Основы игростроения на Java, или учебник по LWJGL 3

Основан на учебнике от автора Antonio Hernández Bejarano

(https://lwjglgamedev.gitbooks.io/3d-game-development-with-lwjgl)

«Metal is sacred! The flesh is weak!
Deus machina! Ave Omnissiah!»

– адептус механикус

Предисловие

Что это вообще за учебник? Зачем он нужен? Зачем нужно применять чистый Lwjgl в 2017 году? На этот и другие вопросы вы найдете ответы, прочитав эту книгу. Если честно, я даже за книгу не считаю этот учебник. Взялся я за него просто от нечего делать. Думаю, а почему бы не попробовать запилить свой игровой движок на java? А почему бы и да? Взялся значит я гуглить гайды по чистому Lwjgl 3 и не нашел ничего путного, кроме одного учебника, который написан чуть более, чем полностью на забугорном языке. Вот и решил я его перевести для вас, чтобы вы наконец научились делать игровые движки и сделали наконец свой Minecraft, с блэкджеком и шлюхами.

Читая, вы можете обратить внимание на очень крутое форматирование! Это всё LibreOffice, в котором я и писал. Всё он, родной! Наконец свободный софт стал действительно лучше проприетарного, ура! Упор сделан на удобность чтения, возможно по этому вы найдете некоторые странные моменты.

Помните о том, что эта книга требует определенного уровня знаний Java. Если же со знаниями этого языка у вас всё плохо — немедленно закройте эту книгу и начните читать какую-то книгу по Java(вроде той же «Философия Java»).

Что до лицензий, то я понятия не имею, есть ли лицензия на тот текст, который я переводил. Так что не стесняйтесь распространять этот документ на лево и на право! Вдруг автор оригинала решит подать суд на всех нас, вот смеху-то будет!

Что до самого текста, то тут присутствует «отсебятина». Я старался освятить некоторые незаторонутые в оригинале моменты, так что эту книгу можно считать дополненным вариантом оригинала. Хотя, возможно, многое окажется лишним.

В общем, приятного чтения, господа!

Глава 1. Первые шаги

В этой главе вы установите библиотеку, узнаете о игровых циклах, узнаете кое-что о координатах и рендеринге.

Часть 1. Быстрый старт

Добро пожаловать в мир Lwjgl3! В первую очередь, вам нужно скачать саму библиотеку. В этом учебнике я буду использовать исключительно среду разработки Intellij Idea и сборочную систему Gradle, так что рекомендую вам тоже использовать этот стек, как минимум на время прочтения этого учебника.

Возьмем за основу то, что у вас установлена среда разработки и уже создан Gradle проект. Вам остаётся всего лишь перейти на сайт https://www.lwjgl.org/, скачать и установить оттуда библиотеку. Как это сделать написано на сайте.

Теперь вы должны опробовать пример, расположенный по этой ссылке: https://www.lwjgl.org/guide. Если он заработал, значит вы всё сделали правильно и готовы продолжить. Если же пример с сайта не работает, то вам очень не повезло и я вам сочувствую.

Часть 2. Игровой цикл

В этой части мы будем говорить о игровом цикле и именно с этой темы мы начнём разработку нашей игры. Игровой цикл — это базовый компонент любой игры. Обычно, это бесконечный цикл, который отвечает за периодическую обработку пользовательского ввода, обновление состояния игры и отрисовки(рендеринг) на экране. Круто? Конечно круто! Взгляните на структуру игрового цикла:

```
while (keepOnRunning) {
    handleInput();
    updateGameState();
    render();
}
```

Ох, это далеко не все! Вы, конечно, этого не заметили, но в приведенном мною фрагменте кода есть довольно много подводных камней. Прежде всего скорость, с которой работет

игровой цикл, будет отличаться в зависимости от машины, на которой она работает. Если машина достаточно быстрая, пользователь даже не сможет увидеть того, что происходит в игре. Более того, этот цикл будет потреблять все ресурсы машины. И что же делать? Ну вот как выйти из этой ситуации? Спокойно, давайте применим логику. Нам нужен игровой цикл, который работает с одинаковой скоростью на всех машинах, ведь так? Конечно так! Предположим, что у нас компьютер очень мощьный(на самом деле я уверен, что это не так) и мы хотим чтобы наша игра работала с частотой 50 кадров в секунду(FPS). Ну вот хотим и всё тут! Но как же видоизменится наш игровой цикл?

Всё очень просто. Вот как он изменится:

```
double secsPerFrame = 1.0d / 50.0d;

while (keepOnRunning) {
    double now = getTime();
    handleInput();
    updateGameState();
    render();
    sleep(now + secsPerFrame – getTime());
}
```

Это крайне интересный код. Тут изображен простой игровой цикл, который в теории можно применить для решения некоторых задач. Однако, этот подход не лишен проблем. Прежде всего приведенный цикл предполагает, что методы обновления и рендеринга вписываются в доступное(свободное) время, которые у нас есть для отрисовки с постоянной частотой кадров 50 FPS(то есть `secsPerFrame`, который равен 20мс(миллисекунды)). Выходит, методы обновления и рендеринга должны выполняться за время меньшее чем 20мс. Методы должны работать крайне быстро, ведь 20мс — это очень мало времени(тем более у нас Java =)).

Наш компьютер может уделять приориты другим задачам, которые не дадут игровому циклу выполниться в течении этих наших 20мс. В итоге, наше игровое состояние будет обновлятся с разными времеными промежутками, что не допустимо для большинства задач, таких как игровая физика. К примеру, один раз игровое состояние обновилось через 20мс, в другой через 21мс, в третий через 25мс.

Что до метода ожидания(`sleep`), он не точен. У этого метода есть погрешность и она может составлять десятые доли секунд, что критично в нашем случае, так как частота кадров уже будет не постоянной, даже если наши методы обновления и рендеринга выполненятся без существенных затрат времени.

Далее будет описан простой и универсальный подход к построению игровых циклов, который отлично себя проявляет в разных ситуациях. Этот подход использует паттерн, который люди обычно называют «Fixed Step Game Loop»(игровой цикл с фиксированным шагом).

Прежде всего, мы можем отдельно контролировать период обновления состояния игры и период, когда игра отображается на экране. Почему мы это делаем? Да потому что постоянная скорость обновления игрового состояния — это критически важно для подавляющего большинства задач. Хотя, если наш рендеринг не выполняется вовремя, нет смысла отображать старые кадры при обработке нашего игрового цикла, а значит появляется возможность пропустить некоторые кадры, что может положительно сказаться на производительности.

Примерно так выглядит универсальный игровой цикл:

```
double secsPerUpdate = 1.0d / 30.0d;
double previous = getTime();
double steps = 0.0;
while (true) {
  double loopStartTime = getTime();
  double elapsed = loopStartTime - previous;
  previous = current;
  steps += elapsed;

  handleInput();

  while (steps >= secsPerUpdate) {
    updateGameState();
    steps -= secsPerUpdate;
  }

  render();
  sync(current);
}
```

Этот игровой цикл позволяет обновлять состояние игры через фиксированные промежутки времени. Для того, чтобы

непрерывный рендеринг не израсходовал все ресурсы компьютера был создан метод синхронизации:

```
private void sync(double loopStartTime) {
   float loopSlot = 1f / 50;
   double endTime = loopStartTime + loopSlot;
   while(getTime() < endTime) {
       try {
           Thread.sleep(1);
       } catch (InterruptedException ie) {}
   }
}
```

Рассмотрим его. Мы вычисляем, сколько времени будет продолжаться итерация цикла игры (`loopSlot`), сумма полученого времени и времени, которое было затрачено в игровом цикле будет время ожидания. Теперь нам нужно подождать это время. Вместо того, чтобы сделать одно большое ожидание, мы делаем много маленьких. Это и позволит избежать погрешностей при использовании метода `sleep`.

Для этого выполняем следующее:

1. Вычисляем когда мы должны выйти из цикла ожидания и запустить еще одну итерацию игрового цикла(`endTime`).

2. Запускаем цикл, который будет выполняться до тех пор, пока текущее время меньше чем конечное время(`endTime`). В самом цикле запускаем ожидание(`sleep`), всего на одну миллисекунду. Так как время ожидания равно одну миллисекунду, а сам процесс ожидания зациклен — получается одно большое ожидание, которое состоит из множества маленьких.

Перед тем, как мы приступим к созданию полноценного игрового движка, нужно поговорить о ещё одном способе контроля рендеринга. В приведенном выше коде мы делаем микро ожидания, чтобы контролировать, сколько времени нам нужно ждать. Но мы можем использовать другой подход для ограничения частоты кадров. Мы можем использовать вертикальную синхронизацию(v-sync). Основной целью v-sync является предотвращение разрыва экрана(tearing).

Tearing(тиринг) - это визуальный эффект, который возникает при обновлении видеопамяти во время ее визуализации. Результатом

будет таков, что часть изображения будет представлять предыдущее изображение, а другая часть будет представлять новое.

Рис. 1.1. Разрывы кадров(tearing) в игре Counter-Strike 1.6

Если мы включим v-sync, мы не отправим изображение на GPU, пока оно отображается на экране. Если проще, то мы синхронизируем частоту кадров с частотой обновления монитора(пока кадр не отрисуется на экране, GPU новый не выдаст). Включить v-sync можно следующей строкой:

```
glfwSwapInterval(1);
```

Эта строка указывает, что перед тем, как рисовать на экране, нужно подождать, по крайней мере, одного обновления экрана. На самом деле, мы не рисуем непосредственно на экране. Вместо этого мы храним информацию в буфере, и меняем ее с помощью этого метода:

```
glfwSwapBuffers(windowHandle);
```

Таким образом, если мы включим v-sync, мы получим постоянную частоту кадров, не выполняя микро ожидания для проверки доступного времени. При этом, частота кадров будет соответствовать частоте обновления нашей видеокарты. То есть, если частота обновления равна 60 Гц (60 раз в секунду), у нас будет 60 кадров в секунду. Мы можем уменьшить эту скорость, установив число выше 1 в методе `glfwSwapInterval`. К примеру, число 2 даст 30 кадров в секунду.

Вы, вероятно уже заметили, что буквы «GLFW» часто встречаются в методах. **GLFW** — это библиотека для работы с окном, через

OpenGL. То есть всё, что связанно с окнами, работает при помощи библиотеки GLFW.

Приступим к реорганизации исходного кода. Прежде всего, мы инкапсулируем весь код инициализации окна GLFW в классе с именем `Window`, который позволяет параметризовать его характеристики (например, название и размер). Класс `Window` также предоставляет метод обнаружения нажатия клавиш, который будет использоваться в нашем игровом цикле:

```
public boolean isKeyPressed(int keyCode) {
    return glfwGetKey(windowHandle, keyCode) == GLFW_PRESS;
}
```

Класс `Window` помимо того, что предоставляет код инициализации, также должен знать об изменении размера. По тому ему нужен обратный вызов(callback), который будет вызываться всякий раз, когда изменяется размер окна. Колбэк будет получать ширину и высоту в пикселях фреймбуфера (в данном примере — это область отображения). Если вам нужна ширина, высота области отображения в координатах экрана, вы можете использовать метод `glfwSetWindowSizeCallback`.

Координаты экрана далеко не всегда соответствуют пикселям. Запомните, нас интересуют именно пиксели, а не координаты экрана! Вот как выглядит resize callback(обратный вызов, реагирующий на изминение размера экрана):

```
// Setup resize callback
glfwSetFramebufferSizeCallback(windowHandle, (window, width, height) ->
{
    Window.this.width = width;
    Window.this.height = height;
    Window.this.setResized(true);
});
```

Теперь созданим класс `Renderer`, который будет обрабатывать нашу логику рендеринга. Пускай пока у него будет пустой метод init и другой метод очистки экрана с настроенным четким цветом:

```
public void init() throws Exception {

}
```

```
public void clear() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}
```

Обратите внимание на метод `clear()`! Этот метод отвечает за очистку экрана и должен вызываться либо в начеле метода рендеринга, либо в конце. Этот метод крайне важен, ведь он предотвращает появляние артефактов при рендеринге. Дело в том, что каждый кадр заменяет предыдущий, вернее должен заменять. Метод очистки удаляет старый кадр до появления нового. Если же очистка при рендеринге будет отсутствовать, кадры попросту будут накладываться друг на друга. Результат этого наложения будет полностью непредсказуем! Запомните о методе очистке и не забывайте его использовать!

Затем создадим интерфейс `IGameLogic`, который инкапсулирует нашу логику игры. Сделав это, мы сделаем наш игровой движок универсальным для разных задач. Этот интерфейс будет иметь методы для ввода данных, для обновления состояния игры и для отображения данных, относящихся к игре.

```
public interface IGameLogic {
    void init() throws Exception;
    void input(Window window);
    void update(float interval);
    void render(Window window);
}
```

Затем создадим класс с именем `GameEngine`, который будет содержать код игрового цикла. Этот класс будет реализовывать интерфейс `Runnable`, так как игровой цикл будет запущен внутри отдельного потока:

```
public class GameEngine implements Runnable {
//...[какой-то код]...
    private final Thread gameLoopThread;

    public GameEngine(String windowTitle, int width, int height, boolean,
vsSync, IGameLogic gameLogic) throws Exception {
        gameLoopThread = new Thread(this, "GAME_LOOP_THREAD");
        window = new Window(windowTitle, width, height, vsSync);
        this.gameLogic = gameLogic;
//...[какой-то код]...
}
```

Параметр `vSync` позволяет нам выбрать, хотим ли мы использовать v-sync или нет. Вы можете увидеть, что мы создаем новый поток,

который будет выполнять метод `run` нашего класса `GameEngine`, который будет содержать наш игровой цикл:

```
public void start() {
    gameLoopThread.start();
}
@Override
public void run() {
    try {
        init();
        gameLoop();
    } catch (Exception excp) {
        excp.printStackTrace();
    }
}
```

Созданный нами класс `GameEngine` содержит метод запуска, который только запускает наш `Thread`(поток), по этому метод `run` будет выполняться асинхронно. Этот метод будет выполнять задачи инициализации и будет запускать цикл игры, пока наше окно не будет закрыто. Очень важно инициализировать GLFW внутри потока, который собирается обновить его позже. Таким образом, в этом методе `init` инициализируются экземпляры `Window` и `Renderer`.

В исходном коде вы увидите, что мы создали другие вспомогательные классы(утилиты), такие как `Timer` (который будет предоставлять методы для вычисления прошедшего времени). Все эти утилиты будут использоваться логикой игрового цикла.

Класс `GameEngine` просто делегирует(перекладывает часть работы) методы ввода и обновления экземпляру `IGameLogic`. В методе рендеринга он также делегирует экземпляр `IGameLogic` и обновляет окно.

```
protected void input() {
    gameLogic.input(window);
}

protected void update(float interval) {
    gameLogic.update(interval);
}

protected void render() {
    gameLogic.render(window);
    window.update();
}
```

Наша начальная точка, а именно класс, который содержит основной метод, только создаст экземпляр `GameEngine` и запустит его:

```
public class Main {

    public static void main(String[] args) {
        try {
            boolean vSync = true;
            IGameLogic gameLogic = new DummyGame();
            GameEngine gameEng = new GameEngine("GAME",
                600, 480, vSync, gameLogic);
            gameEng.start();
        } catch (Exception excp) {
            excp.printStackTrace();
            System.exit(-1);
        }
    }
}
```

В конце нам нужно только создать игровой логический класс, который в этой главе будет простым. Он просто изменит цвет окна всякий раз, когда пользователь нажимает клавишу вверх, или вниз. Метод `render` просто очистит окно этим цветом.

Вот как выглядит наш игровой класс:

```
public class DummyGame implements IGameLogic {

    private int direction = 0;
    private float color = 0.0f;
    private final Renderer renderer;

    public DummyGame() {
        renderer = new Renderer();
    }

    @Override
    public void init() throws Exception {
        renderer.init();
    }

    @Override
    public void input(Window window) {
        if ( window.isKeyPressed(GLFW_KEY_UP) ) {
            direction = 1;
        } else if ( window.isKeyPressed(GLFW_KEY_DOWN) ) {
            direction = -1;
        } else {
            direction = 0;
        }
    }
```

```
        @Override
        public void update(float interval) {
            color += direction * 0.01f;
            if (color > 1) {
                color = 1.0f;
            } else if ( color < 0 ) {
                color = 0.0f;
            }
        }

        @Override
        public void render(Window window) {
            if ( window.isResized() ) {
                glViewport(0, 0, window.getWidth(), window.getHeight());
                window.setResized(false);
            }
            window.setClearColor(color, color, color, 0.0f);
            renderer.clear();
        }
    }
```

В методе `render` узнаём, когда размер окна был изминён для обновления окна просмотра, чтобы найти центр координат окне(ведь его размер изменился, а значит центр находится уже в другом месте).

Созданная нами иерархия классов поможет нам отделить игровой движок от кода конкретной игры. Нам нужно изолировать общие задачи, которые каждая игра будет использовать из общей логики, ресурсов конкретной игры, чтобы повторно использовать наш игровой движок. Говоря простыми словами, нам нужно применять принципы ООП по максимуму. В последующих главах иерархия классов будет изминена, так как наш самопальный движок будет обрастать функционалом и возможностями.

Часть 3. Некоторые особенности потоков

Если вы попытаетесь запустить исходный код, приведенный выше в OSX, вы получите сообщение об ошибке:

```
Exception in thread "GAME_LOOP_THREAD" java.lang.ExceptionInInitializerError
```

Да, можно бесконечно ругать Apple за такие косяки, но давайте попробуем углубиться в проблему.

Что же эта ошибка означает? Ответ заключается в том, что некоторые функции библиотеки GLFW нельзя вызывать в потоке, который не является основным потоком. К стати, эта проблема актуальна практически для всех фреймворков(речь не только о Java) для построения GUI. Мы выполняем инициализацию, включая создание окна в методе `init` класса `GameEngine`. Этот метод вызывается в методе `run` того же класса, который вызывается новым потоком, а не тем, который используется для запуска программы.

Это ограничение библиотеки GLFW, и это означает, что нам следует избегать создания новых потоков для игрового цикла. Мы могли бы попытаться создать все связанные с окнами действия в основном потоке, но мы ничего не сможем сделать. Проблема в том, что вызовы OpenGL должны выполняться в том же потоке, что и его контекст.

На платформах Windows и Linux, хотя мы не используем основной поток для инициализации GLFW, код будет работать. Данная проблема с GLFW происходит только в OSX(Вероятно по тому на маках и не играют =)), по этому нам нужно изменить исходный код метода `run` класса `GameEngine` следующим образом:

```
public void start() {
    String osName = System.getProperty("os.name");
    if ( osName.contains("Mac") ) {
        gameLoopThread.run();
    } else {
        gameLoopThread.start();
    }
}
```

Мы просто игнорируем поток игрового цикла, если операционная система OSX и выполняем код цикла игры непосредственно в главном потоке. Это далеко не идеальное решение, но это позволит вам запускать код на на OSX. Ничего другого толком не работает, видать это единственное рабочее решение.

Вообще, вы вполне можете изучить, поддерживает ли LWJGL3 другие библиотеки графического интерфейса, чтобы проверить, относится ли это ограничение к ним. (Большое спасибо Тимо Бюльманну за то, что он указал на эту проблему).

Часть 4. Некоторые особенности платформы OSX

Весь код, который был представлен выше должен хорошо работать как в Windows, так и в Linux. Однако, если у вас Mac с установленной на нем OSX, вам стоит сделать некоторые изминения в коде, специально для этой платформы. И так, как гласит документация GLWF относительно OSX:

```
The only OpenGL 3.x and 4.x contexts currently supported by OS X are forward-compatible, core profile contexts. The supported versions are 3.2 on 10.7 Lion and 3.3 and 4.1 on 10.9 Mavericks. In all cases, your GPU needs to support the specified OpenGL version for context creation to succeed.
```

Поэтому, чтобы поддерживать функции, описанные в последующих главах, нам нужно добавить эти строки в класс `Window` до создания окна:

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
```

Это позволит программе использовать самую последнюю версию OpenGL между 3.2 и 4.1. Если эти строки не включены, используется устаревшая версия OpenGL.

И так, господа, глава первая объявляется завершенной! Конечно, сюда будут вноситься исправления и дополнения. Я очень надеюсь на то, что текст этой главы оказался простым для понимания. Если же вы обнаружили ошибку, сообщите о ней автору.

Глава 2. Краткие сведения о координатах.

В этой главе мы поговорим немного о координатах и системах координат. Мы попытаемся ввести некоторые фундаментальные математические концепции для последующих тем, которые бубут рассмотрены в последующих главах.

Мы размещаем объект в пространстве, задавая его координаты. Представте себе обыкновенную географическую карту. Каждая точка на такой карте характеризуется всего двумя числами — широтой и долготой. Всего лишь пара чисел точно идентифицирует точку на карте. Эта пара чисел - точечные координаты (в действительности всё немного сложнее, так как карта представляет собой проекцию не идеального эллипсоида, земли, поэтому требуется больше данных, но это хорошая аналогия).

Система координат - это система, которая использует одно или несколько чисел, то есть одну или несколько координат, чтобы однозначно указать положение точки. Существуют разные системы координат (декартовы, полярные и т. Д.). Вы можете преобразовывать координаты из одной системы в другую. Мы будем использовать декартову систему координат.

Декартовая(прямоугольная) система координат - прямолинейная система координат, с взаимно перпендикулярными осями на плоскости, или в пространстве. Такая система координат может состоять из двух осей(двухмерная(2D) система координат), или из трех(трехмерная(3D) система координат).

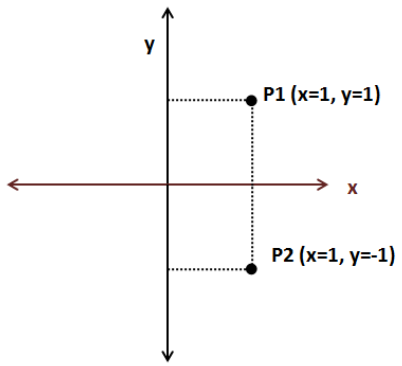

Рис 2.1. Пример двухмерной декартовой системы координат.

Продолжая аналогию с картами, системы координат определяют начало координат. Для географических координат начало находится в точке, где пересекается экватор и нулевой меридиан. Координаты для конкретной точки различны в зависимости от того, где мы устанавливаем начало координат. Система координат также может определять ориентацию оси. На предыдущем рисунке координата x увеличивается до тех пор, пока мы двигаемся вправо, а координата y увеличивается по мере продвижения вверх. Но мы могли бы использовать альтернативную декартову систему координат с иной ориентацией одной из осей, в которой мы получили бы другие координаты.

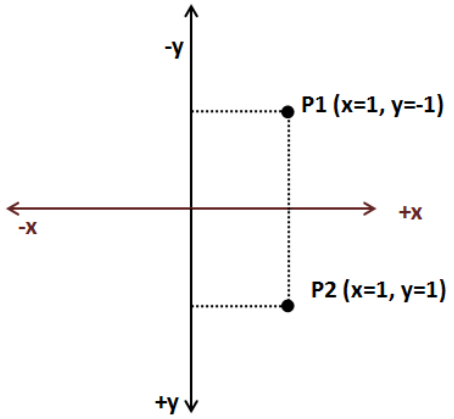

Рис 2.2. Изминенная ориентация оси y в декартовой системе координат.

Как вы можете видеть, мы должны определить некоторые произвольные параметры, такие как происхождение и ориентация оси для того, чтобы придать соответствующий смысл для пары чисел, которые представляют собой координаты. Мы будем называть эту систему координат с набором произвольных параметров, как координаты пространства. Для того, чтобы работать с набором координат, мы должны использовать одно и то же координатное пространство. Хорошая новость заключается в том, что мы можем преобразовать координаты из одного пространства в другое, просто выполняя переводы и вращения.

Если мы имеем дело с 3D-координатами, мы нуждаемся в дополнительной оси Z. 3D-координата будет образована набором из трех чисел (x, y, Z).

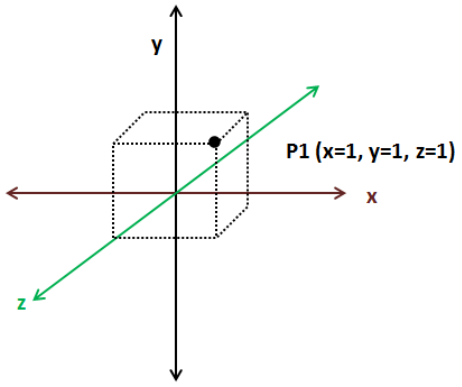

Рис 2.3. Пример трехмерной(3D) системы координат.

Как и в 2D декартовых координатных пространствах, мы можем изменить ориентацию осей в трехмерных координатных пространствах, пока оси перпендикулярны. На следующем рисунке показано другое трехмерное координатное пространство:

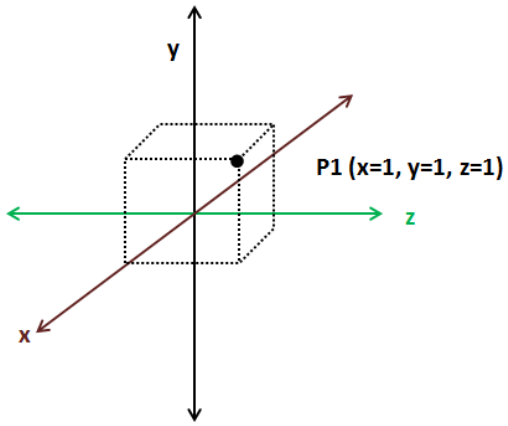

Рис 2.4. Трехмерная система координат с изминенной ориентацией осей.

3D-системы координат могут быть классифицированы в двух типах: леворучная и праворучная. Для того, чтобы определить какая система координат перед вами, возьмите свою руку и сформируйте «L» при помощи большого и указательного пальца, средний палец должен указывать в направлении, перпендикулярном двум другим. Большой палец должен указывать на направление увеличения оси x, указательный палец должен указывать, где ось y увеличивается, а средний палец

должен указывать, где увеличивается ось z. Если вы можете сделать это левой рукой, тогда это леворучная система координат, если вам нужно использовать правую руку — это праворучная система координат.

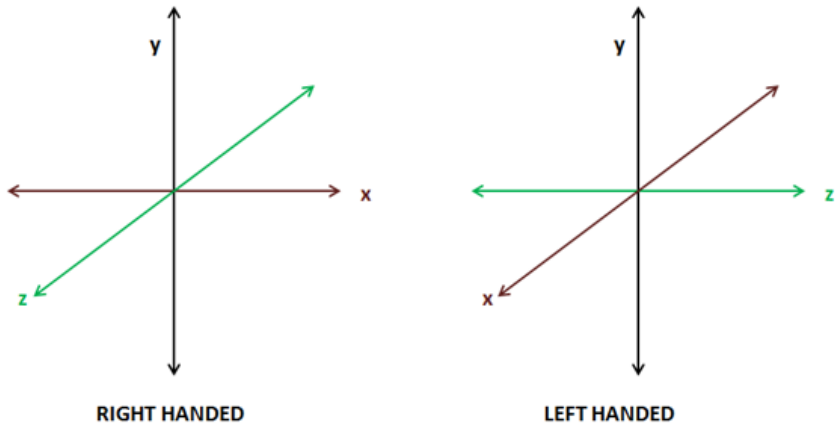

Рис 2.5. Праворучная и леворучная система координат.

Эти двухмерные координатные пространства эквивалентны, поскольку, применяя поворот, мы можем трансформировать одно в другое. Трехмерные координатные пространства, напротив, не все равны. Вы можете трансформироваться только из одного в другое, применяя вращение, если они оба имеют одинаковую ручность, то есть, если они леворучне или праворучные.

Теперь, когда мы определили некоторые основные понятия, давайте поговорим о некоторых часто используемых терминах при работе с 3D-графикой. Когда рассмотрим в последующих главах, как отрисовывать 3D-модели, мы увидим, что мы используем разные трехмерные координатные пространства. Это связанно с тем, что каждое из этих пространств координат имеет контекст и цель. Набор координат бессмыслен, если он не ссылается на что-то. К примеру у вас есть координаты 40.438031, -3.676626, они могут как иметь смысл, так и не иметь его вовсе. Но если я скажу, что это географические координаты (широта и долгота), вы увидите, что они являются координатами места в Мадриде.

Когда мы будем загружать 3D-объекты, мы получим множество трехмерных координат. Эти координаты выражаются в

трехмерном пространстве координат, которое называется пространством координат объекта. Когда графические дизайнеры создают эти 3D-модели, они ничего не знают о 3D-сцене, в которой будет отображаться эта модель, поэтому они могут определять координаты только с использованием координатного пространства, которое имеет отношение только к модели.

Когда мы будем рисовать 3D-сцену, все наши 3D-объекты будут относиться к так называемому пространству координат мира. Нам нужно будет преобразовать пространство 3D-объектов в мировые пространственные координаты. Некоторые объекты должны быть повернуты, растянуты или увеличены и переведены для правильного отображения в 3D-сцене.

Нам также нужно будет ограничить диапазон отображаемого 3D-пространства. Это похоже на перемещение камеры через наше трехмерное пространство. Нам ведь не нужно то пространство, которое мы не видим. Затем нам нужно будет преобразовать координаты пространства мира в координаты камеры(точки обзора). Наконец, эти координаты должны быть преобразованы в координаты экрана, которые являются двухмерными, поэтому нам нужно проецировать 3D-координаты в двумерное координатное пространство экрана.

На следующем рисунке показаны координаты OpenGL (ось z перпендикулярна экрану), а координаты - от -1 до +1.

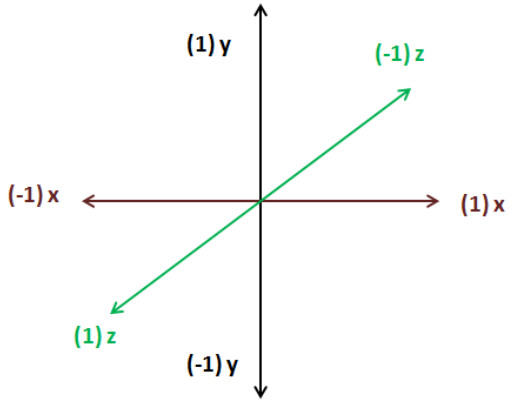

Рис 2.6. Трехмерная система координат OpenGL.

Весь этот теоретический материал будет подкреплен практикой в следующих главах. Вы можете не переживать, если у вас не получилось чего-то понять.

И так, вторая глава окончена! В этой главе представлена исключительно теория. В этой главе был сделан особый упор на понимаемость материала, т.к. дословный перевод для неподготовленного человека оказался несколько сложным. Напоминаю, что о всех найденых ошибках следует писать автору этой книги.

Глава 3. Рендеринг aka отрисовка.

гм...