

Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

К ЗАЩИТЕ ДОПУСТИТЬ
Зав. каф. ЭВМ
_____ Д.И. Самаль

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к дипломному проекту
на тему
ПРОГРАММНОЕ СРЕДСТВО КОМПЬЮТЕРНОГО МОДЕЛИРОВАНИЯ
ФИЗИЧЕСКИХ ЗАКОНОВ РЕАЛЬНОГО МИРА В ВИРТУАЛЬНОЙ СРЕДЕ

БГУИР ДП 1–40 02 01 01 071 ПЗ

Студент	А.Д. Протько
Руководитель	И.В. Лукьянова
Консультанты:	
от кафедры ЭВМ	И.В. Лукьянова
по экономической части	И.В. Смирнов
Нормоконтролер	А.С. Сидорович
Рецензент	

МИНСК 2017

РЕФЕРАТ

Дипломный проект представлен следующим образом. Электронные носители: 1 дискета. Чертёжный материал: 6 листов формата А1. Пояснительная записка: 119 страницы, 34 рисунков, 8 литературных источников, 7 приложений.

Ключевые слова: обнаружение столкновений, твердое тело, физический движок, частица, разработка игр, физика твердых тел, движок реального времени.

Предметной областью дипломного проекта является сфера разработки игр. Объектом разработки является прикладное программное обеспечение.

Цель разработки – создание компьютерной программы, способной производить моделирование некоторых физических законов, а также симуляция поведения твердых тел.

Разработка осуществлялась с использованием языка программирования C++ с помощью интегрированной среды разработки Xcode под управлением OS X.

В результате работы было разработана нативная библиотека для OS X, способная моделировать физически реалистичное поведение различных тел и частиц.

Практическим применением разработки является использование разработанного программного обеспечения в исследовательских (для исследования поведения различных физических тела и взаимодействия между ними) и целях разработки компьютерных игр (большинство компьютерных игр в настоящее время широко используют физические движки).

Разработка является экономически эффективной и позволяет в значительной степени повысить показатели производительности труда.

Дипломный проект в достаточной степени является завершённой самостоятельной разработкой, однако в перспективе логичным дополнением будет являться создание версий для платформ Microsoft Windows и Linux.

Министерство образования Республики Беларусь
Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет: ФКСиС. Кафедра: ЭВМ.

Специальность: 40 02 01 «Вычислительные машины, системы и сети».

Специализация: 400201-01 Проектирование и применение локальных сетей.

УТВЕРЖДАЮ
Заведующий кафедрой ЭВМ
_____ Д.И. Самаль

ЗАДАНИЕ

по дипломному проекту студента
Протьюко Антона Дмитриевича

- 1 Тема проекта: «Программное средство компьютерного моделирования физических законов реального мира в виртуальной среде» – утверждена приказом по университету от 7 февраля 2017 г. № 238-с.
- 2 Срок сдачи студентом законченного проекта: 1 июня 2017 г.
- 3 Исходные данные к проекту:
 - 3.1 Платформа: OS X.
 - 3.2 Язык программирования: C++.
 - 3.3 Среда разработки: Xcode.
- 4 Содержание пояснительной записки (перечень подлежащих разработке вопросов):

Введение. 1. Обзор литературы. 2. Системное проектирование. 3. Функциональное проектирование. 4. Разработка программных модулей. 5. Программа и методика испытаний. 6. Руководство пользователя. 7. Расчёт экономической эффективности дипломного проекта. Заключение. Список использованных источников. Приложения.
- 5 Перечень графического материала (с точным указанием обязательных чертежей):
 - 5.1 Вводный плакат. Плакат.
 - 5.2 Заключительный плакат. Плакат.
 - 5.3 Программное средство компьютерного моделирования физических законов реального мира в виртуальной среде. Схема структурная.

- 5.4 Программное средство компьютерного моделирования физических законов реального мира в виртуальной среде. Диаграмма классов.
- 5.5 Программное средство компьютерного моделирования физических законов реального мира в виртуальной среде. Диаграмма последовательности.
- 5.6 Программное средство компьютерного моделирования физических законов реального мира в виртуальной среде. Схема программы.
- 6 Содержание задания по экономической части: «Технико-экономическое обоснование разработки программного средства компьютерного моделирования физических законов реального мира в виртуальной среде».

ЗАДАНИЕ ВЫДАЛ

И.В. Смирнов

КАЛЕНДАРНЫЙ ПЛАН

Наименование этапов дипломного проекта	Объём этапа, %	Срок выполнения этапа	Примечания
Изучение литературы	10	02.02 – 15.03	
Структурное проектирование	10	16.03 – 01.04	
Функциональное проектирование	20	2.04 – 23.04	
Разработка программных модулей	20	24.04 – 30.04	
Программа и методика испытаний	10	01.05 – 07.05	
Расчёт экономической эффективности	10	08.05 – 14.05	
Завершение оформления пояснительной записки	10	15.05 – 21.05	
Подготовка документов	10	22.05 – 28.05	

Дата выдачи задания: 2 февраля 2017

Руководитель

И.В. Лукьянова

ЗАДАНИЕ ПРИНЯЛ К ИСПОЛНЕНИЮ

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	6
1 ОБЗОР ЛИТЕРАТУРЫ.....	8
2 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ	18
3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ.....	27
4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ	56
5 ТЕСТИРОВАНИЕ И МЕТОДИКА ИСПЫТАНИЙ	65
6 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ	70
7 ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ ПРОГРАММНОГО СРЕДСТВА КОМПЬЮТЕРНОГО МОДЕЛИРОВАНИЯ ФИЗИЧЕСКИХ ЗАКОНОВ РЕАЛЬНОГО МИРА В ВИРТУАЛЬНОЙ СРЕДЕ	74
ЗАКЛЮЧЕНИЕ	81
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	83
ПРИЛОЖЕНИЕ А.....	84
ПРИЛОЖЕНИЕ Б	88
ПРИЛОЖЕНИЕ В	94
ПРИЛОЖЕНИЕ Г	105
ПРИЛОЖЕНИЕ Д.....	112
ПРИЛОЖЕНИЕ Е	118
ПРИЛОЖЕНИЕ Ж	119

ВВЕДЕНИЕ

Физический движок является важной частью большинства современных компьютерных игр. В реальном мире твердые тела не ведут себя странно, например, не проходят друг через друга сами по себе. Но в виртуальном пространстве игры объекты ничего не делают, если мы не говорим им об этом, и программисты должны постараться, чтобы гарантировать, что объекты не проходят друг через друга. Это роль одного из центральных компонентов любого игрового движка – системы обнаружения столкновений.

Система столкновения игрового движка часто тесно связана с физическим движком. Конечно, область физики обширна, и то, что большинство сегодняшних игровых движков называют «физикой», более точно описывается как моделирование динамики твердого тела. Твердое тело – идеализированный, бесконечно твердый, недеформируемый объект. Термин динамика относится к процессу определения того, как эти жесткие тела движутся и взаимодействуют во времени под воздействием сил. Моделирование динамики твердого тела позволяет передавать движения объектам в игре с высокой степенью интерактивности и естественным хаотическим образом – эффект, который намного сложнее достичь при использовании заранее подготовленных анимационных треков для перемещения объектов.

Динамическое моделирование активно использует систему обнаружения столкновений, чтобы правильно имитировать различные физические модели поведения объектов в системе, включая отскакивание друг от друга, скольжение при трении, вращение и остановку. Конечно, система обнаружения столкновений может использоваться автономно, без моделирования динамики – во многих играх нет системы «физики» вообще. Но все игры, которые связаны с объектами, движущимися в двух- или трехмерном пространстве, имеют некоторую форму обнаружения столкновений.

Основная цель системы обнаружения столкновений – определить, контактировал ли какой-либо из объектов в игровом мире. Чтобы ответить на этот вопрос, каждый логический объект представлен одной или более геометрическими фигурами. Эти формы обычно довольно просты, такие как окружности, прямоугольники и капсулы. Однако могут использоваться и более сложные формы. Система столкновений определяет, пересекаются ли какие-либо из фигур (то есть перекрываются) в любой данный момент времени.

В настоящее время большинство игровых движков обладают определенными возможностями физического моделирования. Некоторые физические эффекты, такие как реалистичное поведение тел, просто ожидаются игроками. Другие эффекты, такие как веревки, одежда, волосы или сложные механизмы, управляемые физикой, могут поднять игру на абсолютно иной уровень. В последние годы некоторые игровые студии начали

экспериментировать с продвинутыми физическими симуляторами, включая моделирование жидкости и деформируемых тел в реальном времени.

Задача данного дипломного проекта состоит в разработке программного обеспечения, реализующего физический движок реального времени, осуществляющий моделирование физики твердых тел в виртуальном пространстве.

В соответствии с поставленной целью были определены следующие задачи:

- обнаружение столкновений между динамическими объектами и геометрией статического мира;
- моделирование поведения твердых тела под действием силы тяжести и других сил;
- триггерные объекты (определяют, когда объекты входят, покидают или находятся внутри предварительно определенных областей в игровом мире);
- моделирование моментов сил при столкновении тел.

1 ОБЗОР ЛИТЕРАТУРЫ

В физическом движке одной из главных вещей является кинематика объектов – как они движутся со временем. Многие игровые движки включают физическую систему для моделирования движения объектов в виртуальном игровом мире физически реалистичным способом. С технической точки зрения, игровые физические движки, как правило, связаны с определенной областью физики, известной как динамика. Это исследование того, как силы влияют на движение объектов. До недавнего времени системы игровой физики были сфокусированы почти исключительно на конкретной субдисциплине, известной как классическая динамика твердого тела [1]. Это название означает, что в физическом моделировании игры сделаны два важных упрощающих допущения:

1) Классическая (ньютоновская) механика. Объекты моделирования считаются подчиненными законам движения Ньютона. Объекты достаточно велики, чтобы не было квантовых эффектов, и их скорости были достаточно низкими, чтобы не было релятивистских эффектов [2].

2) Твердые тела. Все объекты в моделировании являются абсолютно прочными и не могут деформироваться. Другими словами, их форма постоянна. Эта идея хорошо согласуется с предположениями, сделанными системой обнаружения столкновений. Более того, предположение о жесткости значительно упрощает математику, необходимую для моделирования динамики твердых объектов [3].

Физические движки также способны обеспечить различные ограничения движения твердых тел в игровом мире. Наиболее распространенное ограничение – это невозможность проникновения, другими словами, объекты не могут проходить друг через друга. Таким образом, физическая система пытается обеспечить реалистичные коллизионные отклики всякий раз, когда обнаруживается, что тела взаимопроникают. Это является одной из основных причин, по которым трудно взаимодействовать между физическим движком и системой обнаружения столкновений.

Большинство физических систем также позволяют разработчикам устанавливать другие виды ограничений, чтобы обеспечить реалистичные взаимодействия между физически симулированными твердыми телами. Они могут включать шарниры, призматические суставы (слайдеры), шаровые шарниры, колеса, «рэгдолы» для подражания бессознательным или мертвым персонажам и так далее.

Физическая система обычно разделяет структуру данных столкновений, и фактически она обычно управляет выполнением алгоритма обнаружения столкновений в рамках своей процедуры обновления временного шага. Как правило, между твердыми телами в моделировании динамики и столкновениями, управляемыми движком столкновений, имеется взаимно однозначное соответствие. Например, в Havok объект *hkpRigidBody* сохраняет

ссылку на один и только один *hkpCollidable* (хотя и присутствует возможность его создать без твердого тела). В PhysX эти два понятия более тесно интегрированы: *NxActor* служит одновременно и как объект, который можно перемещать, так и как твердое тело для моделирования динамики [4]. Эти твердые тела и их соответствующие элементы, отвечающие за столкновения, обычно сохраняются в одноэлементной структуре данных, известной как физический мир.

Твердые тела в физическом движении, как правило, отличаются от логических объектов, которые составляют виртуальный мир с точки зрения игрового процесса. Позиции и ориентации игровых объектов могут определяться физическим поведением. Для этого каждый кадр запрашиваются данные о каждом твердом теле у физического движка и эти данные применяются каким-либо образом к соответствующим игровым объектам. Также возможно, чтобы движение игрового объекта управляло положением и вращением твердого тела в физическом мире. Один логический игровой объект может быть представлен одним или многими твердым телом в физическом мире. Простой объект, такой как камень, оружие или бочонок, может соответствовать одному твердому телу, но сложная машина или персонаж могут состоять из множества взаимосвязанных тел.

1.1 Разделение линейной и угловой динамики

Твердое тело (если к нему не применено никаких ограничений) – это такое тело, которое может свободно перемещаться по всем трем декартовым осям и может свободно вращаться вокруг этих трех осей. Мы говорим, что такое тело имеет шесть степеней свободы (DOF).

Таким образом, движение твердого тела можно разделить на две независимые компоненты:

1) Линейная динамика. Это описание движения тела, когда мы игнорируем все вращательные эффекты. Мы можем использовать только линейную динамику для описания движения идеализированной точечной массы, т.е. бесконечно малой массы, которая не может вращаться.

2) Угловая динамика. Это описание вращательного движения тела. Чисто вращательное движение происходит, если каждая частица в теле движется по кругу вокруг одной линии. Эта линия называется осью вращения. Тогда радиус-векторы от оси до всех частиц одновременно испытывают одинаковое угловое смещение.

Как можно себе представить, эта возможность отделить линейные и угловые компоненты движения твердого тела чрезвычайно полезна при анализе или моделировании его поведения. Это означает, что можно вычислить линейное движение тела без учета вращения, как если бы оно было идеализированной точечной массой, а затем наложить свое угловое движение сверху, чтобы прийти к полному описанию движения тела.

1.2 Центр масс

Для целей линейной динамики неограниченное твердое тело действует так, как если бы вся его масса была сосредоточена в одной точке, известной как центр масс. Центр масс по существу является точкой равновесия тела для всех возможных положений. Другими словами, масса твердого тела распределена равномерно вокруг его центра масс во всех направлениях.

Для тела с равномерной плотностью центр масс лежит в центре тела. То есть, если бы мы должны были разделить тело на множество очень маленьких кусочков, сложить позиции всех этих частей в виде векторной суммы, а затем разделить на количество штук, мы получили бы довольно хорошее приближение к расположению центра масс. Если плотность тела неоднородна, положение каждого кусочка должно быть взвешено по массе этой части, что означает, что в целом центр масс действительно является средневзвешенным положением кусков. Так что,

$$r_{cm} = \frac{\sum_{\forall i} m_i r_i}{\sum_{\forall i} m_i} = \frac{\sum_{\forall i} m_i r_i}{m}, \quad (1.1)$$

где m представляет общую массу тела, а r – радиус-вектор или вектор позиции, т.е. вектор, простирающийся от начала мировых координат до рассматриваемой точки.

1.3 Линейная динамика

Для целей линейной динамики положение твердого тела можно полностью описать с помощью радиус-вектора r_{cm} , который простирается от начала мирового пространства до центра масс тела, как показано на рисунке 1.1.

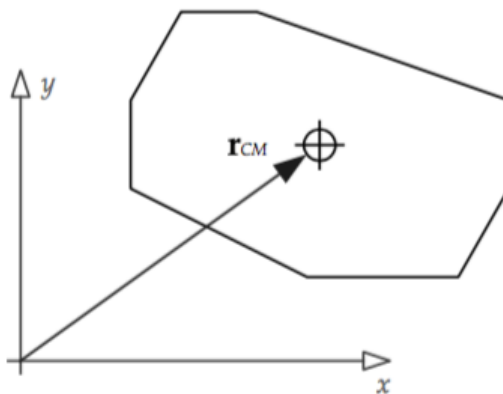


Рисунок 1.1 – Радиус-вектор центра масс тела

Основными необходимыми понятиями линейной динамики являются:

- скорость;
- ускорение;
- сила;
- импульс.

Линейная скорость твердого тела определяет скорость и направление, в котором движется центр масс тела. Это векторная величина, обычно измеряемая в метрах в секунду. Скорость – это первая производная от позиции, поэтому можно написать

$$v(t) = \frac{dr(t)}{dt} = \dot{r}(t). \quad (1.2)$$

Дифференцирование вектора такое же, как дифференцирование каждого компонента независимо, поэтому

$$v_x(t) = \frac{dr_x(t)}{dt} = \dot{r}_x(t), \quad (1.3)$$

и так далее для y - и z -компонент.

Линейное ускорение является первой производной от линейной скорости по времени или второй производной от положения центра масс тела в зависимости от времени. Ускорение – это векторная величина, обычно обозначаемая символом a . Таким образом, можно написать

$$a(t) = \frac{dv(t)}{dt} = \dot{v}(t). \quad (1.4)$$

Сила определяется как что-либо, что заставляет объект с массой ускоряться или замедляться. Сила имеет как величину, так и направление в пространстве, поэтому все силы представлены векторами. Силу часто обозначают символом F . Когда несколько сил прилагается к твердому телу, их влияние на линейное движение тела получается простым суммированием векторов силы:

$$F = \sum_{i=1}^N F_i. \quad (1.5)$$

Второй закон Ньютона гласит, что сила пропорциональна ускорению и массе:

$$F(t) = ma(t) = m\ddot{r}(t). \quad (1.6)$$

Когда мы умножаем линейную скорость тела на его массу, результатом будет величина, известная как линейный импульс. Линейный импульс принято обозначать символом p :

$$p(t) = mv(t). \quad (1.7)$$

Когда масса постоянна, справедливо равенство (1.6). Но если масса не постоянна, как было бы в случае с ракетой, топливо которой постепенно истощается и превращается в энергию, то уравнение (1.6) не совсем точно. Правильная формулировка фактически такова:

$$F(t) = \frac{dp(t)}{dt} = \frac{d(m(t)v(t))}{dt}, \quad (1.8)$$

что, конечно, сводится к (1.6), когда масса постоянна и может быть выведена вне производной. Линейный импульс для нас не представляет большой проблемы. Понятие импульса станет еще более полезным в угловой динамике. Подробнее эти понятия описаны в [5].

1.4 Угловая динамика

До сих пор мы сосредоточились на анализе линейного движения центра масс тела (который действует так, как если бы это была точечная масса). Как говорилось ранее, твердое тело будет вращаться вокруг своего центра масс. Это означает, что мы можем сложить угловое движение тела с линейным движением его центра масс, чтобы получить полное описание общего движения тела. Изучение вращательного движения тела в ответ на приложенные силы называется угловой динамикой.

В двух измерениях угловая динамика практически идентична линейной динамике. Для каждой линейной величины имеется угловой аналог, а математика работает почти также.

Основными необходимыми понятиями угловой динамики являются:

- угловая скорость;
- угловое ускорение;
- момент инерции;
- момент силы.

Каждое твердое тело можно рассматривать как тонкий лист материала. Все линейное движение происходит в плоскости xy , и все вращения происходят вокруг оси z .

Ориентация твердого тела в 2D полностью описывается углом θ , измеренным в радианах относительно некоторого согласованного нулевого вращения. Например, мы можем указать, что $\theta = 0$, когда гоночный автомобиль смотрит прямо вниз по положительной оси x в мировом

пространстве. Этот угол, конечно, является меняющейся во времени функцией, поэтому мы обозначаем ее $\theta(t)$.

Угловая скорость измеряет скорость изменения угла поворота тела с течением времени. В двухмерном пространстве угловая скорость является скаляром, более правильно называемым угловой скоростью, поскольку термин «скорость» действительно применим только к векторам. Он обозначается скалярной функцией $\omega(t)$ и измеряется в радианах в секунду. Угловая скорость является производной от угла поворота $\theta(t)$ по времени:

$$\omega(t) = \frac{d\theta(t)}{dt} = \dot{\theta}(t). \quad (1.9)$$

И как легко догадаться, угловое ускорение, обозначаемое $\alpha(t)$ и измеренное в радианах в секунду в квадрате, является скоростью изменения угловой скорости:

$$\alpha(t) = \frac{d\omega(t)}{dt} = \dot{\omega}(t) = \ddot{\theta}(t). \quad (1.10)$$

Вращательный эквивалент массы – это величина, известная как момент инерции. Так же, как масса описывает, насколько легко или трудно изменить линейную скорость точечной массы, момент инерции определяет, насколько легко или трудно изменить угловую скорость твердого тела вокруг определенной оси. Если масса тела сосредоточена вблизи оси вращения, относительно вращаться относительно этой оси будет относительно легче, и поэтому она будет иметь меньший момент инерции, чем тело, масса которого распространяется от этой оси.

Поскольку мы сейчас сосредоточиваемся на двумерной угловой динамике, ось вращения всегда равна z , а момент инерции тела является простым скалярным значением. Момент инерции обычно обозначается символом I . Как рассчитывать момент инерции подробно описано в [6].

До сих пор мы предполагали, что все силы приложены к центру массы твердого тела. Однако, вообще говоря, силы могут быть применены в произвольных точках тела. Если линия действия силы проходит через центр массы тела, тогда сила будет производить только линейное движение, как мы уже видели. В противном случае сила будет вводить вращательную силу, известную как крутящий момент, в дополнение к линейному движению, которое она обычно вызывает. Это показано на рисунке 1.2.

Мы можем рассчитать крутящий момент, используя векторное умножение. Во-первых, мы выражаем расположение, в котором сила применяется как вектор r , простирающийся от центра масс тела до точки приложения силы. (Другими словами, вектор r находится в пространстве тела, где начало пространства тела определяется как центр массы.) Это

проиллюстрировано на рисунке 1.3. Крутящий момент N , вызванный силой F , приложенной к точке тела с радиус-вектором r ,

$$N = r \times F. \quad (1.11)$$

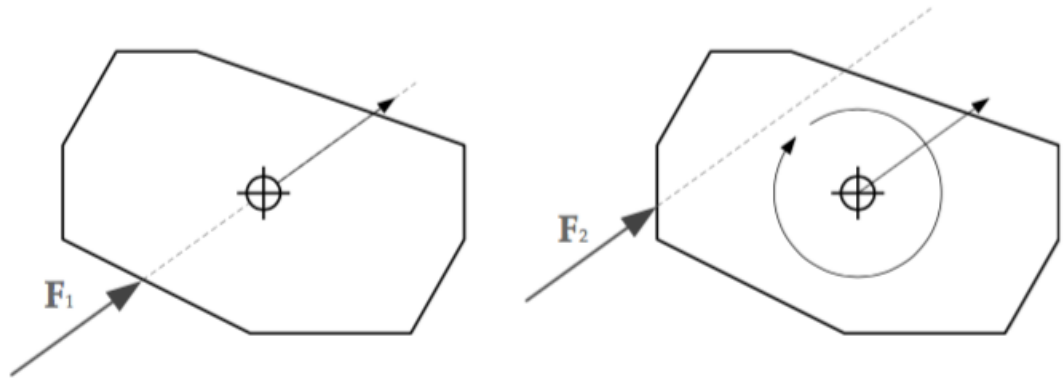


Рисунок 1.2 – Приложение силы к твердому телу

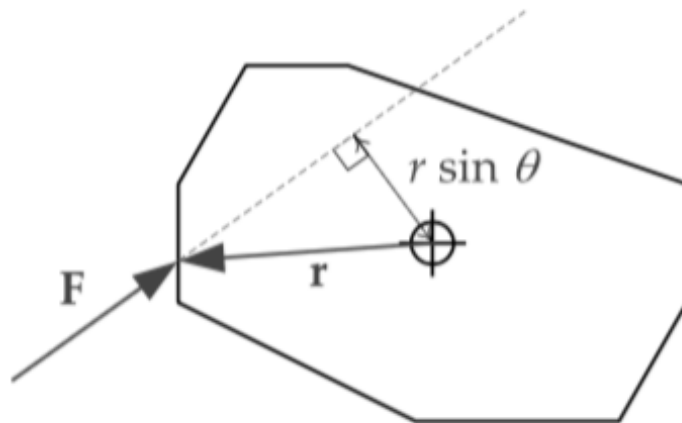


Рисунок 1.3 – Радиус-вектор к точке приложения силы

Из (1.11) следует, что крутящий момент увеличивается по мере того, как сила прикладывается дальше от центра масс. Это объясняет, почему рычаг может помочь нам перемещать тяжелый предмет. Это также объясняет, почему сила, приложенная непосредственно через центр массы, не создает крутящего момента и вращения – величина вектора r в этом случае равна нулю.

Когда к твердому телу прикладываются две или более силы, векторы крутящего момента, создаваемые каждым из них, можно суммировать, так же, как мы можем суммировать силы.

В двух измерениях векторы r и F оба должны лежать в плоскости xy , поэтому N всегда будет направлено вдоль положительной или отрицательной

оси z . По существу, мы будем обозначать двумерный крутящий момент через скаляр N_z , который является только z -компонентой вектора N .

Крутящий момент связан с угловым ускорением и моментом инерции во многом таким же образом, как сила связана с линейным ускорением и массой:

$$N_z = I\alpha(t) = I\dot{\omega}(t) = I\ddot{\theta}(t). \quad (1.12)$$

Все, что мы обсуждали до сих пор, предполагает, что наши твердые тела ни с чем не сталкиваются, и их движение не ограничено никаким другим способом. Когда тела сталкиваются друг с другом, моделирование динамики должно предпринимать шаги, чтобы гарантировать, что они реалистично реагируют на столкновение и что они никогда не остаются в состоянии взаимопроникновения после того, как этап моделирования завершен. Это называется реакцией на столкновение.

Однако прежде чем обсуждать реакцию на столкновение, мы должны дать определение энергии. Когда сила перемещает тело на расстояние, мы говорим, что сила работает. Работа представляет собой изменение энергии, то есть сила либо добавляет энергию к системе твердых тел (например, взрыва), либо удаляет энергию из системы (например, трение). Энергия приходит в двух формах. Потенциальная энергия тела V есть энергия, которую он имеет просто из-за того, где он находится относительно силового поля, такого как гравитационное или магнитное поле. (Например, чем выше тело находится над поверхностью Земли, тем больше энергия его гравитационного потенциала.) Кинетическая энергия тела T представляет собой энергию, возникающую из-за того, что она движется относительно других тел в системе. Полная энергия $E = V + T$ изолированной системы тел является сохраняющейся величиной, что означает, что она остается постоянной, если энергия не истощается из системы или не добавляется извне системы.

Кинетическая энергия, возникающая при линейном движении, может быть записана в виде

$$T_{linear} = \frac{1}{2} mv^2. \quad (1.13)$$

1.5 Реакция на столкновения

Когда два тела сталкиваются в реальном мире, происходит сложный набор событий. Тела слегка сжимаются, а затем отскакивают, изменяя свои скорости и теряя энергию, чтобы воспроизвести звук и тепло. Большинство симуляций динамики твердого тела в реальном времени аппроксимируют все эти детали простой моделью, основанной на анализе импульсов и кинетических энергий сталкивающихся объектов, называемых законом восстановления Ньютона для мгновенных столкновений без трения [7]. Он

делает следующие упрощающие допущения о столкновении:

- сила столкновения действует в течение бесконечно короткого периода времени, превращая его в то, что мы называем идеализированным импульсом. Это приводит к мгновенному изменению скоростей тел в результате столкновения;

- в точке контакта между поверхностями объектов нет трения. Это еще один способ сказать, что импульс, действующий на разделение тел во время столкновения, является нормальным для обеих поверхностей – в импульсе столкновения отсутствует тангенциальная составляющая;

- природа сложных субмолекулярных взаимодействий между телами во время столкновения может быть аппроксимирована одной величиной, известной как коэффициент восстановления, обычно обозначаемый символом ϵ . Этот коэффициент описывает, сколько энергии теряется при столкновении. При $\epsilon = 1$, столкновение совершенно упругое, и никакая энергия не теряется. Когда $\epsilon = 0$, столкновение совершенно неэластичное, и кинетическая энергия обоих тел теряется. Тела будут склеиваться после столкновения, продолжая двигаться в том направлении, в котором их взаимный центр масс двигался перед столкновением.

Весь анализ столкновений основан на идее сохранения линейного импульса. Итак, для двух тел 1 и 2 мы можем написать

$$p_1 + p_2 = p'_1 + p'_2, \quad (1.14)$$

где p_1 и p_2 – импульсы тел до столкновения; p'_1 и p'_2 – импульсы тел после столкновения.

Чтобы разрешить конфликт, используя закон восстановления Ньютона, мы сообщаем импульс двум телам. Импульс подобен силе, которая действует в течение бесконечно короткого периода времени и тем самым вызывает мгновенное изменение скорости тела, к которому оно применяется. Обозначим импульс Δp , так как это изменение импульса.

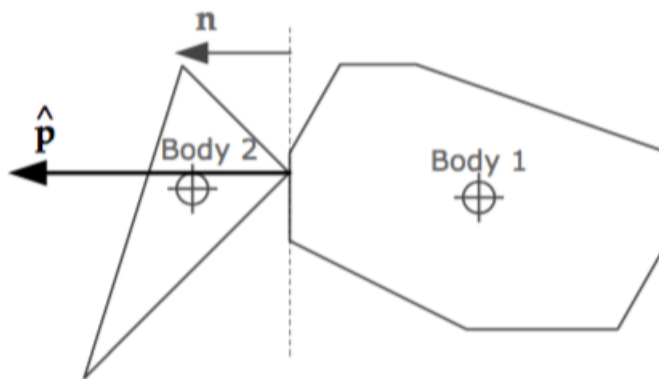


Рисунок 1.4 – Разрешающий импульс при столкновении тел

Поскольку мы предполагаем, что в столкновении нет трения, импульсный вектор должен быть нормальным к обеим поверхностям в точке контакта. Это показано на рисунке 1.4. Если предположить, что нормаль поверхности направлена к телу 1, то тело 1 испытывает импульс Δp , а тело 2 испытывает равный по модулю, но противоположный по направлению импульс Δp . Следовательно, импульсы двух тел после столкновения могут быть выражены через их импульсы до столкновения и импульс Δp следующим образом:

$$p'_1 = p_1 + \Delta p, \quad (1.15)$$

$$m_1 v'_1 = m_1 v_1 + \Delta p, \quad (1.16)$$

$$v'_1 = v_1 + \frac{\Delta p}{m_1} n. \quad (1.17)$$

Коэффициент восстановления обеспечивает ключевую связь между относительными скоростями тел до и после столкновения. Учитывая, что центры масс тел имеют скорости перед столкновением и впоследствии, коэффициент восстановления «определяется следующим образом:

$$v'_2 - v'_1 = \varepsilon(v_2 - v_1). \quad (1.18)$$

Подставив уравнения (1.17) и (1.18) при допущении, что тела не могут вращаться, получим

$$\Delta p = \frac{(\varepsilon + 1)(v_2 n - v_1 n)}{\frac{1}{m_1} + \frac{1}{m_2}} n. \quad (1.19)$$

Решение становится более сложным, если учитывать поворот тел. В этом случае нужно смотреть на скорости точек соприкосновения на двух телах, а не на скорости их центров масс, и нужно вычислить импульс таким образом, чтобы придать реалистичный вращательный эффект как результат столкновения. Более подробно это описано в [8].

2 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

Целью данного дипломного проекта является разработка программного средства для моделирования реалистического поведения тел путем ограниченной симуляции некоторых основных физических законов. Таким образом, в ходе проектирования было выделено восемь основных модулей:

- 1) Модуль физического мира.
- 2) Модуль обнаружения столкновений.
- 3) Модуль разрешения контактов.
- 4) Модуль рэйкастинга.
- 5) Модуль движения тел.
- 6) Модуль линейной динамики.
- 7) Модуль угловой динамики.
- 8) Модуль отладочной отрисовки.

Рассмотрим подробнее цели и задачи каждого из вышеперечисленных модулей.

2.1 Модуль физического мира

Физический мир представляет собой центральный компонент нашего физического движка. Он хранит в себе информацию о всех физических телах, а также описания соответствующих им геометрических примитивов, являющихся, по сути, формой тел, если таковые присутствуют. Этот модуль осуществляет взаимодействие между остальными и именно через него происходит контроль глобальных параметров физической симуляции, таких как, частота обновления, сила гравитации и т.д.

2.2 Модуль обнаружения столкновений

Чтобы объекты имели возможность сталкиваться, мы должны предоставить им описание столкновения, формы объекта, его положения и ориентации в мировом пространстве. Это особая структура данных, отдельная от представления игрового процесса объекта (код и данные, определяющие его роль и поведение в игре) и от его визуального представления.

С точки зрения обнаружения столкновений в целом более предпочтительны формы, геометрически и математически простые. На примере представления реальных объектов можно привести следующие:

- скала может быть смоделирована как сфера;
- капот автомобиля может быть представлен прямоугольной коробкой;
- человеческое тело может быть аппроксимировано совокупностью взаимосвязанных капсул (фигуры в форме таблеток).

В идеале мы должны прибегать к более сложной форме только тогда, когда более простое представление оказывается недостаточным для

достижения желаемого поведения в игре. На рисунке 2.1 показаны несколько примеров использования простых форм для аппроксимации фигур объектов для обнаружения столкновений.



Рисунок 2.1 – Использование простых фигур для представления формы объектов

Для дальнейшего описания модуля введем такие понятия, как форма и трансформация. Форма описывает геометрические свойства объекта, а трансформация описывает его положение и ориентацию в мировом пространстве. Нам необходима трансформация по трем причинам:

- 1) С технической точки зрения, форма описывает только геометрию объекта (то есть, будь то сфера, коробка, капсула или какой-то другой примитив). Она также может описывать размер объекта (например, радиус сферы или размеры ящика). Но форма обычно определяется с центром в начале координат и в некоторой канонической ориентации относительно осей координат. Чтобы ее правильно использовать, форма должна быть трансформирована таким образом, чтобы позиционировать ее и соответствующим образом ориентировать ее в мировом пространстве.

- 2) Объекты являются динамическими. Перемещение произвольной сложной формы в пространстве может быть дорогостоящим, если нам нужно индивидуально перемещать составляющие части формы (вершины, плоскости и так далее). Но с помощью трансформации любая форма может перемещаться в пространстве легко, независимо от того, насколько она проста или сложна.

- 3) Информация, описывающая некоторые более сложные виды фигур, может занимать большой объем памяти. Таким образом, может быть полезно разрешить нескольким объектам совместно использовать одно описание

формы. Например, в гоночной игре информация о форме для многих автомобилей может быть идентичной. В этом случае все сопутствующие автомобили в игре могут иметь одну форму автомобиля.

Модуль обнаружения столкновений упаковывает информацию о столкновении в удобную структуру данных, которая может быть создана для каждого обнаруженного контакта. Эта информация часто включает в себя разделительный вектор – вектор, по которому мы можем провести объекты, чтобы правильно вывести их из столкновения. Структура также, как правило, содержит информацию о том, какие два объекта столкнулись, в том числе, какие отдельные фигуры пересекались. Модуль также может возвращать дополнительную информацию, такую как скорость тел, проецируемую на разделительную нормаль. На рисунке 2.2 представлен пример обнаружения столкновений.

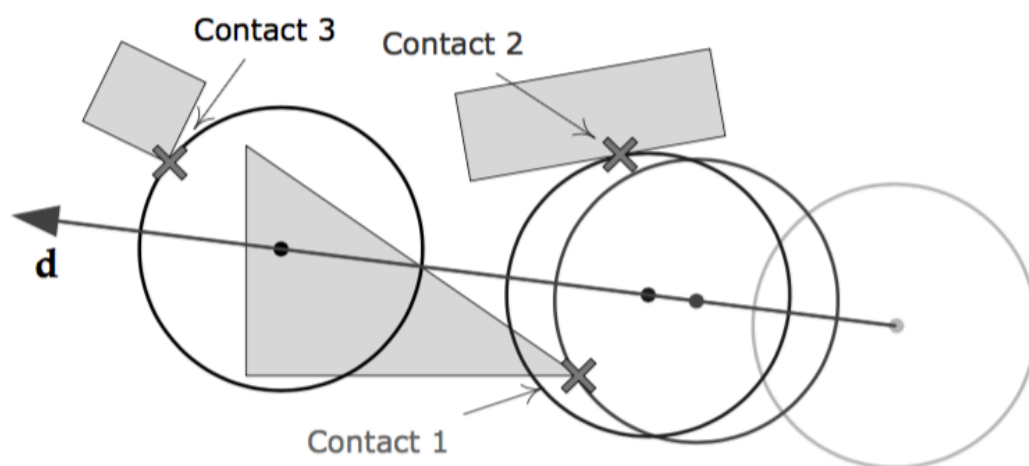


Рисунок 2.2 – Обнаружение столкновений

Система столкновений использует аналитическую геометрию – математические описания трехмерных объемов и поверхностей – для того, чтобы вычислить пересечения между формами.

Большинство систем обнаружения столкновений широко используют теорему, известную как теорема о разделительной оси. Она утверждает, что если найдется ось, вдоль которой не пересекается проекция двух выпуклых форм, то мы можем быть уверены, что две формы вообще не пересекаются. Если такая ось не существует и формы выпуклы, то мы точно знаем, что они пересекаются. Если формы вогнуты, то они не могут быть взаимопроницаемыми, несмотря на отсутствие разделительной оси. Это одна из причин, по которым предпочтительнее использовать выпуклые фигуры при обнаружении столкновений.

Эту теорему легче всего представить в двух измерениях. Интуитивно она говорит, что если линия может быть найдена, так что объект А целиком

находится на одной стороне линии, а объект В целиком на другой стороне, то объекты А и В не пересекаются. Такая линия называется разделительной линией и всегда перпендикулярна оси разделения. Поэтому, как только мы нашли разделяющую линию, легко убедиться, что теория на самом деле верна, глядя на проекции наших фигур на ось, перпендикулярную разделительной линии.

Проекция двумерной выпуклой формы на ось действует как тень, которую объект должен был оставить на тонкой проволоке. Это всегда отрезок линии, лежащий на оси, который представляет максимальные размеры объекта в направлении оси. Мы можем также думать о проекции как о минимальной и максимальной координатах вдоль оси, которую мы можем записать как полностью закрытый интервал. Как видно на рисунке 2.3, когда разделительная линия существует между двумя формами, их проекции не пересекаются вдоль оси разделения. Однако выступы могут перекрываться по другим, не разделяющим осям.

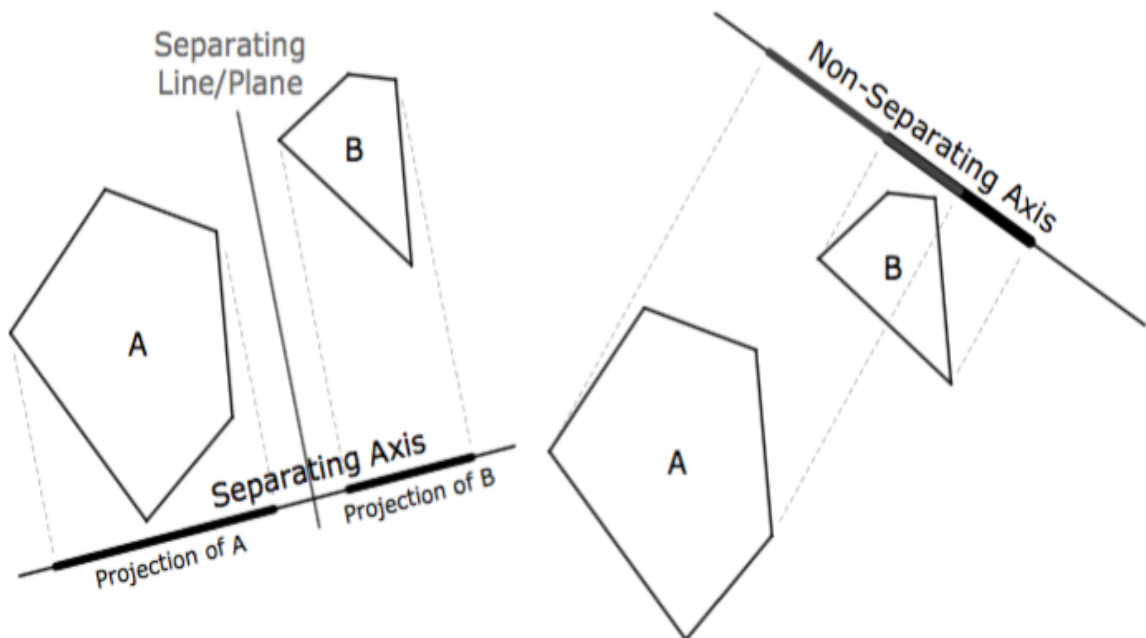


Рисунок 2.3 – Разделительная ось

В трех измерениях разделительная линия становится разделительной плоскостью, но ось разделения остается осью (то есть бесконечной линией). Опять же, проецирование трехмерной выпуклой формы на ось является отрезком линии, который мы можем представить полностью закрытым интервалом.

Одним из примеров этого принципа в действии является проверка пересечения сферы со сферой. Если две сферы не пересекаются, то ось, параллельная отрезку, соединяющему центральные точки сфер, всегда будет

действительной разделительной осью (хотя могут существовать и другие разделительные оси, в зависимости от того, как далеко друг от друга находятся две сферы). Чтобы визуализировать это, можно рассмотреть предел, когда две сферы как раз коснутся, но еще не вступили в контакт. В этом случае единственной разделяющей осью является одна параллель к сегменту от центра к центру. По мере того, как сферы раздвигаются, мы можем поворачивать ось разделения все больше и больше в любом направлении. Это показано на рисунке 2.4.

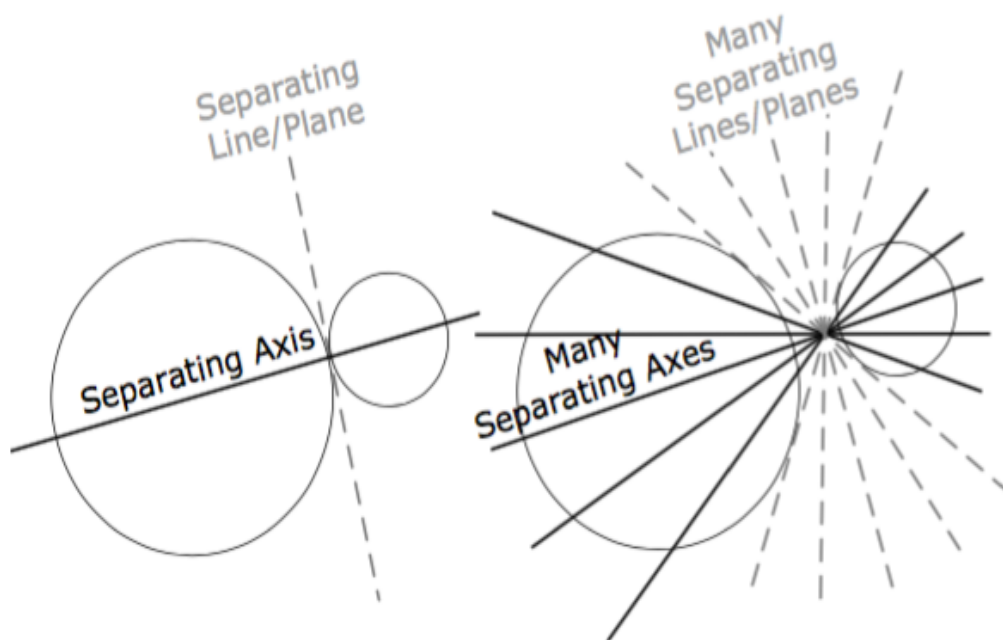


Рис 2.4 – Проверка столкновения сфер

2.3 Модуль разрешения контактов

Этот модуль, используя данные о столкновении, полученные от модуля обнаружения столкновений, корректирует поведение тел после контакта. Это значит, что на основе скоростей, масс и других характеристик столкнувшихся тел, он прикладывает к ним такие силы, чтобы произвести реалистичный эффект отскакивающих в результате коллизии объектов.

2.4 Модуль рэйкастинга

Еще одна важная задача физического движка – ответить на гипотетические вопросы о столкновениях в мире. Например:

- если пуля вылетит из оружия игрока в определенном направлении, какова будет первая цель, на которую она попадет, если таковая имеется;
- может ли транспортное средство переместиться из пункта А в пункт

Б без каких-либо действий по пути;

– найти все объекты противника в пределах заданного радиуса персонажа.

В общем случае такие операции называются коллизийными запросами. Наиболее распространенный вид запроса – это *collision cast*, иногда просто называемый *cast*. Этот термин определяет, что будет, если гипотетический объект будет помещен в физический мир и перемещен вдоль луча или отрезка. Он отличается от обычных операций обнаружения столкновений, потому что объект, находящийся в процессе трансляции, на самом деле не находится в физическом мире – он никак не может повлиять на другие объекты в мире. В качестве объектов для проведения данных запросов могут быть сферы, отрезки, кубы и так далее. На рисунке 2.5 представлен пример запросов для окружности.

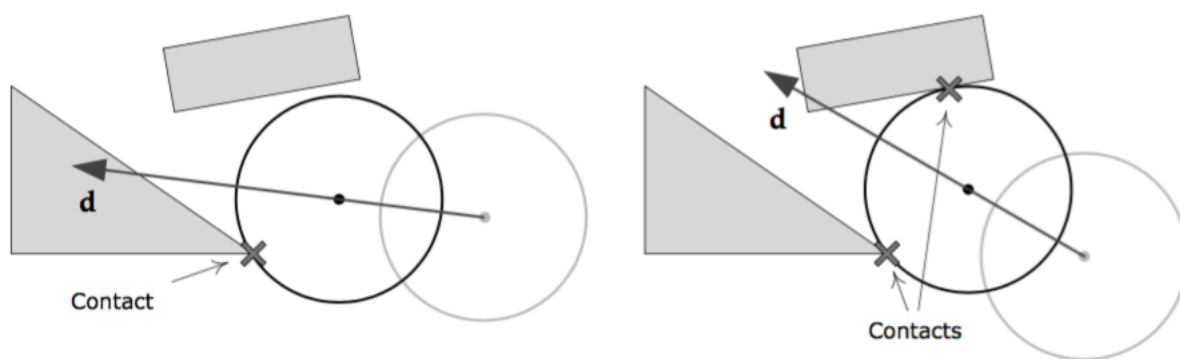


Рисунок 2.5 – Коллизийные запросы для окружности

Рэйкастинг часто используется в играх. Например, мы могли бы спросить систему столкновений, может ли персонаж А видеть персонажа В. Чтобы определить это, мы просто проводим направленный отрезок линии от глаз персонажа А до персонажа В. Если луч обращается к символу В, мы знаем, что А может «видеть» В. Но если луч ударяет какой-то другой объект до достижения символа В, мы знаем, что поле зрения блокируется этим объектом. Рэйкасты используются системами оружия (например, для определения попадания пули), механикой игрока (например, чтобы определить, есть ли твердая почва под ногами персонажа), системы искусственного интеллекта (например, проверки прямой видимости, запросы движения и т.д.), системы транспортных средств (например, для определения местоположения и привязки шин транспортного средства к местности) и т.д.

Другой распространенный запрос заключается в том, чтобы спросить систему столкновений, насколько воображаемая выпуклая форма сможет перемещаться вдоль направленного отрезка перед тем, как она ударит что-то твердое. Это называется столкновение со сферой, когда проверяемая форма

представляет собой сферу или другую фигуру. Как и в случае с трансформированием луча, заливка фигуры обычно описывается указанием начальной точки, расстояния до перемещения и, конечно, типа, размеров и ориентации фигуры, которую мы хотим наложить.

Есть два случая для проверки выпуклой фигуры:

- фигура уже взаимопроникает или контактирует, по крайней мере, с одним объектом, препятствуя своему удалению от своего исходного положения;
- фигура не пересекается ни с чем другим в начальном местоположении, поэтому она может свободно перемещаться на ненулевое расстояние вдоль своего пути.

В первом случае система столкновения обычно сообщает о контакте между фигурой и всеми элементами, с которыми она изначально пересекается. Эти контакты могут находиться внутри фигуры или на ее поверхности, как показано на рисунке 2.6.

Во втором случае фигура может переместиться на ненулевое расстояние вдоль отрезка перед тем, как с чем-то столкнуться. Мы предполагаем, что она столкнется с чем-то и, как правило, это только один объект. Тем не менее, фигура может удариться одновременно о несколько фигур, в зависимости от траектории. И, конечно, если столкнувшаяся фигура является невыпуклым многоугольником, она может в конечном итоге касаться более чем одной его части одновременно. Мы можем с уверенностью сказать, что вне зависимости от того, какая из выпуклых фигур проверяется, для кастинга можно создать несколько точек контакта. В этом случае контакты всегда будут на поверхности формы, а не внутри нее (потому что мы знаем, что фигура не соприкасалась с чем-либо, когда она начала свое движение). Этот случай показан на рисунке 2.5.

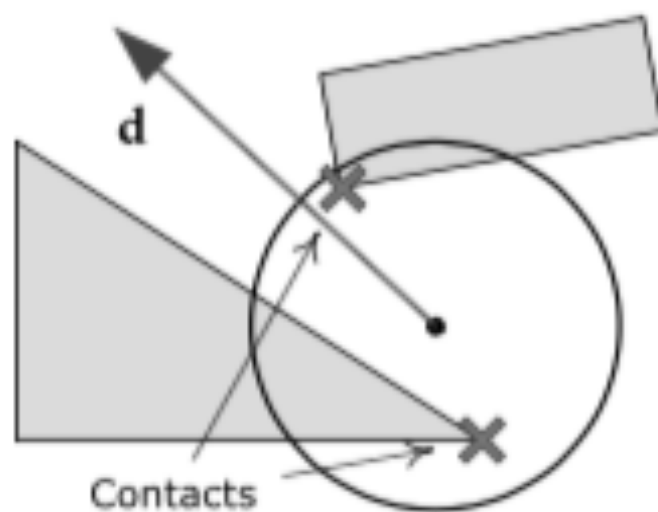


Рисунок 2.6 – Проверка сферы с несколькими контактами

2.5 Модуль движения тел

Данный модуль производит расчеты, связанные с движением, для всех тел, находящихся в физическом мире. Он определяет и сообщает в какой позиции и с каким вращением окажется физическое тело при данных силах, действующих на него в следующий момент времени. Модуль движения тел использует модули угловой и линейной динамики для этих вычислений, и впоследствии обновляет состояние физического мира новыми данными. Так как динамика движения тел разделена на угловую и линейную, этот модуль также должен применить изменения как позиции объекта, так и его вращения. На рисунке 2.7 изображено линейное и угловое движение тела. Еще одной задачей модуля является вычисление моментов вращения, если сила к телу была приложена не в центре масс, наряду с расчетом центров масс тел и их плотности.

2.6 Модуль линейной динамики

Модуль предоставляет возможности для управления всем что связано с линейным движением тел – приложение сил, импульсов и так далее. Любое количество сил может быть применено к телу в физическом мире. Сила всегда действует за конечный промежуток времени. (Если бы это действовало мгновенно, это было бы импульсом). Силы носят динамический характер – они часто изменяют свои направления и величины с каждым кадром. Таким образом, применять силу к телу можно только раз в течении кадра физического мира и всего времени воздействия силы.

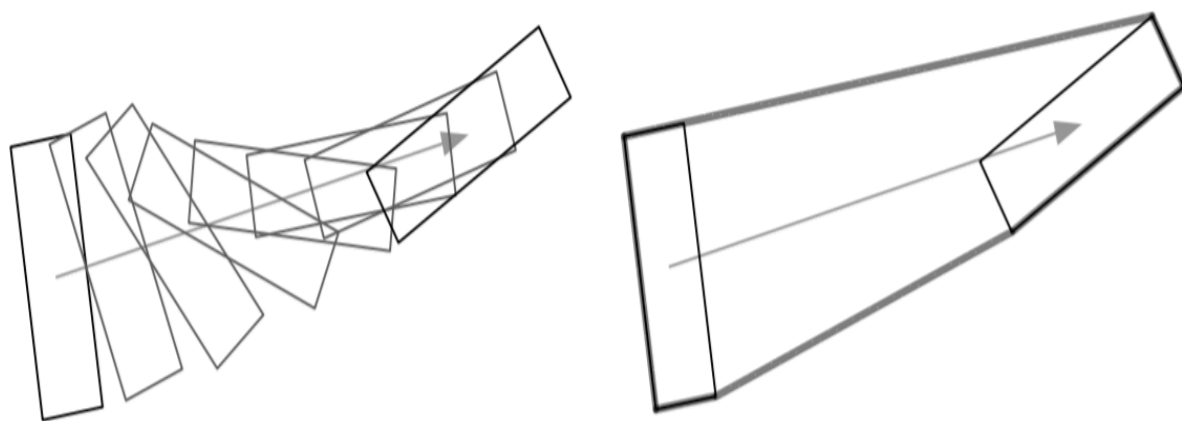


Рисунок 2.7 – Движение объекта с вращением

2.7 Модуль угловой динамики

Модуль предоставляет возможности для управления всем что связано с

угловым движением тел. Самая главная функция – это приложение момента вращения к телу. Чистый крутящий момент можно применить к телу, применяя две равные и противоположные силы к точкам, равноудаленным от центра масс. Линейные движения, вызванные такой парой сил, будут компенсировать друг друга, и это оставляет только их вращательные эффекты.

2.8 Модуль отладочной отрисовки

При разработке физического движка очень важно иметь возможность видеть результаты симуляции тел, чтобы можно было оценить правильность его работы. Модуль отладочной отрисовки ставит своей целью графическое отображение всех объектов физического мира в наглядной форме.

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`, `ParticleAnchoredSpring`, `ParticleBungee`, `ParticleBuoyancy`, `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, замедляются и останавливаются, 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 рассматривать 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 вместе с соответствующими 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 сложены 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 `s`, 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`, `ParticleAnchoredSpring`, `ParticleBungee`, `ParticleBuoyancy`, `ParticleFakeSpring`

Одна из самых полезных сил, которую можно добавить в движок, – 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 находится в классе `ParticleAnchorSpringGenerator`.

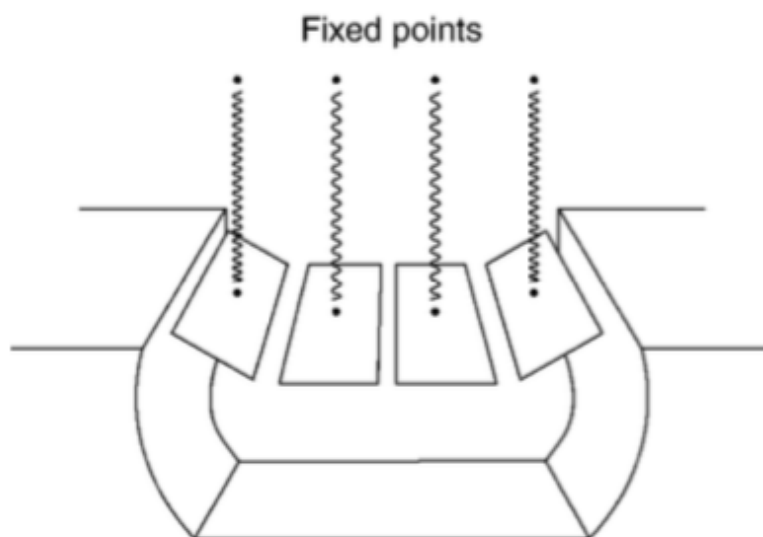


Рисунок 3.1 – Пружинный мост

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 близко 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.

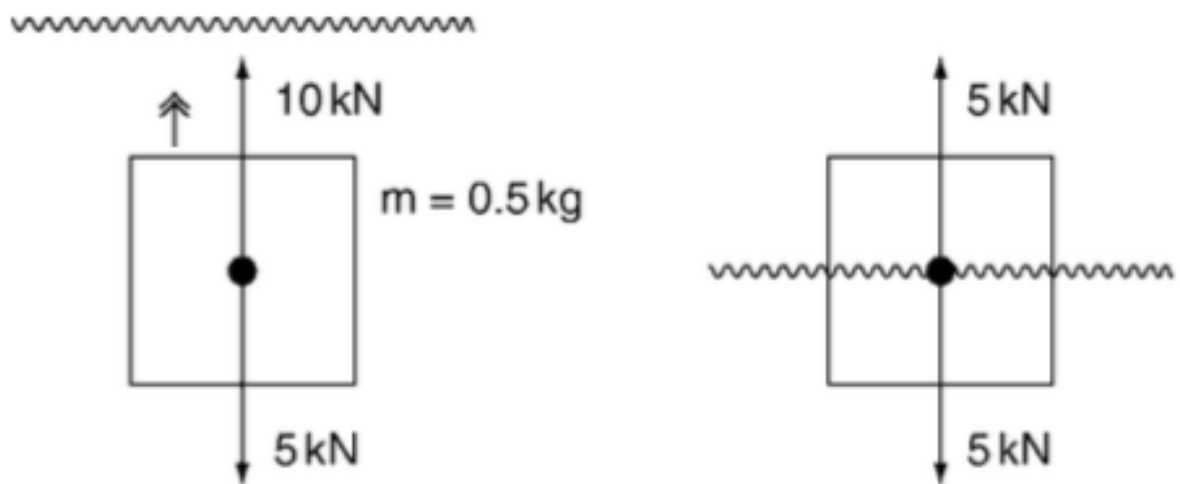


Рисунок 3.2 – Плавучий блок и сила выталкивания

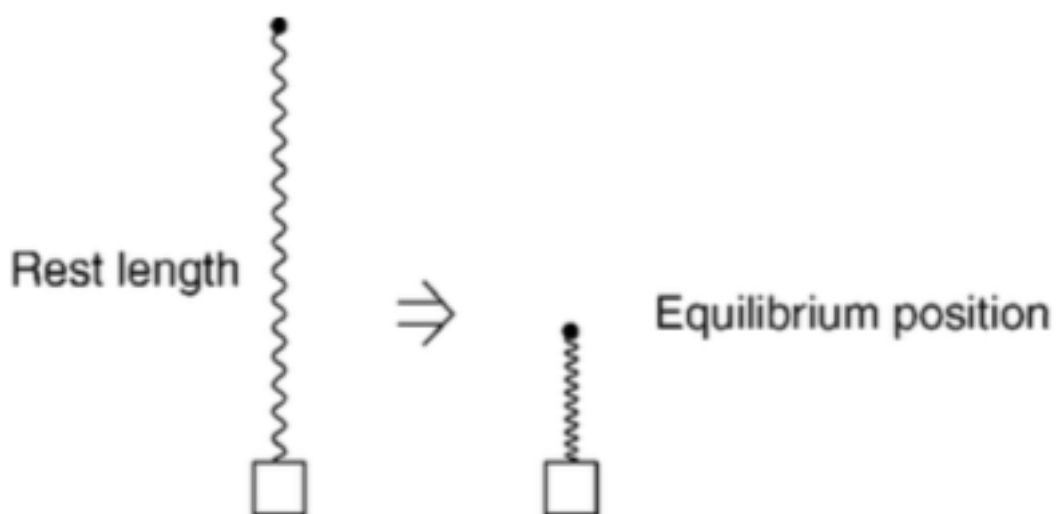


Рисунок 3.3 – Длина покоя пружины и позиция равновесия

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 ограничений. 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 восстановления 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, взаимодействуют 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.

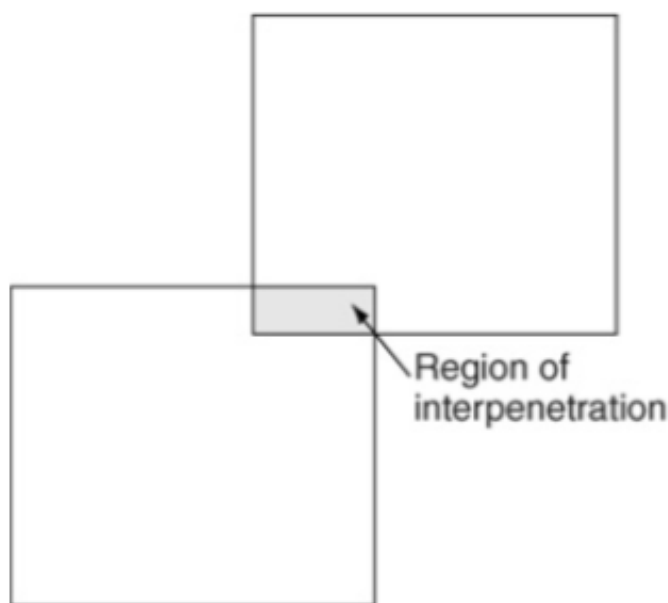


Рисунок 3.4 – Взаимопроникновение объектов

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`.

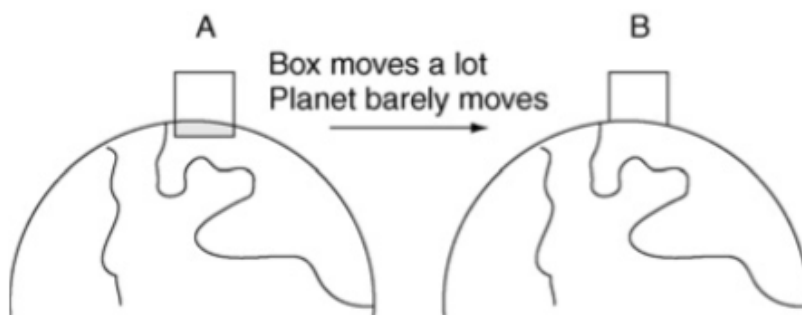


Рисунок 3.5 – Проблема разрешения взаимопроникновений

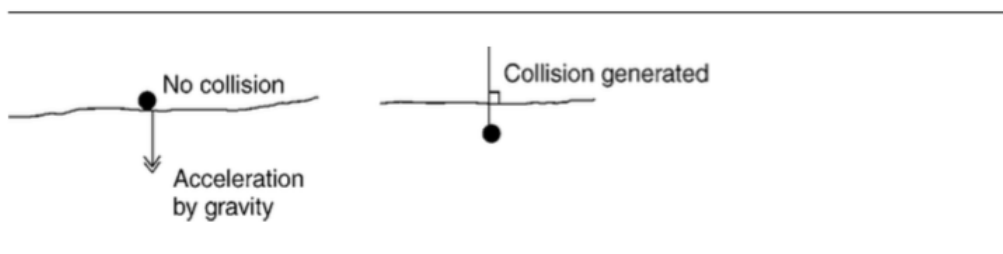


Рисунок 3.6 – Вибрация при столкновении покоя

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 большей 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 отличие 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.

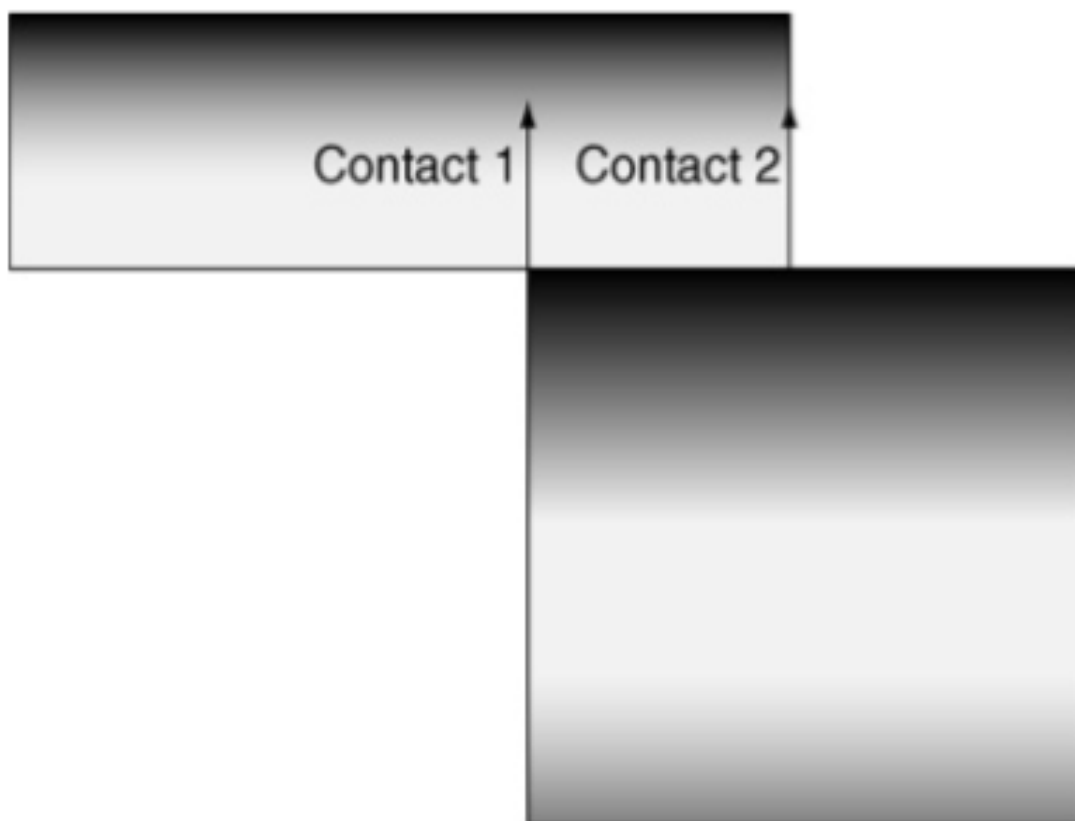


Рисунок 3.7 – Два контакта при столкновении

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. Именно 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 в 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 распознаватели и разрешает список контактов.

Необходимо добавлять 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 отличие 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 `RigidBody` 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.

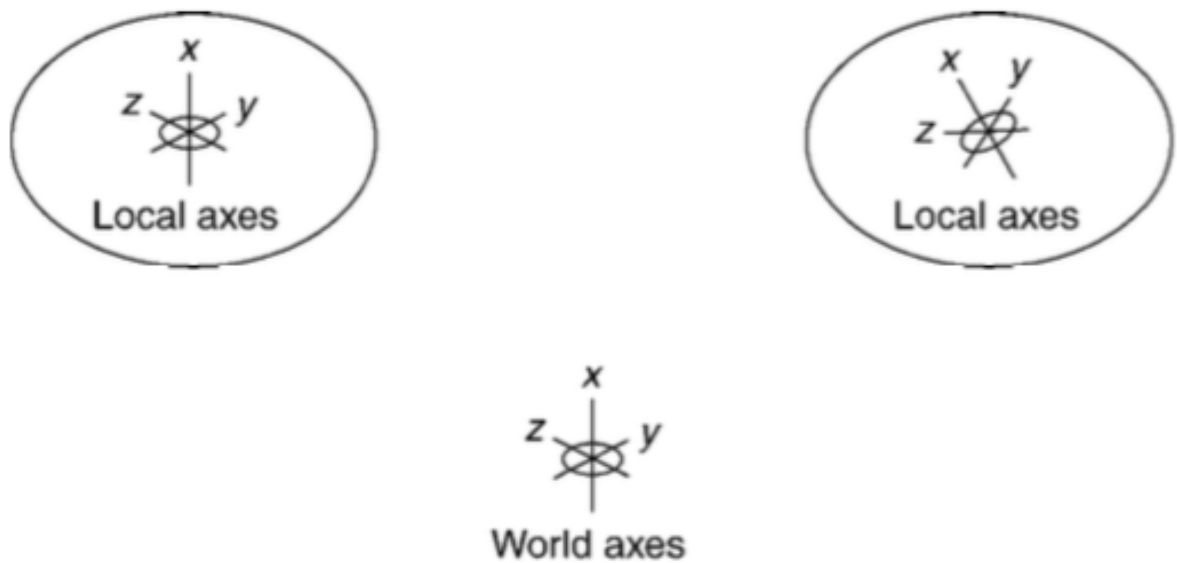


Рисунок 3.8 – Локальность момента инерции

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 скомбинируем 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.

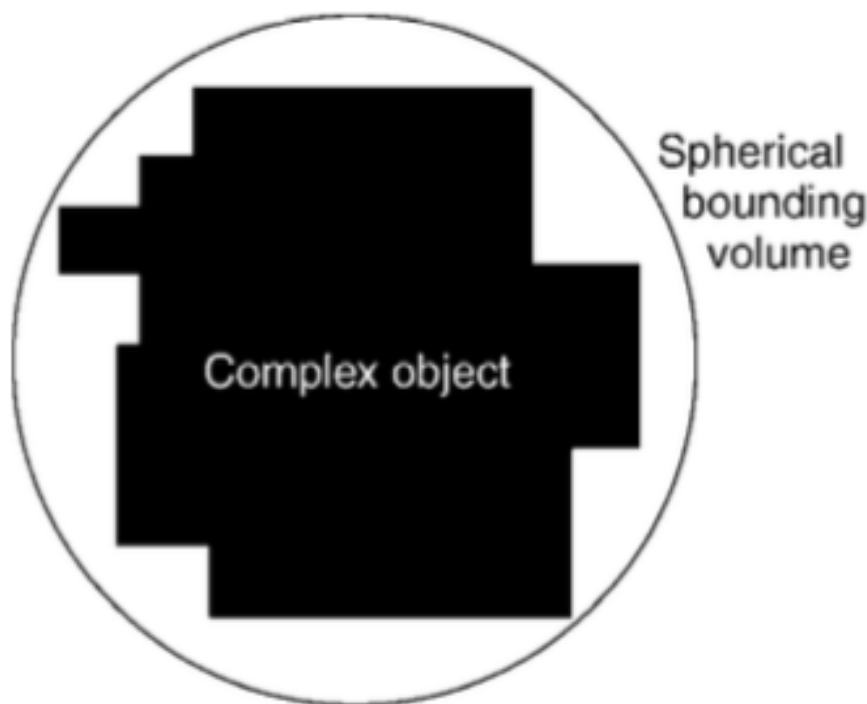


Рисунок 3.9 – Ограничивающий том

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.

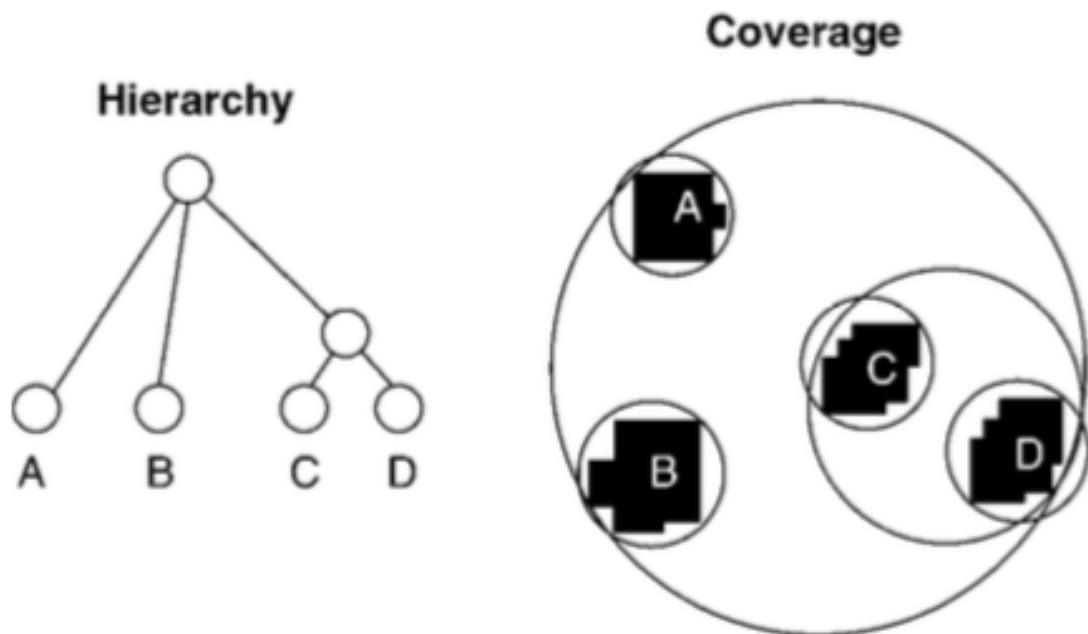


Рисунок 3.10 – Иерархия ограничивающих томов

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.

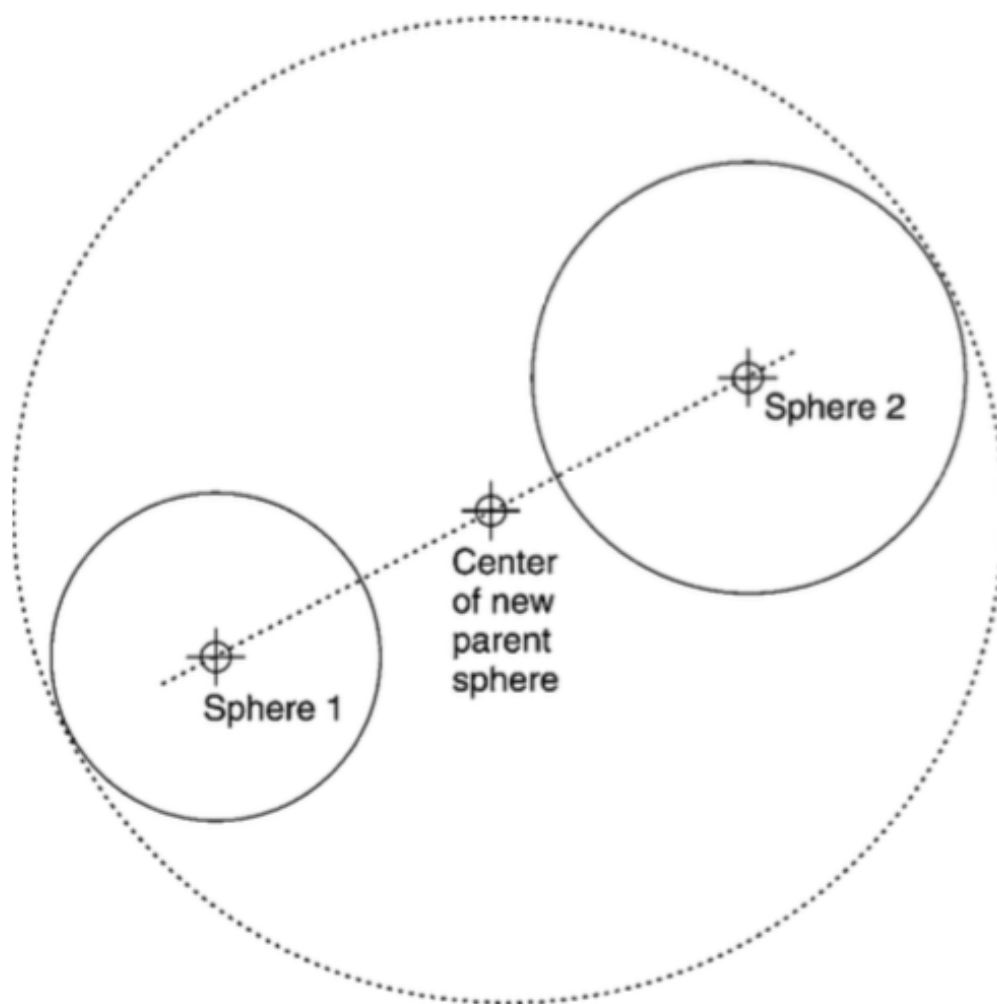


Рисунок 3.11 – Построение родительской сферы

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.

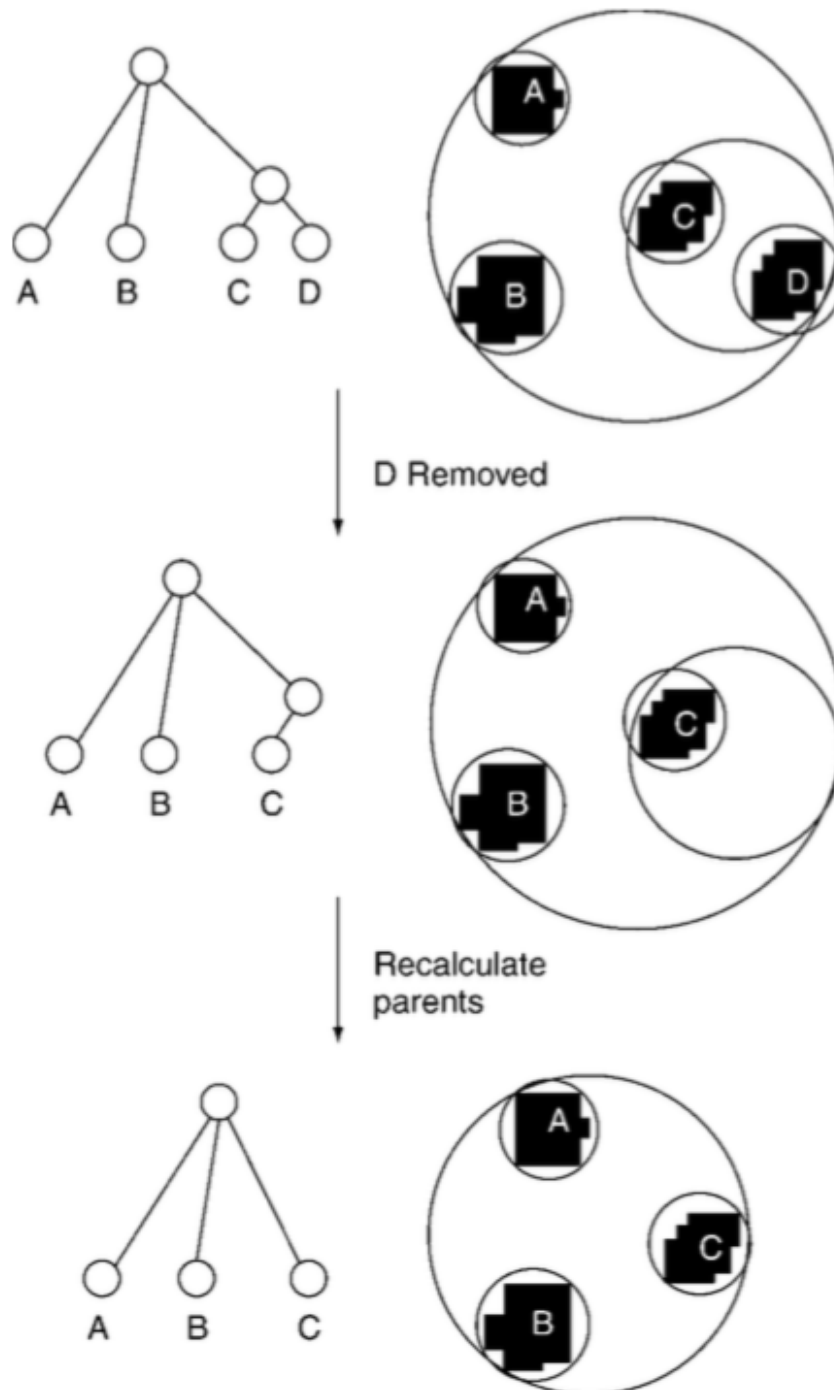


Рисунок 3.12 – Удаление узла

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.

4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

4.1 Класс Particle

Класс Particle представляет частицу. Частица - это самый простой объект, который можно моделировать в физической системе. Она имеет данные о местоположении (без ориентации), вместе со скоростью. Она может быть интегрирована вперед во времени и иметь приложенные к ней силы и импульсы. Частица управляет своим состоянием и предоставляет доступ через набор методов.

Есть два набора данных у частицы: характеристики и состояние.

Характеристики являются свойствами частицы, не зависящей от ее текущих кинематических свойств. Сюда входят масса, момент инерции и демпфирующие свойства. Две одинаковые частицы будут иметь одинаковые значения для своих характеристик.

Состояние включает в себя все характеристики, а также включает кинематические свойства частицы в текущем моделировании.

Когда данные состояния изменяются, производные значения необходимо обновлять: это может быть достигнуто либо путем интегрирования, либо путем вызова функции `calculateInternals`. Такой двухэтапный процесс используется, потому что пересчет внутренних элементов может быть дорогостоящим процессом. Все изменения состояния должны выполняться одновременно, что позволяет пересчитать их одним вызовом.

Класс содержит следующие поля:

1) `inverseMass`. Хранит обратную массу частицы. Полезнее хранить обратную массу, потому что интегрирования становятся проще, и в режиме реального времени более полезно иметь объекты с бесконечной массой, чем с нулевой массой (абсолютно неустойчивая при численном моделировании).

2) `damping`. Удерживает величину демпфирования, приложенного к линейному движению.

3) `position`. Хранит линейное положение частицы в мировом пространстве.

4) `velocity`. Хранит линейную скорость частицы в мировом пространстве.

5) `forceAccum`. Хранит накопленную силу, которая будет применяться только на следующей итерации моделирования. Это значение обнуляется на каждом этапе интеграции.

6) `acceleration`. Хранит ускорение частицы. Это значение может использоваться для установки ускорения из-за силы тяжести (его основная цель) или любого другого постоянного ускорения.

Класс Particle реализует следующие методы:

1) `integrate`. Интегрирует частицу вперед по времени на заданную

величину. Эта функция использует метод интегрирования Ньютона-Эйлера, который является линейным приближением к правильному интегралу. По этой причине могут возникать неточности в некоторых случаях.

2) `velocity`. Хранит линейную скорость частицы в мировом пространстве.

3) `setMass`. Устанавливает массу частицы. Метод делает недействительными внутренние данные частицы. Либо функцию интеграции, либо функцию `calculateInternals` следует вызывать перед попыткой получить какие-либо настройки частицы.

4) `setAcceleration`. Устанавливает постоянное ускорение частицы.

5) `setVelocity`. Устанавливает скорость частицы.

6) `clearAccumulator`. Убирает все приложенные силы к частице. Вызывается каждый раз после интегрирования.

7) `setMass`. Устанавливает массу частицы. Новая масса не может быть равна нулю, также очень маленькие значения могут вызвать странное поведение. Метод делает недействительными внутренние данные частицы. Либо функцию интеграции, либо функцию `calculateInternals` следует вызывать перед попыткой получить какие-либо настройки частицы.

8) `setPosition`. Устанавливает текущую мировую позицию частицы.

9) `setInverseMass`. Устанавливает обратную массу частицы. Метод делает недействительными внутренние данные частицы. Следует вызывать либо функцию интегрирования, либо функцию `calculateInternals` перед попыткой получить какие-либо настройки частицы.

Метод `integrate` выглядит следующим образом:

```
// We don't integrate things with zero mass.
if (inverseMass <= 0.0f)
    return;

assert(duration > 0.0);

// Update linear position.
position.addScaledVector(velocity, duration);

// Work out the acceleration from the force
Vector3 resultingAcc = acceleration;
resultingAcc.addScaledVector(forceAccum,
inverseMass);

// Update linear velocity from the acceleration.
velocity.addScaledVector(resultingAcc, duration);
// Impose drag.
velocity *= real_pow(damping, duration);

// Clear the forces.
```

```
clearAccumulator();
```

Сначала мы проверяем массу частицы, так как не имеет смысла интегрировать частицы с бесконечной массой. Затем мы добавляем к текущей позиции частицы ее скорость умноженную на время интегрирования. Далее нам необходимо пересчитать скорость для следующего интегрирования. К текущему ускорению мы добавляем результирующую силу, действующую на частицу, умноженную на обратную массу. Результат умножаем на время интегрирования и прибавляем к текущей скорости. Это и будет скорость частицы на следующем этапе интеграции.

4.2 Класс ParticleWorld

Основная задача класса это хранить частицы и предоставляет средства для их обновления.

Класс ParticleWorld реализует следующие методы:

1) ParticleWorld. Создает новый симулятор частиц, который может обрабатывать до заданного количества контактов на кадр. Также можно указать количество итераций для разрешения контактов. Если не указано количество итераций, будет использовано число в два раза большее количества контактов.

2) generateContacts. Позволяет каждому зарегистрированному контактному генератору сообщать о своих контактах. Возвращает количество сгенерированных контактов.

3) integrate. Интегрирует все частицы в этом мире вперед во времени на заданный период.

4) runPhysics. Обрабатывает всю физику для мира частиц.

5) startFrame. Инициализирует мир для симуляции кадра. Это очищает аккумуляторы силы для частиц в мире. После вызова этого метода, к частицам можно прикладывать силы.

```
unsigned limit = maxContacts;
ParticleContact *nextContact = contacts;

for(ContactGenerators::iterator
g = contactGenerators.begin();
    g != contactGenerators.end();
    g++)
{
    unsigned used = (*g)->addContact(nextContact, limit);
    limit -= used;
    nextContact += used;
    // We've run out of contacts to fill. This means we're missing
    // contacts.
    if (limit <= 0)
        break;
```

```

}

// Return the number of contacts used.
return maxContacts - limit;

```

Метод `generateContacts` перебирает все зарегистрированные генераторы контактов и через интерфейс класса `ParticleContactGenerator` вызывает у них метод `addContact`, пока не будет достигнут лимит контактов для фрейма или все генераторы не будут обработаны.

4.3 Класс `ParticleForceGenerator`

Этот класс предоставляет интерфейс для подсчета и обновления силы приложенной к частице. Также является базовым для всех генераторов силы, таких как `ParticleGravity`, `ParticleDrag`, `ParticleAnchoredSpring`, `ParticleAnchoredBungee`, `ParticleFakeSpring`, `ParticleSpring` и других. Его метод `updateForce` необходимо переопределить в производных классах.

4.4 Класс `ParticleGravity`

Силовой генератор, который прикладывает гравитационную силу. Один экземпляр может быть использован для нескольких частиц. Этот класс реализует интерфейс `ParticleForceGenerator`

Класс `ParticleGravity` реализует следующие методы:

- 1) `ParticleGravity`. Создает новый генератор с заданной силой тяжести.
- 2) `updateForce`. Прикладывает гравитационную силу к заданной частице.

Метод `updateForce` реализован следующим образом:

```

// Check that we do not have infinite mass
if (!particle->hasFiniteMass())
    return;

// Apply the mass-scaled force to the particle
particle->addForce(gravity * particle->getMass());

```

Сначала мы проверяем является ли конечной масса частицы(гравитация не будет воздействовать на частицы с бесконечной массой), а затем прикладываем к частице силу, равную произведению ускорения гравитации на массу частицы.

4.5 Класс ParticleContactGenerator

Это основной полиморфный интерфейс для контактных генераторов, применяющихся к частицам. Его метод `addContact` заполняет заданную контактную структуру `ParticleContact` созданным контактом. Указатель контакта должен указывать на первый доступный контакт в массиве контактов. Метод возвращает количество записей, которые были записаны.

4.6 Класс ParticleLink

Класс `ParticleLink` соединяет две частицы вместе, генерируя контакт, если они нарушают ограничения их связи. Этот класс используется в качестве базового класса для кабелей и стержней, и может использоваться в качестве базового класса для пружин с ограничением их расширения. Класс `ParticleLink` является наследником `ParticleContactGenerator` и реализует интерфейс контактных генераторов.

Класс `ParticleLink` реализует метод `addContact`. Генерирует контакты, чтобы эта связь частиц не нарушалась. Этот класс может генерировать только один контакт, поэтому указатель может быть указателем на один элемент, предельный параметр считается равным как минимум одному, а возвращаемое значение равно 0, если расстояние не было чрезмерно увеличенным, или 1, если контакт необходим.

4.7 Класс ParticleRod

Этот класс представляет стержень, связывающий две частицы. `ParticleRod` создает контакты, если частицы находятся слишком далеко, или слишком близко.

Класс содержит одно поле `length`, которое хранит длину, на которой необходимо держать частицы.

Класс реализует интерфейс `ParticleLink` и переопределяет его метод `addContact`. Этот метод заполняет структуру контакта, необходимую для предотвращения расхождения связанных частиц.

```
// Find the length of the rod
real currentLen = currentLength();

// Check if we're over-extended
if (currentLen == length)
{
    return 0;
}
// Otherwise return the contact
contact->particle[0] = particle[0];
contact->particle[1] = particle[1];
```

```

// Calculate the normal
Vector3 normal = particle[1]->getPosition() -
    particle[0]->getPosition();
normal.normalise();

// The contact normal depends on whether we're extending or
compressing
if (currentLen > length)
{
    contact->contactNormal = normal;
    contact->penetration = currentLen - length;
}
else
{
    contact->contactNormal = normal * -1;
    contact->penetration = length - currentLen;
}

// Always use zero restitution (no bounciness)
contact->restitution = 0;

return 1;

```

Сначала мы вычисляем расстояние между частицами и проверяем отличается ли оно от необходимой длины стержня, так как если нет, то контакт генерировать нет нужды. Далее мы вычисляем нормаль контакта в зависимости от того, нужно ли нам сблизить или растянуть частицы, и глубине проникновения. Коэффициент восстановления для таких контактов всегда будет равен нулю.

4.8 Класс ParticleCable

Кабели связывают пары частиц, создавая контакт, если они оказываются слишком далеко друг от друга.

Класс содержит следующие поля:

- 1) `maxLength`. Хранит максимальную длину кабеля.
- 2) `restitution`. Хранит коэффициент восстановления кабеля или его упругость.

Класс реализует интерфейс `ParticleLink` и переопределяет его метод `addContact`. Этот метод заполняет структуру контакта, необходимую для предотвращения растяжения кабеля.

```

real length = currentLength();
// Find the length of the cable
// Check if we're over-extended
if (length < maxLength)
{

```

```

        return 0;
    }

    // Otherwise return the contact
    contact->particle[0] = particle[0];
    contact->particle[1] = particle[1];

    // Calculate the normal
    Vector3 normal = particle[1]->getPosition() - particle[0]-
>getPosition();
    normal.normalise();
    contact->contactNormal = normal;

    contact->penetration = length-maxLength;
    contact->restitution = restitution;

    return 1;

```

Сначала мы вычисляем расстояние между частицами и проверяем превышает ли оно допустимую длину кабеля, так как если нет, то контакт генерировать нет нужды. Далее мы вычисляем нормаль контакта, равную нормализованной разнице между позициями частиц, и записываем ее вместе с коэффициентом восстановления в контактную структуру.

4.9 Класс ParticleConstraint

Класс ParticleConstraint выполняет ту же роль, что и ParticleLink, за исключением того, что он является базовым для генераторов контактов для соединений частиц с неподвижной точкой в пространстве.

Класс содержит следующие поля:

- 1) particle. Хранит частицу которую, для которой нужно применить ограничение.
- 2) anchor. Хранит координаты точки в пространстве для соединения с частицей.

Класс ParticleConstraint реализует метод addContact. Генерирует контакты, чтобы эта связь частиц не нарушалась. Этот класс может генерировать только один контакт, поэтому указатель может быть указателем на один элемент, предельный параметр считается равным как минимум одному, а возвращаемое значение равно 0, если расстояние не было чрезмерно увеличенным, или 1, если контакт необходим.

4.10 Класс ParticleCableConstraint

Класс представляет собой кабель, соединяющий частицу и точку в пространстве. Класс ParticleCableContraint является наследником

ParticleConstraint.

Класс содержит следующие поля:

- 1) `maxLength`. Хранит максимальную длину кабеля.
- 2) `restitution`. Хранит коэффициент восстановления кабеля или его упругость.

Класс реализует интерфейс `ParticleConstraint` и переопределяет его метод `addContact`. Этот метод заполняет структуру контакта, необходимую для предотвращения растяжения кабеля.

```
// Find the length of the cable
real length = currentLength();

// Check if we're over-extended
if (length < maxLength)
{
    return 0;
}

// Otherwise return the contact
contact->particle[0] = particle;
contact->particle[1] = 0;

// Calculate the normal
Vector3 normal = anchor - particle->getPosition();
normal.normalise();
contact->contactNormal = normal;

contact->penetration = length-maxLength;
contact->restitution = restitution;

return 1;
```

Сначала мы вычисляем расстояние между частицей и опорной точкой, и проверяем превышает ли оно допустимую длину кабеля, так как если нет, то контакт генерировать нет нужды. Далее мы вычисляем нормаль контакта, равную нормализованной разнице между позициями частиц, и записываем ее вместе с коэффициентом восстановления в контактную структуру.

4.11 Класс `ParticleRodConstraint`

Этот класс представляет стержень, связывающий частицу с неподвижной точкой в пространстве. `ParticleRodConstraint` создает контакты, если частицы находятся слишком далеко, или слишком близко.

Класс содержит одно поле `length`, которое хранит длину, на которой необходимо держать частицу от точки.

Класс реализует интерфейс `ParticleConstraint` и переопределяет

его метод `addContact`. Этот метод заполняет структуру контакта, необходимую для предотвращения расхождения связанных частиц.

```
// Find the length of the rod
real currentLen = currentLength();

// Check if we're over-extended
if (currentLen == length)
{
    return 0;
}
// Otherwise return the contact
contact->particle[0] = particle;
contact->particle[1] = 0;

// Calculate the normal
Vector3 normal = anchor - particle->getPosition();
normal.normalise();

// The contact normal depends on whether we're extending or
compressing
if (currentLen > length)
{
    contact->contactNormal = normal;
    contact->penetration = currentLen - length;
}
else
{
    contact->contactNormal = normal * -1;
    contact->penetration = length - currentLen;
}
// Always use zero restitution (no bounciness)
contact->restitution = 0;

return 1;
```

Сначала мы вычисляем расстояние между частицей и точкой, и проверяем отличается ли оно от необходимой длины стержня, так как если нет, то контакт генерировать нет нужды. Далее мы вычисляем нормаль контакта в зависимости от того, нужно ли нам сблизить или растянуть частицы, и глубине проникновения. Коэффициент восстановления для таких контактов всегда будет равен нулю.

5 ТЕСТИРОВАНИЕ И МЕТОДИКА ИСПЫТАНИЙ

Так как дипломный проект состоит из программных модулей, для того чтобы протестировать систему в целом, необходимо протестировать каждый из модулей её составляющих. Для тестирования модулей были написаны отдельные небольшие приложения покрывающие функционал каждого из модулей. Тестовые приложения используют тестируемые модули для симуляции каких-либо физических явлений и предоставляют графическое отображение и производимого моделирования. Всего было написано 5 приложений, каждое из которых проверяет один или несколько модулей:

- 1) Ballistics
- 2) Platform
- 3) Explosion
- 4) Bridge
- 5) Fracture

5.1 Ballistics

Это приложение проверяет в основном класс `Particle`. Данное приложение занимается созданием множества частиц с различными значениями начальной скорости, массы, демпфирования и ускорения. На рисунке 5.1 представлено работающее приложения, создающее двигающиеся частицы. Основной цикл программы постоянно вызывает метод `integrate` у созданных частиц, а после проверяет вышла ли она из зоны видимости, и если да, то удаляет ее.

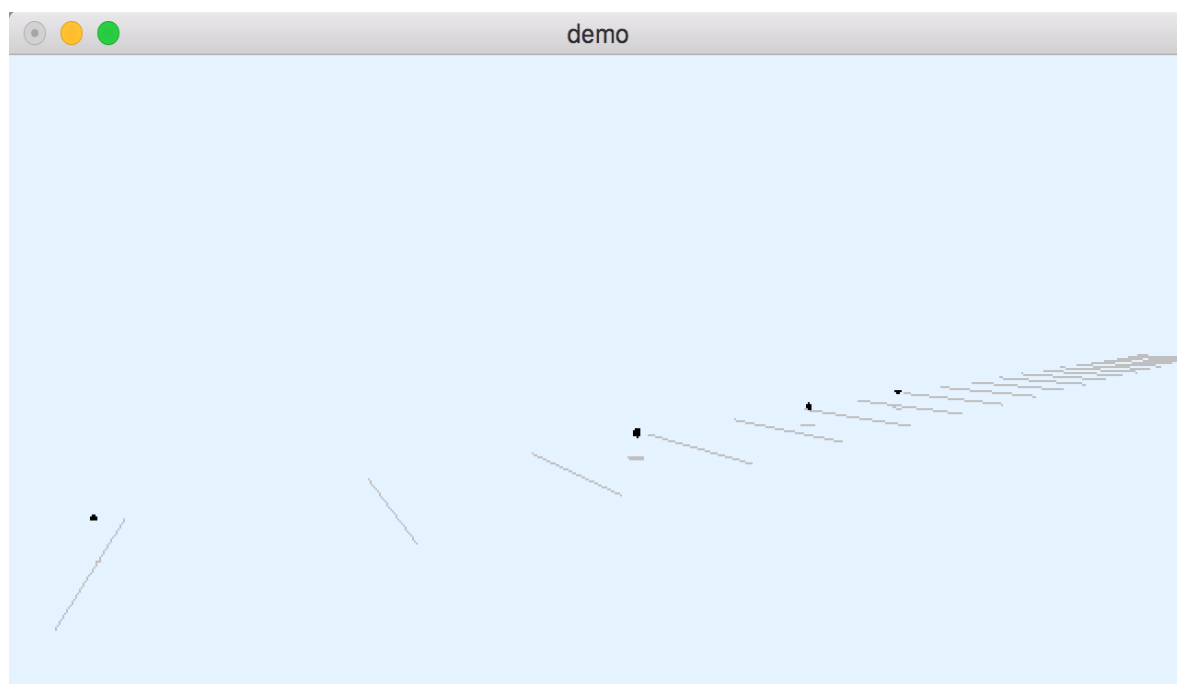


Рисунок 5.1 – Работающее приложение ballistics в режиме 1

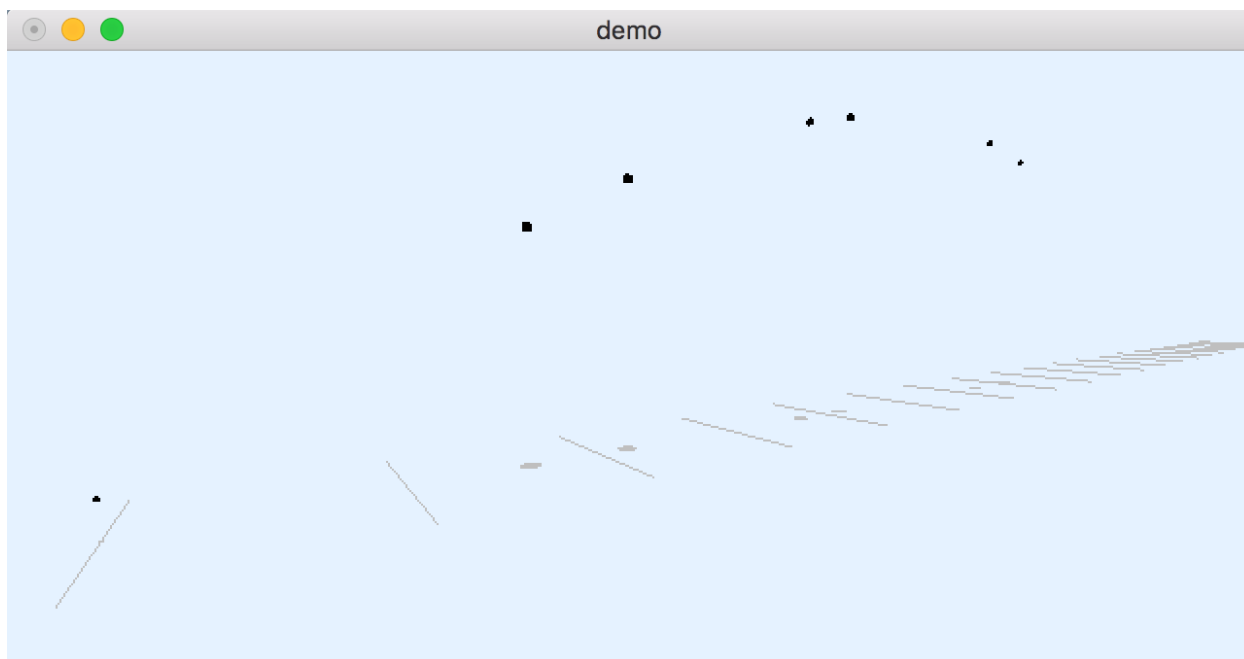


Рисунок 5.2 – Работающее приложение в режиме 2

Приложение может создавать несколько типов частиц, ведущих себя по разному. Так на рисунке 5.1 создаются быстро двигающиеся частицы с малой массой, летящие параллельно земле. В то время как на рисунке 5.2 можно наблюдать более тяжелые частицы, летящие по навесной траектории.

5.2 Platform

Данное приложение использует класс `Particle` вместе с `ParticleRod` для создания статичных конструкций из частиц. Приложение создает множество частиц и связывает их стержнями в виде тавровой платформы. Приложение также по нажатию определенных клавиш помещает дополнительную массу на определенные частицы в результате чего, центр массы всей платформы изменяется, и она теряет равновесие. Приложение также проверяет возможности класса `ParticleRod` удерживать вместе частицы и, следовательно, удерживать платформу в постоянной форме. На рисунке 5.3 показано работающее приложение. Красной точкой указывается дополнительный вес, прикладываемый на определенные частицы.

5.3 Bridge

Данное приложение схоже с предыдущим, оно создает подвесной мост из частиц `Particle`, связанных кабелями `ParticleCable`. Также как и в `platform` можно добавлять дополнительную массу для определенной части моста для изменения его состояния. При изменении позиции дополнительной массы, изменяются состояния частиц, и, следовательно, кабелей их

соединяющих – частицы начинают расходиться и кабели в ответ генерируют контакты для того, чтобы их свести обратно. На рисунке 5.4 показана работа приложения bridge.

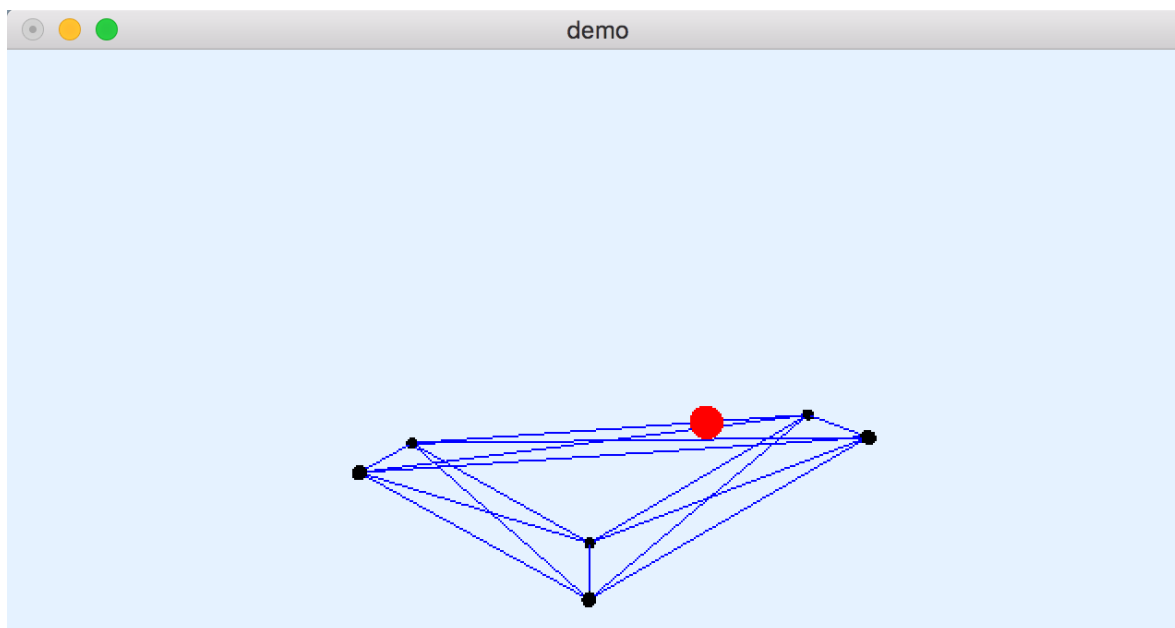


Рисунок 5.3 – Работающее приложение platform

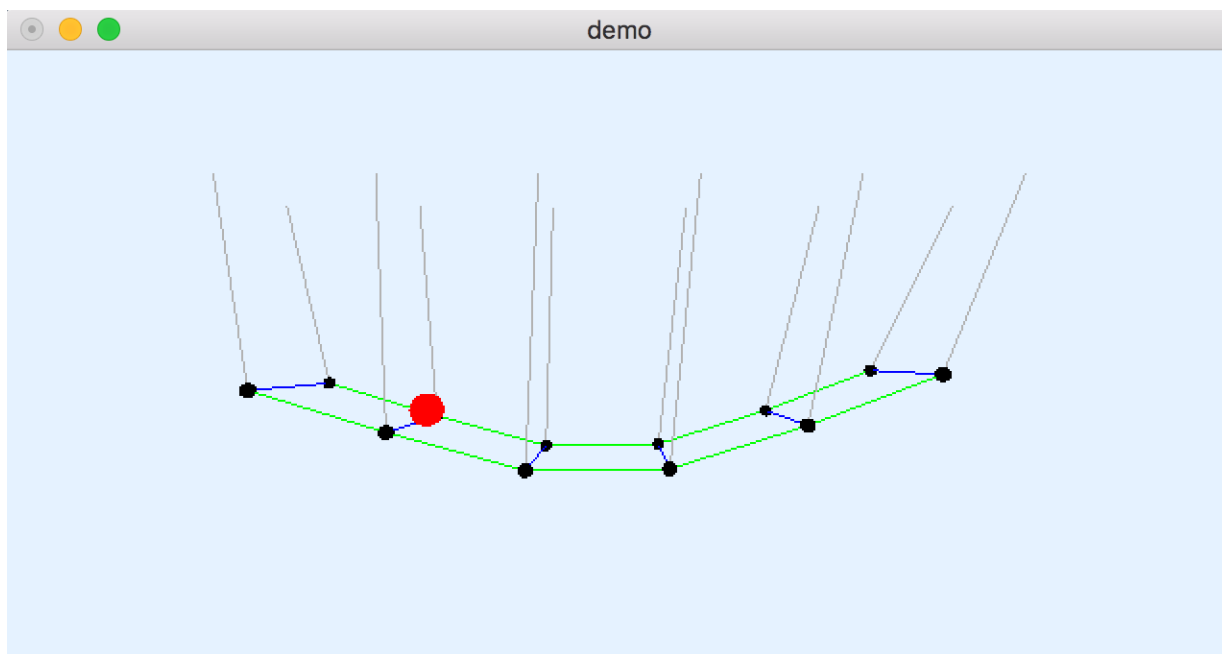


Рисунок 5.4 – Работающее приложение bridge

5.4 Explosion

Данное приложения отвечает за проверку классов обнаружения столкновений и из разрешения. В приложении создается множество твердых

тел кубической и сферической форм. Далее они все помещаются рядом друг с другом, и к ним прикладывается сила для того, чтобы произошло множество столкновений между объектами. Приложение тестирует работу классов `RigidBody`, `CollisionSphere`, `CollisionBox`, `IntersectionTest`, `CollisionDetector`. На рисунке 5.5 представлена работа приложения.

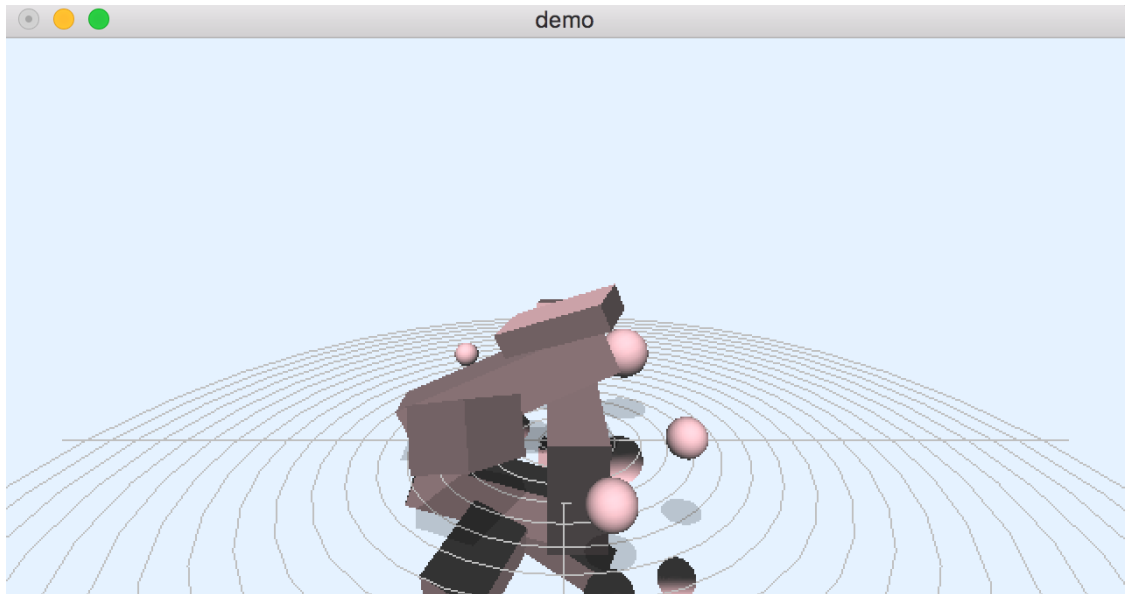


Рисунок 5.5 – Работающее приложение explosion

5.5 Fracture

Приложение `fracture` также как и `explosion`, проверяет систему обнаружения столкновений, классы `RigidBody`, `CollisionBox` и

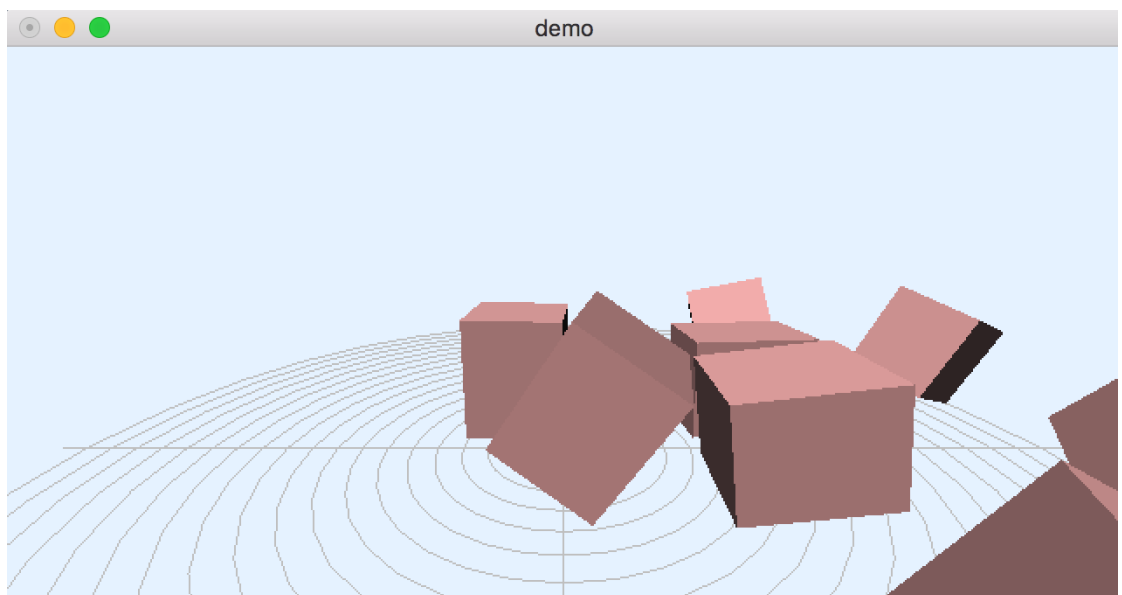


Рисунок 5.6 – Приложения `fracture` после столкновения

Particle. Цель приложения также является проверить возможности динамического создания и удаления объектов во время симуляции. Приложение создает кубическое тело и частицу, летящую в него. При столкновении твердое тело разделяется на множество более мелких. На рисунках 5.6 и 5.7 представлены ситуации до столкновения и после.

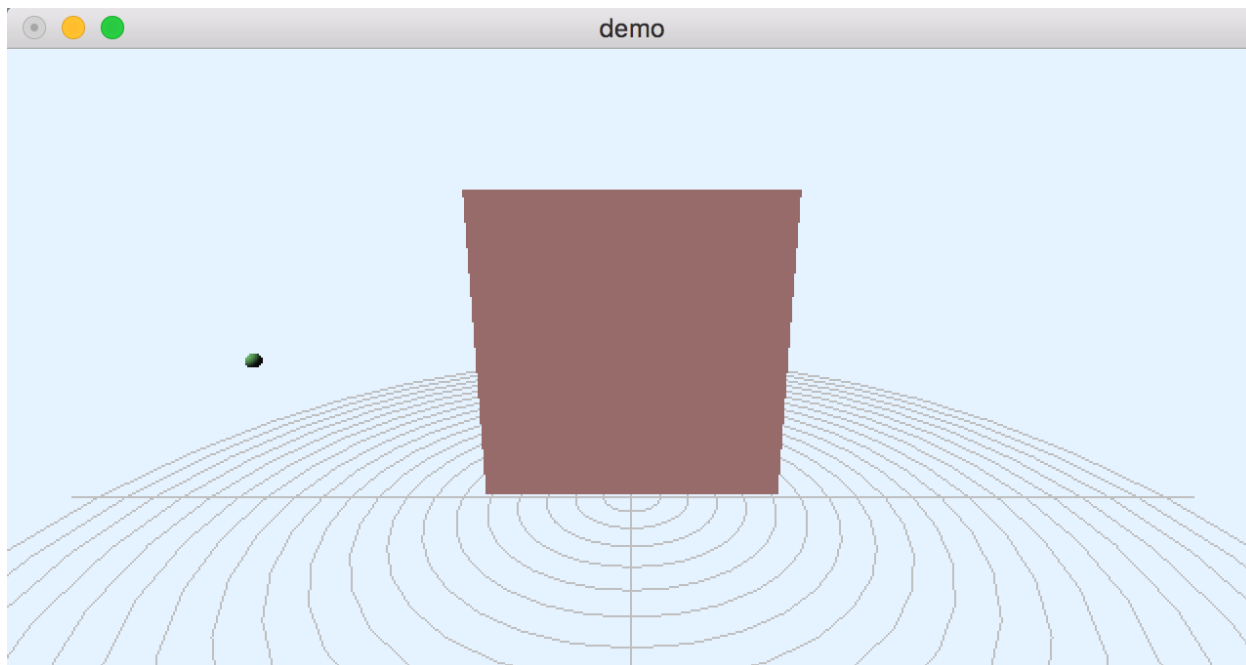


Рисунок 5.7 – Приложение fracture до столкновения

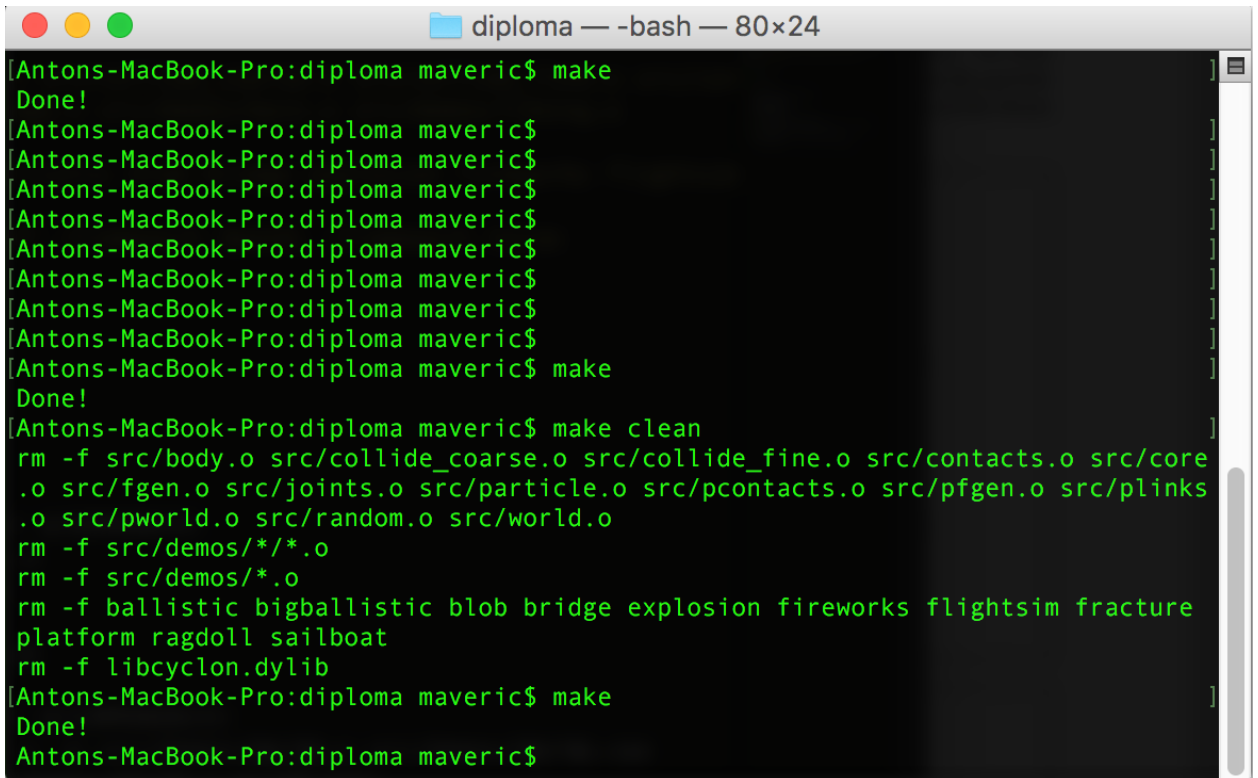
6 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

6.1 Минимальные системные требования

- Intel Core i5 3210M 2.5 ГГц;
- 4 ГБ ОЗУ;
- 48 КБ свободного места на диске;
- OS X 10.10 Yosemite.

6.2 Рекомендуемые системные требования

- Intel Core i5 Broadwell 2.7 ГГц;
- 8 ГБ ОЗУ;
- 123 КБ свободного места на диске;
- OS X 10.11 El Capitan.

A screenshot of a macOS terminal window titled "diploma — -bash — 80x24". The terminal shows a series of commands and their outputs. The user enters "make" and receives "Done!". Then, they enter "make clean", which results in a long list of files being removed, including "src/body.o", "src/collide_coarse.o", "src/collide_fine.o", "src/contacts.o", "src/core.o", "src/fgen.o", "src/joints.o", "src/particle.o", "src/pcontacts.o", "src/pfgen.o", "src/plinks.o", "src/pworld.o", "src/random.o", "src/world.o", "src/demos/*.o", and various executable files like "ballistic", "bigballistic", "blob", "bridge", "explosion", "fireworks", "flightsim", "fracture", "platform", "ragdoll", "sailboat", and "libcyclon.dylib". Finally, the user enters "make" again and receives "Done!". The prompt "Antons-MacBook-Pro:diploma maveric\$" is visible throughout.

```
[Antons-MacBook-Pro:diploma maveric$ make
Done!
[Antons-MacBook-Pro:diploma maveric$
[Antons-MacBook-Pro:diploma maveric$
[Antons-MacBook-Pro:diploma maveric$
[Antons-MacBook-Pro:diploma maveric$
[Antons-MacBook-Pro:diploma maveric$
[Antons-MacBook-Pro:diploma maveric$
[Antons-MacBook-Pro:diploma maveric$
[Antons-MacBook-Pro:diploma maveric$
[Antons-MacBook-Pro:diploma maveric$ make
Done!
[Antons-MacBook-Pro:diploma maveric$ make clean
rm -f src/body.o src/collide_coarse.o src/collide_fine.o src/contacts.o src/core
.o src/fgen.o src/joints.o src/particle.o src/pcontacts.o src/pfgen.o src/plinks
.o src/pworld.o src/random.o src/world.o
rm -f src/demos/*.o
rm -f src/demos/*.o
rm -f ballistic bigballistic blob bridge explosion fireworks flightsim fracture
platform ragdoll sailboat
rm -f libcyclon.dylib
[Antons-MacBook-Pro:diploma maveric$ make
Done!
Antons-MacBook-Pro:diploma maveric$
```

Рисунок 6.1 – Результат установки в терминальном окне

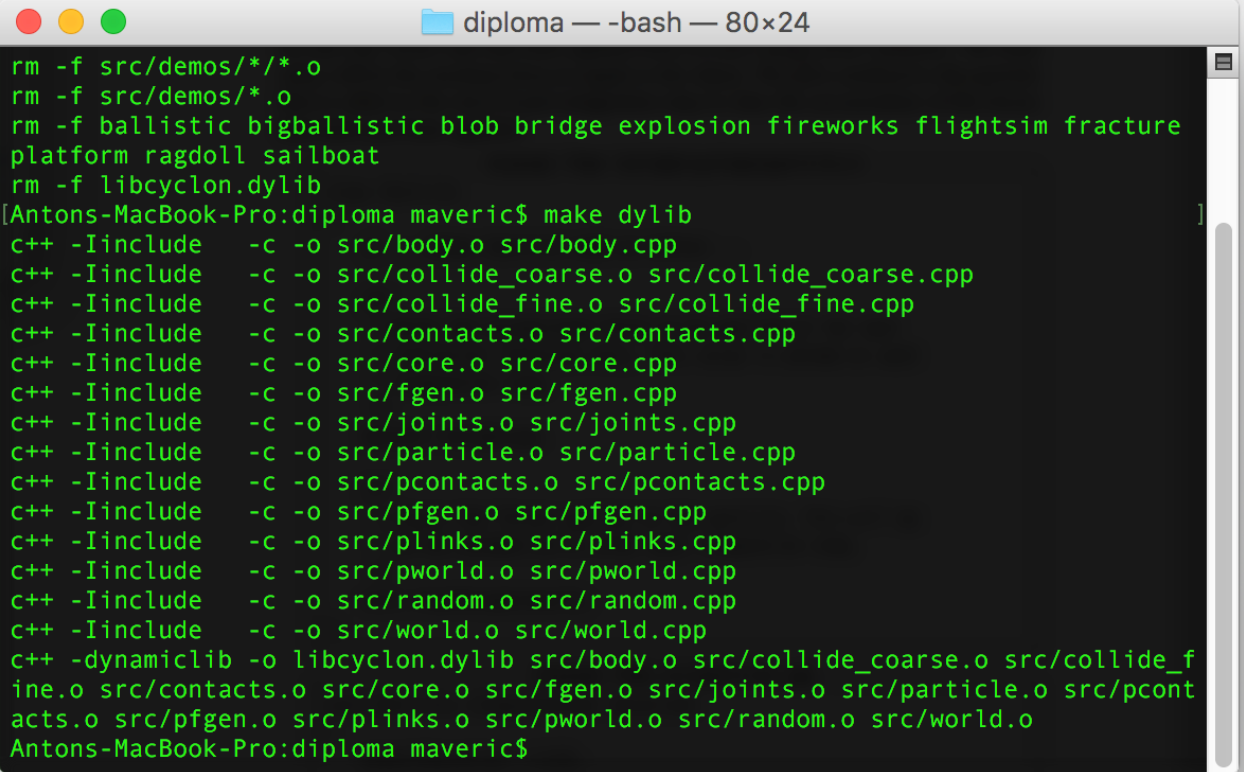
6.3 Инструкции по установке

Все дальнейшие инструкции по установке программы подразумевают, что на компьютере установлена версия операционной системы OS X не ниже Yosemite. Основным компонентом, необходимым для установки приложения является утилита make. К счастью, система OS X поставляется с

предустановленной утилитой, следовательно ее установка не требуется. Для установки необходимо выполнить следующие действия:

- 1) Перейти в директорию, в которой располагаются исходные коды приложения.
- 2) В терминале выполнить команду `make`.

В результате выполнения вышеописанных действий, пользователь увидит вывод программы, изображённый на рисунке 6.1.



```
rm -f src/demos/*/*.o
rm -f src/demos/*.o
rm -f ballistic bigballistic blob bridge explosion fireworks flightsim fracture
platform ragdoll sailboat
rm -f libcyclon.dylib
[Antons-MacBook-Pro:diploma maveric$ make dylib
c++ -Iinclude -c -o src/body.o src/body.cpp
c++ -Iinclude -c -o src/collide_coarse.o src/collide_coarse.cpp
c++ -Iinclude -c -o src/collide_fine.o src/collide_fine.cpp
c++ -Iinclude -c -o src/contacts.o src/contacts.cpp
c++ -Iinclude -c -o src/core.o src/core.cpp
c++ -Iinclude -c -o src/fgen.o src/fgen.cpp
c++ -Iinclude -c -o src/joints.o src/joints.cpp
c++ -Iinclude -c -o src/particle.o src/particle.cpp
c++ -Iinclude -c -o src/pcontacts.o src/pcontacts.cpp
c++ -Iinclude -c -o src/pfgen.o src/pfgen.cpp
c++ -Iinclude -c -o src/plinks.o src/plinks.cpp
c++ -Iinclude -c -o src/pworld.o src/pworld.cpp
c++ -Iinclude -c -o src/random.o src/random.cpp
c++ -Iinclude -c -o src/world.o src/world.cpp
c++ -dynamiclib -o libcyclon.dylib src/body.o src/collide_coarse.o src/collide_f
ine.o src/contacts.o src/core.o src/fgen.o src/joints.o src/particle.o src/pcont
acts.o src/pfgen.o src/plinks.o src/pworld.o src/random.o src/world.o
Antons-MacBook-Pro:diploma maveric$
```

Рисунок 6.2 – Результат создания динамической библиотеки в терминальном окне

6.4 Использование приложения

Разработанная программа представляется библиотекой в виде исходных кодов на языке C++. Возможны два варианта использования библиотеки. Первый состоит в том, чтобы вручную добавить в существующий проект все файлы с исходным кодом. Для такого способа не требуется никаких дополнительных действий. Второй состоит в том, чтобы использовать динамическую библиотеку. Так, для того чтобы получить динамическую библиотеку, необходимо выполнить следующую последовательность действий:

- 1) Перейти в директорию, в которой располагаются исходные коды приложения.
- 2) В терминале выполнить команду `make dylib`.

В результате выполнения вышеописанных действий, пользователь

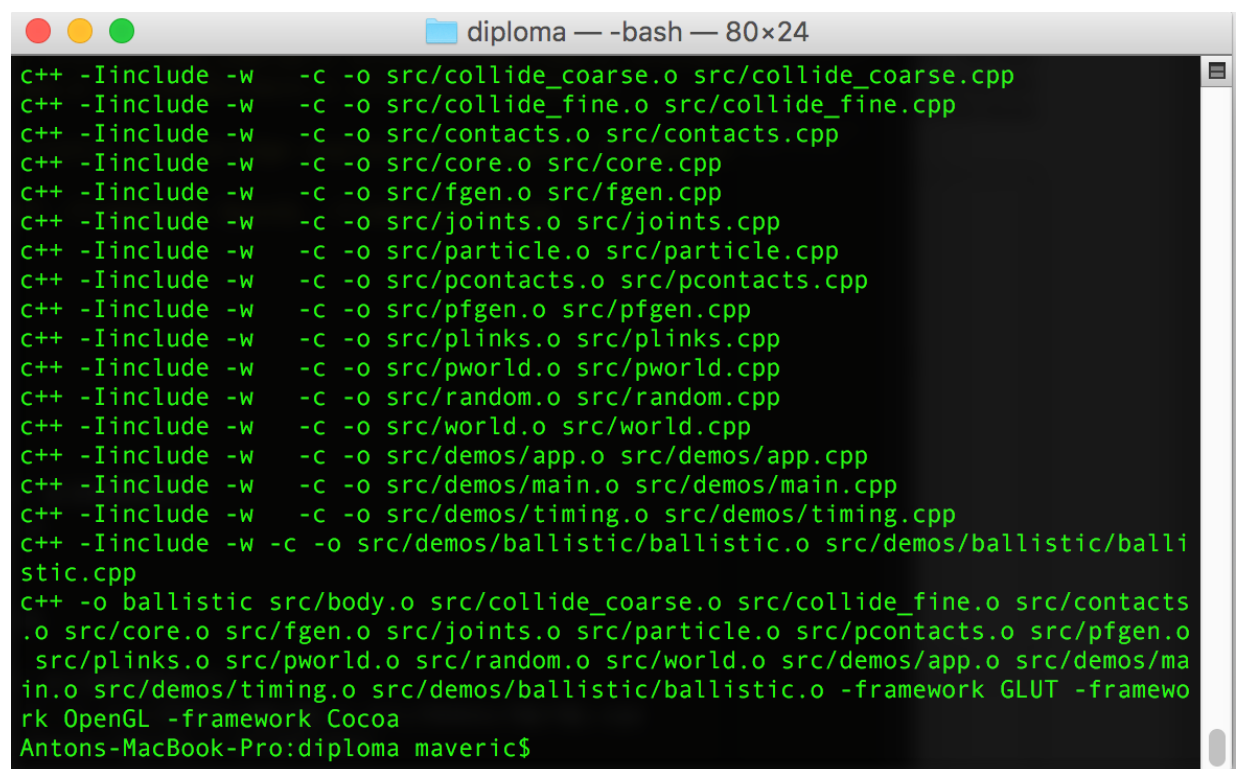
увидит вывод программы, изображённый на рисунке 6.2 и в корневой директории проекта будет создан файл динамической библиотеки.

Также вместе с библиотекой поставляются несколько небольших демонстрационных приложений, с помощью которых можно ознакомиться с основными функциями библиотеки.

Для того чтобы создать и запустить демонстрационные приложения, необходимо выполнить следующую последовательность действий:

- 1) Перейти в директорию, в которой располагаются исходные коды приложения.
- 2) В терминале выполнить команду `make demos`.
- 3) В терминале выполнить одну из следующих команд:
`./ballistic`, `./bigballistic`, `./blob`, `./bridge`,
`./explosion`, `./fireworks`, `./flightsim`, `./fracture`,
`./platform`, `./ragdoll`, `./sailboat`.

В результате выполнения вышеописанных действий, пользователь увидит вывод программы, изображённый на рисунке 6.3 и в корневой директории проекта будут созданы файлы с демонстрационными приложениями.



```
diploma — -bash — 80x24
c++ -Iinclude -w -c -o src/collide_coarse.o src/collide_coarse.cpp
c++ -Iinclude -w -c -o src/collide_fine.o src/collide_fine.cpp
c++ -Iinclude -w -c -o src/contacts.o src/contacts.cpp
c++ -Iinclude -w -c -o src/core.o src/core.cpp
c++ -Iinclude -w -c -o src/fgen.o src/fgen.cpp
c++ -Iinclude -w -c -o src/joints.o src/joints.cpp
c++ -Iinclude -w -c -o src/particle.o src/particle.cpp
c++ -Iinclude -w -c -o src/pcontacts.o src/pcontacts.cpp
c++ -Iinclude -w -c -o src/pfgen.o src/pfgen.cpp
c++ -Iinclude -w -c -o src/plinks.o src/plinks.cpp
c++ -Iinclude -w -c -o src/pworld.o src/pworld.cpp
c++ -Iinclude -w -c -o src/random.o src/random.cpp
c++ -Iinclude -w -c -o src/world.o src/world.cpp
c++ -Iinclude -w -c -o src/demos/app.o src/demos/app.cpp
c++ -Iinclude -w -c -o src/demos/main.o src/demos/main.cpp
c++ -Iinclude -w -c -o src/demos/timing.o src/demos/timing.cpp
c++ -Iinclude -w -c -o src/demos/ballistic/ballistic.o src/demos/ballistic/ballistic.cpp
c++ -o ballistic src/body.o src/collide_coarse.o src/collide_fine.o src/contacts.o src/core.o src/fgen.o src/joints.o src/particle.o src/pcontacts.o src/pfgen.o src/plinks.o src/pworld.o src/random.o src/world.o src/demos/app.o src/demos/main.o src/demos/timing.o src/demos/ballistic/ballistic.o -framework GLUT -framework OpenGL -framework Cocoa
Antons-MacBook-Pro:diploma maveric$
```

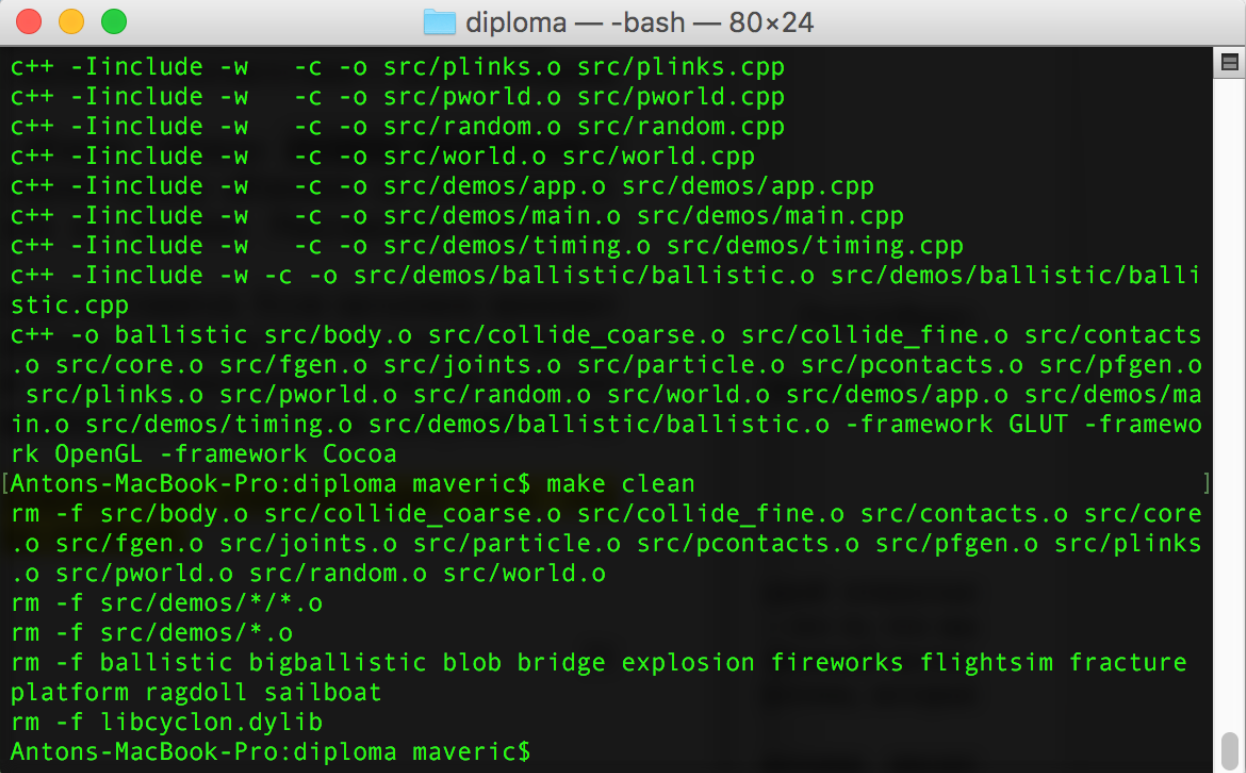
Рисунок 6.3 – Результат создания демонстрационных приложений

Для того чтобы очистить папку проекта от созданных файлов необходимо выполнить следующую последовательность действий:

- 1) Перейти в директорию, в которой располагаются исходные коды приложения.

2) В терминале выполнить команду `make clean`.

В результате выполнения вышеописанных действий, пользователь увидит вывод программы, изображённый на рисунке 6.4 и все созданные пользователем файлы будут удалены.



```
diploma — -bash — 80x24
c++ -Iinclude -w -c -o src/plinks.o src/plinks.cpp
c++ -Iinclude -w -c -o src/pworld.o src/pworld.cpp
c++ -Iinclude -w -c -o src/random.o src/random.cpp
c++ -Iinclude -w -c -o src/world.o src/world.cpp
c++ -Iinclude -w -c -o src/demos/app.o src/demos/app.cpp
c++ -Iinclude -w -c -o src/demos/main.o src/demos/main.cpp
c++ -Iinclude -w -c -o src/demos/timing.o src/demos/timing.cpp
c++ -Iinclude -w -c -o src/demos/ballistic/ballistic.o src/demos/ballistic/ballistic.cpp
c++ -o ballistic src/body.o src/collide_coarse.o src/collide_fine.o src/contacts.o src/core.o src/fgen.o src/joints.o src/particle.o src/pcontacts.o src/pfgen.o src/plinks.o src/pworld.o src/random.o src/world.o src/demos/app.o src/demos/main.o src/demos/timing.o src/demos/ballistic/ballistic.o -framework GLUT -framework OpenGL -framework Cocoa
[Antons-MacBook-Pro:diploma maveric$ make clean]
rm -f src/body.o src/collide_coarse.o src/collide_fine.o src/contacts.o src/core.o src/fgen.o src/joints.o src/particle.o src/pcontacts.o src/pfgen.o src/plinks.o src/pworld.o src/random.o src/world.o
rm -f src/demos/*/*.o
rm -f src/demos/*.o
rm -f ballistic bigballistic blob bridge explosion fireworks flightsim fracture platform ragdoll sailboat
rm -f libcyclon.dylib
Antons-MacBook-Pro:diploma maveric$
```

Рисунок 6.4 – Результат удаления созданных файлов

7 ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ ПРОГРАММНОГО СРЕДСТВА КОМПЬЮТЕРНОГО МОДЕЛИРОВАНИЯ ФИЗИЧЕСКИХ ЗАКОНОВ РЕАЛЬНОГО МИРА В ВИРТУАЛЬНОЙ СРЕДЕ

7.1 Описание функций, назначения и потенциальных пользователей ПО

В рамках дипломного проекта было разработано приложение для моделирования реалистического поведения тел путем ограниченной симуляции некоторых основных физических законов.

Программный продукт относится к программному обеспечению функционального назначения, предназначено для использования широким кругом пользователей и свободной продажи на рынке ИТ.

Актуальность разработки данного ПО у потенциальных пользователей основана на отсутствии большого числа аналогов и наличии значительного спроса среди потребителей.

7.2 Расчёт затрат на разработку ПО

Для осуществления упрощённого расчёта затрат на разработку ПО следует произвести расчёт следующих статей:

- затраты на основную заработную плату разработчиков;
- затраты на дополнительную заработную плату разработчиков;
- отчисления на социальные нужды;
- прочие затраты (амортизация оборудования, расходы на электроэнергию, командировочные расходы, накладные расходы и т.п.).

Расчёт затрат на основную заработную плату разработчиков осуществляется на численности и состава команды, размеров месячной заработной платы каждого из участников команды, а также общей трудоёмкости разработки программного обеспечения.

Основная заработная плата исполнителей на конкретное программное средство определяется по формуле:

$$Z_o = \sum_{i=1}^n T_{\text{ч}_i} * t_i, \quad (7.1)$$

где n – количество исполнителей, занятых разработкой конкретного ПО;

$T_{\text{ч}_i}$ – часовая заработная плата i -го исполнителя (руб.);

t_i – трудоёмкость работ, выполняемых i -м исполнителем (ч).

В проекте занято два человека: руководитель и разработчик.

Часовая заработная плата определяется на основе месячной заработной платы путём деления на количество рабочих часов в месяце (примем 168

часов). Для руководителя, при заработной плате равной 1500 рублей, часовая заработная плата равна 8,93 рубля. Для разработчика, при заработной плате равно 1000 рублей, часовая заработная плата равна 5,95 рубля.

Трудоемкость определяется исходя из сложности разработки программного продукта и объема выполняемых им функций. В нашем случае она составляет 40 дней или 320 часов.

Тогда основная зарплата исполнителей равна:

$$З_o = (8,93 + 5,95) * 320 = 4761,6 \text{ (руб)} \quad (7.2)$$

Таблица 7.1 – Расчет затрат на основную заработную плату команды

№	Участник команды	Выполняемые работы	Месячная заработная плата, р.	Часовая заработная плата, р.	Трудоёмкость работ, часов	Основная заработная плата, р.
1	Руководитель проекта	Контроль, помощь	1500	8,93	320	2857,6
2	Программист 1-й категории	Разработка	1000	5,95	320	1904
ПРЕМИЯ (50%)						2380,8
Итого затраты на основную заработную плату разработчиков						7142,4

Затраты на дополнительную заработную плату команды разработчиков включает выплаты, предусмотренные законодательством о труде (оплата отпусков, льготных часов, времени выполнения государственных обязанностей и других выплат, не связанных с основной деятельностью исполнителей), и определяется по формуле:

$$З_d = \frac{З_o * Н_d}{100}, \quad (7.3)$$

где $З_o$ – затраты на основную заработную плату с учетом премии (руб.);

$Н_d$ – норматив дополнительной заработной платы, 15%.

В результате подстановки получим:

$$З_d = \frac{7142,4 * 15}{100} = 1071,36 \text{ (руб)} \quad (7.4)$$

Отчисления на социальные нужды (в фонд социальной защиты населения и на обязательное страхование) определяются в соответствии с действующими законодательными актами по формуле:

$$З_{соц} = \frac{(З_o + З_d) * Н_{соц}}{100}, \quad (7.5)$$

где $N_{\text{соц}}$ – норматив отчислений на социальные нужды (34%) и обязательное страхование (0,6%).

$$Z_{\text{соц}} = \frac{(7142,4 + 1071,36) * 34,6}{100} = 2841,96 \text{ (руб)} \quad (7.6)$$

Расчет прочих затрат осуществляется в процентах от затрат на основную заработную плату команды разработчиков с учетом премии (табл.7.1) по формуле:

$$Z_{\text{пз}} = \frac{Z_o * N_{\text{пз}}}{100}, \quad (7.7)$$

где $N_{\text{пз}}$ – норматив прочих затрат, 100%.

$$Z_{\text{пз}} = \frac{7142,4 * 100}{100} = 7142,4 \text{ (руб)} \quad (7.8)$$

Полная сумма затрат на разработку программного обеспечения находится путем суммирования всех рассчитанных статей затрат (см. таблицу 7.2).

Таблица 7.2 – Затраты на разработку программного обеспечения

Статья затрат	Сумма, руб.
Основная заработная плата команды разработчиков	7142,4
Дополнительная заработная плата команды разработчиков	1071,36
Отчисления на социальные нужды	2841,96
Прочие затраты	7142,4
Общая сумма затрат на разработку	18198,12

7.3 Оценка результата от продажи ПО

Экономический эффект представляет собой прирост чистой прибыли, полученный организацией в результате использования разработанного ПО. Как правило, он может быть достигнут за счет:

- уменьшения (экономии) затрат на заработную плату за счет замены «ручных» операций и бизнес-процессов информационной системой;
- ускорения скорости обслуживания клиентов и рост возможности обслуживания большего их количества в единицу времени, т.е. рост производительности труда;
- появления нового канала сбыта продукции или получения заказов (как в случае внедрения интернет-магазина);
- и т.п.

Экономический эффект организации-разработчика программного обеспечения в данном случае заключается в получении прибыли от его продажи множеству потребителей. Прибыль от реализации в данном случае напрямую зависит от объемов продаж, цены реализации и затрат на разработку данного ПО.

Таким образом, необходимо сделать обоснование предполагаемого объема продаж – количества копий (лицензий) программного обеспечения, которое будет куплено клиентами за год (N). Принимая во внимания примерное количество скачиваний аналогичных приложений в год (60 тыс. человек) и учитывая различия в количестве функций, можно спрогнозировать 20 тыс. скачиваний за год.

Далее следует определить цену на одну копию (лицензию) ПО. Цена формируется на рынке под воздействием спроса и предложения. Тогда расчет прибыли от продажи одной копии (лицензии) ПО осуществляется по формуле:

$$П_{ед} = Ц - НДС - \frac{З_p}{N}, \quad (7.9)$$

где Ц – цена реализации одной копии (лицензии) ПО (руб.);

$З_p$ – сумма расходов на разработку и реализацию (руб.);

N – количество копий (лицензий) ПО, которое будет куплено клиентами за год;

$П_{ед}$ – прибыль, получаемая организацией-разработчиком от реализации одной копии программного продукта (руб.);

НДС – сумма налога на добавленную стоимость (руб.).

Цена одной копии (лицензии) программного обеспечения выбирается на основе цены на аналогичное программное обеспечение на рынке. Для приложений данного направления, средняя стоимость составляет 2 руб.

Сумма налога на добавленную стоимость рассчитывается по формуле:

$$НДС = \frac{Ц * \%НДС}{100 + \%НДС}, \quad (7.10)$$

где НДС – ставка налога на добавленную стоимость, (20%).

$$НДС = \frac{2 * 20}{100 + 20} = 0,33 \text{ (руб)} \quad (7.11)$$

Подставив значения в формулу 7.9, получим:

$$П_{ед} = 2 - 0,33 - \frac{18198,12}{20000} = 0,76 \text{ (руб)} \quad (7.12)$$

Суммарная годовая прибыль по проекту в целом будет равна:

$$\Pi = \Pi_{\text{ед}} * N \quad (7.13)$$

$$\Pi = 0,76 * 20000 = 15200 \text{ (руб)} \quad (7.14)$$

Рентабельность затрат на разработку ПО рассчитывается по следующей формуле:

$$P = \frac{\Pi}{Z_p} * 100\% \quad (7.15)$$

$$P = \frac{15200}{18198,12} * 100\% = 83,5\% \quad (7.16)$$

Проект является экономически эффективным, т.к. рентабельность затрат на разработку программного обеспечения превышает среднюю процентную ставку по банковским депозитным вкладам.

Чистая прибыль рассчитывается по формуле:

$$\text{ЧП} = \Pi - \frac{\Pi * N_{\text{приб}}}{100}, \quad (7.17)$$

где $N_{\text{приб}}$ – ставка налога на прибыль, (18%).

$$\text{ЧП} = 15200 - \frac{15200 * 18}{100} = 12464 \text{ (руб)} \quad (7.18)$$

7.4 Расчёт показателей эффективности инвестиций в разработку ПО

Перед тем, как произвести расчёт показателей эффективности инвестиций в разработку ПО, необходимо сравнить размер инвестиций в разработку и получаемый годовой экономический эффект. Т.к. сумма инвестиций больше, чем сумма годового эффекта, инвестиции окупятся более чем за год. В таком случае экономическая целесообразность инвестиций в разработку и использование программного продукта осуществляется на основе расчета и оценки следующих показателей:

- чистый дисконтированный доход ЧДД);
- срок окупаемости инвестиций ($T_{\text{ок}}$);
- рентабельность инвестиций ($P_{\text{и}}$).

Так как приходится сравнивать разновременные результаты (экономический эффект) и затраты (инвестиции в разработку программного продукта), необходимо привести их к единому моменту времени – началу расчетного периода, что обеспечивает их сопоставимость.

Для этого необходимо использовать дисконтирование путем умножения соответствующих результатов и затрат на коэффициент дисконтирования соответствующего года t , который определяется по формуле:

$$\alpha = \frac{1}{(1 + E_n)^t}, \quad (7.19)$$

где E_n – норма дисконта (в долях единиц), равная или больше средней процентной ставки по банковским депозитам действующей на момент осуществления расчетов;

t – порядковый номер года периода реализации инвестиционного проекта (предполагаемый период использования разрабатываемого ПО пользователем и время на разработку).

Предполагаемый период использования разрабатываемого программного обеспечения пользователем возьмем на уровне среднего периода использования схожего программного обеспечения – 3 года.

Чистый дисконтированный доход рассчитывается по формуле:

$$\text{ЧДД} = \sum_{t=0}^n (P_t * \alpha_t - Z_t * \alpha_t), \quad (7.20)$$

где, n – расчетный период, лет;

P_t – результат (экономический эффект), полученный в году t , руб.;

Z_t – затраты (инвестиции в разработку программного обеспечения), полученный в году t , руб.

Срок окупаемости проекта, т.е. момент, когда суммарный дисконтированный результат (эффект) станет равным (превысит) дисконтированную сумму инвестиций. Т.е. определяется через какой период времени инвестиционный проект начнет приносить инвестору прибыль.

Рентабельность инвестиций ($P_{\text{и}}$) рассчитывается как отношение суммы дисконтированных результатов (эффектов) к осуществленным инвестициям:

$$P_{\text{и}} = \frac{\sum_{t=0}^n P_t * \alpha_t}{\sum_{t=0}^n Z_t * \alpha_t} \quad (7.21)$$

Таблица 7.3 – Расчет эффективности инвестиционного проекта по разработке программного обеспечения

Показатель	Расчетный период		
	2	3	4
Результат			
1. Экономический эффект	12464	12464	12464

Продолжение таблицы 7.3

1	2	3	4
2. Дисконтированный результат	10719,04	9098,72	7727,68
Затраты			
3. Инвестиции в разработку программного средства	18198,12	0	0
4. Дисконтированные инвестиции	15650,38	0	0
5. Чистый дисконтированный доход по годам	-4931,34	9098,72	7727,68
6. Чистый дисконтированный доход нарастающим итогом	-4931,34	4167,38	11895,1
Коэффициент дисконтирования	0,86	0,73	0,62

Из расчетов видно, что срок окупаемости проекта ($T_{ок}$) будет равен 2 годам. Среднегодовая рентабельность инвестиций составит

$$P_{и} = \left(\frac{27545,44}{15650,38} * 100\% \right) / 3 = 58.67\% \quad (7.22)$$

Рассчитав все параметры, можно сделать вывод о том, что данный программный продукт будет являться одним из лучших на рынке и инвестиции в данный проект являются целесообразными. Он сможет составить конкуренцию другим аналогичным продуктам по причине наличия уникальных характеристик и возможностей, которые он предоставляет пользователям.

ЗАКЛЮЧЕНИЕ

Данный проект демонстрирует создание физического движка и испытывает его в различных ситуациях. Как уже упоминалось с самого начала, подход, который был применен является хорошим сочетанием простоты в управлении и симуляции. В конечном счете, однако, любой подход имеет свои ограничения.

Большие стеки объектов могут быть не слишком стабильными в данном движке. Разумеется, можно установить объекты в нужное положение и заморозить их, и они упадут при столкновении, но легкое прикосновение, скорее всего, заставит их вибрировать. В худшем случае это может привести к тому, что блоки в верхней части стека будут заметно перемещаться и вибрировать по краям.

Это вызвано итерационным алгоритмом разрешения столкновений. Алгоритм не идеально позиционирует объекты после разрешения столкновения. Для одного объекта (или даже небольшого количества сложенных объектов) это не проблема. Для больших стеков погрешности будут накапливаться до тех пор, пока не станут очень заметными.

Разумным решением будет помещать объекты в сон, таким образом стеки будут казаться стабильными, и это не будет выявлять ограничений движка.

Еще один недостаток заключается в том, что трение силы реакции, учитывается, когда контакт разрешается, но не когда контакт происходит как побочный эффект другого разрешения. Это мешает одному движимому объекту, прислоненному к другому, оставаться на месте. Похоже, что объекты будут скользить друг от друга, независимо от введенных трений.

Это еще один побочный эффект алгоритма разрешения столкновений: он не учитывает трения одного контакта при разрешении другого.

Те же накапливающиеся погрешности, которые приводят к неустойчивости стеков, также могут приводить к заметным артефактам, когда множество твердых тел соединены в линию. В дополнение к тому, что много соединений обуславливает нагрузку на программу, разработанный движок рассматривает каждое соединение последовательно. Соединения на одном конце цепи могут быть сильно затронуты изменениями на другом конце.

Итеративное разрешение не является наилучшим вариантом для сильно ограниченных наборов твердых тел (хотя оно и может справиться с небольшими группами).

Наконец жесткие пружины являются такой же проблемой для как твердых тел, так и для частиц, по той же причине. Хотя есть возможность использовать генераторы заменяющих сил, проблема не может быть полностью решена.

Почти все, что можно сделать с помощью любого существующего физического движка, можно сделать с помощью движка, который был создан в рамках дипломного проекта.

Но физический движок не идеален. Была построена очень быстрая и оптимизированная система, но пришлось пожертвовать некоторой точностью вычислений, особенно когда дело касается разрешения столкновений.

В целом, учитывая всё вышесказанное, можно заключить, что в результате проделанной работы было разработано законченное целостное приложение, способное моделировать физическое поведение различных объектов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- [1] Суслов, Г. К. Теоретическая механика / Г. К. Суслов. - М. : Гостехиздат, 1946. - 180 с.
- [2] Четаев, Н. Г. Теоретическая механика / Н. Г. Четаев. - М. : Наука, 1987. - 92 с.
- [3] Павловский, М. А. Теоретическая механика. Статика. Кинематика / М. А. Павловский, Л. Ю. Акинфиева, О. Ф. Бойчук. - Киев: Вища школа, 1989. - 351 с.
- [4] Gregory, J. Game Engine Development / J. Gregory. - Boca Raton: CRC Press, 2015 - 690 с.
- [5] Kodicek, D. Mathematics and Physics for Game Programmers / D. Kodicek. - Hingham: Charles River Media, 2005. - С. 404 - 415.
- [6] Lengyel, E. Mathematics for 3D Game Programming and Computer Graphics, Second Edition / E. Lengyel. - Hingham: Charles River Media, 2003. - С. 187 - 213.
- [7] Eberly, D. H. Game Physics / D. H. Eberly. - San Francisco: Morgan Kaufmann, 2003. - С. 93 - 96.
- [8] Ericson, C. Real-Time Collision Detection / Christer Ericson. - San Francisco: Morgan Kaufmann, 2005. - С. 56 - 62.

ПРИЛОЖЕНИЕ А
(обязательное)
Код программы. Модуль pworld

```
#ifndef CYCLONE_PWORLD_H
#define CYCLONE_PWORLD_H

#include "pfggen.h"
#include "plinks.h"

namespace cyclone {

    class ParticleWorld
    {
    public:
        typedef std::vector<Particle*> Particles;
        typedef std::vector<ParticleContactGenerator*>
ContactGenerators;

    protected:
        Particles particles;

        bool calculateIterations;

        ParticleForceRegistry registry;

        ParticleContactResolver resolver;

        ContactGenerators contactGenerators;
        ParticleContact *contacts;

        unsigned maxContacts;

    public:

        ParticleWorld(unsigned maxContacts, unsigned
iterations=0);
        ~ParticleWorld();
        unsigned generateContacts();
        void integrate(real duration);
        void runPhysics(real duration);
        void startFrame();
        Particles& getParticles();

        ContactGenerators& getContactGenerators();

        ParticleForceRegistry& getForceRegistry();
    };
};
```

```

        class          GroundContacts          :          public
cyclone::ParticleContactGenerator
    {
        cyclone::ParticleWorld::Particles *particles;

        public:
            void          init(cyclone::ParticleWorld::Particles
*particles);

            virtual unsigned addContact(cyclone::ParticleContact
*contact,
                unsigned limit) const;
    };

} // namespace cyclone

#endif // CYCLONE_PWORLD_H

#include <cstdint>
#include <cyclone/pworld.h>

using namespace cyclone;

ParticleWorld::ParticleWorld(unsigned maxContacts, unsigned
iterations)
:
resolver(iterations),
maxContacts(maxContacts)
{
    contacts = new ParticleContact[maxContacts];
    calculateIterations = (iterations == 0);
}

ParticleWorld::~~ParticleWorld()
{
    delete[] contacts;
}

void ParticleWorld::startFrame()
{
    for (Particles::iterator p = particles.begin();
        p != particles.end();
        p++)
    {
        // Remove all forces from the accumulator
        (*p)->clearAccumulator();
    }
}

unsigned ParticleWorld::generateContacts()

```

```

    {
        unsigned limit = maxContacts;
        ParticleContact *nextContact = contacts;

        for (ContactGenerators::iterator g =
contactGenerators.begin();
            g != contactGenerators.end();
            g++)
        {
            unsigned used = (*g)->addContact(nextContact, limit);
            limit -= used;
            nextContact += used;

            // We've run out of contacts to fill. This means we're
missing
            // contacts.
            if (limit <= 0) break;
        }

        // Return the number of contacts used.
        return maxContacts - limit;
    }

void ParticleWorld::integrate(real duration)
{
    for (Particles::iterator p = particles.begin();
        p != particles.end();
        p++)
    {
        // Remove all forces from the accumulator
        (*p)->integrate(duration);
    }
}

void ParticleWorld::runPhysics(real duration)
{
    // First apply the force generators
    registry.updateForces(duration);

    // Then integrate the objects
    integrate(duration);

    // Generate contacts
    unsigned usedContacts = generateContacts();

    // And process them
    if (usedContacts)
    {
        if (calculateIterations)
resolver.setIterations(usedContacts * 2);
        resolver.resolveContacts(contacts, usedContacts,
duration);
    }
}

```

```

    }
}

ParticleWorld::Particles& ParticleWorld::getParticles()
{
    return particles;
}

ParticleWorld::ContactGenerators&
ParticleWorld::getContactGenerators()
{
    return contactGenerators;
}

ParticleForceRegistry& ParticleWorld::getForceRegistry()
{
    return registry;
}

void GroundContacts::init(cyclone::ParticleWorld::Particles
*particles)
{
    GroundContacts::particles = particles;
}

unsigned GroundContacts::addContact(cyclone::ParticleContact
*contact,
                                   unsigned limit) const
{
    unsigned count = 0;
    for (cyclone::ParticleWorld::Particles::iterator p =
particles->begin();
        p != particles->end();
        p++)
    {
        cyclone::real y = (*p)->getPosition().y;
        if (y < 0.0f)
        {
            contact->contactNormal = cyclone::Vector3::UP;
            contact->particle[0] = *p;
            contact->particle[1] = NULL;
            contact->penetration = -y;
            contact->restitution = 0.2f;
            contact++;
            count++;
        }

        if (count >= limit) return count;
    }
    return count;
}

```

ПРИЛОЖЕНИЕ Б
(обязательное)
Код программы. Модуль plinks

```
#ifndef CYCLONE_PLINKS_H
#define CYCLONE_PLINKS_H

#include "pcontacts.h"

namespace cyclone {

    class ParticleLink : public ParticleContactGenerator
    {
    public:
        Particle* particle[2];

    protected:
        real currentLength() const;

    public:
        virtual unsigned addContact(ParticleContact
*contact,
                                unsigned limit) const =
0;
    };

    /**
    * Cables link a pair of particles, generating a contact
if they
    * stray too far apart.
    */
    class ParticleCable : public ParticleLink
    {
    public:
        /**
        * Holds the maximum length of the cable.
        */
        real maxLength;

        /**
        * Holds the restitution (bounciness) of the cable.
        */
        real restitution;

    public:
        /**
        * Fills the given contact structure with the contact
needed
        * to keep the cable from over-extending.
        */
    };
};

}
```



```

        virtual      unsigned      addContact(ParticleContact
*contact,
                                unsigned limit) const;
    };

    /**
    * Rods link a pair of particles, generating a contact if
they
    * stray too far apart or too close.
    */
    class ParticleRod : public ParticleLink
    {
    public:
        /**
        * Holds the length of the rod.
        */
        real length;

    public:
        /**
needed
        * Fills the given contact structure with the contact
        * to keep the rod from extending or compressing.
        */
        virtual      unsigned      addContact(ParticleContact
*contact,
                                unsigned limit) const;
    };

    class      ParticleConstraint      :      public
ParticleContactGenerator
    {
    public:

        Particle* particle;

        Vector3 anchor;

    protected:

        real currentLength() const;

    public:

        virtual      unsigned      addContact(ParticleContact
*contact,
                                unsigned limit) const = 0;
    };

```

```

        class ParticleCableConstraint : public
ParticleConstraint
    {
    public:
        /**
         * Holds the maximum length of the cable.
         */
        real maxLength;

        /**
         * Holds the restitution (bounciness) of the cable.
         */
        real restitution;

    public:
        /**
         * Fills the given contact structure with the contact
needed
         * to keep the cable from over-extending.
         */
        virtual unsigned addContact(ParticleContact
*contact,
                                   unsigned limit) const;
    };
    /**
     * Rods link a particle to an anchor point, generating a
contact if they
     * stray too far apart or too close.
     */
    class ParticleRodConstraint : public ParticleConstraint
    {
    public:
        /**
         * Holds the length of the rod.
         */
        real length;

    public:
        /**
         * Fills the given contact structure with the contact
needed
         * to keep the rod from extending or compressing.
         */
        virtual unsigned addContact(ParticleContact
*contact,
                                   unsigned limit) const;
    };
} // namespace cyclone

#endif // CYCLONE_CONTACTS_H

using namespace cyclone;

```

```

real ParticleLink::currentLength() const
{
    Vector3 relativePos = particle[0]->getPosition() -
                           particle[1]->getPosition();
    return relativePos.magnitude();
}

unsigned ParticleCable::addContact(ParticleContact *contact,
                                   unsigned limit) const
{
    // Find the length of the cable
    real length = currentLength();

    // Check if we're over-extended
    if (length < maxLength)
    {
        return 0;
    }

    // Otherwise return the contact
    contact->particle[0] = particle[0];
    contact->particle[1] = particle[1];

    // Calculate the normal
    Vector3 normal = particle[1]->getPosition() -
particle[0]->getPosition();
    normal.normalise();
    contact->contactNormal = normal;

    contact->penetration = length-maxLength;
    contact->restitution = restitution;

    return 1;
}

unsigned ParticleRod::addContact(ParticleContact *contact,
                                 unsigned limit) const
{
    // Find the length of the rod
    real currentLen = currentLength();

    // Check if we're over-extended
    if (currentLen == length)
    {
        return 0;
    }

    // Otherwise return the contact
    contact->particle[0] = particle[0];
    contact->particle[1] = particle[1];

    // Calculate the normal

```

```

        Vector3 normal = particle[1]->getPosition() -
particle[0]->getPosition();
        normal.normalise();

        // The contact normal depends on whether we're extending
or compressing
        if (currentLen > length) {
            contact->contactNormal = normal;
            contact->penetration = currentLen - length;
        } else {
            contact->contactNormal = normal * -1;
            contact->penetration = length - currentLen;
        }

        // Always use zero restitution (no bounciness)
        contact->restitution = 0;

        return 1;
    }

    real ParticleConstraint::currentLength() const
    {
        Vector3 relativePos = particle->getPosition() - anchor;
        return relativePos.magnitude();
    }

    unsigned ParticleCableConstraint::addContact(ParticleContact
*contact,
                                                unsigned limit) const
    {
        // Find the length of the cable
        real length = currentLength();

        // Check if we're over-extended
        if (length < maxLength)
        {
            return 0;
        }

        // Otherwise return the contact
        contact->particle[0] = particle;
        contact->particle[1] = 0;

        // Calculate the normal
        Vector3 normal = anchor - particle->getPosition();
        normal.normalise();
        contact->contactNormal = normal;

        contact->penetration = length-maxLength;
        contact->restitution = restitution;

        return 1;
    }

```

```

    }

    unsigned ParticleRodConstraint::addContact(ParticleContact
*contact,
                                             unsigned limit) const
    {
        // Find the length of the rod
        real currentLen = currentLength();

        // Check if we're over-extended
        if (currentLen == length)
        {
            return 0;
        }

        // Otherwise return the contact
        contact->particle[0] = particle;
        contact->particle[1] = 0;

        // Calculate the normal
        Vector3 normal = anchor - particle->getPosition();
        normal.normalise();

        // The contact normal depends on whether we're extending
or compressing
        if (currentLen > length) {
            contact->contactNormal = normal;
            contact->penetration = currentLen - length;
        } else {
            contact->contactNormal = normal * -1;
            contact->penetration = length - currentLen;
        }

        // Always use zero restitution (no bounciness)
        contact->restitution = 0;

        return 1;
    }

```

ПРИЛОЖЕНИЕ В
(обязательное)
Код программы. Модуль pfgen

```
#ifndef CYCLONE_PFGEN_H
#define CYCLONE_PFGEN_H

#include "core.h"
#include "particle.h"
#include <vector>

namespace cyclone {

    /**
     * A force generator can be asked to add a force to one
or more
     * particles.
     */
    class ParticleForceGenerator
    {
    public:

        /**
         * Overload this in implementations of the interface
to calculate
         * and update the force applied to the given particle.
         */
        virtual void updateForce(Particle *particle, real
duration) = 0;
    };

    /**
     * A force generator that applies a gravitational force.
One instance
     * can be used for multiple particles.
     */
    class ParticleGravity : public ParticleForceGenerator
    {
        /** Holds the acceleration due to gravity. */
        Vector3 gravity;

    public:

        /** Creates the generator with the given
acceleration. */
        ParticleGravity(const Vector3 &gravity);

        /** Applies the gravitational force to the given
particle. */
    };
}
```

```

        virtual void updateForce(Particle *particle, real
duration);
    };

    /**
instance    * A force generator that applies a drag force. One
    * can be used for multiple particles.
    */
    class ParticleDrag : public ParticleForceGenerator
    {
        /** Holds the velocity drag coefficient. */
        real k1;

        /** Holds the velocity squared drag coefficient. */
        real k2;

    public:

        /** Creates the generator with the given
coefficients. */
        ParticleDrag(real k1, real k2);

        /** Applies the drag force to the given particle. */
        virtual void updateForce(Particle *particle, real
duration);
    };

    /**
    * A force generator that applies a Spring force, where
    * one end is attached to a fixed point in space.
    */
    class ParticleAnchoredSpring : public
ParticleForceGenerator
    {
    protected:
        /** The location of the anchored end of the spring.
*/
        Vector3 *anchor;

        /** Holds the spring constant. */
        real springConstant;

        /** Holds the rest length of the spring. */
        real restLength;

    public:
        ParticleAnchoredSpring();

        /** Creates a new spring with the given parameters.
*/
        ParticleAnchoredSpring(Vector3 *anchor,

```

```

        real springConstant,
        real restLength);

    /** Retrieve the anchor point. */
    const Vector3* getAnchor() const { return anchor; }

    /** Set the spring's properties. */
    void init(Vector3 *anchor,
              real springConstant,
              real restLength);

    /** Applies the spring force to the given particle.
*/
    virtual void updateForce(Particle *particle, real
duration);
};

/**
 * A force generator that applies a bungee force, where
 * one end is attached to a fixed point in space.
 */
class ParticleAnchoredBungee : public
ParticleAnchoredSpring
{
public:
    /** Applies the spring force to the given particle.
*/
    virtual void updateForce(Particle *particle, real
duration);
};

/**
 * A force generator that fakes a stiff spring force, and
where
 * one end is attached to a fixed point in space.
 */
class ParticleFakeSpring : public ParticleForceGenerator
{
    /** The location of the anchored end of the spring.
*/
    Vector3 *anchor;

    /** Holds the spring constant. */
    real springConstant;

    /** Holds the damping on the oscillation of the
spring. */
    real damping;

public:

```



```

        /** Creates a new spring with the given parameters.
*/
        ParticleFakeSpring(Vector3      *anchor,      real
springConstant,
        real damping);

        /** Applies the spring force to the given particle.
*/
        virtual void updateForce(Particle *particle, real
duration);
    };

    /**
     * A force generator that applies a Spring force.
     */
    class ParticleSpring : public ParticleForceGenerator
    {
        /** The particle at the other end of the spring. */
        Particle *other;

        /** Holds the spring constant. */
        real springConstant;

        /** Holds the rest length of the spring. */
        real restLength;

    public:

        /** Creates a new spring with the given parameters.
*/
        ParticleSpring(Particle *other,
            real springConstant, real restLength);

        /** Applies the spring force to the given particle.
*/
        virtual void updateForce(Particle *particle, real
duration);
    };

    /**
     * A force generator that applies a spring force only
     * when extended.
     */
    class ParticleBungee : public ParticleForceGenerator
    {
        /** The particle at the other end of the spring. */
        Particle *other;

        /** Holds the spring constant. */
        real springConstant;

        /**

```

```

        * Holds the length of the bungee at the point it
begins to
        * generator a force.
        */
        real restLength;

    public:

        /** Creates a new bungee with the given parameters.
*/
        ParticleBungee(Particle *other,
            real springConstant, real restLength);

        /** Applies the spring force to the given particle.
*/
        virtual void updateForce(Particle *particle, real
duration);
    };

    /**
plane of
    * A force generator that applies a buoyancy force for a
    * liquid parrallel to XZ plane.
    */
    class ParticleBuoyancy : public ParticleForceGenerator
    {
        /**
        * The maximum submersion depth of the object before
        * it generates its maximum boyancy force.
        */
        real maxDepth;

        /**
        * The volume of the object.
        */
        real volume;

        /**
will be
        * The height of the water plane above y=0. The plane
        * parrallel to the XZ plane.
        */
        real waterHeight;

        /**
of
        * The density of the liquid. Pure water has a density
        * 1000kg per cubic meter.
        */
        real liquidDensity;

    public:

```

```

        /** Creates a new buoyancy force with the given
parameters. */
        ParticleBuoyancy(real maxDepth, real volume, real
waterHeight,
                        real liquidDensity = 1000.0f);

        /** Applies the buoyancy force to the given particle.
*/
        virtual void updateForce(Particle *particle, real
duration);
    };

    /**
    * Holds all the force generators and the particles they
apply to.
    */
    class ParticleForceRegistry
    {
    protected:

        /**
        * Keeps track of one force generator and the particle
it
        * applies to.
        */
        struct ParticleForceRegistration
        {
            Particle *particle;
            ParticleForceGenerator *fg;
        };

        /**
        * Holds the list of registrations.
        */
        typedef          std::vector<ParticleForceRegistration>
Registry;
        Registry registrations;

    public:
        /**
        * Registers the given force generator to apply to
the
        * given particle.
        */
        void add(Particle* particle, ParticleForceGenerator
*fg);

        /**
        * Removes the given registered pair from the
registry.

```

```

        * If the pair is not registered, this method will
have
        * no effect.
        */
        void remove(Particle* particle,
ParticleForceGenerator *fg);

        void clear();

        /**
of
        * Calls all the force generators to update the forces
        * their corresponding particles.
        */
        void updateForces(real duration);
    };
}
#endif // CYCLONE_PFGEN_H
void ParticleForceRegistry::updateForces(real duration)
{
    Registry::iterator i = registrations.begin();
    for (; i != registrations.end(); i++)
    {
        i->fg->updateForce(i->particle, duration);
    }
}

void ParticleForceRegistry::add(Particle* particle,
ParticleForceGenerator *fg)
{
    ParticleForceRegistry::ParticleForceRegistration
registration;
    registration.particle = particle;
    registration.fg = fg;
    registrations.push_back(registration);
}

ParticleGravity::ParticleGravity(const Vector3& gravity)
: gravity(gravity)
{
}

void ParticleGravity::updateForce(Particle* particle, real
duration)
{
    // Check that we do not have infinite mass
    if (!particle->hasFiniteMass()) return;

    // Apply the mass-scaled force to the particle
    particle->addForce(gravity * particle->getMass());
}

```

```

ParticleDrag::ParticleDrag(real k1, real k2)
: k1(k1), k2(k2)
{
}

void ParticleDrag::updateForce(Particle* particle, real
duration)
{
    Vector3 force;
    particle->getVelocity(&force);

    // Calculate the total drag coefficient
    real dragCoeff = force.magnitude();
    dragCoeff = k1 * dragCoeff + k2 * dragCoeff * dragCoeff;

    // Calculate the final force and apply it
    force.normalise();
    force *= -dragCoeff;
    particle->addForce(force);
}

ParticleSpring::ParticleSpring(Particle *other, real sc,
real rl)
: other(other), springConstant(sc), restLength(rl)
{
}

void ParticleSpring::updateForce(Particle* particle, real
duration)
{
    // Calculate the vector of the spring
    Vector3 force;
    particle->getPosition(&force);
    force -= other->getPosition();

    // Calculate the magnitude of the force
    real magnitude = force.magnitude();
    magnitude = real_abs(magnitude - restLength);
    magnitude *= springConstant;

    // Calculate the final force and apply it
    force.normalise();
    force *= -magnitude;
    particle->addForce(force);
}

ParticleBuoyancy::ParticleBuoyancy(real maxDepth,
                                   real volume,
                                   real waterHeight,
                                   real liquidDensity)
:
maxDepth(maxDepth), volume(volume),

```

```

    waterHeight(waterHeight), liquidDensity(liquidDensity)
    {
    }

    void ParticleBuoyancy::updateForce(Particle* particle, real
duration)
    {
        // Calculate the submersion depth
        real depth = particle->getPosition().y;

        // Check if we're out of the water
        if (depth >= waterHeight + maxDepth) return;
        Vector3 force(0,0,0);

        // Check if we're at maximum depth
        if (depth <= waterHeight - maxDepth)
        {
            force.y = liquidDensity * volume;
            particle->addForce(force);
            return;
        }

        // Otherwise we are partly submerged
        force.y = liquidDensity * volume *
            (depth - maxDepth - waterHeight) / 2 * maxDepth;
        particle->addForce(force);
    }

    ParticleBungee::ParticleBungee(Particle *other, real sc,
real rl)
    : other(other), springConstant(sc), restLength(rl)
    {
    }

    void ParticleBungee::updateForce(Particle* particle, real
duration)
    {
        // Calculate the vector of the spring
        Vector3 force;
        particle->getPosition(&force);
        force -= other->getPosition();

        // Check if the bungee is compressed
        real magnitude = force.magnitude();
        if (magnitude <= restLength) return;

        // Calculate the magnitude of the force
        magnitude = springConstant * (restLength - magnitude);

        // Calculate the final force and apply it
        force.normalise();
        force *= -magnitude;
    }

```

```

        particle->addForce(force);
    }

    ParticleFakeSpring::ParticleFakeSpring(Vector3 *anchor, real
sc, real d)
    : anchor(anchor), springConstant(sc), damping(d)
    {
    }

    void ParticleFakeSpring::updateForce(Particle* particle,
real duration)
    {
        // Check that we do not have infinite mass
        if (!particle->hasFiniteMass()) return;

        // Calculate the relative position of the particle to the
anchor
        Vector3 position;
        particle->getPosition(&position);
        position -= *anchor;

        // Calculate the constants and check they are in bounds.
        real gamma = 0.5f * real_sqrt(4 * springConstant -
damping*damping);
        if (gamma == 0.0f) return;
        Vector3 c = position * (damping / (2.0f * gamma)) +
            particle->getVelocity() * (1.0f / gamma);

        // Calculate the target position
        Vector3 target = position * real_cos(gamma * duration) +
            c * real_sin(gamma * duration);
        target *= real_exp(-0.5f * duration * damping);

        // Calculate the resulting acceleration and therefore the
force
        Vector3 accel = (target - position) * ((real)1.0 /
(duration*duration)) -
            particle->getVelocity() * ((real)1.0/duration);
        particle->addForce(accel * particle->getMass());
    }

    ParticleAnchoredSpring::ParticleAnchoredSpring()
    {
    }

    ParticleAnchoredSpring::ParticleAnchoredSpring(Vector3
*anchor,
                                                    real sc, real
rl)
    : anchor(anchor), springConstant(sc), restLength(rl)
    {
    }

```

```

    void ParticleAnchoredSpring::init(Vector3 *anchor, real
springConstant,
                                   real restLength)
    {
        ParticleAnchoredSpring::anchor = anchor;
        ParticleAnchoredSpring::springConstant = springConstant;
        ParticleAnchoredSpring::restLength = restLength;
    }

    void ParticleAnchoredBungee::updateForce(Particle* particle,
real duration)
    {
        // Calculate the vector of the spring
        Vector3 force;
        particle->getPosition(&force);
        force -= *anchor;

        // Calculate the magnitude of the force
        real magnitude = force.magnitude();
        if (magnitude < restLength) return;

        magnitude = magnitude - restLength;
        magnitude *= springConstant;

        // Calculate the final force and apply it
        force.normalise();
        force *= -magnitude;
        particle->addForce(force);
    }

    void ParticleAnchoredSpring::updateForce(Particle* particle,
real duration)
    {
        // Calculate the vector of the spring
        Vector3 force;
        particle->getPosition(&force);
        force -= *anchor;

        // Calculate the magnitude of the force
        real magnitude = force.magnitude();
        magnitude = (restLength - magnitude) * springConstant;

        // Calculate the final force and apply it
        force.normalise();
        force *= magnitude;
        particle->addForce(force);}

```


ПРИЛОЖЕНИЕ Г
(обязательное)
Код программы. Модуль pcontacts

```
#ifndef CYCLONE_PCONTACTS_H
#define CYCLONE_PCONTACTS_H

#include "particle.h"

namespace cyclone {

    /*
     * Forward declaration, see full declaration below for
complete
     * documentation.
     */
    class ParticleContactResolver;

    class ParticleContact
    {
        // ... Other ParticleContact code as before ...

        /**
         * The contact resolver object needs access into the
contacts to
         * set and effect the contact.
         */
        friend class ParticleContactResolver;

    public:
        /**
         * Holds the particles that are involved in the
contact. The
         * second of these can be NULL, for contacts with the
scenery.
         */
        Particle* particle[2];

        /**
         * Holds the normal restitution coefficient at the
contact.
         */
        real restitution;

        /**
         * Holds the direction of the contact in world
coordinates.
         */
        Vector3 contactNormal;
    };
};
```

```

        /**
         * Holds the depth of penetration at the contact.
         */
        real penetration;

        /**
         * Holds the amount each particle is moved by during
interpenetration
         * resolution.
         */
        Vector3 particleMovement[2];

    protected:
        /**
         * Resolves this contact, for both velocity and
interpenetration.
         */
        void resolve(real duration);

        /**
         * Calculates the separating velocity at this
contact.
         */
        real calculateSeparatingVelocity() const;

    private:
        /**
         * Handles the impulse calculations for this
collision.
         */
        void resolveVelocity(real duration);

        /**
         * Handles the interpenetration resolution for this
contact.
         */
        void resolveInterpenetration(real duration);

};

/**
 * The contact resolution routine for particle contacts.
One
 * resolver instance can be shared for the whole
simulation.
 */
class ParticleContactResolver
{
protected:
    /**
     * Holds the number of iterations allowed.

```

```

        */
        unsigned iterations;

        /**
record    * This is a performance tracking value - we keep a

        * of the actual number of iterations used.
        */
        unsigned iterationsUsed;

    public:
        /**
        * Creates a new contact resolver.
        */
        ParticleContactResolver(unsigned iterations);

        /**
        * Sets the number of iterations that can be used.
        */
        void setIterations(unsigned iterations);
        void resolveContacts(ParticleContact *contactArray,
                            unsigned numContacts,
                            real duration);
    };

    class ParticleContactGenerator
    {
    public:
        virtual unsigned addContact(ParticleContact
*contact,
                                unsigned limit) const =
0;
    };

} // namespace cyclone

#endif // CYCLONE_CONTACTS_H

using namespace cyclone;

void ParticleContact::resolve(real duration)
{
    resolveVelocity(duration);
    resolveInterpenetration(duration);
}

real ParticleContact::calculateSeparatingVelocity() const
{
    Vector3 relativeVelocity = particle[0]->getVelocity();
    if (particle[1]) relativeVelocity -= particle[1]-
>getVelocity();

```

```

        return relativeVelocity * contactNormal;
    }

void ParticleContact::resolveVelocity(real duration)
{
    // Find the velocity in the direction of the contact
    real separatingVelocity = calculateSeparatingVelocity();

    // Check if it needs to be resolved
    if (separatingVelocity > 0)
    {
        // The contact is either separating, or stationary -
there's
        // no impulse required.
        return;
    }

    // Calculate the new separating velocity
    real newSepVelocity = -separatingVelocity * restitution;

    // Check the velocity build-up due to acceleration only
    Vector3 accCausedVelocity = particle[0]-
>getAcceleration();
    if (particle[1]) accCausedVelocity -= particle[1]-
>getAcceleration();
    real accCausedSepVelocity = accCausedVelocity *
contactNormal * duration;

    // If we've got a closing velocity due to acceleration
build-up,
    // remove it from the new separating velocity
    if (accCausedSepVelocity < 0)
    {
        newSepVelocity += restitution *
accCausedSepVelocity;
    if (newSepVelocity < 0) newSepVelocity = 0;
    }

    real deltaVelocity = newSepVelocity - separatingVelocity;

    // We apply the change in velocity to each object in
proportion to
    // their inverse mass (i.e. those with lower inverse mass
[higher
    // actual mass] get less change in velocity)..
    real totalInverseMass = particle[0]->getInverseMass();
    if (particle[1]) totalInverseMass += particle[1]-
>getInverseMass();

    // If all particles have infinite mass, then impulses
have no effect
    if (totalInverseMass <= 0) return;

```

```

        // Calculate the impulse to apply
        real impulse = deltaVelocity / totalInverseMass;

        // Find the amount of impulse per unit of inverse mass
        Vector3 impulsePerIMass = contactNormal * impulse;

        // Apply impulses: they are applied in the direction of
the contact,
        // and are proportional to the inverse mass.
        particle[0]->setVelocity(particle[0]->getVelocity() +
            impulsePerIMass * particle[0]->getInverseMass()
        );
        if (particle[1])
        {
            // Particle 1 goes in the opposite direction
            particle[1]->setVelocity(particle[1]->getVelocity()
+
                impulsePerIMass * -particle[1]->getInverseMass()
            );
        }
    }

    void ParticleContact::resolveInterpenetration(real duration)
    {
        // If we don't have any penetration, skip this step.
        if (penetration <= 0) return;

        // The movement of each object is based on their inverse
mass, so
        // total that.
        real totalInverseMass = particle[0]->getInverseMass();
        if (particle[1]) totalInverseMass += particle[1]-
>getInverseMass();

        // If all particles have infinite mass, then we do nothing
        if (totalInverseMass <= 0) return;

        // Find the amount of penetration resolution per unit of
inverse mass
        Vector3 movePerIMass = contactNormal * (penetration /
totalInverseMass);

        // Calculate the the movement amounts
        particleMovement[0] = movePerIMass * particle[0]-
>getInverseMass();
        if (particle[1]) {
            particleMovement[1] = movePerIMass * -particle[1]-
>getInverseMass();
        } else {
            particleMovement[1].clear();
        }
    }

```

```

        // Apply the penetration resolution
        particle[0]->setPosition(particle[0]->getPosition() +
particleMovement[0]);
        if (particle[1]) {
            particle[1]->setPosition(particle[1]->getPosition()
+ particleMovement[1]);
        }
    }

    ParticleContactResolver::ParticleContactResolver(unsigned
iterations)
    :
    iterations(iterations)
    {
    }

    void ParticleContactResolver::setIterations(unsigned
iterations)
    {
        ParticleContactResolver::iterations = iterations;
    }

    void
ParticleContactResolver::resolveContacts(ParticleContact
*contactArray,
                                         unsigned
numContacts,
                                         real duration)
    {
        unsigned i;

        iterationsUsed = 0;
        while(iterationsUsed < iterations)
        {
            // Find the contact with the largest closing
velocity;
            real max = REAL_MAX;
            unsigned maxIndex = numContacts;
            for (i = 0; i < numContacts; i++)
            {
                real sepVel =
contactArray[i].calculateSeparatingVelocity();
                if (sepVel < max &&
                    (sepVel < 0 || contactArray[i].penetration >
0))
                {
                    max = sepVel;
                    maxIndex = i;
                }
            }
        }
    }

```

```

        if (maxIndex == numContacts) break;

        contactArray[maxIndex].resolve(duration);

        // Update the interpenetrations for all particles
        Vector3 *move =
contactArray[maxIndex].particleMovement;
        for (i = 0; i < numContacts; i++)
        {
            if (contactArray[i].particle[0] ==
contactArray[maxIndex].particle[0])
            {
                contactArray[i].penetration -= move[0] *
contactArray[i].contactNormal;
            }
            else if (contactArray[i].particle[0] ==
contactArray[maxIndex].particle[1])
            {
                contactArray[i].penetration -= move[1] *
contactArray[i].contactNormal;
            }
            if (contactArray[i].particle[1])
            {
                if (contactArray[i].particle[1] ==
contactArray[maxIndex].particle[0])
                {
                    contactArray[i].penetration += move[0] *
contactArray[i].contactNormal;
                }
                else if (contactArray[i].particle[1] ==
contactArray[maxIndex].particle[1])
                {
                    contactArray[i].penetration += move[1] *
contactArray[i].contactNormal;
                }
            }
        }
        iterationsUsed++;
    }
}

```

ПРИЛОЖЕНИЕ Д
(обязательное)
Код программы. Модуль particle

```
#ifndef CYCLONE_PARTICLE_H
#define CYCLONE_PARTICLE_H

#include "core.h"

namespace cyclone {

    class Particle
    {
    public:

        // ... Other Particle code as before ...

    protected:

        real inverseMass;

        real damping;

        Vector3 position;

        /**
         * Holds the linear velocity of the particle in
         * world space.
         */
        Vector3 velocity;

        Vector3 forceAccum;

        /**
         * Holds the acceleration of the particle. This value
         * can be used to set acceleration due to gravity
(its primary
         * use), or any other constant acceleration.
         */
        Vector3 acceleration;

        /*@}*/

    public:

        void integrate(real duration);
```



```

void setMass(const real mass);

/**
 * Gets the mass of the particle.
 *
 * @return The current mass of the particle.
 */
real getMass() const;

void setInverseMass(const real inverseMass);

/**
 * Gets the inverse mass of the particle.
 *
 * @return The current inverse mass of the particle.
 */
real getInverseMass() const;

/**
 * Returns true if the mass of the particle is not-
infinite.
 */
bool hasFiniteMass() const;

/**
 * Sets both the damping of the particle.
 */
void setDamping(const real damping);
real getDamping() const;
void setPosition(const Vector3 &position);

void setPosition(const real x, const real y, const
real z);

void getPosition(Vector3 *position) const;

Vector3 getPosition() const;
void setVelocity(const Vector3 &velocity);

void setVelocity(const real x, const real y, const
real z);

void getVelocity(Vector3 *velocity) const;

Vector3 getVelocity() const;

void setAcceleration(const Vector3 &acceleration);

```

```

        void setAcceleration(const real x, const real y,
const real z);

        void getAcceleration(Vector3 *acceleration) const;

        Vector3 getAcceleration() const;

        void clearAccumulator();

        void addForce(const Vector3 &force);

    };
}

#endif // CYCLONE_BODY_H

#include <assert.h>
#include <cyclone/particle.h>

using namespace cyclone;

/*
 * -----
 * FUNCTIONS DECLARED IN HEADER:
 * -----
 */

void Particle::integrate(real duration)
{
    // We don't integrate things with zero mass.
    if (inverseMass <= 0.0f) return;

    assert(duration > 0.0);

    // Update linear position.
    position.addScaledVector(velocity, duration);

    // Work out the acceleration from the force
    Vector3 resultingAcc = acceleration;
    resultingAcc.addScaledVector(forceAccum, inverseMass);

    // Update linear velocity from the acceleration.
    velocity.addScaledVector(resultingAcc, duration);

    // Impose drag.
    velocity *= real_pow(damping, duration);

```

```

        // Clear the forces.
        clearAccumulator();
    }

void Particle::setMass(const real mass)
{
    assert(mass != 0);
    Particle::inverseMass = ((real)1.0)/mass;
}

real Particle::getMass() const
{
    if (inverseMass == 0) {
        return REAL_MAX;
    } else {
        return ((real)1.0)/inverseMass;
    }
}

void Particle::setInverseMass(const real inverseMass)
{
    Particle::inverseMass = inverseMass;
}

real Particle::getInverseMass() const
{
    return inverseMass;
}

bool Particle::hasFiniteMass() const
{
    return inverseMass >= 0.0f;
}

void Particle::setDamping(const real damping)
{
    Particle::damping = damping;
}

real Particle::getDamping() const
{
    return damping;
}

void Particle::setPosition(const Vector3 &position)
{
    Particle::position = position;
}

```

```

    void Particle::setPosition(const real x, const real y, const
real z)
    {
        position.x = x;
        position.y = y;
        position.z = z;
    }

    void Particle::getPosition(Vector3 *position) const
    {
        *position = Particle::position;
    }

    Vector3 Particle::getPosition() const
    {
        return position;
    }

    void Particle::setVelocity(const Vector3 &velocity)
    {
        Particle::velocity = velocity;
    }

    void Particle::setVelocity(const real x, const real y, const
real z)
    {
        velocity.x = x;
        velocity.y = y;
        velocity.z = z;
    }

    void Particle::getVelocity(Vector3 *velocity) const
    {
        *velocity = Particle::velocity;
    }

    Vector3 Particle::getVelocity() const
    {
        return velocity;
    }

    void Particle::setAcceleration(const Vector3 &acceleration)
    {
        Particle::acceleration = acceleration;
    }

    void Particle::setAcceleration(const real x, const real y,
const real z)
    {
        acceleration.x = x;
        acceleration.y = y;
        acceleration.z = z;
    }

```

```

}

void Particle::getAcceleration(Vector3 *acceleration) const
{
    *acceleration = Particle::acceleration;
}

Vector3 Particle::getAcceleration() const
{
    return acceleration;
}

void Particle::clearAccumulator()
{
    forceAccum.clear();
}

void Particle::addForce(const Vector3 &force)
{
    forceAccum += force;
}

```

ПРИЛОЖЕНИЕ Е
(обязательное)
Спецификация

ПРИЛОЖЕНИЕ Ж
(обязательное)
Ведомость документов