3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

Разработанный физический движок, состоит из четырех частей:

– силовые генераторы (и генераторы крутящего момента) исследуют текущее состояние игры и рассчитывают, какие силы нужно применять к каким объектам;

– симулятор твердого тела обрабатывает движение твердых тел в ответ на эти силы.

– детектор столкновений быстро находит столкновения и другие контакты как между объектами, так и между объектом и неподвижными частями уровня. Детектор столкновений создает набор контактов, которые будут использоваться преобразователем столкновений;

– преобразователь столкновений обрабатывает набор контактов и регулирует движение твердых тел так, чтобы точно отобразить их состояние.

Каждый из этих компонентов имеет свои внутренние детали и сложности, но мы можем рассматривать их как отдельные единицы. Эти компоненты можно более подробно разделить на следующие группы классов:

1) Класс `Particle`.
2) Класс `ParticleForceGenerator` и его производные.
3) Классы `ParticleSpring`, `ParticleAnchredSpring`, `ParticleBugee`, `ParticleBouyancy`, `ParticleFakeSpring`.
4) Класс `ParticleContact` и обработка столкновений.
5) Класс `ParticleLink` и его производные.
6) Класс `ParticleWorld`.
7) Класс `RigidBody`.
8) Обнаружение столкновений.

3.1 Класс Particle

Ньютон создал три закона движения, которые с большой точностью описывают, как ведет себя точечная масса. Точечная масса – это то, что мы далее будем называть частицей, но ее не следует смешивать с физикой частиц, изучающей крошечные частицы, такие как электроны или фотоны, которые определенно не соответствуют законам Ньютона.

Помимо частиц нам нужна физика вращения, которая вводит дополнительные усложнения, которые были добавлены к законам Ньютона. Однако даже в этих случаях можно применить законы о точечной массе.

Частица имеет положение, но не ориентацию. Другими словами, мы не можем сказать, в каком направлении направлена частица: это либо не имеет значения, либо не имеет смысла. Для каждой частицы нам нужно отслеживать различные свойства: нам понадобятся ее текущее положение, скорость и ускорение. Положение, скорость и ускорение являются векторами. Далее нам понадобится ввести первые два закона Ньютона.

Первый закон рассказывает нам, что происходит, если вокруг нет никаких сил. Объект будет продолжать двигаться с постоянной скоростью. Другими словами, скорость частицы никогда не изменится, и ее положение будет постоянно обновляться в зависимости от скорости. Это может быть не интуитивно: движущиеся объекты, которые мы видим в реальном мире, замедлятся и остановятся, если они не будут постоянно вынуждены двигаться вперед. В этом случае объект испытывает силу – силу сопротивления (или трение, если он скользит вдоль чего-либо). В реальном мире мы не можем уйти от сил, действующих на тело; мы можем представить себе движение предметов в пространстве. То, о чем говорит этот закон, состоит в том, что если бы мы могли удалить все силы, тогда объекты продолжали бы двигаться с одной и той же скоростью.

В нашем физическом движке мы могли просто предположить, что на частицы не действуют никакие силы и напрямую использовать первый закон. Чтобы имитировать сопротивление, мы могли бы добавить специальные силы сопротивления. Это хорошо для простого движка, но это может вызвать проблемы с более сложными системами. Проблемы возникают из-за погрешностей при работе с числами с плавающей точкой. Эти погрешности могут привести к тому, что объекты будут двигаться быстрее из-за собственной согласованности.

Лучшее решение – включить грубое приближение сопротивления непосредственно в движок. Это позволяет нам убедиться, что объекты не ускоряются с помощью числовых погрешностей, и это может позволить нам моделировать некоторые виды сопротивлений. Если нам нужно сложное сопротивление (например, аэродинамическое сопротивление в симуляторе полета или гоночной игре), мы все еще можем сделать это длинным путем, создав специальную силу сопротивления. Чтобы избежать путаницы, мы называем простую форму демпфирования сопротивлением.

При движении тела мы будем удалять часть скорости объекта при каждом такте обновления. Параметр демпфирования определяет скорость этого замедления. Если демпфирование равно нулю, то скорость будет сведена к нулю: это будет означать, что объект не сможет поддерживать никакого движения без постоянной силы и будет выглядеть странно. Значение равное единице означает, что объект сохраняет всю свою скорость (что эквивалентно отсутствию демпфирования). Так как мы хотим, чтобы движение объектов выглядело реалистично, то значения меньшие единицы, но близкие к ней, оптимальны.

Второй закон Ньютона дает нам механизм, посредством которого силы изменяют движение объекта. Сила – это то, что изменяет ускорение объекта (т.е. скорость изменения скорости). Следствием этого закона является то, что мы не можем ничего сделать, чтобы объект мог непосредственно изменить свое положение или скорость; мы можем сделать это косвенно, применяя силу, чтобы изменить ускорение и ждать, пока объект достигнет нашей целевой позиции или скорости.

Из-за второго закона мы будем рассматривать ускорение частицы отдельно от скорости и положения. И скорость, и положение отслеживаются от кадра к кадру во время игры. Они меняются, но не напрямую, а только под влиянием ускорений. Ускорение, напротив, может меняться от одного момента времени к другому. Мы можем просто установить ускорение объекта по своему усмотрению, и поведение объекта будет выглядеть реалистично. Если мы непосредственно зададим скорость или положение, частица будет казаться дрожащей или прыгающей. Поэтому поля положения и скорости будут меняться только внутренними процессами и не должны изменяться вручную (за исключением установки исходного положения и скорости для объекта). Поле ускорения может быть установлено в любое время.

Добавив положение и скорость, которые мы сохраняем для каждой частицы, необходимо сохранить ее массу, чтобы мы могли правильно рассчитать ее отклик на силы. Многие физические движки просто хранят скалярное значение массы для каждого объекта. Тем не менее, есть лучший способ получить тот же эффект.

Если масса объекта равна нулю, то ускорение будет бесконечным, если сила не равна нулю. Это не та ситуация, которая должна когда-либо произойти: ни одна частица, которую мы можем моделировать, никогда не должна иметь нулевую массу. Если мы попробуем смоделировать частицы с нулевой массой, это вызовет ошибки деления на ноль в коде.

Однако, часто бывает полезно моделировать бесконечные массы. Это объекты, которые никакая сила не может сдвинуть. Они очень полезны для неподвижных объектов в игре: стены или пол, например, нельзя перемещать во время игры. Если мы используем бесконечную массу, то ускорение, как мы и ожидаем, будет равно нулю. К сожалению, мы не можем представить истинную бесконечность в большинстве компьютерных языков, а оптимизированные математические инструкции на всех распространенных процессорах не справляются с бесконечностями. Идеальное решение, в котором легко получить бесконечные массы, но невозможно получить массы нулевые. Решением будет хранить обратную массу. Это решает нашу задачу для представления объектов нулевой или бесконечной массы: бесконечные объекты массы имеют нулевую обратную массу, которую легко установить. Объекты нулевой массы имели бы бесконечную обратную массу, которая не может быть указана в большинстве языков программирования.

В каждом кадре движок должен поочередно просматривать каждый объект, выработать его ускорение и выполнить шаг интегрирования. Расчет ускорения в этом случае будет тривиальным: мы будем использовать только ускорение силы тяжести.

Метод интегрирования состоит из двух частей: одна для обновления положения объекта, а другая для обновления его скорости. Положение будет зависеть от скорости и ускорения, а скорость будет зависеть только от ускорения.

Интегрирование требует интервала времени, в течение которого

необходимо обновить позицию и скорость: поскольку мы обновляем каждый кадр, мы используем временной интервал между кадрами в качестве времени обновления.

Метод интегрирования добавлен в класс `Particle`, потому что он просто обновляет внутренние данные частиц. Он принимает только временной интервал и обновляет положение и скорость частицы, не возвращая данные.

Хотя мы рассмотрели поведения объекта при воздействии на него силы, мы не рассмотрели, что происходит, когда действует более одной силы. Понятно, что поведение будет отличаться от того, если действует одна сила: одна сила может действовать в противоположном направлении на другую или параллельно ее усиливать. Нам нужен механизм для выработки общего поведения в результате всех действующих сил.

Принцип Д'Аламбера приходит здесь на помощь. Сам по себе этот принцип более сложный, чем мы будем рассмотривать его здесь. Простыми словами, мы просто складываем силы вместе, используя векторное сложение, и применяем единственную силу, которая получается. Чтобы использовать этот результат, мы используем вектор в качестве аккумулятора силы. В каждом кадре мы обнуляем вектор и добавляем каждую приложенную силу, используя векторное сложение. Конечным значением будет результирующая сила, применяемая к объекту. Мы добавляем к частице метод, который вызывается в конце каждого шага интегрирования, чтобы очистить накопитель от только что примененных сил.

Эта стадия накопления должна быть завершена непосредственно перед интегрированием частицы. Все задействованные силы должны иметь возможность добавить свои значения в аккумулятор. Мы можем это сделать, вручную добавив код в наш цикл обновления фрейма, который добавляет соответствующие силы. Это подходит для сил, которые будут выполняться только для нескольких кадров.

Так как большинство сил будет применяться к объекту в течение длительного периода времени, мы можем упростить управление этими долгосрочными силами, создав реестр. Силы можно зарегистрировать в нем в нем вместе с соответсвующеми частицами, а затем они будут прикладываться к частице в каждом кадре.

3.2 Класс `ParticleForceGenerator` и его производные

У нас есть механизм для приложения нескольких сил к объекту. Теперь нам необходимо выяснить, откуда эти силы. Сила тяжести достаточно проста: она всегда присутствует для всех объектов в игре.

Некоторые силы возникают из-за поведения объекта – например, специальная сила сопротивления. Другие силы являются следствием окружающей среды, в которой находится объект: сила плавучести для плавающего объекта и взрывная сила от взрыва являются примерами. Есть

некоторые силы, которые являются результатом того, как объекты взаимодействуют. Наконец, есть силы, которые появляются, потому что игрок (или персонаж с искусственным интеллектом) запросил их: например, сила ускорения в автомобиле или тяга из реактивного снаряда.

Другая проблема – динамическая природа некоторых сил. Описать силу тяжести легко, потому что она всегда постоянна. Мы можем вычислить ее один раз и оставить установленной на все оставшееся время. Большинство других сил постоянно меняется. Некоторые меняются в результате изменения положения или скорости объекта: при более высоких скоростях сопротивление сильнее, а сила пружины больше, чем больше она сжимается. Другие меняются из-за внешних факторов: взрыв рассеивается, или взлет реактивного снаряда игрока внезапно заканчивается, когда он отпускает кнопку тяги.

Для их расчета нам нужно иметь дело с рядом различных сил с очень разными механиками. Некоторые могут быть постоянными, другие могут применять некоторые изменения к текущим свойствам объекта (например, положение и скорость), некоторые могут зависеть от ввода пользователя, а другие могут быть основаны на времени.

Если мы бы просто запрограммировали все эти типы сил в физическом движке и установили параметры для их взаимодействия для каждого объекта, код быстро стал бы неуправляемым. В идеале мы хотели бы уметь абстрагироваться от деталей о том, как рассчитывается сила, и позволить физическому движку просто работать с силами в целом. Это позволило бы нам применить любое количество сил к объекту, не зная детали того, как вычисляются эти силы.

Это сделано это через класс `ParticleForceGenerator`. Может быть столько разных типов силовых генераторов, сколько есть видов сил, но каждому объекту не нужно знать, как работает генератор. Объект просто использует общий интерфейс, чтобы найти силу, связанную с каждым генератором: эти силы затем могут быть сложеы и применены на этапе интегрирования. Это позволяет нам применять к объекту любое количество сил любого типа, которые мы выбираем. Это также позволяет нам создавать новые типы силы для новых игр или уровней, как нам нужно, без необходимости переписывать какой-либо код в физическом движке.

Не каждый физический движок имеет концепцию силовых генераторов: многие требуют рукописного кода для добавления сил или же ограничивают возможные силы несколькими стандартными опциями. Наличие общего решения является более гибким и позволяет больше экспериментировать.

Для реализации этого мы будем использовать объектно-ориентированный шаблон проектирования, называемый интерфейсом.

Интерфейс силового генератора должен сообщать только текущую силу. Она может быть накоплена и применена к объекту.

Метод `updateForce` вызывается во время кадра, для которого

требуется сила, и принимает указатель на частицу, запрашивающую силу. Длительность кадра необходима для некоторых силовых генераторов.

Мы передаем указатель на объект в функцию, чтобы генератор сил не отслеживал сам объект. Это также позволяет нам создавать генераторы силы, которые могут быть использованы с несколькими объектами одновременно. Пока экземпляр генератора не содержит данных, специфичных для конкретного объекта, он может просто использовать переданный объект для вычисления силы.

Силовой генератор не возвращает никакого значения, и это обеспечивает гибкость, которую мы будем использовать, полностью поддерживая различные виды сил. Вместо этого, если силовой генератор хочет применить силу, он может вызвать метод `addForce` для передаваемого объекта.

Так же необходимо иметь возможность регистрировать, на какие частицы влияют определенные силовые генераторы. Мы могли бы добавить такую структуру данных в каждую частицу с, как связанный список или растущий массив генераторов. Это был бы корректный подход, но он имеет последствия для производительности: либо у каждой частицы должно быть много неиспользуемой памяти (с использованием растущего массива), либо новые регистрации вызовут много операций с памятью (создание элементов в связанных списках). Для производительности и модульности лучше отделить дизайн и иметь центральный регистр частиц и силовых генераторов. Этот функционал представлен классами `ParticleForceRegistry` и `ParticleForceRegistration`.

Мы также можем реализовать генератор силы сопротивления. Сила сопротивления – это сила, действующая на тело и зависящая от его скорости. Полная модель сопротивления связана с более сложной математикой, которую мы легко можем выполнить в реальном времени. Это реализовано в классе `ParticleDrag`.

Сила вычисляется только на основе свойств объекта, который передается. Единственными данными, хранящимися в классе, являются значения для двух констант. Как и ранее, один экземпляр этого класса может быть разделен между любым числом объектов, имеющих одинаковые коэффициенты сопротивления.

Эта модель сопротивления значительно сложнее, чем простое затухание. Его можно использовать для моделирования типа сопротивления, которое, например, испытывает мяч для гольфа в полете. Однако для аэродинамики, необходимой в имитаторе полета, однако, этого может быть недостаточно.

3.3 Классы `ParticleSpring`, `ParticleAnchredSpring`, `ParticleBugee`, `ParticleBouyancy`, `ParticleFakeSpring`

Одна из самых полезных сил, которую можно добавить в движкок, – это

сила пружины. Хотя пружины имеют очевидное применение в гоночных играх (для имитации подвески автомобиля), они также находят применение в представлении мягких или деформируемых объектов. Только пружины и частицы могут создавать целый ряд впечатляющих эффектов, таких как веревки, флаги, одежда из ткани и водяную рябь. Наряду с жесткими ограничениями они могут представлять практически любой объект.

Реальные пружины имеют диапазон длины: это их предел эластичности. Если мы продолжим растягивать металлическую пружину, в конечном итоге мы превысим ее эластичность и она будет деформироваться. Аналогично, если мы слишком сильно сжимаем пружину, ее катушки касаются, и дальнейшее сжатие невозможно.

Мы могли бы закодировать эти ограничения в нашем силовом генераторе для создания реалистичной модели пружины. Однако в подавляющем большинстве случаев нам не нужна эта сложность. Единственным исключением из этого правила является случай пружин, которые могут быть сжаты до определенного предела. Так обстоит дело с подвесками автомобилей: после того как они сжаты до этого момента, они больше не действуют как пружины, а скорее как столкновение между двумя объектами.

Мы внедрили четыре силовых генератора, основанных на пружинных силах. Хотя каждый из них имеет несколько иной способ расчета текущей длины пружины, все они используют закон Гука для вычисления результирующей силы.

Основной механизм обработки остается общим, но он окружен вспомогательными классами и функциями (в данном случае различными типами генераторов пружинных сил), которые часто очень похожи друг на друга.

Основной пружинный генератор просто вычисляет длину пружины, используя определенное уравнение, и затем использует закон Гука для вычисления силы. Он реализован в классе наследнике `ParticleForceGenerator` `ParticleSpring`.

Генератор создается с тремя параметрами. Первый – указатель на объект на другом конце пружины, второй – на постоянную пружины, а третий – длина пружины. Поскольку он содержит данные, зависящие от пружины, один экземпляр нельзя использовать для нескольких объектов. Вместо этого нам нужно создать новый генератор для каждого объекта.

Заметьте также, что генератор силы создает силу только для одного объекта. Если мы хотим связать два объекта с пружиной, нам нужно будет создать и зарегистрировать генератор для каждого из них.

Во многих случаях мы не хотим связывать два объекта вместе с пружиной; скорее мы хотим, чтобы один конец пружины находился в фиксированной точке в пространстве. Это можно применить, например, для опорных кабелей на пружинящем канатном мосте. Один конец пружины

прикреплен к мосту, другой закреплен в пространстве как на рисунке 3.1.

В этом случае форма генератора пружин, которую мы создали ранее, работать не будет. Мы изменили его так, чтобы генератор ожидал фиксированного местоположения, а не объекта, на который он ссылается. Код генератора сил также изменяется, чтобы использовать местоположение непосредственно, а не брать его из объекта. Реализация генератора якорных сил находиться в классе `ParticleAnchorSpringGenerator`.

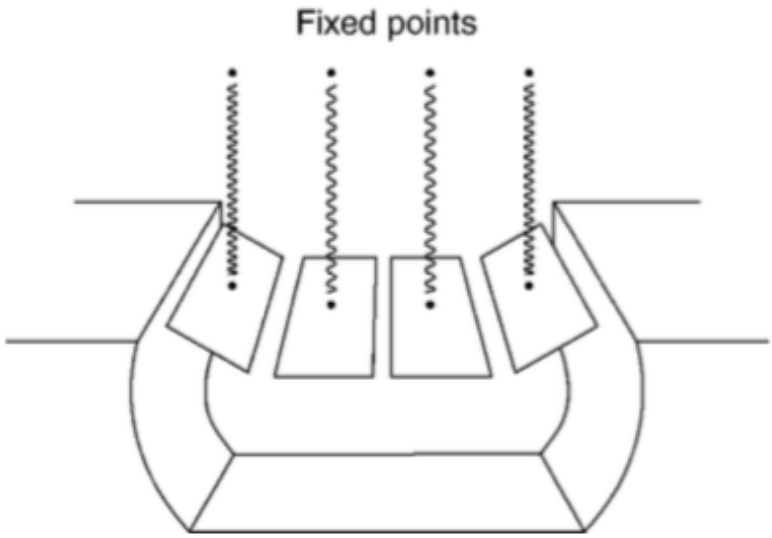

Рисунок 3.1 – Пружинный мост

Эластичный банджи создает только стягивающие силы. Это полезно для хранения пары объектов: они будут стянуты вместе, если разойдутся слишком далеко, но также могут быть очень близо друг от друга. Это задача класса `ParticleBungee`.

Следующий генератор – это генератор плавучести. Плавучесть – это то, что удерживает объект на плаву. Архимед определил, что сила выталкивания зависит от веса воды, которую замещает объект.

Вычисление точной силы плавучести для объекта подразумевает точное определение его формы, поскольку форма влияет на объем замещенной воды, которая используется для вычисления силы, но маловероятно, что нам понадобится такой уровень точности.

Вместо этого мы можем использовать вычисления для пружин в качестве приближения. Когда объект находится рядом с поверхностью, мы используем силу пружины, чтобы придать ему плавучесть. Сила пропорциональна глубине объекта, точно так же, как сила пружины пропорциональна растяжению или сжатию пружины. Как мы видели на рисунке 3.2, это будет точно для прямоугольного блока, который не полностью погружен в воду. Для любого другого объекта это будет немного неточно, но не настолько, чтобы быть заметным.

Когда блок полностью погружен, он ведет себя несколько иначе. Вдавливание его глубже в воду не приведет к вытеснению воды, так что, если мы предположим, что вода имеет ту же плотность, сила будет одинаковой. Точечные массы, с которыми мы имеем дело не имеют размера, поэтому мы не можем сказать, насколько велики они, чтобы определить, полностью ли они погружены в воду. Вместо этого мы можем просто использовать фиксированную глубину: когда мы создаем силу плавучести, мы определяем глубину, на которой объект считается полностью погруженным. В этот момент сила выталкивания не будет увеличиваться при более глубоком погружении.

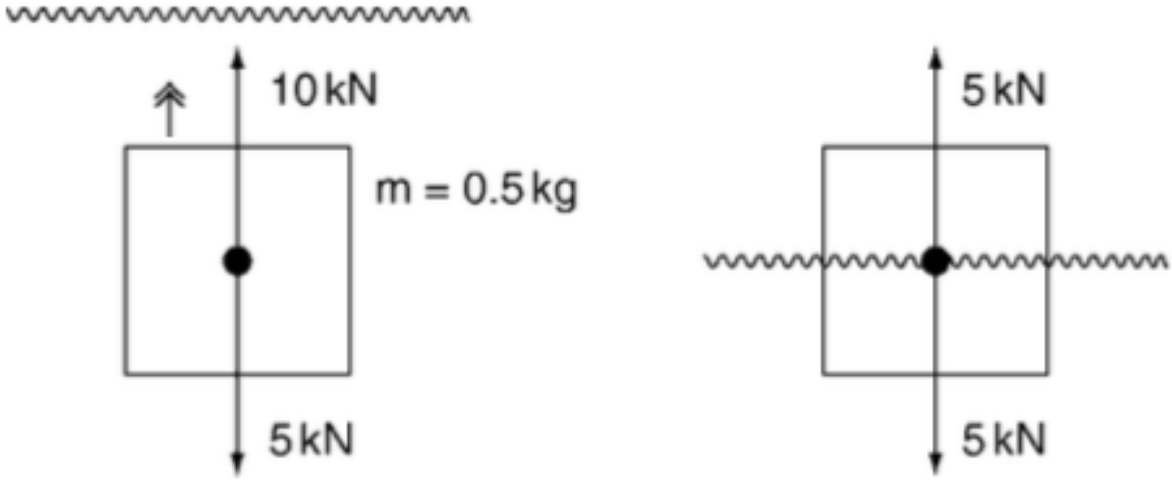

Рисунок 3.2 – Плавучий блок и сила выталкивания

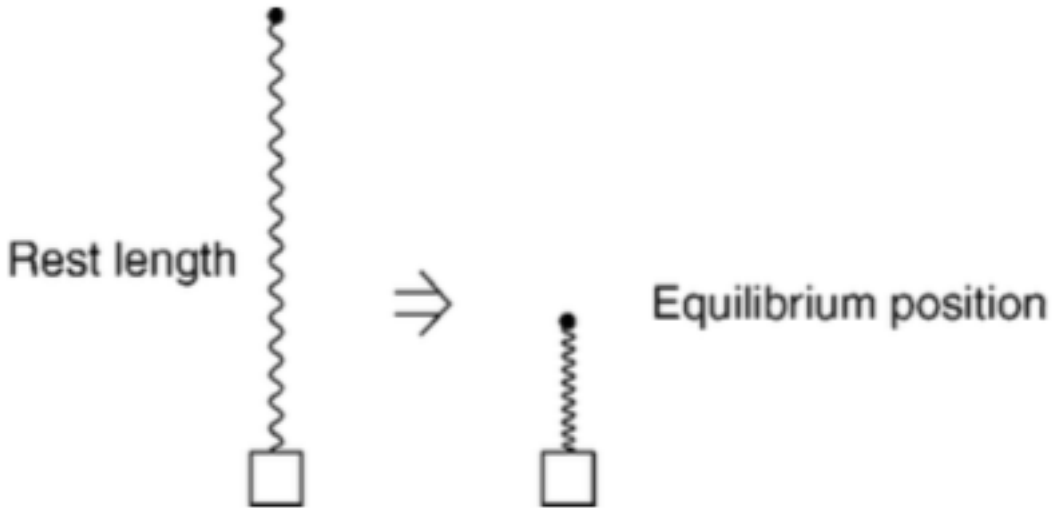

Рисунок 3.3 – Длинна покоя пружины и позиция равновесия

Этот генератор применяется только к одному объекту, потому что он содержит данные для размера и фигуры объекта. Один экземпляр можно было бы дать нескольким объектам с одинаковым размером и объемом, плавающим в одной жидкости, но лучше всего создать новый экземпляр для каждого объекта, чтобы избежать путаницы. Это генератор реализован в классе `ParticleBuoyancy`.

В реальной жизни почти все действует как пружины. С моделью поведения пружин можно промоделировать все. Столкновения между объектами можно смоделировать так же, как и с силой выталкивания: объекты столкнутся, и пружинная сила снова разведет их обратно. При правильных параметрах пружины для каждого объекта этот метод дал бы нам идеальные столкновения.

Далее будет описан генератор жесткой пружины, который работает несколько иначе, чем обычный генератор пружины. Он выглядит как якорный пружинный генератор, который мы рассмотрели ранее, с одним существенным отличием: он больше не имеет естественной длины пружины. Следствием этого является то, что пружина всегда должна иметь нулевую длину. Если пружина имеет нулевую длину покоя, то любое смещение одного конца пружины приводит к растяжению пружины. Если мы зафиксируем один конец пружины, то всегда будет сила в направлении закрепленного конца.

Для пружины, у которой два конца могут свободно двигаться, определить направление силы гораздо труднее. В этом случае необходимо включить движение другого конца пружины в уравнение, что сделало бы его очень сложным.

Аналогичная проблема возникает, если мы привязываем один конец, но используем ненулевую длину покоя. В одном измерении ненулевая длина покоя эквивалентна перемещению точки равновесия, как показано на рисунке 3.3. То же самое справедливо в трех измерениях, но поскольку пружине позволено свободно вращаться, эта точка равновесия теперь движется и возникают те же проблемы, что и для незакрепленной пружины.

Таким образом, один конец этой пружины нужно держать в заранее определенном фиксированном месте. Как и с предыдущими закрепленными пружинами, мы можем изменять это местоположение вручную от кадра к кадру. Класс `ParticleFakeSpring` реализует это поведение.

3.4 Класс `ParticleContact` и обработка столкновений

Далее мы введем понятие жестких ограничениях. Сначала мы рассмотрим наиболее общие жесткие ограничения – столкновения и контакты между объектами. Та же математика может быть использована для других видов жестких ограничений, таких как стержни или нерастяжимые веревки, которые могут использоваться для соединения объектов друг с другом.

Чтобы обрабатывать жесткие ограничения, мы добавим систему

разрешения конфликтов. Далее термин столкновение относится к любой ситуации, в которой касаются два объекта. Тот же самый процесс, который мы используем для разрешения быстрых столкновений, будет использоваться для разрешения контакта покоя.

Когда два объекта сталкиваются, их движение после столкновения может быть рассчитано из их движения до столкновения: это и есть разрешение столкновения. Мы разрешаем столкновение, проверяя, чтобы два объекта имели правильные скорости, которые были бы результатом столкновения.

Для обработки столкновений у нас есть класс `ContactResolver`. Он принимает множество столкновений и применяет соответствующие импульсы к задействованным объектам. Каждое столкновение представляется в классе `ParticleContact`. Он содержит указатель на каждый объект, участвующий в столкновении; вектор, представляющий контактную нормаль, с точки зрения первого объекта; и элемент данных для коэффициента востановления для контакта. Если мы имеем дело с коллизией между объектом и статичной средой (то есть, есть только один задействованный объект), то указатель на второй объект будет равен `NULL`.

Чтобы разрешить один контакт, мы применяем уравнения столкновения описанные в методах `resolve`, `resolveVelocity`, `calculateSeparatingVelocity`.

Точки столкновения обычно обнаруживаются с помощью детектора столкновения. Детектор столкновения – это кусок кода, отвечающий за нахождение пар объектов, которые сталкиваются, или отдельных объектов, которые сталкиваются с некоторой частью неподвижной среды.

В нашем движке конечным результатом алгоритма обнаружения столкновений является набор данных `ParticleContact`, заполненных соответствующей информацией. При обнаружении столкновений необходимо учитывать геометрию объектов: их форму и размер. До сих пор мы предполагали, что имеем дело с частицами, что позволяет нам вообще не учитывать геометрию.

Это различие мы сохраним в неизменном виде даже с более сложными трехмерными объектами: симулятор физики (та часть движка, которая управляет законами движения, разрешением столкновения и силами) не будет нуждаться в подробностях формы фигуры. Система обнаружения столкновений отвечает за вычисление любых геометрических свойств, таких как, когда и где касаются два объекта, и нормаль контакта между ними.

Существует целый ряд алгоритмов, используемых для нахождения точек контакта. Некоторые алгоритмы обнаружения столкновений могут принимать во внимание способ перемещения объектов и пытаться прогнозировать вероятные столкновения в будущем. Проще всего просмотреть набор объектов и проверить, взаимопроникают ли какие-либо два объекта.

Два объекта взаимопроникают, если они частично пересекаются друг с другом, как показано на рисунке 3.4. Когда мы обрабатываем столкновение между частично внедренными объектами, недостаточно изменить их скорость. Если объекты сталкиваются с небольшим коэффициентом восстановления, их скорость разделения может быть почти равна нулю. В этом случае они никогда не раздвинутся, и игрок увидит объекты, склеенные вместе невозможным способом.

Рисунок 3.4 – Взаимопроникновение объектов

В рамках разрешения столкновений нам необходимо разрешить взаимопроникновение.

Когда два объекта взаимопроникают, мы раздвигаем их так, чтобы их разделить. Мы ожидаем, что детектор столкновения скажет нам, насколько далеко объекты взаимопроникают, через часть структуры данных `ParticleContact`, которую он создает. Расчет глубины взаимопроникновения зависит от геометрии сталкивающихся объектов, и, как мы видели ранее, это область системы обнаружения столкновений, а не физического симулятора.

Чтобы разрешить взаимопроникновение, мы проверяем его глубину. Если она уже равна нулю или меньше, нам не нужно предпринимать никаких действий. В противном случае мы можем сдвинуть эти два объекта достаточно далеко, чтобы глубина проникновения стала равной нулю. Глубина проникновения должна указываться в направлении нормали контакта. Если мы перемещаем объекты в направлении нормали контакта, на расстояние, равное глубине проникновения, объекты больше не будут находиться в контакте. То же самое происходит, когда у нас есть только один объект,

задействованный в контакте: глубина проникновения находится в направлении нормали контакта.

Итак, мы знаем общее расстояние, на которое должны перемещать объекты (то есть глубина), и направление, в котором они будут перемещаться; Нам нужно определить, сколько каждый отдельный объект должен быть перемещен.

Если в контакте есть только один объект, то это просто: объект должен переместиться на все расстояние. Если у нас есть два объекта, у нас есть целый ряд вариантов. Мы могли бы просто перемещать каждый объект на одну и ту же величину: на половину глубины проникновения. Это может работать в некоторых ситуациях, но может вызвать проблемы с реалистичностью как показано на рисунке 3.5.

Для этого мы перемещаем два объекта обратно пропорционально их массе. Объект с большой массой почти не двигается, а объект с крошечной массой проходит почти все расстояние. Если один из объектов имеет бесконечную массу, то он не будет двигаться: другой объект перемещается на все расстояние.

Это реализовано в методах `resolve` и `resolveInterpenetration` класса `ParticleContact`.

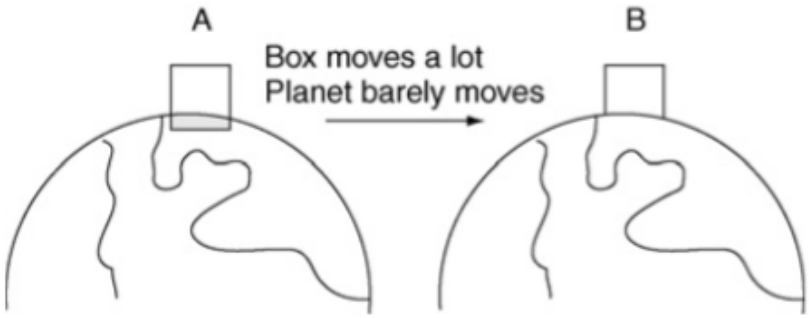

Рисунок 3.5 – Проблема разрешения взаимопроникновений

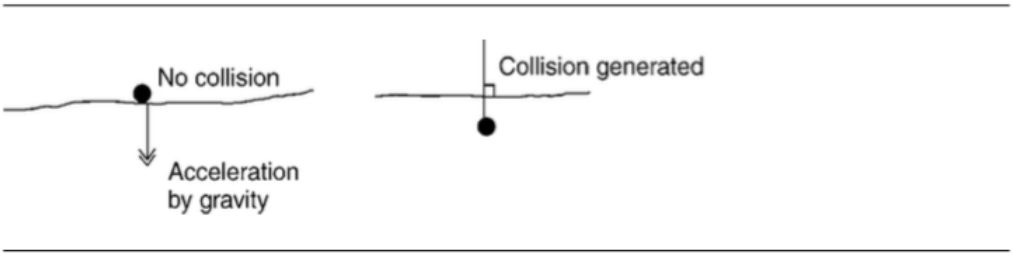

Рисунок 3.6 – Вибрация при столкновении покоя

Рассмотрим ситуацию, показанную на рисунке 3.6. У нас есть частица, покоящаяся на земле. На нее действует только одна сила – гравитация. В

первом кадре частица устемляется вниз. Ее скорость увеличивается, но положение остается постоянным. Во втором кадре положение обновляется, и скорость снова увеличивается. Теперь она движется вниз и начинает проникать в землю. Детектор столкновения улавливает взаимопроникновение и создает столкновение.

Поэтому в третьем кадре ее скорость будет направлена вверх, и будет переносить ее с земли и в воздух. Скорость восходящего движения будет небольшой, но ее может быть достаточно, чтобы движение было заметно. В частности, если кадр 1 или 2 аномально длинный, скорость будет значительно большой и частица полетит вверх.

Чтобы решить эту проблему, мы можем сделать две вещи. Сначала мы должны обнаружить контакт раньше чем он произойдет. В примере два кадра прошли, прежде чем мы узнали, что есть проблема. Если мы сделаем так, что наш детектор столкновений будет возвращать контакты, которые почти, но не полностью взаимопроникающие, тогда мы получаем контакт для обработки после кадра 1.

Во-вторых, нам нужно распознать, когда объект имеет скорость, которая могла возникнуть только из его сил, действующих во время одного кадра. После кадра 1 скорость частицы обусловлена только силой тяжести, действующей на нее в течении одного кадра. Мы можем выяснить, какова была бы скорость, если бы на нее воздействовала только сила, просто умножая силу на продолжительность кадра. Если фактическая скорость объекта меньше или равна этому значению (или даже немного выше его, если мы признаем, что ошибки округления могут ползти), мы знаем, что частица была неподвижна в предыдущем кадре. В этом случае контакт, скорее всего, будет касательным, а не встречным. Вместо того, чтобы выполнять расчет импульса для столкновения, мы можем применить импульс, который приведет к нулевой скорости разделения.

Это то, что произошло бы с контактом покоя: никакая скорость сближения не успела бы появиться, поэтому после контакта не было бы разделительной скорости. В нашем случае мы видим, что скорость будет лишь побочным результатом того, как мы разбиваем время на кадры, и поэтому мы можем обращаться с объектом так, как если бы он имел нулевую скорость перед контактом. Частице задается нулевая скорость. Это происходит в каждом кадре: на самом деле частица всегда остается в кадре 1 на рисунке 3.6.

Когда у нас есть два объекта в контакте покоя, нас интересует их относительная скорость, а не абсолютная скорость любой из них. Эти два объекта могут находиться в контакте друг с другом в одном направлении, но перемещаться друг относительно друга в другом направлении. Например, ящик может опираться на землю, даже если он одновременно скользит по ее поверхности. Мы хотим, чтобы код для вибрирующих контактов справлялся с парами объектов, которые скользят друг относительно друга. Это означает, что мы не можем использовать абсолютную скорость любого объекта.

Чтобы справиться с этой ситуацией, вычисления скорости и ускорения

выполняются только в направлении контактной нормали. Сначала мы определим скорость в этом направлении и проверим, не могла ли она быть вызвана только компонентой ускорения в том же направлении. Если это так, то скорость изменяется так, что в этом направлении нет разделяющей или скорости сближения. В любом другом направлении все еще может быть относительная скорость, но она игнорируется.

Чтобы поддерживать два объекта в состоянии покоя, мы понемногу изменяем скорости в каждом кадре. Изменения достаточно велики, чтобы компенсировать увеличение скорости, которое возникнет в результате их сближения друг с другом в течение одного кадра. Данное поведение описано в методе `resolveVelocity` в классе `ParticleContact`.

Подход с микростолкновениями, описанный выше, является лишь одной из многих возможных. Контакт покоя является одной из сложнейших задач в физическом движке. Существует много способов решений, а также множество вариаций и настроек.

Более физически реалистичный подход состоял бы в том, чтобы предположить, что сила будет применяться к частице со стороны земли. Эта сила реакции отталкивает объект назад, так что его полное ускорение в вертикальном направлении становится равным нулю. Независимо от того, насколько сильно частица стремиться вниз, земля будет толкать ее вверх с той же силой. Мы можем создать генератор силы, который работает таким образом, чтобы не было ускорения на земле.

Это хорошо работает для частиц, которые могут иметь только один контакт с землей. Для более сложных твердых тел ситуация становится значительно более сложной. У нас может быть несколько точек соприкосновения объекта с землей (или, что еще хуже, у нас может быть целая серия контактов между объектом и неподвижными точками). Не очевидно, как вычислить силы реакции при каждом контакте, чтобы общее движение объекта было правильным.

Получатель коллизии получает список контактов из системы обнаружения столкновений и нуждается в обновлении моделируемых объектов, чтобы учесть контакты. У нас есть три участка кода для выполнения этого обновления:

– функция разрешения столкновений, которая применяет импульсы к объектам для имитации их отскока друг от друга;

– функция разрешения взаимного проникновения, которая перемещает объекты друг от друга, чтобы они не находились частично друг в друге;

– код контактов покоя, который находится внутри функции разрешения конфликтов, и следит за контактами, которые могут скорее соприкасаться, чем сталкиваться.

Какая из этих функций требует вызова, зависит от скорости столкновения и глубины взаимопроникновения. Разрешение взаимопроникновения должно возникать только в том случае, если контакт

имеет глубину проникновения больше нуля. Аналогично, нам может потребоваться выполнить разрешение взаимопроникновения, без разрешения коллизии, если объекты взаимопроникают, но разделяются.

Независимо от комбинации необходимых функций каждый контакт разрешается по одному за раз. Это упрощение в отличии от того, как это происходит в реальном мире. На самом деле каждый контакт будет происходить в несколько иной момент времени или распределен в течение определенного промежутка времени. Некоторые контакты применяются последовательно; другие будут совмещаться и действовать одновременно на объекты, которые они затрагивают.

Если объект имеет два одновременных контакта, как показано на рисунке 3.7, то изменение его скорости для разрешения одного контакта может изменить его скорость разделения на другом контакте. На рисунке, если мы разрешим первый контакт, то второй контакт вообще перестает быть столкновением: он теперь разрешен. Однако, если мы разрешим только второй контакт, первый контакт все еще нуждается в разрешении: изменение скорости было недостаточным.

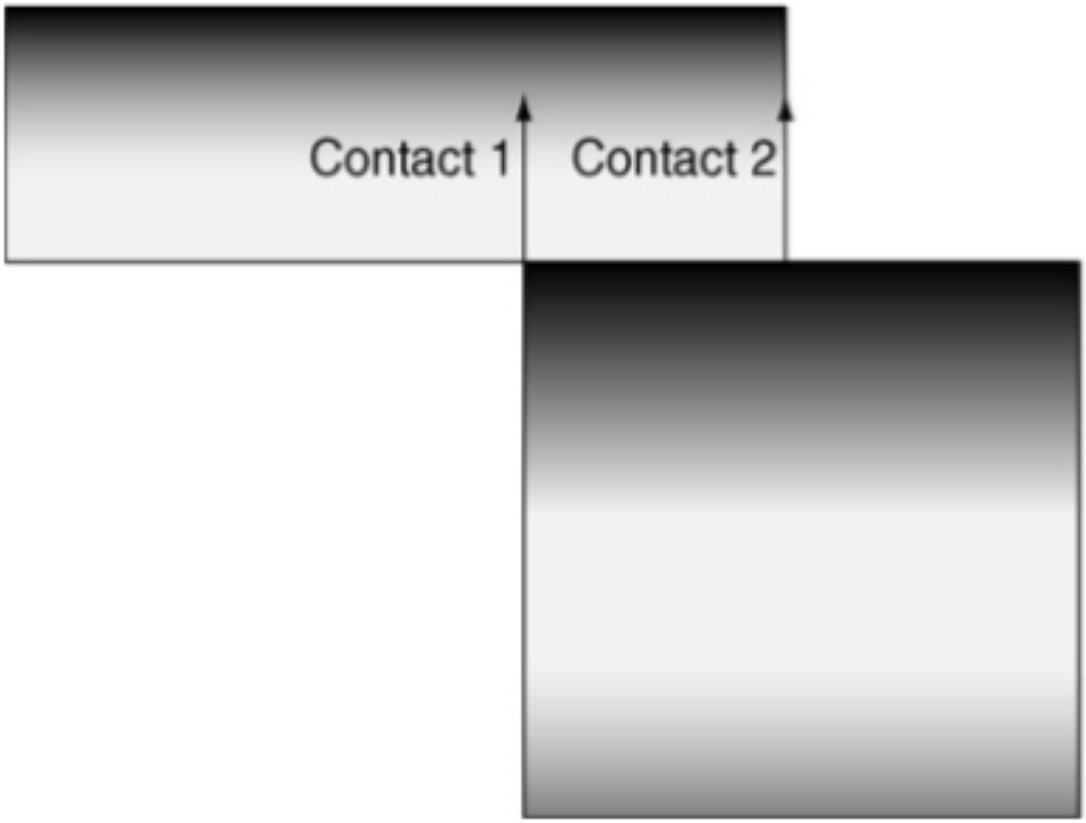

Рисунок 3.7 – Два контакта при столкновении

Чтобы избежать ненужной работы в подобных ситуациях, сначала мы решаем самый сложный контакт: контакт с наименьшей разделительной

скоростью (т.е. самой отрицательной). Другими словами, наиболее сильные столкновения обычно доминируют в поведении моделирования. Если нам нужно определить приоритет столкновений, которые должны быть обработаны, это должны быть те, которые дают наибольший реализм.

Рисунок 3.7 иллюстрирует сложность алгоритма разрешения контактов. Если мы справимся с одним столкновением, то мы можем изменить скорость разделения для других контактов. Мы не можем просто сортировать контакты по их разделительной скорости, а затем обрабатывать их по порядку. Как только мы обработали первое столкновение, следующий контакт может иметь положительную скорость разделения и не нуждаться в какой-либо обработке. Существует также другая, более тонкая проблема, которая не возникает во многих ситуациях с частицами. У нас может возникнуть ситуация, при которой мы разрешаем один контакт, затем следующий, но разрешение второго возвращает первый контакт в столкновение, поэтому нам нужно его снова разрешить.

Обработчик контактов, который мы будем использовать, следует этому алгоритму:

1) Вычислить скорость разделения каждого контакта, отслеживая контакт с самым низким (то есть самым отрицательным) значением.

2) Если нижняя разделительная скорость больше или равна нулю, мы закончили: выход из алгоритма.

3) Обработать ответ на столкновение для контакта с наименьшей разделительной скоростью.

4) Выполнить предыдущие пункты заданное количество итераций.

Алгоритм автоматически пересмотрит контакты, которые он ранее разрешил, и будет игнорировать контакты, которые разделяются. Он разрешает наиболее сильное столкновение на каждой итерации.

Число итераций должно быть не меньше количества контактов (чтобы обработать их хотя бы один раз) и может быть больше. Для простого моделирования частиц, с таким же количеством итераций, как и с контактами, это должно отлично работать. Мы используем двойное число контактов, так как обычно больше итераций требуется для сложных взаимосвязанных наборов контактов.

Разрешение взаимного проникновения происходит по тому же алгоритму, что и разрешения столкновения. Как и прежде, нам нужно пересчитать все глубины взаимопроникновения между каждой итерацией. Напомним, что глубина взаимопроникновения обеспечивается детектором столкновения. Мы не хотим снова выполнять обнаружение столкновений после каждой итерации, так как это требует слишком много времени. Чтобы обновить глубину взаимопроникновения, мы отслеживаем, насколько мы перемещали два объекта на предыдущей итерации. Затем проверяются объекты в каждом контакте. Если какой-либо объект был перемещен в последнем кадре, то его глубина взаимопроникновения обновляется путем нахождения компоненты перемещения в направлении нормали контакта. Это

происходит в методе `resolveContacts` класса `ParticleContactResolver`.

Перерасчет скорости разделения и глубины взаимопроникновения при каждой итерации является наиболее трудоёмкой частью этого алгоритма. Для очень большого числа контактов это может сильно влиять на скорость работы физического движка. На практике большая часть обновлений не будет иметь эффекта: один контакт может не оказать никакого влияния на другой контакт.

3.5 Класс `ParticleLink` и его производные

Далее мы рассмотрим несколько типов соединений, которые можно смоделировать с использованием методов, описанных ранее.
Можно думать о столкновении как о действии, чтобы удерживать два объекта на некотором минимальном расстоянии друг от друга. Контакт создается между двумя объектами, если они когда-либо оказываются слишком близко. Иммено так мы можем использовать контакты для ограничения движения объектов.

Кабель – это такое ограничение, которое вынуждает два объекта быть на расстоянии не большем, чем его длина. Если у нас есть два объекта, соединенных кабелем, они не будут испытывать никаких эффектов, пока они близко друг к другу. Когда же кабель будет натянутым, объекты не смогут дальше двигаться. В зависимости от характеристик кабеля объекты могут отскакивать от этого предела так же, как при столкновении объектов. Кабель имеет коэффициент восстановления, который контролирует этот эффект отскока.

Мы можем моделировать кабели, генерируя контакты, когда концы кабеля слишком далеко отходят друг от друга. Контакт очень похож на тот, который используется для столкновений, за исключением того, что его контактная ситуация меняется на противоположную: она стягивает объекты, а не отбрасывает их друг от друга. Глубина взаимопроникновения контакта соответствует тому, насколько далеко кабель растянут за его пределами. Все это описано в классе `ParticleCable`, который в свою очередь унаследован от класса `ParticleLink`.

`ParticleCable` действует как детектор столкновения: он проверяет текущее состояние кабеля и может вернуть контакт, если кабель достиг своего предела. Затем этот контакт должен быть добавлен ко всем остальным, сгенерированным детектором столкновения, и обработан в обычном алгоритме распознавания контактов.

Стержни комбинируют поведение кабелей и столкновений. Два объекта, соединенные стержнем, не могут ни разделиться, ни сблизиться. Они находятся на фиксированном расстоянии друг от друга.

Мы можем реализовать это так же, как и генератор контактов для кабелей. В каждом кадре мы смотрим на текущее состояние стержня и

генерируем контакт, чтобы сблизить концы или удержать их на расстоянии друг от друга.

Однако мы должны добавить две модификации. Во-первых мы всегда должны использовать нулевой коэффициент восстановления. Они должны находиться на постоянном расстоянии друг от друга, поэтому их относительная скорость вдоль линии между ними должна быть равна нулю.

Во-вторых, если мы будем генерировать только один из двух контактов в каждом кадре, мы получим вибрирующий стержень. В соседних кадрах стержень, вероятно, будет слишком коротким, а затем слишком длинным, и каждый контакт будет тянуть его назад и вперед. Чтобы избежать этого, мы генерируем оба контакта в каждом кадре. Если ни один из контактов не требуется (то есть скорость разделения больше нуля или отсутствует взаимопроникновение), то ничего не произойдет. Наличие дополнительного контакта помогает сохранить алгоритм контактного распознавателя от сверхкомпенсации, поэтому стержень будет более стабильным. Недостатком такого подхода является то, что для сложных сборок стержней количество итераций, необходимых для достижения действительно устойчивого решения, может резко возрасти.

3.6 Класс `ParticleWorld`

Физический движок состоит из трех компонентов:

1) Частицы. Они сами отслеживают свое положение, движение и массу. Чтобы начать симуляцию, нам необходимо определить, какие частицы нужны и установить их начальную скорость. Нам также нужно установить их обратную массу.

2) Генераторы силы. Они используются для отслеживания сил, которые применяются течении игры.

3) Система столкновений. Она накапливает набор контактных объектов и передает их для разрешения.

В каждом кадре мы берем каждую частицу, вычисляем ее внутренние данные, вызываем ее генераторы силы, а затем вызываем ее интегратор для обновления его положения и скорости. Затем мы находим контакты с частицами и передаем все контакты для всех частиц в преобразователь столкновений.

Чтобы упростить этот процесс, мы построим простую структуру, содержащую любое количество твердых тел. Мы храним твердые тела в связанном списке, точно так же, как мы это сделано для силовых генераторов. Этот связанный список содержится в классе `ParticleWorld`, представляющем весь физический мир.

В каждом кадре сначала вызывается метод `startFrame`, который готовит каждый объект к применению сил. После вызова этого метода могут быть применены дополнительные силы.

Мы также создадим систему регистрации контактов. Точно так же, как и с генераторами силы, мы создаем полиморфный интерфейс для контактных детекторов. Каждый из них в свою очередь вызывается из `ParticleWorld` и может вносить любые контакты, которые он находит обратно в `ParticleWorld`, вызывая его метод `addContact`.

Для выполнения физики вызывается метод `runPhysics`. Он вызывает все силовые генераторы для применения своих сил, а затем выполняет интегрирование всех объектов, запускает контактные распознователи и разрешает список контактов.

Необходимо добавать вызов `startFrame` в начале каждого кадра игры и вызов `runPhysics` везде, где нужна обработка физики. Типичный игровой цикл может выглядеть так:

```
void loop {
     while(1) {
          world.startFrame();
          runGraphicsUpdate();
          updateCharacters();
          world.runPhysics();
     }
}
```

3.7 Класс `RigidBody`

Этот класс представляет твердое тело, во многом он похож на класс `Particle`, но в отличии от него, здесь учитываются не только линейные эффекты при приложении силы, но и угловые.

Этот класс содержит матрицу, чтобы хранить текущую матрицу преобразования объекта. Эта матрица полезна для рендеринга объекта и будет полезна в разных расчетах физики, настолько, что стоит хранить его внутри класса.

Эта матрица должна быть получена из ориентации и положения тела в каждом кадре, чтобы убедиться, что она правильная. Мы не будем обновлять матрицу в физике или использовать ее каким-либо образом, где ее ориентация и положение могут поменяться. Мы не пытаемся хранить одну и ту же информацию в двух местах: эта матрица преобразования просто действует как кеш, чтобы избежать повторного пересчета этой важной величины.

Функция `calculateDerivedData` будет служить для вычисления матрицы преобразования и всех производных данных.

Из второго закона движения Ньютона мы видели, что изменение скорости зависит от силы, действующей на объект, и массы объекта. Для вращения мы имеем очень похожий закон. Изменение угловой скорости зависит от двух факторов: у нас есть крутящий момент вместо силы, и момент инерции вместо массы.

Угловое ускорение зависит от размера силы, которую мы оказываем, и от того, как далеко от оси вращения мы ее применяем. Возьмем пример гаечного ключа и гайки: мы можем открутить гайку, если приложим больше усилий к гаечному ключу или если надавим дальше по ручке (или используем длинный ключ). При переходе силы в крутящий момент важно значение силы, равно как и расстояние от оси вращения.

Каждое усилие, применяемое к объекту, генерирует соответствующий крутящий момент. Всякий раз, когда мы применяем силу к твердому телу, мы должны использовать его так, как делали это в классе `Particle`: произвести линейное ускорение. Нам оно также потребуется использовать его для создания крутящего момента.

В трех измерениях важно заметить, что крутящий момент должен иметь ось. Мы можем применить силу поворота вокруг любой выбранной нами оси. Для свободно вращающегося объекта крутящий момент может привести к повороту объекта относительно любой оси.

Итак, у нас есть крутящий момент – вращательный эквивалент силы. Теперь мы приходим к моменту инерции: угловой эквивалент массы.

Момент инерции объекта является мерой того, насколько сложно изменить скорость вращения этого объекта. В отличие от массы, однако, это зависит от того, как мы вращаем объект.

Момент инерции зависит от массы объекта и расстояния этой массы от оси вращения.

Ясно, что мы не можем использовать одно значение для момента инерции, как это было сделано для массы. Это полностью зависит от выбранной нами оси. Мы можем компактно представлять все различные значения в матрице, называемой тензором инерции.

Трудно представить, что тензор инерции означает либо в геометрических, либо в математических терминах. Он представляет собой тенденцию поворота объекта в направлении, отличающемся от направления, в котором применяется крутящий момент.

Для свободно вращающегося объекта, если мы применим крутящий момент, то не всегда получим вращение вокруг той же оси, к которой вы применили крутящий момент. Это принцип, на котором основаны гироскопы: они сопротивляются падению, потому что переводят любое вызванное гравитацией падение в противоположное воздействие, чтобы снова встать прямо. Тензор инерции контролирует этот процесс: перенос вращения с одной оси на другую.

По тем же причинам, что и для массы, мы будем хранить тензор обратных инерций, а не необработанный тензор инерции. В класс `RigibBody` добавлен элемент для хранения тензора инерции `Matrix3 inverseInertiaTensor`.

Имея обратный тензор, мы можем вычислить угловое ускорение непосредственно, не выполняя при этом обратную операцию.

Перед тем, как мы можем оставить инерционный тензор, остается еще один тонкий момент. Рассмотрим примеры на рисунке 3.8. В первом примере локальные оси объекта находятся в одном направлении с мировыми осями. Если мы применим крутящий момент вокруг оси X, то получим тот же момент инерции, будем ли мы работать в локальных или мировых координатах.

Во второй части рисунка объект поворачивается. Теперь чью ось X нам нужно использовать? Фактически крутящий момент выражается в мировых координатах, поэтому вращение будет зависеть от момента инерции объекта вокруг оси X в мировых координатах. Однако тензор инерции определяется в локальных координатах объекта.

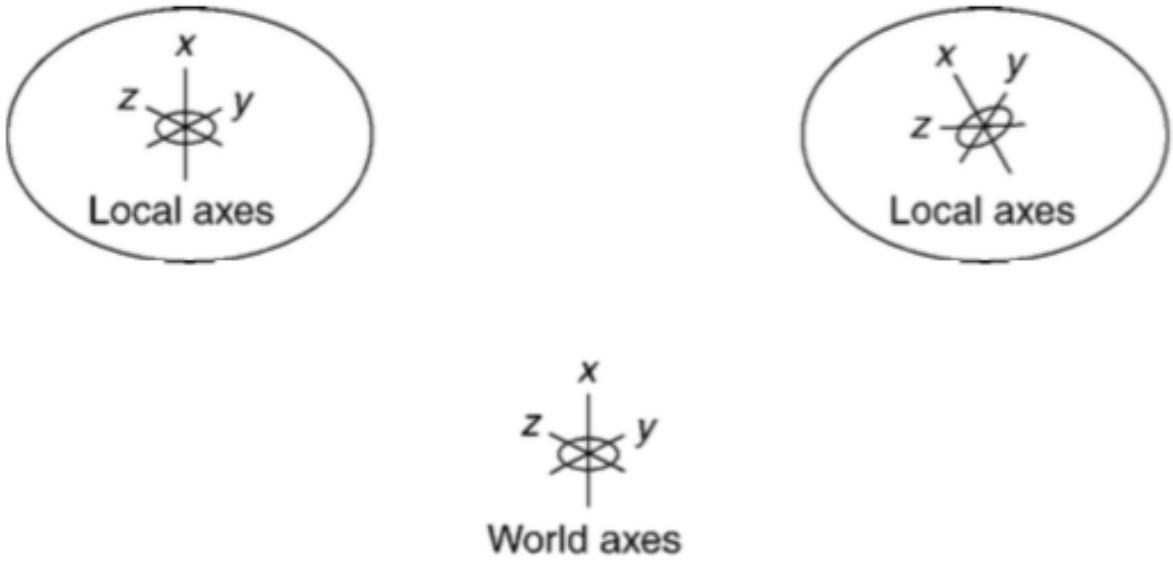

Рисунок 3.8 – Локальность момента инерции

Мы не хотим пересчитывать тензор инерции, суммируя массы в каждом кадре, поэтому нам нужен более простой способ получить тензор инерции в мировых координатах. Мы можем добиться этого, создав новую производную величину: тензор обратной инерции в мировых координатах. В каждом кадре мы можем применить изменение базовой трансформации для преобразования константного инерционного тензора в координатах объекта в соответствующую матрицу в мировых координатах.

Как и с матрицей трансформации, мы пересчитываем производные величины в каждом кадре.

Когда мы преобразуем тензор инерции, нас интересует только вращательная составляющая преобразования объекта. Не имеет значения, где объект находится в пространстве, но только в каком направлении он направлен. Поэтому мы рассматриваем матрицу преобразования 4 × 3, как если бы это была матрица 3 × 3 (то есть только матрица вращения).

Итак в каждом кадре мы вычисляем матрицу трансформации,

преобразуем тензор обратной инерции в мировые координаты и затем выполняем интегрирование твердого тела с этим преобразованным вариантом.

Так же, как у нас есть эквивалент второго закона движения Ньютона, мы также можем найти угловую версию принципа Д'Аламбера. Напомним, что принцип Д’Аламбера позволяет нам складывать целый ряд сил в одну силу, а затем применять только эту силу. Эффект одной накопленной силы идентичен эффекту всех его составляющих сил. Мы воспользуемся этим, просто скомбинировав все силы, применяемые к объекту, и затем только вычисляем его ускорение один раз, основываясь на итоговой сумме.

Тот же принцип применяется к моментам: эффект целой серии крутящих моментов равен влиянию одного крутящего момента, который объединяет их все.

Другим следствием принципа Д'Аламбера является то, что мы можем накапливать крутящие моменты, вызванные силами, точно так же, как мы накапливаем любые другие моменты. Заметим, что мы не можем просто накапливать силы, а затем брать крутящий момент, эквивалентный полученной силе. Мы могли бы иметь две силы, которые нейтрализуют друг друга как линейные силы, но в совокупности создают большой крутящий момент.

Итак, у нас есть аккумуляторы `forceAccum` и `torqueAccum` – один для сил и другой для крутящего момента. Каждая прилагаемая сила добавляется как к аккумулятору силы, так и к крутящему моменту.

Также есть важная особенность относительно места применения силы. Оно должно быть выражено в мировых координатах. Мы можем сделать это, просто изменив положение координат объекта на матрицу преобразования, чтобы получить позицию в мировых координатах. Направление силы должно быть выражено в мировых координатах, тогда как точка приложения ожидается в локальных координатах объекта.

3.8 Обнаружение столкновений

Обнаружение столкновений может быть очень трудоемким процессом. Каждый объект, в игре, может сталкиваться с любым другим объектом, и каждая такая пара должна быть проверена. Если в игре есть сотни объектов, может потребоваться сотни тысяч проверок. И каждая проверка должна понять геометрию двух вовлеченных объектов, которые могут состоять из тысяч многоугольников. Поэтому, чтобы выполнить полное обнаружение столкновений, нам может потребоваться огромное количество длительных проверок. Это невозможно за момент времени между кадрами.

Ограничивающий том – это область пространства, которая содержит какой-либо объект. Чтобы представить фигуру для определения столкновений, используется простая форма, как правило, сфера или коробка. Фигура должна быть достаточно большой, чтобы гарантировать, что весь

объект находится внутри ее.

Затем фигуру можно использовать для выполнения некоторых простых тестов на столкновение. Если два объекта имеют ограниченные тома, которые не касаются, то нет способа, которым объекты внутри них могут находиться в контакте. На рисунке 3.9 показан объект со сферическим ограничивающим томом.

Идеально ограничивающие тома должны быть максимально приближены к их объекту. Если два близких граничных тома касаются, то их объекты, вероятно, коснутся. Если большая часть пространства внутри ограничивающих томов не занята, то их касание вряд ли означает, что объекты находятся в контакте.

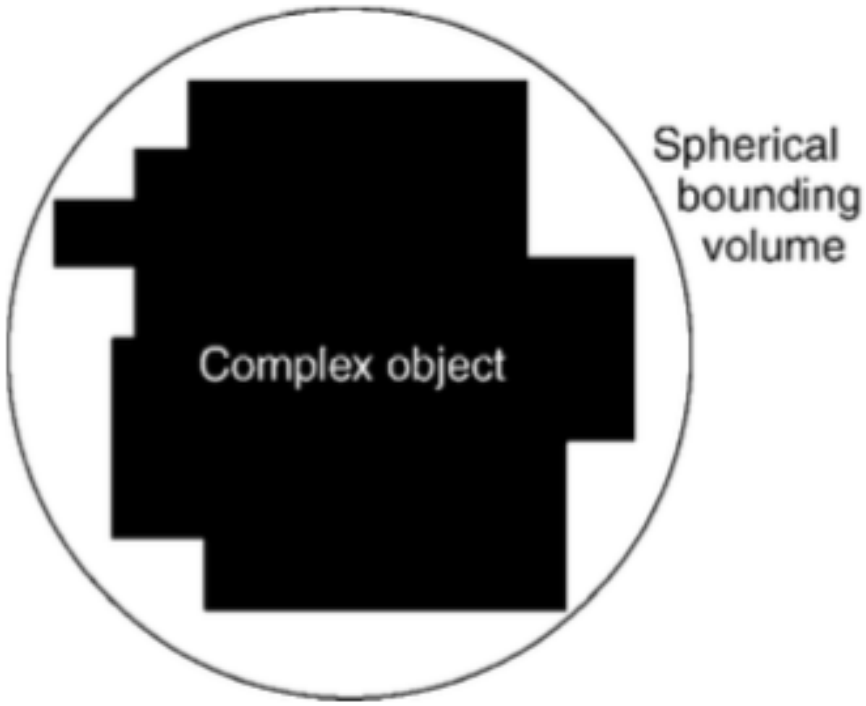

Рисунок 3.9 – Ограничивающий том

Сферы удобны, потому что их легко представить. Хранения центра сферы и ее радиуса достаточно.

```
struct BoundingSphere
{
    Vector3 center;
    real radius;
};
```

Также очень легко проверить, пересекаются ли две сферы. Они пересекаются, если расстояние между их центрами меньше суммы их радиусов. Сферы – хорошая форма для ограничивающих томов для

большинства объектов.

Кубические коробки также часто используются. Они могут быть представлены как центральная точка и набор размеров, по одному для каждой оси. Эти размеры часто называют полуразмерами или полуширинами, поскольку они представляют расстояние от центральной точки до края коробки, что составляет половину общего размера коробки в соответствующем направлении.

```
struct BoundingBox
{
    Vector3 center;
    Vector3 halfSize;
};
```

С каждым объектом, заключенным в ограничивающий том, мы можем выполнить простой тест, чтобы увидеть, могут ли объекты быть в контакте. Если ограничивающие томы касаются друг друга, то проверка может быть заменена с грубого детектора столкновения на более детальное изучение детектором мелкого столкновения. Это ускоряет обнаружение столкновений, но все равно включает проверку каждой пары объектов. Мы можем избежать выполнения большинства этих проверок путем организации ограничивающих томов в иерархиях.

В иерархии ограничивающих томов каждый объект в своем ограничивающем томе хранится в листьях структуры дерева. Ограничивающие томы нижнего уровня связаны с родительскими узлами в структуре данных, каждая из которых имеет свой собственный ограничивающий том. Ограничивающий том для родительского узла достаточно велик, чтобы заключить в него все дочерние объекты.

Мы могли бы вычислить ограничительную рамку на каждом уровне иерархии, чтобы она наилучшим образом соответствовала содержащемуся в ней объекту. Это даст нам наилучший возможный набор иерархических томов. Тем не менее, много раз мы можем выбрать более простой путь выбора ограничивающего тома для родительского узла, который включает ограничивающие тома всех его потомков. Это приводит к увеличению томов высокого уровня, но перерасчет ограничивающих томов может быть намного быстрее. Поэтому существует компромисс между эффективностью запроса определения потенциальных коллизий и скоростью построения структуры данных.

На рисунке 3.10 показана иерархия, содержащая четыре объекта и три слоя. Обратите внимание, что нет объектов, прикрепленных к родительским узлам на рисунке. Это не является абсолютным требованием: у нас могли бы быть объекты выше в дереве, при условии, что их ограничивающий том полностью охватывает их потомков. Однако в большинстве реализаций объекты находятся только внизу. Также общепринятой практикой является

наличие только двух детей для каждого узла в дереве (т.е. структуры данных двоичного дерева). Для этого есть математические причины (с точки зрения скорости выполнения запросов на столкновение), но наилучшей причиной использования двоичного дерева является простота реализации: она упрощает структуру данных и упрощает некоторые из алгоритмов.

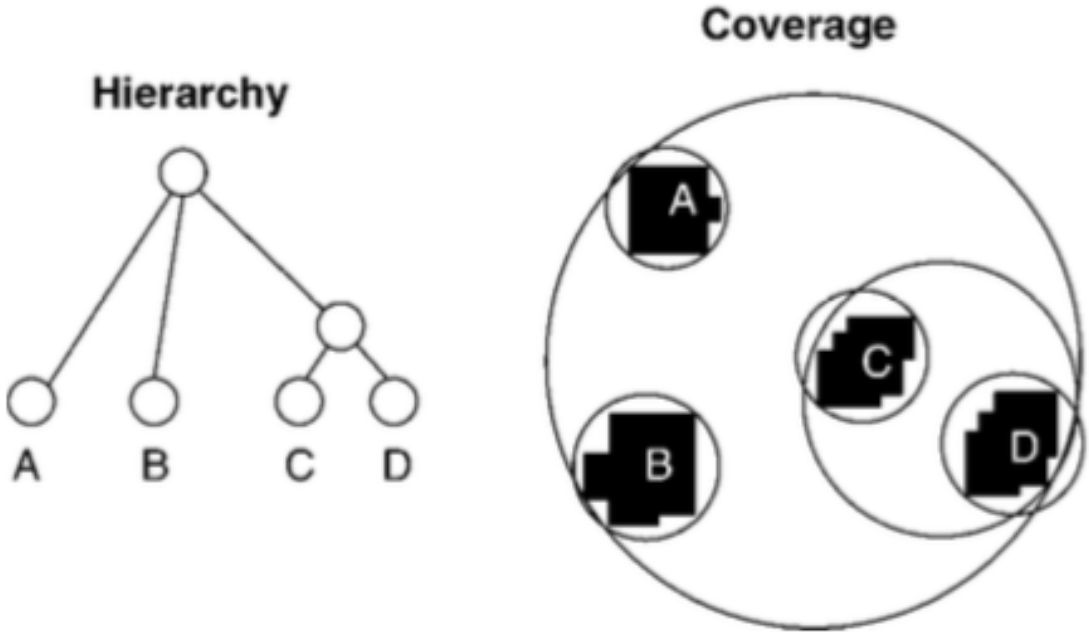

Рисунок 3.10 – Иерархия ограничивающих томов

Мы можем использовать такую иерархию для ускорения обнаружения столкновений: если ограничивающие тома двух узлов в дереве не касаются, то ни один из объектов, которые спускаются с этих узлов, не может быть в контакте. Путем тестирования двух ограничивающих томов в иерархии мы можем сразу исключить всех потомков.

```
struct PotentialContact
{
     RigidBody* body[2];
};
```

Если два высокоуровневых узла касаются друг друга, необходимо учитывать дочерние элементы каждого узла. Только комбинации тех детей, которые касаются, могут иметь потомков, которые находятся в контакте. На каждом этапе рассматриваются только те комбинации томов, которые касаются друг друга. Алгоритм, наконец, генерирует список потенциальных контактов между объектами. Этот список точно такой же, как если бы он был создан, рассматривая каждую возможную пару ограничивающих томов, но он во много раз быстрее.

Важным вопросом является построение иерархии. Возможно, ваш графический движок уже имеет иерархию ограничивающих томов. Иерархии томов широко используются для уменьшения количества объектов, которые нужно отрисовать. Корневой узел иерархии имеет свой том, проверенный на текущей камере. Если какая-либо часть ограничивающего тома может быть замечена камерой, то ее дочерние узлы проверяются рекурсивно. Если узел не может быть замечен камерой, ни один из его потомков не должен быть проверен. Это тот же алгоритм, который мы использовали для обнаружения контактов: на самом деле он эффективно проверяет наличие столкновений с видимой областью игры.

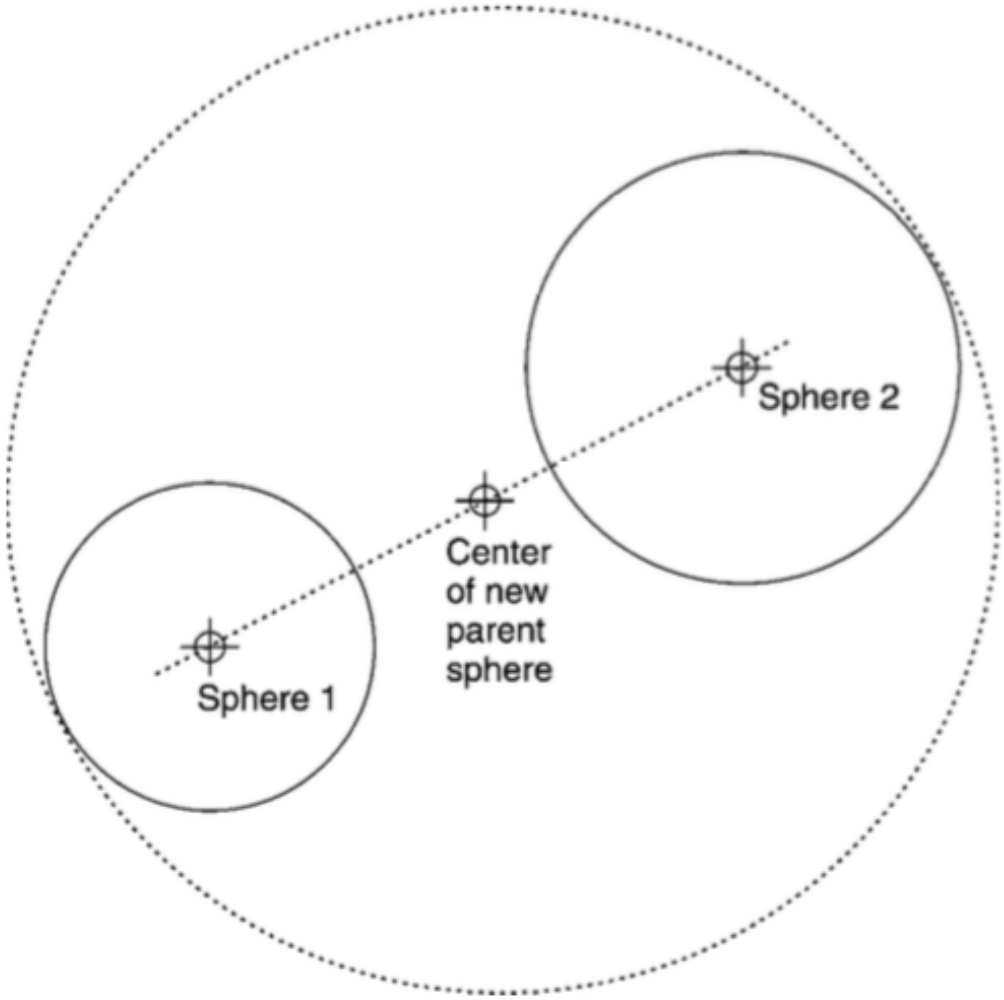

Рисунок 3.11 – Построение родительской сферы

Построение вставками является единственным подходящим для использования во время игры. Оно может регулировать иерархию без необходимости полностью ее перестраивать. Алгоритм начинается с существующего дерева (это может быть пустое дерево, если мы начинаем с нуля). Объект добавляется в дерево путем рекурсивного спуска по дереву: на

каждом узле выбирается дочерний элемент, который лучше всего подходит для нового объекта. В конце концов достигается существующий лист, который затем заменяется новым родителем как для существующего листа, так и для нового объекта.

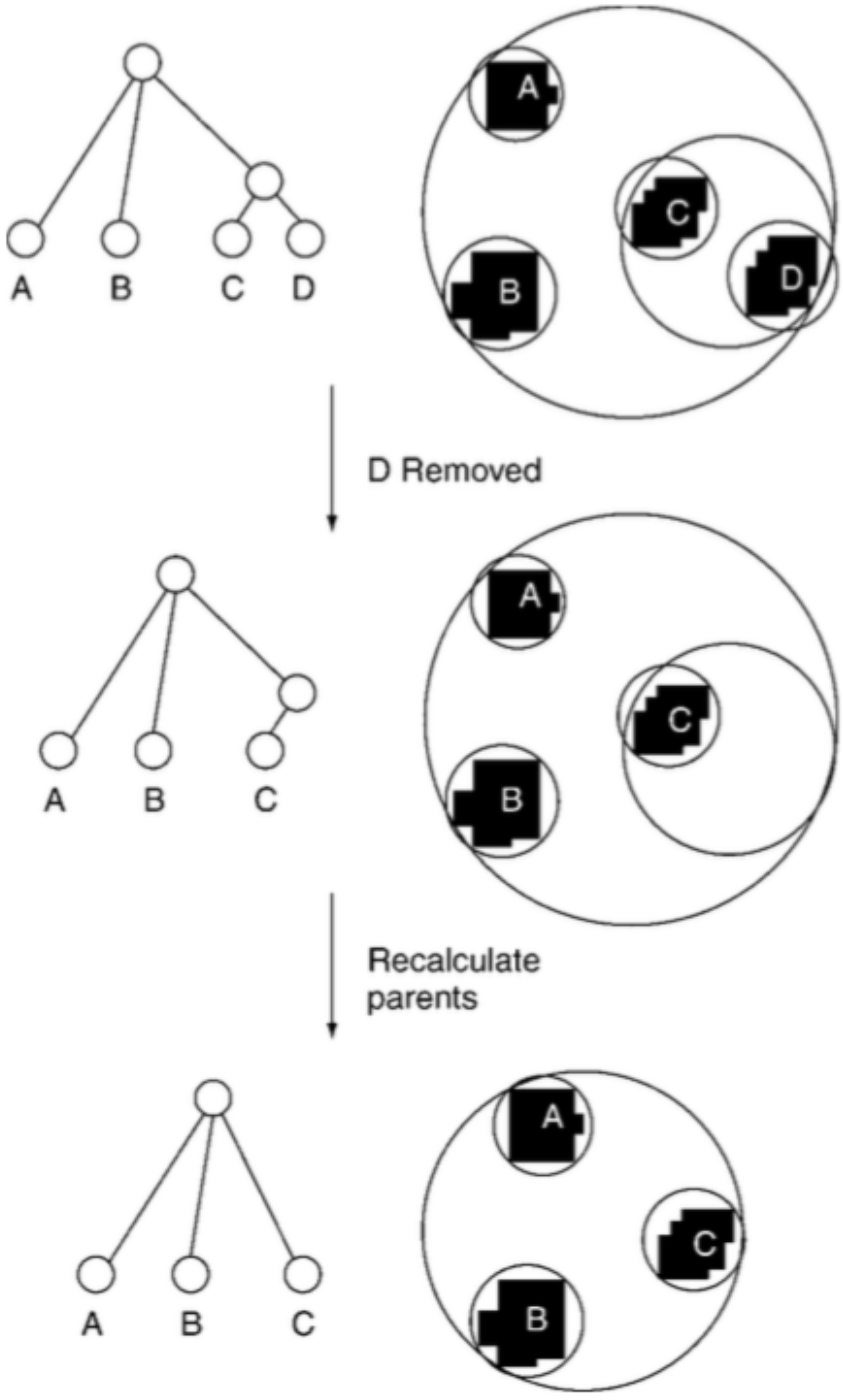

Рисунок 3.12 – Удаление узла

На каждом узле дерева мы выбираем дочерний элемент, ограничивающий том которого будет наименее расширен с добавлением

нового объекта. Новый ограничивающий том вычисляется на основе текущего ограничивающего тома и нового объекта. Находим линию между центрами обеих сфер, равно как и расстояние между краями двух сфер вдоль этой линии. Затем центральная точка помещается на эту линию между двумя крайними значениями, а радиус равен половине вычисленного расстояния. На рисунке 3.11 показан этот процесс.

Мы можем выполнить аналогичный алгоритм для удаления объекта. В этом случае полезно иметь доступ к родительскому узлу любого узла в дереве. Удаление объекта из иерархии предполагает замену его родительского узла его соседним элементом и пересчет ограничивающих томов дальше по иерархии, как показано на рисунке 3.12.