Условия и интерфейсы игры

Владислав Панченко
Unreal Engine разработчик в Sperasoft

Проверка связи

Если у вас нет звука:

- убедитесь, что на вашем устройстве и на колонках включён звук
- обновите страницу вебинара (или закройте страницу и заново присоединитесь к вебинару)
- откройте вебинар в другом браузере
- перезагрузите компьютер (ноутбук) и заново попытайтесь зайти

Поставьте в чат:

\+ если меня видно и слышно

— если нет

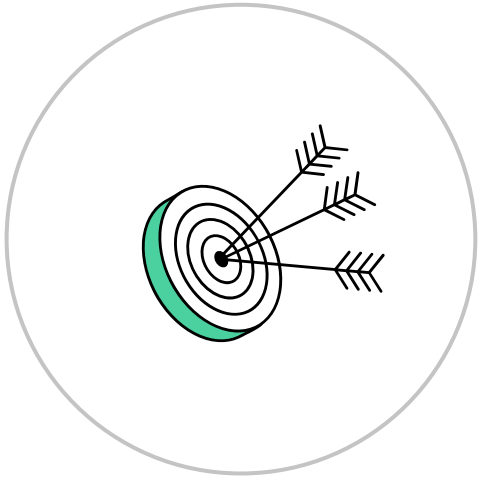
Владислав Панченко

О спикере:

- Unreal Engine разработчик в Sperasoft
- Работает разработчиком с 2019 года
- Опыт разработки на C++ более 7 лет

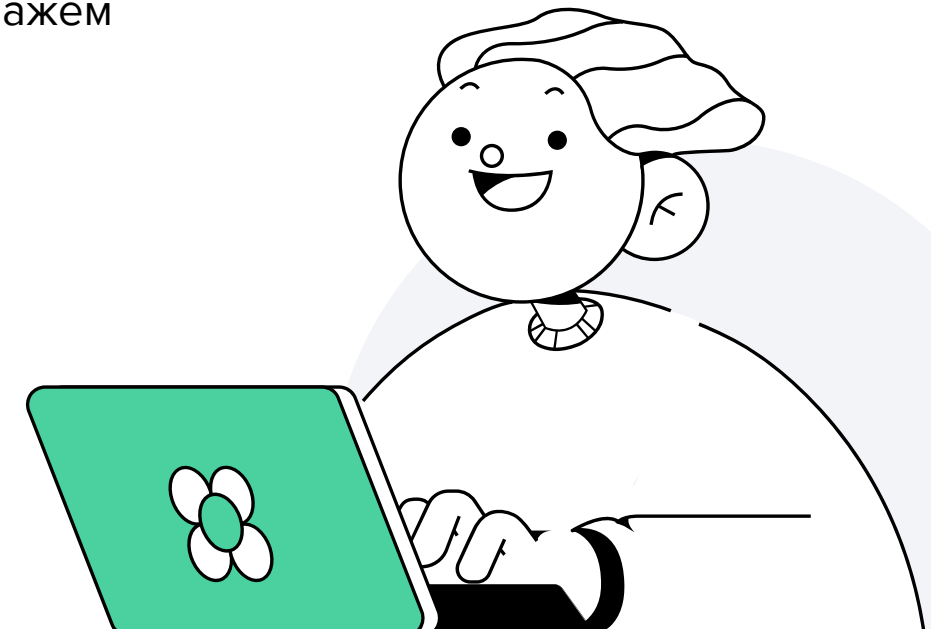
Цели занятия

- Научить неигрового персонажа наносить урон игроку
- Реализовать механику нанесения урона неигровому персонажу игроком
- Добавить механику спауна и следования для НПС
- Задать условия игры
- Разработать виджет отображения очков



План занятия

1. Нанесение урона противнику
2. Смерть НПС
3. Нанесение урона неигровым персонажем
4. Spawn Enemy
5. Игровые условия
6. Подсчёт очков



*Нажми на нужный раздел для перехода

Нанесение урона противнику

1

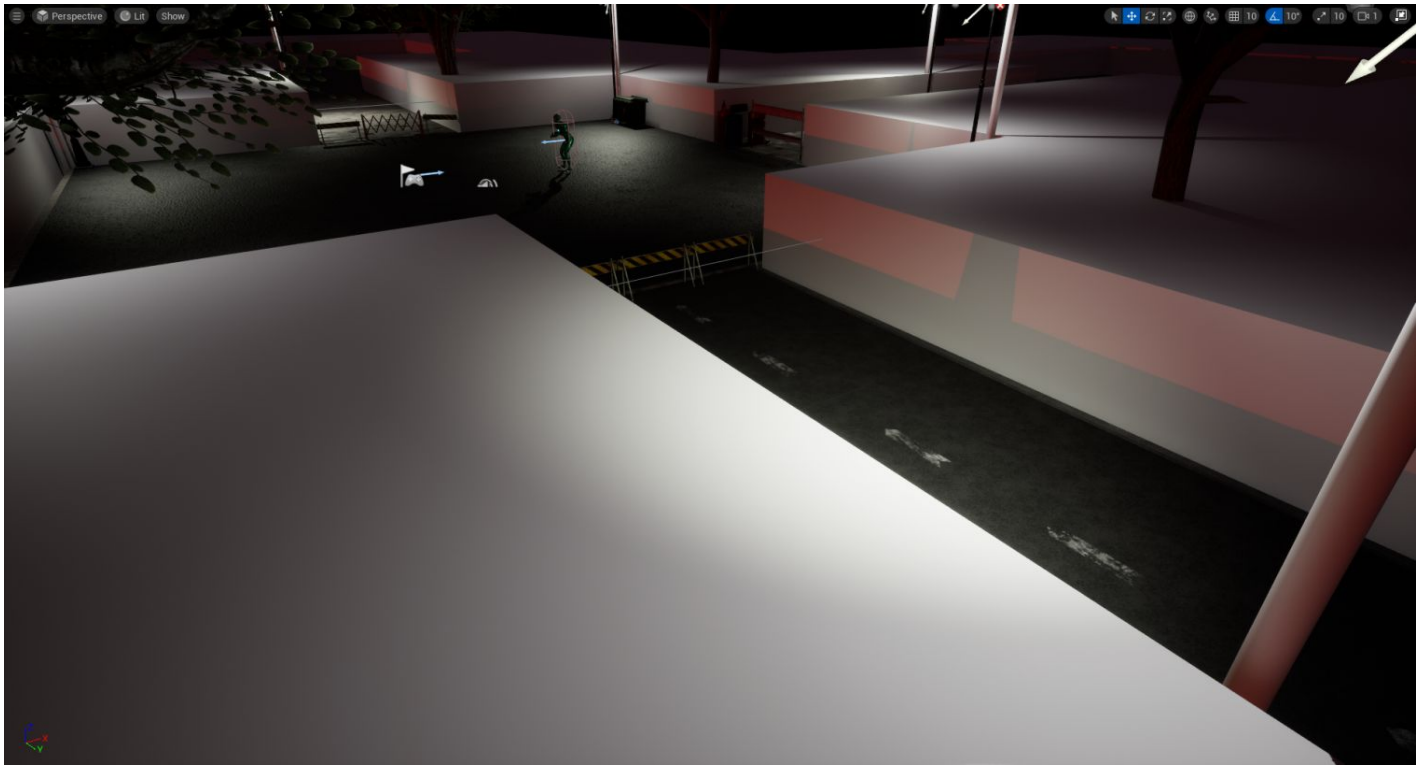
Preview

Посмотрим на небольшие предварительные улучшения уровня в проекте.
Например, были добавлены бесплатные художественные декорации*

* Все материалы были взяты из бесплатных ресурсов на Marketplace

Preview

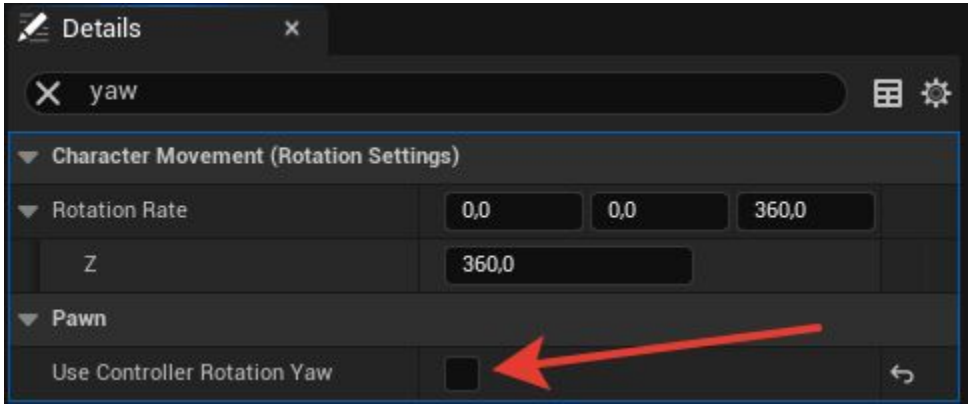
Но это всё тот же ранее созданный нами уровень

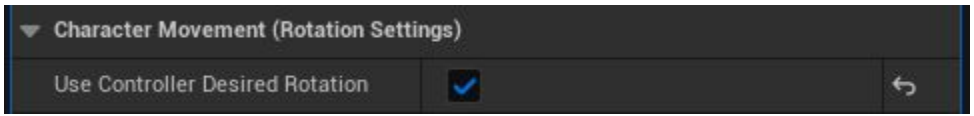


Refactoring

При выполнении домашнего задания вы могли заметить, что противник не очень приятно поворачивается при движении к игроку.

Чтобы персонаж противника осуществлял более корректные повороты по отношению к своему противнику перейдём в CH_Enemy и выполним следующие настройки:





LMABaseWeapon

Научим персонаж наносить урон неигровым персонажам.

Вспомним, что расчёт стрельбы происходит в функции Shoot базового класса оружия, где мы получаем результат стрельбы.

Переходим в заголовочный файл оружия, где объявляем новую функцию и переменную, которая будет хранить количество урона:

```
UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Weapon")
float Damage = 20;

void MakeDamage(const FHitResult& HitResult);
```

MakeDamage

Данная функция в качестве аргумента будет принимать результат попадания трейса. Из него можно узнать в кого попали:

```
void ALMABaseWeapon::MakeDamage(const FHitResult& HitResult)
{
    const auto Zombie = HitResult.GetActor();
    if (!Zombie) return;

    const auto Pawn = UGameplayStatics::GetPlayerPawn(GetWorld(), 0);
    if (!Pawn) return;

    const auto Controller = Pawn->GetController<APlayerController>();
    if (!Controller)return ;

    Zombie->TakeDamage(Damage, FDamageEvent(), Controller, this);
}
```

Overview

Чтобы получить информацию о том, в кого мы попали, необходимо воспользоваться функцией, которая доступна в структуре **FHitResult – GetActor**. Данная функция возвращает указатель на актора, если мы попали в один из его компонентов коллизии.

Далее воспользуемся функцией **TakeDamage**, которая в качестве аргументов принимает следующие параметры:

- количество полученного урона.
- FDamageEvent – позволяет задавать дополнительные параметры. Так как нам это в настоящее время не нужно, зададим пустой конструктор
- контроллер, ответственный за урон
- класс актора, ответственный за нанесённый урон

Shoot

А вызывать данную функцию можно в теле функции **Shoot:**

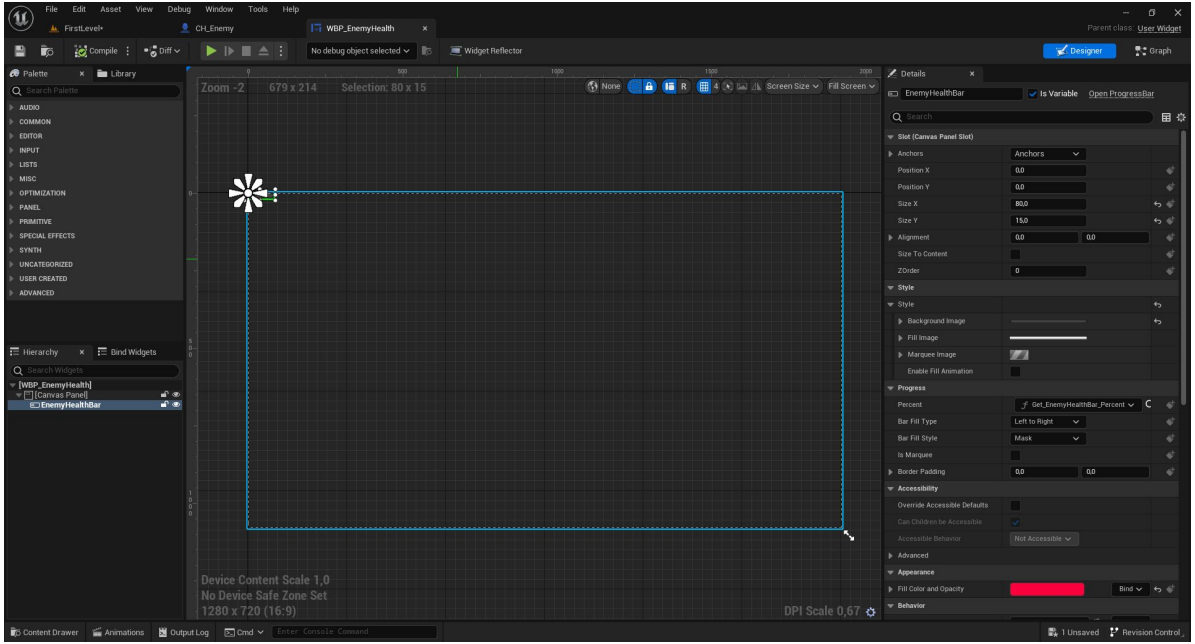
```
void ALMABaseWeapon::Shoot()
{
    …
    if (HitResult.bBlockingHit)
    {
        MakeDamage(HitResult);
        TracerEnd = HitResult.ImpactPoint;
    }
    …
}
```

Скомпилируем код и перейдём в Unreal Editor

EnemyHealthBar

Установим видимость количества жизней у противников.

Создадим новый пользовательский интерфейс – **WBP_EnemyHealth**. Добавим компонент холста и единственный компонент **ProgressBar**

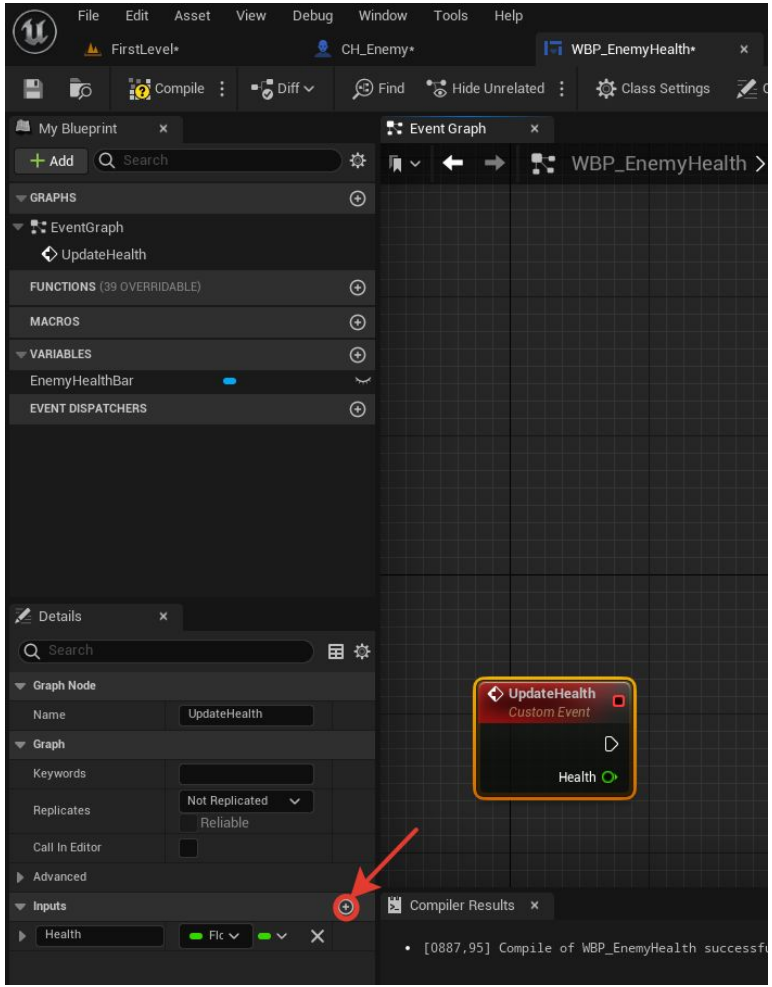


Logic update

Зададим логику обновления шкалы здоровья НПС. Настроим, чтобы шкала здоровья НПС скрывалась до определенного момента и не занимала всю площадь экрана.

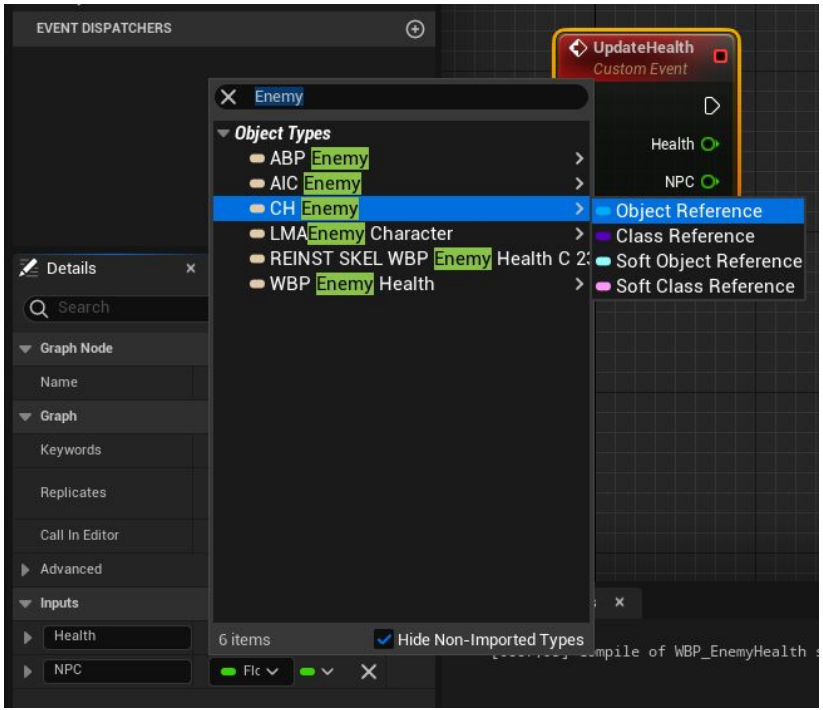
В прошлый раз мы пользовались биндом. Теперь рассмотрим другой вариант.

1. Перейдём в поле **Graph** пользовательского интерфейса.
2. Создадим своё пользовательское событие (в свободном месте жмём ПКМ > **Add Custom Event)**. Дадим имя событию – **UpdateHealth**.
3. Предоставим входной параметр, который будет к нему применяться. Данные параметры равносильны аргументам функций в коде. Тип данных у него будет **float** и назовём его **health**



Add input variable

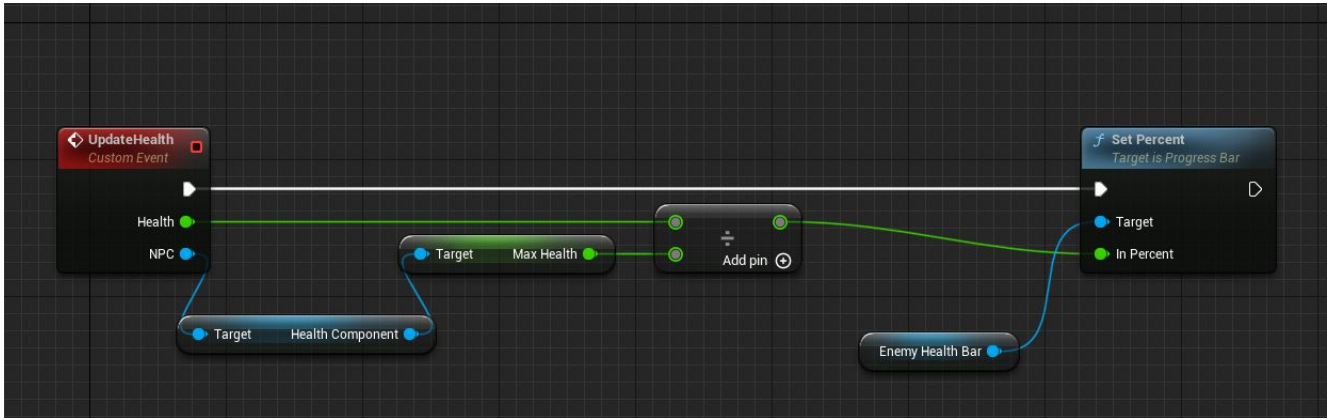
Зададим ещё один входной параметр, который будет типом указателя на неигрового персонажа



Set Percent

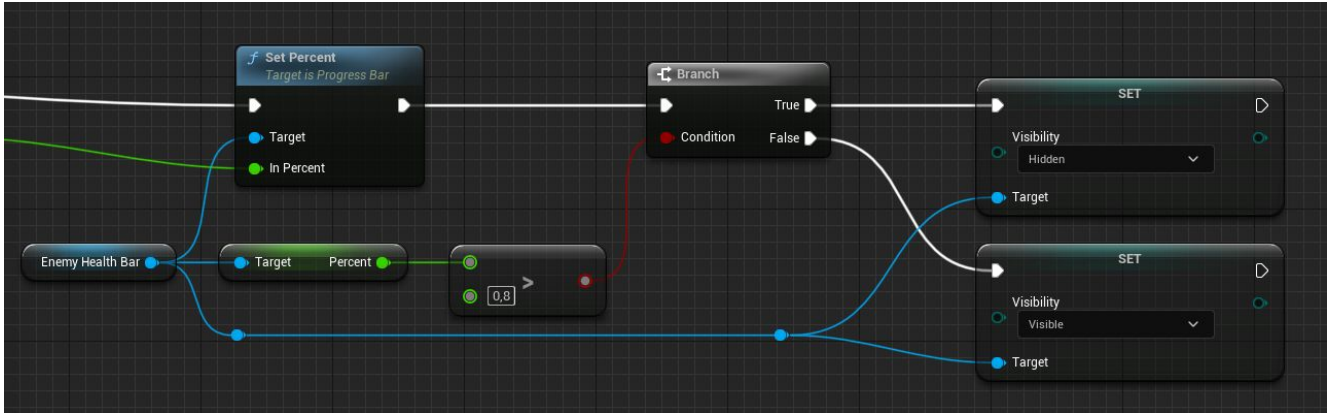
Данное событие будет вызывать в неигровом персонаже. Вспомним, что к нему мы добавляли компонент здоровья. Это значит, что мы можем обратиться к нему и узнать его максимальное здоровье, которое мы разделим на текущее состояние здоровья.

Далее необходимо выбрать созданный ProgressBar и вызывать его функцию SetPercent, в которую передадим текущее состояние здоровья



Set Visibility

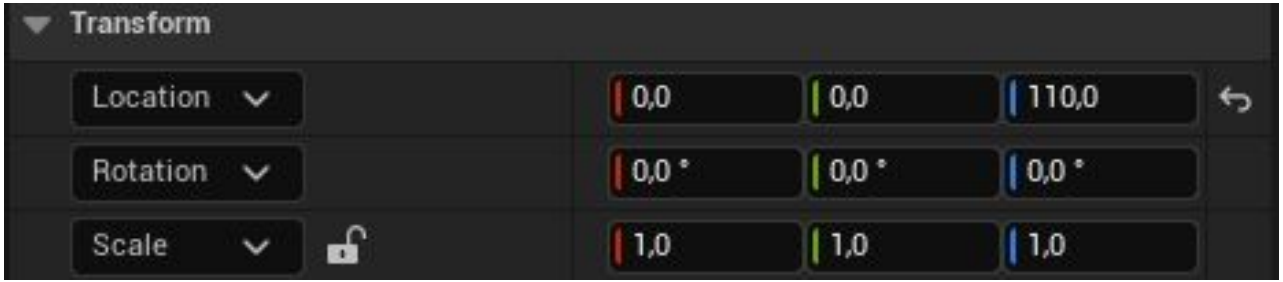
На последнем этапе установим, что если текущее состояние здоровья больше 80%, то шкала здоровья будет скрываться.



Set Visibility

Далее переходим в CH_Enemy. Добавим новый компонент – **Widget**. Настроем его следующим образом.

Шаг 1. Зададим параметры локации



Widget settings

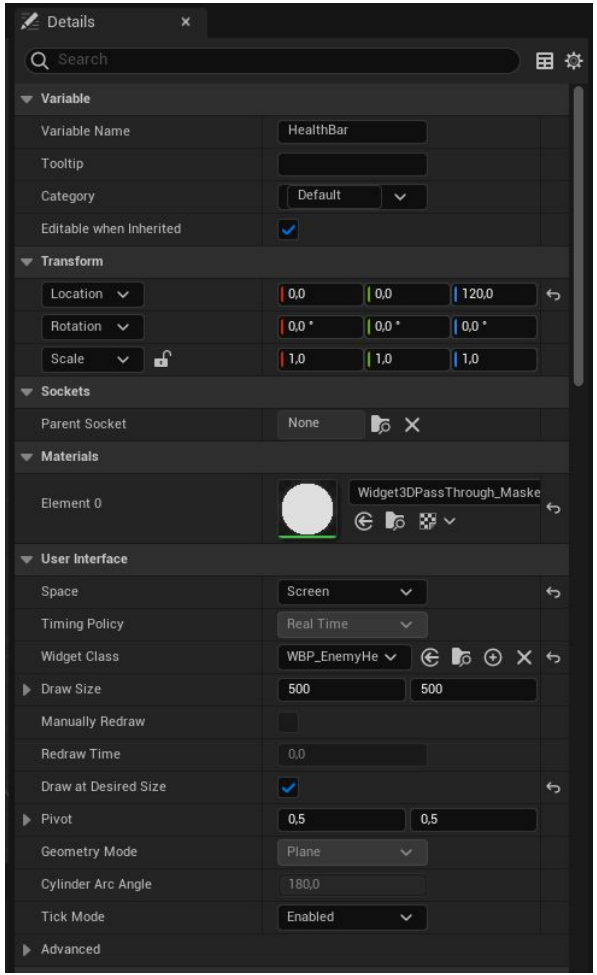
Шаг 3. Поле Space установим в **Screen.** Это позволит отображать пользовательский интерфейс без учёта 3D пространства, в слое дисплея.

После всех настроек в этом блоке проекта можно попробовать изменить заданные параметры, чтобы наглядно увидеть разницу их настроек

Widget settings

Шаг 4. В поле Widget Class передадим виджет со здоровьем противника.

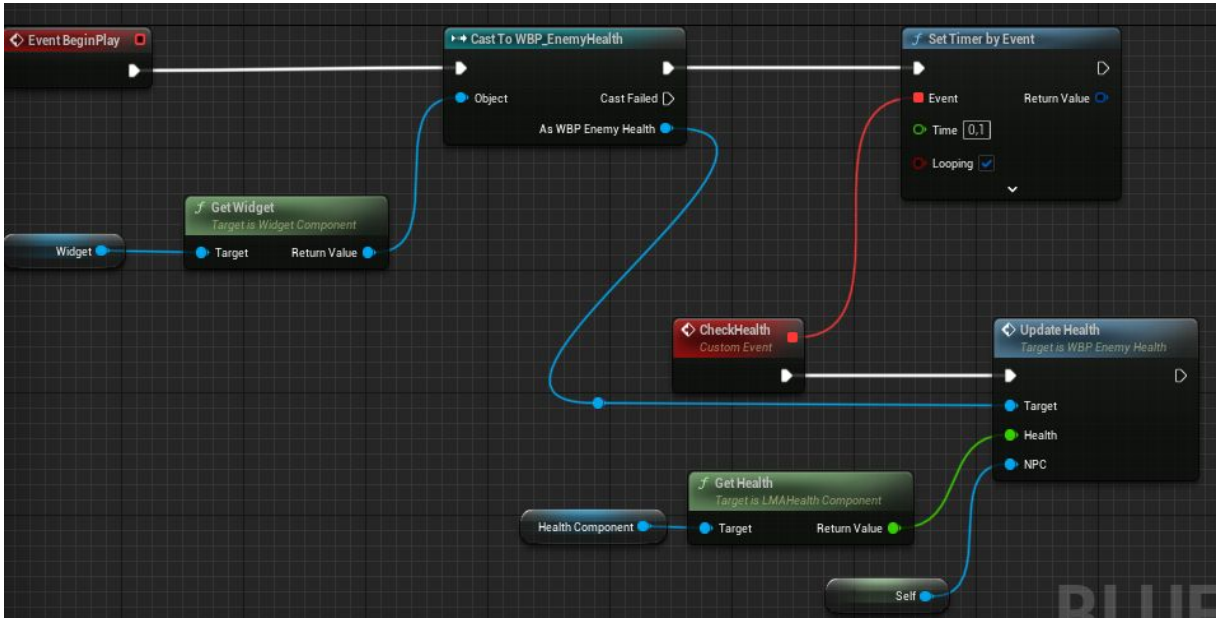
Значения поля Draw at Desired Size установим в true. Это будет означать, что health bar будет изображён над противником, без учёта его нахождения на холсте



Widget logic

В проекте добавлен компонент здоровья к неигровому персонажу, в коде которого был реализован делегат изменения здоровья.

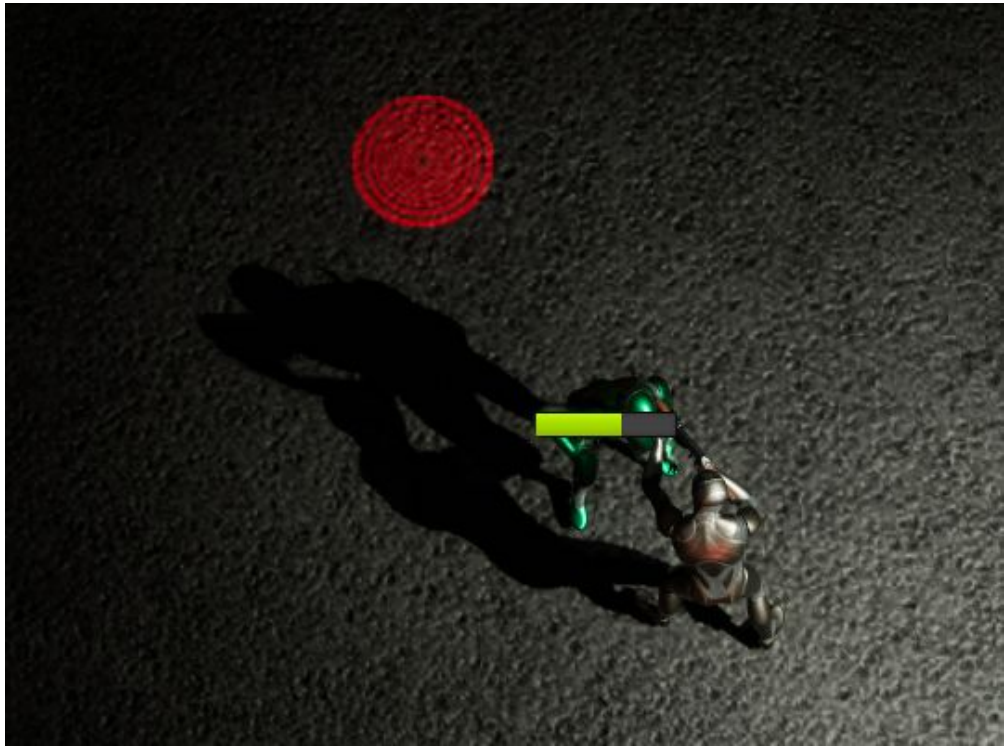
Одно из решений – переопределить данный делегат, чтобы вызывать его в коде. В учебных целях потренируемся и обратимся к нашему виджету прямо из **Event Graph** противника:



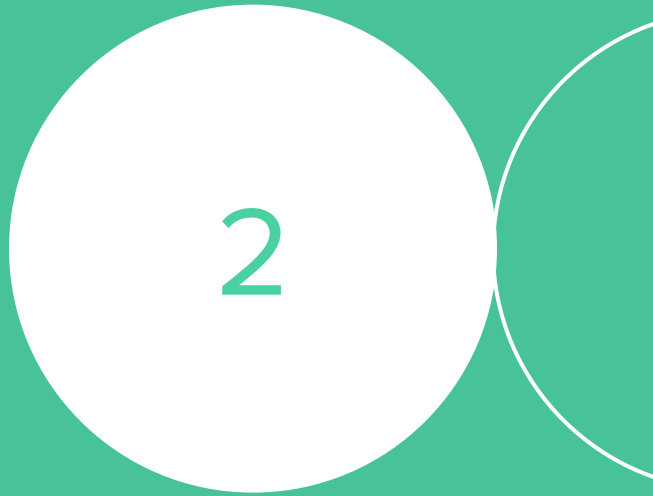
Итоги

1. Воспользуемся функций GetWidget, которая вернёт указатель на тот виджет, который мы указали в строке Widget Class.
2. Скастуем данный указатель к WBP_EnemyHealth.
3. Создадим зацикленный таймер, который будет проверять текущее состояние здоровья противника.

В результате, если мы запустим игру, то увидим, что при попадании по НПС, его шкала здоровья изменяется



Смерть НПС



AnimMontage

После того, как мы научили нашего персонажа наносить урон неигровым персонажам, сделаем так, чтобы в тот момент, когда его жизни закончатся, он погибал.

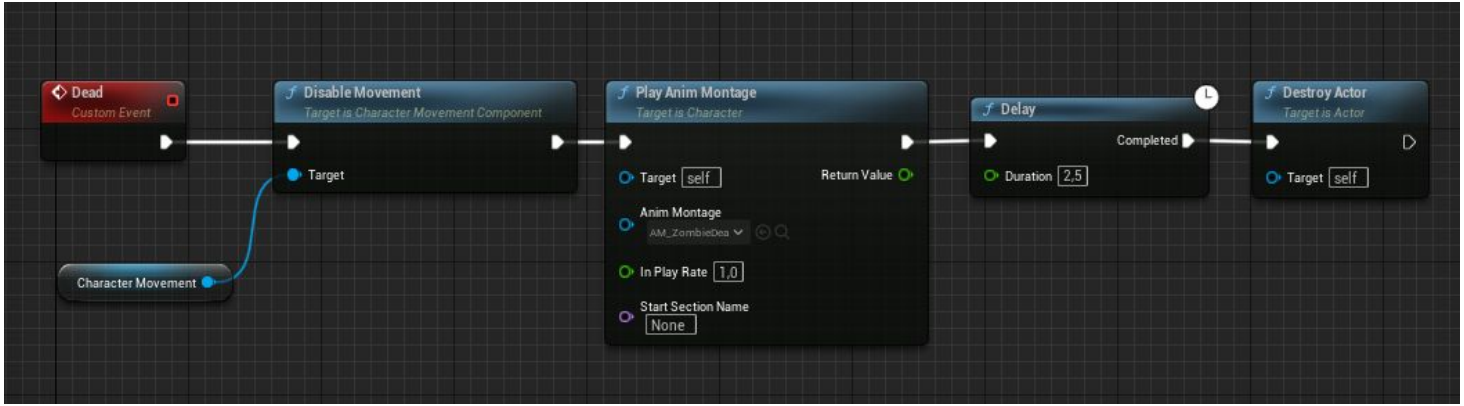
Для начала необходимо создать анимационный монтаж. В анимациях противника есть подходящая анимационная последовательность, которая называется **ZombieDeath**. На её основе создаём монтаж

Custom Event

После перейдём в базовый класс противника **CH_Enemy**.

Создадим новое пользовательское событие и назовем его **Dead**. Теперь сформулируем его логику:

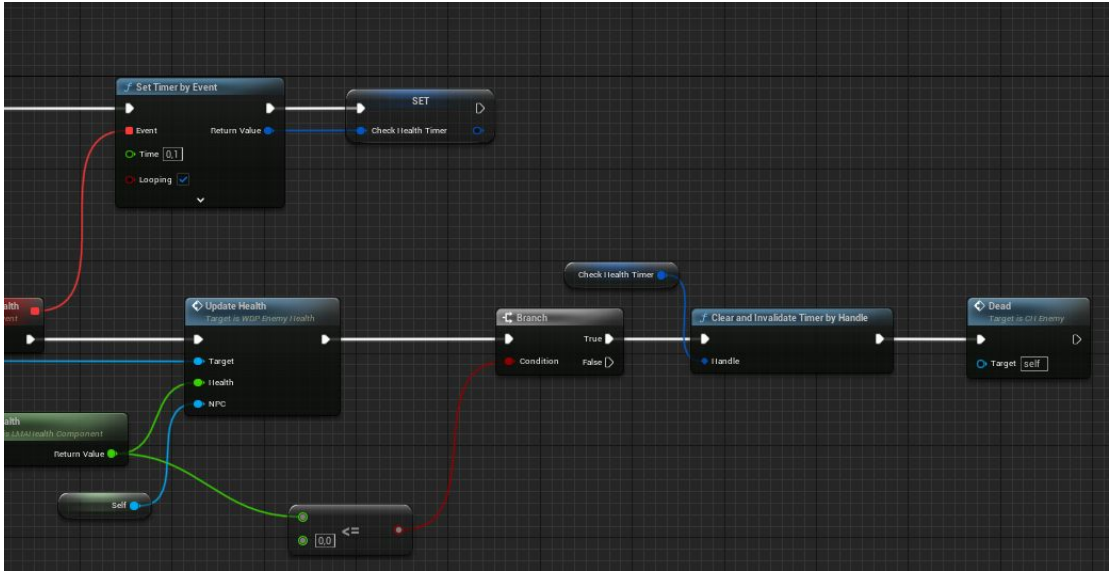
- все движения, кроме монтажа, должны останавливаться
- проигрывается непосредственный монтаж
- выполняется небольшая задержка
- уничтожается объект



Dead

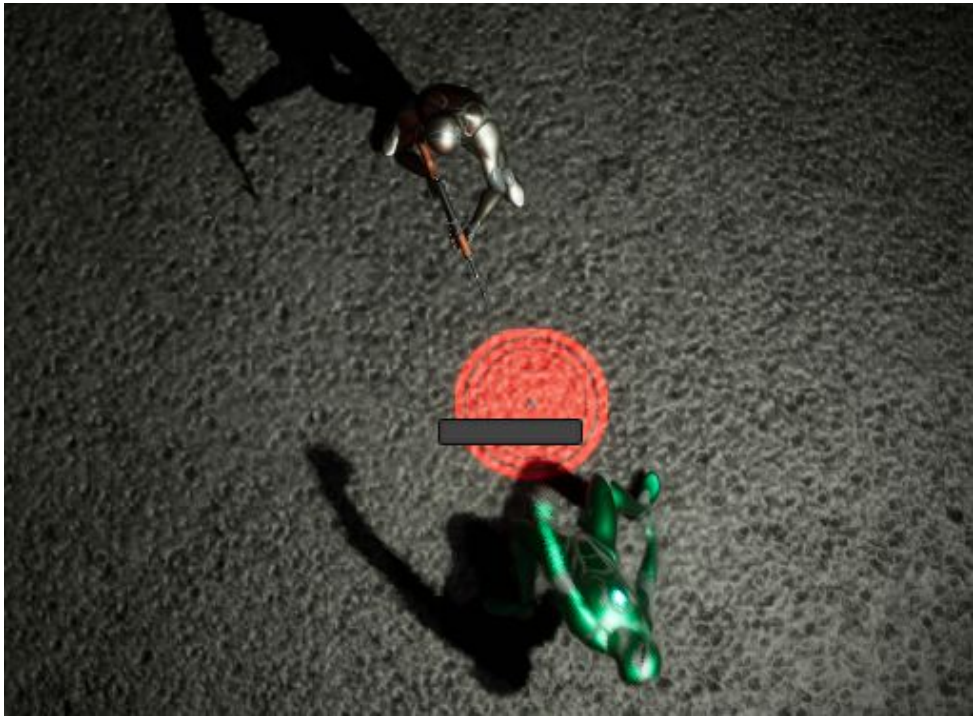
Дополним сформированную логику:

1. вынесем созданный таймер в самостоятельную переменную
2. проверим количество жизней НПС на 0. Если заданное условие получить правду, таймер останавливается и вызывается событие



Название

В результате, если запустить игру и довести жизни НПС до 0, он прекратит преследование, воспроизведётся анимация смерти и противник исчезнет



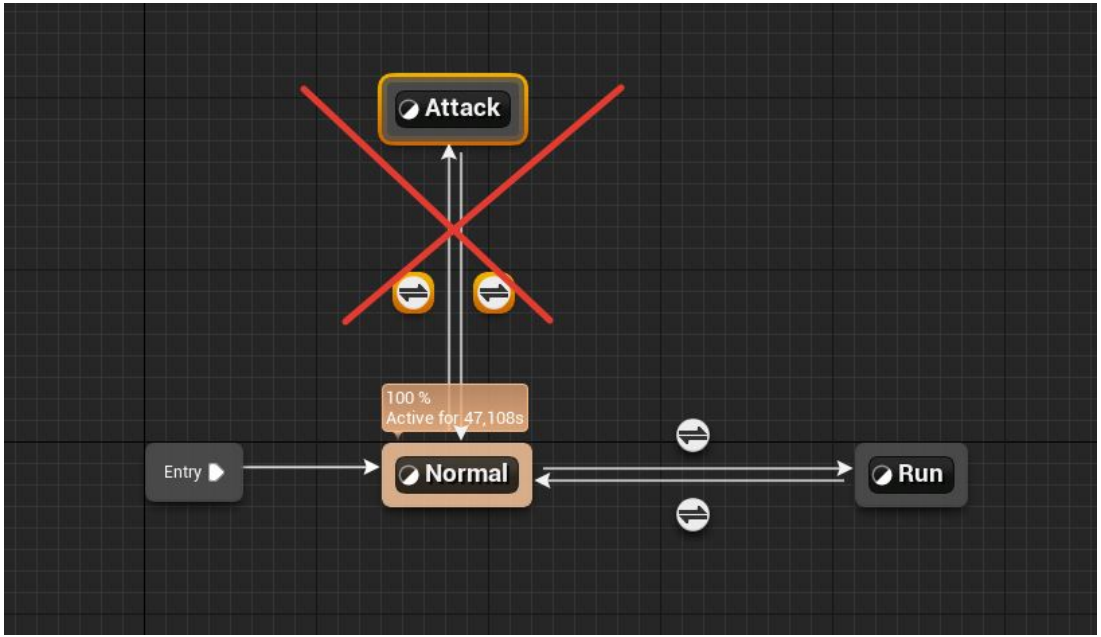
Нанесение урона неигровым персонажем

3

Refactoring

Видоизменим анимационный блюпринт ABP_Enemy.

Более корректно контролировать атаку противника в рамках анимационного монтажа, поэтому уберём состояние атаки



AnimMontage

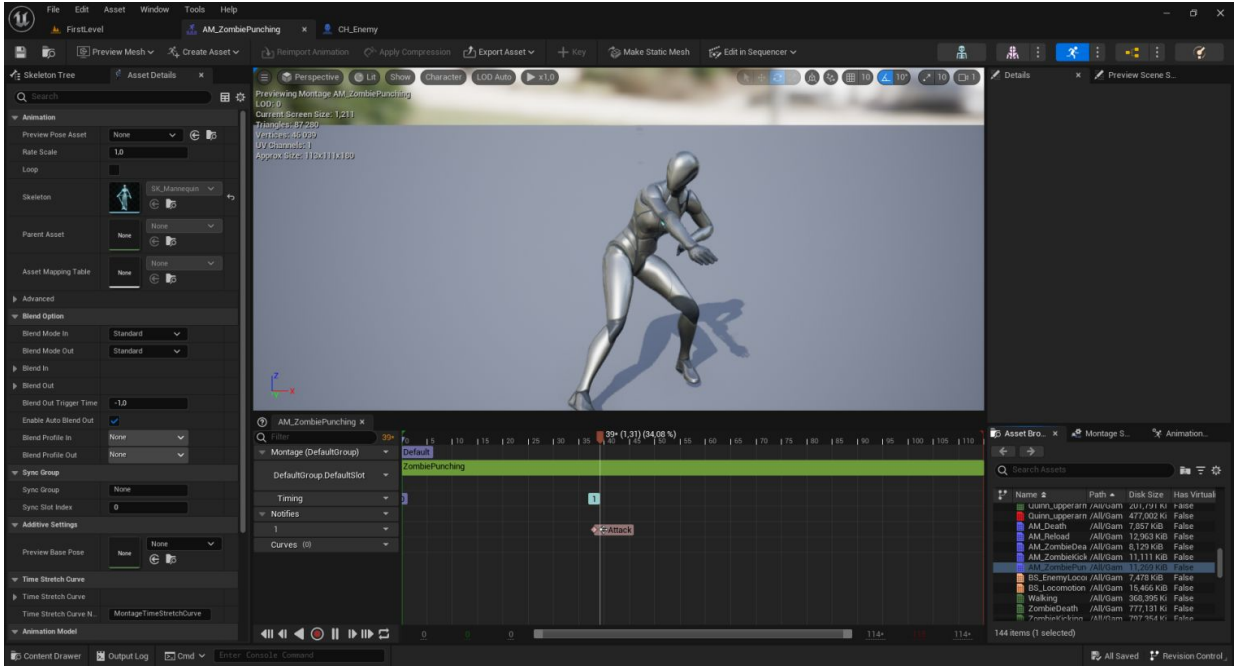
На основе имеющихся анимаций создадим 2 анимационных монтажа:

- ZombiePunching
- ZombieKicking

ZombiePunching

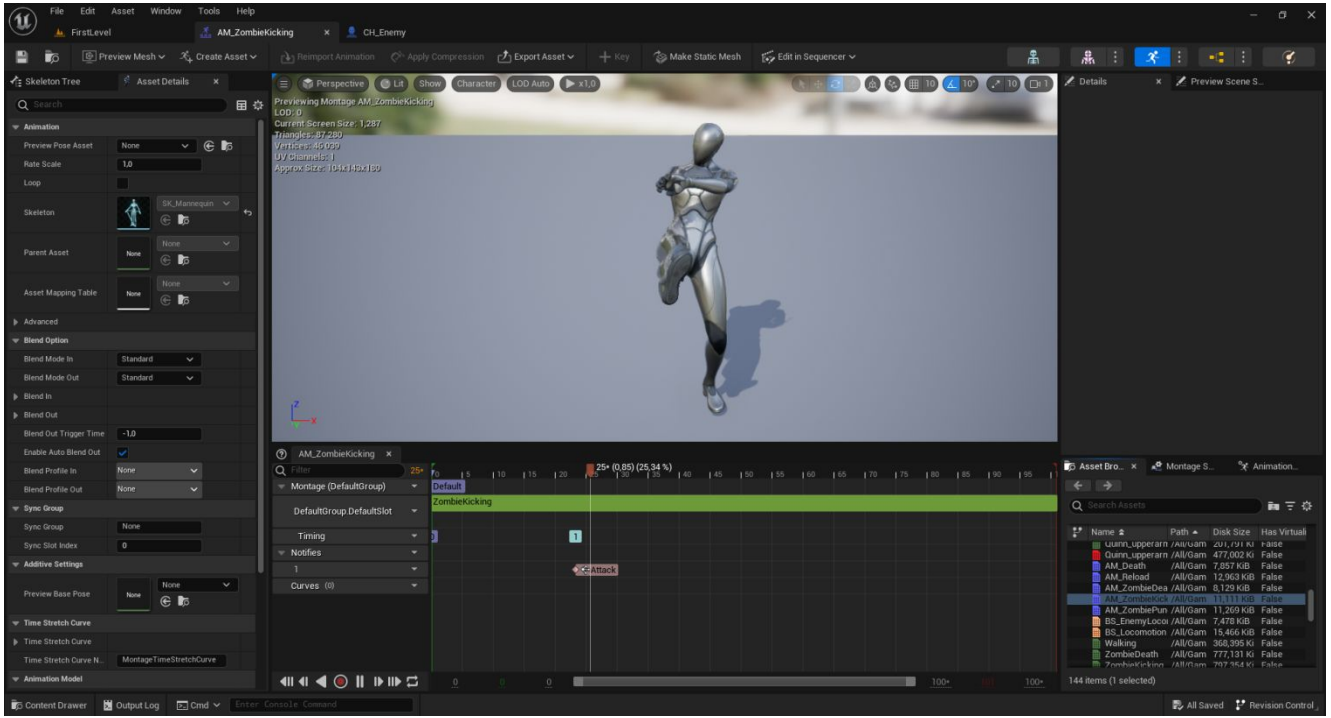
Чтобы определить в какой момент анимации игроку может прийти урон, необходимо подобрать корректный фрагмент, когда рука НПС будет примерно находиться в моменте удара.

В данный кадр добавим новое событие Notify. Назовём его – Attack



ZombieKicking

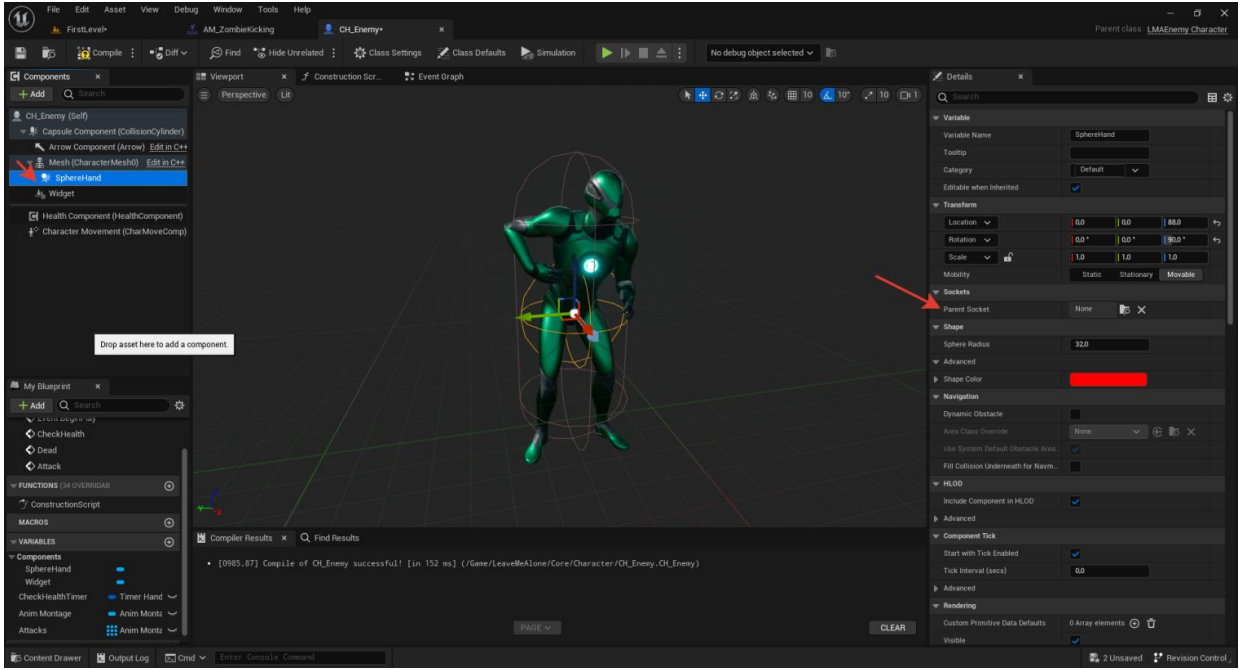
Аналогичные действия мы произведём по отношению к анимации ZombieKicking



Sphere Collision

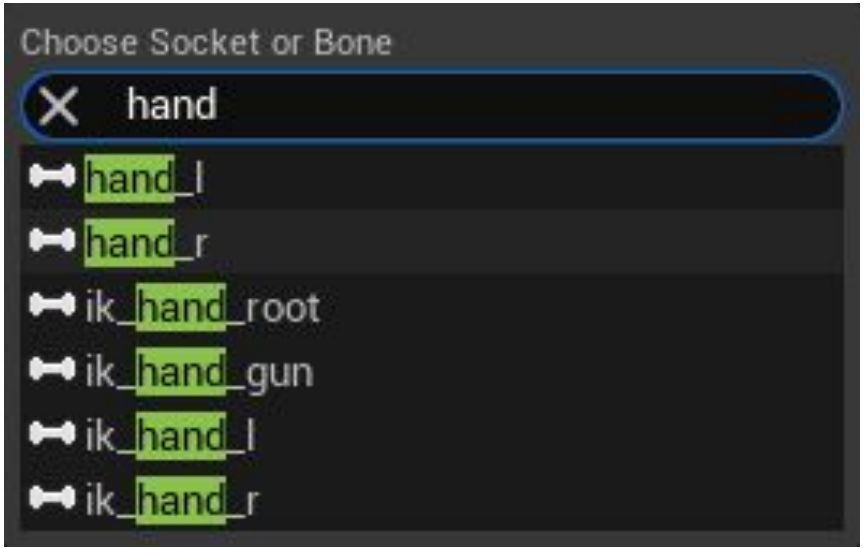
Перейдём в CH_Enemy и добавим для НПС новый компонент сферической коллизии.

Далее перетащим созданный компонент под иерархию компонента Mesh, где имеется возможность прикрепить его к кости hand_r в поле Parent socket



Hierarchy

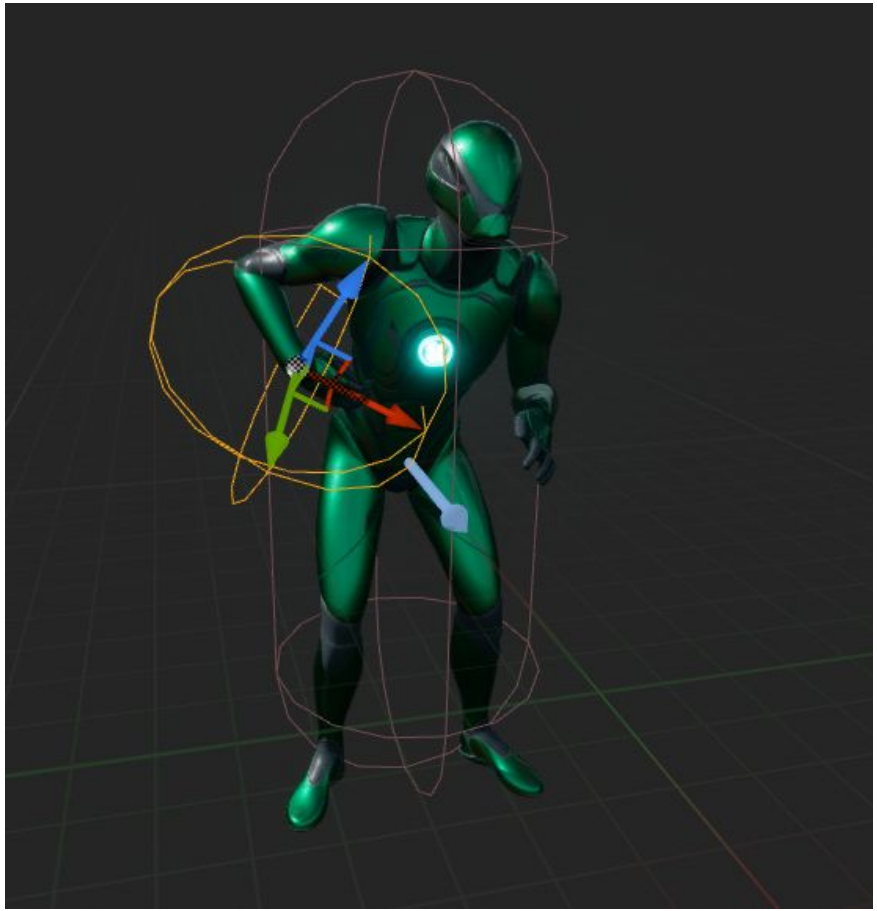
Данное поле знает всю иерархию костей персонажа, а значит здесь можно напрямую прикрепить сферу к кости руки



Итоги

В результате рука неигрового персонажа будет иметь собственный объект коллизии, с которым можно воспроизвести определённую логику.

Настройку сферы для ноги и логику удара ногой выполним по аналогии самостоятельно



Custom Event

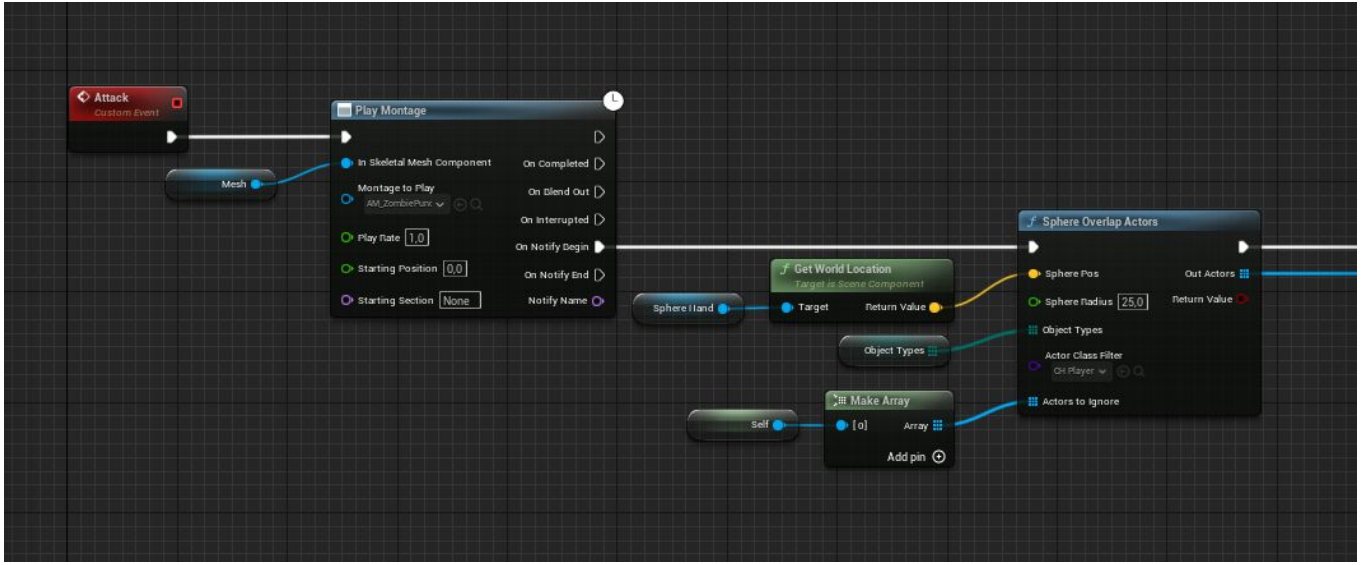
Создадим пользовательское событие, которые в дальнейшем будем вызывать в дереве поведения искусственного интеллекта.

В этом событии будем воспроизводить анимационный монтаж удара рукой, воспользовавшись функцией Play Montage. Из этой функции можем получить событие On Notify Begin, которое хранит в себе ключ, который мы создали в анимационном монтаже

Attack

Далее нужно понять пересекла ли рука какую-либо коллизию.

Для этого воспользуемся функцией Sphere Overlap Actors. Входные параметры этой функции будут состоять из координат сферической коллизии в мире. Получим глобальные координаты этого компонента



Sphere Overlap Actors

В поле Object Types передадим тот тип объекта, которому соответствует наш игровой персонаж.

Проверить к какому типу относится игрок можно, если перейти в CH_Player и посмотреть его категорию Collision, где имеется интересующее нас поле – Object Type = Pawn.

Это значит, что именно такое значение мы передаём в Object Types функции Sphere Overlap Actors

Sphere Overlap Actors

В поле **Actor Class Filter** передадим персонажа игрока.

В **Actors to Ignore** вытянет булавку и вызовем функцию **Make Array**.

Чтобы противник не наносил урон сам себе, внесём значение **Self** в данный массив. По сути, данное значение равносильно this в коде

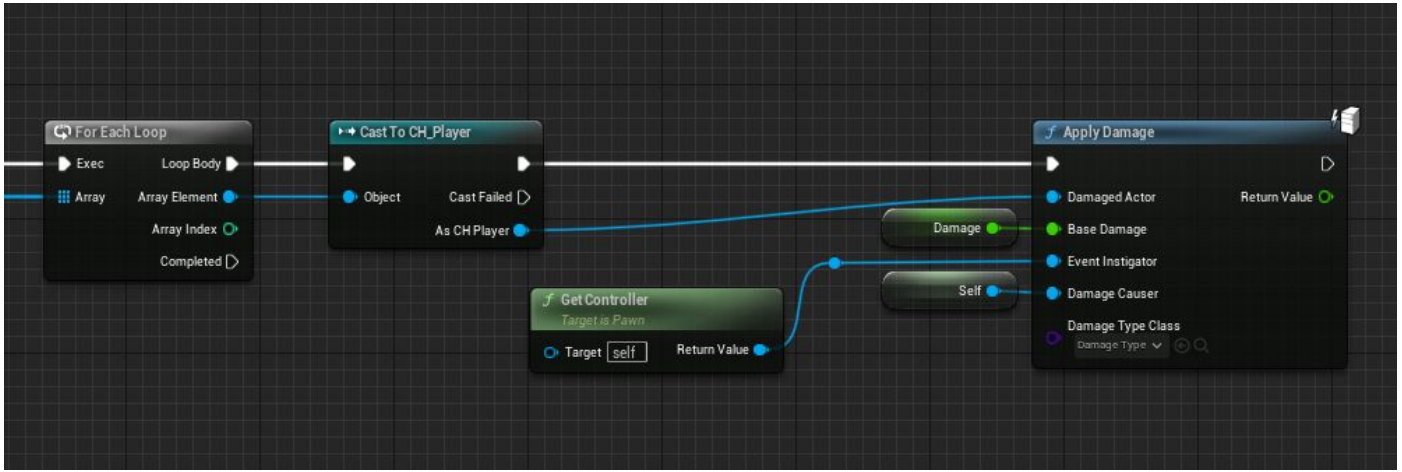
For Each Loop

На выходе данная функция имеет два параметра:

- массив коллизий, с которыми соприкоснулась сфера
- второй аргумент – это логическое значение с результатом – попали ли мы вообще куда-то.

Нас интересует массив, по которому нужно пройтись циклом и найти в нём нашего персонажа.

Различных For в Blueprints имеется несколько видов, и все они в том или ином смысле соответствуют классическим кодовым циклам, к которым мы привыкли. Вызываем цикл For Each Loop



Apply Damage

Внутри цикла будем проверять каждый элемент массива на его принадлежность к нашему персонажу путём кастинга к CH_Player. Если он совпадёт, вызываем функцию Apply Damage.

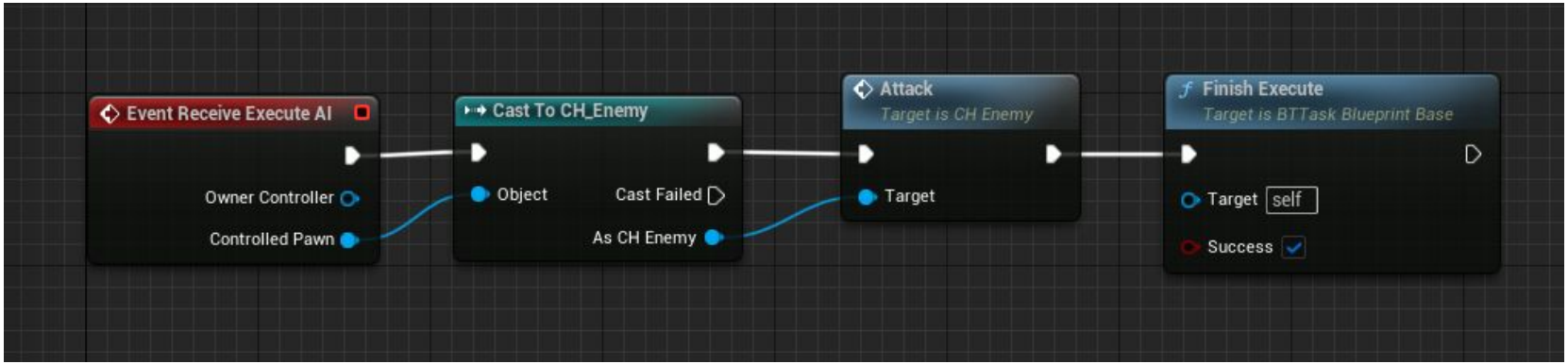
- В поле Damaged Actor можно передать нашего игрока,
- Для поля Base Damage создадим переменную, которую сможем в дальнейшем редактировать. Она будет иметь тип float. По умолчанию зададим ей значение = 10. Передадим переменную в качестве аргумента.
- В поле Event Instigator передадим контроллер, ответственный за нанесённый урон.
- В параметр Damage Causer нужно задать актора, ответственного за урон.
- В Damage Type Class передадим пустой конструктор Damage Type.

Как вы могли заметить, данная логика полностью соответствует тому, что мы делали в коде, когда наносили урон вражескому персонажу нашим игроком

BTTask

Перейдём в дерево поведения искусственного интеллекта и создадим новую задачу выполнения. Назовем её **BTT_Attack**.

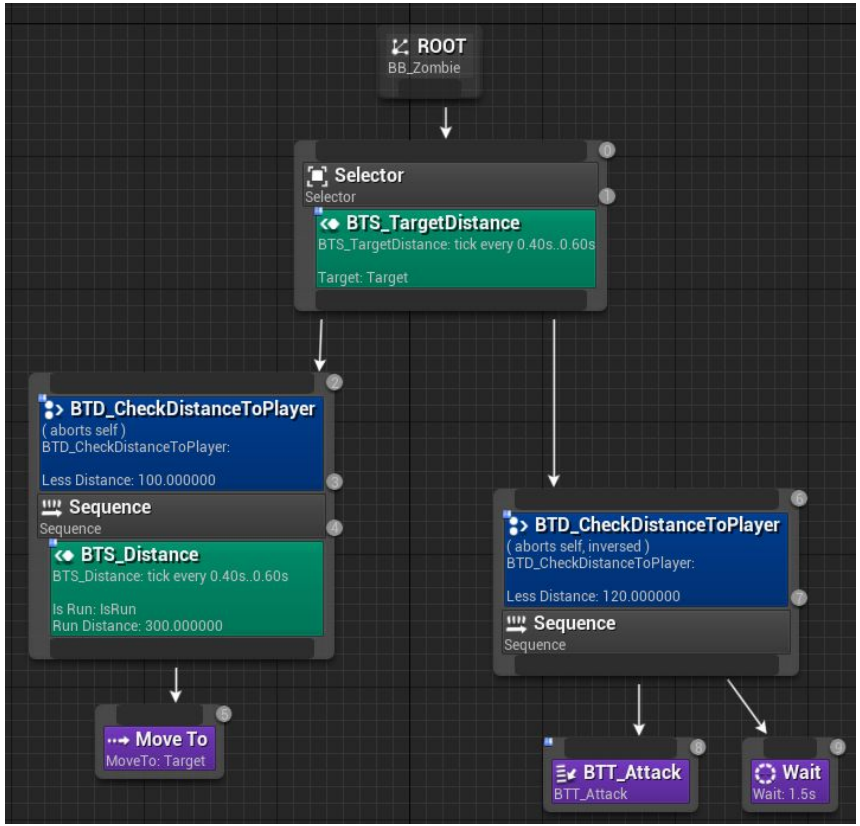
Логика в ней простая – мы проверяем, что контролируемая искусственным интеллектом пешка действительно является CH_Enemy. Если каст успешен, то вызываем в нём событие атаки, которое мы реализовали только что



Behavior Tree

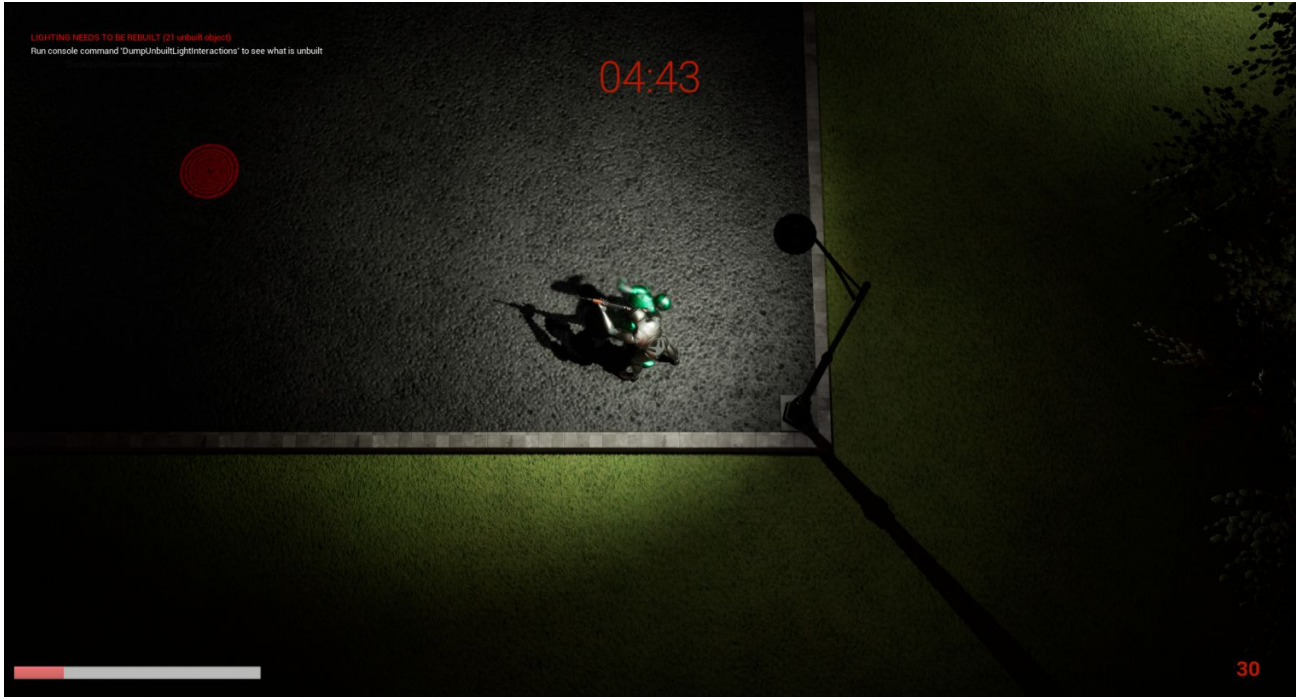
После чего обновим наше дерево и вызовем новую созданную задачу, чтобы НПС мог её выполнить.

Теперь, когда НПС будет приближаться к игроку на расстояние 120 см, он будет атаковать игрока, немного ожидать и атаковать снова



Итоги

В результате, если мы запустим игру, то увидим, что НПС атакует игрока при приближении. Жизни отнимаются корректно



Итоги

Если персонаж не убежит, то он погибает и вызывается экран смерти

Spawn Enemy

Preview

Теперь, когда мы можем наносить урон противнику, а он может наносить его нам и когда реализованы события смерти НПС и игрока, можно приступать к реализации основной логики игры. НПС должны порождаться и атаковать игрока

NavMesh

Для начала вернёмся на уровень и сформируем навигационную сетку таким образом, чтобы она покрывала всю игровую карту

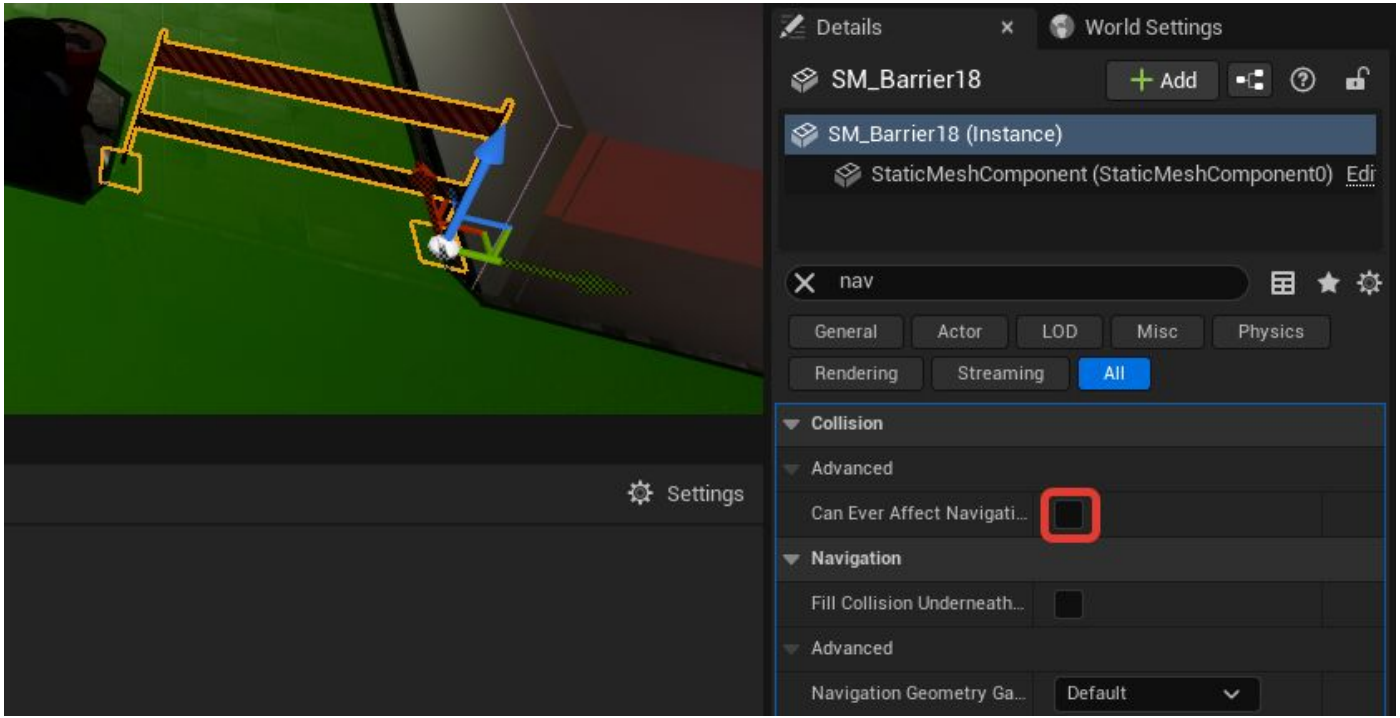
NavMesh

Здесь мы можем столкнуться с неприятным моментом. Это связано с тем, что художественные препятствия просчитываются навигационное системой, как препятствия. Это в последствии не позволит искусственному интеллекту преодолеть их, так как сетка в нашем проекте имеет статический тип.

Чтобы обойти эти сложности, можно выбрать несколько путей. Самый простой – это исключить данные объекты из числа препятствий

Obstacle

Для этого выбираем препятствие, ищем его поле Can Ever Affect Navigation и устаналиваем его в false



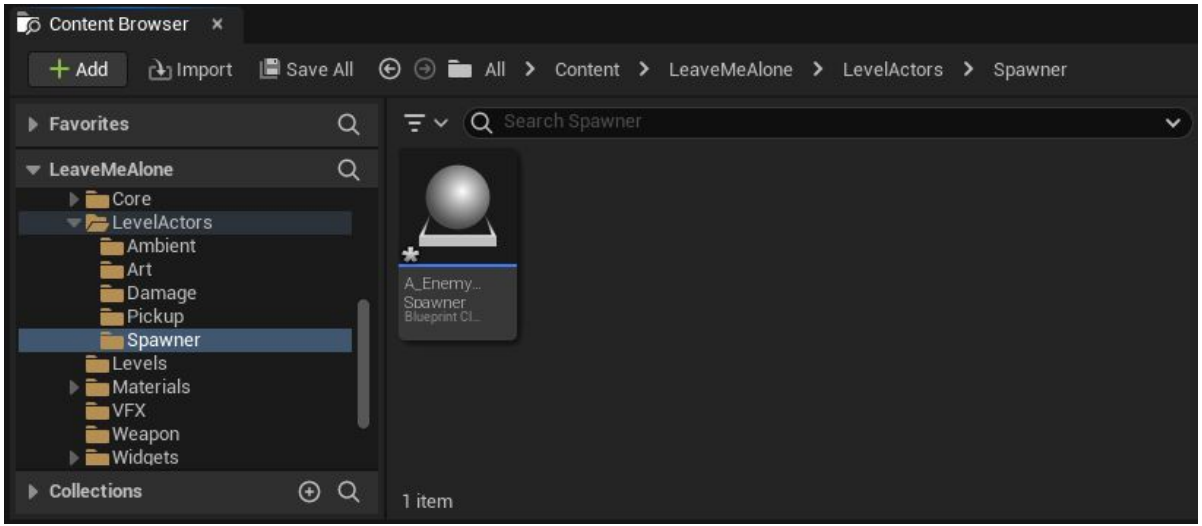
Fix

В результате объекты не мешают просчёту сетки и после их открытия искусственный интеллект сможет преодолеть данный участок

Spawner Actors

Теперь мы можем создать объект, в котором будет происходить спаун наших противников.

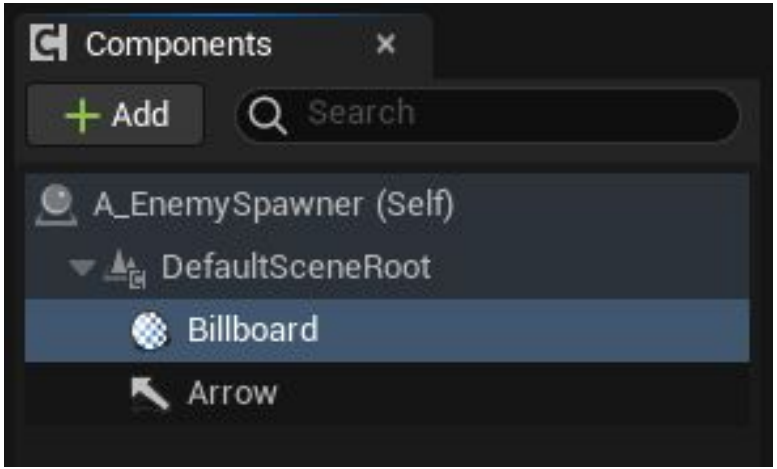
Для начала создадим новую директорию Spawner в папке LevelActors.
Внутри неё создадим новый БП класс на основе базового класса актора.
Зададим ему имя A_EnemySpawner



Add Components

Отличная работа! Во многом логика, связанная с корректной работой вражеского персонажа, уже реализована. А значит при спауне вражеский персонаж корректно будет принимать контроллер искусственного интеллекта, а также начнёт воспроизводить нужно дерево поведения.

Всё, что нужно сделать, это задать A_EnemySpawner несколько компонентов:



Billboard

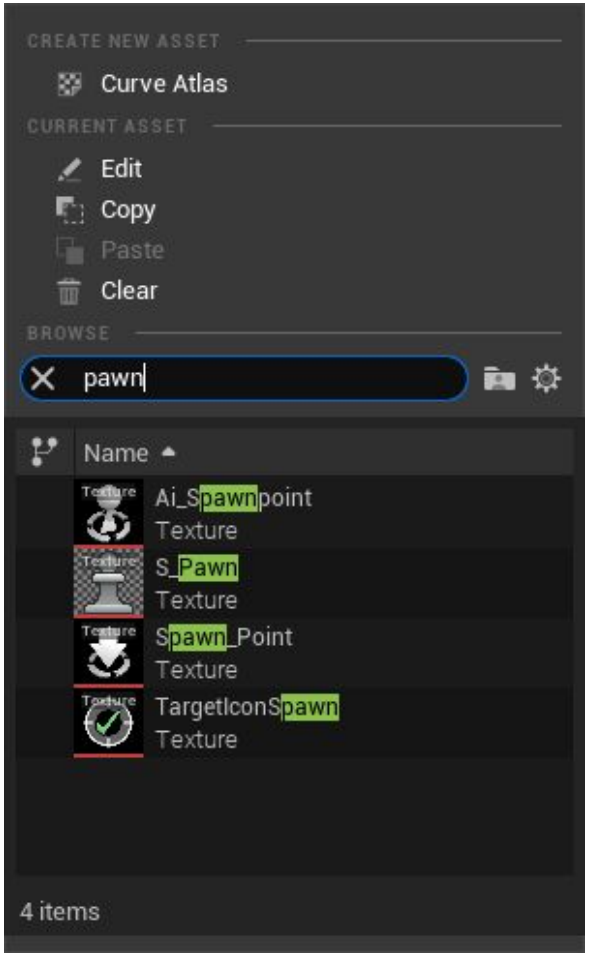
Компонент биллборда поможет нам при работе в эдиторе. Это больше косметическое удобство.

Настраиваем его следующим образом:

Components settings

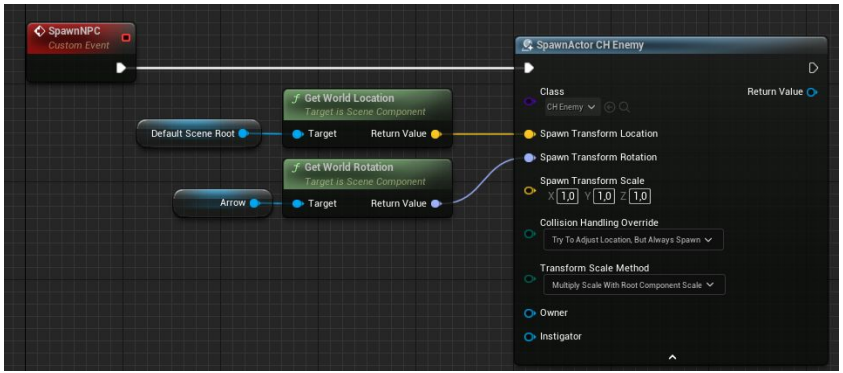
Если не будет видна текстура спауна Ai_Spawnpoint, нажмите на знак шестерёнки и поставьте галочку в поле Show Engine Content.

Компонент стрелки определит ротацию, в которую будет определён заспауненный противник



Custom Event

1. Перейдём в Event Graph и создадим пользовательское событие **SpawnNPC**.
2. В нём вызовем функцию **Spawn Actor from Class**. В параметр Class передадим **CH_Enemy**.
3. Выполним **Split Struct Pin** для параметра Spawn Transform
4. Передадим в поле локации положение корневого компонента в мире. Ротация будет определяться положением стрелки, которую мы добавили в качестве компонента.
5. Изменим значение Collision Handing Override на Try To Adjust Location, But Always Spawn. Это означает, что если в точке спауна будут помехи, то НПС всё равно заспаунится. Немного изменим трансформацию точки спауна



Игровые условия

5

FirstLevel

Реализуем игровой цикл, в котором будет протекать игра. Для начала установим наши спаунеры на уровне

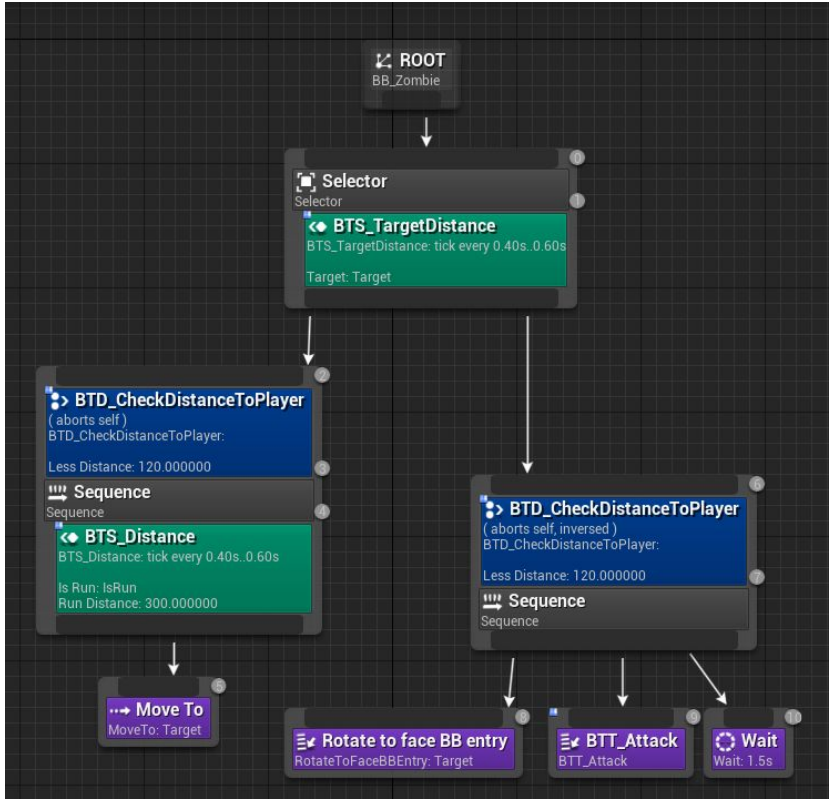
Refactoring

Перейдём в дерево поведения искусственного интеллекта и немного модернизируем его.

Чтобы НПС не били воздух, а пытались поразить игрока, добавим новую задачу, которая уже реализована в движке – **Rotate to face BB entry**.

В качестве ключа она принимает точку в трёхмерном пространстве с местом расположения цели.

В нашем случае, достаточно будет передать ключ Target. Теперь НПС будут всегда проверять корректно ли они расположены относительно игрока перед тем, как его ударить



Refactoring

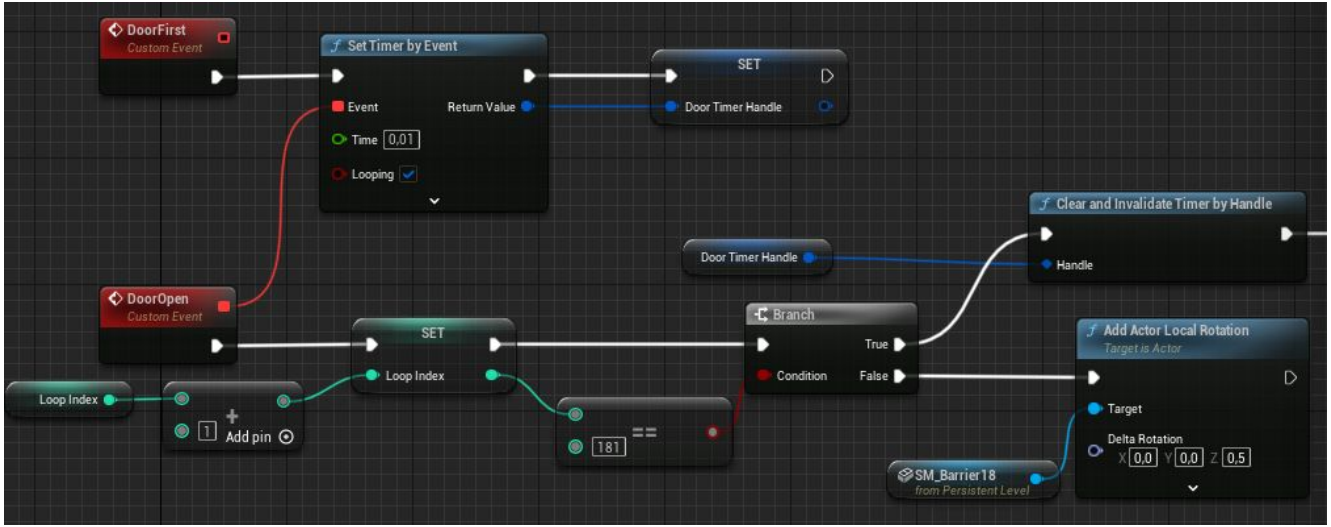
Далее перейдём в Level Blueprint и зададим логику игрового уровня. То есть сформируем геймплейную часть нашей игры.

Для начала удалим НПС, над которым мы проводили испытание на уровне. Теперь все неигровые персонажи будут появляться только динамически

Custom Event

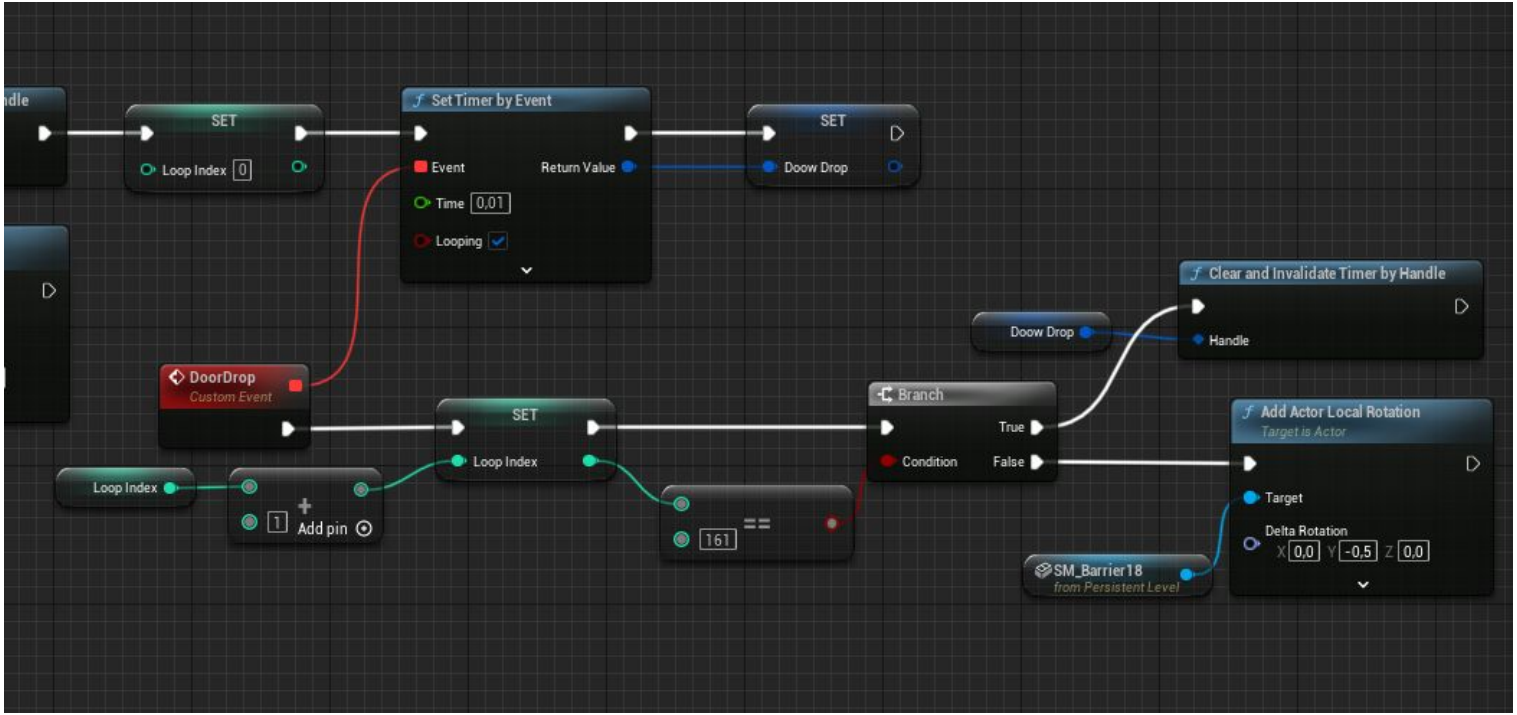
Сделаем так, чтобы ворота через которые противники будут нас атаковать открывались со стартом игры. Ранее мы уже создавали подобную логику, но теперь она будет работать не по триггеру.

Создадим пользовательское событие, которое будет в точности повторять наши предыдущие действия



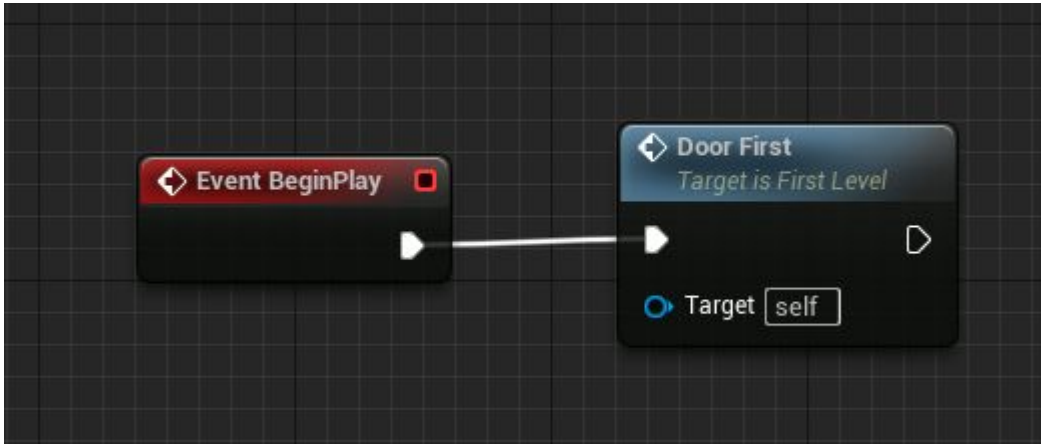
DoorDrop

Далее немного её модернизируем, чтобы ворота не только отодвигались, но и падали



Logic

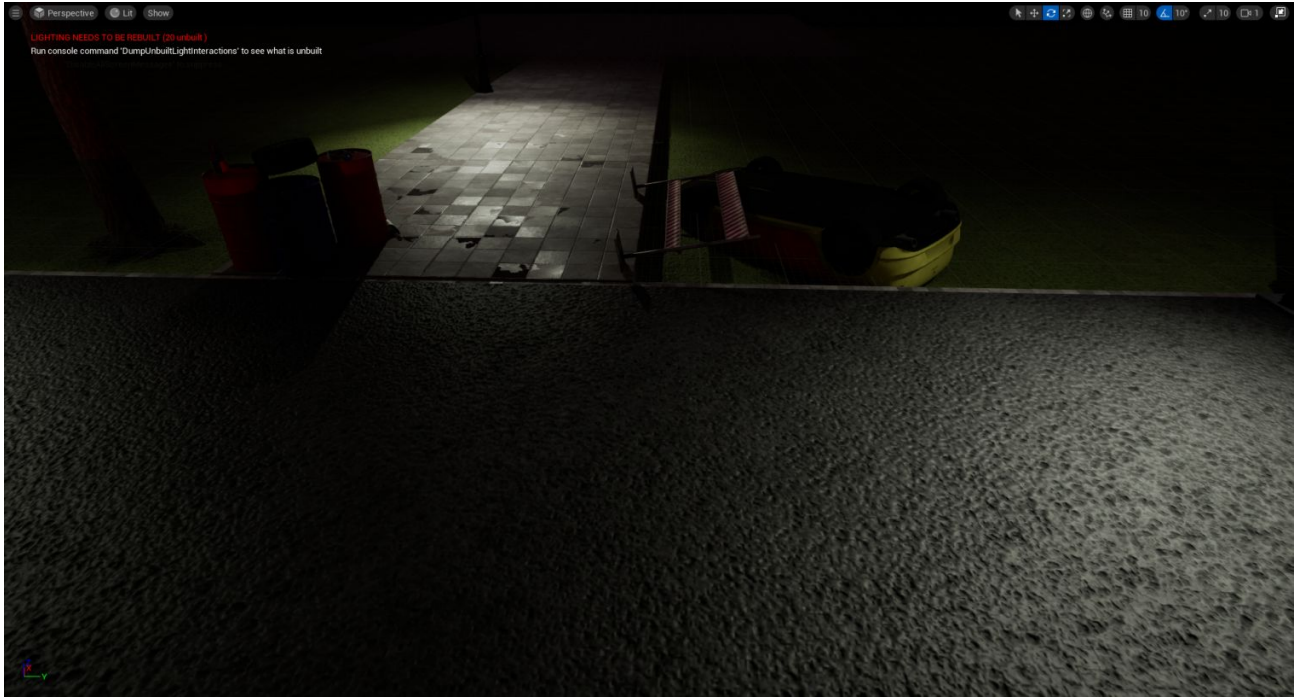
По сути, подход аналогичен. Единственное, нам требуется сбросить переменную LoopIndex. Подберите подходящие значения по оси Pitch, чтобы ваш предмет не пересекался с другими сетками коллизий



Итоги

Далее вызовем данное событие на Event BeginPlay.

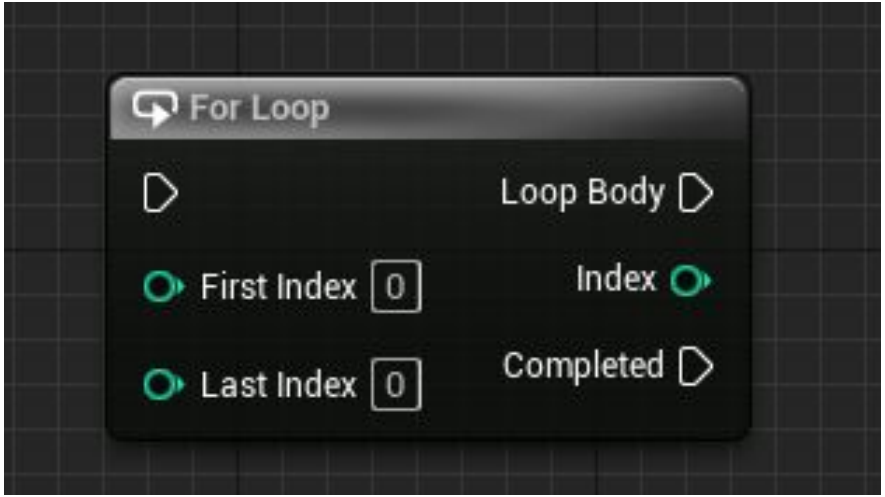
В результате должен получиться подобный механизм:



Custom Event

Активируем врагов. Сделаем это разнообразно, чтобы задать динамическую сложность по мере прохождения уровня. Мы не геймдизайнеры, но технически это можно воспроизвести.

Создадим новое пользовательское событие и назовем его **FirstSpawnerActivate**. Для этого нужно вызвать цикл for loop, который визуально может быть не совсем понятен. По своей сути он работает, как аналогичный цикл в C++. Он принимает первые и последний индекс цикла



Loop settings

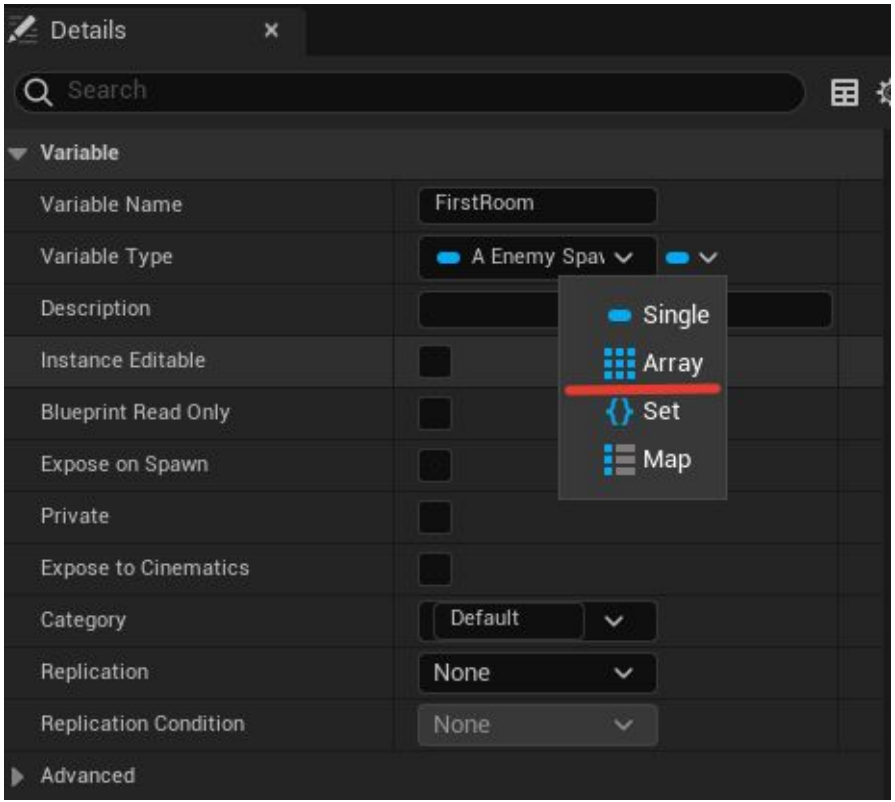
Оставим начальный индекс 0, а последний индекс зададим в отдельной переменной, которую назовём **Last Index**.

Логика работы цикла следующая – мы будет спаунить НПС с небольшой задержкой. Каждую итерацию спауна сохраняем в отдельную переменную. Когда спаунер отработает то число раз, которое мы укажем в переменной Last Index, он запустит спаун сразу на двух точках спауна. И так далее

Array

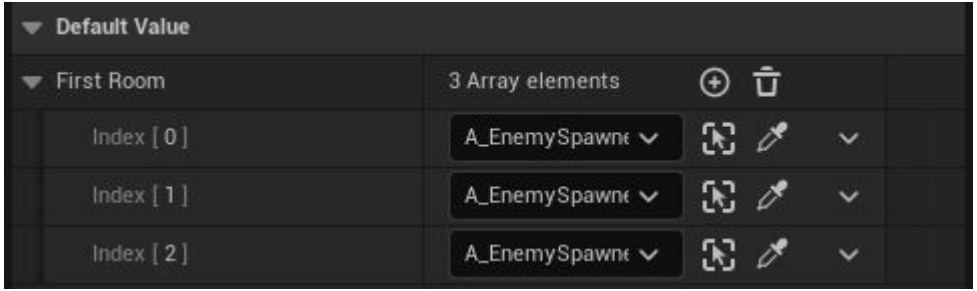
Нам понадобится переменная, типа массива указателей на расположенные на уровне спаунеры НПС. Чтобы её задать, создаём обычную переменную с типом A_EnemySpawner.

Далее в поле типа можно задать, будет ли она одиночной переменной, массивом и т.д.



Fill array

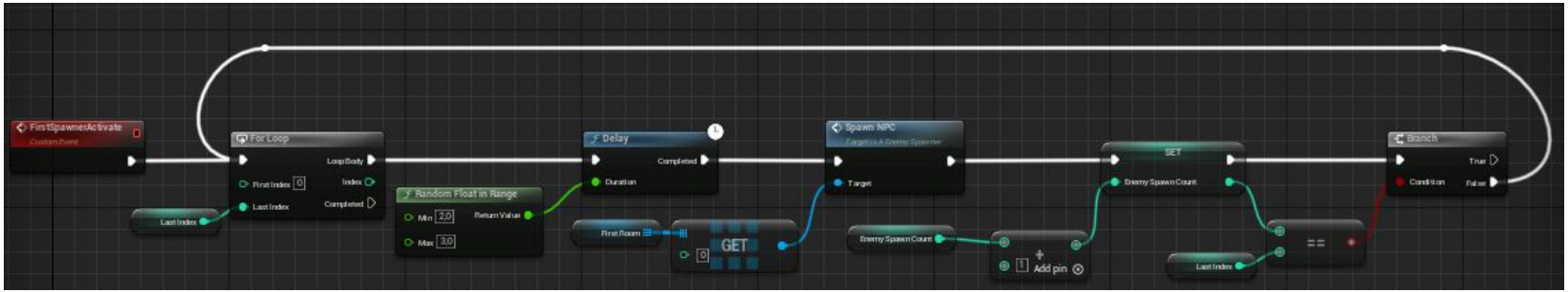
Заполняем наш массив данными спаунерами, которые расположены на уровне в первой комнате



Event Logic

В результате наша логика будет выглядеть следующим образом:

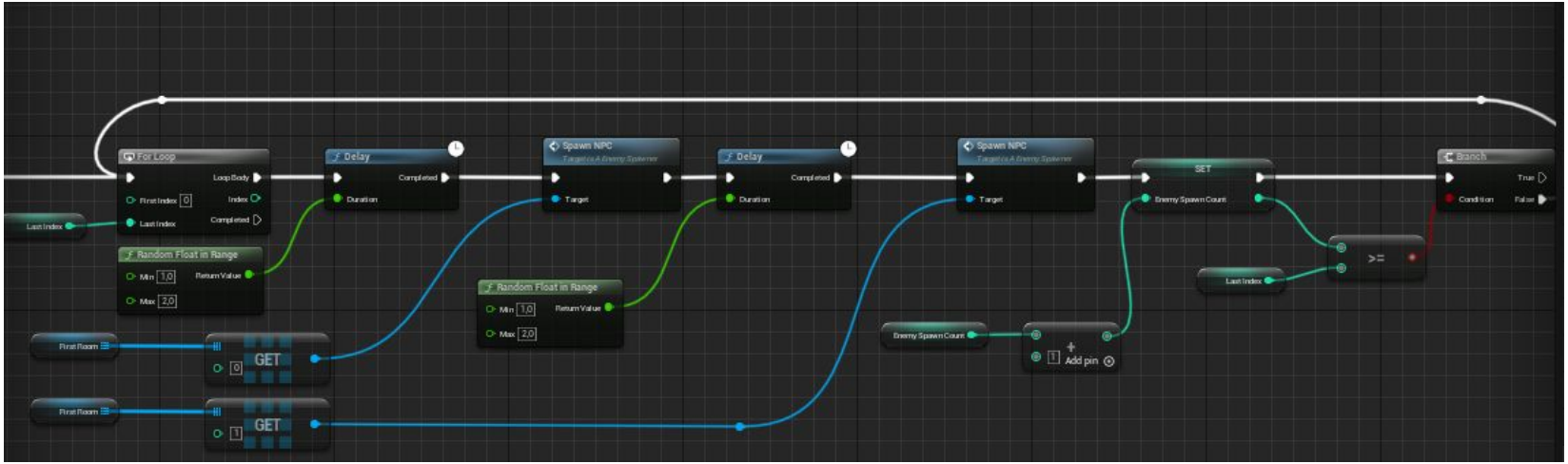
1. вызываем цикл
2. добавляем задержку, после которой обращаем к массиву со спаунерами и у его нулевого элемента вызываем функцию SpawnNPC
3. изменяем значение переменной EnemySpawnCount. Если оно не равно Last Index, повторно вызываем цикл



Custom Event

Создадим новое пользовательское событие, которое назовём **SecondSpawnerActivate**. Оно будет выполнять схожие действия. Отличие в том, что мы вновь обнуляем переменную EnemySpawnCount в самом начале, а в теле цикла – активируем спаун уже на двух точках.

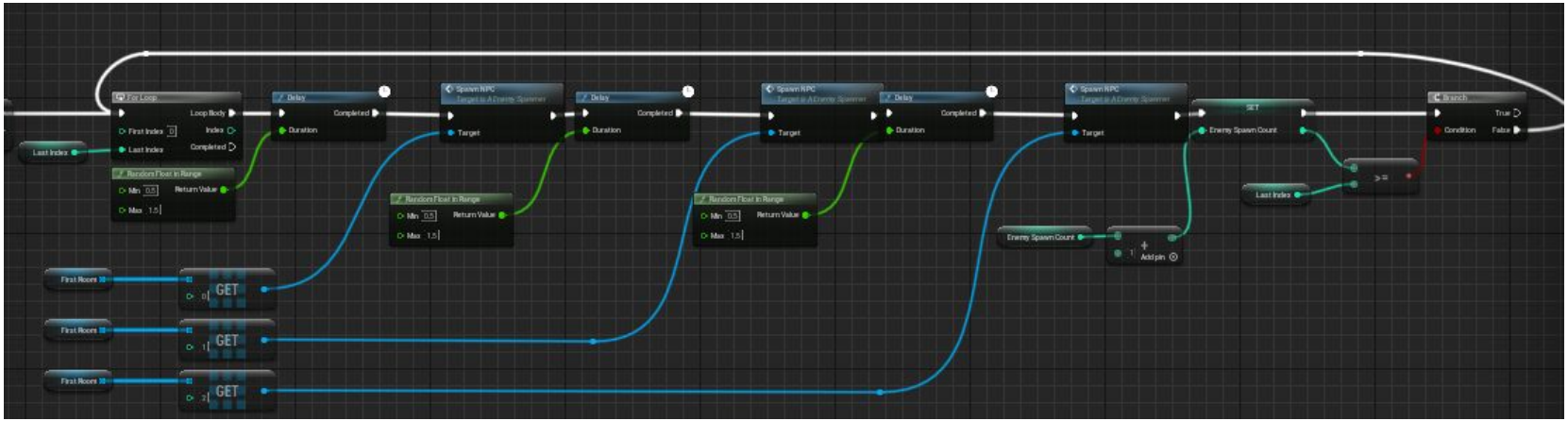
Также уменьшим время cooldown’а в функции Delay



Custom Event

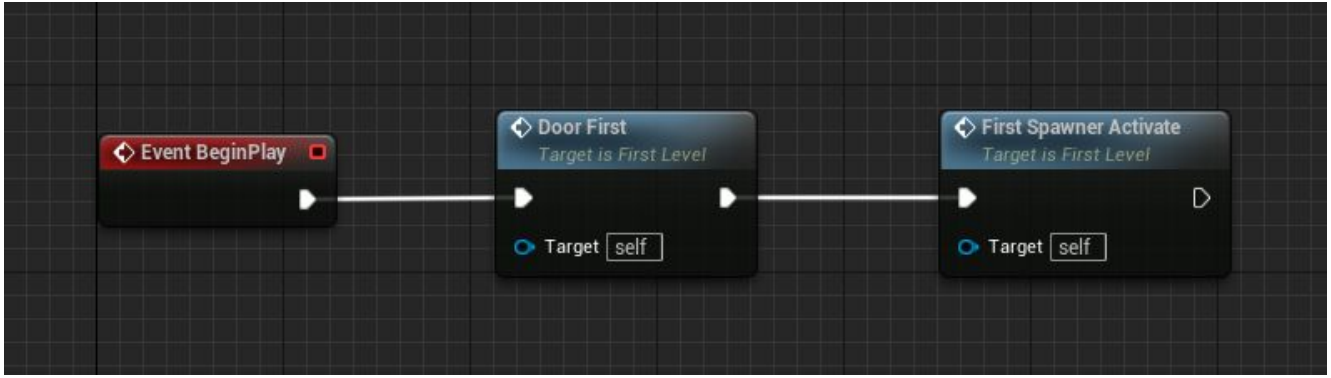
Создадим третье пользовательское событие, которое назовём **ThirdSpawnerActivate**. Оно будет выполнять схожие действия. Отличие в том, что мы вновь обнуляем переменную EnemySpawnCount в самом начале, а в теле цикла – активируем спаун уже на трёх точках.

Также вновь уменьшим время cooldown’a в функции Delay



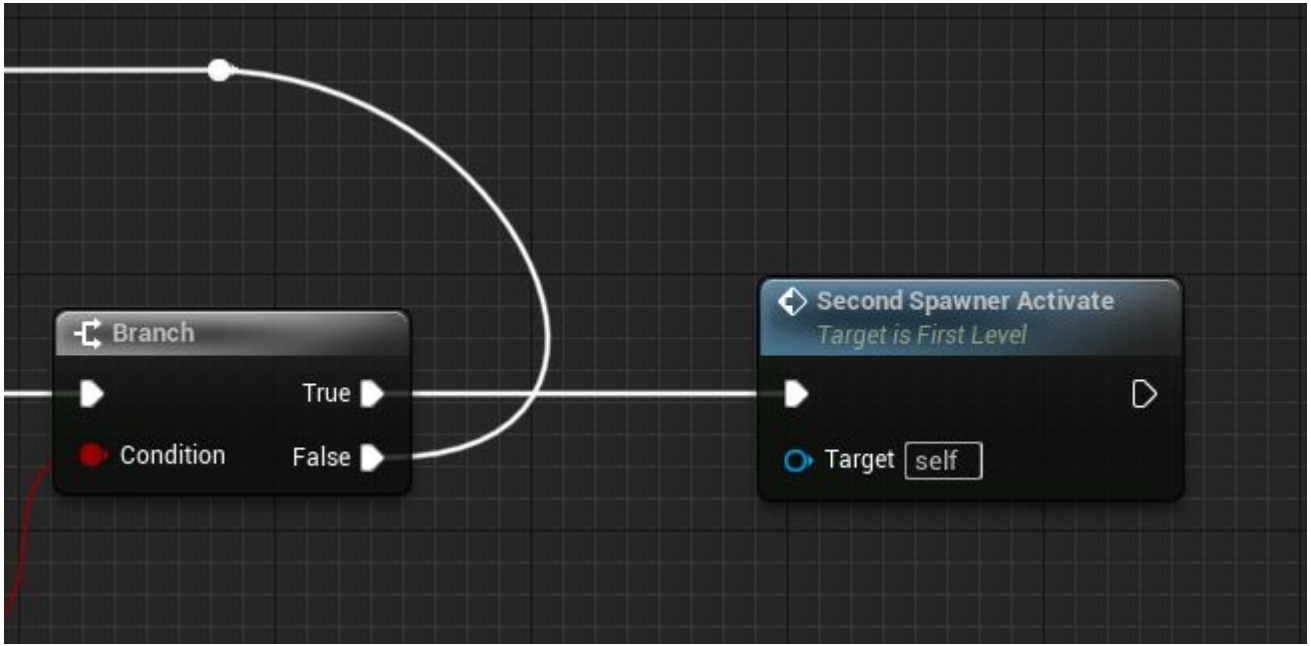
First Activate

Далее определимся с порядком вызова созданных событий. Будем вызывать FirstSpawnerActivate на Event BeginPlay сразу после открытия ворот



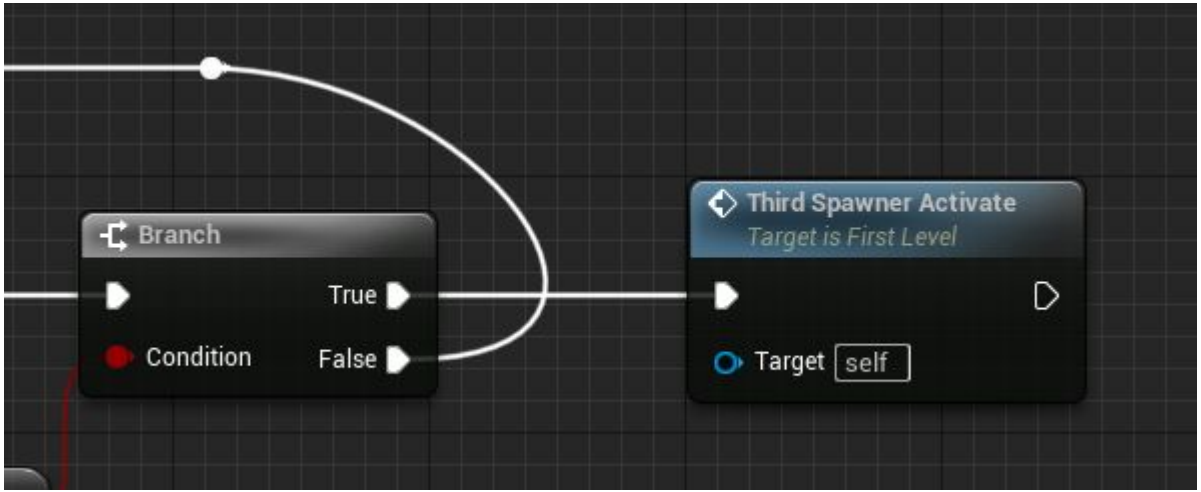
Second Activate

Вызываем SecondSpawnerActivate в конце логики FirstSpawnerActivate. В тот момент, когда наш цикл прекратит свою работу



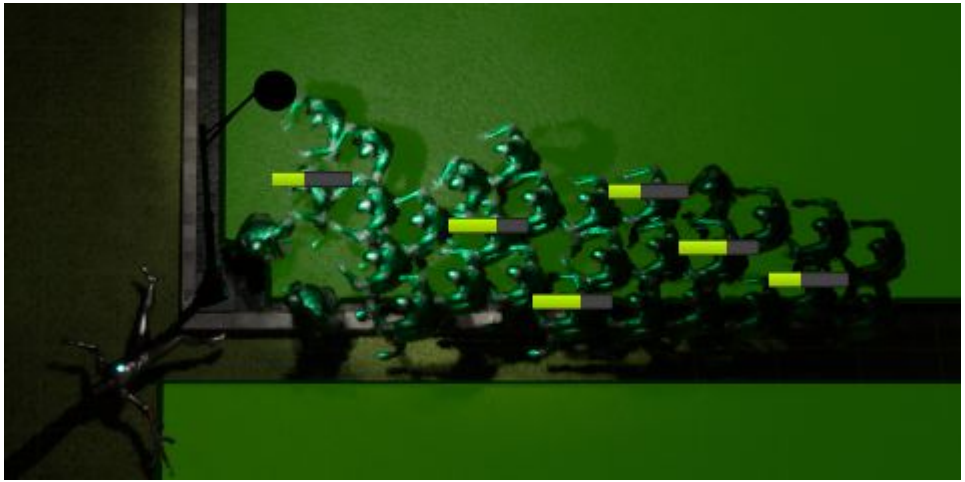
Third Activate

Вызываем ThirdSpawnerActivate в конце логики SecondSpawnerActivate. В тот момент, когда наш цикл прекратит свою работу

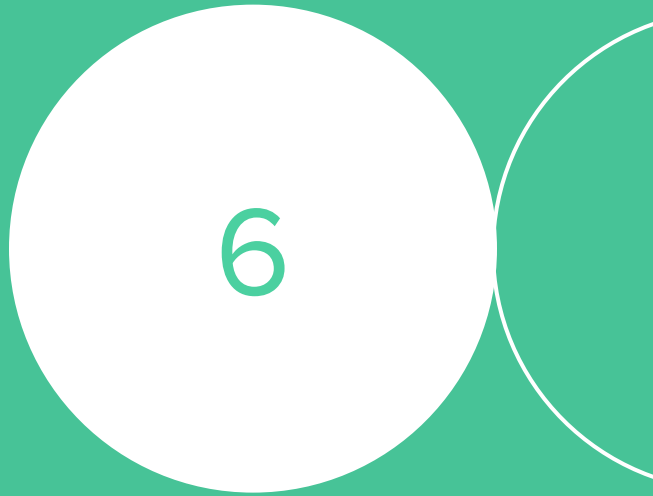


Итоги

В результате, если запустить нашу игру, нас будет атаковать всё больше и больше врагов. Со временем это станет совсем сложно



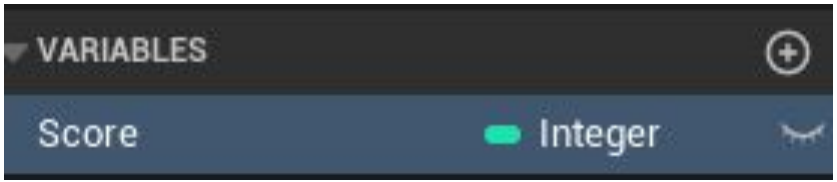
Подсчёт очков



Score

На заключительном этапе сделаем так, чтобы наша игра имела какое-то статистическое отображение. После смерти каждого НПС мы будет предоставлять игроку дополнительные очки, которые будут в результате суммироваться.

Для начала заходим в нашего игрока CH_Player и добавляем ему новую целочисленную переменную, которую назовём Score

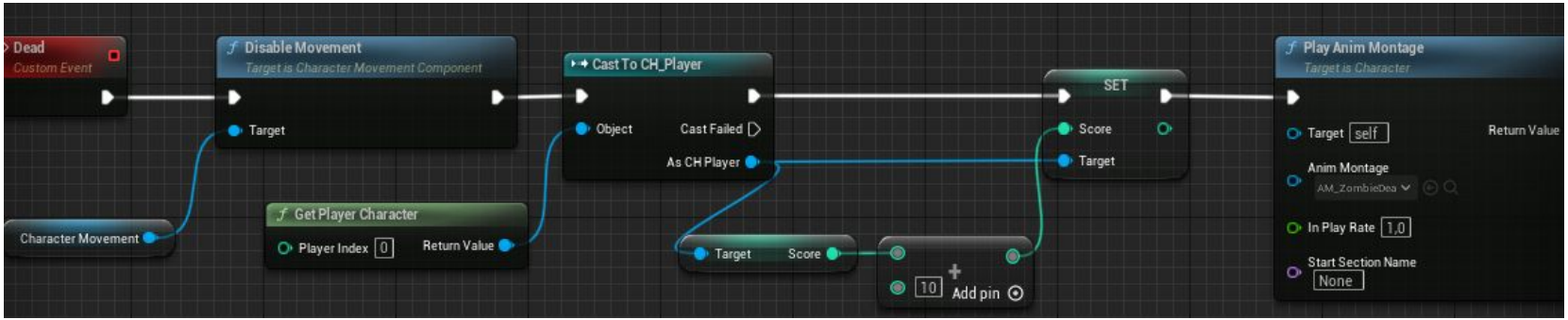


Refactoring

Далее зададим условия, после которых начисляются баллы для игрока.

В нашем проекте событие смерти НПС расположено в классе CH_Enemy. Обратимся к событию Dead и расширим его.

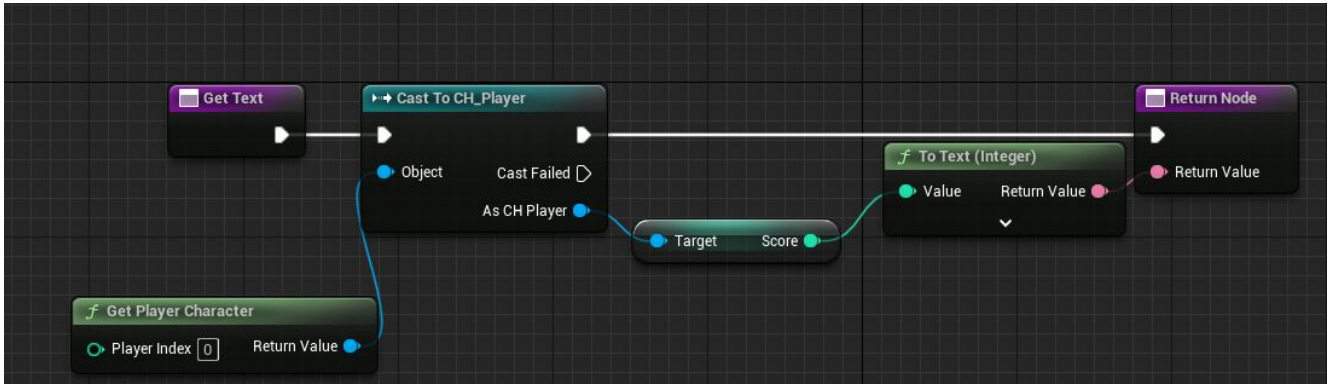
Перед вызовом логики проигрывания анимационного монтажа, мы будет кастоваться на игрового персонажа и устанавливать ему новое значение переменной Score



User Interface

Воспользуемся готовым решением и продублируем ранее созданный пользовательский интерфейс WBP_Timer. Переименуем новый объект в WBP_Score. Изменим его цветовое наполнение и изменим его размер.

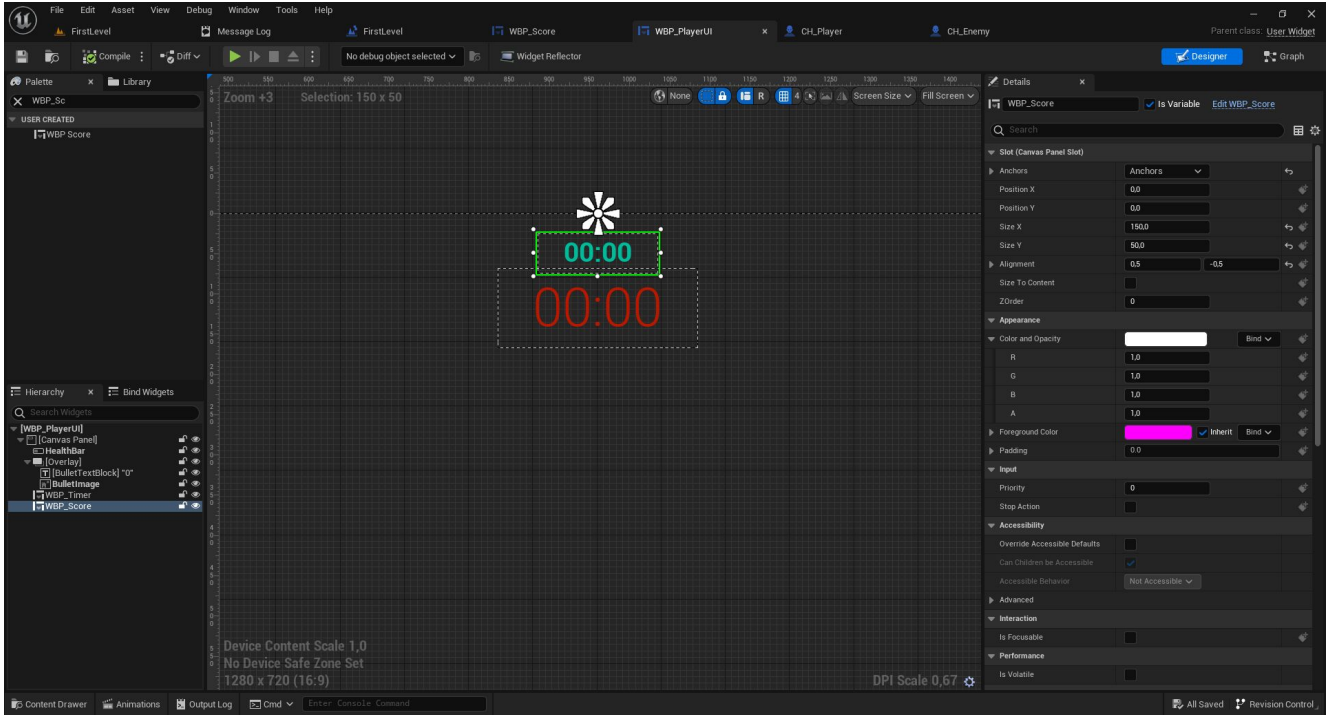
Далее изменим бинд функции GetText



Логика уже известна: мы обращаемся к игровому персонажу и передаём значение его переменной в выходную ноду

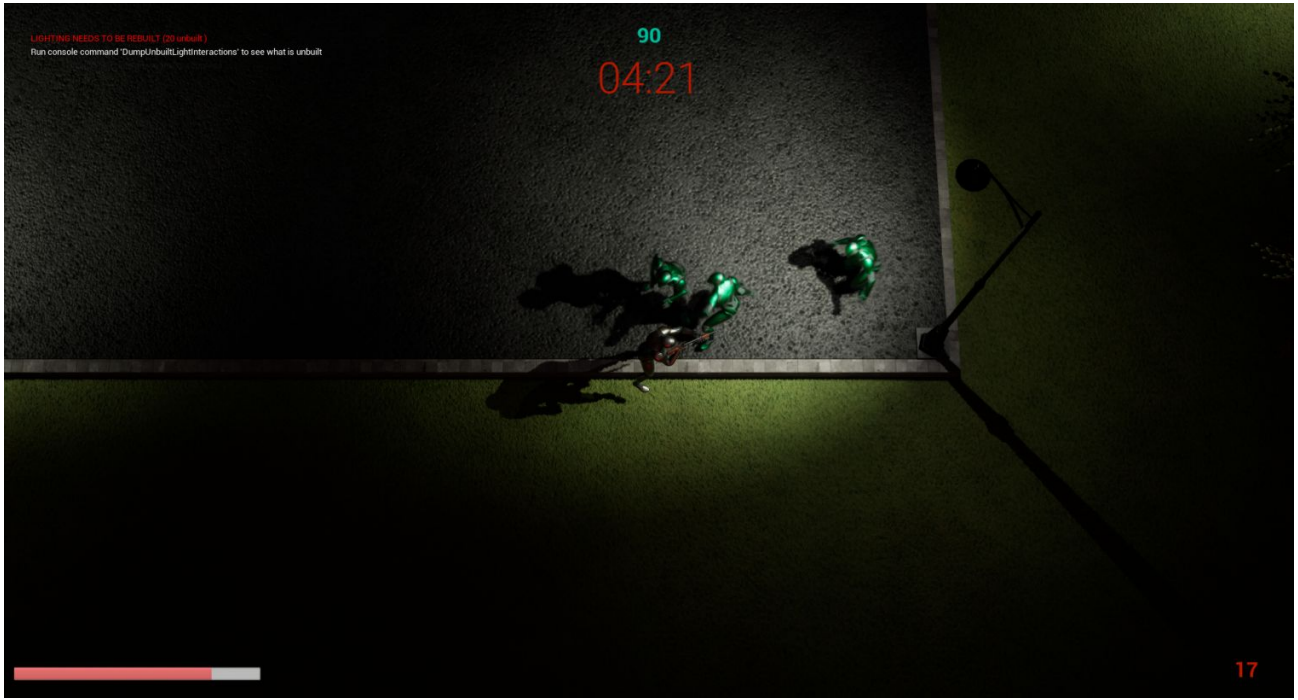
Widget settings

Осталось перейти в пользовательский интерфейс WBP_PlayerUI и добавить новый компонент. Настроим его расположение на панели



Итоги

В результате, если мы запустим игру и уничтожим несколько противников, начнём набирать очки



Итоги

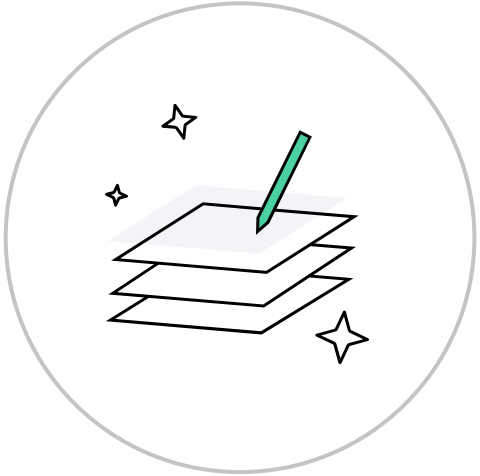
Итоги занятия

- Реализовали получение урона игроку от неигрового персонажа
- Научили нашего игрока уничтожать противника
- Добавили механику спауна НПС
- Сформировали небольшой алгоритм развития игры, расширив его условия
- Разработали виджет отображения очков

Домашнее задание

Давайте посмотрим ваше домашнее задание.

1. Вопросы по домашней работе задавайте в чате группы
2. Задачи можно сдавать по частям
3. Зачёт по домашней работе ставят после того, как приняты все задачи



Задавайте вопросы и пишите отзыв о лекции

Владислав Панченко
Unreal Engine разработчик в Sperasoft