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.

<u>В общем, приятного чтения, господа!</u>

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.

Взгляните на структуру игрового цикла:

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

Ох, это далеко не все! В представленном выше 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 маленький промежуток 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` определяем, 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 маки не для игр), 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.
```

Говоря простым языком, некоторые функции, описанные в этой книге работать с вашим маком не будут. Для того, чтобы функции работали, нужно включить их поддержику добавлением ряда строк 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.

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, или из трех.

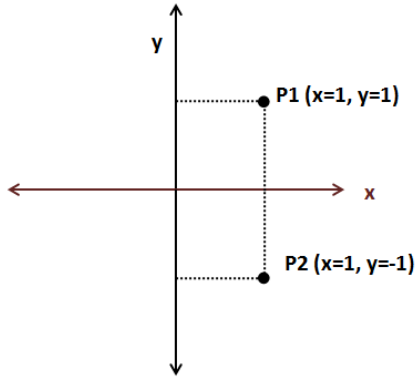

Рис 2.1. Пример двухмерной декартовой системы координат.

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.

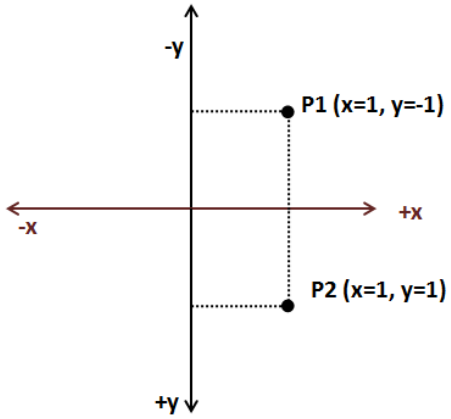

Рис 2.2. Изминенная ориентация оси y в декартовой системе координат.

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).

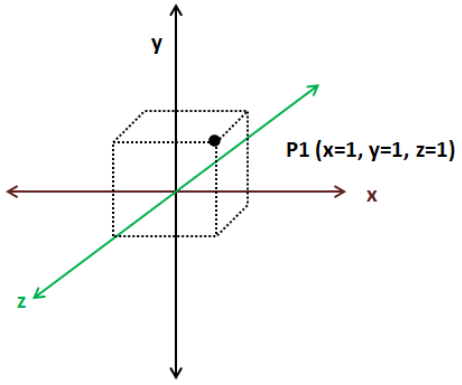

Рис 2.3. Пример трехмерной(3D) системы координат.

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:

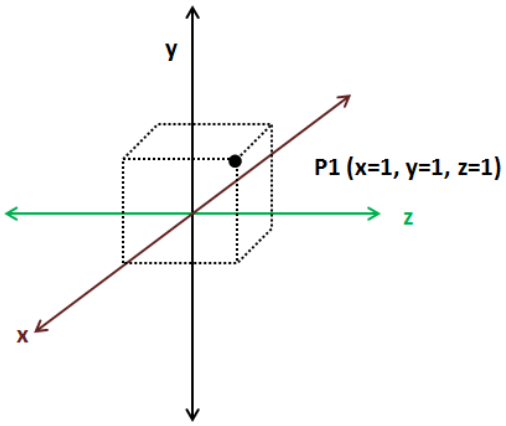

Рис 2.4. Трехмерная система координат с измененной ориентацией осей.

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.

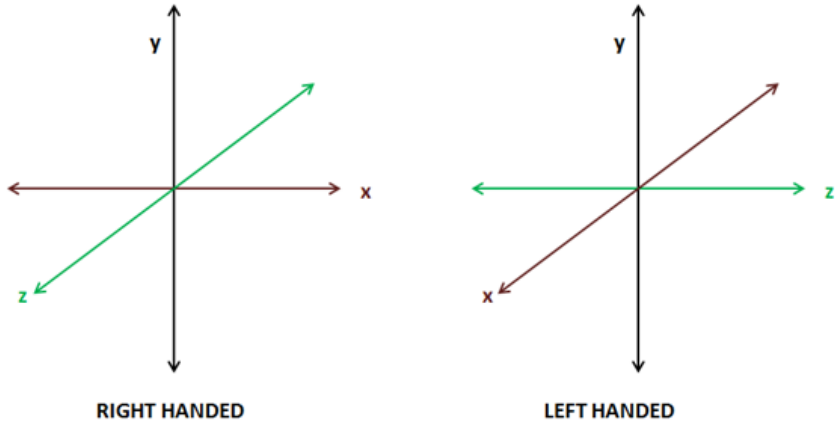

Рис 2.5. Праворучная и леворучная система координат.

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.

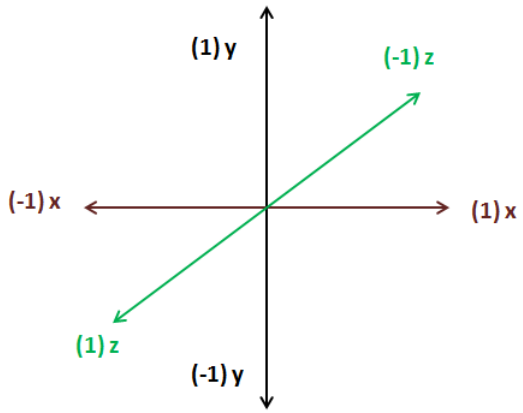

Рис 2.6. Трехмерная система координат OpenGL(ось z перпендикулярна экрану, координаты от -1 до +1).

Глава 3. Рендеринг.

В этой главе мы изучим процессы, которые происходят при рендеринге сцены с использованием OpenGL. Если вы привыкли к более старым версиям OpenGL, где применяется конвейер с фиксированной функцией, вы можете завершить эту главу, задаваясь вопросом, почему всё должно быть так сложно сложно. Современный OpenGL куда круче своих стареньких предшественников, поэтому будет использоваться именно он.

Последовательность шагов, которая заканчивается созданием 3D-представления на вашем 2D-экране, называется графическим конвейером. В первых версиях OpenGL использовалась модель, которая называлась конвейером с фиксированной функцией. Эта модель использовала набор шагов в процессе рендеринга, который определял фиксированный набор операций. У программиста был ограничен набор функций, доступных для каждого шага. Таким образом, операции, которые могут быть применены, были ограничены самим API (например, «set fog» или «add light», реализация этих функций была фиксированной и не могла быть изменена).

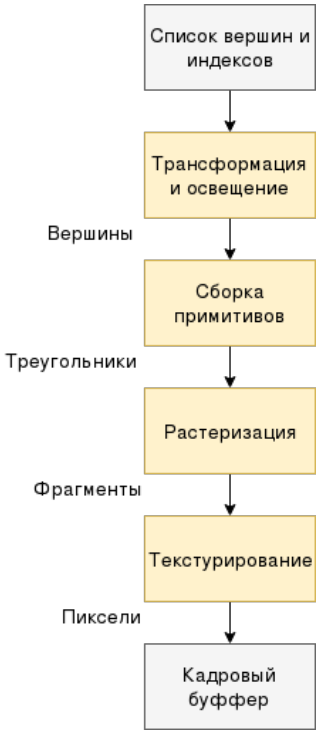

Рис 3.1. Схема конвейера OpenGL.

В OpenGL 2.0 представлена концепция программируемого конвейера. В этой модели различные этапы составления графического конвейера можно контролировать или программировать с помощью набора конкретных программ, называемых шейдерами.

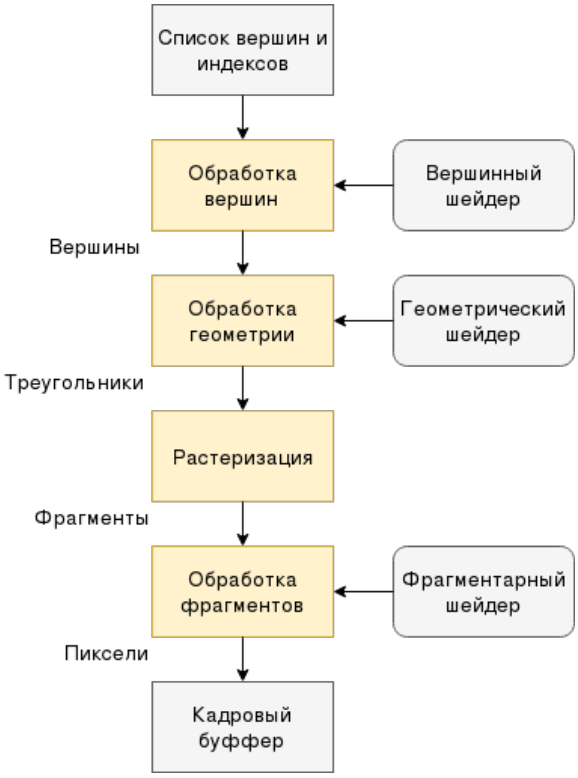

Рис 3.2. Схема упрощенной версии программируемого конвейера OpenGL.

Отрисовка начинает принимать в качестве своего ввода список вершин в виде буферов вершин(Vertex Buffer). Но что такое вершина? Вершина - это структура данных, которая описывает точку в 2D или 3D пространстве. Точка в трехмерном пространстве описывается заданными координатами x, y, z.

Буфер вершин(Vertex Buffer) - это структура данных, которая объединяет все вершины, которые нужно визуализировать, используя массивы вершин и делает эту информацию доступной для шейдеров в графическом конвейере.

Шейдер(Shader) — небольшая программа, которая выполняется прямо на графическом процессоре. Шейдер отличается от обычных програм тем, что выполняется непосредственно на видеокарте, а так

же он пишется на особом «шейдерном» языке программирования — GLSL(к примеру шейдеры для DirectX пишутся на языке HLSL). При программировании шейдеров следует уделять особое внимание числам, циклам и условиям. Видеокарта «не любит» целые числа, циклы и условия, так что нужно стараться обходится без них. Очевидно, что без этого полностью обойтись не получится, так что нужно максимально минимизировать их количество. К стати, вместо целых чисел(int) можно использовать вещественные(float).

Вернемся к вершинам. Вершины обрабатываются вершинным шейдером(Vertex Shader), основной целью которого является вычисление прогнозируемого положения каждой вершины в пространстве экрана. Этот шейдер может генерировать и другие данные, связанные с цветом или текстурой, но его главная цель - проецировать вершины в пространство экрана, то есть генерировать точки.

Вершинный шейдер(Vertex Shader) — шейдер, отвечающий за выполнение операций над вершинами. Каждое выполнение программы действует ровно на одну вершину. Если посмотреть на рисунок треугольника, то у него 3 вершины, соответственно вершинный шейдер выполнится 3 раза. Вершинный шейдер задаст конечные позиции вершин с учетом положения камеры, а так же подготовит и выведет некоторые переменные, требуемые для фрагментного шейдера.

Этап обработки геометрии соединяет вершины, которые преобразуются вершинным шейдером для формирования треугольников. Он делает это, принимая во внимание порядок хранения вершин и их группировки с использованием разных моделей. Почему треугольники? Треугольник похож на основной рабочий блок для графических карт. Это простая геометрическая форма, которая может быть объединена и преобразована для создания сложных 3D-сцен. Этот этап также может использовать

определенный шейдер для группировки вершин. В нашем случае применяется геометрический шейдер(Geometry Shader).

Геометрический шейдер(Geometry Shader) — шейдер, получающий на вход уже собранный примитив, вершины которого были обработаны вершинным шейдером. Он имеет доступ сразу ко всем вершинам примитива и в результате своей работы он отбрасывает исходный примитив, создавая вместо него ноль или более примитивов (при этом типы входных и выходных примитивов могут не совпадать). Геометрический шейдер не может существовать без вершинного.
Стадия растеризации принимает треугольники, сгенерированные на предыдущих этапах, копирует их и преобразует в фрагменты размером в пиксель.

Эти фрагменты используются на этапе обработки фрагментов с помощью фрагментарного шейдера(Fragment Shader) для генерации пикселей, назначая им конечный цвет, который записывается в фреймбуфер. Фреймбуфер является конечным результатом графического конвейера. Он содержит значение каждого пикселя, которое должно быть нарисовано на экране.

Фрагментарный шейдер(Fragment Shader) — шейдер, обрабатывающий каждую видимую часть(в этой книге - пиксели) конечного изображения.

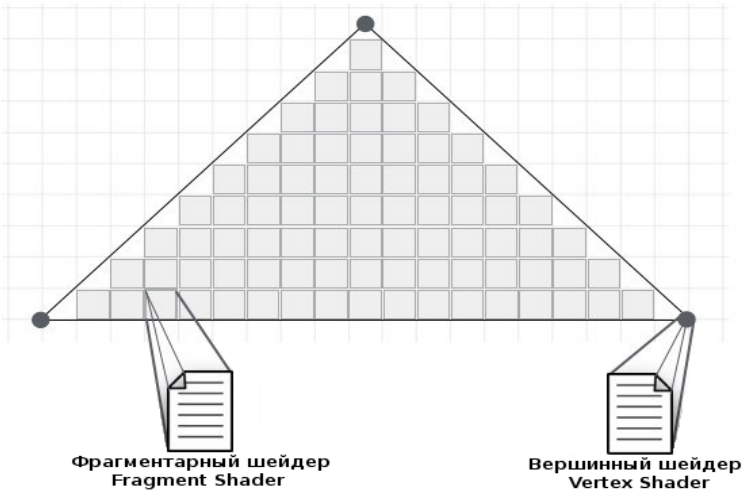

Рис 3.3. Сравнение фрагментарного и вершинного шейдеров.

Имейте в виду, что 3D-карты предназначены для распараллеливания всех операций, описанных выше. Входные данные могут быть параллельными процессами для генерации конечной сцены(прим. перевод).

Итак, давайте начнем писать нашу первую шейдерную программу. Шейдеры OpenGL, как упоминалось ранее, пишутся на языкае GLSL (OpenGL Shading Language), который основан на ANSI C. Сначала мы создадим файл с именем «vertex.vs» (расширение для Вершинного Шейдера (Vertex Shader)) в каталоге ресурсов со следующим содержимым:

```
#version 330
layout (location=0) in vec3 position;

void main()
{
    gl_Position = vec4(position, 1.0);
}
```

Первая строка - это директива(препроцессор шейдеров(Shader Preprocessor)), в которой указывается версия используемого языка GLSL.

Версия GLSL	Версия OpenGL	Дата	Препроцессор шейдеров (Shader Preprocessor)
1.10.59	2.0	Апрель 2004	#version 110
1.20.8	2.1	Сентябрь 2006	#version 120
1.30.10	3.0	Август 2008	#version 130
1.40.08	3.1	Март 2009	#version 140
1.50.11	3.2	Август 2009	#version 150
3.30.6	3.3	Февраль 2010	#version 330
4.00.9	4.0	Март 2010	#version 400
4.10.6	4.1	Июль 2010	#version 410
4.20.11	4.2	Август 2011	#version 420
4.30.8	4.3	Август 2012	#version 430
4.40	4.4	Июль 2013	#version 440
4.50	4.5	Август 2014	#version 450
4.60	4.6	Июль 2017	#version 460

Табл 3.1. Соответствие версий GLSL, OpenGL, дат принятия и директив.

Вторая строка определяет формат ввода для этого шейдера. Данные в буфере OpenGL могут быть любыми, то есть язык GLSL не заставляет вас передавать определенную структуру данных с предопределенной семантикой. С точки зрения шейдера он ожидает получить буфер с данными. Это может быть позиция, позиция с некоторой дополнительной информацией или тем, что мы хотим. Вершинный шейдер просто получает массив float`ов(числа с плавающей запятой). Когда мы заполняем буфер, мы определяем куски буфера, которые будут обрабатываться шейдером.

Итак, сначала нам нужно, чтобы этот кусок был чем-то значимым для нас. В этом случае мы говорим, что, начиная с позиции 0, мы ожидаем получить вектор, состоящий из 3 атрибутов (x, y, z).

Шейдер имеет главный блок, как и любая другая программа C. Он просто возвращает полученную позицию в выходной переменной gl_Position без применения какого-либо преобразования. Теперь вам может быть интересно, почему вектор трех атрибутов был преобразован в вектор из четырех атрибутов (vec4). Это связано с тем, что gl_Position ожидает результат в формате vec4, поскольку он использует однородные координаты. То есть он ожидает что-то в форме (x, y, z, w), где w представляет дополнительное измерение. Зачем добавлять другое измерение? В последующих главах вы увидите, что большинство операций, которые нам нужны, основаны на векторах и матрицах. Некоторые из этих операций не могут быть объединены, если у нас нет этого дополнительного измерения. Например, мы не могли объединить операции поворота и перевода. (это дополнительное измерение позволяет нам комбинировать аффинные и линейные преобразования. Узнать об этом можно подробнее, прочитав отличную книгу «3D Math Primer for Graphics and Game development, от авторов Fletcher Dunn and Ian Parberry»).

Афинное преобразование — это преобразование плоскости, которое взаимно однозначно и образом любой прямой является прямая.

Взаимно однозначное преобразование — это преобразование, которое разные точки переводит в разные, и в каждую точку переходит какая-то точка. Частным случаем аффинных преобразований являются просто движения (без какого-либо сжатия или растяжения).

Движения — это такие преобразования, которые сохраняют расстояние между любыми двумя точками неизменным, а именно параллельные переносы, повороты, различные симметрии и их комбинации.

Давайте посмотрим на наш первый фрагментарный шейдер. Мы создадим файл с именем «фрагмент.fs» (расширение для фрагментарного шейдера) в каталоге ресурсов со следующим содержимым:

```
#version 330
out vec4 fragColor;

void main( )
{
    fragColor = vec4(0.0, 0.5, 0.5, 1.0);
}
```

Структура очень похожа на наш вершинный шейдер. В этом случае мы установим фиксированный цвет для каждого фрагмента. Выходная переменная определяется во второй строке и устанавливается как vec4 fragColor.
Теперь, когда мы создали свои шейдеры, нужно понять как их использовать. Для этого нужно выполнить следующую последовательность шагов:

1. *Создать программу OpenGL.*
2. *Загрузите файлы вершин и шейдеров.*
3. *Для каждого шейдера создайте новую шейдерную программу и укажите ее тип (вершина, фрагмент).*
4. *Скомпилируйте шейдер.*
5. *Присоедините шейдер к программе.*
6. *Соберите(свяжите) программу.*

В конце шейдер будет загружен в графическую карту, и мы сможем использовать его, указав идентификатор программы.

```
package org.lwjglb.engine.graph;
import static org.lwjgl.opengl.GL20.*;

public class ShaderProgram {
    private final int programId;
    private int vertexShaderId;
    private int fragmentShaderId;
    public ShaderProgram() throws Exception {
        programId = glCreateProgram();
        if (programId == 0) {
            throw new Exception("Could not create Shader");
        }
    }

    public void createVertexShader(String shaderCode) throws Exception {
        vertexShaderId = createShader(shaderCode, GL_VERTEX_SHADER);
    }

    public void createFragmentShader(String shaderCode) throws Exception {
        fragmentShaderId = createShader(shaderCode, GL_FRAGMENT_SHADER);
    }

    protected int createShader(String shaderCode, int shaderType) throws Exception {
        int shaderId = glCreateShader(shaderType);
        if (shaderId == 0) {
            throw new Exception("Error creating shader. Type: " + shaderType);
        }

        glShaderSource(shaderId, shaderCode);
        glCompileShader(shaderId);

        if (glGetShaderi(shaderId, GL_COMPILE_STATUS) == 0) {
            throw new Exception("Error compiling Shader code: " + glGetShaderInfoLog(shaderId,
1024));
        }
        glAttachShader(programId, shaderId);
        return shaderId;
    }

    public void link() throws Exception {
        glLinkProgram(programId);
        if (glGetProgrami(programId, GL_LINK_STATUS) == 0) {
            throw new Exception("Error linking Shader code: " + glGetProgramInfoLog(programId,
1024));
        }

        if (vertexShaderId != 0) {
            glDetachShader(programId, vertexShaderId);
        }
        if (fragmentShaderId != 0) {
            glDetachShader(programId, fragmentShaderId);
        }

        glValidateProgram(programId);
        if (glGetProgrami(programId, GL_VALIDATE_STATUS) == 0) {
            System.err.println("Warning validating Shader code: " +
glGetProgramInfoLog(programId, 1024));
        }
```

```
        }

        public void bind() {
            glUseProgram(programId);
        }

        public void unbind() {
            glUseProgram(0);
        }

        public void cleanup() {
            unbind();
            if (programId != 0) {
                glDeleteProgram(programId);
            }
        }
    }
```

Конструктор `ShaderProgram` создает новую программу в OpenGL и предоставляет методы для добавления шейдеров вершин и фрагментов. Эти шейдеры скомпилированы и присоединены к программе OpenGL. Когда все шейдеры присоеденены, следует вызвать метод связывания(линковки), который связывает(линкует) весь код и проверяет, что все сделано правильно.

Как только программа шейдера была связана, скомпилированные шейдеры вершин и фрагментов могут быть освобождены (при помощи вызова `glDetachShader`).

Что касается проверки, это делается с помощью вызова `glValidateProgram`. Этот метод используется в основном в отладочных целях, и его следует удалять, когда ваша программа достигает стадии релиза. Этот метод пытается проверить правильность шейдера с учетом текущего состояния OpenGL. Это означает, что проверка может быть неудачной в некоторых случаях, даже если шейдер правилен, из-за того, что текущее состояние недостаточно полно для запуска шейдера (некоторые данные, возможно, еще не загружены). Таким образом, вместо отказа мы просто выводим сообщение об ошибке.

`ShaderProgram` также предоставляет методы для активации этой программы для рендеринга (bind) и прекращения ее использования (unbind). Наконец, он предоставляет метод очистки(`cleanup`) для освобождения всех ресурсов, когда они больше не нужны.

Поскольку у нас есть метод очистки, давайте изменим наш класс интерфейса `IGameLogic`, чтобы добавить метод очистки:

```
void cleanup();
```

Этот метод будет вызываться, когда игровой цикл завершится, поэтому нам нужно изменить метод `run` класса `GameEngine`:

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

Теперь мы можем использовать наши шейдеры для отображения треугольника. Мы сделаем это в методе `init` нашего класса `Renderer`. Прежде всего, мы создаем шейдерную программу:

```
public void init() throws Exception {
    shaderProgram = new ShaderProgram();
    shaderProgram.createVertexShader(Utils.loadResource("/vertex.vs"));
    shaderProgram.createFragmentShader(Utils.loadResource("/fragment.fs"));
    shaderProgram.link();
}
```

Мы создали класс утилит(`Utils`), который предоставляет метод для извлечения содержимого файла из пути к классу. Этот метод используется для извлечения содержимого наших шейдеров.

Теперь мы можем определить наш треугольник как массив float`ов. Мы создаем единый массив float, который будет определять вершины треугольника. Как видите, в этом массиве нет структуры. Как и сейчас, OpenGL не может знать структуру этих данных. Это всего лишь последовательность float`ов:

```
float[] vertices = new float[] {
     0.0f,  0.5f, 0.0f,       // Вершина 1
    -0.5f, -0.5f, 0.0f,       // Вершина 2
     0.5f, -0.5f, 0.0f        // Вершина 3
};
```

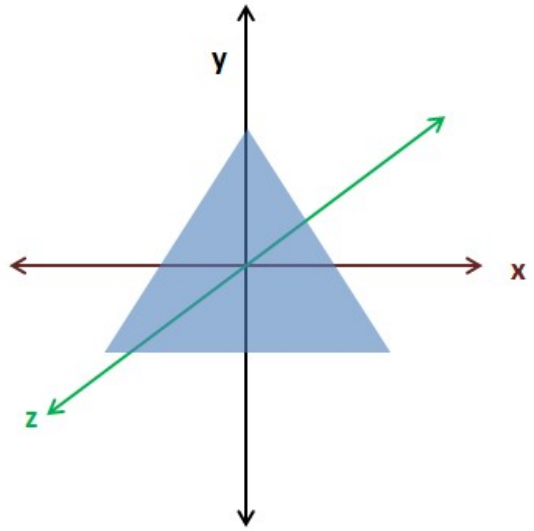

Рис. 3.4. Треугольник в нашей системе координат, созданный по массиву `vertices`.

Теперь, когда у нас есть координаты, нам нужно сохранить их в графической карте и сообщить OpenGL о структуре. Самое время ввести два новых важных понятия: объекты вершинного массива (Vertex Array Objects aka VAO) и объект буфера вершин (Vertex Buffer Object aka VBO). Помните, что в конце мы делаем отправку данных в память графической карты. Эти данные моделируют объекты, которые мы хотим нарисовать. Когда мы храним их, мы получаем идентификатор, который позже нами будет использоваться. Этот самый идентификатор и позволит ссылаться на эти данные.

Объект буффера вершин(Vertex Buffer Object aka VBO) - это буфер памяти, хранящийся в памяти графической карты(GPU), в котором хранятся вершины. Грубо говоря, **VBO** - это такое средство OpenGL, позволяющее загружать определенные данные в память GPU. Использование VBO даёт прирост производительности за счёт того, что данные хранятся непосредственно в памяти GPU, а не в оперативной памяти.

Именно в VBO мы и перенесем наш массив float`ов, который моделирует треугольник. Как было упомянуто ранее, OpenGL ничего не знает о структуре наших данных. Фактически он может содержать не только координаты, но и другую информацию, такую как текстуры,

цвет и т. д.

Объект вершинного массива (VAO) - это объект, который содержит один или несколько VBO, которые обычно называются списками атрибутов. Каждый список атрибутов может содержать один тип данных: положение, цвет, текстура и т. д. Вы можете хранить все, что захотите, в каждом слоте. Грубо говоря, **VAO** представляет собой массив, в элементах которого хранится информация о том, какую часть некого VBO использовать, и как эти данные нужно интерпретировать. Таким образом, один VAO по разным индексам может хранить координаты вершин, их цвета, нормали и прочие данные. Переключившись на нужный VAO мы можем эффективно обращаться к данным, на которые он «указывает», используя только индексы.

VAO похож на обертку, которая группирует набор определений для данных, которые будут храниться в видеокарте. Когда мы создаем VAO, получаем идентификатор. Мы используем этот идентификатор для его отображения содержащихся в нем элементов, используя определения, которые мы указали при его создании. (прим. перевод)

Первое, что мы должны сделать, это сохранить наш массив float`ов в `FloatBuffer`. Это связано с тем, что мы должны взаимодействовать с библиотекой OpenGL, написанной на C, поэтому мы должны преобразовать наш массив float во что-то, что может управляться библиотекой.

```
FloatBuffer verticesBuffer = MemoryUtil.memAllocFloat(vertices.length);
verticesBuffer.put(vertices).flip();
```

Мы используем класс `MemoryUtil` для создания буфера в свободной памяти, чтобы он был доступен библиотеке OpenGL. После того, как мы сохранили данные (с помощью метода `put`), нам нужно сбросить положение буфера до позиции 0 с помощью метода `flip` (то есть, мы говорим, что мы закончили писать в нем). Помните, что объекты Java выделены в пространстве, называемом кучей.

Куча - большая группа памяти, зарезервированная в памяти процесса JVM. Память, находящаяся в куче, не может быть доступна с помощью нативного кода (JNI, механизм, который позволяет вызывать нативный код из Java, не позволяет этого). Единственный способ обмена данными между Java и нативным кодом - это прямое распределение памяти на Java.

Если вы использовали предыдущие версии LWJGL ранее, важно подчеркнуть несколько моментов. Возможно, вы заметили, что мы не используем класс `BufferUtils` для создания буферов. Вместо этого мы используем класс `MemoryUtil`. Это связано с тем, что `BufferUtils` не очень эффективен и поддерживается только для обратной совместимости. Вместо этого LWJGL 3 предлагает два метода управления буфером:

- Буферы с автоматическим управлением, то есть буферы, которые автоматически собираются сборщиком мусора. Эти буферы используются для коротких операций или для данных, которые передаются на GPU и не должны присутствовать в памяти процесса. Это делается с помощью класса `org.lwjgl.system.MemoryStack`.
- Управляемые вручную буферы. В этом случае нам нужно осторожно освободить их, как только мы закончим с ними работу. Эти буферы предназначены для длительных операций или для больших объемов данных. Это делается с помощью класса `MemoryUtil`.

(Вы можете изучить детали на сайте lwjgl: https://blog.lwjgl.org/memory-management-in-lwjgl-3/)

Так как наши данные отправляются на GPU, мы можем использовать буферы с автоматическим управлением. Но позже мы будем использовать их для хранения потенциально больших объемов данных, из-за этого нам нужно будет управлять ими вручную. Именно по этой причине мы используем класс MemoryUtil, и поэтому мы освобождаем буфер в блоке `finally`. В следующих главах будет рассмотрено, как использовать буферы с автоматическим управлением.

Теперь нам нужно создать VAO и связать(забиндить) его.

```
vaoId = glGenVertexArrays();
glBindVertexArray(vaoId);
```

Затем нам нужно создать VBO, связать его(забиндить) и поместить в него данные.

```
vboId = glGenBuffers();
glBindBuffer(GL_ARRAY_BUFFER, vboId);
glBufferData(GL_ARRAY_BUFFER, verticesBuffer, GL_STATIC_DRAW);
memFree(verticesBuffer);
```

Сейчас самая важная часть. Нам нужно определить структуру наших данных и сохранить их в одном из списков атрибутов VAO. Это делается следующей строкой:

```
glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
```

Параметры:

- Индекс: Указывает место, где шейдер ожидает эти данные.
- Размер: Указывает количество компонентов на атрибут вершины (от 1 до 4). Так как у нас 3D-координаты, это должно быть 3.
- Тип: Определяет тип каждого компонента в массиве, в данном случае — float.
- нормализация: Указывает, должны ли значения быть нормализованы или нет.
- шаг: Задает смещение байта между последовательными атрибутами вершины. (будет объяснено позже)
- смещение: Задает смещение первого компонента в буфере.

После того, как мы завершим наш VBO, мы должны отвязать (разбиндить) его и VAO (привязать их к 0).

```
// Unbind the VBO
glBindBuffer(GL_ARRAY_BUFFER, 0);

// Unbind the VAO
glBindVertexArray(0);
```

Как только вы это сделаете, следует освободить память кучи, выделенную `FloatBuffer`. Это делается путем вызова `memFree` вручную, поскольку сборка мусора Java не будет очищать память вне кучи.

```
if (verticesBuffer != null) {
    MemoryUtil.memFree(verticesBuffer);
}
```

Весь этот код должен быть в методе `init`. Наши данные уже находятся в графической карте, готовой к использованию. Нам нужно только изменить наш метод рендеринга, чтобы использовать его каждый шаг рендеринга во время работы нашего игрового цикла.

```
public void render(Window window) {
    clear();

    if ( window.isResized() ) {
        glViewport(0, 0, window.getWidth(), window.getHeight());
        window.setResized(false);
    }

    shaderProgram.bind();

    // Bind to the VAO
    glBindVertexArray(vaoId);
    glEnableVertexAttribArray(0);

    // Draw the vertices
    glDrawArrays(GL_TRIANGLES, 0, 3);

    // Restore state
    glDisableVertexAttribArray(0);
    glBindVertexArray(0);

    shaderProgram.unbind();
}
```

Как вы можете видеть, мы просто очищаем окно, привязываем программу шейдеров, связываем VAO, рисуем вершины, хранящиеся в VBO, связанные с VAO, и восстанавливаем состояние.

Также был добавилен метод очистки в класс `Renderer`, который освобождает выделенные ресурсы:

```
public void cleanup() {
    if (shaderProgram != null) {
        shaderProgram.cleanup();
    }
```

```
        glDisableVertexAttribArray(0);

        // Delete the VBO
        glBindBuffer(GL_ARRAY_BUFFER, 0);
        glDeleteBuffers(vboId);

        // Delete the VAO
        glBindVertexArray(0);
        glDeleteVertexArrays(vaoId);
    }
```

И это все! Если вы всё сделали правильно, должен появиться треугольник!

По-сути это ваша первая игра на OpenGL! Конечно, это не Crysis 3 на ультрах, но уровень minecraft`а достигнут однозначно(наша игра хоть не лагает, да и графика в ней лучше, так что она уже лучше чем minecraft)! Вам может показаться, что мы написали слишком много для рисования простенького треугольника. Да, этот треугольник ложит большую частсь инде треша на лопатки, однако это не оправдание. Осознайте то, что мы вводим ключевые концепции и готовим базовую инфраструктуру для выполнения более сложных задач. В принципе для многих уровень треугольника является достаточным. Если вам нравится ваш треугольник, смело выкладывайте его в Steam, или другие платформы цифровой дистрибьюции. Остальные, должны дочитать книгу до конца, чтобы сделать что-то по настоящему крутое! (ибо больше возможности завершить обучение у вас не будет)