Отложенные операции

Не секрет, что вы не можете выполнять методы, которые меняют структуру сущности во время твердотельного (Solid) итерирования. Это означает, что вы можете менять только состояние индивидуальных трейтов, но не добавлять или удалять их. Основное преимущество твердотельных цепей - они позволяют параллельное оперирование, что часто является более предподчительным вариантом для мультипоточного выполнения.

Для выполнения подобных операций в твердотельных цепях пригодятся отложенные операции (*Deferreds*). Как выражено в наименовании, они не выполняются немедленно, но откладываются на потом, на более подходящий момент. Пожалуйста, учитывайте, что API отложенных операций доступен только для C++, поскольку вся solid-семантика тоже доступна только в C++ и не существует возможности создания твердотельных цепей в Blueprints.

Установка трейтов

Предположим, мы реализуем игровую стратегию реального времени. Пользователь может выбрать несколько юнитов и дать им задачу. Например, направить их на атаку вражеской позиции. Мы можем отложить установку трейта, пока мы итерируемся параллельно по выбранным юнитам, используя соответствующий API метод. Взглянем на иллюстративный код:

```
FVector Destination = GetUserClickedPoint(); // Получить текущую точку на карте, куда кликнул игрок.
auto SolidChain = Mechanism->EnchainSolid(TFilter<FUnit, FSelected>()); // Объединить в цепь все выбранные юниты.
SolidChain->OperateConcurrently([Destination](FSolidSubjectHandle Unit) // Оперировать над выбранными юнитами параллельно.
{
    Unit.SetTraitDeferred(FMoveToPointOrder{Destination}); // Отложить добавление трейта, который направит юнит в нужную точку.
});
```

Удаление трейтов

Удаление трейтов тоже может быть отложено подобным способом. Вот пример использования API, которое удалит статус «buffed» у юнитов, когда его время иссякнет:

```
SolidChain->OperateConcurrently([DeltaSeconds](FSolidSubjectHandle Unit, FBuffed& Buffed) // Оперировать над выбранными юнитами параллельно.
{
    Buffed.Timeout -= DeltaSeconds; // Уменьшить установленный во время
```

```
инициализации счётчик времени на время, прошедшее с последнего кадра.
    if (Buffed.Timeout <= 0.0f)
    {
        Unit.RemoveTraitDeferred<FBuffed>(); // Отсложенно убираем состояние
использованной способности.
    }
});
```

Также есть возможность удалить все трейты одновременно. Для этого надо вызвать нужный отложенный метод, как это сделано здесь:

```
Unit.RemoveAllTraitsDeferred();
```

Спавн сущностей

Не только трейты могут быть добавлены или удалены через отложенные операции, но полностью вся сущность может быть создана или уничтожена указанным способом. Так, если вы имеете несколько юнитов, которые создают заряженную пулю, вы можете добиться этого таким образом:

```
SolidChain->OperateConcurrently([Mechanism,
DeltaSeconds](FSolidSubjectHandle Unit, FCharging& Charging, FDamageDealer&
DamageDealer)
{
    Charging.Timeout -= DeltaSeconds;
    if (Charging.Timeout <= 0.0f)
    {
        Mechanism->SpawnDeferred(FProjectile{DamageDealer.Power}); //
Создание новой сущности с трейтом FProjectile.
    }
});
```

Удаление сущностей

Процесс удаления сущностей достаточно аналогичен. Убьём все юниты, у которых параметр жизни равен или меньше нуля. Просто сделаем что-то наподобие такого:

```
SolidChain->OperateConcurrently([](FSolidSubjectHandle Unit, FHealth&
Health)
{
    if (Health.Level <= 0.0f)
    {
```

```
        Unit.DespawnDeferred();
    }
});
```

Применение отложенных операций

Когда конкретно? Это довольно логичный вопрос, когда речь идёт об отложенных операциях. Стандартный ответ: «когда придёт время». В наших реалиях это означает, что операция будет применена тогда, когда ассоциированная с механизмом цепь будет удалена и текущее состояние фреймворка станет не-твердотельным (non-Solid).

Стандартное поведение минимизирует задачи программиста и гарантирует, что отложенные операции будут выполнены тогда, когда надо, но, может, вы хотите иметь полный контроль над тем, когда и где произойдёт применение. Это и есть причина, по которой были предоставлены *Применители отложенных операций* (*Deferreds Applicators*).

Применители создаются в отдельных областях видимости вызовом метода UMechanism::CreateDeferredsApplicator, как представлено здесь:

```
{ // Начало внешней области видимости.
    auto Applicator = Mechanism->CreateDeferredsApplicator();
    Mechanism->EnchainSolid(...)->OperateConcurrently([](){
    // Ваша первая механика, которая вызовет отложенные операции.
    });
  // Отложенные операции ещё не применяются.
    Mechanism->EnchainSolid(...)->OperateConcurrently([](){
    // Ваша вторая механика, которая вызовет отложенные операции.
    });
  // Теперь отложенные операции применятся.
} // Конец области видимости.
```

Обратите внимание, что Применитель создаётся в своей отдельной области видимости (обрамлённой фигурными скобками). Применитель будет выполнять отложенные операции, когда он удалится (т.е. выйдет из области видимости). Это гарантируется, поскольку переменная `Applicator` является локальной в указанном `{}` блоке.

From:
http://turbanov.ru/wiki/ - **Turbopedia**

Permanent link:
http://turbanov.ru/wiki/ru/toolworks/docs/apparatus/deferred

Last update: **2022/06/08 22:46**