Simple Hierarchical

Unity

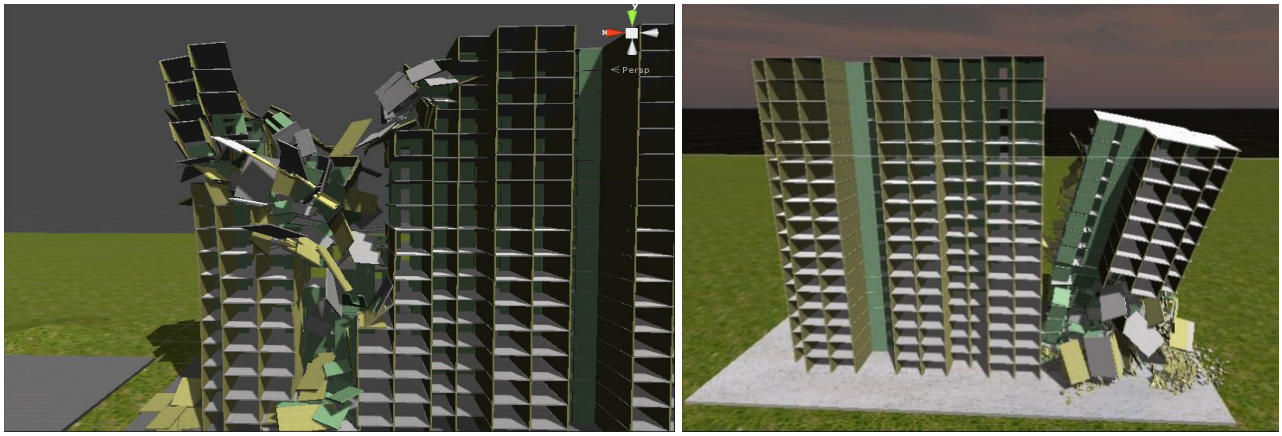
Destruction Reactive System

РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

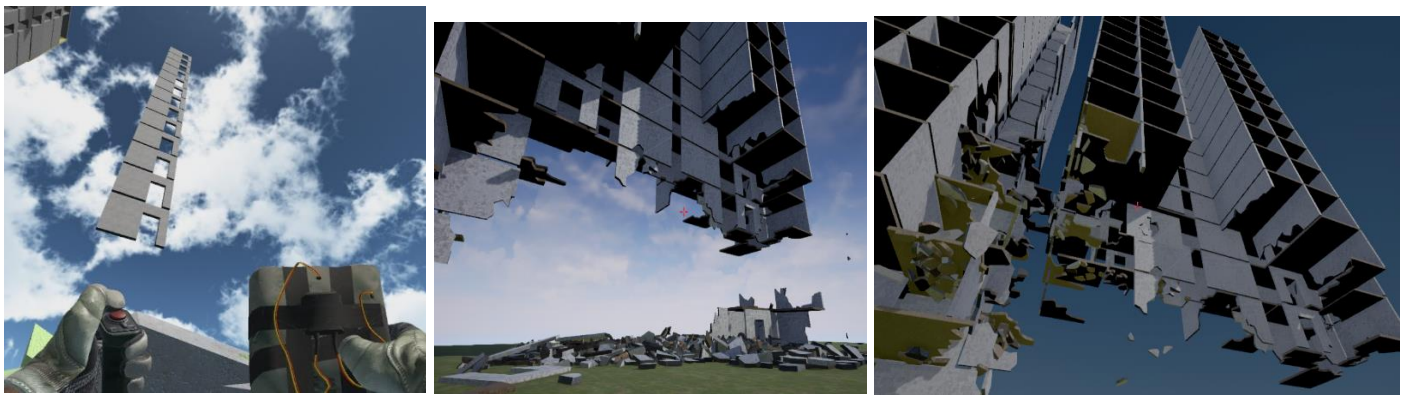
Содержание

История и общее описание

Simple Hierarchical Unity Destruction Reactive System («простая иерархическая реактивная система разрушений для Unity», или SHUDRS, англ. “shudder” – «дрожать» :D Черт, можете называть это всё просто SHDestruction, это уже содержит весь смысл), или SHDestruction (далее почти везде – «система») – это скриптовое решение, решающее проблемы разрушения объектов в Unity во время игры. Хоть это и не совсем проблема, потому что можно использовать Instantiate() для спавна разрушенного объекта на месте целого, когда тот разрушается, и т.д., я испытывал ОЧЕНЬ много проблем во время построения гигантских объектов, таких, как башни, дома, у которых нужно учитывать их собственную устойчивость. Главной проблемой была, как всегда, производительность. PhysX действительно не любит, когда сцена перегружается физическими объектами во время игры, и поэтому мои попытки сделать все стены дома физически активными и/или скрепить их joint’ами вызывали лаги... хм, если честно, это были фризы, при которых время вычисления ***одного*** кадра (deltaTime) доходило до 1500-5000 мс. Плюс, эксперименты в Unreal Engine показали, что, когда PhysX не справляется с нагрузкой, он просто... удаляет проблематичные объекты со сцены. Мне нужно было что-то делать с этим кошмаром.



Красиво, но до чего же лагает...



*Это и не падало, и не работало от слова «вообще». Иди в ж*пу, PhysX…*

На всякий случай: я не пытаюсь сказать, что NVidia плохо работает над своим движком, и что он – вообще неюзабельный. У меня слишком мало опыта, чтобы утверждать такие вещи. В данном случае PhysX меня просто не устраивал.

Итак, спустя какое-то время, я начал представлять разрушаемый объект как структуру, состоящую из более мелких частей, соединенных каким-то известным образом. И таких структур может быть несколько, и они тоже могут быть как-то связаны, и так далее. Информация о соединениях *внутри* структуры дает возможность проверять *соединенность*, т.е. держать вместе частички, которые выглядят как соединенные физически, и разделять структуру на подструктуры в противном случае. Информация о соединениях *снаружи*, *между* различными структурами, дает возможность обрабатывать весь разрушаемый объект иерархическим образом (поэтому система и имеет такое название), часть за частью. Всё это для оптимизации и определенной логики – мы представляем стол, состоящий из досок, дом – из стен, а некоторые стены имеют окна и рамы, и так далее. Как выяснилось впоследствии, такой подход также дает возможность проверять *устойчивость* во всей структуре и отдельных ее частях.

Структура разрушаемого объекта – основная идея

На данный момент, самый большой возможный разрушаемый объект в SHDestruction состоит из Конструкций. Конструкции состоят из более мелких частей, называемых Элементами. Часть каждого Элемента – Фрагмент – это самая маленькая частичка. Важно, что разрушаемый объект не обязательно должен иметь тип «Структура», если он не так велик: это может быть Конструкция или даже Элемент.

Итак, как это работает:

Фрагмент – это единственный элемент разрушаемого объекта (далее - РО), который представлен на сцене физически, потому что только у него может быть коллайдер. Вообще говоря, он *должен* иметь коллайдер, чтобы реагировать на физические воздействия извне.

Каждый раз, когда происходит событие разрушения, неважно, по какой причине, Фрагмент посылает сигнал своему Элементу, который, после получения сигналов ото всех среагировавших Фрагментов (это происходит в цикле Update()), начинает производить проверку соединенности в цикле LateUpdate(). (Порядок срабатывания обработчиков очень важен, поэтому я сделал TimeManager, о нем позже.) Здесь может быть две ситуации: когда разрушение в течение данного кадра привело к разделению Элемента на части, и когда не привело. В первом случае Элемент будет разделен на несколько отдельных Элементов, исполняя логику, что каждый отдельный Элемент в иерархии – это группа Фрагментов, соединенных физически, и только их. Я думаю, это можно называть одной из главных идей. Во втором случае, информация о соединениях корректируется, но от Элемента ничего не отделяется.

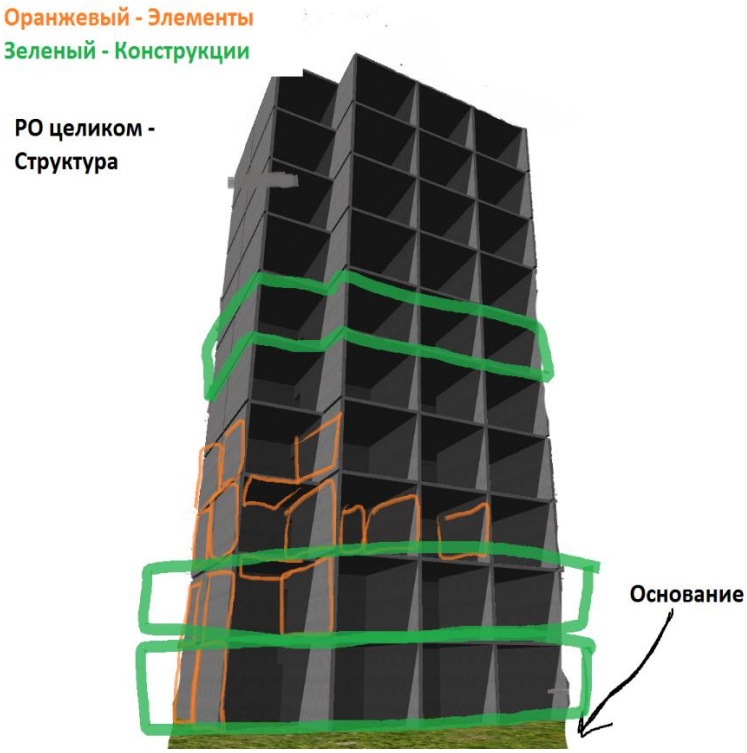
Если вышеописанный процесс происходил в Корневом Элементе (Root Element, Элемент, который не принадлежит РО большего размера), то здесь вычисления останавливаются. В противном случае сигнал от Элемента поступает в Конструкцию, к которой он относится, которая задерживает проверку соединенности до следующего кадра (если быть точным, проверка начнется сразу же после кого, как на следующем кадре отработают все циклы Update(); я использую «yield return null» для задержки, далее смотреть «Unity Event Execution Order» в документации Unity), и затем проверяется соединенность, и, если что-то не так, выделяются подконструкции (т.е. повторяется практически то же самое – смысл один и тот же, хотя код и различается).

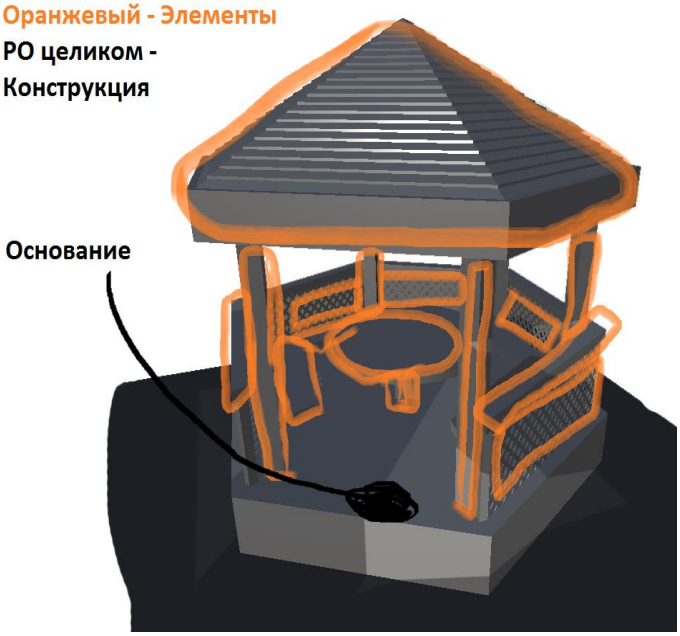
Затем снова – если это была Корневая Конструкция, то это всё, если нет, то нужно послать сигнал в Структуру, к которой принадлежит Конструкция, задержать проверку на один кадр и выполнить ее. После разделения (или не-разделения) Структуры на части, вычисления прекращаются: событие разрушения обработано.

Почему аж четыре уровня обработки? :D Честно говоря, я не знаю, но потом выяснилось, что четыре уровня оптимальнее для производительности, чем два (Фрагменты и Элемент), и их вполне достаточно для любых ситуаций, когда надо что-то сломать.

Зачем растягивать обработку на три кадра? Что ж, правильное разрушение объектов с учетом соединений и прочего – это довольно прожорливая операция, поэтому это было вынужденное решение. Но также растягивание оказалось полезным в том смысле, что так проще контролировать порядок обработки различных уровней (напоминаю, что нужно обязательно сначала обработать все Элементы, затем – все Конструкции, затем – Структуру).

Простые примеры РО и организации иерархии в них:





Они немного специфичны, правда, но я думаю, что вы уловили суть.

Стоп, а что такое «основание»? И как будут вести себя Элементы, которые перестали быть физически соединенными с чем-либо в РО?

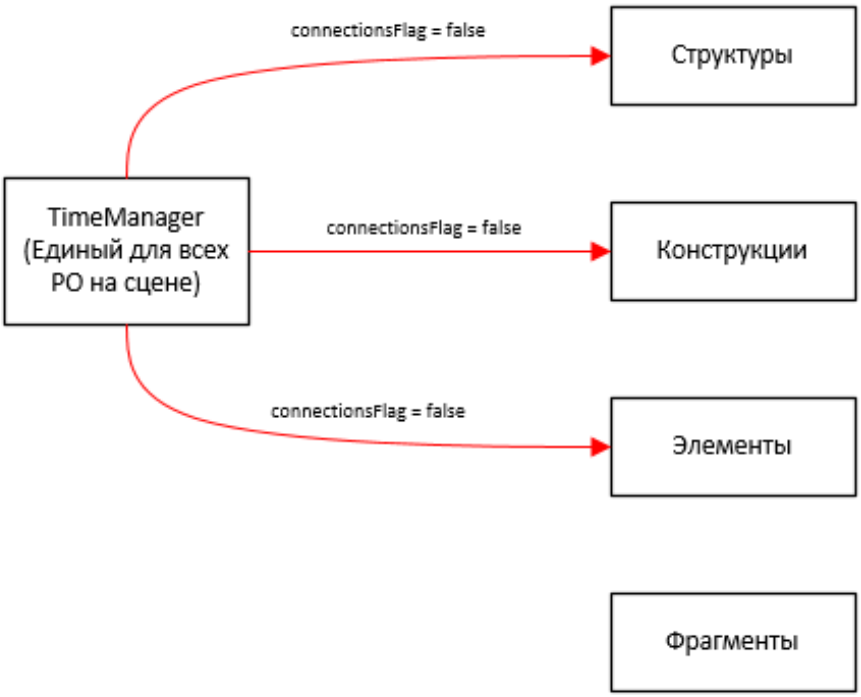
Хорошие вопросы. Здесь используются некоторые компоненты PhysX. Основание – это специальный объект; подразумевается, что он не двигается (статичный) под РО (или над РО, если это неразрушаемый потолок, например). РО использует основание во время инициализации, генерируя информацию о т.н. «связях с землей». Эта информация очень важна, она используется во время вычисления текущей устойчивости, а также при проверке, не находится ли обрабатываемая часть РО в воздухе. Если проверка на связь с «землей» не выполняется, то новый Элемент получает Rigidbody и ведет себя как отдельное физическое тело. Кстати, все Корневые РО имеют Rigidbody в корне иерархии (кинематический, пока все связи с основанием не будут потеряны). Это в каком-то смысле оптимизация, потому что при такой организации не нужно иметь

отдельный Rigidbody на каждом Элементе, а из-за возможности использования т.н. составных коллайдеров физический движок обрабатывает весь РО как ***один*** физический объект.

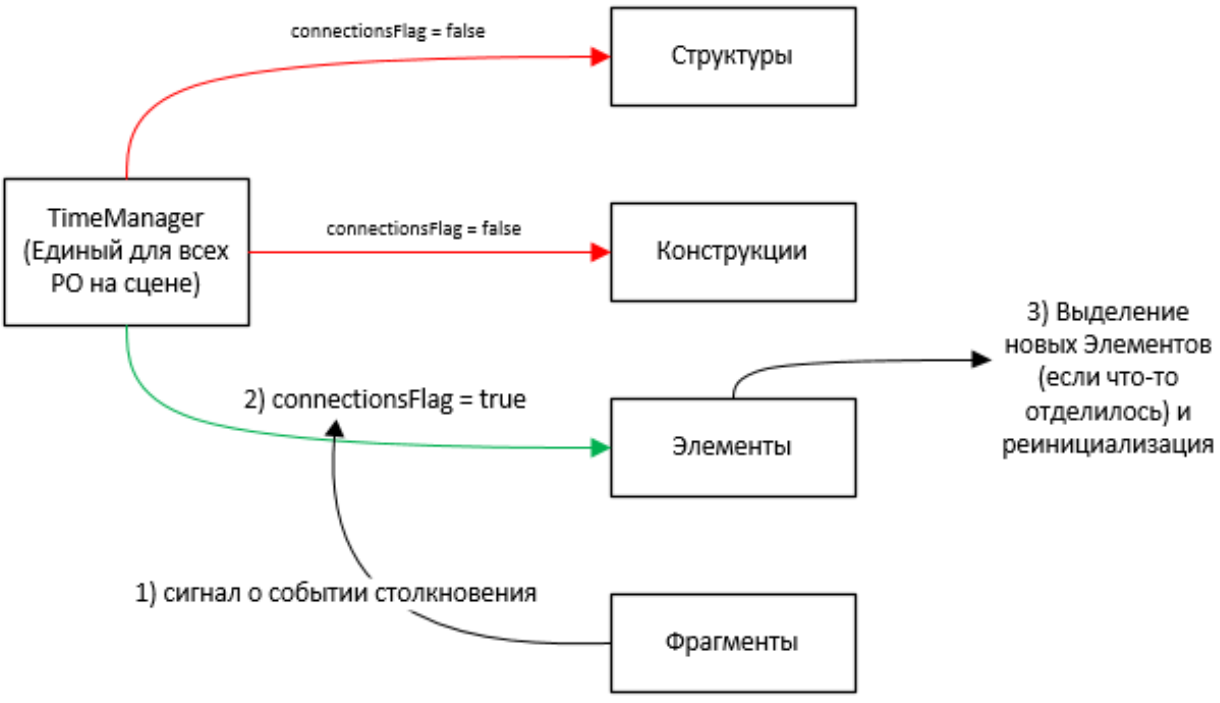
Также важно упомянуть, что, честно говоря, проверка соединенности – это не единственная проверка, происходящая после события разрушения. Каждый уровень проверки соединенности (Элементный, Конструкционный и Структурный) запускает проверку устойчивости во время ре-инициализации РО (которая происходит, когда что-то отделяется от РО). Итак, РО ждет 30 кадров и затем пересчитывает показатели устойчивости, и, если проверка устойчивости не выполняется, система уничтожает некоторую информацию в РО таким образом, что РО начинает падать (как будто бы он из-за гравитации больше не мог стоять ровно). Так как 30 кадров задержки отсчитываются для каждого уровня *отдельно*, то и проверки устойчивости происходят последовательно на каждом уровне (в такой же последовательности, т.е. Элементный – на 30+1 кадре, Конструкционный – на 30+2, Структурный – на 30+3). SHUDRS позволяет настраивать крайнее значение устойчивости (и выключать проверку устойчивости, если нужно), чтобы добиться наилучших результатов.

Что вообще такое подразумевается под «проверкой устойчивости»? Система вычисляет две точки в пространстве для каждого Элемента, Конструкции и Структуры. Это «точка масс» и «точка поддержки». Точка масс – это среднее арифметическое позиций всех Фрагментов, по-прежнему принадлежащих РО. Что-то вроде усредненно распределенной массы. (Хотя и с точки зрения физики, это не совсем так, потому что Фрагменты могут иметь разный объем и плотность, но такое приближение всё еще допустимо.) Точка поддержки – это, по сути (способ вычисления немного меняется для разных типов РО, см. комментарии в коде), среднее арифметическое позиций всех Фрагментов, которые участвуют в соединениях с землей и Фрагментами из других Элементов. В большинстве случаев неустойчивость означает, что точка масс и точка поддержки слишком отдалились друг от друга (вроде того, если бы мы держали очень тяжелый предмет за один конец, он бы упал, потому что второй конец перевешивает). Опять же, это весьма неидеальный подход для сноса странных, неустойчиво выглядящих конструкций, но один из хороших до тех пор, пока я всё еще не хочу работать с PhysX и на полном серьезе внедрять знания из сопромата. :D

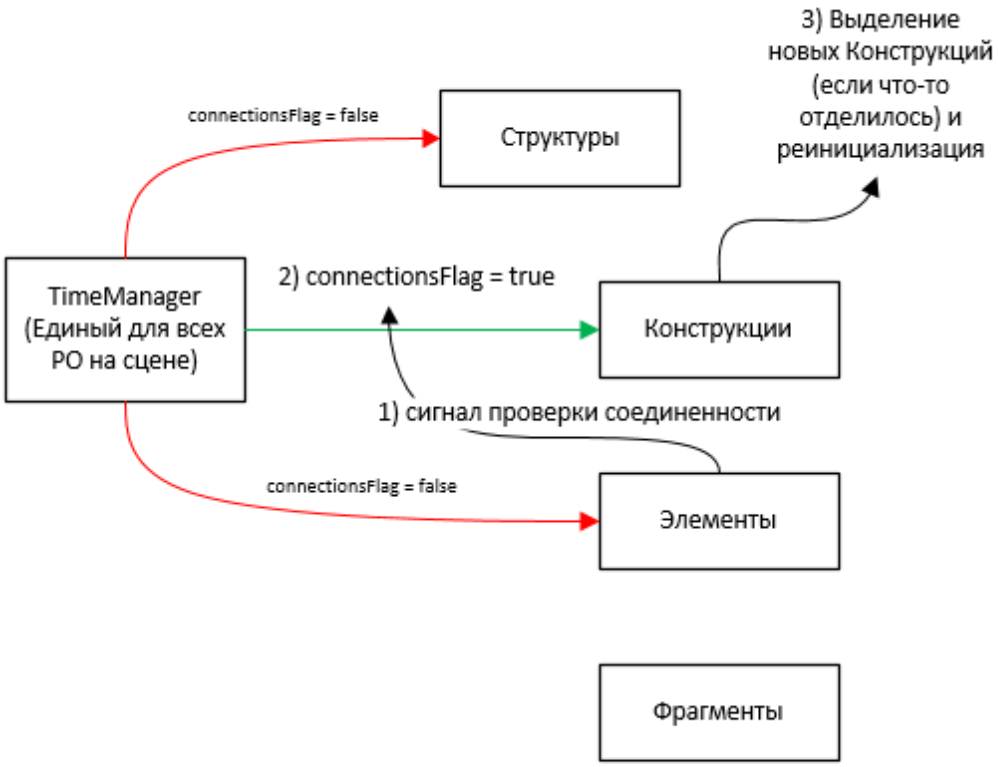
Полная временная схема работы SHUDRS представлена на следующих страницах.



Простой



Кадр 1

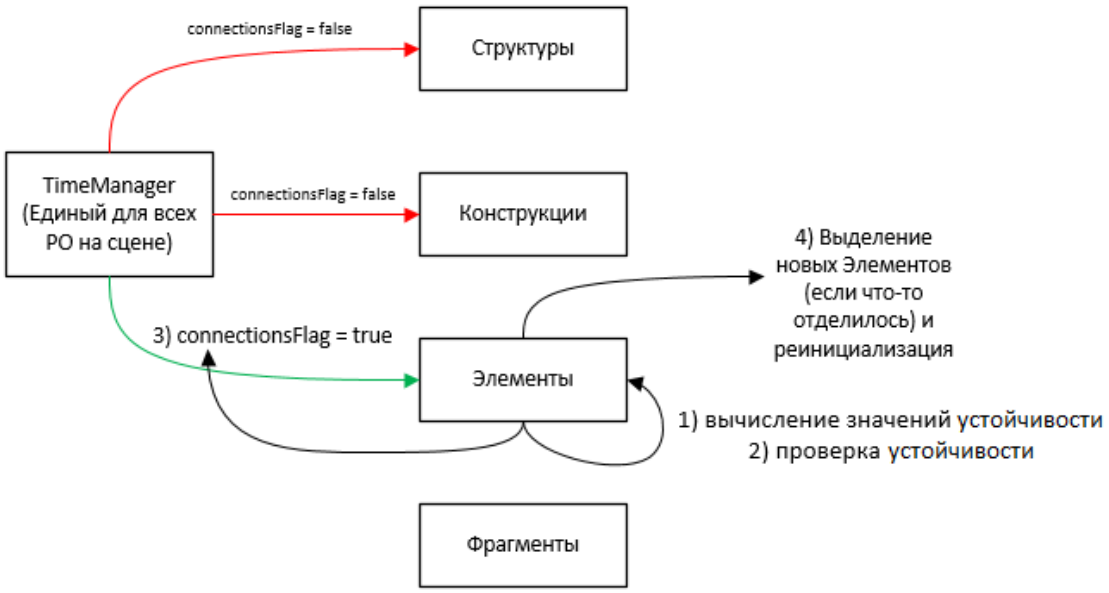

3) Выделение
новых Конструкций
(если что-то
отделилось) и
реинициализация
connectionsFlag = false
Структуры
TimeManager
(Единый для всех
РО на сцене)
2) connectionsFlag = true
Конструкции
1) сигнал проверки соединенности
connectionsFlag = false
Элементы
Фрагменты

Кадр 2

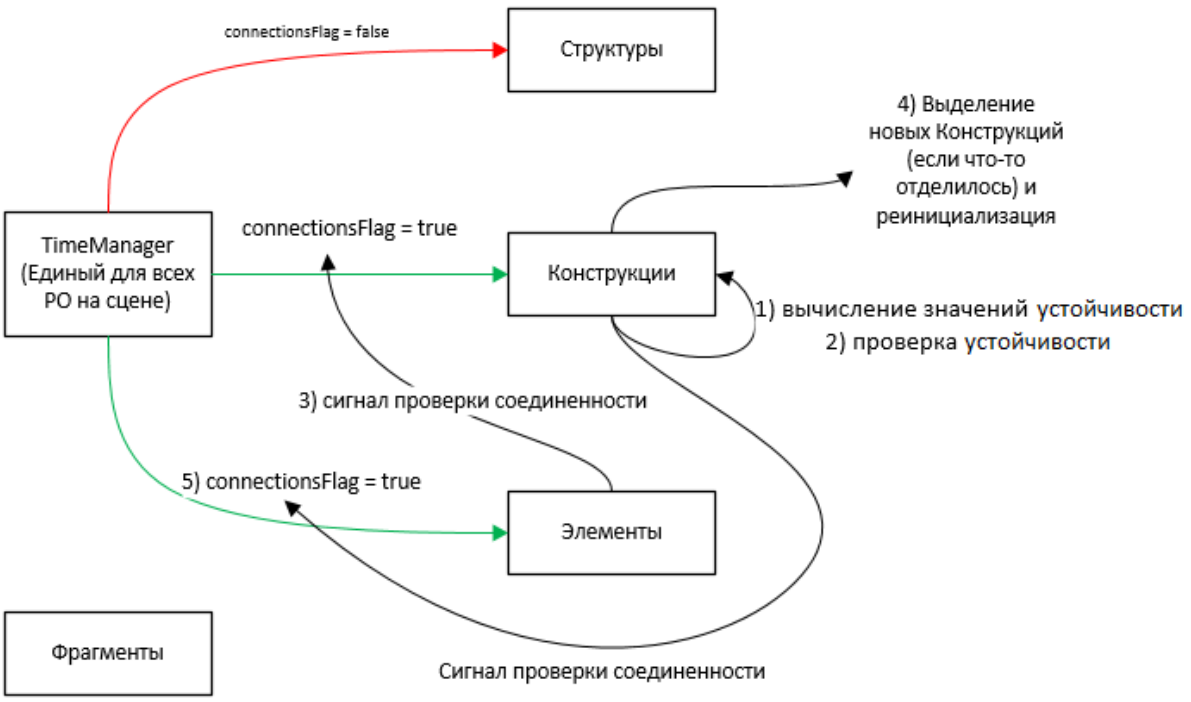

3) Выделение
новых Структур
(если что-то
отделилось) и
реинициализация
2) connectionsFlag = true
Структуры
1) сигнал проверки соединенности
TimeManager
(Единый для всех
РО на сцене)
connectionsFlag = false
Конструкции
connectionsFlag = false
Элементы
Фрагменты

Кадр 3

(Здесь понять схему становится немного сложнее, но это и в принципе не особо нужно вообще: проверки устойчивости выполняются практически всегда, и поэтому никаких РО не выделяется на кадрах 31-33; к тому же, иногда и проверки соединенности не посылают сигналы на следующий уровень, потому что всё нормально, и ничего далее проверять не нужно. :))



Кадр 31



Кадр 32

Пространство имен *SHUDRS*

Полностью состоящая из скриптов, SHDestruction – это сложная система, содержащая несколько скриптов, которые работают вместе, передавая информацию и контроль друг другу. Пространство имен системы содержит:

- Подпространство имен *.Destructibles*. Здесь представлены все скрипты, которые работают непосредственно с РО на сцене.

    - *Fragment* – это класс, чьи объекты должны быть прикреплены к каждому мелкому обломку РО, например, кусок стены или доски и т.д. Требует наличия коллайдера, и из-за этого РО физически представлена на сцене именно набором Фрагментов и может быть через них найдена различными методами класса Physics. Имеет главную точку входа – метод DestroyFragment(), который может быть вызван из пользовательского кода для классических пофрагментных разрушений.

    - *Element* – это базовый абстрактный класс для Корневых и Конструкционных Элементов. По сути, оба являются контейнерами для соединенных Фрагментов и могут их инициализировать их (т.е. производить или обновлять необходимую информацию) и производить проверки соединенности и устойчивости всего Элемента в целом. Имеет матрицу достижимости Фрагментов, используемую во время проверки соединенности, и список структур-соединений, которые представляют собой соединения этого Элемента с другими Элементами. Из-за чудесного использования C#'ом ссылок на объекты практически *повсюду*, всего две ссылки – на «наш» и «чужой» соединенные Фрагменты – дают системе информацию также и о том, кому эти Фрагменты принадлежат в иерархии. :D Реализует интерфейс *IDestructible*.

    - *RootElement* – это такой же контейнер, как и Элемент, но, так как он не принадлежит Конструкции, у него есть Rigidbody, т.е. он является самостоятельным физическим объектом. Методы обработки данных отличаются соответственно. Наследует класс *Element*. Реализует интерфейсы *IDestructible и IRootDestructible*.

    - *ConstrElement* – это контейнер для Фрагментов в случае, когда Элемент принадлежит Конструкции. Опять же, методы обработки имеют немного другое строение по сравнению с Корневым Элементом. Наследует класс *Element*. Реализует интерфейс *IDestructible*.

- *Construction* – это базовый абстрактный класс для Корневых и Структурных Конструкций. Содержит методы (снова...) для инициализации, ре-инициализации и вычисления различных данных о соединенности и устойчивости. Имеет матрицу достижимости своих Элементов (она работает точно так же, как работает матрица достижимости Фрагментов внутри Элемента), которая используется при проверке соединенности. Структуры-соединения, когда они нужны, берутся по ссылкам вложенных Элементов. Реализует интерфейс *IDestructible*.

- *RootConstruction* – это класс, представляющий собой контейнер для Конструкционных Элементов. Может рассчитывать их соединенность и выделять новые Конструкции, содержащие более одного Элемента, соединенных вместе физически. Имеет Rigidbody в корне и никому не принадлежит. Наследует класс *Construction*. Реализует интерфейсы *IDestructible и IRootDestructible*.

- *StructConstrucion* – это контейнер типа Конструкция с примерно похожими методами обработки, но он обязательно является частью Структуры. Наследует класс *Construction*. Реализует интерфейс *IDestructible*.

- *Structure* – самый большой возможный РО. Не может принадлежать чему-то еще. Имеет матрицу достижимости Конструкций, из которых состоит, и может разделять себя на отдельные Структуры. Реализует интерфейсы *IDestructible и IRootDestructible*.

- Также присутствуют два интерфейса, которые были написаны по большей части для того, чтобы дать пользователю доступ к компонентам SHUDRS через его собственные скрипты, например, для возможности принудительно разрушать РО:

  - *IDestructible* – это интерфейс, содержащий две пользовательские точки входа в систему – одна для отсоединения полностью целого Элемента/Конструкции/Структуры (например, в случае обвала пола, крыши и т.д.), вторая – для превращения связанного РО в РО абсолютно без связей (он будет вести себя как «карточный домик»; а Элемент и вовсе рассыплется на обломки). Очевидно, что обе эти точки могут быть вызваны и без помощи интерфейса из самих контейнеров (такие методы

помечены комментарием "*(can be accessed through interface)*"). Но я думаю, что в использовании интерфейса есть некоторые преимущества, например, можно найти Фрагмент какого-то РО, получить из родительской иерархии объект данного интерфейса и вызвать метод разрушения, не разбираясь, РО какого типа мы нашли. И эти оба интерфейса могут быть расширены самим пользователем какими-то методами или свойствами, если ему так будет нужно. :)

- *IRootDestructible* также является интерфейсом для бестипного доступа к корню иерархии (я уже использовал объект этого интерфейса для подобной задачи в методе OnCollisionEnter() во Фрагменте, можете найти).

КСТАТИ, да, я совсем забыл, я *специально* сделал систему такой гибкой, особенно классы из подпространства *.Weaponry*, чтобы вы, мои друзья, мои расширять ее для своих нужд. ;)

Но нам нужно продолжать:

- Пространство имен *.Weaponry* содержит два базовых абстрактных класса, к которым вы ***обязательно*** должны создать наследников. Это два типа оружейных снарядов, которые имеют возможность производить разрушения по умолчанию. Эти классы:

  - *Projectile* – это тип снарядов, которые летают в воздухе, на них действует гравитация и трение воздуха, и воздействуют на объекты, когда сталкиваются с ними во время полета, т.е. эффект такого снаряда проявляется при столкновении. Некоторые примеры подтипов: пуля, танковый снаряд, ракета, пушечное ядро, и так далее. Этот базовый класс содержит пару специальных методов, таких, как поиск столкновений с помощью луча (поэтому коллайдер для работы не нужен, только Rigidbody, а длина луча – расстояние чуть большее, чем то, которое снаряд пройдет за данный кадр), параметры, такие, как маска столкновений и эффект при ударе; и вы можете добавить какие-то собственные свойства в классах-наследниках. Класс Bullet дан как пример реализации. Встроенный метод PerformDestruction() дан как пример метода, с помощью которого можно вызывать разрушения снарядным оружием.

  - *Explosive,* напротив, проявляет себя не от удара, а при *активации*; ему не надо сталкиваться с чем-то. Как граната, или, я не знаю, пробивной

заряд. Так же, как и в классе Projectile, здесь описаны некоторые базовые свойства и методы, и вам нужно создать наследников, расширяющих этот класс. Встроенный метод PerformDestruction() дан как пример метода разрушения для взрывающего оружия.

Почему я сделал эти классы такими «бессмысленными» и зачем я их сделал вообще? Что ж, мне показалось удобным иметь такую структуру типов вооружения, с помощью которых можно уничтожать вещи, потому что такая структура четко определена и в то же время имеет большую гибкость, простор для расширения. Да и, как выяснилось позже, более комфортно оказалось пускать из снарядов лучи и «оверлап»-сферы, чем пытаться обработать все события столкновения в методе OnCollisionEnter() внутри каждого Фрагмента. (И, кстати, быстрые объекты очень любят проходить друг сквозь друга без какого-либо эффекта, а если включить постоянное определение столкновений, то позиция удара всё равно будет неправильной из-за того, что удар произошел между физическими кадрами, а позиция записалась уже во время очередного кадра.) Но, если очень хочется, вы можете переписать мою систему вооружения под себя, как вам удобнее. Пока не трогаете скрипты из подпространства *.Destructibles*, всё должно работать.

И остались классы с префиксом *Editor_*, которые связаны целиком не с моей системой разрушений, а с написанием кода для кастомного Инспектора Unity, и парочка классов-помощников. Вот они:

- *DestructibleBaseObject* – это класс, который нужен только затем, чтобы его объектами помечать объекты на сцене как статические для системы SHURD. Если на GameObject'е есть объект этого класса, подразумевается, что этот объект не будет ни двигаться, ни ломаться, ни исчезать, и т.д., т.е. РО на него опирается (при инициализации, например, с помощью этих объектов определяется, какие Фрагменты касаются «земли»). Если не соблюдать это правило, то это приведет (скорее всего) к нежелательным результатам вроде зависания РО в воздухе.

- *DestructionUtility* – это файл-помощник, содержащий структуру «Соединение» и класс «Матрица достижимости», написанные специально для системы разрушений, и пустой статический класс (я думал, что он понадобится, как можно увидеть из комментария над ним, но теперь он даже немного иронично выглядит :D). Структура «Соединение» нужна для хранения физических связей между Фрагментами, относящимися к разным Элементам, и содержит две

ссылки – на оба Фрагмента (первый принадлежит тому Элементу, который хранит экземпляр структуры). Класс «Матрица достижимости» - это моё собственное специфичное представление стандартного класса .NET BitArray (пришлось переписать, потому что последний не сериализуется в Unity) и специально устроен так (в особенности, его индексатор), чтобы было максимально удобно работать с матрицами достижимости невзвешенного графа.

- *FragmentationSettings* – один объект этого класса (хотя бы один!) должен быть на каждом РО в корне иерархии. Их может быть и больше (т.е. такой объект может «висеть» и где-нибудь в середине иерархии на каком-нибудь контейнере Фрагментов). Представляет собой расширенные настройки разрушения – тэги и подстроки имен, на что следует реагировать; признак материала; системы частиц, показывающиеся при разрушении и так далее – для набора Фрагментов, кому он является по иерархии «родителем». Если такие настройки есть еще где-то внутри иерархии, и глубина нахождения больше, чем у других настроек, то первые перезапишут вторые (т.е. настройки для Фрагментов отдельного Конструкционного Элемента перезапишут настройки для всех Фрагментов РО, висящие на корне, и далее в таком же духе).

- *TimeManager* – это класс, хронометрирующий процессы, происходящие внутри РО, такие, как послойные проверки на соединенность и устойчивость. Когда его не было, я примерно представлял порядок выполнения операций на различных слоях – Элементом, Конструкционном и Структурном – но не был до конца уверен, что система не будет творить, что ей вздумается в некоторых ситуациях (управление временем в Unity всегда было делом непростым, что поделать), поэтому решил написать этакий «генератор сигналов», последовательно выполняющий проверки в LateUpdate(). Так что на сцене, чтобы РО разрушались, нужно иметь ***ровно один*** объект этого класса (и если вы не добавите его сами, об этом позаботится один из контейнеров в РО).

Весь код системы отлично прокомментирован (я уж действительно постарался, чтобы ничего не осталось загадкой, почему оно сделано именно так :D), так что легко понять, зачем нужен тот или иной метод или поле. :)

Почему я не использовал многопоточность? Что ж, в моем случае, это было бы слишком сложно сделать, а производительность вряд ли бы

улучшилась достаточно сильно. Много объектов, у каждого собственный поток (а если создавать пул, есть вероятность блокирования процессов одного другими), плюс подавляющее количество вычислений выполняется напрямую с объектами классов, наследующих MonoBehaviour, что запрещает использование вторичного потока.

Но если вы захотите-таки улучшить систему, введя многопоточность – пожалуйста, я не стану вас останавливать; главное, не забудьте поделиться результатами, мне ж тоже интересно. :)

Почему я не предоставляю качественную документацию (такую, как Unity Scripting API)? Я думаю, достаточно настоящей инструкции, потому что все методы и поля, кроме обозначенных точек входа в систему, в принципе не должны вызываться и изменяться пользователем соответственно и беспокоить его (чтобы избежать риска сломать всю систему :D); а точки входа и без того хорошо описаны здесь.

Как использовать?

- Начальная настройка разрушаемого объекта (РО):

1. Составьте иерархию будущего РО таким образом:

Структура *(опционально)*

Конструкция1 *(опционально)*

Элемент1

Фрагмент1

Фрагмент2

…

Элемент2 *(опционально)*

Фрагмент1

Фрагмент2

…

…

Конструкция2 *(опционально)*

…

…

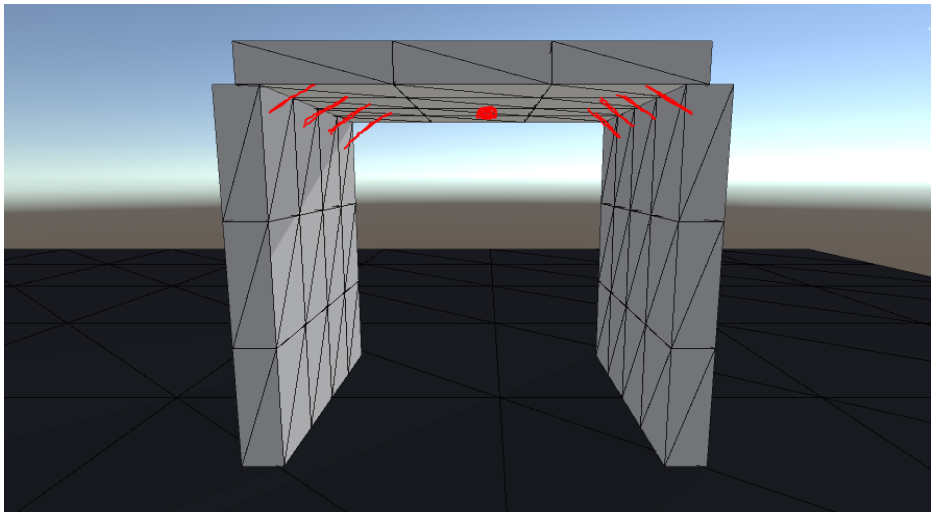
“(опционально)” означает, что РО может быть необязательно Структурой, но Элементом или Конструкцией, и на каждом уровне может быть сколько угодно Конструкций, Элементов и Фрагментов. Короче, просто постройте *иерархическое* представление своего объекта. Убедитесь в том, что все объекты в иерархии – это Фрагменты или контейнеры, содержащие их. В иерархии ***не*** должно быть «мусора» - иначе эти объекты также будут использованы системой в качестве Фрагментов/контейнеров.

2. Добавьте на корень иерархии соответствующий размеру иерархии скрипт (Корневой Элемент, Корневая Конструкция или Структура) и кликните по кнопке “Initialize!” Скрипт сделает всё остальное за вас, добавив соответствующие скрипты на контейнеры и Фрагменты.
3. Если после инициализации вы что-то поменяли в иерархии объекта (это разрешается), то потом перед использованием в игре еще раз кликните “Initialize!” в корне.
4. Если вам хочется очистить все объекты иерархии РО от скриптов, созданных системой, нажмите кнопку «Cleanup» на корневом скрипте.

Для базовой настройки РО – это всё! Но также есть возможность более детальной настройки.

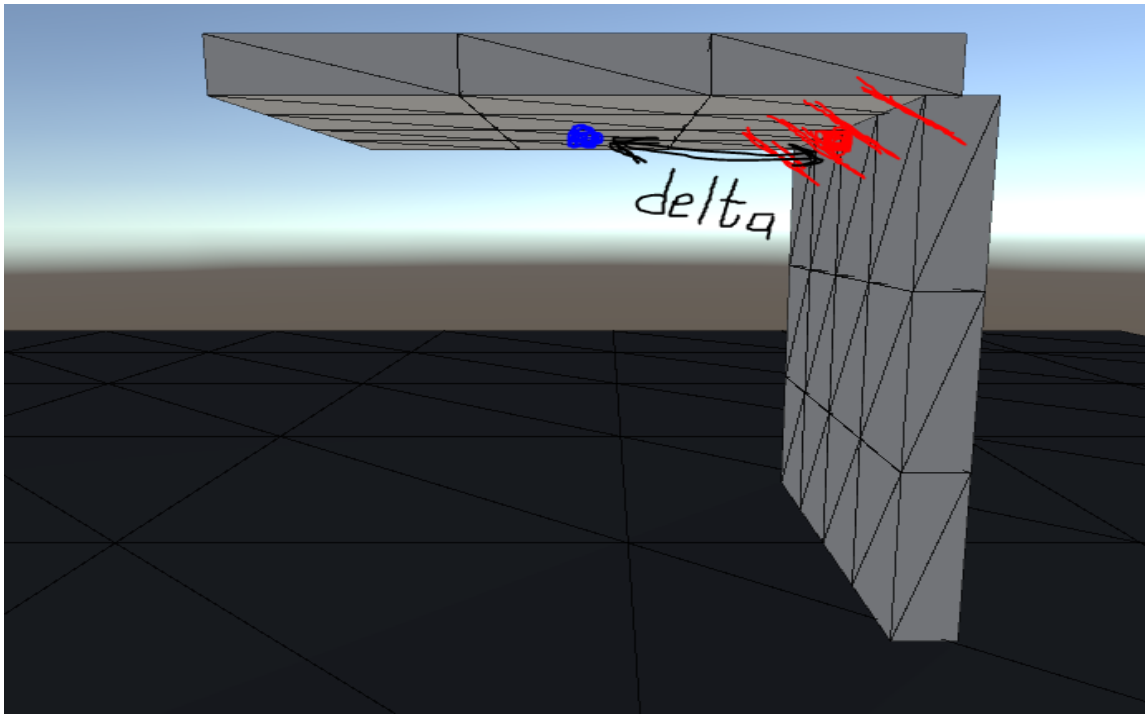
- Подборка крайнего значения устойчивости:

У каждого контейнера, такого, как Элемент, Конструкция или Структура, есть свойство, называющееся «Крайнее значение стабильности», Stability Edge Value. Вообще говоря, это максимальная длина вектора между точками масс и поддержки (смотря код, можно понять, как они вычисляются для разных типов контейнеров, а они вычисляются по-разному), или, если проще, максимальное возможное расстояние между эти двумя точками. Давайте представим «стол», состоящий из трех досок-Элементов:



В начале точка поддержки верхней доски будет в ее центре, потому что слева и справа есть по четыре Фрагмента, соединяющих доску с другими досками. Кстати, точка масс, как среднее арифметическое позиций Фрагментов доски, будет тоже в центре.

Но если мы уничтожим одну из досок, на которую опиралась верхняя доска…



Точка масс (синяя) осталась на том же месте, поскольку никакие из Фрагментов самой доски уничтожены не были, но точка поддержки (красная) теперь там, где центр оставшихся поддерживающих Фрагментов. Длина дельта-вектора между ними (черный) – это то, что мы и хотим измерить и сравнить с заданным крайним значением, чтобы понять, устойчив ли РО всё еще. Если нет, то РО должен упасть.

Итак, что в итоге? Иногда нужно поднастроить крайнее значение устойчивости: если оно слишком мало, объекты будут казаться слишком неустойчивыми, если велико – будут всегда оставаться странновато устойчивыми.

Три дополнительных момента:

1. Точки масс и поддержки вычисляются для каждого типа РО по очень разным методам, и я бы очень посоветовал посмотреть исходный код.
2. Глобальное (для всех РО) значение этого параметра по умолчанию может быть задано в коде, там, где объявляется эта переменная (значение после знака «равно» - и есть значение по умолчанию).

3. Если задать это значение меньше или равным 0, то это отключит проверку устойчивости в данном РО.

- Оптимизации для рендера:

Если Элемент или Конструкция еще не были повреждены, то есть возможность отображать (рендерить) не Элемент/Конструкцию как набор Фрагментов, а как целиковый объект. Если у вас есть меш неповрежденного Элемента/Конструкции, добавьте на соответствующий контейнер Mesh Filter и Mesh Renderer и разместите в них меш и материал. Затем поставьте у этого контейнера галочку «Переключать реднереры» («Use Renderers Switch»). Система сделает всё остальное сама. Заметьте, что если компонентов, нужных для рендера, на объекте не окажется (а галочка стоит), система выключит переключение сама.

- Дополнительные свойства Фрагмента:

- Уничтоженный объект (Damaged GO) – если какой-то Фрагмент после вылета из Элемента должен выглядеть из-за повреждений как-то иначе (например, металлическая труба погнется, и т.д.), то в это поле нужно поместить префаб с поврежденной версией данного Фрагмента. Нужны, по факту, только Mesh Renderer и Mesh Filter на этом объекте.

- Неразрушаемый Фрагмент (Fragment Is Indestructible) – некоторые РО имеют в своей конструкции части, которые не уничтожаются никогда (как, например, арматура в бетоне или стальной каркас дома), и, так как лишний неразрущающийся «мусор» в иерархии запрещен, можно добавить эти объекты в состав РО и пометить данной галочкой. Они не могут быть разрушенными ничем, но по-прежнему являются функциональной частью РО.

- Начальное здоровье (Start Health) – просто числовой параметр из 90-х, когда у всего в играх была шкала здоровья, и когда она обнулялась, то объект умирал. :D Если вам нужно подобное поведение, используйте этот параметр на своё усмотрение. Это значение будет в начале игры записано

как текущее. Текущее значение здоровья – внутренний параметр, доступный для пользовательского кода.

- Использование компонента «Fragmentation Settings»:

Каждый РО содержит один такой компонент в корне иерархии, но могут быть и компоненты, прикрепленные к контейнерам, задающие локальные настройки (перезаписывающие глобальные «корневые»). Какой компонент будет ближайшим для Фрагмента в плане глубины иерархии, тот и будет использован. Его всегда можно найти, т.к. Фрагмент отображает название объекта, которому принадлежат эти настройки.

По умолчанию настройки содержат:

- Массивы строк с тэгами и подстроками имен; если объект имеет тэг или подстроку в имени, которые присутствуют в этих массивах, и если объект коснулся Фрагмента с этими настройками, то Фрагмент будет двигаться под воздействием столкновения. Массивы называются «tagsToMove» и «nameSubstringsToMove» (соотв. «тэги, чтобы двигаться» и «подстроки имен, чтобы двигаться»). Второй нужен, потому что иногда всё, что нужно, тэгами не пометишь.
- Строка «materialTag» и массив строк «specialTags» - это еще один способ как-то разнообразить возможность воздействия на Фрагменты. Первый параметр – это материал Фрагментов (к примеру, можно пометить их как стекло или камень и обрабатывать соответствующим образом); второй – дополнительные строки, которые пользователь может задавать и обрабатывать сам.
- «directionalDebris» и «goDownDebris» - это GameObject’ы с «шаблонными» системами частиц («шаблонный» означает, что при столкновении материал и форма частиц будет задана системой автоматически). Первая система частиц – это выброс крупинок Фрагмента, как будто в него что-то врезалось на большой скорости. Вторая – крупинки просто осыпаются на землю (используется при вылете Фрагмента из Элемента). Пара примеров есть в комплекте, но вы всегда можете сделать свои собственные системы частиц.

- Возможности для дебаггинга:

Так как SHUDRS – это не совсем простая штука, и мне надо было бороться с багами, я вводил несколько возможностей визуализации данных:

1. Есть возможность печати матрицы достижимости контейнера путем нажатия соответствующей кнопки на компоненте. Если вы хорошо знаете теорию графов, и как граф описывается матрицей достижимости, то это может помочь вам в устранении проблем, когда объекты остаются связанными, когда не должны (и наоборот).
2. Показ точек устойчивости с помощью Gizmos (Show Stability Gizmos) – если поставить эту галочку, то точки масс и поддержки контейнера будут графически видны в окне сцены и игры.

- А что насчет РО внутри или присоединенного к другому РО?

В первом случае просто оставьте его внутри, сняв галочку «isKinematic» на Rigidbody корня (как журнальный столик внутри комнаты здания). Во втором, как в случае с настенными лампами, сделайте их частью того РО, к которому они крепятся. **Ни в коем случае** нельзя помечать стену как «статическую» добавлением DestructibleBaseObject – помните, помеченные таким образом объекты не должны ни двигаться, ни разрушаться!

- Можно ли помещать какие-то объекты в иерархии в качестве потомков Фрагмента?

Хм, надо бы взглянуть…

Пытается найти документы о том, как работает его система

Устраивает на столе жуткий бардак

Вспоминает, что он не писал никаких документов об этом

А, да, конечно, можно! :D Надо лишь помнить о нескольких вещах, вроде той, что эти объекты будут удалены вместе с Фрагментом после его разрушения, и т.д.

Известные проблемы

На данный момент имеются следующие проблемы, которые, мне кажется, стоит записать здесь.

- Иногда проверка устойчивости говорит, что PO (или его часть) устойчив, но она кажется неустойчивой, и настройки не помогают. Это не баг, скорее недостаток системы, в которой сложные параметры вычисляются с использованием сильных приближений и упрощений. Извините :)
- В некоторых совсем уж сумасшедших случаях может (наверное) происходить конфликт вычислений в PO, но, я надеюсь, что это всего лишь моя паранойя :D

Подводя итоги

SHUDRS – это довольно простая, легкая для физ.движка и довольно быстрая в обработке реактивная система для уничтожения больших вещей, принимая во внимание их параметры устойчивости. Ее можно использовать для создания гигантских структур и их обработки/разрушения без создания головоломного кода. Система имеет множество возможностей для расширения и позволяет полностью контролировать процесс разрушения, как вам этого хочется.

Обратная связь и сказать «спасибо» :)

Меня зовут Дмитрий, в сети известен как NotSoOld (также DimikYoo или TheDimaSomov), я 20-летний программист из России. Если у вас есть какие-нибудь вопросы или предложения, можете написать мне сюда: 1000lop@gmail.com.

Сказать «спасибо»:
PayPal: 1000lop@gmail.com
Яндекс.Деньги: 4100 1150 2730 370

БОНУС: как это сделано в Battlefield

Когда кто-то говорит о разрушаемости, первая игра, приходящая всем на ум – это Battlefield (части 3, 4 или 1; очень старые игроки еще, возможно, вспомнят Bad Company 2 :D). Я играл в эту серию игр действительно много и делал пометки, наблюдая, как всё вокруг меня превращается в пыль где-нибудь на Каспийской границе или в Шанхае.

Собственно, главная идея в Battlefield – «Не нужно усложнять вещи слишком сильно. НИКОГДА.» Что я имею в виду? Если вы поиграете в игры, упомянутые выше, вы не найдете ни одного интерактивно разрушаемого дома выше, чем в 2-3 этажа. Они состоят из особых объектов (назовем их «стены»), которые могут быть уничтожены каждый сразу и целиком чем-нибудь мощным, и когда количество уничтоженных «стен» достигает критического максимума, вся конструкция сразу же падает. Разрушение каждой «стены» и падение конструкции заранее записано в анимацию, поэтому вообще не грузит физ.движок и неплохо выглядит для неискушенного глаза. Двухэтажные строения могут иметь специальный (но тривиальный) скрипт для просчета их устойчивости.

Иногда я примечал дома, которые были сами по себе неразрушаемые, но имели «модульные» фасады, которые так и отваливались - модулями.

Всё, что имеет б*о*льшие размеры, пре-анимировано, и в некоторых случаях, вокруг генерируется настолько мало пыли, что видно, как объект уходит под землю, и на его месте вырастают его обломки. Жесть. :D

Некоторые PO сделаны по принципу Элементов в моей системе – они используют прикрепление к недвижимой земле и матрицу достижимости (вспомните деревянные изгороди).

Итак, каков же будет вывод? DICE проделали действительно хорошую работу, создав ошеломляющие и любопытные интерактивные объекты практически из... *ничего*. Я ими очень горжусь. Может быть, иногда действительно надо всего лишь не думать о чересчур сложных вещах и знать, когда пора остановиться, и успех будет обеспечен. (И, к сожалению, это точно не про меня. :D)


E
42
[HEAT]Nightmareg


45
90
30MM CANNON READY
29 G
340.2 B
100
268.7