Краткое введение в ECS

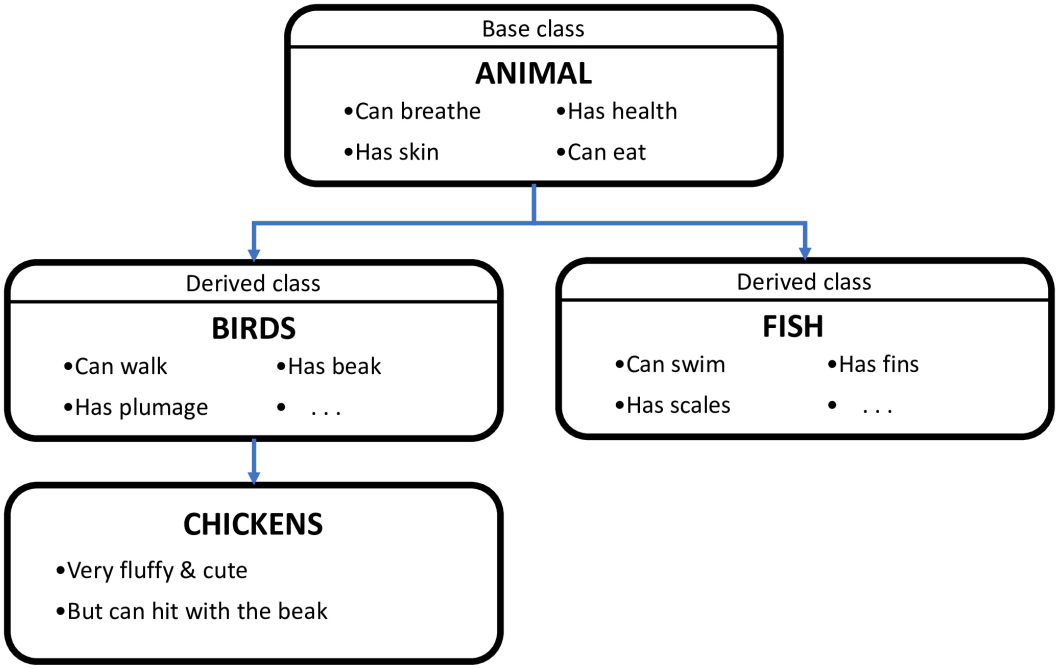
Причуды и ограничения ООП

Прежде чем переходить к обсуждению ECS, стоит обратиться к Объектно-ориентированному программированию (ООП) и его проблемам, так как пользовательский API Unreal Engine'а основан именно на парадигмах ООП. Actor-component модель тоже, в некотором смысле, ограничена, и по своей натуре в большей степени является статичной.

Программируя игры (или программное обеспечение вообще) в стиле ООП, мы разбиваем всевозможные сущности и концепции в иерархию их соответствующих типов. В терминах Unreal движка все эти абстракции называются Объектами (или ''UObjects'', если мы говорим про C++ составляющую). Итак, мы создаём иерархию типов (классов), наследовавшись от единого корня - `UObject` предка. Каждый созданный новый блупринт (или C++ класс) является наследником этого базового класса.

Время идёт и мы замечаем, что наши классы начинают всё более и более походить друг на друга, тогда мы проводим так называемый рефакторинг кода (или Blueprint-ассетов), выделяя схожие свойства в отдельный базовый класс. Последний, в свою очередь, становится предком и мы наследуем от него новые типы, которые должны предоставлять определённую функциональность.

Довольно известный пример этом подходу - дерево наследования животных, где классом-корнем выступает абстракция, представляющая собой любое живое существо (аналог `UObject`), а остальные абстракции являются отдельными животными видами:



Потомственные типы наследуют свойства их соответствующих базовых типов, и, таким образом, изменяя свойства базового класса, мы также меняем поведение всех потомков. Мы

можем так же ввести такую технологию, как “перегружаемые методы” ("virtual overridable methods"), чтобы иметь возможность переопределять тела функций в классах-потомках.

Весь этот кипишь выглядит довольно неплохо, разве нет? Да, всем этим можно пользоваться, но пока у вас нету, например, 100 классов. Проблема в том, что игровая логика становится очень размазанной по всем уровням иерархии. Вы всё чаще забываете, что было определено в конкретном классе, какие методы можно перегружать, какие нет. Количество взаимосвязей увеличивает возможность ошибки, а неразбериха только усугубляется. Ваши коллеги, которые писали код некоторых из используемых классов, становятся хранителями каких-то “тайных знаний”. С ростом дерева классов время, требуемое на рефакторинг, увеличивается с экспоненциальной скоростью. Этот “ритуал” становится неотъемлемой частью работы, но к самой разработке утилит для целевого пользователя всё это мало относится. Совершенно неприемлемый подход, учитывая то, как ограничено бывает время и ресурсы.

Спасительный ECS

Решением этих хлопот может быть такой подход, как Entity–component–system (или кратко - ECS).

В ECS нету никаких иерархий, каждый объект игры (т.е. *сущность*) изначально нейтральный. Он состоит только из частей данных, называемых *компонентами* (или *деталями*). Вы можете комбинировать их на конкретной сущности произвольным образом.

Вы хотите дать своему RPG-герою возможность наносить урон ядом? Просто добавьте компонент “отравляющий” и, считай, дело сделано; конечно, если была реализована соответствующая *система*, которая эту логику определяет.

Система - это удалённый агент, оперирующий над деталями. Это также особенность ECS. Системы представляют собой что-то наподобие внешнего кода, который берёт определённые компоненты-детали, как свои входные данные, и, оперируя над ними, возвращает результат. Системы выполняются одна за другой и вместе формируют то, что называется игровым механизмом.

На этом всё. Но как такой простой, удобный и гибкий принцип не был реализован в Unreal Engine до сих пор? Ну, на самом деле был: возьмите Niagara, например. Устройство рендера так же может быть рассмотрено как разработанный софт. Другими словами, какие сорта специальных data-ориентированных подходов в UE не возьми, ни один из них не является настолько выделенным, как ООП-подход. Мы же предоставляем новый принцип, универсальный для всех инструментов разработчика!

Убедитесь в том, что вы читали Словарь Apparatus, чтобы узнать вводимые нами новые термины.

Про ECS+

Apparatus берёт более широкое направление, чем обычный ECS, поэтому мы и называем его ECS+. Вводится бóльшая гибкость, имеется поддержка наследования деталей (компонентов), динамические belt-ы (*chunks* и *archetypes*), subject-booting и др. Само понятие ECS+ должно рассматриваться как синоним слову “Apparatus”, так как плагин, на самом деле, реализует и

полноценно использует именно эту терминологию с самого своего начала.

Вообще говоря, в нашей философии разработки, проектирование никогда не должно быть “простым” или “математически корректным”, но напротив более применимым и удобным в использовании как для конечного пользователя, так и для программиста. Unreal Engine, конечно, является хорошим примером такого подхода со всеми своими концепциями и паттернами. Соединяя в себе оба паттерна ООП и ECS, Apparatus включает всё самое лучшее из 3-х миров, считая мир UE-блупринтов.

Встроенные сущности

На низком уровне, `Subjective` - это интерфейс, и любой класс, который его имплементирует, может быть назван сущностью. Есть несколько уже встроенных в плагин subject’ов:

- `SubjectiveActorComponent` (на базе `ActorComponent`),
- `SubjectiveUserWidget` (на базе `UserWidget`),
- `SubjectiveActor` (на базе `Actor`).

Этого должно хватить в подавляющем большинстве случаев использования, но вы, естественно, можете реализовать свои subject-классы в C++.

Встроенные механизмы

Класс `Mechanical` - это также интерфейс, с уже реализованными основными механизмами:

- `MechanicalActor` (наследованный от `Actor`),
- `MechanicalGameModeBase` (`GameModeBase`),
- `MechanicalGameMode` (`GameMode`).

Вряд ли вам потребуется дополнять функционал своими механизмами. Если такая потребность возникнет, вы также можете достичь этого при помощи C++.