БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ
КАФЕДРА ВЕБ-ТЕХНОЛОГИЙ И КОМПЬЮТЕРНОГО МОДЕЛИРОВАНИЯ

Курсовая работа

Создание клиент-серверного игрового приложения под операционную систему Android

Научный руководитель:
Дедков Даниил Юрьевич

Исполнители:
студенты 3 курса, 2 группы
Бабеня Вячеслав Игоревич
Кочурко Александр Александрович
Яковчик Антон Евгеньевич

Минск 2015

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В наше время сложно представить себе человека без сотового телефона, планшетного компьютера, смартфона или любого другого портативного мультимедийного устройства. Мы привыкли к тому, что всегда под рукой не только средство связи, но и множество полезных функций, таких как: калькулятор, органайзер, конвертер, календарь, часы. Смартфоны становятся новой мобильной игровой платформой, соревнуясь с классическими карманными игровыми системами вроде Nintendo DS или Playstation Portable. В устройстве смартфона все довольно просто.

Главным образом он состоит из нескольких отдельных блоков - памяти, процессора, который занимается вычислениями, хранилища данных, радиомодуля, который в свою очередь состоит из приемника и передатчика и отвечает за связь. Самое интересное здесь - операционная система, установленная на встроенную память. От операционной системы и ее версии зависят все основные возможности устройства. Смартфоны, как и персональные компьютеры, существуют в абсолютно разных комплектациях и под управлением разных операционных систем, разновидности которых мы рассмотрим далее. По мере роста продаж мобильных устройств во всем мире, растет и спрос на различные приложения для них.

Каждая уважающая себя компания, стремится иметь хотя бы одно мобильное приложение, чтобы быть у своего клиента ”всегда под рукой”. А существование некоторых компаний и вовсе сложно представить без мобильных устройств и специализированных программ, при помощи которых можно, например, управлять базами данных или следить за состоянием своего продукта на рынке в любой момент времени. К сожалению, на сегодняшний день не существуют определенного стандарта средства разработки мобильных приложений. Каждый производитель пытается сделать операционную систему в своем устройстве более уникальной и запоминающейся пользователю, и как следствие возникают вопросы совместимости различных приложений на разных операционных системах.

Основной целью данной работы является разработка клиент-серверного игрового приложения на примере игры жанра Shooter для мобильных устройств на базе операционной системы Android.

Android - портативная (сетевая) операционная система для коммуникаторов, планшетных компьютеров, электронных книг, цифровых проигрывателей, наручных часов и нетбуков основанная на ядре Linux. Изначально разрабатывалась компанией Android Inc., которую затем купила Google. Впоследствии Google инициировала создание альянса Open Handset Alliance (OHA), который сейчас занимается поддержкой и дальнейшим развитием платформы. Android позволяет создавать Java-приложения, управляющие устройством через разработанные Google библиотеки. Android Native Development Kit позволяет портировать (но не отлаживать)

библиотеки и компоненты приложений, написанные на C и других языках;

Для достижение поставленной цели необходимо решить следующие задачи:

- Создание инфраструктуры сервера
- Проектирование базы данных
- Разработка и проектирование User Interface
- Создание логики и описание поведения игровых моделей
- Разработка дизайна приложения

В качестве языка разработки для сервера и клиента был выбран язык Java. Мы посчитали его наиболее приемлимым для клиентской части нашего приложения, поскольку наиболее развивающиеся и поддерживаемые средства разработки под ОС Android (Android SDK) основаны на использовании Java. А для максимальной совместимости сервера и клиента для реализации сервера был также выбран этот язык.

ГЛАВА 1
НЕКОТОРЫЕ СВЕДЕНИЯ О ТЕХНОЛОГИЯХ, ЗАДЕЙСТВОВАННЫХ В РАЗРАБОТКЕ 2D ИГР

1.1. Процессы и потоки в Android OS. Управление памятью приложений

Когда запускается компонент приложения и приложение не имеет других запущенных компонентов, Android создает новый процесс для приложения с одним потоком исполнения. По умолчанию все компоненты одного приложения запускаются в одном процессе, в потоке называемом «главный». Если компонент приложения запускается и уже существует процесс для данного приложения(какой-то компонент из приложения существует), тогда компонент запущен в этом процессе и использует его поток выполнения. Вы можете изменить данное поведение, задав разные процессы для разных компонентов вашего приложения. Кроме того вы можете добавить потоки в любой процесс.

Задать отдельный процесс для компонента можно с помощью файла манифеста. Каждый тег компонента(activity, service, receiver и provider) поддерживает атрибут android:process. Данный атрибут позволяет задать процесс, в котором будет выполняться компонент. Также вы можете задать процесс в котором будут выполняться компоненты разных приложений. Также данный атрибут поддерживается тегом application, что позволяет задать определенный процесс для всех компонентов приложения.

Android пытается поддерживать процесс приложения как можно дольше, но когда потребуются ресурсы старые процессы будут вытеснены по иерархии важности.

Существует 5 уровней иерархии важности: (процессы первого уровня из списка будут удалены последними)

- Процесс с которым взаимодействует пользователь(Foreground process) К таким процессам относится например: активити с которым взаимодействует пользовать; сервис(экземпляр Service), с которым взаимодействует пользователь; сервис запущенный методом startForeground(); сервис, который выполняет один из методов своего жизненного цикла; BroadcastReceiver который выполняет метод onReceive().
- Видимый процесс Процесс, в котором не выполнены условия из пункта №1, но который влияет на то, что пользователь видит на экране. К примеру, вызван метод onPause() активити.

- Сервисный процесс Служба запущенная методом startService()
- Фоновый процесс Процесс выполняемый в фоновом режиме, который невиден пользователю.
- Пустой процесс

Отмечу, что в компонентах приложения существует метод onLowMemory(), но полагаться на то, что данный метод будет вызван нельзя, также как нельзя на 100

Когда запускается приложение, система создает «главный» поток выполнения для данного приложения, который также называется UI-потоком. Этот поток очень важен, так как именно в нем происходит отрисовка виджетов(кнопочек, списков), обработка событий вашего приложения. Система не создает отдельный поток для каждого экземпляра компонента. Все компоненты, которые запущенны в одном процессе будут созданы в потоке UI. Библиотека пользовательского интерфейса Android не является потоково-безопасной, поэтому необходимо соблюдать два важных правила:

- Не блокировать поток UI
- Не обращаться к компонентам пользовательского интерфейса не из UI-потока

AsyncTask позволяет выполнить асинхронную работу и делать обновления пользовательского интерфейса. Для обновления реализуйте метод onPostExecute(), а всю фоновую работу заключите в метод doInBackground(). После того, как вы реализуете свою собственную задачу, необходимо ее запустить методом execute().

Параметры передаваемые в AsyncTask:

- Параметры
- Прогресс(единицы задающие ход изменения задачи)
- Результат выполнения задачи

Отмечу пару важных моментов, которые нужно учитывать:

- Метод doInBackGround() выполняется в фоновом потоке, потому доступа к потоку UI внутри данного метода нет.
- Методы onPostExecute() и onProgressUpdate() выполняются в потоке UI, потому мы можем смело обращаться к нашим компонентам UI.

Android не поддерживает swap памяти. Это означает, что любые манипуляции, связанные с памятью, например, создание новых объектов, никак не влияют на выделенную память : она постоянна в RAM. Поэтому, единственно верный способ полного освобождения памяти текущего приложения - это освободить ссылки на объекты, таким образом делая память доступной сборщику мусора.

С целью обеспечения всех своих текущих потребностей в RAM, Android старается поделить её между между процессами. Это может быть достигнуто следующими путями :

Каждый процесс приложения ответвляется от Zygote процесса. Zygote про-

цесс начинает работу при запуске системы и загружает общие ресурсы (например, activity themes). Для того, чтобы новый процесс приложения, система ответвляет Zygote процесс, после чего загружает и запускает код приложения в новом процессе. Это позволяет большей части страничной памяти RAM, выделенной для ресурсов, быть поделённой между всеми процессами.

1.2. TCP и UDP протоколы

TCP и UDP

Протокол UDP (User Datagram Protocol) – протокол транспортного уровня, входящий в стек протоколов TCP/IP, обеспечивающий негарантированную доставку данных без установления виртуального соединения.

Поскольку на протокол не возлагается задач по обеспечению гарантированной доставки, а лишь требуется обеспечивать связь между различными программами, то структура заголовка дейтаграммы UDP (так называется пакет протокола) выглядит достаточно просто – она включает в себя всего четыре поля. Первые два поля содержат номера UDP-портов программы-отправителя и программы-получателя. Два остальных поля в структуре заголовка дейтаграммы предназначены для управления обработкой – это общая длина дейтаграммы и контрольная сумма заголовка.

Протокол TCP (Transmission Control Protocol) является транспортным протоколом стека протоколов TCP/IP, обеспечивающим гарантированную доставку данных с установлением виртуального соединения.

Протокол предоставляет программам, использующим его, возможность передачи непрерывного потока данных. Данные, подлежащие отправке в сеть, разбиваются на порции, каждая из которых снабжается служебной информацией, то есть формируются пакеты данных. В терминологии TCP пакет называется сегментом.

В соответствии с функциональным назначением протокола структура TCP-сегмента предполагает наличие следующих информационных полей:

- номер порта-отправителя и номер порта-получателя – номера портов, идентифицирующие программы, между которыми осуществляется взаимодействие;
- поля, предназначенные для обеспечения гарантированной доставки: размер окна, номер последовательности и номер подтверждения
- управляющие флаги – специальные битовые поля, управляющие протоколом.

Реализация режима гарантированной доставки Для обеспечения гарантированной доставки протокол TCP использует механизм отправки подтверждения. С целью снижения загрузки сети протокол TCP допускает посылку одного подтверждения сразу для нескольких полученных сегментов. Объем данных, которые могут

быть переданы в сеть отправителем до получения подтверждения, определяется специальным параметром протокола TCP - размером окна. Размер окна согласуется при установлении соединения между отправителем и получателем и может автоматически изменяться программными модулями протокола TCP в зависимости от состояния канала связи. Если в процессе передачи данных потери происходят достаточно часто, то размер окна уменьшается, и наоборот – окно может иметь большой размер, если высока надежность канала данных.

Для того, чтобы данные могли быть правильно собраны получателем в нужном порядке, в заголовке TCP-сегмента присутствует информация, определяющая положение вложенных данных в общем потоке. Отправляя подтверждение, получатель указывает положение данных, которые он ожидает получить в следующем сегменте, тем самым косвенно сообщая отправителю, какой фрагмент общего потока был успешно принят. Соответствующие поля заголовка TCP-сегмента получили название номер последовательности и номер подтверждения.

Установление соединения Перед началом передачи потока данных абоненты должны согласовать параметры передачи: размер окна и начальные номера последовательностей, относительно которых будет отсчитываться положение передаваемых в сегментах данных внутри общего потока. Очевидно, что такое согласование предполагает обмен специальными сегментами и выделение ресурсов, в частности, блоков памяти, необходимых для приема и обработки данных и подтверждений. Соответствующая последовательность действий называется установлением виртуального соединения.

1.3. Netty

Netty - это non-blocking I/O (NIO) клиент-серверный фреймворк предназначенный для разработки сетевых приложений на языке Java. Асинхронный событийно-ориентированный фрэймворк используемый для упрощения написания сетевых програм таких как TCP и UDP серверы. В отличии от традиционных Java-реализаций для работы с сетевыми протоколами, использующих синхронную модель передачи данных, Netty позволяет использовать асинхронную передачу, а также службы уровня операционной системы для достижения максимальной скорости передачи данных.

Для своей работы NIO использует:

- буферы — типы для хранения данных;
- каналы — аналоги потоков для быстрой записи или чтения данных.

1.4. Физика???

Конечного пользователя интересует визуальное поведение игровых объектов. Оно определяется положением тел в пространстве и времени и изучается кинематикой. Однако кинематика не рассматривает причины возникновения движения — этим занимается динамика. Поэтому моделирование игровой сцены строится в следующем порядке: моделируется динамика (т.е. взаимодействие тел), на его основе строится кинематическое описание, которое непосредственно и видит игрок. Фактически моделирование игровой физики сводится к решению соответствующей обратной задачи механики.

1.4.1. Кинематическое описание

1.4.1.1. Материальная точка Положение тела в пространстве в любой момент времени задается функцией радиус вектора от времени, получившей исторически название уравнение движения:

$$\mathbf{r} = \mathbf{r}(t). \tag{1.1}$$

Непосредственный вид уравнения движения может быть задан лишь в ограниченном числе тривиальных случаев. В реальных же задачах мы можем получить на основе динамики дифференциальное уравнение (или систему в случае многих тел), решением которого будет искомое уравнение движения.

$$\ddot{\mathbf{r}} = f(t, \mathbf{r}(t), \dot{\mathbf{r}}(t)). \tag{1.2}$$

Первой и второй производной является скорость и ускорение соответственно. Так как механическое состояние системы полностью характеризуется заданием координат и скоростей [**Landau1**], то возможны уравнения не выше второго порядка. В данном случае, скорость и ускорение — величины векторные, поскольку они имеют величину и направление.

1.4.1.2. Твердое тело Лишь некоторые тела могут быть промоделированы заданием лишь одного положения в пространстве (например, материальная точка), большинство тел имеет форму и по разному ведут себя в зависимости от ориентации в пространстве. Существует несколько способов задания ориентации тела в трехмерном пространстве. Изменение положения тела в пространстве характеризуются двумя псевдовекторами — угловой скоростью и угловым ускорением.

1.4.1.3. Углы Эйлера Углы Эйлера — углы, описывающие поворот абсолютно твердого тела в трехмерном евклидовом пространстве.

Углы Эйлера определяют три поворота системы[**Berezkin**], которые позволяют привести любое положение системы к текущему. Обозначим начальную систему координат как (x, y, z), конечную как (X, Y, Z). Пересечение координатных плоскостей xy и XY называется линией узлов N.

- Угол α между осью x и линией узлов — угол прецессии.
- Угол β между осями z и Z — угол нутации.
- Угол γ между осью X и линией узлов — угол собственного вращения.

Повороты системы на эти углы называются прецессия, нутация и поворот на собственный угол (вращение). Такие повороты некоммутативны и конечное положение системы зависит от порядка, в котором совершаются повороты.

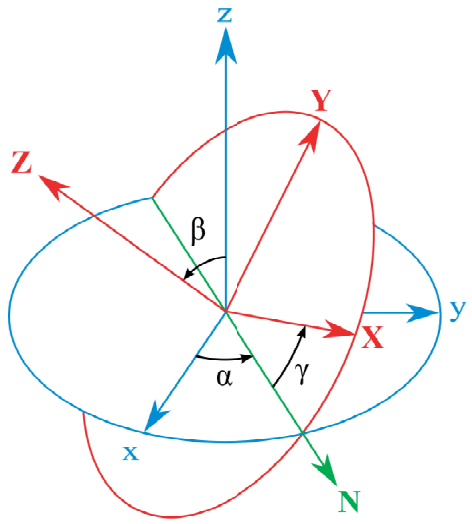

Рис. 1.1 – Углы Эйлера

1.4.1.4. Матрица поворота Матрицей поворота (или матрицей направляющих косинусов) называется ортогональная матрица, которая используется для выполнения собственного ортогонального преобразования в евклидовом пространстве. При умножении любого вектора на матрицу поворота длина вектора сохраняется. Определитель матрицы поворота равен единице. Обычно считают, что, в отличие от матрицы перехода при повороте системы координат (базиса), при умножении на матрицу поворота вектора-столбца координаты вектора преобразуются в соответствии с поворотом самого вектора (а не поворотом координатных осей; то есть при этом координаты повернутого вектора получаются в той же, неподвижной системе координат). Однако отличие той и другой матрицы лишь в знаке угла поворота, и одна может быть получена из другой заменой угла поворота на противоположный; та и другая взаимно обратны и могут быть получены друг из друга транспонированием.

Любое вращение в трехмерном пространстве может быть представлено как

композиция поворотов вокруг трех ортогональных осей (например, вокруг осей декартовых координат). Этой композиции соответствует матрица, равная произведению соответствующих трех матриц поворота. Матрицами вращения вокруг оси декартовой системы координат на угол α в трехмерном пространстве являются:

- Вращение вокруг оси x:

$$\mathrm{M}_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}, \tag{1.3}$$

- Вращение вокруг оси y:

$$\mathrm{M}_y(\alpha) = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{pmatrix}, \tag{1.4}$$

- Вращение вокруг оси z:

$$\mathrm{M}_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{1.5}$$

Во всех примерах приведены матрица поворота от результирующей системы координат к исходной.

1.4.1.5. Кватернион Кватернионы[**Kantor**] предоставляют удобное математическое обозначение положения и вращения объектов в пространстве. В сравнении с углами Эйлера, кватернионы позволяют проще комбинировать вращения[**Altmann**], а также избежать проблемы, связанной с невозможностью поворота вокруг оси, независимо от совершенного вращения по другим осям. В сравнении с матрицами они обладают большей вычислительной устойчивостью и могут быть более эффективными. Кватернионы нашли свое применение в компьютерной графике, робототехнике, навигации, молекулярной динамике.

Кватернион выгодно использовать в компьютерных вычислениях, так как представление кватерниона требует лишь 4 компоненты, против 9 компонент (3 вектора по 3 компоненты) у углов Эйлера и 9 компонент у матрицы поворота.

1.4.2. Динамическое описание

Согласно силовой модели механики равнодействующая сил приложенных к телу вызывает ускорение (второй закон Ньютона)

$$\mathbf{a} = \frac{\sum_{i=1}^{n} \mathbf{F}_i}{m}. \tag{1.6}$$

Всего существует только четыре природы силовых взаимодействий: гравитационная, электромагнитная, сильная и слабая. Последние две проявляют себя только на микроуровне, поэтому их нет необходимости рассматривать в задачах игровой физики. Гравитационная природа представлена полем тяготения, которое является консервативным. Все остальные силы являются проявлением взаимодействий электромагнитной природы на микро и макроуровне. Однако оказывается весьма неудобным при рассмотрении макроявлений использовать электромагнитные силы на микроуровне, поэтому вводятся макрообощения характеризующее соответствующее явление в целом, например, закон Кулона для трения или закон Гука для упругости.

Во многих случаях даже такое макроописание является слишком подробным и сложно реализуемым на компьютере, поэтому часто моделируют только абсолютно твердые тела, для взаимодействия которых выполняется закон сохранения импульса.

1.4.2.1. Динамика вращательного движения Для вращательного движения тела можно ввести ряд характеристик являющимися аналогами уже рассмотренных для поступательного движения. Радиус вектору $\mathbf{r}$ соответствует кватернион q или матрица вращения R, скорости $\mathbf{v}$ — угловая скорость $\boldsymbol{\omega}$ направленная вдоль оси вращения, ускорению $\mathbf{a}$ — $\boldsymbol{\alpha}$ угловое ускорение. Вызванное силой вращение зависит не только от силы, но и от положения точки приложения силы по отношению к мгновенному центру вращения, величина аналогичная силе носит название момента силы:

$$\mathbf{M} = \mathbf{r} \times \mathbf{F}, \tag{1.7}$$

она также как и сила является аддитивной величиной.

Мера отклика на момент силы зависит не только от массы тела, но и от геометрии распределения масс в теле. Данная величина получила название момента инерции тела:

$$J = \int_V \rho r^2 \mathrm{d}V. \tag{1.8}$$

В общем случае момент инерции зависит от направления оси вращения и поэтому является тензорной величиной J (фактически двухмерной матрицей 3×3), компоненты которого могут быть выражены как:

$$\mathrm{J}_{ij} = \int_V (\delta_{ij} r^2 - \mathbf{r}_i \mathbf{r}_j) \rho \mathrm{d}V, \tag{1.9}$$

где δ_{ij} символ Кронекера, а $\mathbf{r}$ радиус-вектор из центра вращения к точке интегрирования. Тензор инерции всегда является положительно определенным и может быть приведен к диагональному виду. Момент инерции относительно произвольно расположенной оси вращения может быть выражен через момент инерции параллельной оси вращения проходящей через центр масс по теореме Гюйгенса–Штейнера:

$$J = J_C + md^2. \tag{1.10}$$

Таким образом, динамическое тело может быть описано следующими минимальными характеристиками:

- масса (скаляр)
- позиция (вектор)
- скорость (вектор)
- ускорение (вектор)
- сумма сил (вектор)
- момент инерции (матрица)
- поворот (кватернион)
- угловая скорость (вектор)
- угловое ускорение (вектор)
- сумма моментов сил (вектор)

ГЛАВА 2
ГЕОМЕТРИЯ ТВЕРДЫХ ТЕЛ

2.1. Геометрические примитивы

Описание движения тела полностью абстрагировано от его формы или, как принято выражаться, геометрии — это очень важно уяснить для понимания внутренней архитектуры физического движка.

Основными геометрическими примитивами используемыми в физических движках являются:

- луч,
- плоскость,
- полигон,
- сфера,
- треугольник

Для реализации более сложных геометрических тел используется композиция тел представленных выше.

Рассмотрим более подробно один основных из геометрических примитивов — сферу. Сфера может быть описана в пространстве положением своего геометрического цента(он же является и центром масс) и радиуса. Таким образом для описания сферы достаточно 4 компонент : вектора положения центра сферы(3 компоненты) и радиуса сферы.

2.2. Контакт геометрических тел

Реализации твердого тела и геометрического объекта как правило раздельны — геометрия тела играет роль только в процессе обнаружения столкновений, рассматриваемый ниже.

Модули обнаружения столкновений (*collision detection*) и реагирования на столкновения (*collision responce*) представляют собой ключевые компоненты любого физического движка. Именно они в процессе симуляции осуществляют главную нагрузку на процессор, и именно от них требуется максимальная точность.

2.2.1. Обнаружение столкновений

Методы обнаружения столкновений подразделяются на две категории: статические и динамические. В первом случае движение тел рассматривается как после-

довательность «квантовых скачков» — проверка на пересечение двух геометрий производится в промежутках между скачками, как если бы объекты в этом время не двигались (то есть, имели нулевые скорости). Основным допущением этого метода является временная когерентность системы — предполагается, что тела обладают небольшими скоростями, и их позиции не меняются слишком резко с течением времени. Иными словами, статическое обнаружение столкновений плохо подходит для моделирования, например, пушки, стреляющей в бетонную стену: за шаг времени (Δt) снаряд пройдет расстояние, значительно превышающее толщину стены, и, следовательно, попросту пролетит сквозь нее — статическая проверка это столкновение не обнаружит.

Динамическое обнаружение столкновений[**Ericson**] использует другой метод, при котором учитывается скорость тела — движок предсказывает, какое положение в пространстве займет объект на следующем шаге времени, если будет двигаться с той же скоростью. Фактически, делается проверка на пересечение отрезка, представляющего собой траекторию движения тела, с препятствием — вычисляется точка, в которой тело столкнется с этим препятствием, и на основании этой информации движок делает соответствующие корректировки. К сожалению, динамический метод довольно сложен (особенно для нетривиальных геометрий) и, в целом, плохо адаптирован для динамики как таковой — он чаще используется в кинематике. Поэтому далее будет рассматриваться статический метода — он, при всех его недостатках, хорошо себя оправдывает в большинстве физических ситуаций.

В физическом движке функция проверки столкновения между двумя объектами должна давать на выходе следующую информацию:

- точка контакта
- вектор нормали к поверхности столкновения в этой точке (нормаль контакта)
- глубина взаимного проникновения объектов

Эти три свойства, вкупе со ссылками на столкнувшиеся тела, дают новую сущность — контакт. Соответственно, процесс обнаружения столкновений в терминологии физического движка также называют генерированием контактов (*contact generation*). В сложных движках на одно столкновение между двумя телами приходится несколько контактов — так называемый *contact manifold*. Далее будет рассматриваться простейший случай — столкновение сфер, где будет достаточно одного контакта на пару столкнувшихся тел.

Рассмотрим одно очевидным свойство. Если сумма радиусов двух сфер превышает расстояние между их центрами, следовательно, сферы пересекаются. При этом нормаль столкновения — это нормированный вектор от центра второй сферы в сторону центра первой, точка столкновения — экстремальная точка на поверх-

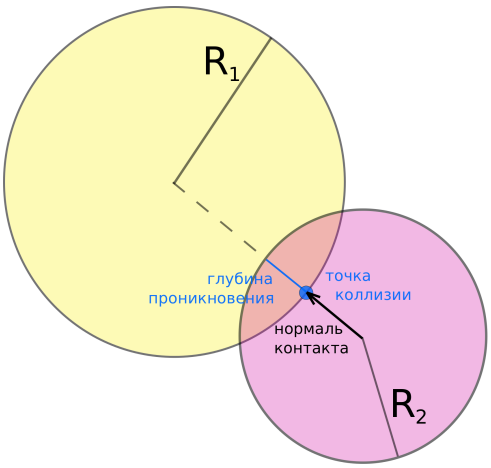

Рис. 2.1 – Контакт двух сфер

ности второй сферы в направлении этого вектора, а глубина взаимного проникновения — разница между суммой радиусов и расстоянием (см. рисунок 2.1).

Таким образом можно легко обнаружить столкновение таких двух примитивов как сферы.

2.2.2. Реагирование на столкновение

Получив необходимую информацию о контакте(точку контакта, нормаль контакта и глубину проникновения), модулю реагирования на столкновения необходимо скоординировать скорости тел таким образом, чтобы тела перестали пересекаться и проникать друг в друга.

При моделировании используются алгоритм отличный от реального поведения: объекты выталкиваются вдоль вектора нормали на величину, равную половине глубины взаимопроникновения:

$$\mathbf{x} = \frac{1}{2}\mathbf{x}_0 + d\mathbf{n} \tag{2.1}$$

где $\mathbf{x}_0$ — точка контакта,

d — глубина взаимопроникновения,

$\mathbf{n}$ — вектор нормали контакта.

Первый объект выталкивается вдоль $+\mathbf{n}$, а второй — вдоль $-\mathbf{n}$ (вектор нормали контакта указывает от второго тела к первому).

Эту операцию называют также корректировкой позиций (*position correction*). Но такой подход нестабилен, кроме того, необходимо учитывать массы тел и, следовательно, импульс.

Импульс — это векторная величина, мера механического движения тела. Вра-

щательным аналогом импульса является момент импульса.

$$\mathbf{L} = \mathbf{r} \times \mathbf{p} \tag{2.2}$$

где $\mathbf{p}$ — линейный импульс,

$\mathbf{r}$ — радиус-вектор (точка, к которой приложен линейный импульс).

Далее будет рассмотрен один из методов разрешения контакта более точно моделирующий реальность с физической точки зрения.

2.3. Разрешение контакта

Продолжим рассмотрение частного случая - контакта двух шаров. Пусть скорость первого шара равна $\mathbf{v}_1$, скорость второго - $\mathbf{v}_2$. Тогда относительную скорость$\mathbf{v}_{rel}$, без учета вращения, в точке контакта $\mathbf{x}_0$ можно рассмотреть, например, в системе отсчета второго шара, она будет равна:

$$\mathbf{v_{rel}} = \mathbf{v_1} - \mathbf{v_2} \tag{2.3}$$

Из курса механики известно, что если тело вращается вокруг центра масс $\mathbf{x}$ с угловой скоростью $\boldsymbol{\omega}$, то скорость его точки $\mathbf{p}$ можно выразить как

$$\mathbf{v_p} = (\mathbf{p} - \mathbf{x}) \times \boldsymbol{\omega} \tag{2.4}$$

Таким образом ограниченной степени свободы с учетом угловой скорости принимает вид:

$$\mathbf{v_{rel}} = \mathbf{v_1} + (\mathbf{p} - \mathbf{x_1}) \times \boldsymbol{\omega_1} - \mathbf{v_2} - (\mathbf{p} - \mathbf{x_2}) \times \boldsymbol{\omega_2} \tag{2.5}$$

Выразим проекцию относительной скорости $\mathbf{v_{rel}}$ на нормаль контакта $\mathbf{n}$, она определяется скалярным произведением:

$$v_{proj} = \mathbf{v_{rel}}\mathbf{n} \tag{2.6}$$

Если потребовать:

$$v_{proj} = 0 \tag{2.7}$$

то тем самым будет ограничена одна степень свободы системы — движение тел вдоль нормали контакта. Важно понимать, что, удовлетворение этого требования, лишь запрещает контактирующим телам проникать глубже друг в друга, никак не разрешая коллизию в случае, если тела пересеклись.

Преобразуем объединив в одно выражение:

$$v_{proj} = \mathbf{v_1}\mathbf{n} + (\mathbf{p} - \mathbf{x_1}) \times \boldsymbol{\omega_1}\mathbf{n} - \mathbf{v_2}\mathbf{n} - (\mathbf{p} - \mathbf{x_2}) \times \boldsymbol{\omega_2}\mathbf{n} \tag{2.8}$$

Применяя формулу векторного преобразования:

$$\mathbf{a} \times \mathbf{bc} = \mathbf{c} \times \mathbf{ab} \tag{2.9}$$

преобразуя, получаем:

$$v_{proj} = \mathbf{nv_1} + \mathbf{n} \times (\mathbf{p} - \mathbf{x_1})\boldsymbol{\omega_1} - \mathbf{nv_2} - \mathbf{n} \times (\mathbf{p} - \mathbf{x_2})\boldsymbol{\omega_2} \tag{2.10}$$

Эту формулу можно переписать в виде:

$$\mathbf{n_1} = \mathbf{n} \tag{2.11}$$

$$\mathbf{w_1} = \mathbf{n} \times (\mathbf{p} - \mathbf{x_1}) \tag{2.12}$$

$$\mathbf{n_2} = -\mathbf{n} \tag{2.13}$$

$$\mathbf{w_2} = -(\mathbf{n} \times (\mathbf{p} - \mathbf{x_2})) \tag{2.14}$$

Тогда уравнение для $\mathbf{v_{proj}}$ примет вид:

$$\mathbf{v_{proj}} = \mathbf{n_1 v_1} + \mathbf{w_1}\boldsymbol{\omega_1} + \mathbf{n_2 v_2} + \mathbf{w_2}\boldsymbol{\omega_2} \tag{2.15}$$

Откуда можно заключить, что ограничение, вносимое в систему из двух тел контактом, можно описать четырьмя векторами: $\mathbf{n_1}$, $\mathbf{w_1}$, $\mathbf{n_2}$, $\mathbf{w_2}$. Следует отметить, что записанные подряд координаты этих векторов образуют строку Матрицы Якоби

$$\mathrm{J} = \left(\mathbf{n_1 w_1 n_2 w_2}\right) \tag{2.16}$$

и ограничение можно описать как

$$\mathrm{J}\mathbf{v_{rel}} = 0 \tag{2.17}$$

где $\mathbf{v_{rel}}$ – вектор обобщенных скоростей:

$\mathbf{v_{rel}} = \left(\mathbf{v_1}\boldsymbol{\omega_1}\mathbf{v_2}\boldsymbol{\omega_2}\right)$

2.3.1. Метод *Sequential Impulses*

Остается понять, как можно удовлетворить условию $v_{proj} = 0$.

Взаимодействие столкнувшихся тел осуществляется передачей импульса в направлении нормали контакта $\mathbf{n}$ некоторой амплитуды λ к точке контакта x_0 к первому телу и такого же импульса противоположенного направления второму телу. Другими словами, тела при столкновении обмениваются импульсом $\lambda\mathbf{n}$.

Таким образом решение поставленной задачи можно свести к нахождению λ

Рассмотрим, как изменится линейная и угловая скорость тела, если приложить к его точке $\mathbf{x_0}$ импульс в направлении нормали $\mathbf{n}$ и модулем λ:

линейная составляющая высчитывается просто: делением вектора импульса на массу тела.

$$\mathbf{v_{res}} = \mathbf{v} + \frac{\lambda \mathbf{n}}{m}; \tag{2.18}$$

где $\mathbf{v_{res}}$ — скорость после коллизии,

$\mathbf{v}$ — вектор скорости тела

Для вычисления угловой составляющей воспользуемся формулой:

$$\boldsymbol{\omega}_{\mathbf{res}} = \boldsymbol{\omega} + ((\lambda \mathbf{n}) \times (\mathbf{p} - \mathbf{x}))\mathbf{I}^{-1} \tag{2.19}$$

где $\boldsymbol{\omega}_{\mathbf{res}}$ — угловая скорость после коллизии,

$\boldsymbol{\omega}$ — вектор угловой скорости тела,

$\mathbf{p}$ — точка контакта,

$\mathbf{x}$ — центр масс тела,

$\mathbf{I}$ — тензор инерции

Заметим, что изменения линейной и угловой скоростей можно выразить через уже найденные $\mathbf{n_1}$, $\mathbf{w_1}$, $\mathbf{n_2}$, $\mathbf{w_2}$:

$$v_{proj} = \mathbf{n_1}(\mathbf{v_1} + \frac{\lambda \mathbf{n_1}}{m_1}) + \mathbf{w_1}(\boldsymbol{\omega_1} + \lambda \mathbf{w_1} {I_1}^{-1}) + \mathbf{n_2}(\mathbf{v_2} + \frac{\lambda \mathbf{n_2}}{m_2}) + \mathbf{w_2}(\boldsymbol{\omega_2} + \lambda \mathbf{w_2} {I_2}^{-1}) \tag{2.20}$$

Преобразуем, получаем:

$$v_{proj} = \mathbf{n_1 v_1} + \mathbf{n_2 v_2} + \mathbf{w_1 \omega_1} + \mathbf{w_2 \omega_2} + \frac{\lambda \mathbf{n_1 n_2}}{m_1} + \mathbf{w_1 w_1} {I_1}^{-1} + \frac{\mathbf{n_1 n_2}}{m_2} + \lambda \mathbf{w_2 w_2} {I_2}^{-1} \tag{2.21}$$

Как было сказано выше, контакт считается решенным, если $\mathbf{v_{proj}}$ обращается в ноль. Получаем уравнение:

$$a = \mathbf{n_1 v_1} + \mathbf{n_2 v_2} + \mathbf{w_1 \omega_1} + \mathbf{w_2 \omega_2} \tag{2.22}$$

$$b = \frac{\mathbf{n_1 n_1}}{m_1} + \mathbf{w_1 w_1} {I_1}^{-1} + \frac{\mathbf{n2n2}}{m_2} + \mathbf{w_2 w_2} {I_2}^{-1} \tag{2.23}$$

$$a + \lambda b = 0; \tag{2.24}$$

Очевидным решение является $\lambda = -\frac{a}{b}$. Таким образом найден импульс, которым обмениваются тела при введении ограничения степени свободы ($\mathbf{n_1}$, $\mathbf{w_1}$, $\mathbf{n_2}$, $\mathbf{w_2}$).

Данный метод обладает одним недостатком: при большом количестве ограничений придется решать систему уравнений, чтобы удовлетворить всем ограничениям одновременно. Поскольку, большинство алгоритмов для решения систем линейных уравнений работают за $O(n^3)$, такой применяется редко.

Вместо него применяются итеративные методы, постепенно приближающие решение к более точному. Если решать полученную систему уравнений одним из таких методов, называемых *Projected Gauss-Seidel Solver (PGS)* , то последовательность производимых при этом действий математически эквивалентна последовательному решению коллизий, как описывалось выше. Этот подход широко распространен и называется *Sequential Impulses*.

Суть метода в последовательном решении ограничения за ограничением вместо решения системы ограничений. Полученный результат будет сходиться к результату, как если бы их решали в системе.

2.3.1.1. Случай упругого соударения Выше рассматривался случай, когда контакт считался решенным, если относительная скорость тел в проекции на нормаль контакта равняется нулю, то есть, другими словами, случай неупругого соударения. Алгоритм нетрудно модифицировать на случай упругого соударения. Введем коэффициент отскока *bounce*, который принимает значения от 0 до 1, при этом случаю абсолютно неупругого соударения соответствует $bounce = 0$, случаю абсолютно упругого - $bounce = 1$.

Ранее было показано, чему будет равняться проекция скорости $\mathbf{v_{proj}}$, если тела обменяются через данный контакт импульсом с модулем λ (2.21).

Если в случае с абсолютно неупругим соударением стояло условие $\mathbf{v_{proj}} = 0$, то изменив условие следующим образом:

$$\mathbf{v_{proj}} = -bounce v_{init} \tag{2.25}$$

где v_{init} — величина, равная:

$v_{init} = \mathbf{n_1 v_1} + \mathbf{w_1 \omega_1} + \mathbf{n_2 v_2} + \mathbf{w_2 \omega_2}$

Таким образом можно промоделировать соударение с разной степенью упругости.

ЗАКЛЮЧЕНИЕ

В настоящее время игровая индустрия полна физическими движками разных типов. Но большинство из них жертвуют скоростью для достижения лучшей управляемости и поддержки кода, либо, наоборот, жертвуют понятностью и доступностью для разработчиков, но обеспечивая лучшую производительность.

Написание собственной физической библиотеки на языке программирования D по-прежнему остается актуальным, язык позволяет писать быстрые, удобные и интуитивно понятные для разработчика приложения.

- На основе изученных математических и физических законов была написана объемная библиотека математики, включающая в себя реализацию таких математических объектов как *кватернион, вектор, квадратная матрица*.
- Был реализован простейший геометрический примитив — сфера.
- Исследованы особенности пересечения шаров и их контактов в трехмерном пространстве.
- С помощью математической библиотеки реализована физическая библиотека, позволяющая моделировать физическое поведение сфер разной массы и диаметра, упругости.
- Реализовано реалистичное поведение коллизии сфер: изменение направления движения, кручения сфер.
- Реализована возможность создания как упругого так и абсолютно неупругого контакта сфер.
- В качестве интегратора для моделирования движения использовался интегратор Эйлера. Метод интегрирования Эйлера позволяет получить высокую скорость расчетов при допустимой погрешности.

Физическая библиотека прошла успешное первичное тестирование. Как и ожидалось шары вступают в контакт, обладая разными скоростями, вращениями и массами, а затем реалистично разлетаются приобретая новые скорости и вращения.