку проекта и откройте *MuffinWar.uproject*.

Нажмите на *Play*, чтобы запустить игру. *Нажимайте левой клавишей мыши* внутри огороженной области для создания маффина.

В этой части туториала мы создадим AI, который будет бродить по экрану. Когда вражеский маффин попадает в поле зрения ИИ, он подходит к врагу и атакует его.

Для создания ИИ-персонажа, нам нужно три вещи:

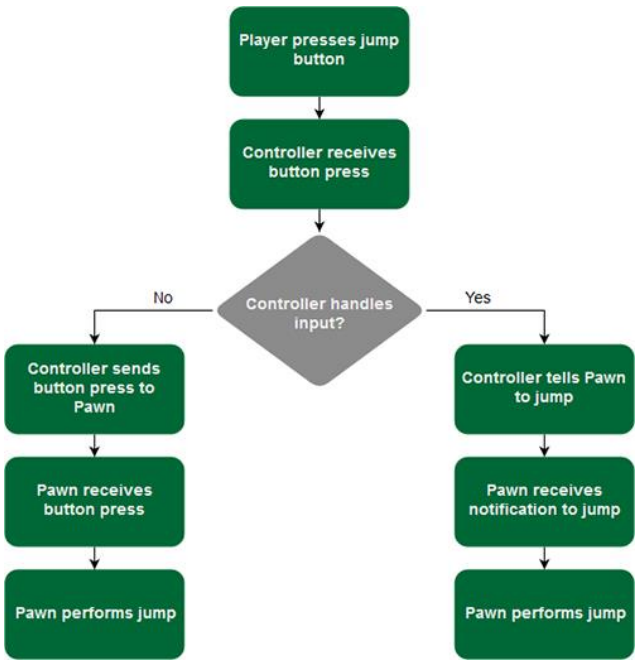
1. *Тело:* физическая форма персонажа. В нашем случае телом является маффин.
2. *Душа:* сущность, управляющая персонажем. Это может быть игрок или ИИ.
3. *Мозг:* то, как ИИ принимает решения. Мозг можно создавать разными способами, например, с помощью кода на C++, Blueprints или деревьев поведения.

У нас уже есть тело, так что нужны ещё душа и мозг. Сначала мы создадим *контроллер,* который будет являться «душой».

Что такое «контроллер» (Controller)?

Контроллер — это нефизический актор, который может *вселяться* в Pawn. Вселение позволяет контроллеру (как можно догадаться) *управлять* Pawn. Но что в этом контексте означает «управлять»?

Для игрока это означает, что при нажатии клавиши Pawn будет что-то делать. Контроллер получает ввод игрока и отправляет введённые данные Pawn. Также контроллер может сам обрабатывать ввод и приказывать Pawn выполнить действие.

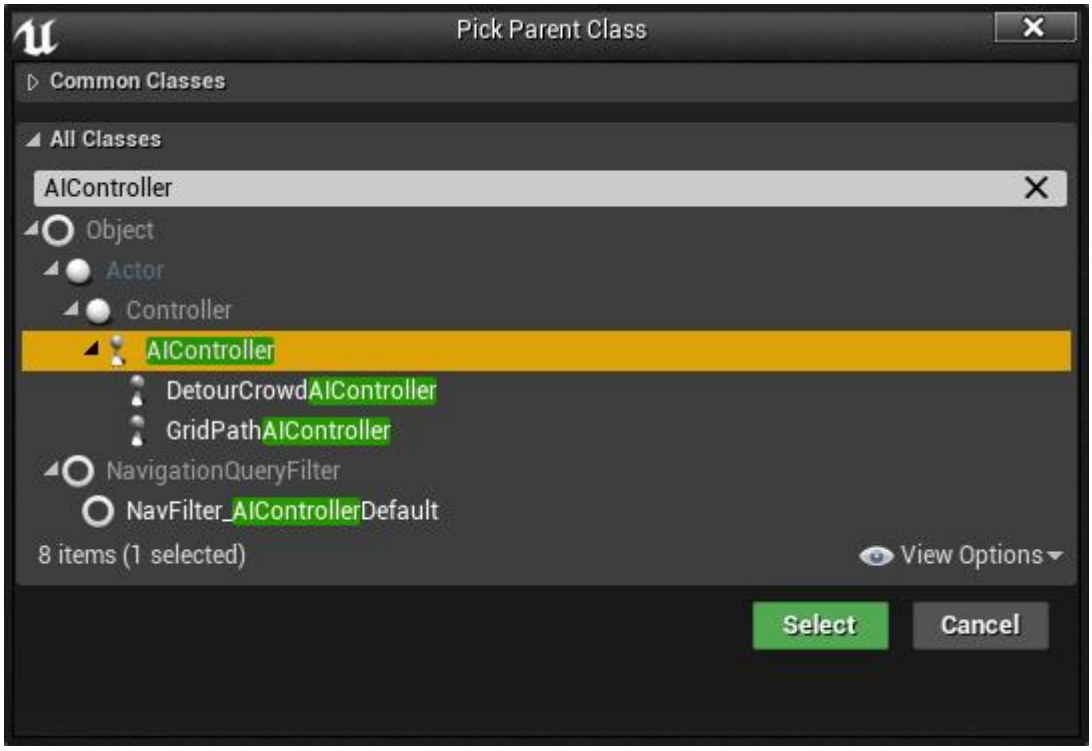


В случае ИИ Pawn может получать информацию от контроллера или мозга (в зависимости от того, как вы это реализуете).

Для управления маффинами с помощью ИИ нужно создать особый тип контроллера, называемый *контроллером ИИ (AI controller)*.

Создание AI Controller

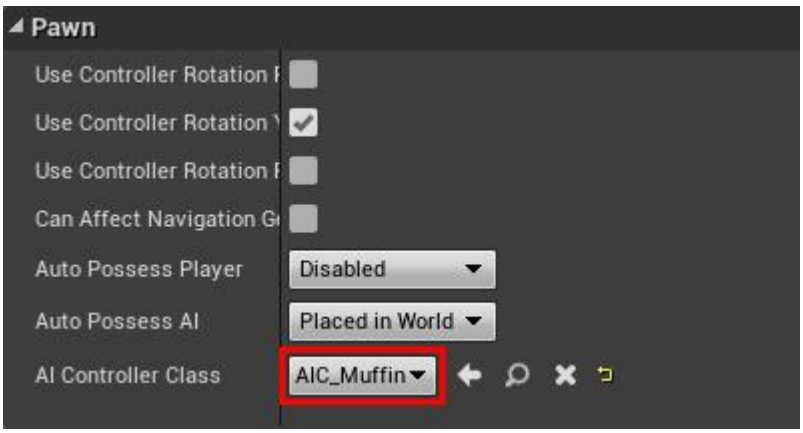
Перейдите к *Characters\Muffin\AI* и создайте новый *Blueprint Class*. Выберите в качестве родительского класса *AIController* и назовите его *AIC_Muffin*.



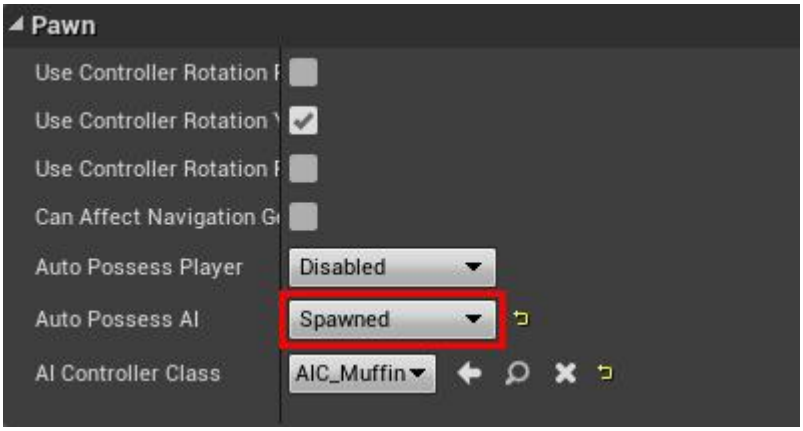
Затем нужно сказать маффину, чтобы он использовал новый контроллер ИИ. Перейдите в *Characters\Muffin\Blueprints* и откройте *BP_Muffin*.

По умолчанию в панели Details должны отображаться параметры Blueprint по умолчанию. Если это не так, то нажмите на *Class Defaults* в Toolbar.

Перейдите в панель Details и найдите раздел *Pawn*. Выберите для *AI Controller Class* значение *AIC_Muffin*. Благодаря этому будет создаваться экземпляр контроллера при создании маффина.



Поскольку мы создаём маффины динамически, нам также выбрать для *Auto Possess AI* значение *Spawned*. Так *AIC_Muffin* будет автоматически вселяться в *BP_Muffin* при создании.



Нажмите на *Compile* и закройте *BP_Muffin*.

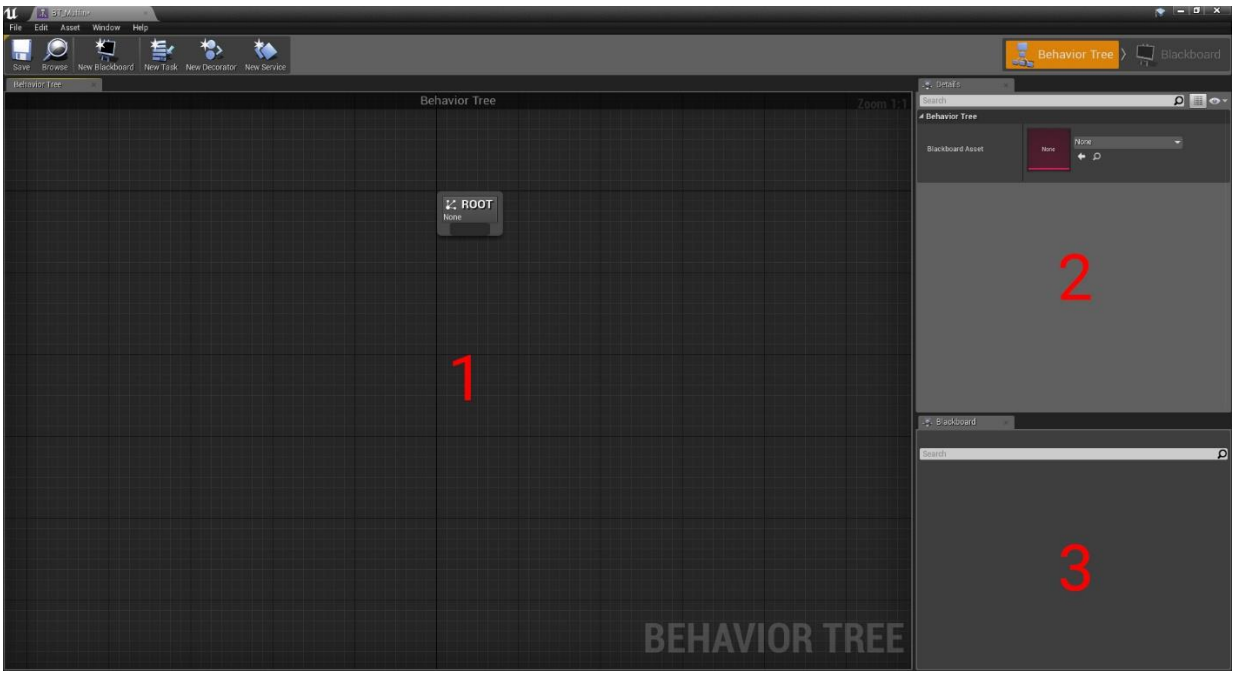
Теперь мы создадим логику, которая будет управлять поведением маффина. Для этого можно использовать *деревья поведения*.

Создание Behavior Tree

Перейдите в *Characters\Muffin\AI* и выберите *Add New\Artificial Intelligence\Behavior Tree*. Назовите его *BT_Muffin* и откройте.

Behavior Tree Editor

В редакторе деревьев поведения есть две новые панели:

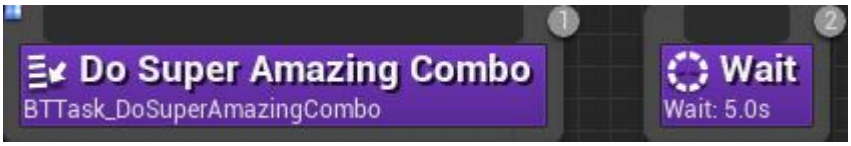


1. *Behavior Tree:* это граф, в котором мы будем создавать ноды для дерева поведения
2. *Details:* здесь отображаются свойства выбранных нодов
3. *Blackboard:* в этой панели показываются ключи Blackboard (подробнее о них позже) и их значения. Отображается только при запущенной игре.

Как и Blueprint, дерево поведения состоит из нодов. В деревьях поведения бывает четыре типа нодов. Первые два — это *task* и *composite*.

Что такое Task и Composite?

Как можно понять из названия, task (задача) — это узел, который что-то «делает». Это может быть что-то сложное, например, цепочка комбо, или что-то простое, например, ожидание.

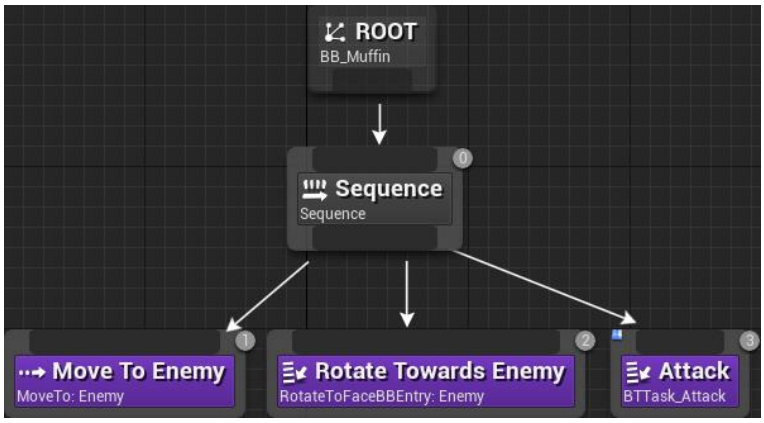


Для выполнения задач нужно использовать композиты (composite). Дерево поведения состоит из множества ветвей (поведений). В корне каждой ветви находится композит. Разные типы композитов имеют разные способы выполнения своих дочерних нодов.

Допустим, у нас есть следующая последовательность действий:

Для последовательного выполнения каждого действия нам нужно использовать композит

Sequence, потому что Sequence выполняет свои дочерние ноды слева направо. Вот, как это будет выглядеть:



Примечание: всё, что начинается с композита, можно назвать *поддеревом*. Обычно это поведения. В нашем примере *Sequence, Move To Enemy, Rotate Towards Enemy* и *Attack* можно считать поведением «атаковать врага».

Если любой из дочерних нодов Sequence *не удаётся выполнить*, то Sequence перестаёт выполняться.

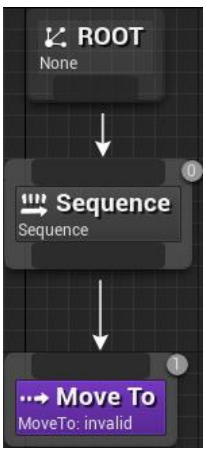
Например, если Pawn не может двигаться к врагу, то *Move To Enemy* выполнить не удастся. Это значит, что *Rotate Towards Enemy* и *Attack* не будут выполнены. Однако они выполнятся, если Pawn удастся двигаться к врагу.

Чуть позже мы также узнаем о композите *Selector*. Пока мы будем использовать *Sequence*, чтобы Pawn перемещался в случайную точку, а затем ждал.

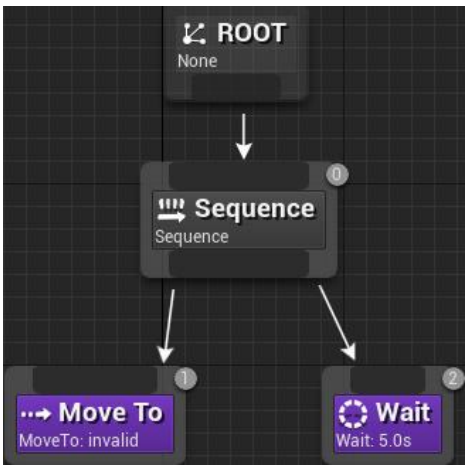
Движение в случайную точку

Создайте *Sequence* и соедините его с *Root*.

Теперь нам нужно двигать Pawn. Создайте *MoveTo* и соедините его с *Sequence*. Этот нод будет двигать Pawn к указанной точке или актору.



Затем создайте *Wait* и соедините его с *Sequence*. Необходимо расположить этот нод *справа* от *MoveTo*. Порядок здесь важен, потому что дочерние ноды выполняются слева направо.



Примечание: проверить порядок выполнения можно, посмотрев на числа в правом верхнем углу каждого нода. Чем меньше значения, тем выше приоритет нодов.

Поздравляю, вы только что создали своё первое поведение! Оно будет перемещать Pawn в выбранную точку, а затем ждать пять секунд.

Для движения Pawn нужно указать точку. Однако *MoveTo* может получать значения, полученные только от *blackboard*, так что давайте его создадим.

Создание Blackboard

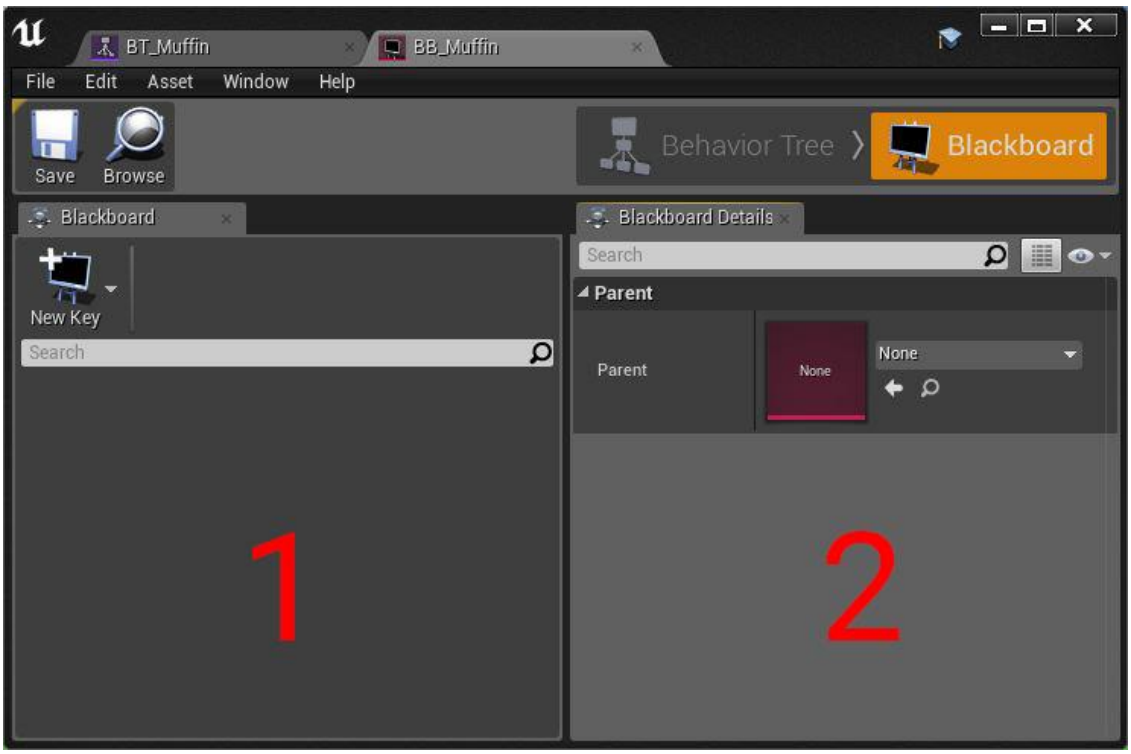
Blackboard — это ресурс, единственное предназначение которого заключается в хранении переменных (называемых *ключами (keys)*). Можно считать его памятью ИИ.

Хотя использовать их и не обязательно, blackboard обеспечивает удобный способ считывания и сохранения данных. Он удобен, потому что многие ноды в деревьях поведения могут получать только ключи blackboard.

Для создания blackboard вернитесь в Content Browser и выберите *Add New\Artificial Intelligence\Blackboard*. Назовите его *BB_Muffin* и откройте.

Blackboard Editor

Редактор blackboard состоит из двух панелей:

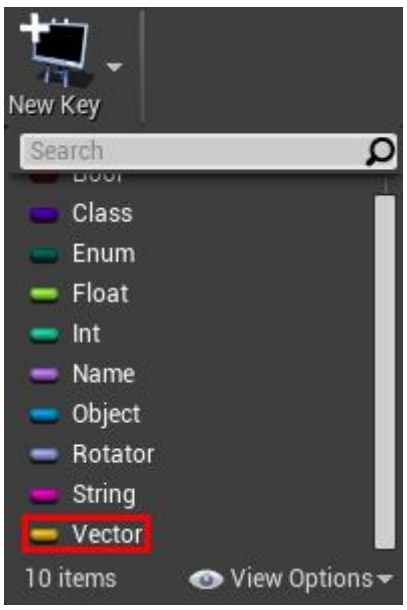


1. *Blackboard:* в этой панели отображается список ключей
2. *Blackboard Details:* в этой панели отображаются свойства выбранного ключа

Теперь нам нужно создать ключ, который будет содержать целевую точку.

Создание ключа целевой точки

Поскольку мы храним точки в 3D-пространстве, нам нужно хранить их как векторы. Нажмите на *New Key* и выберите *Vector*. Назовите его *TargetLocation*.

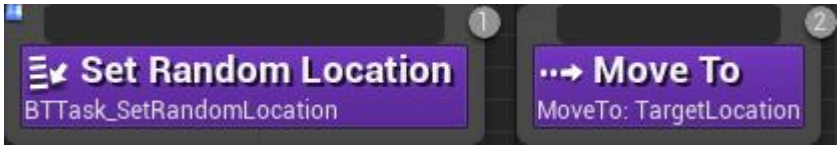


Теперь нам нужен способ генерирования случайной точки и сохранения её в blackboard. Для этого мы воспользуемся третьим типом нодов дерева поведения: *service*.

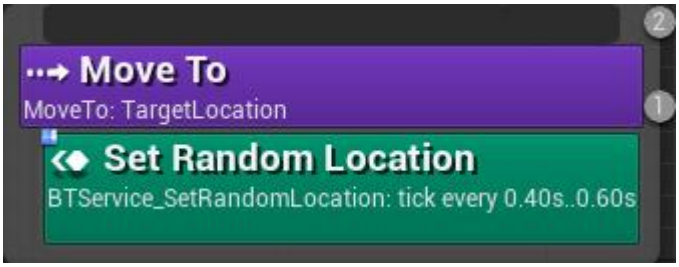
Что такое Service?

Services (службы) похожи на задачи (tasks), с помощью которых мы что-то делаем. Однако вместо того, чтобы заставлять Pawn выполнять действия, мы используем службы для выполнения проверок или обновления blackboard.

Службы — это не отдельные ноды, они присоединяются к задачам или композитам. Благодаря этому создаётся более упорядоченное дерево, потому что нам приходится работать с меньшим количеством нодов. Вот как выглядит использование задачи:



А вот как выглядит использование службы:



Теперь мы можем создать службу, генерирующую случайную точку.

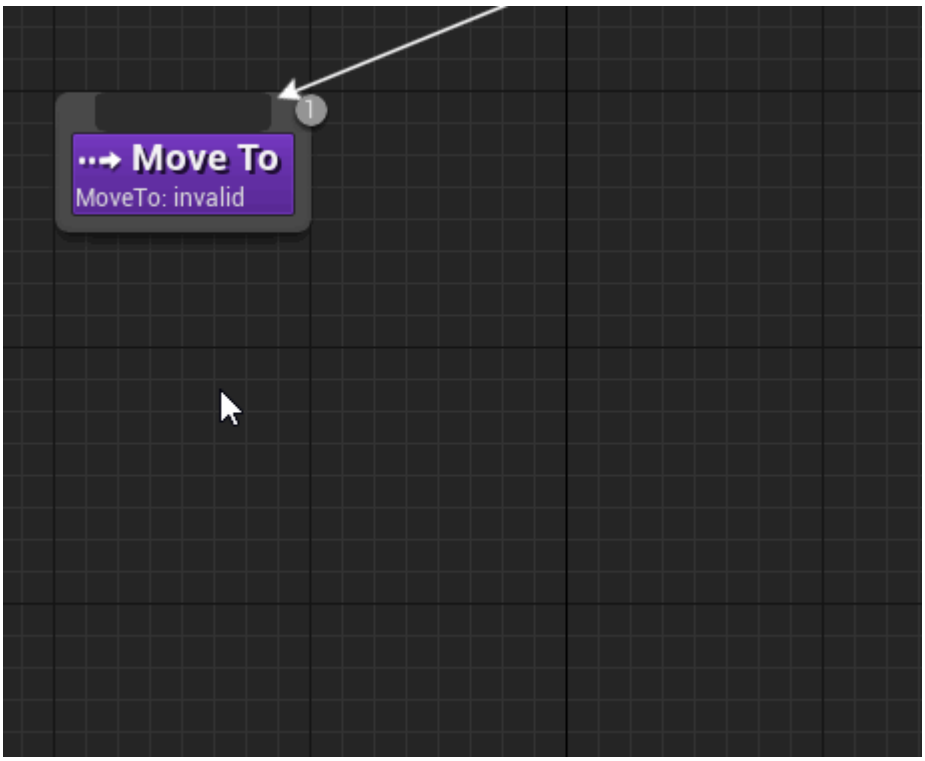
Создание службы

Вернитесь к *BT_Muffin* и нажмите на *New Service*.

При этом будет создана и автоматически открыта новая служба. Назовите её *BTService_SetRandomLocation*. Чтобы переименовать её, нужно будет вернуться в Content Browser.

Служба должна выполняться только тогда, когда Pawn нужно двигаться. Для этого нужно присоединить её к *MoveTo*.

Откройте *BT_Muffin* и *нажмите правой клавишей мыши* на *MoveTo*. Выберите *Add Service\BTService Set Random Location*.



Теперь *BTService_SetRandomLocation* будет активироваться при активации *MoveTo*.

Далее нам нужно генерировать случайную целевую точку.

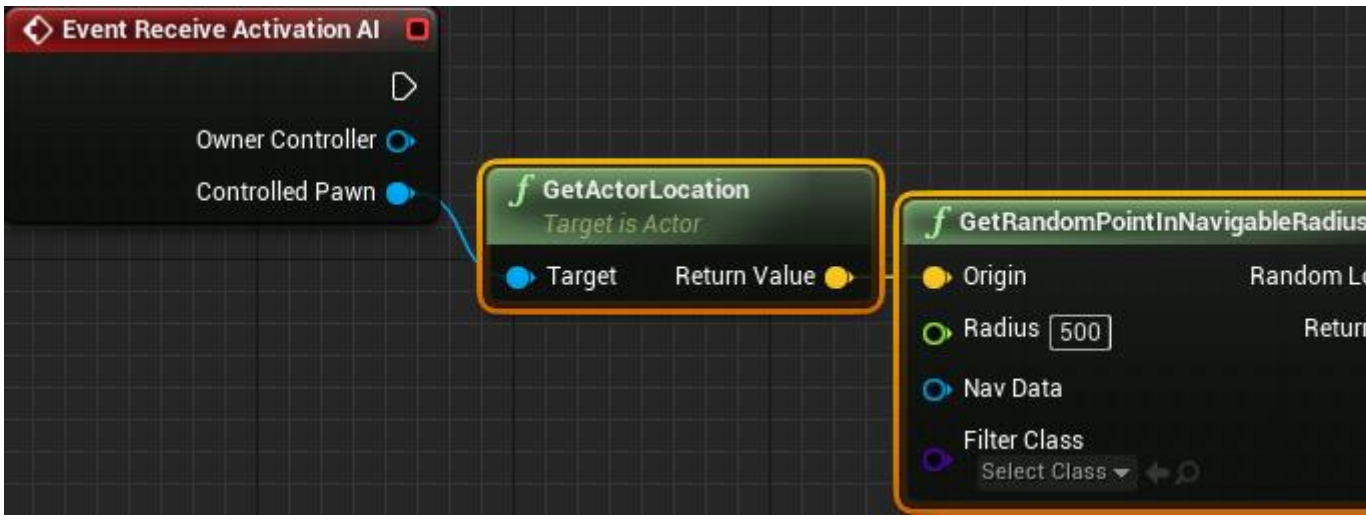
Генерирование случайной точки

Откройте Open *BTService_SetRandomLocation*.

Чтобы узнать, когда активируется служба, создадим нод *Event Receive Activation AI*. Он будет выполняться, когда активируется родитель (нод, к которому он прикреплён).

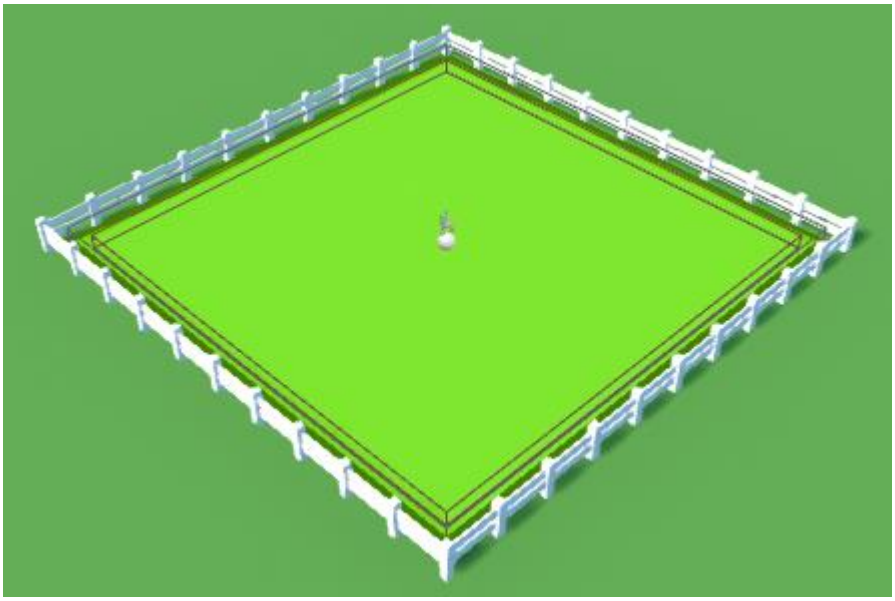
Примечание: существует также событие *Event Receive Activation*, делающее то же самое. Разница между двумя событиями в том, что *Event Receive Activation AI* также предоставляет *Controlled Pawn*.

Для генерирования случайной точки добавим выделенные ноды. Параметру *Radius* присвоим значение *500*.



Это даст нам случайную точку в пределах *500* единиц от Pawn, в которую можно дойти.

Примечание: для определения того, можно ли дойти до точки, *GetRandomPointInNavigableRadius* использует навигационные данные (которые называются *NavMesh*). В этом туториале я заранее создал NavMesh. Его можно визуализировать, перейдя во Viewport и выбрав *Show\Navigation*.



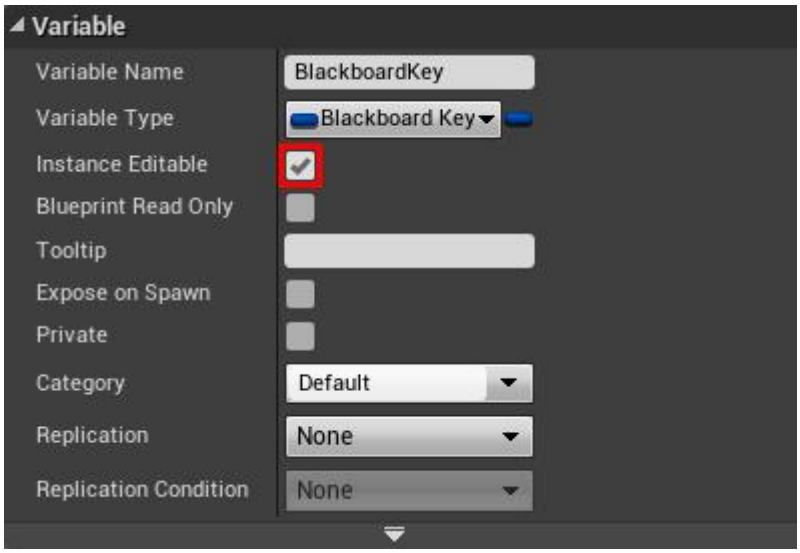
Если вы хотите создать собственный NavMesh, то создайте *Nav Mesh Bounds Volume*. Измените его масштаб так, чтобы он ограничивал область, которая должна быть доступна для движения.

Теперь нам нужно сохранять точку в blackboard. Существует два способа выбора используемого ключа:

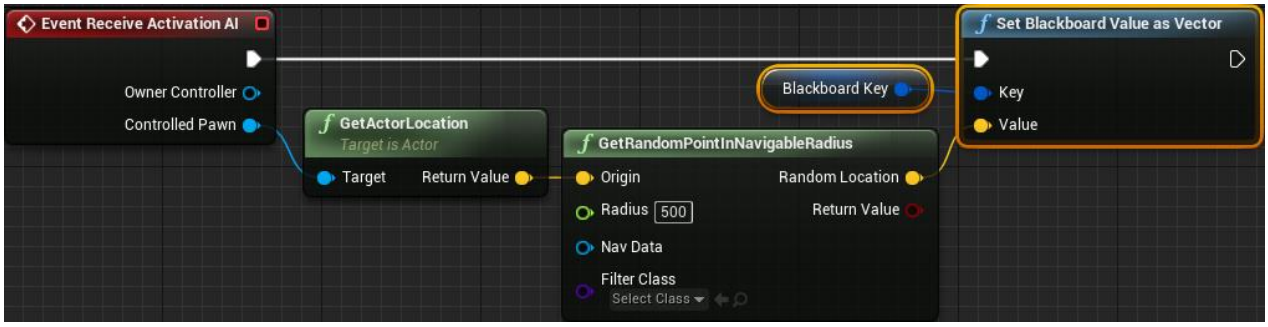
1. Можно указать ключ, использовав его имя в ноде *Make Literal Name*
2. Можно сделать переменную видимой для дерева поведения. Это позволит выбирать ключ из раскрывающегося списка.

Мы воспользуемся вторым способом. Создайте переменную типа *Blackboard Key Selector*.

Назовите её *BlackboardKey* и включите *Instance Editable*. Это позволит сделать переменную видимой при выборе службы в дереве поведения.



Далее создадим следующие выделенные ноды:



Подведём итог:

1. *Event Receive Activation AI* выполняется, когда активируется его родитель (в нашем случае это *MoveTo*)
2. *GetRandomPointInNavigableRadius* возвращает случайную доступную для навигации точку в радиусе *500* единиц от управляемого маффина
3. *Set Blackboard Value as Vector* задаёт ключу blackboard (передаваемому через *BlackboardKey*) значение случайной точки

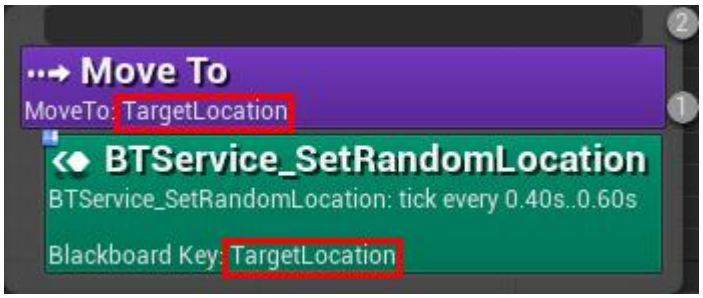
Нажмите на *Compile* и закройте *BTService_SetRandomLocation*.

Теперь мы хотим сообщить дереву поведения, что нужно использовать наш blackboard.

Выбор Blackboard

Откройте *BT_Muffin* и убедитесь, что ничего не выбрано. Перейдите в панель Details. В *Behavior Tree* задайте для *Blackboard Asset* значение *BB_Muffin*.

После этого *MoveTo* и *BTService_SetRandomLocation* будут автоматически использовать первый ключ blackboard. В нашем случае это *TargetLocation*.



Наконец, нам нужно приказать контроллеру ИИ запустить дерево поведения.

Выполнение дерева поведения

Откройте *AIC_Muffin* и соедините *Run Behavior Tree* с *Event BeginPlay*. Выберите для *BTAsset* значение *BT_Muffin*.

Так *BT_Muffin* будет запускаться при создании *AIC_Controller*.

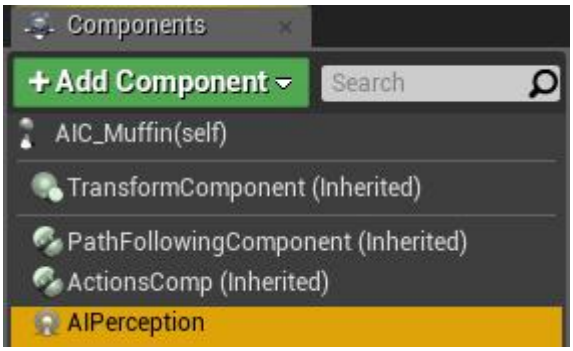
Нажмите на *Compile* и вернитесь в основной редактор. Нажмите на *Play*, создайте несколько маффинов и посмотрите, как они бродят по экрану.

Нам пришлось потрудиться, но мы справились! Теперь мы должны настроить контроллер ИИ таким образом, чтобы он распознавал врагов в области его видимости. Для этого можно воспользоваться *AI Perception*.

Настройка AI Perception

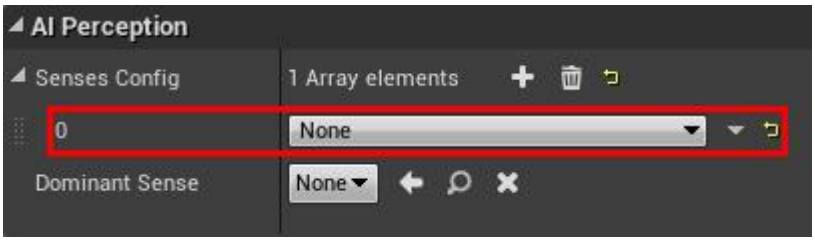
AI Perception — это компонент, который можно добавлять к акторам. С его помощью можно давать ИИ *чувства* (такие как зрение и слух).

Откройте *AIC_Muffin* и добавьте компонент *AIPerception*.

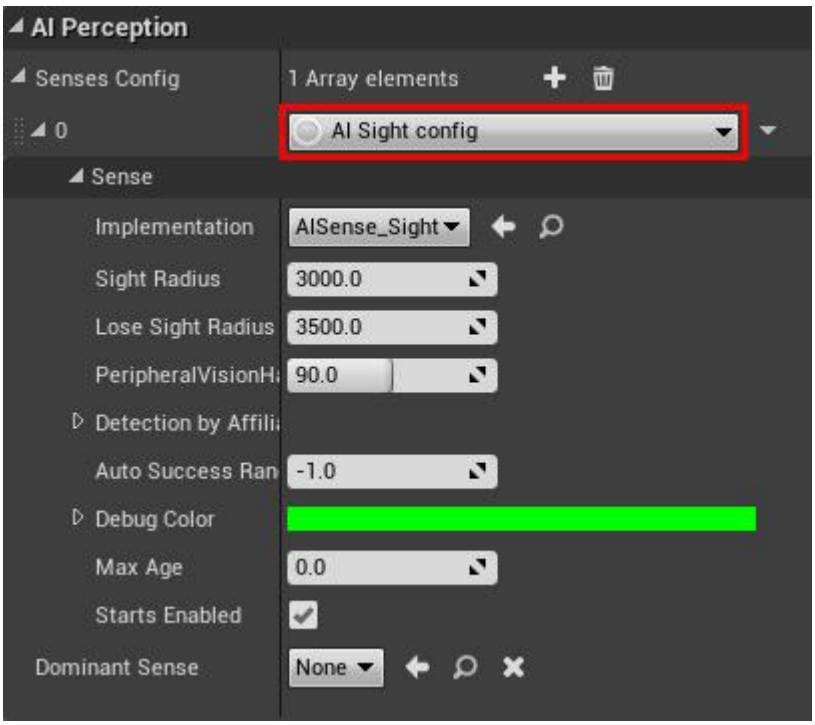


Теперь нам нужно добавить чувство. Мы хотим, чтобы маффин распознавал другого маффина, попадающего в область видимости, так что нужно добавить *зрение*.

Выберите *AIPerception* и перейдите в панель Details. В разделе *AI Perception* добавьте к *Senses Config* новый элемент.



Задайте элемент *0* для *AI Sight config* и разверните его.



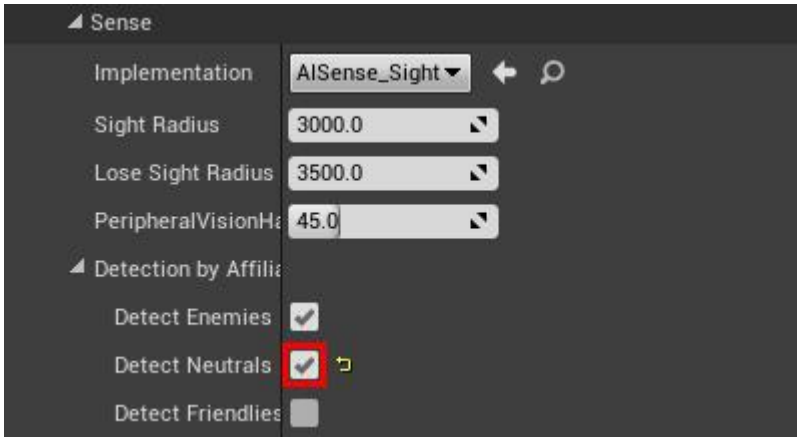
У зрения есть три основных параметра:

1. *Sight Radius:* максимальное расстояние, на которое может видеть маффин. Оставим здесь значение *3000*.

2. *Lose Sight Radius:* если маффин увидел врага, это значение, на которое враг должен отдалиться, чтобы маффин потерял его из виду. Оставим здесь значение *3500*.
3. *Peripheral Vision Half Angle Degrees:* угол обзора маффина. Задайте значение *45*. Это даст маффину угол обзора в *90* градусов.

По умолчанию AI Perception распознаёт только врагов (акторов, назначенных в другую *команду*). Однако по умолчанию у акторов нет команды. Если актор не имеет команды, то AI Perception считает его *нейтральным*.

На момент написания статьи не существовало способа назначения команд с помощью Blueprints. Вместо этого можно просто приказать AI Perpcetion распознавать нейтральных акторов. Для этого разверните *Detection by Affiliation* и включите *Detect Neutrals*.

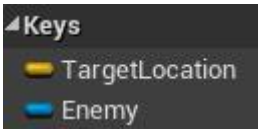


Нажмите на *Compile* и вернитесь в основной редактор. Нажмите на *Play* и создайте несколько маффинов. Нажмите на клавишу ‘ для отображения экрана отладки ИИ. Нажмите клавишу *4* на *цифровом блоке*, чтобы визуализировать AI Perception. Когда маффин попадает в поле зрения, появляется зелёная сфера.

Теперь нам нужно двигать маффина в сторону врага. Для этого дерево поведения должно *знать* о враге. Это можно реализовать сохранением ссылки на врага в blackboard.

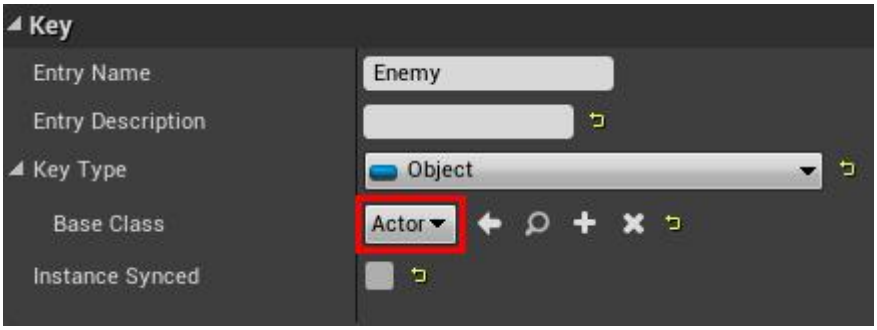
Создание ключа врага

Откройте *BB_Muffin* и добавьте ключ типа *Object*. Переименуйте его в *Enemy*.



Пока мы не можем использовать *Enemy* в *MoveTo*, потому что ключ имеет тип *Object*, но *MoveTo* может получать ключи только типа *Vector* или *Actor*.

Чтобы исправить это, выберите *Enemy* и разверните *Key Type*. Выберите для *Base Class* значение *Actor*. Это позволит дереву поведения распознать *Enemy* как *Actor*.

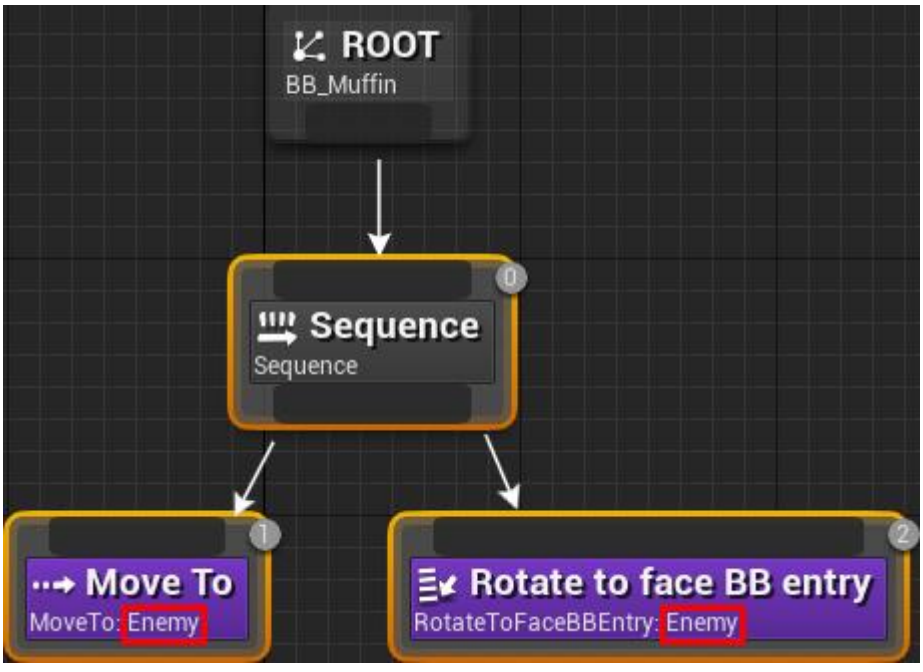


Закройте *BB_Muffin*. Теперь нам нужно создать поведение для движения в сторону врага.

Движение в сторону врага

Откройте *BT_Muffin* и отделите *Sequence* от *Root*. Это можно сделать, *зажав Alt и нажав левой клавишей* на *провод*, соединяющий их. Оставим пока поддерево случайного движения в покое.

Теперь создайте выделенные ноды и задайте для их *Blackboard Key* значение *Enemy*:



При этом Pawn будет двигаться к *Enemy*. В некоторых случаях Pawn не полностью поворачивается к цели, так что нужно использовать ещё и *Rotate to face BB entry*.

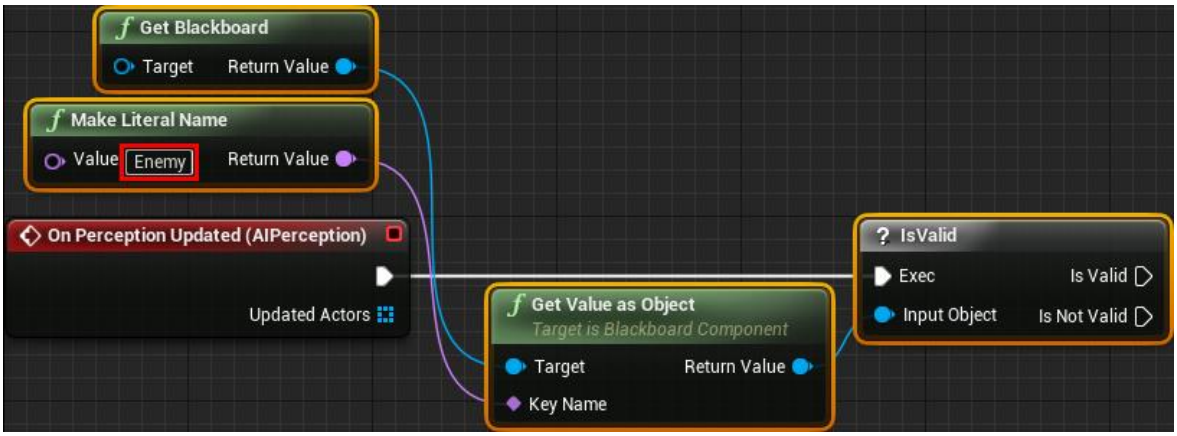
Теперь нужно задавать *Enemy*, когда AI Perception распознаёт другой маффин.

Задание ключа Enemy

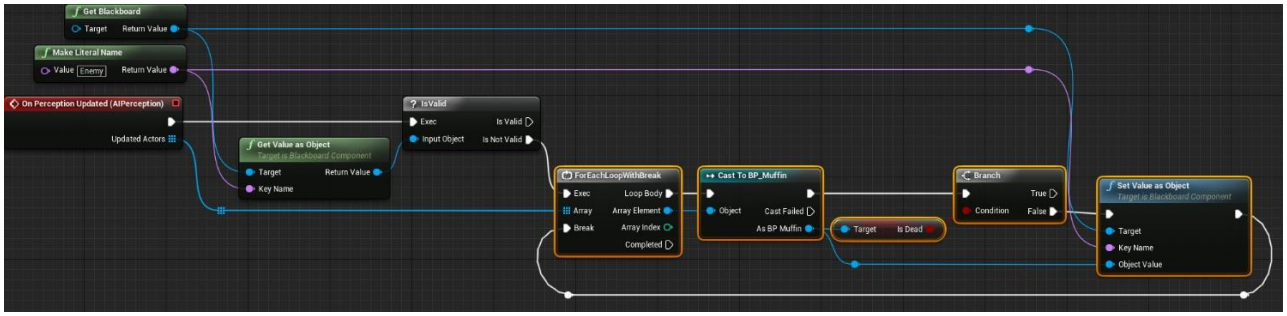
Откройте *AIC_Muffin* и выберите компонент *AIPerception*. Добавьте событие *On Perception Updated*.

Это событие выполняется при обновлении чувства. В нашем случае оно выполняется, когда ИИ кого-нибудь замечает или теряет из виду. Также это событие передаёт список акторов, которые в данный момент обнаружены чувствами.

Добавьте выделенные ноды. Убедитесь, что для *Make Literal Name* задано значение *Enemy*.



Это позволит проверять, есть ли у ИИ уже враг. Если нет, то мы должны дать ему врага. Для этого нужно добавить выделенные ноды:



Подведём итог:

1. *IsValid* проверяет, задан ли ключ *Enemy*
2. Если он не задан, то циклически обходятся все обнаруженные на данный момент акторы
3. *Cast To BP_Muffin* проверяет, является ли актор маффином
4. Если он маффин, то проверяем, мёртв ли он
5. Если *IsDead* возвращает *false*, то задаём маффина как нового *Enemy* и прекращаем цикл

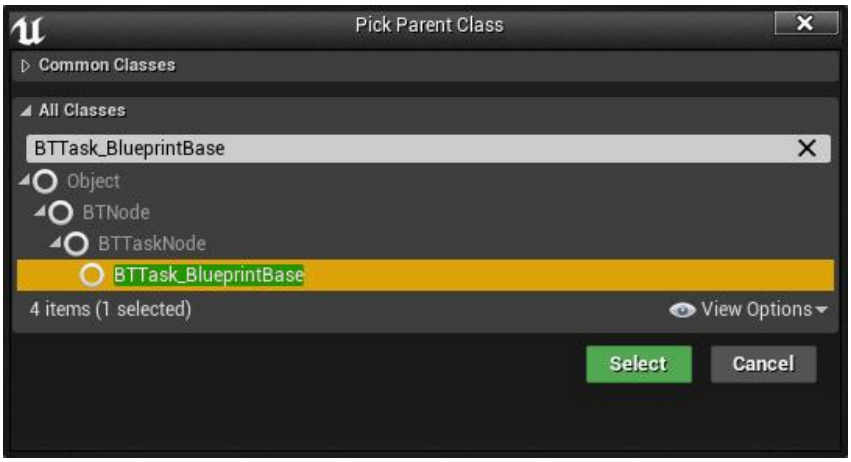
Нажмите на *Compile* и закройте *AIC_Muffin*. Нажмите на *Play* и создайте два маффина так,

чтобы один находился перед другим. Маффин, расположенный сзади, автоматически начнёт двигаться к другому.

Теперь мы создадим собственную задачу, чтобы маффин выполнял атаку.

Создание задачи атаки

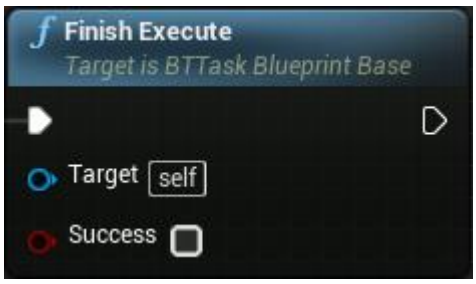
Мы можем создать задачу в Content Browser вместо редактора деревьев поведения. Создайте новый *Blueprint Class* и выберите в качестве родительского класса *BTTask_BlueprintBase*.



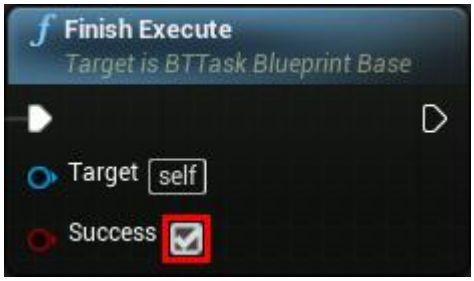
Назовите его *BTTask_Attack* и откройте. Добавьте нод *Event Receive Execute AI*. Этот нод будет выполняться, когда дерево поведения выполняет *BTTask_Attack*.

Сначала нам нужно заставить маффин атаковать. *BP_Muffin* содержит переменную *IsAttacking*. Если она задана, то маффин выполнит атаку. Для этого нужно добавить выделенные ноды:

Если использовать задачу в её нынешнем состоянии, то выполнение остановится на ней, потому что дерево поведения не знает, завершилась ли задача. Чтобы исправить это, добавим к концу цепочки *Finish Execute*.



Затем включите *Success*. Мы используем *Sequence*, поэтому это позволит выполниться нодам после *BTTask_Attack*.



Вот как должен выглядеть граф:

Подведём итог:

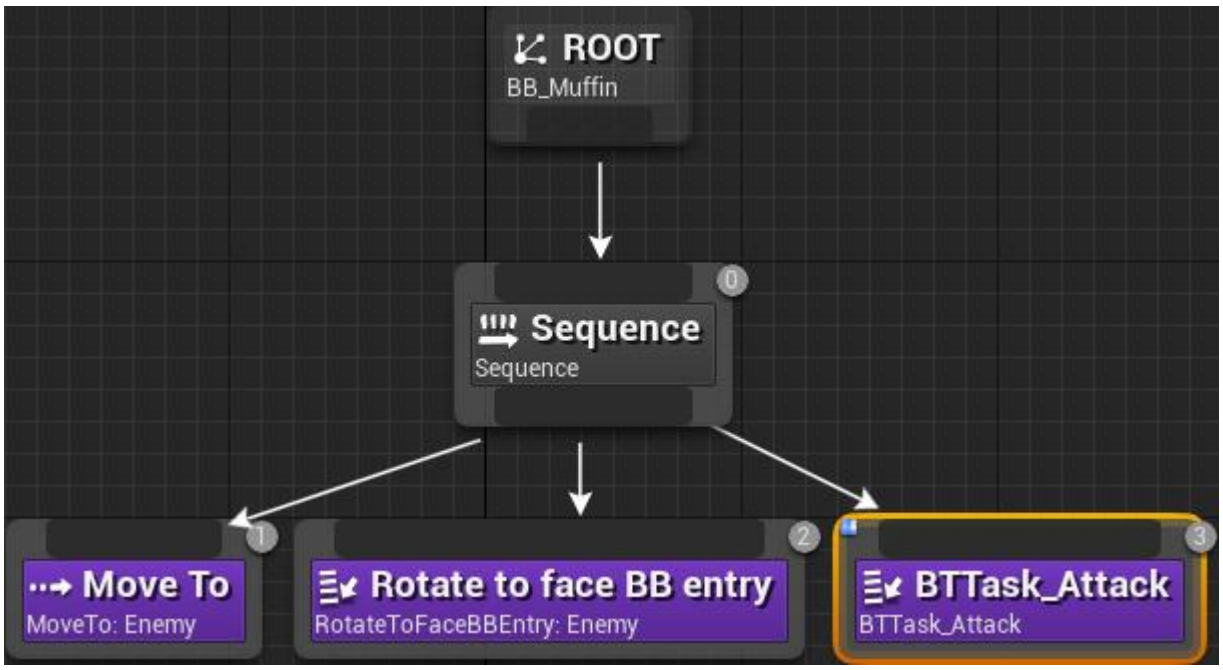
1. *Event Receive Execute AI* выполняется, когда дерево поведения запускает *BTTask_Attack*
2. *Cast To BP_Muffin* проверяет, имеет ли *Controlled Pawn* тип *BP_Muffin*
3. Если да, то задаётся его переменная *IsAttacking*
4. *Finish Execute* даёт дереву поведения понять, что задача *успешно* выполнена

Нажмите на *Compile* и закройте *BTTask_Attack*.

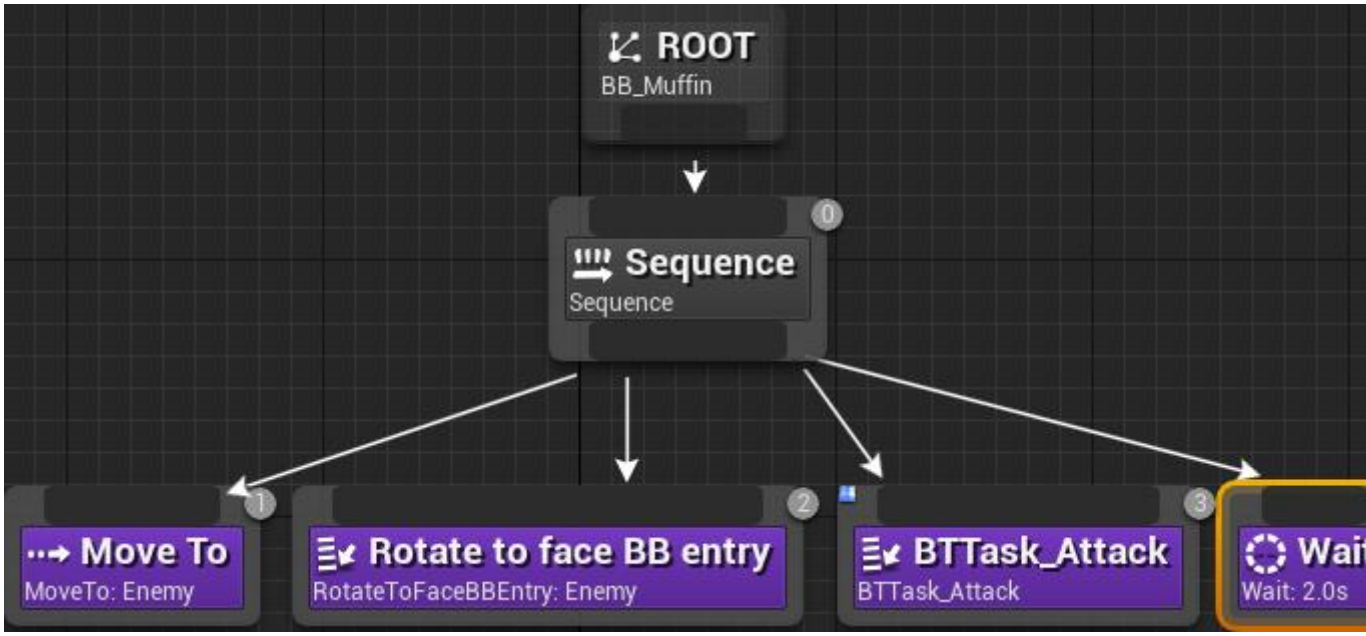
Теперь нам нужно добавить в дерево поведения *BTTask_Attack*.

Добавление атаки в дерево поведения

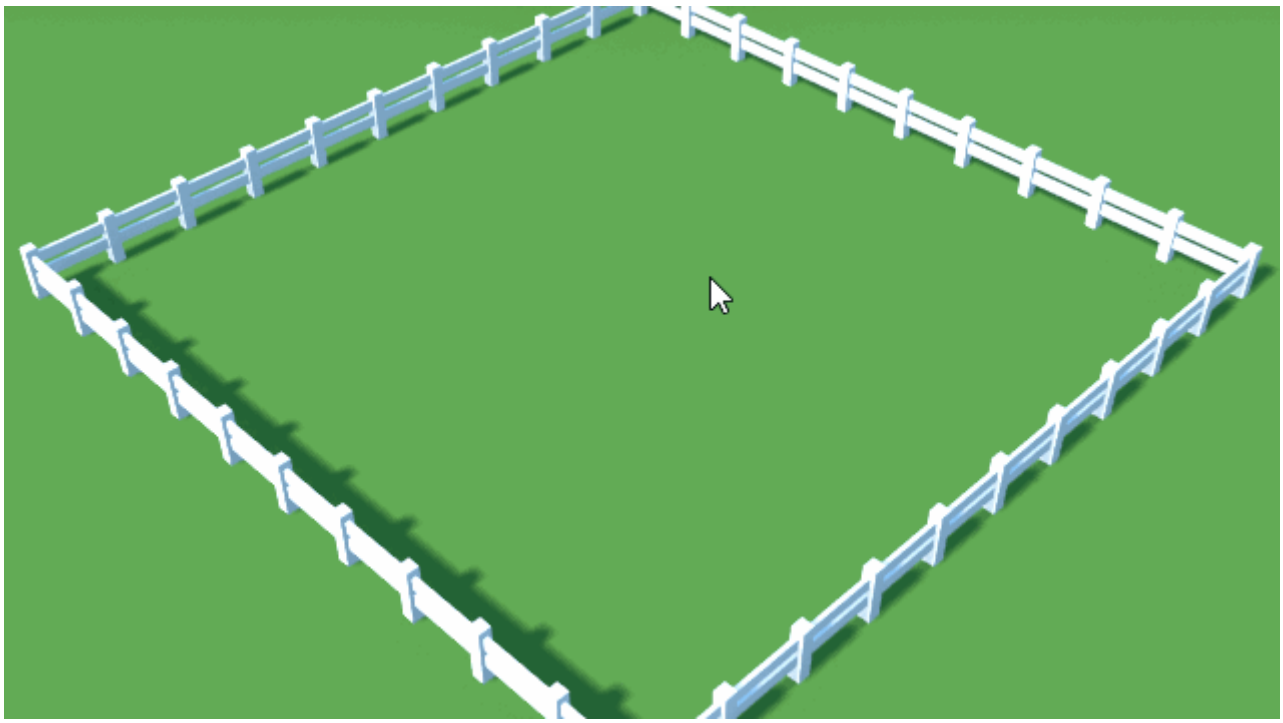
Откройте *BT_Muffin*. Затем добавьте в конец *Sequence* нод *BTTask_Attack*.



Далее добавьте в конец *Sequence* нод *Wait*. Измените значение его переменной *Wait Time* на *2*. Благодаря этому маффин не будет атаковать постоянно.



Вернитесь в основной редактор и нажмите на *Play*. Как и в прошлый раз, создайте два маффина. Маффин начнёт двигаться и поворачиваться к врагу. Затем он атакует и подождёт две секунды. Если он увидит ещё одного врага, то снова повторит ту же последовательность.

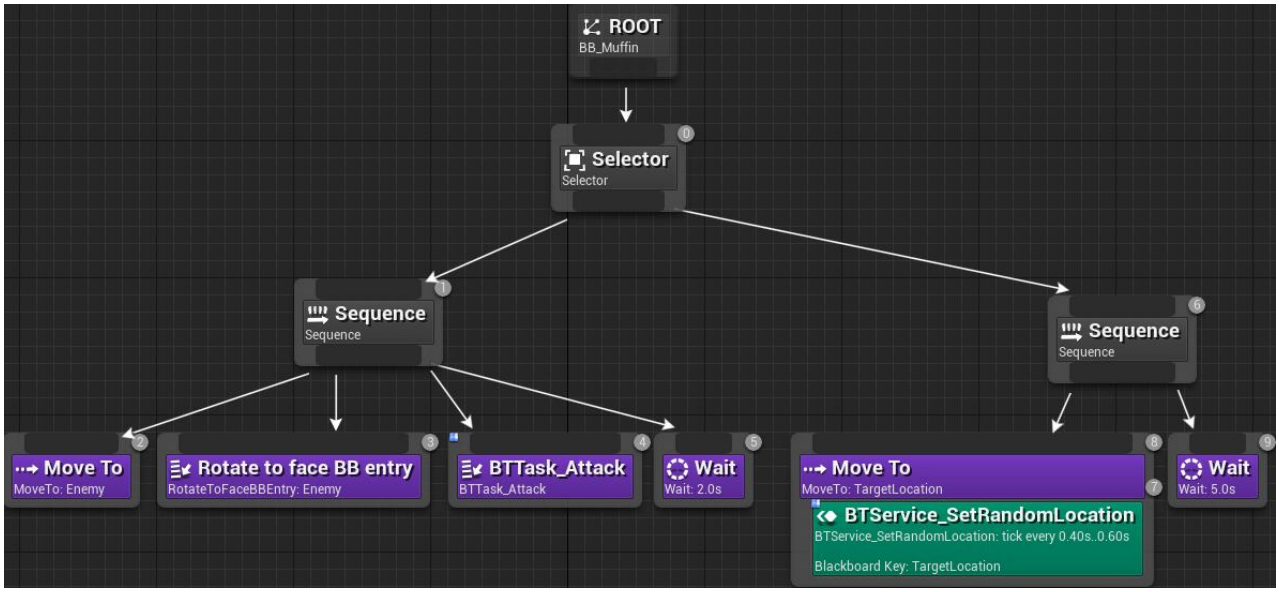


В последней части мы объединим поддеревья атаки и случайного движения.

Объединение поддеревьев

Чтобы объединить поддеревья, можно использовать композит *Selector*. Как и Sequence, он тоже выполняется слева направо. Однако Selector останавливается тогда, когда дочерний нод *успешно выполняется*, а не завершается неудачно. С помощью этого поведения мы можем сделать так, что дерево поведения выполнит только одно поддерево.

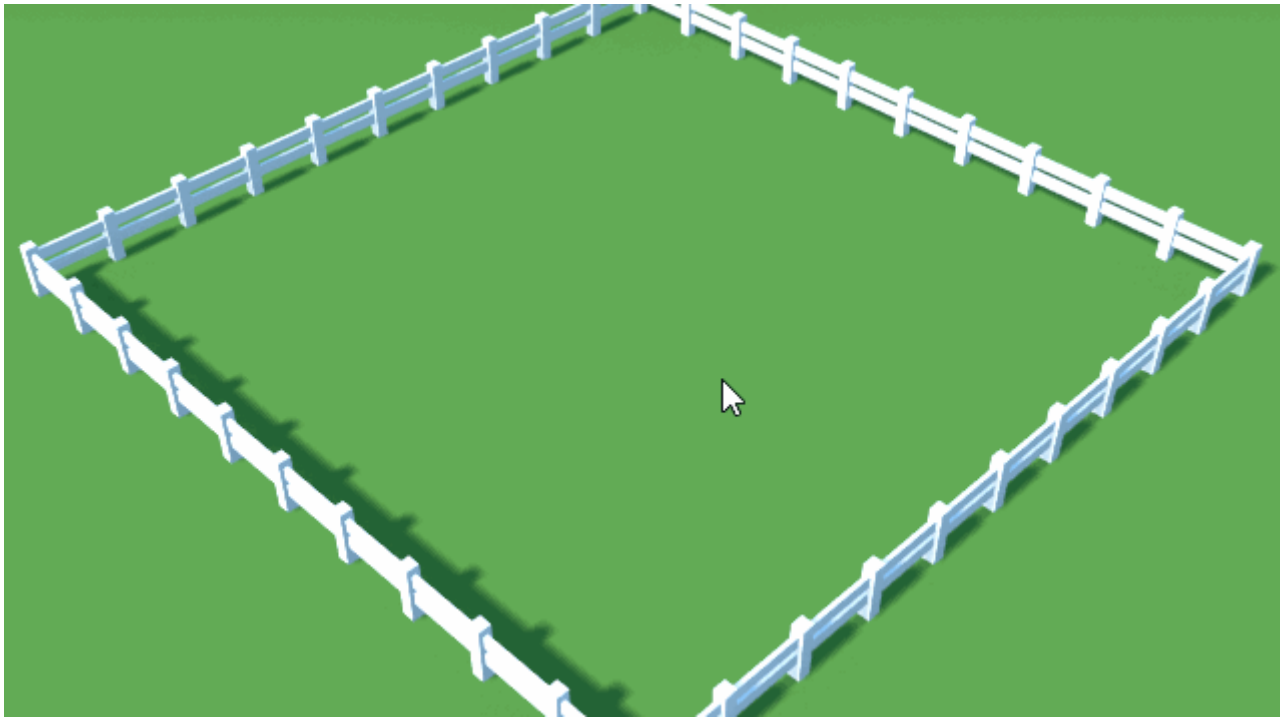
Откройте *BT_Muffin* и создайте *Selector* после нода *Root*. Затем соедините поддеревья следующим образом:



В такой схеме одновременно будет выполняться только одно поддерево. Вот как запускается каждое поддерево:

- *Attack: Selector* запустит первым поддерево атаки. Если все задачи выполнены успешно, то *Sequence* тоже завершится успешно. *Selector* обнаружит это и остановит выполнение. Благодаря этому поддерево случайного движения выполняться не будет.
- *Roam:* селектор попытается сначала выполнить поддерево атаки. Если *Enemy* не задан, то *MoveTo* завершится неудачей. Поэтому *Sequence* тоже завершится неудачей. Поскольку поддерево атаки завершилось неудачно, *Selector* выполнит следующий дочерний элемент, то есть поддерево случайного движения.

Вернитесь в основной редактор и нажмите на *Play*. Создайте несколько маффинов, чтобы протестировать работу.



Постойте, но почему маффин сразу же не атакует другой маффин?

В традиционных деревьях поведения выполнение начинается при каждом обновлении с корня. Это значит, что при каждом обновлении дерево сначала пробует выполнить поддерево атаки, а затем поддерево случайного движения. Это значит, что дерево поведения может мгновенно менять поддеревья при изменении значения *Enemy*.

Однако деревья поведения Unreal работают иначе. В Unreal выполнение продолжается с последнего выполненного нода. Так как AI Perception не чувствует сразу же других акторов, начинает выполняться поддерево случайного движения. Теперь дерево поведения должно дождаться завершения дерева случайного движения, и только потом проверить возможность выполнения поддерева атаки.

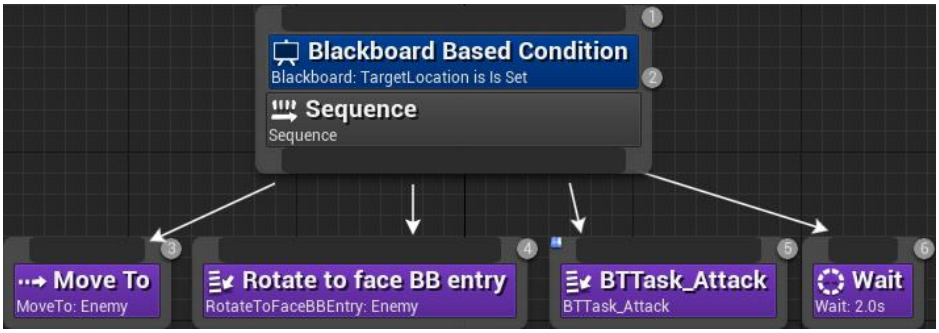
Чтобы исправить это, мы можем использовать последний тип нодов: *декораторы (decorators)*.

Создание декоратора

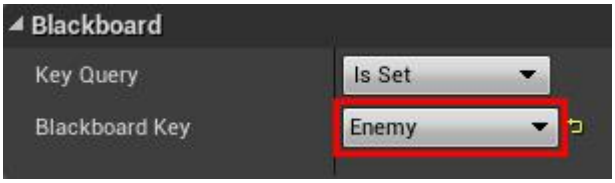
Как и службы (services), декораторы присоединяются к задачам или композитам. Обычно декораторы используются для выполнения проверок. Если результат равен true, то декоратор тоже возвращает true, и наоборот. Благодаря декораторам мы можем управлять выполнением их родительских элементов.

Кроме того, декораторы обладают возможностью *прекращения* поддерева. Это значит, что можно завершить выполнение поддерева случайного движения, если задан *Enemy*. Таким образом маффин сможет атаковать врага сразу же после его обнаружения.

Чтобы использовать прекращение выполнения, мы можем применять декоратор *Blackboard*. Он просто проверяет, задан ли ключ blackboard. Откройте *BT_Muffin* и *нажмите правой клавишей мыши* на *Sequence* поддерева атаки. Выберите *Add Decorator\Blackboard*. При этом к Sequence добавится декоратор Blackboard.



Теперь выберите декоратор *Blackboard* и перейдите в панель Details. Задайте *Blackboard Key* значение *Enemy*.



Так мы будем проверять, задан ли *Enemy*. Если он не задан, то декоратор завершится неудачно и приведёт к неудачному завершению *Sequence*. Это позволит запустить поддерево случайного движения.

Чтобы прекратить выполнение поддерева случайного движения, нам нужно будет использовать параметр *Observer Aborts*.

Использование Observer Aborts

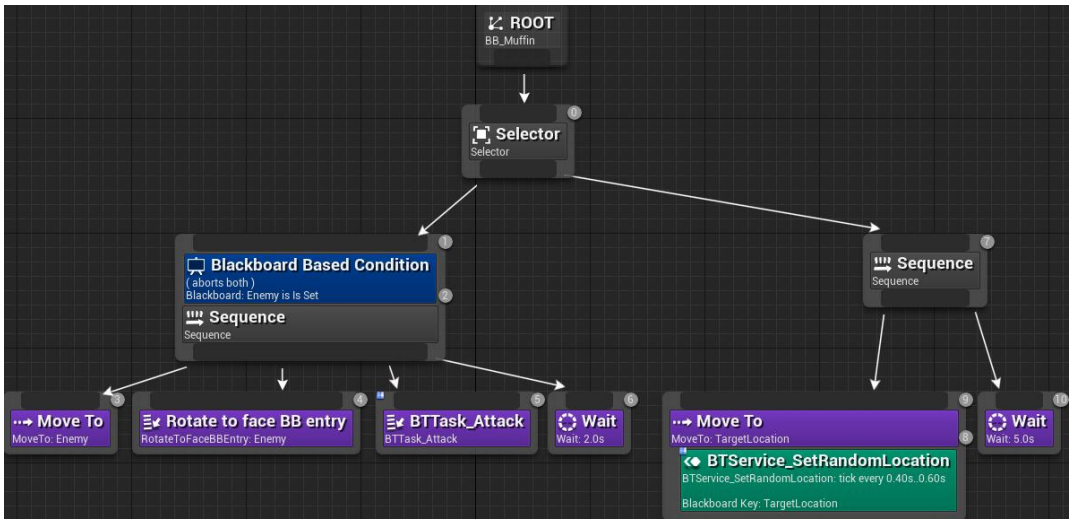
Observer aborts прекращает выполнение поддерева при изменении выбранного ключа blackboard. Существует два типа прекращения:

1. *Self:* этот параметр позволяет поддереву атаки прекратить собственное выполнение, когда *Enemy* становится недействительным. Это может случиться, когда *Enemy* умирает до завершения поддерева атаки.
2. *Lower Priority:* этот параметр позволяет прекращать выполнение деревьев с более низким приоритетом при задании *Enemy*. Поскольку поддерево случайного движения расположено после атаки, оно имеет меньший приоритет.

Выберите для *Observer Aborts* значение *Both*, что включит оба типа прекращения.

Теперь поддерево атаки будет сразу же переходить к случайному движению при отсутствии врага. А поддерево случайного движения немедленно будет переключаться в режим атаки при обнаружении врага.

Вот как выглядит готовое дерево поведения:



Подведём итог поддерева атаки:

1. *Selector* запускает поддерево атаки, если задан *Enemy*
2. Если он задан, то Pawn будет двигаться и поворачиваться в сторону врага
3. Затем он выполнит атаку
4. Наконец, Pawn будет ждать две секунды

Подводим итог поддерева случайного движения:

1. *Selector* запускает поддерево случайного движения, если поддерево атаки завершается неудачно. В нашем случае оно завершается неудачно, если не задан *Enemy*.
2. *BTService_SetRandomLocation* генерирует случайную точку
3. Pawn движется в сгенерированную точку
4. Затем он ждёт пять секунд

Закройте *BT_Muffin* и нажмите на *Play*. Создайте несколько маффинов и приготовьтесь к величайшей «Королевской битве»!