**6. Line Trace**

1. Научим наше оружие стрелять. Нужно получить координату дула – что нам в этом поможет, как создали? Где расположили?

2. Нарисуем линию, вдоль которой будет стрелять оружие – добавляем функцию, где будем ее вызывать? Почему выделили в отдельную функцию? Подключаем два ЗФ для отрисовки линии

3. Добавляем проверочку нашего оружия

**Остановись тут и попробуй вспомнить весь алгоритм того, что нам надо сделать, потому что дальше сам ход мысли следует из вопросов, а тут важно еще понять алгоритм**

4. Первым делом нам нужно получить трансформацию сокета – как сделали это? Что принимает функция?

5. Вводим переменные для начальной точки, направления (вот тут поподробнее – что возвращает функция и что мы вызываем дальше и что возвращает та, которую мы вызываем последней), конечной, максимальной дистанции – как получаем сюда значения? С помощью какой функции рисуем линию? Два новых параметра в этой функции

6. Опять мой баг с сокетами, запомнить его. Какая проблема наблюдается сейчас при стрельбе, как пофиксили?

7. Как получить информацию о пересечении данного луча каких-либо объектов в нашем игровом мире? Где находится большинство функций для этой логики? Какой мы воспользуемся? Что принимает (подробно о первом и четвертом параметре)? Что функция возвращает? Как будем проверять попадание (два способа)? Рисуем сферу в конце попадания, как, что передаем?

8. Что надо исправить у нашего BP-класса персонажа, чтобы отлавливать трейсы?

9. Какая сейчас проблема осталась у нас? Откуда нам надо строить трейс? Подключаем два ЗФ, как получить нашу камеру (исправления в cpp файле компонента оружия)?

10. Как теперь получить контроллера и от него положение и вращение камеры?

11. Как теперь строим наш трейс?

12. Скорректируем отрисовку линии выстрела

13. Однако сейчас у нас попадание идет по \_ персонажа, а не по \_, как это исправили?

14. Осталось еще два бага, как исправили один из них? Как можно исправить второй?

15. Последнее – выведем кость скелета, в которую попали, как?

1. В данном уроке научимся стрелять. Для начала надо открыть Mesh винтовки и разберемся, как получить координату дула. В этом нам поможет Socket, который ранее мы использовали для присоединения оружия к руке персонажа.

Добавив Socket, он размесится в нуле локальной системы координат. Назовем его MuzzleSocket (muzzle – дуло). Путем перемещений через Gizmo помещает сокет к месту, откуда хотим делать выстрел. Мы создали сокет, к которому сможем получить доступ из кода, и сделать выстрел из данной точки.

2. Теперь нарисуем линию, вдоль которой будет стрелять наше оружие.

Для этого создаем у **оружия** функцию MakeShot() – в ней будет вся логика выстрела. Мы можем использовать и функцию Fire, но в дальнейшем у нас будет стрельба по таймеру, поэтому лучше данную логику выделить в отдельную функцию. Плюс, если у нас будет тип оружия винчестер, то надо будет делать несколько выстрелов одновременно.

Добавляем вызов функции в Fire():

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

А также подключим ЗФ для мира (Engine/World.h) и отрисовки различных отладочных вещей – DrawDebugHelpers.h.

3. Добавим проверку на Mesh в BeginPlay(), чтобы не делать эти проверки везде.

Изображение выглядит как текст

Автоматически созданное описание

4. Первым делом в функци MakeShot() нам нужно получить трансформацию сокета. Созданный нами сокет находится в WeaponMesh:

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Первый параметр – имя сокета, второй – система координат, в которой мы хотим получить сокет (оставляем значение по умолчанию – мировая система координат).

5. Так же нам понадобятся несколько переменных.

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Первая – точка начала выстрела TraceStart.

Вторая – единичный вектор направления стрельбы ShootDirection. В данном случае функции GetRotation возвращает кватернион (живи с этим) FQuat (помимо FRotator еще используется и такое). У него четыре компоненты – X, Y, Z и W.

Данный тип нам удобнее использовать, потому что у кватерниона есть функция GetForwardVector(). Если посмотреть ее код, то мы увидим, что там идет возвращение результата вызова функции GetAxisX() – то есть возвращается **вектор поворота** по оси X.

Изображение выглядит как текст

Автоматически созданное описание

Теперь мы знаем начальную точку и направление выстрела.

Третья переменная – TraceEnd – конечная точка выстрела. Чтобы ее получить, нам нужно отложить вектор направления от начальной точки – то есть к переменной TraceStart прибавить ShootDirection, но данный вектор единичный, поэтому домножили его на большое число.

Саму линию рисуем с помощью функции DrawDebugLine.

Пятый параметр отвечает за то, хотим ли мы, чтобы линия отрисовывалась постоянно. Если нет, то следующим указываем длительность рисования.

Последний параметр – толщина.

6. Опять обнаруживаю баг того, что сокет куда-то сбрасывается. Это исправляется тем, что на самом деле у нас сокет меняется на какую-то хуйню. В первый раз это было для самого оружия, теперь для выстрела. Фиксится это либо изменением названия этого сокета-замены, либо в коде изменяем название сокета.

Если мы сейчас запусти игру, то стрелять мы будем перпендикулярно вправо. Туда и направлена красненькая линия в Gizmo, которая соответствует X-у. Надо изменить Rotation у сокета по Y на +90, а по Z на -90 (второе просто чтобы Z смотрело вверх).

7. Теперь посмотрим, как получить информацию о пересечении данного луча каких-либо объектов в нашем игровом мире.

С математической точки зрения нам надо найти пересечение нашего отрезка с объектами на сцене. В нашем случае достаточно найти объект, который луч пересечет **первым**.

В движке «из коробки» есть множество функций, которые позволяют находить такие геометрические пересечения. Мы воспользуемся одной из них.

Большинство из этих функций находятся в объекте UWorld().

Нужная нам функция – LineTraceMultiByChannel. Она возвращает информацию о самом первым объектом, который встретится на пути нашего отрезка (если такой имеется, разумеется).



Первый параметр – структура FHitResult, которая возвращается по ссылке (то есть в нее записываются данные – ее надо заранее объявить). Она содержит всю информацию о пересечении (время, точка пересечения, нормаль, указатель на актора, с которым произошло пересечение и т.д.).

Четвертый параметр – ECollisionChannel – с помощью него наш Trace будет отфильтровывать все объекты на сцене. И наш Trace будет работать только с теми объектами, которые блокируют данный канал. Все остальные будут игнорироваться.

Остальные – по умолчанию.

Также функция возвращает Boolean того, было пересечение или нет. Поэтому мы можем узнать через возвращаемое значение функции, либо через поле bBlockingHit HitResult.



Нарисовали сферу в точке пересечения в конце функции. Вторым параметром у этой функции передается центр сферы, в HitResult этим полем является ImpactPoint. Шестым параметром является какая-то чебушня, а последний отвечает за длительность отрисовки.

8. Надо проверить CollisionChannel нашего персонажа, открыв CapsuleComponent и выбрав Custom Collision Preset и выбрав для Trace Response Visibility значение Block. Если он будет Overlap или Ignore, то наш Trace просто не сработает.

9. Однако сейчас у нас идет выстрел по прямой из нашего оружия, не важно куда при этом смотрит наш прицел. **Для этого нам нужно делать Trace непосредственно из камеры игрока**.

Получить доступ к камере в классе нашего оружия можно следующим образом. Для начала подключим два ЗФ для персонажа и контроллера (в классе нашего оружия):

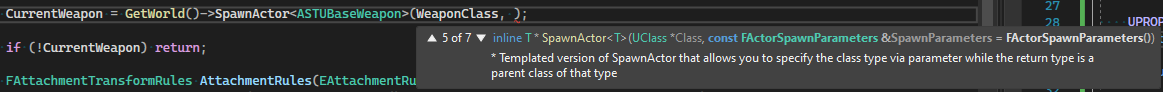
#include "GameFramework/Character.h"

#include "GameFramework/Controller.h"

У акторов, так же как и у компонентов, имеется концепт Owner’а, то есть владельца данного актора. Поэтому мы можем проделать аналогичные действия тем, которые мы делаем для компонентов.

Так как компоненты мы создаем у акторов, то Owner у них определяется автоматически. Для акторов надо указывать явно, кто их владелец. Как это сделать?

Перейдем в cpp файл WeaponComponent и в момент создания оружия мы можем в SpawnActor указать владельца нашего оружия. Мы можем указать его через FActorSpawnParametres:



Мы же сделаем это с помощью вызова функции SetOwner() у самого оружия (можно передать напрямую Character, можно вызвать функцию GetOwner):

Изображение выглядит как текст

Автоматически созданное описание

Поэтому теперь данный код работает правильно:

Изображение выглядит как текст

Автоматически созданное описание

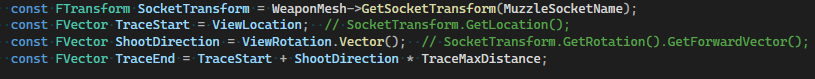
10. В классе Character имеется указатель на Controller (наличие его понятно, обычно персонажи кем-то управляются). Получаем мы его через функцию GetController.

Функция, которая возвращает позицию и ориентацию камеры в пространстве называется GetPlayerViewPoint – данная функция имеет доступ к классу PlayerCameraManager, который является глобальным классом для управления камерами, и уже из него получает указатель на нашу камеру персонажа. У функции два выходных параметра **по ссылке** – положение и вращение камеры.

Изображение выглядит как текст

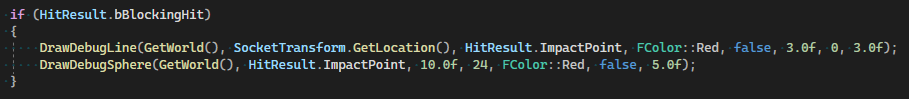
Автоматически созданное описание

11. Теперь мы обладаем информацией о нашей камере и можем переписать наш Trace. Изменяем значения переменных на новые.



Теперь начальная точка трейса – это положение камеры, а направление мы получили из ротатора с помощью вспомогательной функции Vector, которая работает аналогично GetForwardVector кватерниона.

12. Скорректируем отрисовку линии выстрела (так как рисовать мы должны именно из дула рисовать, а конец – точка пересечения):



Если же мы никуда не попали, то:

Изображение выглядит как текст, внутренний, снимок экрана, монитор

Автоматически созданное описание

Теперь Trace идет из камеры и попадает туда, куда указывает наш прицел.

Единственное – угол стрельбы выглядит странно, так как персонаж не наклоняется/поднимается.

13. Обратим внимание вот на что – увеличим размер капсулы у нашего персонажа (через BP). При попадании мы увидим, что оно фиксируется именно для капсулы, а не для самой модельки персонажа (так как она просто больше нее).

Изображение выглядит как внешний, люди, автомат

Автоматически созданное описание

Мы же хотим попадать в Mesh персонажа. Отменим изменения в CapsuleComponent для пресета коллизии, чтобы Visibility стояло в Ignore.

А в компоненте SkeletalMesh’а поменяем пресет на Custom и поставим для Visibility – Block.

Теперь трейс для капсулы игнорируется, а для меша – нет.

14. У нас остались еще две проблемы. Одна из них – мы можем попасть в самих себя, если удачно выставим камеру (так как трейс идет из камеры).

Изображение выглядит как небо, внешний, человек, холм

Автоматически созданное описание

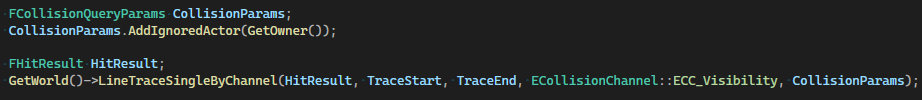
Вторая проблема – мы можем попасть в персонажа, даже находясь ЗА ним (по аналогичной выше причине):

Изображение выглядит как езда на мотоциклах, автомат

Автоматически созданное описание

Второе мы фиксить не будем – один из вариантов – проверять угол между трейсом и направлением из дула острый.

Добавим фикс, чтобы мы не попадали в самих себя – это можно сделать через дополнительный параметр функции трейса с типом FCollisionQueryParams – это структура, и нас интересует ее метод AddIgnoredActor(), которая принимает указатель на Actor и добавляет его в массив акторов, которых мы хотим игнорировать при трейсе.



15. Последнее – выведем кость скелета, в которую попали. Информация о кости скелета находится в поле HitResult с идентификатором BoneName:

Изображение выглядит как текст

Автоматически созданное описание

Обладая данной информацией можно сделать урон для различных частей тела.

Если стреляем по другим объектам (кубу), то информацию о костях у нас нет.