15. Modules, Targets, UnrealBuildTool

1. Первый файл, который мы рассмотрим, называется GeometrySandbox.Build.cs.

Данный билд файл написан на C# - в нем задаются правила, по которым будет собран исходный код нашего проекта - GeometrySandbox.

Когда мы запускаем сборку, нажимая, например, на кнопку Play Local Windows Debugger, то запускается специальная утилита, которая называется Unreal Build Tool.

Это специальная подпрограмма, которая поставляется вместе с Unreal Engine. По сути visual studio запускает данную программу с различными commad line аргументами. В целом мы можем создать какой-нибудь bat файл для сборки нашей игры, который будет вызывать данную программу. И в соответствии с настройками нашего build файла происходит компиляция исходников.

2. Идем далее. Весь код движка состоит из модулей.

Модуль – это кирпичик, в котором инкапсулирована какая-то функциональность. Это может быть искусственный интеллект, спецэффекты, пользовательский интерфейс и так далее. Более того наш проект с игрой также является модулем. В каждом билд файле класс модуля наследуется от специального класса, который называется ModuleRules. По умолчанию в нашем классе имеется несколько настроек. Сейчас нас будет интересовать список, который называется PublicDependencyModuleNames В данном списке указываются модули, от которых будет зависеть наш проект. По умолчанию добавлены 4 основных модуля движка. В solution explorer мы можем открыть проект UE4. Перейти в папку source, в папку runtime В данной папке находятся исходники движка, и каждая папка является модулем. То есть мы можем найти: Core, CoreUObject и если мы откроем модуль, то увидим, что в корне находится тот же самый файл build.cs Для примера можем открыть build файл InputCore и посмотреть каким образом происходит сборка модуля. Можно обратить внимание, что компиляция данного модуля будет зависеть от целевой платформы. Также в билд файле мы можем добавлять зависимости от различных сторонних библиотек. Например, если мы захотим подключить какую-то библиотеку машинного обучения, либо библиотеку компьютерного зрения, все это можно настроить в build файле. Итак, все состоит из модулей, наша игра также является модулем.

3. Также взглянем ещё на два сгенерированных файла, которые находятся в корне проекта: GeometrySandbox.h и .cpp В данном файле с помощью специального макроса указывается основной игровой модуль, то есть, грубо говоря, сама игра может состоять также из нескольких модулей. У нас с вами один модуль, поэтому в файле, который генерируется по умолчанию указано его имя. Данный файл мы с вами трогать не будем вообще.

4. Если открыть главный файл нашего проекта, с расширением .uproject, то можно видеть, что данный файл на самом деле представляет собой разметку JSON. В поле modules находится массив объектов модулей игры, также имеются дополнительные параметры. У нас с вами один модуль - GeometrySandbox. Также указана версия движка, различные описания, которые у нас пустые. С помощью .uproject. Unreal Engine умеет генерировать необходимые файлы для проекта и также запускать в соответствии с настройками Unreal Editor.

5. Двигаемся дальше. Давайте разберемся со следующей вещью. В нашем проекте имеется небольшая каша из заголовочных файлов и forward declaration. На самом деле хорошо, что так получилось, поскольку мы разберем данный вопрос.

Открываем заголовочный файл SandboxPawn и мы видим, что у нас имеется forward declaration для камеры и при этом у нас отсутствует forward declaration для StaticMesh компонента, и также никаких ошибок при сборке проекта у нас нет. Что же у нас происходит? Почему для компонента камеры нам необходимо делать forward declaration, а для static mesh компонента нет? На самом деле мы можем отключить большинство заголовочных файлов, которые у нас имеются в .cpp файлах, например, отключаем static mesh, отключаем input component, переходим в заголовочный файл BaseGeometryActor, убираем include статик меш компонента. В .cpp файле давайте уберем MaterialInstance, Engine и Timer. В .cpp файле класса GeometryHubActor отключаем заголовочный файл мира, в контроллере убираем input component и сбилдим наш проект, не запуская едитора.

Для этого кликаем на проекте правой кнопкой мыши и выбираем build. У нас все успешно скомпилировалось и без наших заголовочных файлов. Это произошло в результате срабатывания механизма, который называется precompiled headers. Это механизм, который позволяет существенно сократить объем работы, который должен проделать препроцессор при компиляции проекта. Идея в том, чтобы заранее препроцессировать группу файлов и затем просто подставлять готовый фрагмент текста.

Откроем build файл модуля Engine. Перед этим давайте посмотрим на наш модуль. Настройка, которая отвечает за precompiled headers называется PCHUsage. То есть precompiled headers usage. И по умолчанию стоит значение, при котором явно будут подключаться shared pch файлы. Открываем билд файл модуля Engine и находим настройку, которая называется SharedPCHHeaderFile. Данный файл и будет добавлен при компиляции нашего модуля.

Давайте найдем его в модуле Engine, в директории public. Открываем. В данном файле находятся заголовки, которые будут препроцессированны, то есть это код, который меняется очень редко. Давайте выполним поиск по файлу. Видим, что здесь находится StaticMeshComponent, также можем найти MaterialInstanceDynamic, TimerManager. InputComponent также находится здесь и так далее. И теперь становится ясно почему у нас все скомпилировалось корректно, то есть данный заголовочный файл был неявно добавлен при сборке нашего проекта.

А теперь давайте немного поменяем настройку и поставим NoPCHs. Если вы используете исходники unreal engine, то вы можете выполнить поиск по проекту и найти данную настройку. То есть теперь мы декларируем, что мы не хотим использовать precompiled headers при сборке нашего проекта. Выполняем build. Проект не скомпилировался, потому что в нашем коде не подключены соответствующие заголовочные файлы. Давайте возвращать их назад. Переходим в SandboxPawn. Делаем forward declaration в заголовочном файле static mesh компонента. В .cpp файле возвращаем заголовочный файл InputComponent и также нам необходимо подключить заголовочный файл статик меша. Переходим в BaseGeometryActor. Также добавляем forward declaration для статик меш компонента. Давайте скопируем его из SandboxPawn. Переходим в .cpp файл. Ctrl+Z возвращаем все заголовочные файлы. Перемещаемся в GeometryHubActor. Возвращаем заголовочный файл мира. И в контроллере возвращаем заголовочный файл InputComponent. Компилируем проект еще раз. Что-то все-таки еще забыли. В .cpp файле BaseGeometryActor также необходимо подключить static mesh компонент. И в SandboxPlayerController мы используем указатель на мир, поэтому подключаем заголовочный файл Engine/World.h Так я описался World. Собираем проект еще раз. Все. Build successs. Проект скомпилировался успешно. Мы корректно подключили все заголовочные файлы, которые нужны в нашем проекте. Единственно мы не подключали заголовочный файл SceneComponent, потому что его include находится в классе актора, который соответственно находится в классе pawnа. Это также можно сделать, хуже от этого не будет, но для некоторых супер базовых типов этого можно не делать, поскольку мы уверены, что они подключены либо через PCH, либо через CoreMinimal.h, либо их include находится где-то в родительском классе. Вернем настройку precompiled headers в дефолтное состояние. Подход, который мы сейчас использовали, называется include what you see. Я также оставлю ссылку на данную страницу документации Unreal, можно почитать подробнее, но в целом мы его сейчас рассмотрели. Давайте также посмотрим как быстро найти заголовочный файл нужного нам класса. Просто вводим в google класс. Открывается страница документации Unreal. Вверху страницы имеется табличка, в поле include содержится интересующий нас заголовочный файл. Либо, если у вас уже открыта страница документации, то в поле поиска можно также ввести интересующий класс и в отфильтрованной информации выбрать страницу с классом.

6. Переходим к двум оставшимся файлам в директории нашего проекта: один из них называется GeometrySandbox.Target.cs а второй GeometrySandbox.EditorTarget.cs

B данных C# файлах указываются дополнительные настройки для сборки, в зависимости от того КАК мы хотим запускать наш проект. Если мы хотим запустить игру под едитором, то используется EditorTarget, если хотим собрать standalone игру то используется обычный таргет, который соответствует TargetType.Game Имеются также дополнительные типы таргетов, например, если мы захотим собрать нашу игру как выделенный сервер, то нам необходимо будет вручную добавить в проект GeometrySandbox.ServerTarget.cs и в переменную Type записать TargetType.Server, то есть в данных файлах мы указываем специфические настройки сборки игры под определенные таргеты. К лекции я также приложу ссылку на документацию по таргетам. В нашем курсе данные файлы мы настраивать никак не будем, будем использовать то, что сгенерировал unreal для нас по умолчанию, но важно помнить, что когда мы запускаем игру под едитором используется EditorTarget а когда мы будем создавать standalone build, упаковывать игру в exe файл, то будет использоваться другой таргет, который относится к типу game.

7. Если открыть директорию проекта папку Binaries/Win64, то можно видеть, что при запуске для едитора наш код собирается в специальный .dll файл. Также можно зайти в директорию binaries движка, открыть поддиректорию win64 и можно увидеть, что для каждого модуля создан специальный .dll файл с TargetEditor, то есть фактически, когда мы скачиваем unreal engine из-под epic launcher к нам на диск копируются заранее скомпилированные модули в виде бинарных .dll файлов, а когда мы хотим скомпилировать unreal из исходников, то компиляция модулей и создание .dll файлов будет происходить на нашей локальной машине.

8. Теперь давайте поговорим про макросы unreal и кодогенерацию. В urneal реализована собственная reflection систем или система рефлексии - это способность программы отслеживать и модифицировать собственную структуру и поведение во время ее выполнения. С помощью данной системы мы можем получать информацию о типах, вызывать функцию по имени, получать имена элементов enum мы можем создать iterator, который будет обходить все поля структуры и получать какую-то информацию. Можем передать указатель на класс, объект которого мы хотим создать. Данный трюк мы с вами уже проделывали, в GameMode, когда указывали для пауна и контроллера классы, на основе которых мы хотим создать объекты. Данная система значительно ускоряет процесс разработки и области ее применения достаточно разные. Самое очевидное - это использование едитора: вывод в интерфейс всех property, коммуникация C++ и blueprint кода, сериализация объектов, например, если мы на сцене создаем несколько акторов, изменяем им какие-то свойства, потом закрываем едитор, переоткрываем его - все значения property на месте, то есть все property сохранились на диск и потом считались с диска.

9. Также на основе данной системы работает сборщик мусора - garbage collector. У unreal собственная система памяти, о ней мы поговорим в следующем уроке, и также в область применения данной системы можно добавить работу с сетью. В чистом C++ всех этих преимуществ естественно нет, в этом и заключается понятие игрового движка.

10. Также важно отметить, что unreal не запрещает писать на чистом C++, то есть вы можете создать обычный C++ класс, сделать в нем обработку неких вычислений и, например, передать результат вычислений в какой-либо actor.

11. Давайте посмотрим, что у нас происходит в самом начале, когда мы создаем класс из-под едитора. Откроем директорию, где находится движок, переходим в папку Engine/Content/Editor/Templates.

В данной директории находятся шаблоны всех основных классов движка, если мы откроем шаблон класса actor, то увидим, что это некий мета файл с кучей переменных, в которые подставляются нужные имена классов, нужные копирайты и так далее и в результате подстановки всех необходимых переменных создается итоговый файл класса. Давайте запустим build нашего проекта. У нас все up to date, давайте добавим пустую строчку и нажмем на build еще раз. Что происходит в самом начале? Мы видим, что в консоль записалась строчка parsing headers GeometrySandboxEditor и далее запустилась специальная утилита, которая называется Unreal Header Tool - это специальный парсер-генератор, который обрабатывает все макросы, которые находятся в нашем исходном коде. Первое, что делает данный генератор - он просматривает все заголовочные файлы и ищет в них заголовочные файлы с суффиксом .generated.h - данный заголовочный файл является маркером для Unreal Header Tool, благодаря ему он понимает, что данный заголовочный файл нужно обработать, то есть в нем присутствуют специфические unreal типы, если данного заголовочного файла он не находит в файле, то он его просто проигнорирует. После этого генератор проходит по всем UENUM, USTRUCT, UCLASS, UPROPERTY, UFUNCTION и генерирует дополнительный код, который необходим для reflection system Unreal Engine. Часть кода попадает в заголовочный файл .generated.h, если мы нажмем f12, то можем увидеть кучу кода генерации. Она не предназначена для чтения человеком, но можно обратить внимание, что создались некие обертки вокруг наших property, а если мы в данном файле попробуем поискать член класса, который не является property, нажимаем Ctrl+F, то мы ничего не найдем. Никакой дополнительный код для данного члена-класса не был сгенерирован. Все файлы генерируются в директории intermediate нашего проекта. Кликнем правой кнопкой мыши на заголовке файла, выбираем Open Contain Folder, открывается поддиректория в папке intermediate, где находятся все сгенерированные файлы. Помимо заголовочных файлов также генерируются .cpp файлы для каждого класса. В данных файлах находится еще больше кода-генерации. Имеются совершенно монструозные типы. И опять же данные файлы также не предназначены для чтения человеком, что происходит внутри во время обычной разработки знать совершенно необязательно. И еще один момент. В классе имеется дополнительный макрос, который называется имя проекта нижнее подчёркивание API. Нажимаем f12 на нем. макрос находится в другом сгенерированном файле, который расположен также в директории intermediate. Можно в этом убедиться. Ему в соответствие ставится другой макрос DLLEXPORT, который имеет разные значения под разные платформы. Мы с вами компилируем под windows, для windows это \_\_declspec(dllexport) что по сути означает, что наш класс может быть экспортирован из .dll файла. Также мы можем спокойно удалить все сгенерированные файлы и если мы сбилдим проект еще раз, то увидим, что Unreal Header Tool их перегенерирует. Также, например, в папке intermediate находятся все файлы проекта visual studio в директории project files, более того, вся папка intermediate содержит временные файлы. Закроем solution. Давайте удалим директорию intermediate. В следующем разделе мы подробнее поговорим про директории, которые находятся в корне проекта, а сейчас давайте многие из них удалим. И ставим только папку config, content и source. Также удалим файл solution. Когда вы в первый раз клонируйте репозиторий, вы будете видеть именно такую структуру нашего проекта. Если кликнуть правой кнопкой мыши на файл .uproject, то можно, например, поменять версию движка, либо можно перегенерировать все файлы visual studio. Если нажать на кнопку show log, то можно увидеть, что запускается Unreal Build Tool, он так же умеет генерировать файлы проекта visual studio. Создалась директория intermediate и также создались все файлы проекта.