Туториал по Unreal Engine. Часть 10: Как создать простой FPS

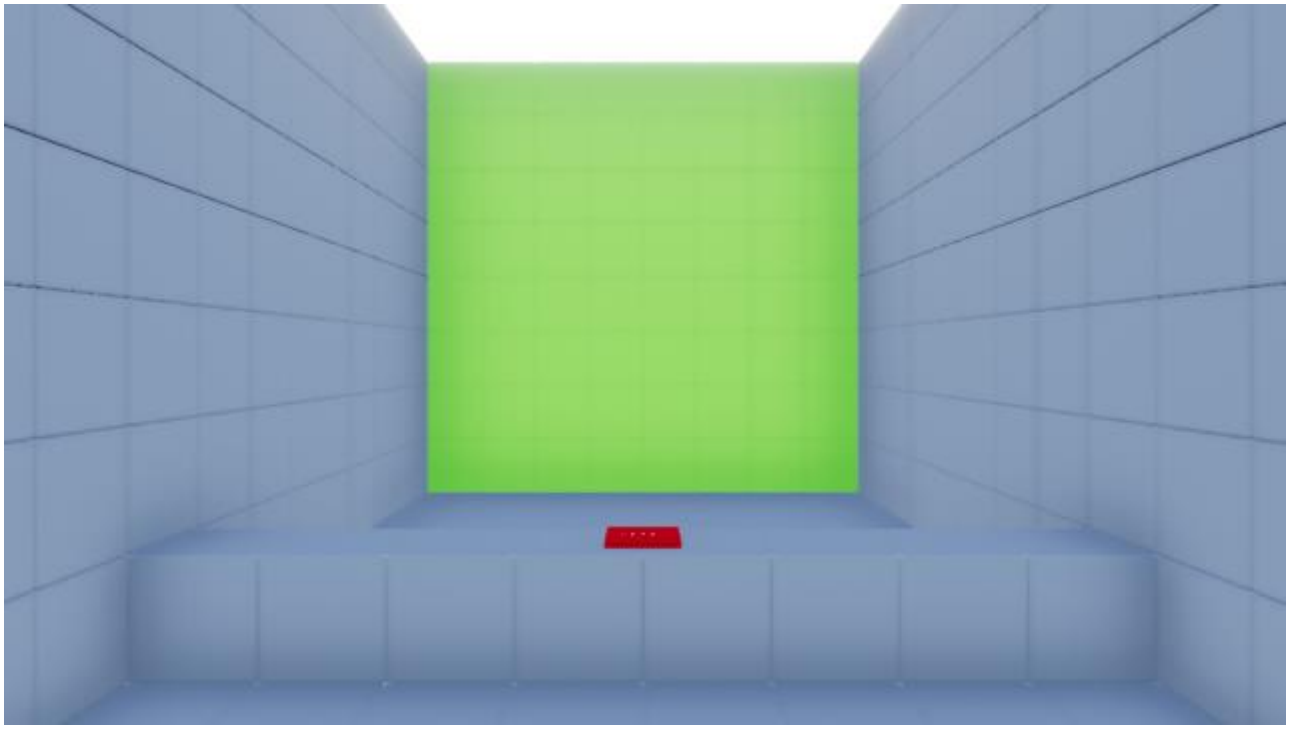
Шутер от первого лица (first-person shooter, FPS) — это жанр, в котором игрок использует оружие и смотрит на мир глазами персонажа. FPS-игры чрезвычайно популярны, что видно по успеху таких франшиз, как *Call of Duty* и *Battlefield*.

Unreal Engine изначально был создан для разработки FPS, поэтому вполне логично использовать его для создания такой игры. В этом туториале вы научитесь следующему:

- Создавать Pawn с видом от первого лица, который сможет двигаться и осматриваться вокруг
- Создавать оружие и привязывать его к Pawn игрока
- Стрелять пулями с помощью трассировки прямых (также известной как трассировка лучей)
- Наносить урон акторам

Приступаем к работе

Скачайте https://koenig-media.raywenderlich.com/uploads/2018/01/BlockBreakerStarter.zip и распакуйте её. Перейдите в папку проекта и откройте *BlockBreaker.uproject*. Вы увидите следующую сцену:

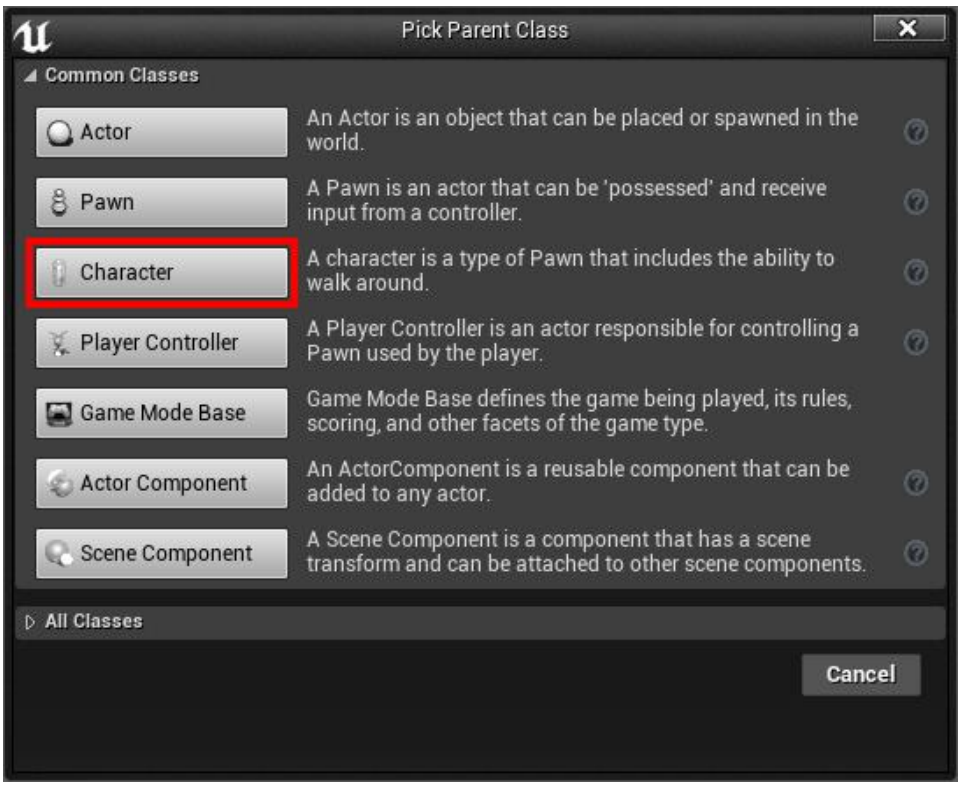


Зелёная стена состоит из множества целей. Когда им наносят урон, они становятся красными. Когда их здоровье достигает нуля, они исчезают. Красная кнопка заново устанавливает все цели.

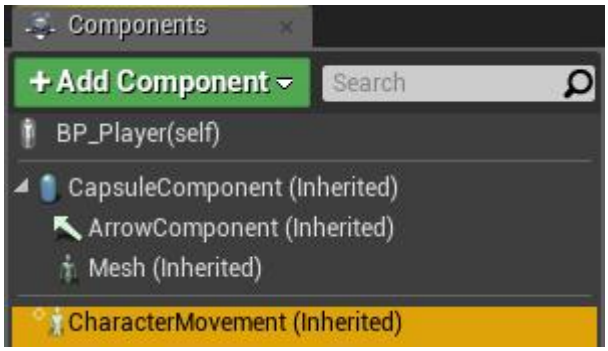
Для начала нужно создать Pawn игрока.

Создание Pawn игрока

Перейдите в папку *Blueprints* и создайте новый *Blueprint Class*. Выберите в качестве родительского класса *Character* и назовите его *BP_Player*.



Character — это разновидность Pawn, но с дополнительным функционалом, например, с компонентом *CharacterMovement*.



Этот компонент автоматически обрабатывает движение, например, ходьбу и прыжки. Мы просто вызываем соответствующую функцию, благодаря чему Pawn перемещается. В этом компоненте также можно задать переменные, такие как скорость ходьбы и прыжков.

Чтобы заставить Pawn двигаться, нам нужно знать, когда игрок нажимает клавишу движения. Для этого мы привяжем движение к клавишам *W*, *A*, *S* и *D*.

Примечание: если вы незнакомы с привязками, то можете прочитать о них в части туториала, посвящённой Blueprints. Привязкой клавиш мы определяем, какие клавиши будут выполнять действие.

Создание привязок движения

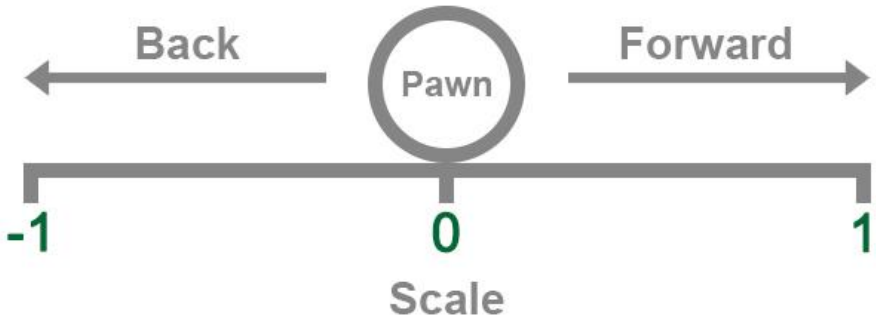
Выберите *Edit\Project Settings* и откройте настройки *Input*.

Создайте два *Axis Mappings* под названием *MoveForward* и *MoveRight*. MoveForward будет управлять движением вперёд и назад. MoveRight — движением влево и вправо.

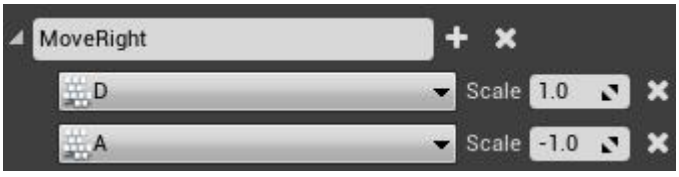
Для *MoveForward* замените клавишу на *W*. После этого создайте ещё одну клавишу и выберите *S*. Измените *Scale* для *S* на *-1.0*.

Примечание: если вы хотите подробнее узнать о поле *Scale*, прочитайте часть туториала, посвящённую Blueprints. В разделе "*Значение оси и масштаб ввода*" говорится, что это и как этим пользоваться.

Позже мы будем умножать значение масштаба на вектор forward Pawn. Это даст нам вектор, направленный *вперёд* при *положительном* масштабе. Если масштаб *отрицателен*, то вектор будет направлен *назад*. С помощью получившегося вектора можно будет двигать Pawn вперёд и назад.



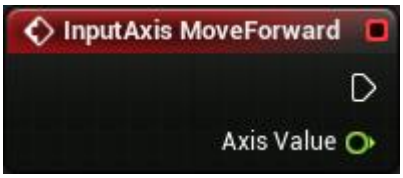
Теперь нам нужно сделать то же для движения влево и вправо. Измените клавишу для *MoveRight* на *D*. Затем создайте новую клавишу и выберите для неё *A*. Измените *Scale* для *A* на *-1.0*.



Теперь, когда мы настроили привязки, нам нужно использовать их для движения.

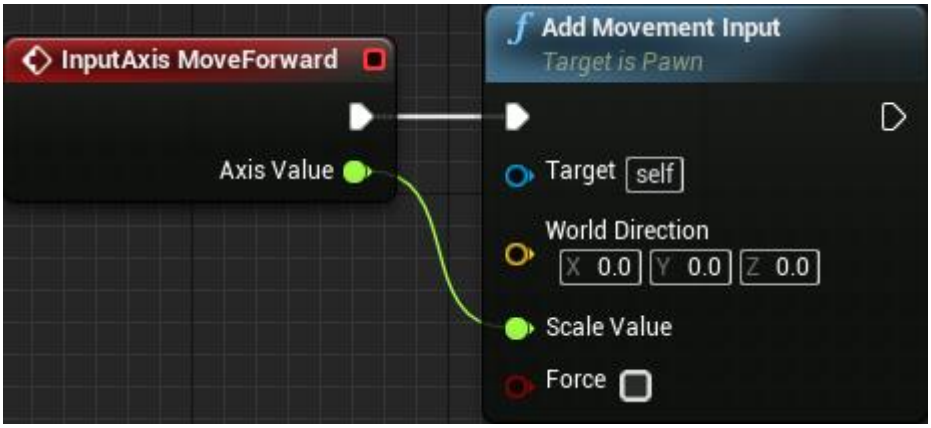
Реализация движения

Откройте *BP_Player*, а затем откройте Event Graph. Добавьте событие *MoveForward* (то, которое указано в списке *Axis Events*). Это событие будет выполняться в каждом кадре, даже если вы ничего не нажимаете.



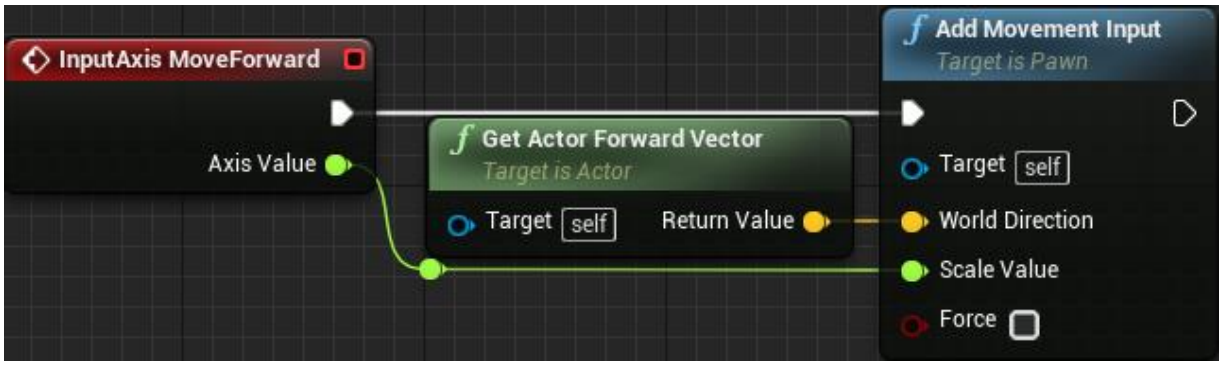
Также оно будет подавать на выход значение *Axis Value*, которое равно заданным ранее значениям *Scale*. Оно будет подавать на выход *1* при нажатии на *W* и *-1* при нажатии на *S*. Если не нажимать клавиши, то на выходе будет *0*.

Далее нужно приказать Pawn двигаться. Добавьте *Add Movement Input* и соедините его следующим образом:



Add Movement Input будет получать вектор и умножать его на *Scale Value*. Это преобразует его в соответствующем направлении. Поскольку мы используем *Character*, то компонент *CharacterMovement* будет перемещать Pawn в этом направлении.

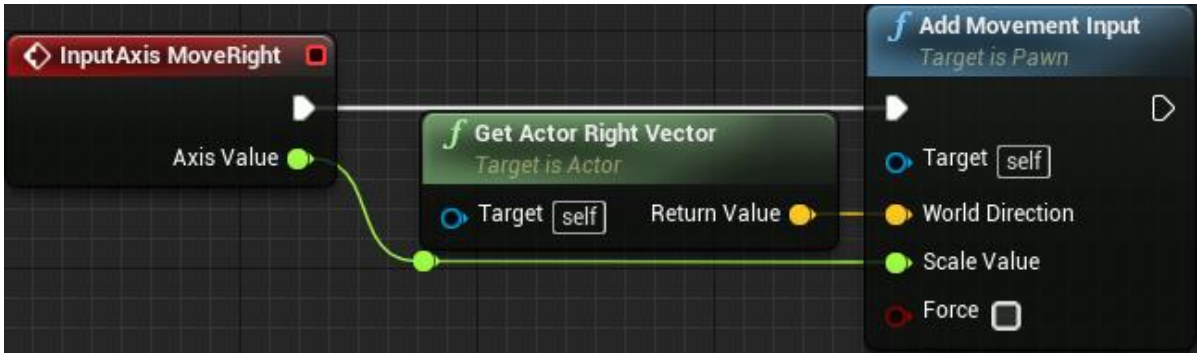
Теперь нам нужно указать направление движения. Так как мы хотим двигаться вперёд, то можем использовать *Get Actor Forward Vector*. При этом будет возвращаться вектор, направленный вперёд. Создайте его и соедините следующим образом:



Подведём итог:

1. *MoveForward* выполняется каждый кадр и передаёт на выход *Axis Value*. Это значение будет равно *1* при нажатии на *W* и *-1* при нажатии на *S*. Если не нажимать ни одну из этих клавиш, то на выходе будет *0*.
2. *Add Movement Input* умножает *вектор forward* Pawn на *Scale Value*. Благодаря этому в зависимости от нажатой клавиши вектор будет направлен вперёд или назад. Если не нажимать клавиши, то вектор не будет иметь направления, то есть Pawn не будет двигаться.
3. Компонент *CharacterMovement* получает результат из *Add Movement Input*, после чего двигает Pawn в этом направлении.

Повторим процесс для *MoveRight*, но заменим *Get Actor Forward Vector* на *Get Actor Right Vector*.



Для проверки движения нужно задать Pawn по умолчанию в игровом режиме.

Задание Pawn по умолчанию

Нажмите на *Compile* и вернитесь в основной редактор. Откройте панель *World Settings* и найдите раздел *Game Mode*. Измените значение *Default Pawn Class* на *BP_Player*.

Примечание: если у вас нет панели World Settings, то перейдите в Toolbar и выберите *Settings\World Settings*.

Теперь при запуске игры вы автоматически будете использовать *BP_Player*. Нажмите на *Play* и воспользуйтесь клавишами *W, A, S* и *D* для движения по уровню.

Теперь мы создадим привязки для поворота головы.

Создание привязок обзора

Откройте *Project Settings*. Создайте ещё два *Axis Mappings* под названием *LookHorizontal* и *LookVertical*.

Замените клавишу для *LookHorizontal* на *Mouse X*.

Эта привязка будет выдавать положительное значение при перемещении мыши *вправо*, и наоборот.

Теперь изменим клавишу для *LookVertical* на *Mouse Y*.

Эта привязка будет выдавать положительное значение при перемещении мыши *вверх*, и наоборот.

Теперь нам нужно создать логику, чтобы смотреть вокруг.

Реализация обзора

Если у Pawn нет компонента *Camera*, то Unreal автоматически создаёт камеру за вас. По умолчанию, камера будет использовать поворот *контроллера*.

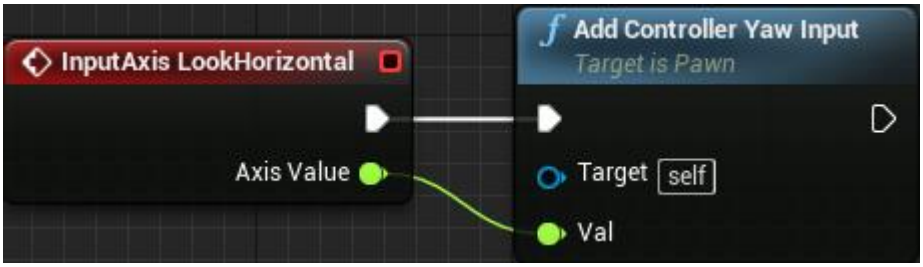
Примечание: если вы хотите узнать больше о контроллерах, то изучите наш туториал "Искусственный интеллект".

Несмотря на то, что контроллеры не являются физическими, у них всё равно есть свой поворот. Это значит, что можно направить взгляд Pawn и камеры в разных направлениях. Например, в игре от третьего лица персонаж и камера не всегда смотрят в одном направлении.

Для поворота камеры в игре от первого лица нам достаточно изменить поворот контроллерая.

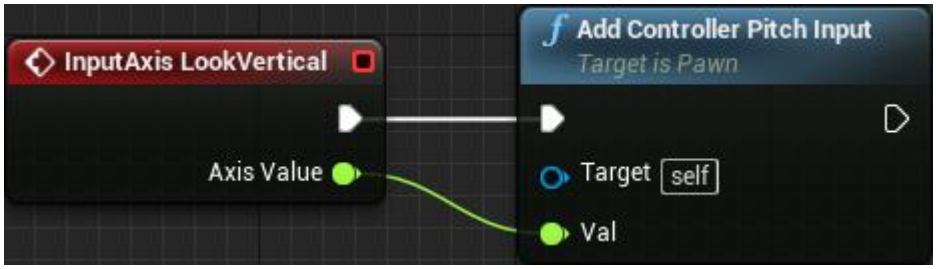
Откройте *BP_Player* и создайте событие *LookHorizontal*.

Чтобы заставить камеру поворачиваться влево или вправо, нам нужно регулировать *рыскание (yaw)*. Создайте *Add Controller Yaw Input* и соедините его следующим образом:



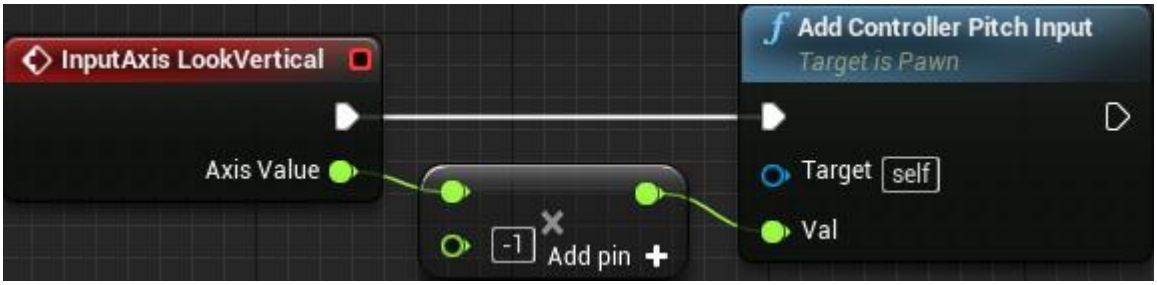
Теперь при горизонтальном движении мыши контроллер будет поворачиваться влево или вправо. Поскольку камера использует поворот контроллера, она тоже будет поворачиваться.

Повторите процесс для *LookVertical*, но замените *Add Controller Yaw Input* на *Add Controller Pitch Input*.



Если запустите игру сейчас, то заметите, что вертикальное движение камеры *инвертировано*. Это значит, что при движении мыши *вверх* камера будет смотреть *вниз*.

Если вы предпочитаете неинвертированное управление, то умножьте *Axis Value* на *-1*. Это инвертирует *Axis Value*, что инвертирует наклон контроллера.



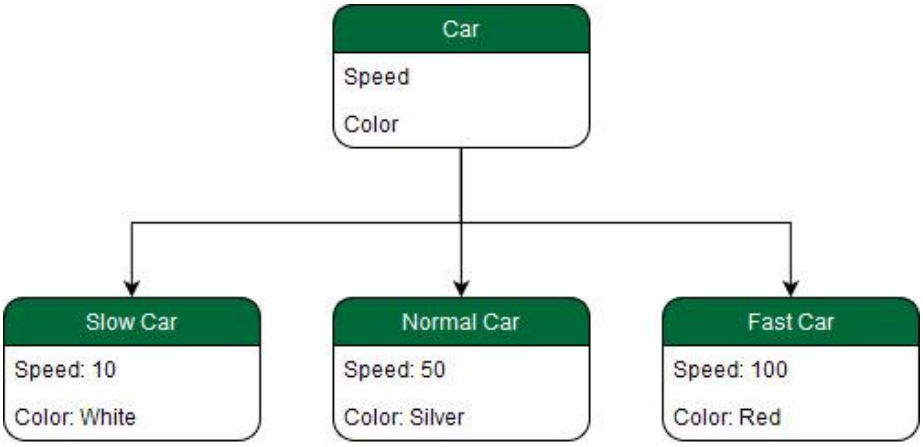
Нажмите на *Compile* и нажмите *Play*. Оглянитесь вокруг с помощью мыши.

Теперь, когда всё движение и обзор готовы, пришла пора создавать оружие!

Создание оружия

Помните, что при создании Blueprint Class можно выбрать родительский класс? Можно так же выбирать в качестве родительских собственные Blueprints. Это полезно, когда имеются разные типы объектов, обладающие общим функционалом или атрибутами.

Допустим, нам нужно создать разные типы автомобилей. Можно создать базовый класс автомобиля, содержащий такие переменные, как скорость и цвет. Затем можно создать классы (дочерние), которые будут использовать в качестве родительского базовый класс автомобиля. Каждый дочерний класс будет содержать те же переменные. Теперь у вас есть простой способ для создания машин с разными значениями скорости и цвета.



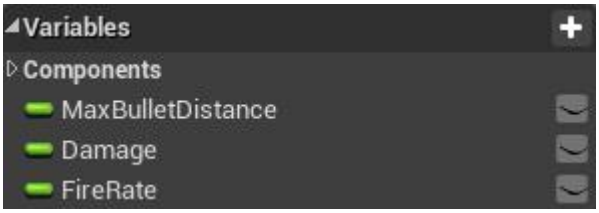
Этим же способом можно создавать оружие. Для этого необходимо для начала создать базовый класс.

Создание базового класса оружия

Вернитесь в основной редактор и создайте *Blueprint Class* типа *Actor*. Назовите его *BP_BaseGun* и откройте.

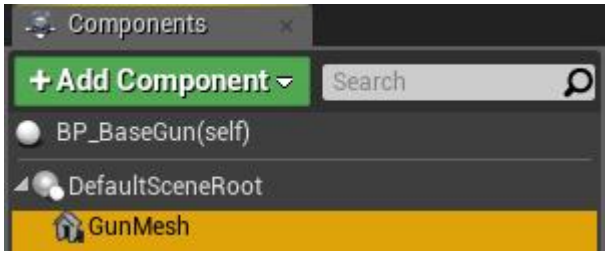
Теперь мы создадим несколько переменных, задающих свойства оружия. Создайте следующие переменные типа *float*:

- *MaxBulletDistance:* максимальная дальность полёта каждой пули
- *Damage:* количество урона, наносимого пулей при попадании в актора
- *FireRate:* промежуток (в секундах) между выстрелами пуль оружием



Примечание: по умолчанию значения всех переменных равны нулю, что вполне подходит для туториала. Однако если вы хотите, чтобы у новых классов оружия было другое значение по умолчанию, то его нужно задать в *BP_BaseGun*.

Теперь нам нужно физическое представление этого оружия. Добавьте компонент *Static Mesh* и назовите его *GunMesh*.



Пока не волнуйтесь о выборе статического меша. Мы займёмся этим в следующем разделе, где будем создавать дочерний класс оружия.

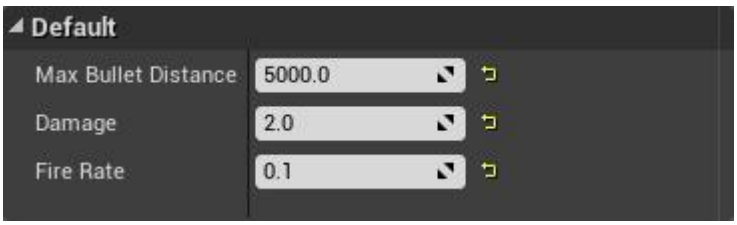
Создание дочернего класса оружия

Нажмите на *Compile* и вернитесь в основной редактор. Для создания дочернего класса *нажмите правой клавишей мыши* на *BP_BaseGun* и выберите *Create Child Blueprint Class*.

GIF

Назовите его *BP_Rifle* и откройте. Откройте *Class Defaults* и задайте переменным следующие значения:

- *MaxBulletDistance:* 5000
- *Damage:* 2
- *FireRate:* 0.1



Это значит, что максимальный путь каждой пули будет равен *5000*. Если она попадёт в актора, то нанесёт *2* единицы урона. При стрельбе очередями интервал перед каждым выстрелом будет равен не менее чем *0.1* секунды.

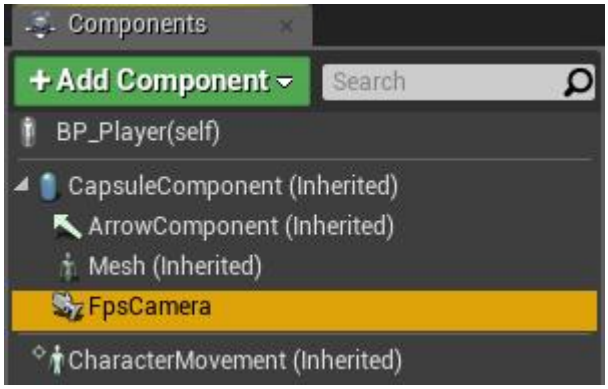
Теперь нам нужно указать меш, который будет использоваться оружием. Выберите компонент *GunMesh* и выберите для *Static Mesh* значение *SM_Rifle*.

Теперь оружие готово. Нажмите на *Compile* и закройте *BP_Rifle*.

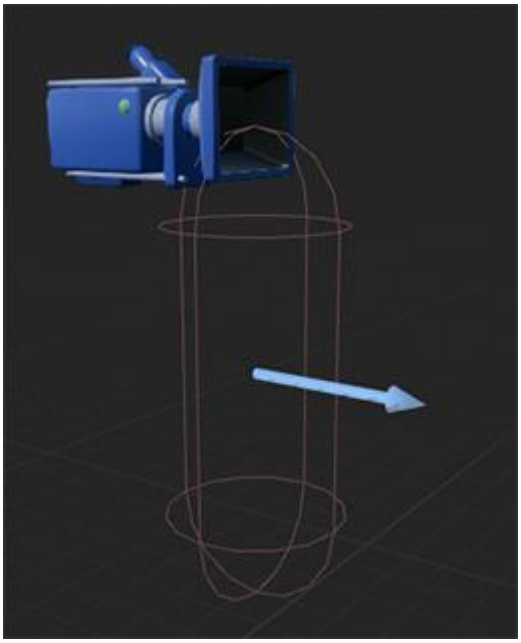
Теперь мы создадим собственный компонент камеры. Он даст нам более удобное управление расположением камеры. Также он позволит нам присоединить оружие к камере, чтобы оно всегда находилось перед камерой.

Создание камеры

Откройте *BP_Player* и создайте компонент *Camera*. Назовите его *FpsCamera*.

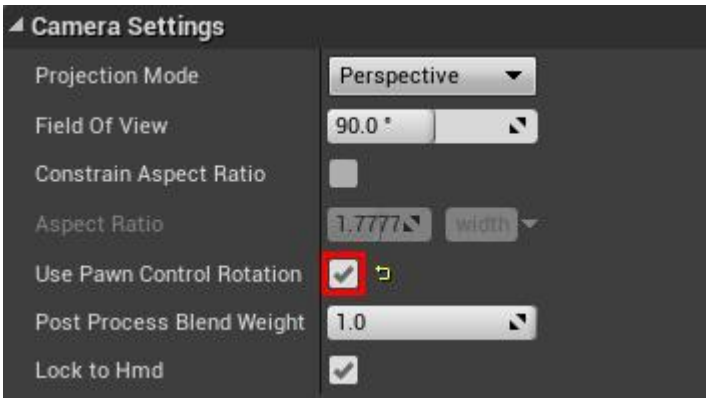


Позиция по умолчанию находится довольно низко, из-за чего игрок может почувствовать себя маленьким. Измените *location* камеры *FpsCamera* на *(0, 0, 90)*.



По умолчанию, компоненты Camera не используют поворот контроллера. Чтобы

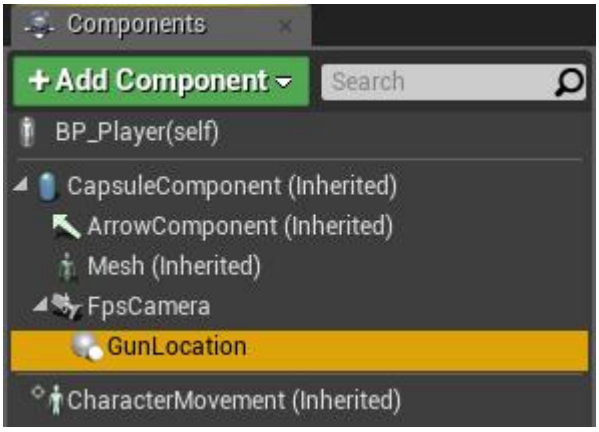
исправить это, перейдите в панель Details и включите *Camera Settings\Use Pawn Control Rotation*.



Теперь нам нужно задать место, в котором должно находиться оружие.

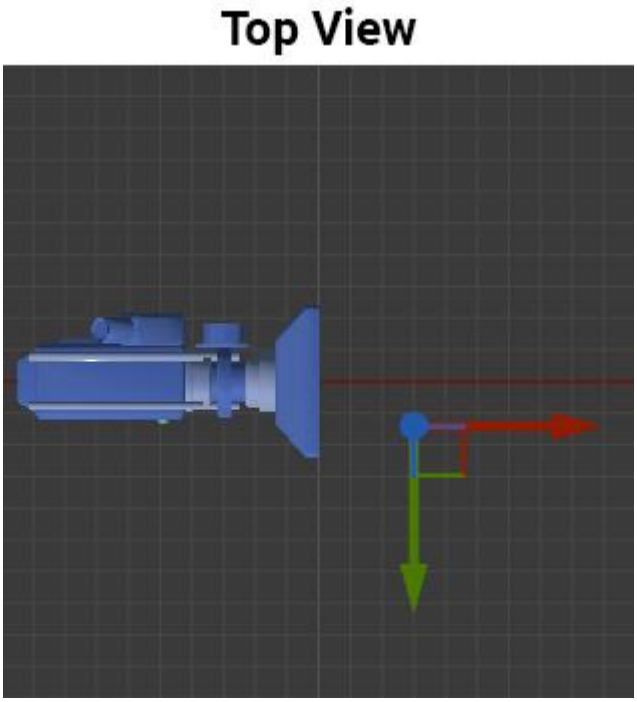
Задание местоположения оружия

Для создания местоположения оружия мы можем использовать компонент *Scene*. Эти компоненты идеально подходят для задания местоположений, потому что содержат только Transform. Выберите *FpsCamera* и создайте компонент *Scene*. Таким образом он прикрепится к камере. Назовите его *GunLocation*.

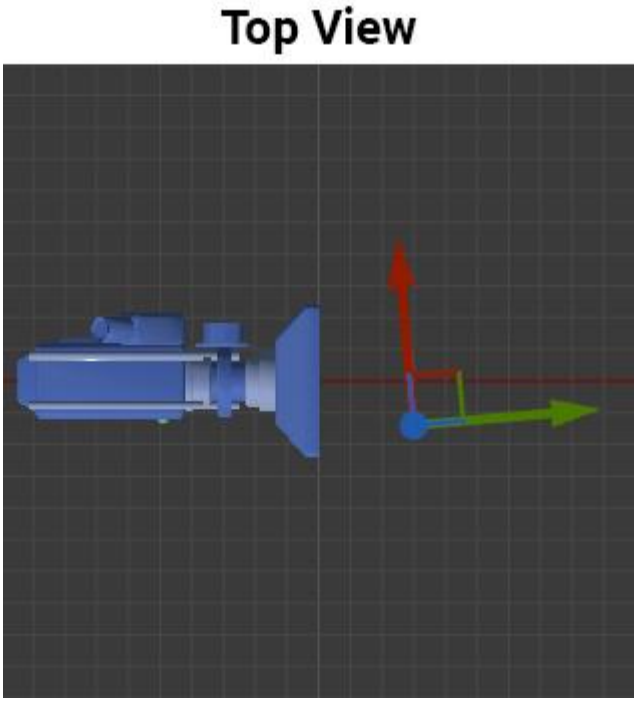


Благодаря тому, что мы прикрепили *GunLocation* к *FpsCamera*, оружие постоянно будет сохранять своё положение относительно камеры. Таким образом, мы всегда будем видеть оружие перед камерой.

Теперь присвоим *location* компонента *GunLocation* значения *(30, 14, -12)*. Так мы расположим оружие впереди и слегка сбоку от камеры.



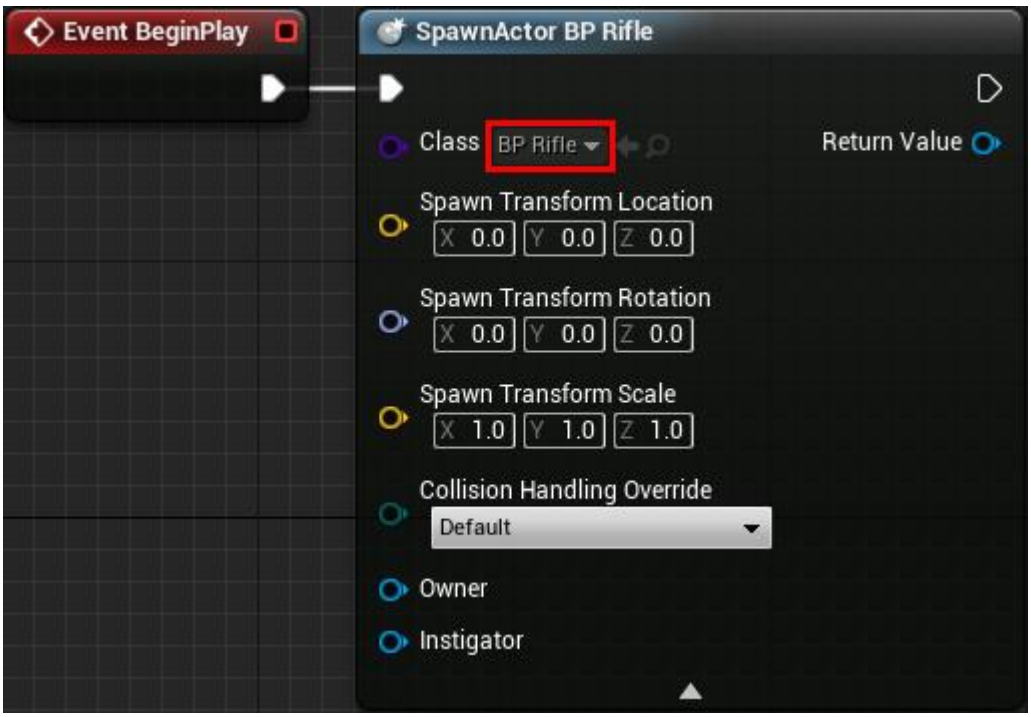
Затем зададим *rotation* значения *(0, 0, -95)*. Когда мы прикрепим оружие, то будет казаться, что оно всегда направлено в центр экрана.



Теперь нам нужно создать оружие и прикрепить его к *GunLocation*.

Создание и прикрепление оружия

Найдите *Event BeginPlay* и создайте *Spawn Actor From Class*. Выберите для *Class* значение *BP_Rifle*.

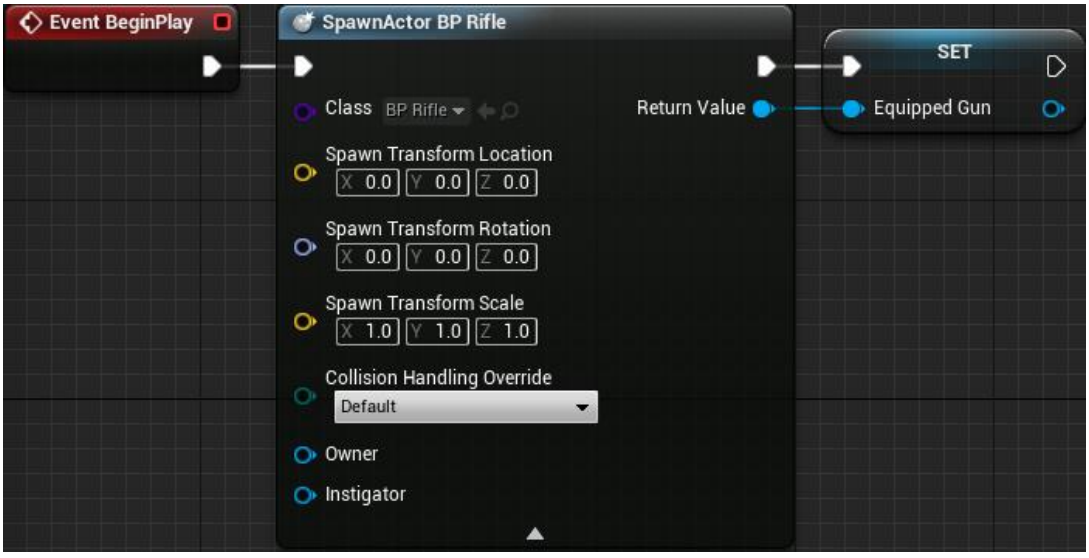


Поскольку нам нужно будет использовать оружие позже, мы сохраним его в переменной. Создайте переменную типа *BP_BaseGun* и назовите её *EquippedGun*.

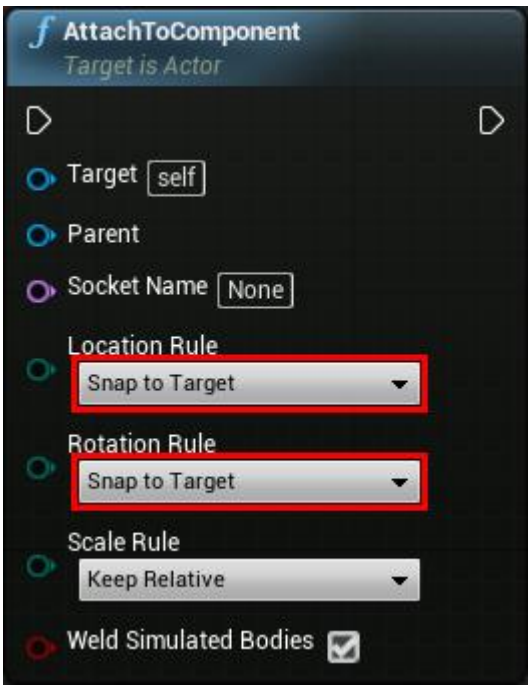
Важно то, что переменная *не* имеет тип *BP_Rifle*, потому что игрок может иметь разные типы оружия, а не только одно. Если создать другой тип оружия, то мы не сможем хранить его в переменной типа *BP_Rifle*. Это будет похоже на то, что мы пытаемся засунуть круг в прямоугольное отверстие.

Выбрав для переменной тип *BP_BaseGun*, мы создали большое «отверстие», подходящее под многие «фигуры».

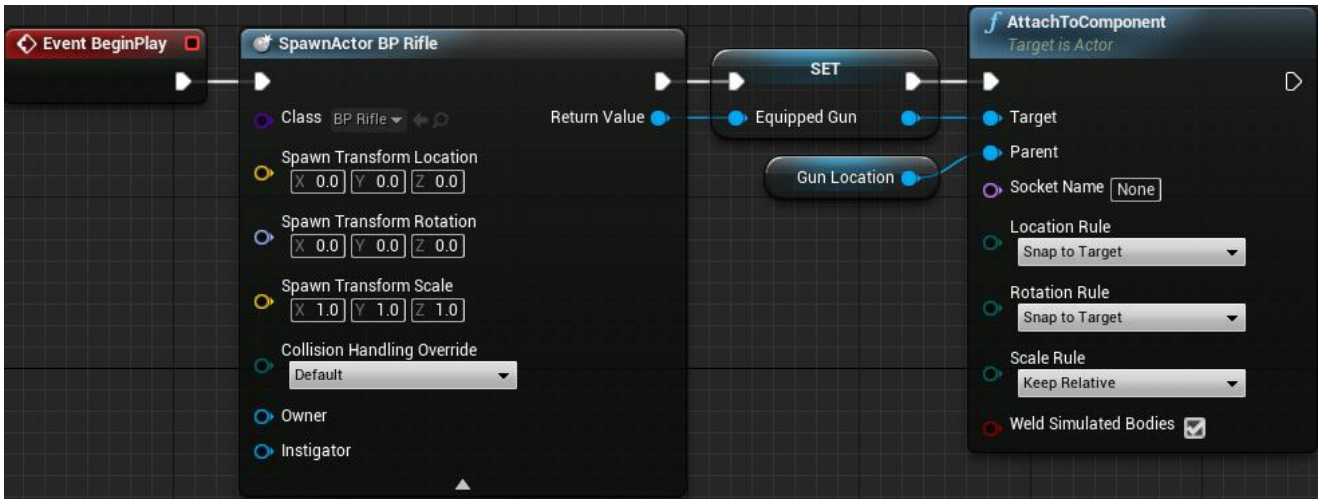
Теперь присвоим *EquippedGun* значение *Return Value Spawn Actor From Class*.



Чтобы прикрепить оружие, мы можем использовать *AttachToComponent*. Создайте его и задайте для *Location Rule* и *Rotation Rule* значение *Snap to Target*. Благодаря этому оружие будет иметь то же местоположение и поворот, что и родитель.



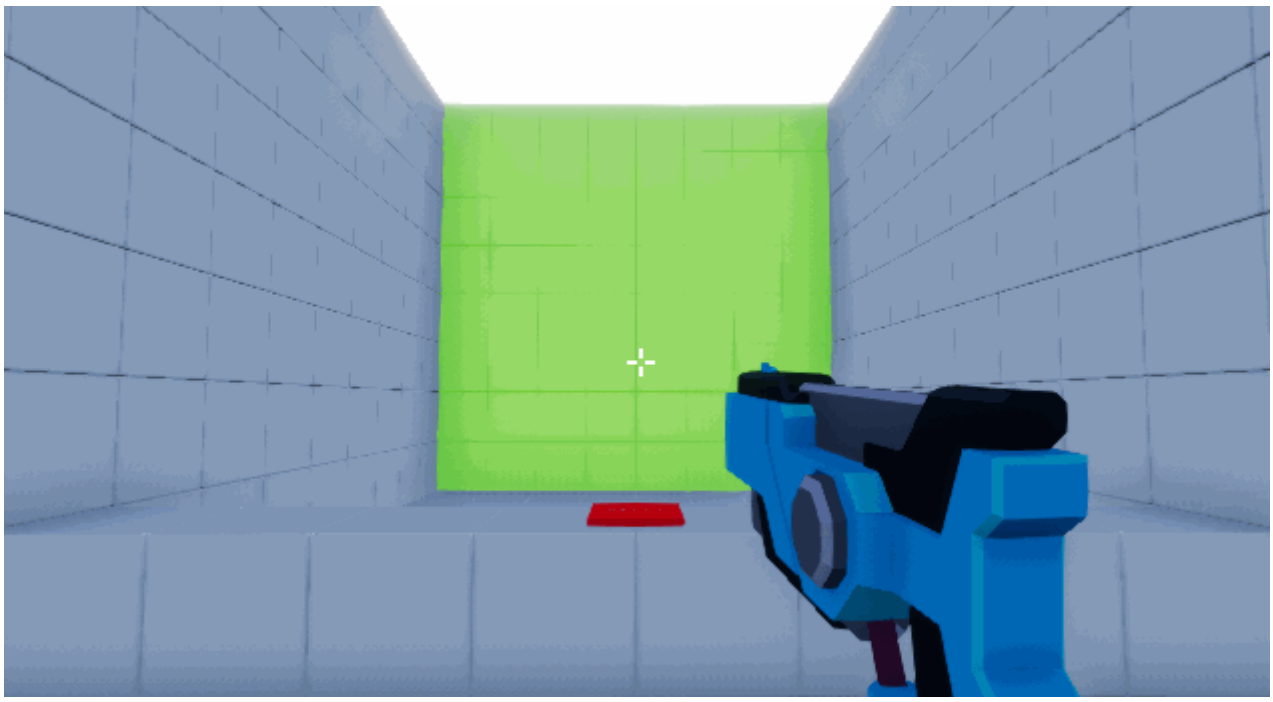
Теперь создадим ссылку на *GunLocation* и соединим всё следующим образом:



Подведём итог:

1. При создании *BP_Player* он будет создавать экземпляр *BP_Rifle*
2. *EquippedGun* будет хранить ссылку на созданный *BP_Rifle* для дальнейшего использования
3. *AttachToComponent* присоединяет оружие к *GunLocation*

Нажмите на *Compile* и нажмите *Play*. Теперь при создании игрока будет создаваться и оружие! При поворотах оружие будет всегда находиться перед камерой.



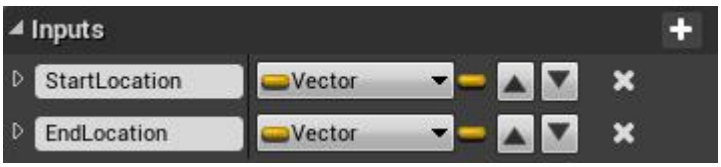
Теперь начинается интересное: мы будем стрелять! Чтобы проверить, попала ли куда-нибудь пуля, мы можем использовать *трассировку прямых*.

Стрельба пулями

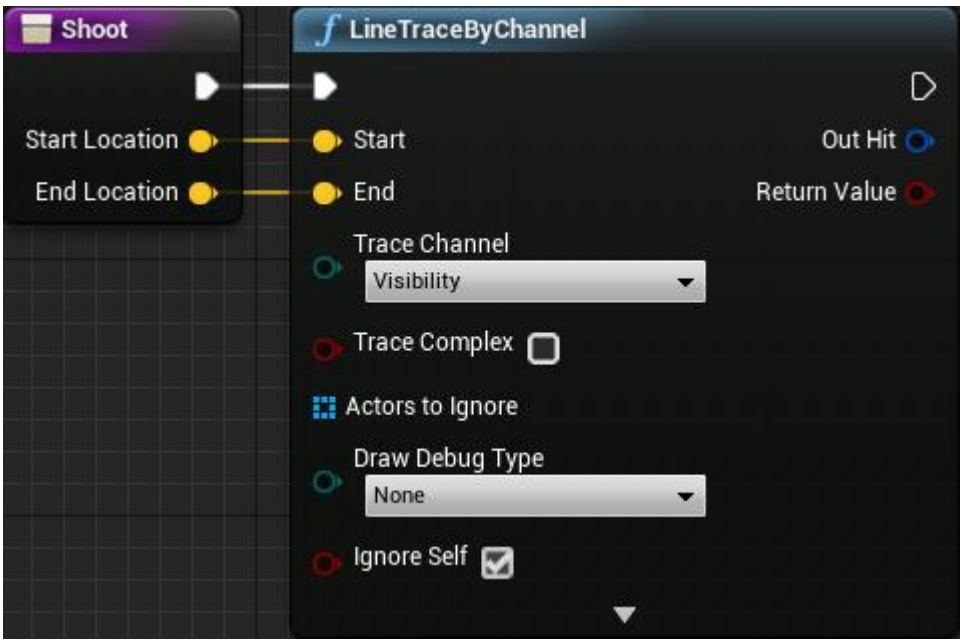
Трассировка прямых — это функция, получающая начальную и конечную точки (образующие прямую). Она проверяет каждую точку на прямой (от начала до конца), пока на что-нибудь не наткнётся. В играх для проверки попадания пули чаще всего используется такой способ.

Поскольку стрельба — это функция оружия, она должна выполняться в классе оружия, а не игрока. Откройте *BP_BaseGun* и создайте функцию *Shoot*.

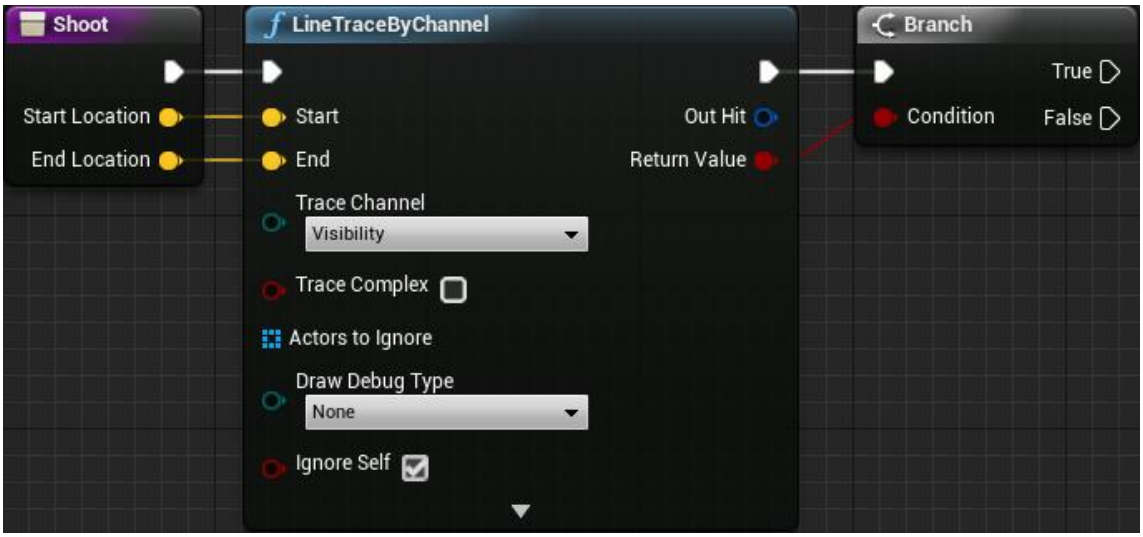
Затем создайте два входа *Vector* и назовите их *StartLocation* and *EndLocation*. Они будут начальной и конечной точками трассировки прямых (которые мы будем передавать из *BP_Player*).



Трассировку прямых можно выполнять с помощью *LineTraceByChannel*. Этот нод проверяет попадания с помощью канала коллизий *Visibility* или *Camera*. Создайте его и соедините следующим образом:



Теперь нам нужно проверить, попадает ли во что-нибудь трассировка прямых. Создайте *Branch* и соедините его так:

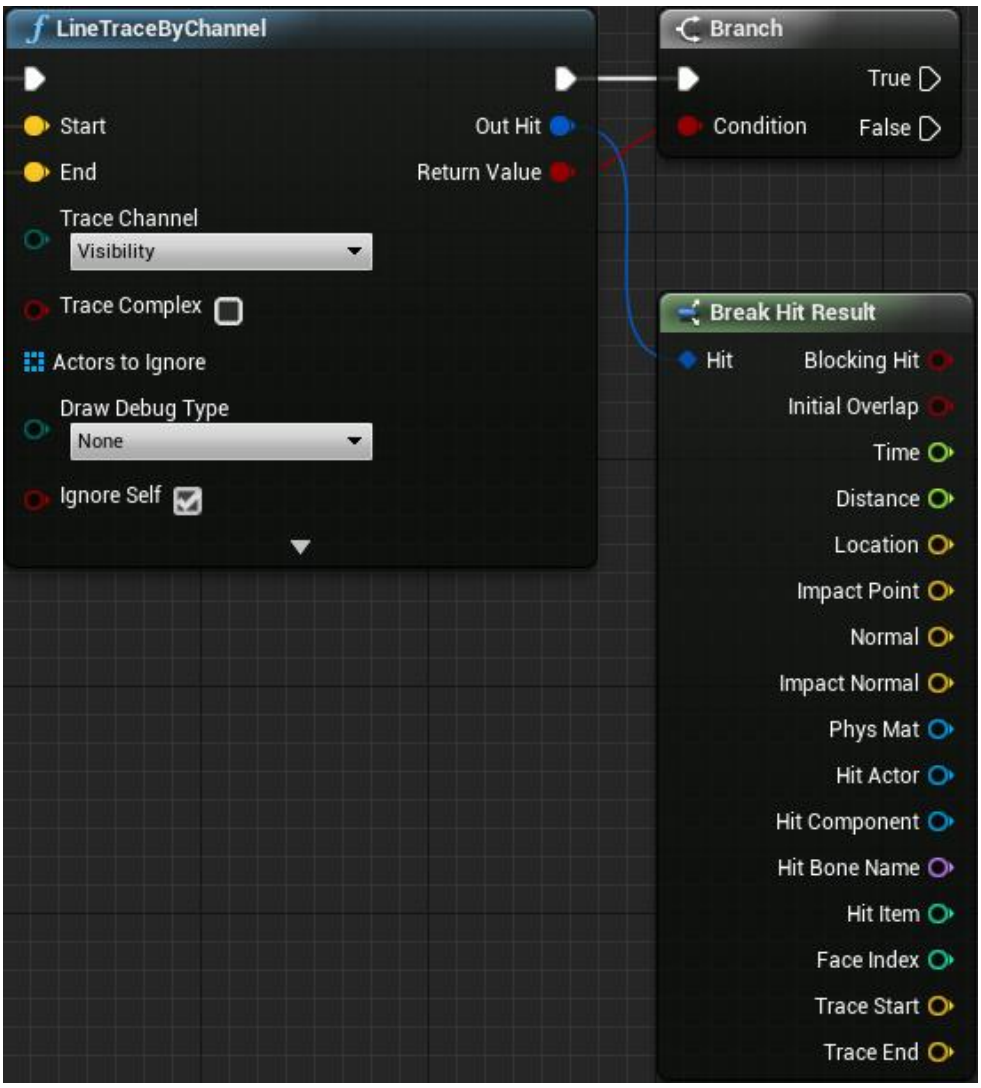


При попадании *Return Value* будет выдавать на выход *true*, и наоборот.

Чтобы дать игроку наглядную обратную связь о том, куда попала пуля, можно использовать эффект частиц.

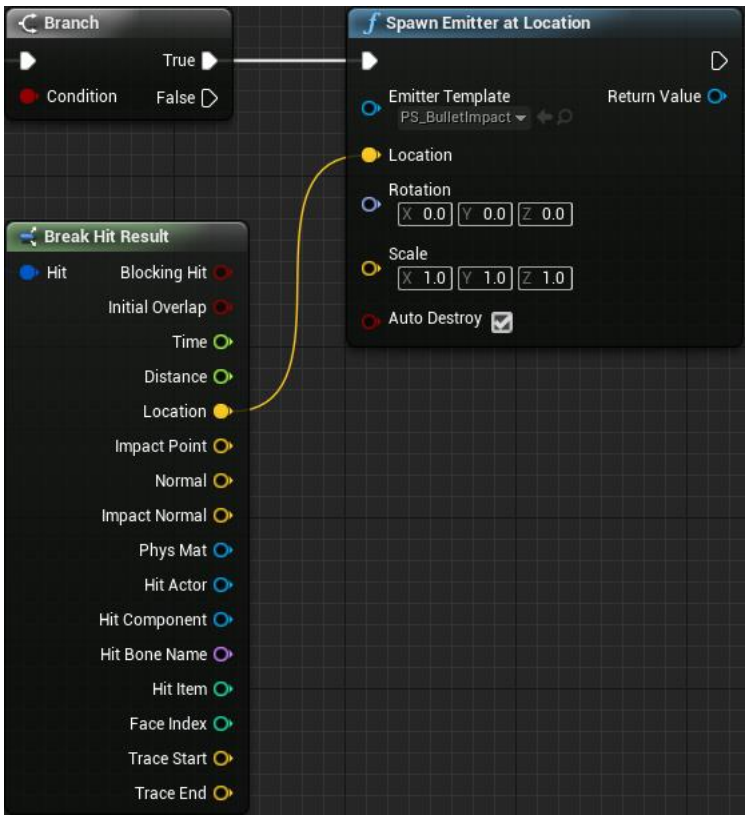
Создание частиц попадания пули

Сначала нам нужно получить местоположение попадания трассировки. *Перетащите левой клавишей мыши Out Hit* на граф. В меню выберите *Break Hit Result*.

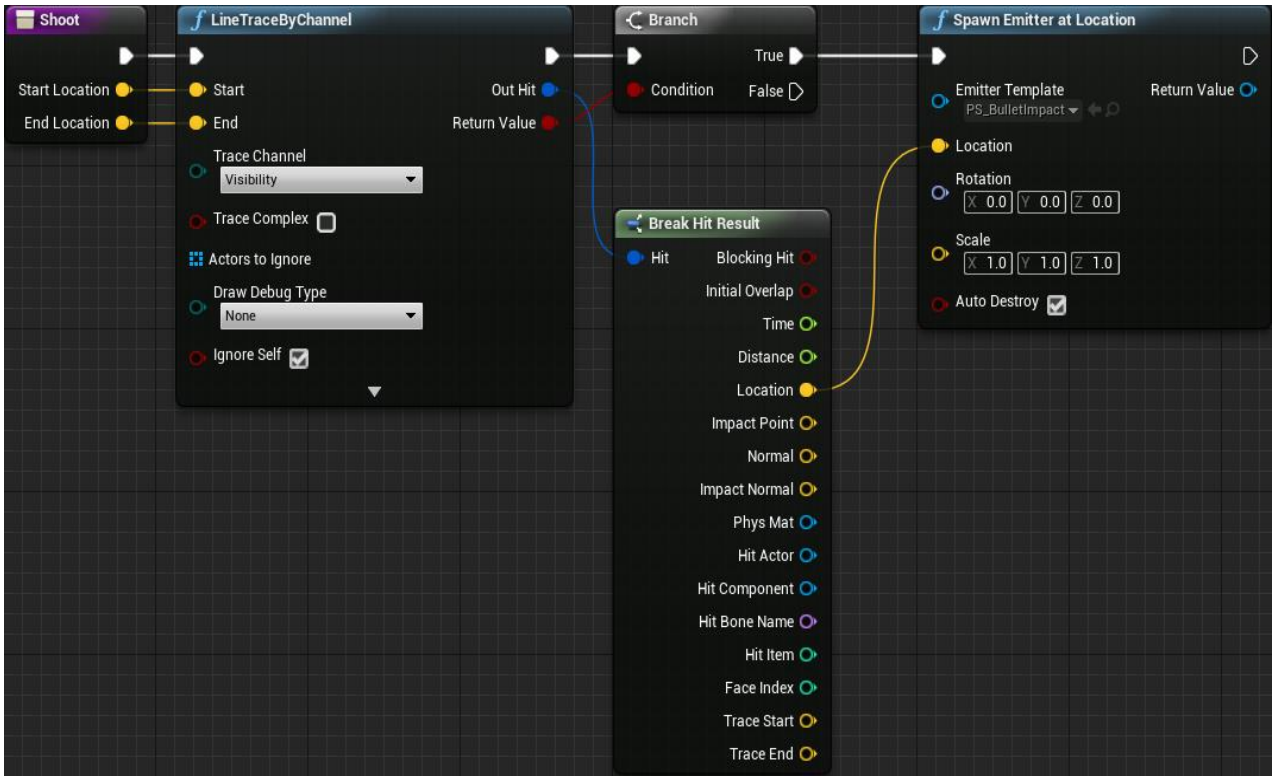


Так мы создадим нод с разными контактами, относящимися к результатам трассировки прямой.

Создайте *Spawn Emitter at Location* и задайте для *Emitter Template* значение *PS_BulletImpact*. Затем соедините его *Location* с *Location* нода *Break Hit Result*.



Вот как это будет выглядеть:



Подведём итог:

1. При выполнении *Shoot* она выполняет трассировку прямой с переданными начальной и конечной точкой

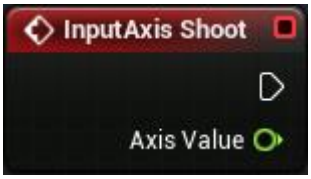
2. Если попадание зафиксировано, то *Spawn Emitter at Location* создаст *PS_BulletImpact* в точке попадания

Теперь, когда логика стрельбы завершена, нам нужно воспользоваться ею.

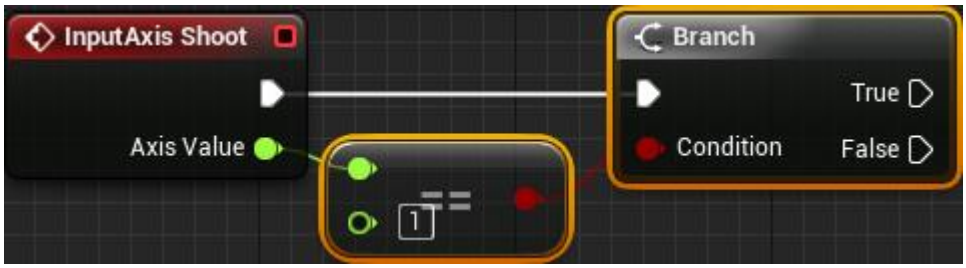
Вызов функции Shoot

Для начала нам нужно создать привязку клавиш для стрельбы. Нажмите на *Compile* и откройте *Project Settings*. Создайте новую *Axis Mapping* под названием *Shoot*. Выберите для неё клавишу *Left Mouse Button* и закройте *Project Settings*.

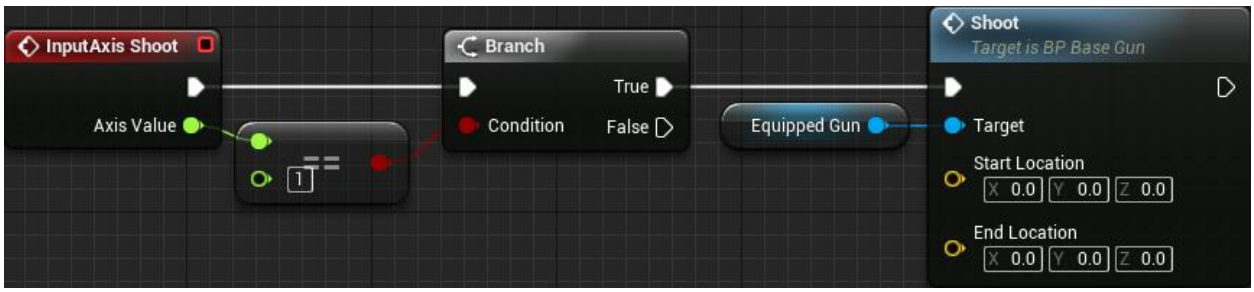
Затем откройте *BP_Player* и создайте событие *Shoot*.



Чтобы проверить, нажимает ли игрок клавишу *Shoot*, нам достаточно проверить, равно ли значение *Axis Value* единице (*1*). Создайте выделенные ноды:



Затем создайте ссылку на *EquippedGun* и вызовите его функцию *Shoot*.



Теперь нам нужно вычислить начальную и конечную точки для трассировки прямой.

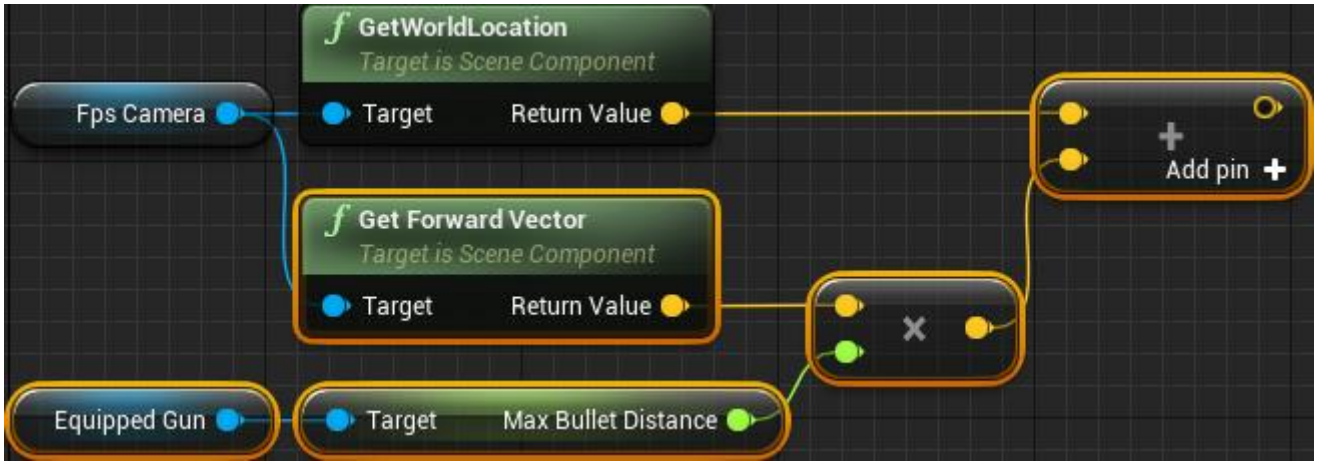
Вычисление точек трассировки прямой

Во многих FPS пуля летит из камеры, а не из оружия. Так делают потому, что камеру уже и так идеально выровнена с прицелом. Поэтому если мы будем стрелять из камеры, то пуля гарантированно полетит туда, где находится курсор.

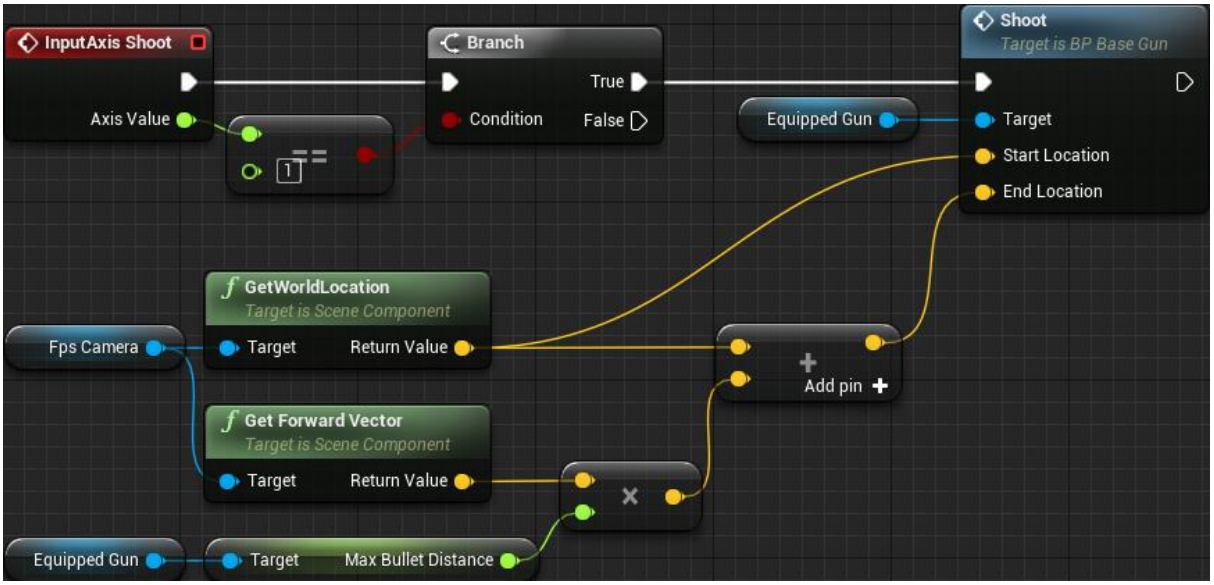
Примечание: некоторые игры всё-таки реализуют стрельбу из оружия. Однако для стрельбы ровно в прицел требуются дополнительные вычисления.

Создайте ссылку на *FpsCamera* и соедините её с *GetWorldLocation*.

Теперь нам нужна конечная точка. Не забывайте, что у оружия есть переменная *MaxBulletDistance*. Это значит, что конечная точка должна находится на расстоянии *MaxBulletDistance* единиц от камеры. Чтобы реализовать это, создайте выделенные ноды:



Затем соедините всё следующим образом:



Подведём итог:

1. Когда игрок нажимает или удерживает *левую клавишу мыши*, оружие будет выпускать пулю с начальной точкой в *камере*
2. Пуля пролетит расстояние, указанное в *MaxBulletDistance*

Нажмите на *Compile*, а затем на *Play*. Удерживайте *левую клавишу мыши*, чтобы начать стрельбу.

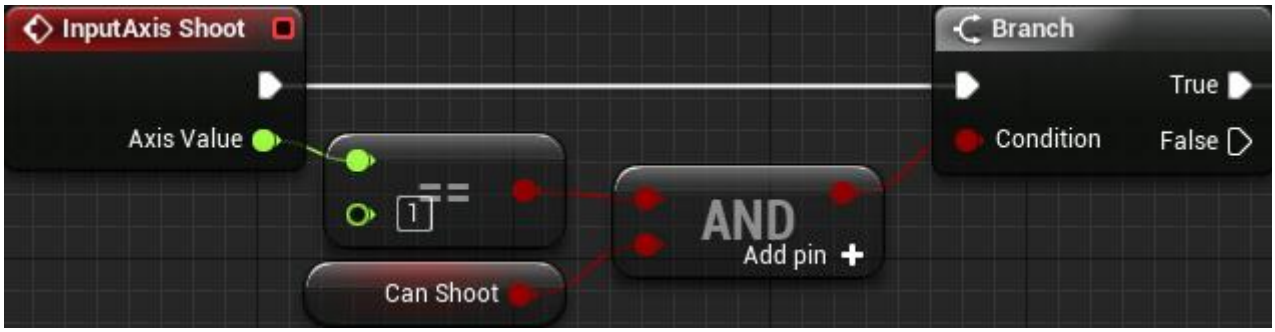
GIF

Пока оружие стреляет в каждом кадре. Это слишком быстро, поэтому на следующем этапе мы уменьшим частоту стрельбы оружия.

Уменьшение частоты стрельбы

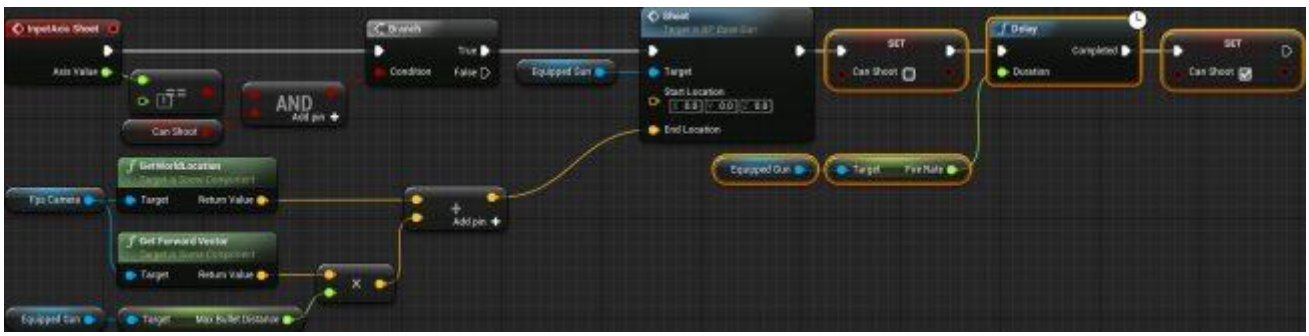
Во-первых, нам нужна переменная, чтобы определить, может ли игрок стрелять. Откройте *BP_Player* и создайте переменную типа *boolean* с названием *CanShoot*. Задайте ей значение *true* по умолчанию. Если *CanShoot* равна *true*, то игрок может стрелять, и наоборот.

Замените раздел *Branch* на следующее:



Теперь игрок может стрелять только если нажата клавиша *Shoot* и переменная *CanShoot* равна *true*.

Теперь добавим выделенные ноды:



Изменения:

1. Игрок может стрелять, только когда удерживает *левую клавишу мыши*, и когда *CanShoot* равна *true*
2. Когда игрок стреляет пулей, переменной *CanShoot* присваивается значение *false*. Это не позволит игроку выстрелить снова.
3. *CanShoot* будет снова присвоено значение *true* через промежуток времени, указанный в *FireRate*

Нажмите на *Compile* и закройте *BP_Player*. Нажмите *Play* и проверьте новую частоты стрельбы.

Теперь мы научим цели и кнопку реагировать на пули. Это можно сделать, добавив им *урон*.

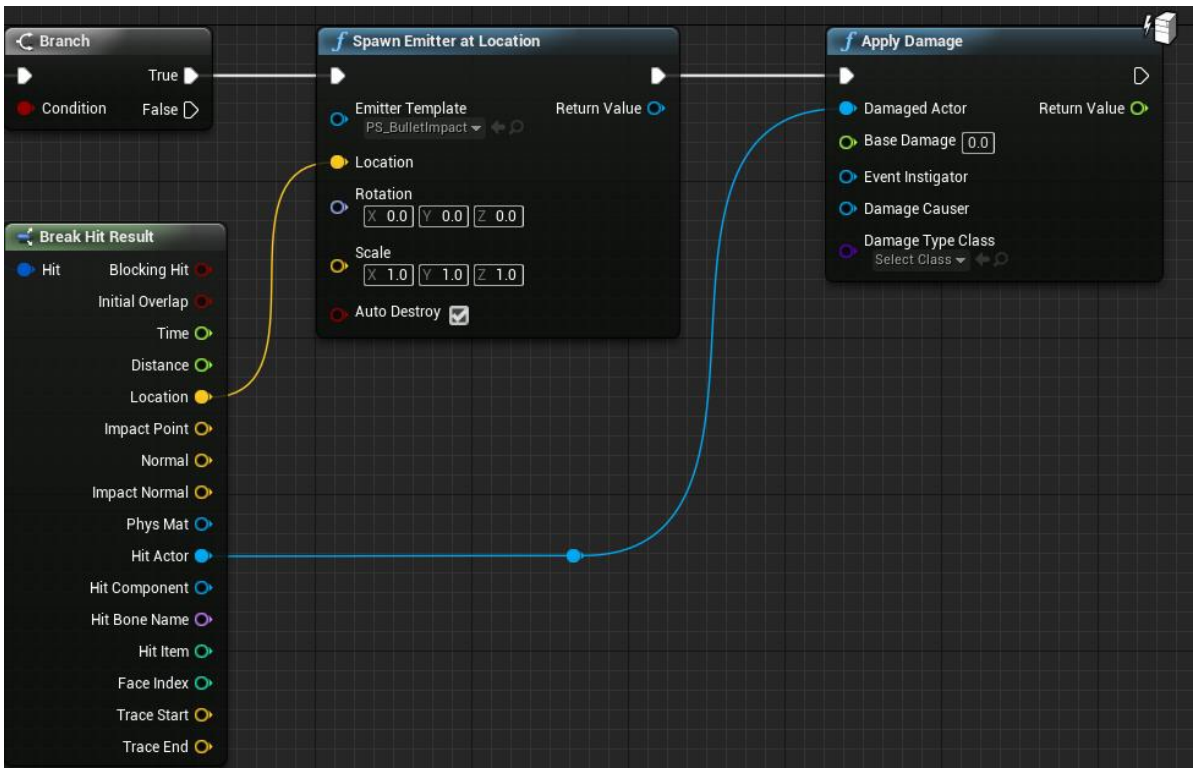
Применение урона

Каждый актор в Unreal имеет возможность получения урона. Однако вы можете выбирать сами, *как* актор будет на него реагировать.

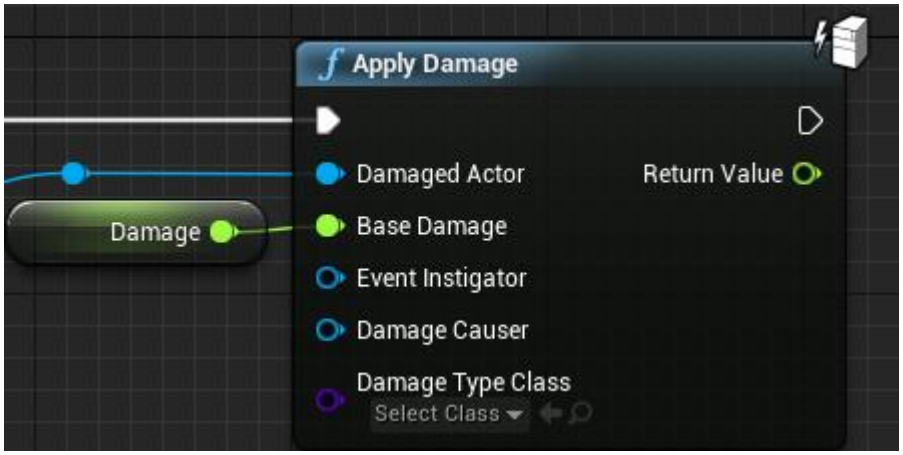
Например, персонаж файтинга при получении урона будет терять здоровье. Однако некоторые объекты, например, воздушный шарик, могут не иметь здоровья. Тогда можно запрограммировать шарик, чтобы он взрывался при получении урона.

Для того, чтобы можно было управлять получением актором урона, нам сначала нужно применить урон. Откройте *BP_BaseGun* и добавьте *Apply Damage* в конце функции *Shoot*.

Теперь нам нужно указать, какому актору мы хотим наносить урон. В нашем случае это актор, в которого попадает трассировка прямой. Соедините *Damaged Actor* с *Hit Actor* нода *Break Hit Result*.



Наконец, нам нужно указать величину нанесённого урона. Получите ссылку на *Damage* и соедините её с *Base Damage*.



Теперь при вызове *Shoot* она будет наносить урон акторам, попавшим в трассировку прямой. Нажмите на *Compile* и закройте *BP_BaseGun*.

Теперь нам нужно обработать то, как актор получает урон.

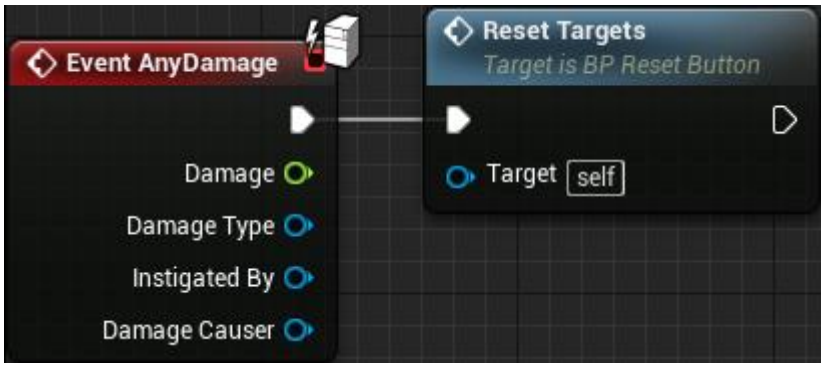
Обработка урона

Сначала мы обработаем то, как урон получают цели. Откройте *BP_Target* и создайте *Event AnyDamage*. Это событие выполняется, когда актор получает урон, *не равный нулю*.

Теперь вызовите функцию *TakeDamage* и соедините контакты *Damage*. Это вычтет здоровье из переменной *Health* цели и обновит цвет цели.

Теперь, когда цель получает урон, она теряет здоровье. Нажмите на *Compile* и закройте *BP_Target*.

Теперь нам нужно обработать то, как получает урон кнопка. Откройте *BP_ResetButton* и создайте *Event AnyDamage*. Затем вызовите функцию *ResetTargets*.



Это будет восстанавливать все цели при получении урона кнопкой. Нажмите на *Compile* и закройте *BP_ResetButton*.

Нажмите на *Play* и начните стрелять по целям. Если вы хотите восстановить цели, то выстрелите по кнопке.