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)

Посвящается игре Minecraft…

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 написать чуть более, чем полностью на заграничном. 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! Ну и ещё мне было попросту лень что-то придумывать, по тому делал так, чтобы читалось максимально удобно.

Будьте предельно внимательны и следите за мыслью. Делать мне больше нечего, как учить вас, ленивых. Но раз взялся, так вы хоть вид сделайте, что вам интересно, ибо бесплатно же. Как говорили умные люди: «На халяву и хлорка сахар, а бесплатный учебник тем более».

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!

В общем, приятного чтения, господа!

Глава 1. Первые шаги

В этой главе вы установите библиотеку, узнаете о понятии «Game Loop», узнаете кое-что о координатах и рендеринге.

Часть 1. Быстрый старт

Добро пожаловать в мир Lwjgl! 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 lwjgl.org и скачать оттуда библиотеку.

В качестве поощрения за то, что вы начали читать, даю готовый код, взятый с сайта lwjgl. Однако, в момент прочтения этот код может перестать быть актуальным, так что потратьте время и возьмите лучше код самостоятельно(если не знаете как, потом будет разъяснение). Вставляете в ваш build.gradle.

```
import org.gradle.internal.os.OperatingSystem
switch ( OperatingSystem.current() ) {
        case OperatingSystem.WINDOWS:
                project.ext.lwjglNatives = "natives-windows"
                break
        case OperatingSystem.LINUX:
                project.ext.lwjglNatives = "natives-linux"
        break
        case OperatingSystem.MAC_OS:
                project.ext.lwjglNatives = "natives-macos"
                break
}

project.ext.lwjglVersion = "3.1.2"

repositories {
        mavenCentral()
}

dependencies {
        compile "org.lwjgl:lwjgl:${lwjglVersion}"
        compile "org.lwjgl:lwjgl-glfw:${lwjglVersion}"
        compile "org.lwjgl:lwjgl-jemalloc:${lwjglVersion}"
        compile "org.lwjgl:lwjgl-openal:${lwjglVersion}"
        compile "org.lwjgl:lwjgl-opengl:${lwjglVersion}"
        compile "org.lwjgl:lwjgl-stb:${lwjglVersion}"
        runtime "org.lwjgl:lwjgl:${lwjglVersion}:${lwjglNatives}"
        runtime "org.lwjgl:lwjgl-glfw:${lwjglVersion}:${lwjglNatives}"
        runtime "org.lwjgl:lwjgl-jemalloc:${lwjglVersion}:${lwjglNatives}"
        runtime "org.lwjgl:lwjgl-openal:${lwjglVersion}:${lwjglNatives}"
        runtime "org.lwjgl:lwjgl-opengl:${lwjglVersion}:${lwjglNatives}"
        runtime "org.lwjgl:lwjgl-stb:${lwjglVersion}:${lwjglNatives}"
}
```

Теперь вы можете скопировать и попробовать запустить 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 удивлены? Если да, то ничего страшного, ведь при помощи 5-ти строчек игру не напишешь! 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?

А вот как:

```
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 некоторых ваших игровых поделий. Однако, тут так же есть некоторые проблемы. 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.

Однако, наш компьютер может уделять приоритеное внимание другим задачам, которые предотвращают выполнение игрового цикла в течении заданного периода времени. В итоге, мы можем в конечном итоге обновить наше игровое состояние с разными времеными шагами, которые не подходят для большинства важных задач, таких как игровая физика. К примеру, один раз игровое состояние обновилось через 20мс, в другой через 21мс, в третий через 25мс.

Что до ожидания(`sleep`), точность его может составлять до десятой доли секунды, по тому, мы не сможем обновиться с постоянной частотой кадров, даже если наши методы обновления и рендеринга не требуют времени. Запомните, ожидание не точно!

Я постараюсь вам показать простой и универсальный подход к построению игровых циклов, который отлично себя проявляет в разных ситуациях. Мой подход использует шаблон(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 важный параметр для многих 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.

Давайте посмотрим на новый, 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`), полученное время и

время, 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 проблемы с ожиданием, о которой было упомянуто выше(`sleep` не точен).

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).

Что такое разрывы экрана, или tearing? **Tearing**(тиринг) - это визуальный эффект, который возникает при обновлении видеопамяти во время ее визуализации. Результатом будет то, что часть изображения будет представлять предыдущее изображение, а другая часть будет представлять новое.

Рис. 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.

Немного о GLFW. Вы, вероятно уже заметили, что эти буквы 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(да-да, это та самая библиотека работы с окнами) в классе с именем `Window`, позволяющим базовую параметризацию его характеристик (например, название и размер)(если вы чего-то не поняли, не волнуйтесь, очень скоро все станет понятно). Этот 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. Обратный вызов будет получать ширину и высоту в пикселях фреймбуфера (в данном примере — это область отображения). Если вам нужна ширина, высота фреймбуфера в координатах экрана, вы можете использовать метод `glfwSetWindowSizeCallback`.

Координаты экрана не обязательно 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);
});
```

Обратите внимание на лямбда выражение! Если вы не полностью его понимаете, воспользуйтесь гуглом для поиска информации по лямбда выражениям в java.

Давайте 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. Однако,

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 начинает наш `Thread`(поток), поэтому 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, такие как `Timer` (который будет предоставлять утилиты для вычисления прошедшего времени) и будут использоваться нашей логикой игрового цикла.

Наш класс `GameEngine` просто делегирует методы ввода и обновления экземпляру `IGameLogic`. В методе рендеринга он также делегирует(перекладывает часть работы на) 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 для в этой главе будет простым. Он просто увеличит, или уменьшит прозрачный(прим. перевод) цвет окна всякий раз, когда пользователь нажимает клавишу вверх / вниз. Метод `render` просто заполнит(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` мы получаем уведомление, когда окно было изменено для обновления окна просмотра, чтобы найти центр координат в центре окна.
Созданная нами иерархия классов поможет нам отделить наш игровой движок от кода конкретной игры. Хотя в данный момент это может показаться необходимым, нам нужно изолировать общие задачи, которые каждая игра будет использовать из общей логики, произведений искусства и ресурсов конкретной игры, чтобы повторно использовать наш игровой движок. В последующих главах нам нужно будет перестроить иерархию классов, поскольку наш игровой движок становится более сложным.

Часть 3. Некоторые особенности потоков

...