**Введение.**

OpenGL (Open Graphics Library) — спецификация, определяющая платформонезависимый (независимый от [языка программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) [программный интерфейс](https://ru.wikipedia.org/wiki/API) (API) для написания приложений, использующих двумерную и трёхмерную [компьютерную графику](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D0%BA%D0%B0).

На базовом уровне, OpenGL — это просто спецификация, то есть документ, описывающий набор функций и их точное поведение.

Работа с OpenGL на любимой JAVA с помощью библиотеки LWJGL.

**Подробнее про OpenGL:**

<https://ru.wikipedia.org/wiki/OpenGL>

**Языки программирования и библиотеки, позволяющие работать с OpenGL:**

<https://www.khronos.org/opengl/wiki/Language_bindings>

Если вы не знакомы с OpenGL, то прочтите эту статью. Она быстро даст вам минимальные основы, используемые в этой методичке:

<https://habr.com/ru/post/111175/>

Для более углубленного изучения OpenGL параллельно с чтением этой методички, можете читать серию уроков по OpenGL):

<http://pmg.org.ru/nehe/index.html>

В этих статьях работа с OpenGL реализовывается на языке СИ. Вам необязательно реализовывать это все на СИ, а лишь достаточно освоить, как они используют функционал OpenGL. Использование OpenGL на разных языках и библиотеках происходит практически одинаково. Практически все методы и классы OpenGL, поставляемые различными библиотеками, для программирования на различных языках – одинаковы (или с совсем маленькими различиями).

Т.к. по классике, программировать для OpenGL принято на СИ\СИ++, то и весь учебный материал по OpenGL, практически, написан с использованием СИ\СИ++.

Т.к. вы решили использовать джаву, то вам придется научиться изучать OpenGL по учебным материалам, написанным на СИ и портировать (переносить код с одного языка на другой) на джава библиотеку LWJGL.

Для этого лишь нужно понять, как осуществлять взаимодействие с OpenGL в LWJGL (некоторые отличия по сравнению с реализацией на СИ) и начать программировать. Этому вас и попытается научить данная методичка.

Также рекомендую искать обучающие ролики по OpenGL на youtube (где в основном будет использоваться СИ) и пытаться портировать код на LWJGL.

Эта методичка не научит вас OpenGL’у! Она предназначена для того, чтобы дать вам «старт», научить «как» использовать OpenGL на JAVA с помощью LWJGL!

Для примеров, в этой методичке будут продемонстрированы реализации самых классических и простых программ, что даст вам понимание как использовать LWJGL и подарит вам успешный «старт» в OpenGL на JAVA!

Я буду использовать JAVA 11.

**Кратко об LWJGL.**

Разработчики пытались сделать библиотеку компактной и простой, руководствуясь принципом: чем меньше способов сделать что-то, тем легче выучить единственный способ, который работает.

Так как эта библиотека ориентирована на разработку игр, то с ее помощью можно работать со звуком, получать доступ к различным устройствам ввода-вывода.

Такие библиотеки, как OpenGL, OpenCL и OpenAL, написаны на C и, следовательно, являются статическими по своей природе. Поскольку LWJGL спроектирован так, чтобы быть максимально простым и максимально приближенным к железу, его API почти полностью статичен и не является объектно-ориентированным.

Разработчики библиотеки сделали так, чтобы библиотека бросала исключение, когда аппаратное ускорение недоступно (ведь нет смысла смотреть на результаты при 5 fps).

LWJGL предлагает использовать так называемые буферы, вместо массивов т.к. буферы – самый быстрый способ передачи данных на уровень OpenGL из Java (оптимизация, быстрота).

Было уделено внимание и безопасности: нет необходимости работать с указателями, если использовать буферы. Буферы регулярно проверяются под капотом, что гарантирует нахождение значений в допустимых диапазонах.

**Более подробно про LWJGL:**

<https://ru.wikipedia.org/wiki/Lightweight_Java_Game_Library>

<http://wiki.lwjgl.org/wiki/About_LWJGL.html>

<http://wiki.lwjgl.org/wiki/General_FAQ.html>

**Игровые движки, библиотеки и фреймворки, использующие LWJGL и основанные на нем.**

Информация, представленная в этом разделе - это выборочная, переведенная информация. Более подробно, и об остальных движках можете узнать из источника.

**Источники:**

<http://wiki.lwjgl.org/wiki/Game_Engines_and_Libraries_Using_LWJGL.html>

**Несколько фреймворков для 2D разработки:**

**LibGDX | 2D, 3D**

LibGDX – библиотека, использующая OpenGL и позволяющая разрабатывать под 2D и 3D. Поддерживает написание под настольные ПК и Android. Она абстрагирует различия между написанием приложения под настольный ПК и Android на основе OpenGL. Это позволяет разрабатывать приложение полностью на рабочем ПК, а для его запуска на Android потребуется всего 6 строк кода. Основана на LWJGL версии 2.

**Slick2D | 2D**

Slick2D - это библиотека, предназначенная для создания быстрых, современных аппаратно-ускоренных 2D-игр. Ведется работа по его портированию на Android. Она предоставляет такие инструменты, как: редактор карт, редактор частиц, редактор растровых шрифтов и т.д.

**PlayN | 2D**

PlayN - это кроссплатформенная библиотека для написания игр, которые могут компилироваться в Desktop Java version, в браузерый HTML5, Android, iOS, Flash и другие системы. Библиотека PlayN является бесплатной, с открытым исходным кодом.

**JOGE | 2D**

[Java Opengl Game engine] – движок для создания 2D игр, написанный на Java. Он быстр и очень прост в освоении

**Также укажу несколько фреймворков для 3D разработки:**

**Ardor | 3D**

Ardor3D - это профессионально ориентированный 3D-движок с открытым исходным кодом на основе Java.

**Clyde | 3D**

Эта библиотека предоставляет средства для создания сетевых 3D игр на Java.

**JPCT | 3D**

JPCT предлагает средства для разработки игр и приложений для браузеров, android, настольных ОС.

**JMonkeyEngine | 3D**

Современный игровой движок, поддерживающий OpenGL версии 2.0 и выше.

**Въезжаем в LWJGL 2.**

LWJGL 2 использовался для создания многих игр, например Minecraft. Первая версия этой библиотеки вышла в 2011 году. Последнее обновление – в 2015 году.

Сейчас ее заменила переписанная версия LWJGL 3. LWJGL3 рассмотрим после краткого введения во вторую версию т.к. будет полезно кратко ознакомиться с LWJGL 2, если вы всерьез планируете изучать современную LWJGL 3 или фреймворк LibGDX (основанный на LWJGL 2).

**Официальный сайт LWJGL 2:**

<http://legacy.lwjgl.org>

**Официальная документация по LWJGL 2:**

<http://wiki.lwjgl.org>

Раздел по LWJGL 2 будет частично основываться на информации из официальной документации. В этой методичке будут переведены с английского некоторые ее разделы. В процессе чтения методички вы будете находить оставленные мною ссылки на информацию, которую я использовал (для более подробного ознакомления или проверки обновлений, если у вас будет желание и время).

**Создание первого LWJGL проекта в IntelliJ IDEA.**

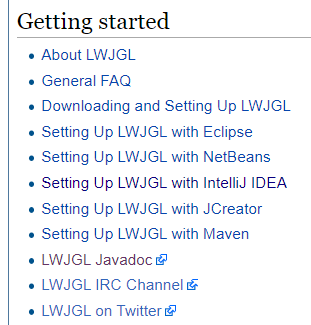
**Первый Дисплей.**

Информация, представленная здесь основана на документации lwjgl 2 по разделу «setting up LWJGL with IntelliJ IDEA»:

<http://wiki.lwjgl.org/wiki/Setting_Up_LWJGL_with_IntelliJ_IDEA.html>

Также, снабдил информацию дополнительными пояснениями, внес уточнения.

Далее, я буду использовать IntelliJ IDEA. Но если вы пользуетесь другими средами разработки, то вы можете найти гайды по ним на сайте документации, в разделе getting started:



Создадим программу с первым дисплеем – окном, в котором будут происходить все действия с графикой и демонстрация результатов программы (аналог традиционного hello world).

Пусть у вас уже есть установленная IDE Intellij IDEA (если нет, то не составит труда самим разобраться в ее установке).

Скачайте и распакуйте в какую-нибудь папку архив с библиотекой lwjgl 2 (я скачал версию 2.9.3) по ссылке:

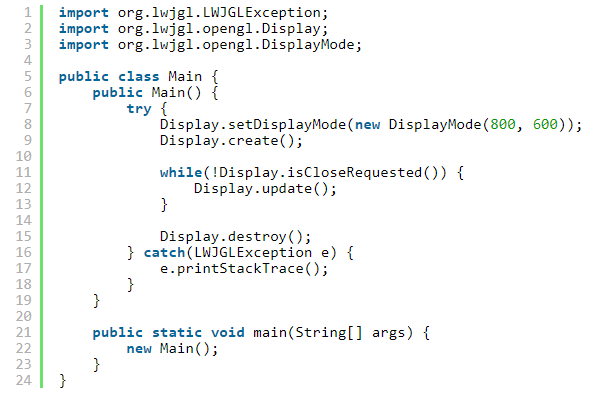
<http://legacy.lwjgl.org/download.php.html>

В IDE создайте новый проект под названием FirstDisplay.

В папке src проекта создайте java класс под названием Main.

Для того, чтобы быстро въехать в курс, я не буду использовать структурирование кода т.е. выносить все в отдельные функции\классы и т.д. т.к. программы, которые будут продемонстрированы в этой методичке достаточно просты, и их рабочий код будет удобно полностью помещать в метод main или же в конструктор класса).

В Main.java запишите (или скопируйте с сайта по ссылке выше) этот код:

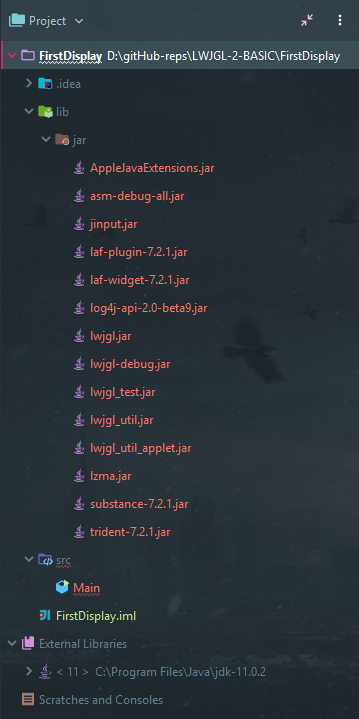


Обратите внимание, что документация рекомендует нам на первых этапах располагать рабочий код в конструкторе класса (в данном случае класса Main) **и оборачивать его в блок try-catch, чтобы отлавливать и выводить ошибки**!

По плану, среда должна подсвечивать вам некоторые сущности в коде как ошибки.

Это происходит потому что мы не подключили скачанную библиотеку и IDE не видит некоторые классы. Давайте подключим их.

Создайте папку lib в папке проекта (не в папке src! А на директорию выше!). Внутрь нее скопируйте папку jar из скачанного архива с библиотекой. Структура проекта должна выглядеть так:



**Замечание 1.**

Документация советует добавить только два .jar архива: lwjgl.jar и lwjgl\_util.jar.

Действительно, в основном мы не пользуемся функционалом всех .jar архивов из библиотеки.

Рекомендовано добавлять только те .jar архивы, которые требуются вашему приложению (чтобы не использовать лишнюю память или избегать некоторые конфликты с совместимостями, если ваше приложение будет сложным и замудренным).

Но т.к. наша цель быстро познакомиться с lwjgl без вникания в тонкости, то мы не будем соблюдать все нормы (морочить голову) и будем добавлять все .jar архивы в последующие проекты, во избежание разбирательства, какой .jar файл нам потребуется, а какой нет.

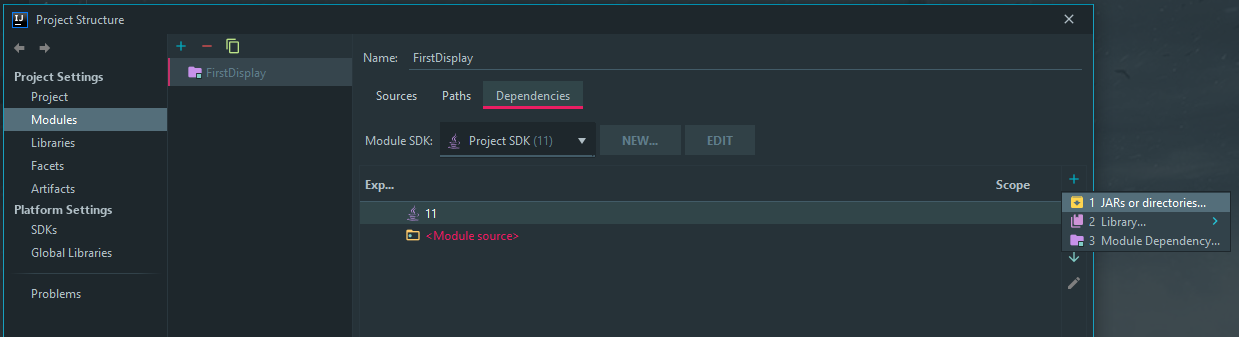
Подробную информацию с описанием этих .jar архивов найдете ниже.

**Замечание 2.**

Если у вас mac-ОС, тогда, скорее всего, нужно удалить AppleJavaExtensions.jar из вашего проекта. Т.к. у меня windows и я никогда не пользовался mac, то не могу сказать точно. Если что, пробуйте решить проблему сами! Подробную информацию про этот .jar архив и другие найдете ниже.

Теперь свяжем наш проект с добавленными в него файлами.

Идем в File -> Project Structure. В разделе Modules переходим во вкладку Dependencies. Кликаем на плюсик и выбираем “JARs or directories”:



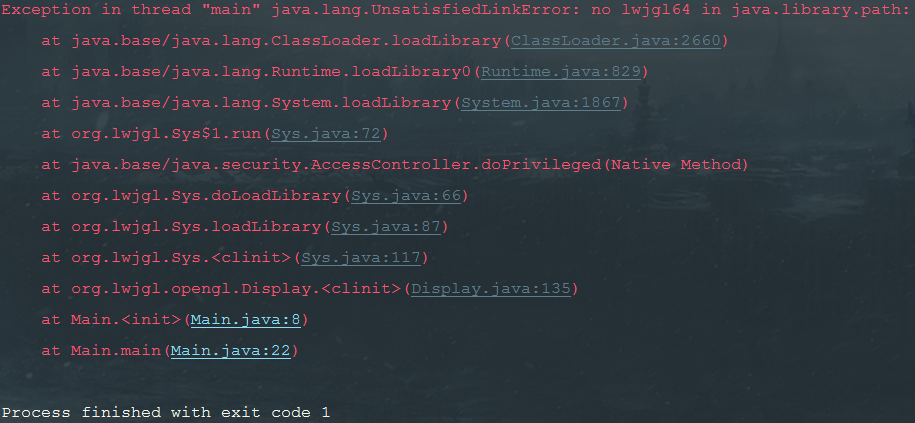
Выбираем все .jar архивы в папке lib проекта (чтобы выделить их все за раз, я зажимал ctrl). Жмем apply, ok.

Если вы все сделали правильно, то теперь сущности в коде перестали подсвечиваться красным (если нет, попробуйте перезапустить IDE).

Попробуем запустить проект (нажимаем Run “Main.main()” или зеленый треугольник).

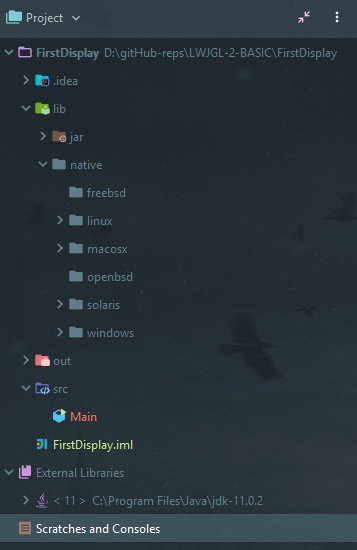
Получаем ошибку (исключение):

Exception in thread "main" java.lang.UnsatisfiedLinkError: no lwjgl(64) in java.library.path



Это происходит потому что мы не подключили части библиотеки, отвечающие за работу на конкретной операционной системе.

Создайте папку native в папке lib проекта. Внутрь нее скопируйте содержимое папки native из скачанного архива с библиотекой. Структура проекта должна выглядеть так:

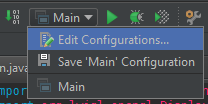


Теперь приложение можно будет запускать на перечисленных операционных системах.

Думаю, вы догадались, что можно было добавить папку только для вашей ОС.

Осталось установить параметры запуска (передавать JVM путь к нативным .dll файлам требуемой ОС при запуске приложения).

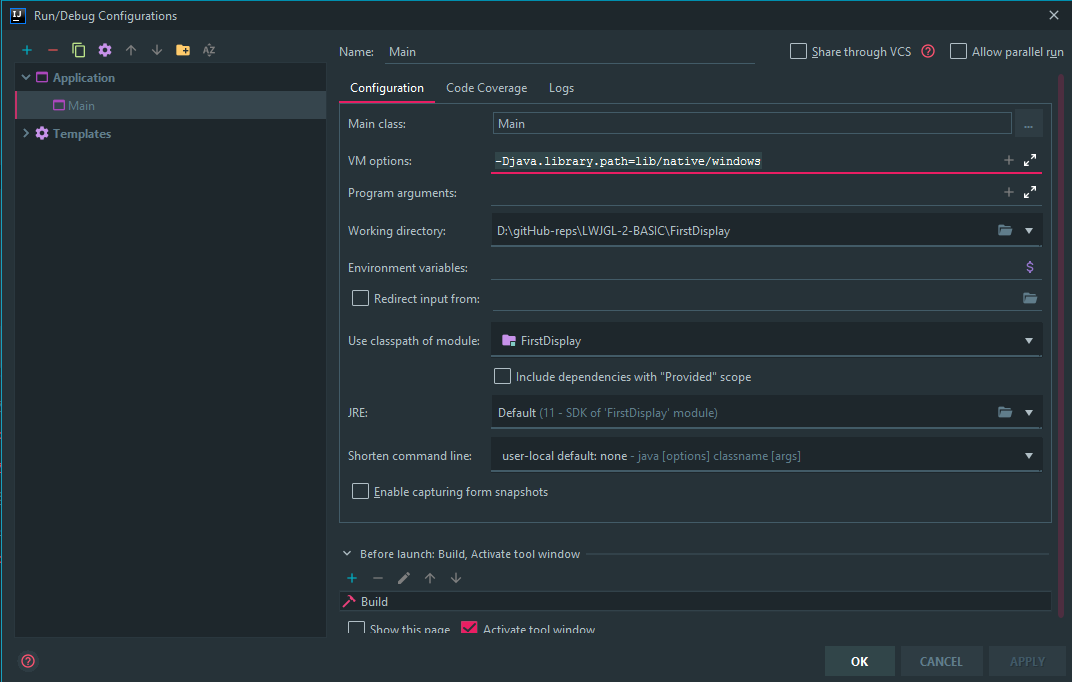
Идем сюда:



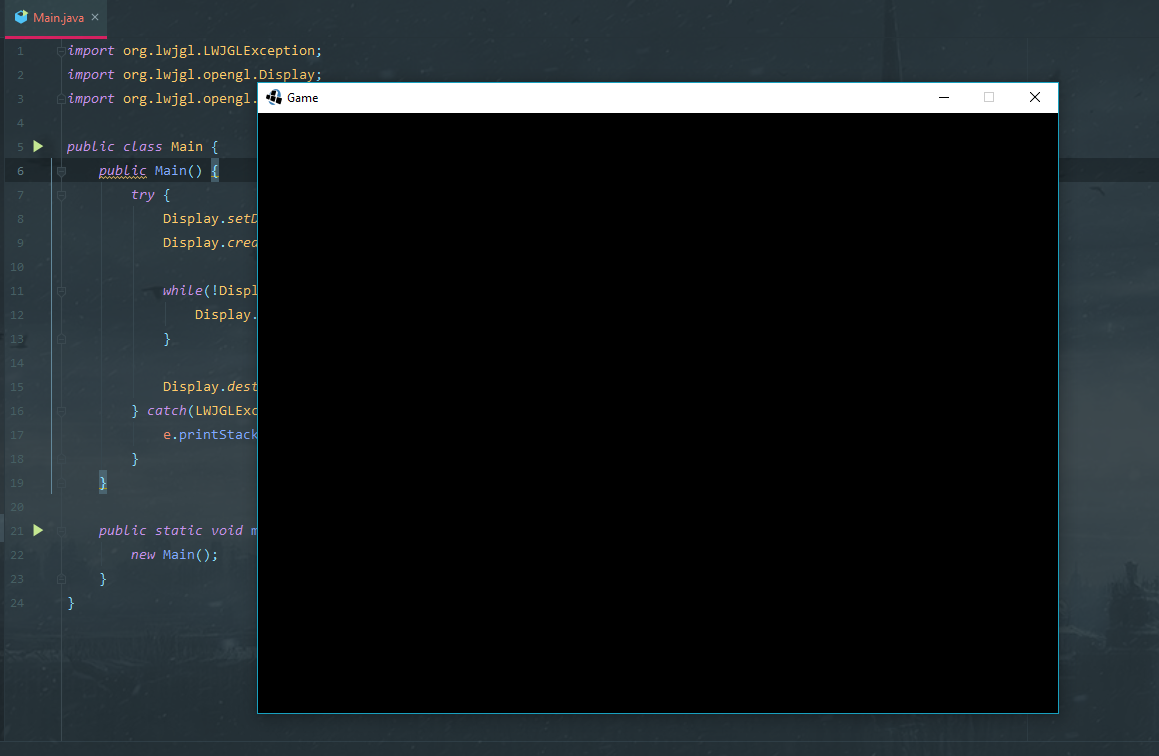
В поле VM options устанавливаем путь к нативным файлам вашей ОС (в моем случае windows):

-Djava.library.path=lib/native/windows

Жмем apply, ok.



Пробуем запустить приложение. Откроется наш первый дисплей:



**Пакет загрузки LWJGL 2 содержит много jar-архивов (папка jar в скачанном архиве). Краткое описание некоторых из них:**

1. **Lwjgl.jar** – содержит ядро LWJGL и является единственным, необходимым файлом для самого простого LWJGL приложения.
2. **Lwjgl\_util.jar** – содержит служебные классы, в том числе классы OpenGL GLU, различные классы векторов, матриц, карт объектов и тд.
3. **Lwjgl\_debug.jar** – содержит то же самое, что и lwjgl.jar, вдобавок предоставляет дополнительную отладочную информацию. Не рекомендуется загружать его одновременно с **lwjgl.jar** т.к. они содержат идентичные классы.
4. **Lwjgl\_test.jar** – содержит примеры написанных приложений для демонстрации (например, игра Space Invaders).
5. **Lwjgl\_util\_applet.jar** – содержит утилиту, используемую для развертывания LWJGL в браузере.
6. **Izma.jar** – необходим, если вы используете только izma-сжатые jar-файлы.
7. **Jinput.jar** – необходим, если вы собираетесь использовать игровые контроллеры (джойстики, несколько мышей и т.д.). Не требуется, если вы собираетесь использовать только клавиатуру и/или мышь.
8. **Asm.jar** - Это необязательный jar, который содержит библиотеку ASM и нужен только в том случае, если вы используете MappedObjects из lwjgl-util.jar.
9. **Asm-util.jar** – содержит утилиты для библиотеки ASM (необходим при использовании asm.jar).
10. **AppleJavaExtensions.jar** – необходим, если вы компилируете приложение не на mac-OS. Если у вас mac-OS, то его использовать нельзя.

LWJGL использует свою реализацию дисплея (окна), который удобно настраивать. За дисплей отвечает класс Display. Этот класс имеет три важных метода:

* **create() –** генерирует окно. Все методы дисплея, устанавливающие его параметры должны вызываться перед вызовом метода create()! После того, как окно создано, можно реализовывать логику построения графики – вызывать методы OpenGL.
* **update() –** для того, чтобы отрисовывать графику с течением времени, требуется бесконечный цикл. Этот метод вызывается в конце бесконечного цикла. Для понимания его работы нужно уяснить первую важную вещь. Процесс отрисовки графики (для понимания будем иметь ввиду видео) происходит отображением очень быстро меняющихся фиксированных изображений - «кадров». Делается это с помощью бесконечного цикла while: одна итерация цикла – отрисовывает один кадр. Например, за секунду может выполниться 20 итераций цикла т.е. отрисоваться 20 быстро сменяющихся изображений, а это будет похоже на видео. Чем больше итераций цикла (кадров) в секунду – тем плавнее будет динамическая графика. Уяснив первую важную вещь, переходим ко второй, более важной. **В LWJGL используется механизм двойного буфера.** Существует два буфера – передний и задний (они одинаковые, но передним считают тот, который хранит сгенерированное изображение, показываемое СЕЙЧАС, пока строится следующее изображение в заднем буфере). Бесконечный цикл, проделав первую итерацию генерирует первое изображение в первом буфере (в переднем). Это изображение показывается на дисплее в течении времени, пока проходит следующая итерация цикла – генерация нового изображения во втором (в заднем) буфере. Когда вторая итерация цикла подходит к концу, вызывается метод update(), который меняет передний и задний буферы местами – показывается последнее сгенерированное изображение (его буфер теперь считается передним). Предыдущее изображение очищается из буфера (вызовом специального метода очистки буферов. Об этом будет далее), который теперь является задним. Происходит генерация следующего изображения в заднем буфере. Этот процесс повторяется бесконечно, пока не будет реализована какая-нибудь логика выхода из цикла, или пока пользователь не закроет дисплей (или, в худшем случае, пока не произойдет исключение). Дополнительно к этому, метод update(), при своем вызове, запрашивает состояние клавиатуры или мыши (об этом в следующей главе).
* **destroy() –** уничтожает созданный дисплей (закроет его) и освобождает все занятые им ресурсы.

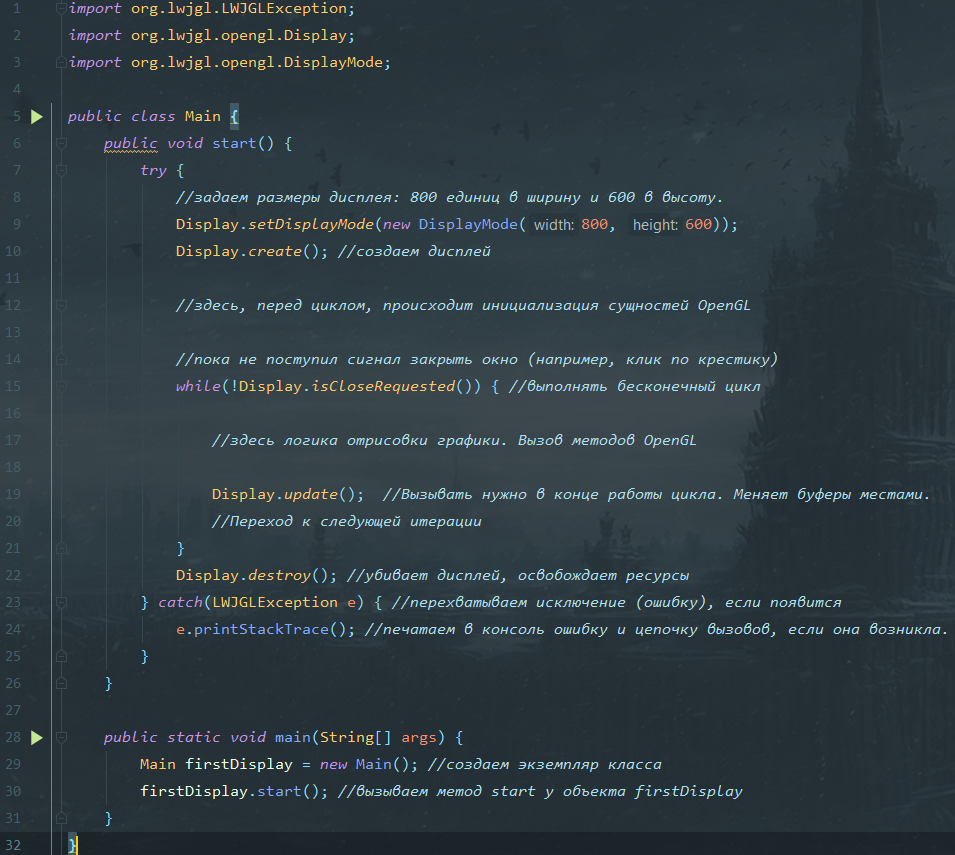
**Замечание.**

По классике, если использовать другие языки программирования и библиотеки для работы с OpenGL (например, С, С++), то там широко используется библиотека GLUT (вызов методов класса GLUT. Про GLUT читайте в следующем замечании, ниже), которая позволяет использовать либо один, либо два буфера (передний и задний).

**В LWJGL нет возможности использовать только один буфер.**

**В LWJGL используется два буфера.**

**Немного модифицируем нашу программу –** вынесем главную часть кода из конструктора в отдельный метод start. А конструктор, в котором он находился удалим.

****

Теперь выглядит человечнее, а не как рекомендовалось в документации.

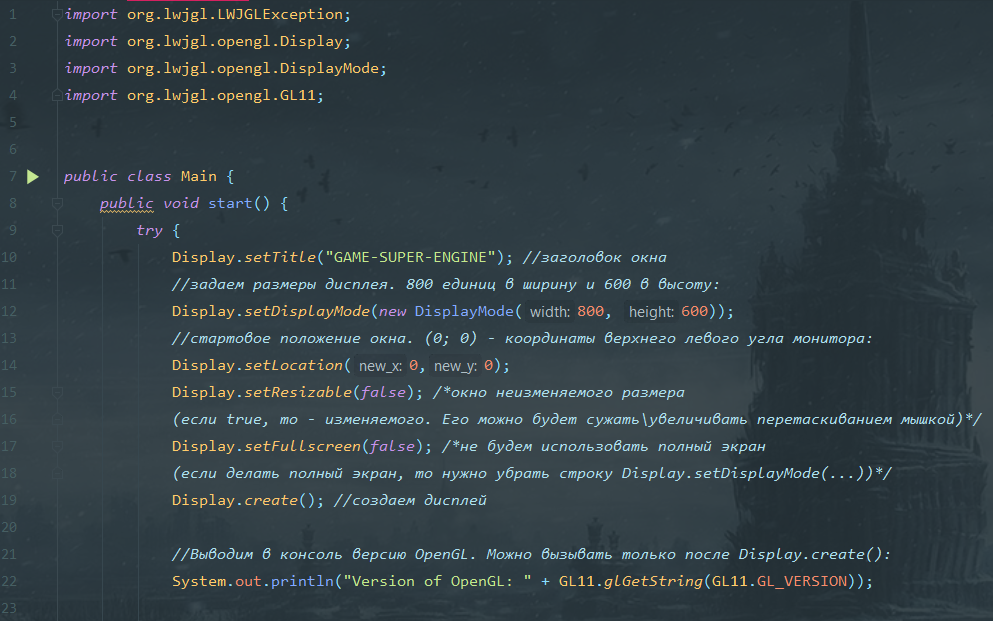
Обратите внимание на импорты. Я не буду навязчиво указывать в разборе каждой, новой программы, какой импорт нужно выполнить и за что каждый из них отвечает. Это настолько легко, что вы должны сами догадываться, плюс вам поможет IDE с автоматическим добавлением, при вызове какого-нибудь метода (в любой непонятной ситуации при вызове метода (если он подсвечивается красным намекая об ошибке) кликаем по нему и жмем alt+enter. IDE предложит импортировать нужные классы или исправить другие ошибки. Описание происходящего в коде, преимущественно, будет на скриншотах в виде комментариев, как на предыдущем скриншоте. Буду стараться каждый метод описывать кратко, а не по 5 строчек его тонкостей из документации.

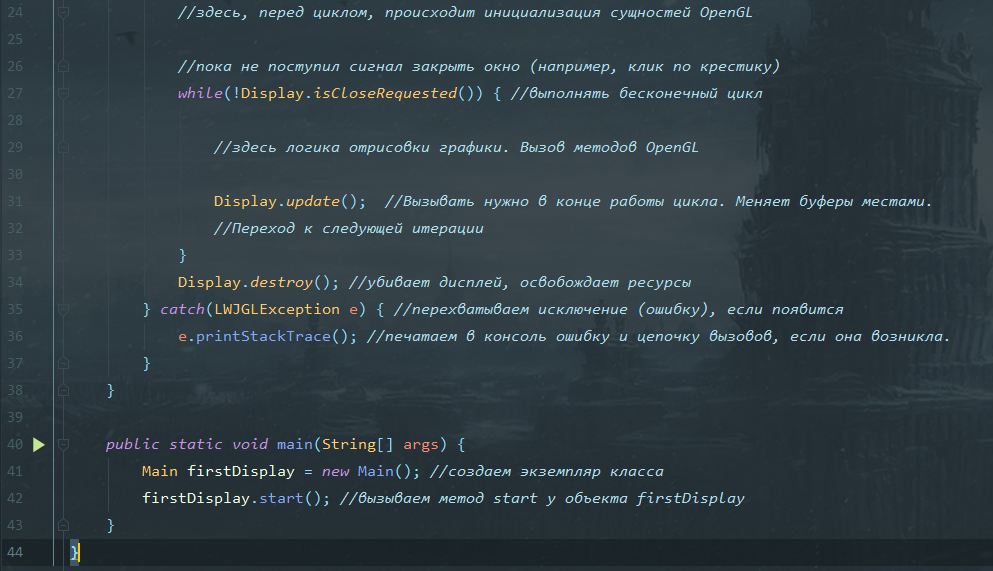
**Источники:**

<http://wiki.lwjgl.org/wiki/LWJGL_Basics_1_(The_Display).html>

**Настраиваем дисплей.**

Добавим кода:





Запустите программу.

Дисплей запустится в верхнем левом углу монитора и у него будет указанный заголовок. Попробуйте расширить мышкой его размеры. Поэкспериментируйте с кодом – попробуйте сделать дисплей полноэкранным. Не забудьте, что полноэкранный режим не будет работать, если заданы начальные размеры дисплея. Попробуйте изменять стартовое местоположение и размеры дисплея и оцените, сколько места на мониторе занимают каждые 100 единиц параметра. Попробуйте строку с выводом версии OpenGL перенести перед вызовом метода **create()** или в функцию main, посмотрите, что произойдет.

**Замечание.**

По классике, если использовать другие языки программирования и библиотеки для работы с OpenGL (например, С, С++), то там широко используется библиотека GLUT (вызов методов класса GLUT).

[OpenGL](https://ru.wikipedia.org/wiki/OpenGL) Utility Toolkit (GLUT) — библиотека утилит для приложений под [OpenGL](https://ru.wikipedia.org/wiki/OpenGL" \o "OpenGL), которая в основном отвечает за системный уровень операций ввода-вывода при работе с операционной системой. Из функций можно привести следующие: создание окна, управление окном, мониторинг за вводом с клавиатуры и событий мыши (про ввод – следующая глава).

**В LWJGL библиотека GLUT не используется**, а ее функционал заменили такие классы, как Display, Keyboard, Mouse и другие. Благодаря этому, работа с окнами и другим функционалом (для которого предназначается GLUT) в LWJGL намного проще и приятнее и реализована под приоритеты разработки игр.

**Дополнительная информация про GLUT:**

<https://ru.wikipedia.org/wiki/GLUT>

**Начинаем использовать OpenGL в LWJGL 2.**

**Замечание.**

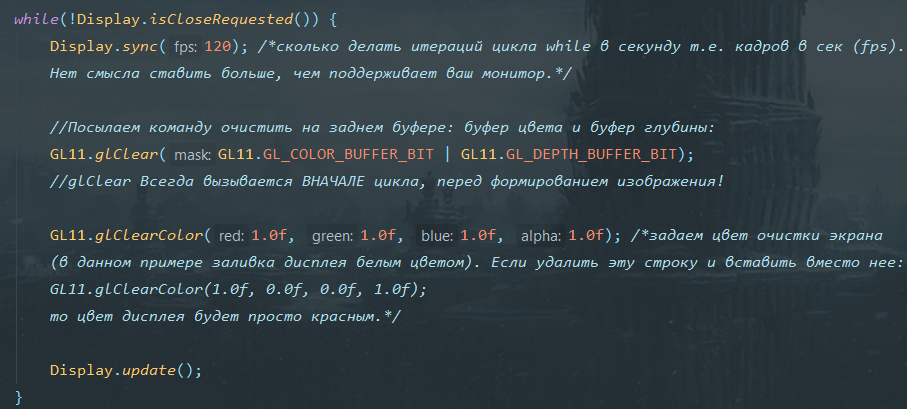
Если вы все еще не прочитали

<https://habr.com/ru/post/111175/>

статью на 10 мин (про основы OpenGL), то сейчас самое время.

Настоятельно рекомендую ее прочитать, чтобы дальше было легче!

**Внесем изменения в цикл while:**



Запустите программу.

Если вы все сделали правильно, то теперь цвет дисплея будет белым.

Заметьте, что в LWJGL вызов OpenGL методов и констант, в аргументах метода glClear (они, кстати, описываются в статье, которую вы прочитали по ссылке выше) осуществляется добавлением «GL11.» перед вызовом метода, а именно происходит обращение к классу GL11 (т.к. вызываемые методы поставляются классом GL11).

В LWJGL методы OpenGL’а разбиты по классам, например, будет встречаться приписка «GL10», «GL15», «GL20», «GL30» и другие. Разработчики LWJGL сгруппировали методы по классам, по определенным признакам.

**Задание.**

Попробуйте сами поискать в интернете информацию об этих классах и разобраться, по какому признаку они их группировали (кажется это связано с версиями OpenGL, но я не уверен), а также посмотрите, какие методы хранит в себе каждый класс.

Информацию можно найти в документации **по LWJGL 2 (не запутайтесь с 3ьей версией!)**.

Составьте мини-табличку (.jpeg изображение) по этим классам, с их описанием и принципами группировок методов на русском языке, и вышлите их мне на почту [**varenitsiam@gmail.com**](mailto:varenitsiam@gmail.com)или свяжитесь со мной на гитхабе**:**

[**https://github.com/VarenytsiaMykhailo**](https://github.com/VarenytsiaMykhailo)

Возможно, на моем гитхабе, я сделаю публичным репозиторий с этой методичкой, куда буду выкладывать обновления методички с исправлениями найденных вами ошибок и недочетов. Таблицу можете загрузить пулл реквестом в этот репозиторий, а о найденных ошибках сообщать созданием топика во вкладке Issues (проблемы).

Лучшую таблицу буду добавлять в обновленную методичку, с указанием автора, который ее сделал)

Кстати, можно сделать так, чтобы не писать постоянно “GL11” и т.д. перед вызовами методов. Для этого нужно статически импортировать соответствующие классы:



По поводу метода **Display.sync()**.

В первую очередь, чтобы он работал, его нужно вызывать в бесконечном цикле рендеринга. Попробуйте поэкспериментировать и вынести его наверх, где настраиваются параметры дисплея.

Т.к. почти у всех мониторов частота 60 или 120 герц, то я буду выставлять значение 120. Погуглите, как узнать частоту вашего монитора.

При настройках параметров дисплея можно использовать метод **Display.setVSyncEnabled(true).** Он включит вертикальную синхронизацию – сделает так, чтобы fps не поднимался выше допустимых значений вашего монитора.

Ведь ваш монитор может выдавать максимум 60 fps, а видеокарта будет способна работать на 200+ fps, что приведет к «трате нагрузки видеокарты на ветер» и лишнему перегреву. Метод **Display.setVSyncEnabled(true)** поможет не грузиться вашей видеокарте сверхлимитом.

В таком случае нет смысла использовать **Display.sync()**, если вы не хотите урезать fps ниже возможностей вашего монитора.

Подробнее про вертикальную синхронизацию:

<https://ru.wikipedia.org/wiki/Вертикальная_синхронизация>

По поводу метода **glClear()**: как указано в статье, которую вы прочитали, этот метод может принимать следующие аргументы, разделенные знаком побитового или «|»:

Возможные аргументы:

* GL\_COLOR\_BUFFER\_BIT — для очистки буфера цвета
* GL\_DEPTH\_BUFFER\_BIT — для очистки буфера глубины
* GL\_ACCUM\_BUFFER\_BIT — для очистки буфера накопления
* GL\_STENCIL\_BUFFER\_BIT — для очистки трафаретного буфера

Чтобы не затуманивать мозги и т.к. это тема OpenGL, пока не буду говорить для чего нужен каждый из этих аргументов, пока он не понадобится, но рекомендую разобраться с ними самостоятельно.

Метод **glClearColor()** в качестве аргументов принимает цвет в формате RGB. 4ый аргумент – прозрачность: 1.0f – не прозрачный. Значения указываются от 0 до 1. Если каким-нибудь аргументом передать, например, значение 33.0f, то оно будет считаться как 1.0f, а отрицательные значения считаются как 0.0f!

Кстати, кто не знает, в JAVA приписка f к числу – означает, что оно типа float. Если не приписать f и указать просто 1.0, то это число будет считаться типом double.

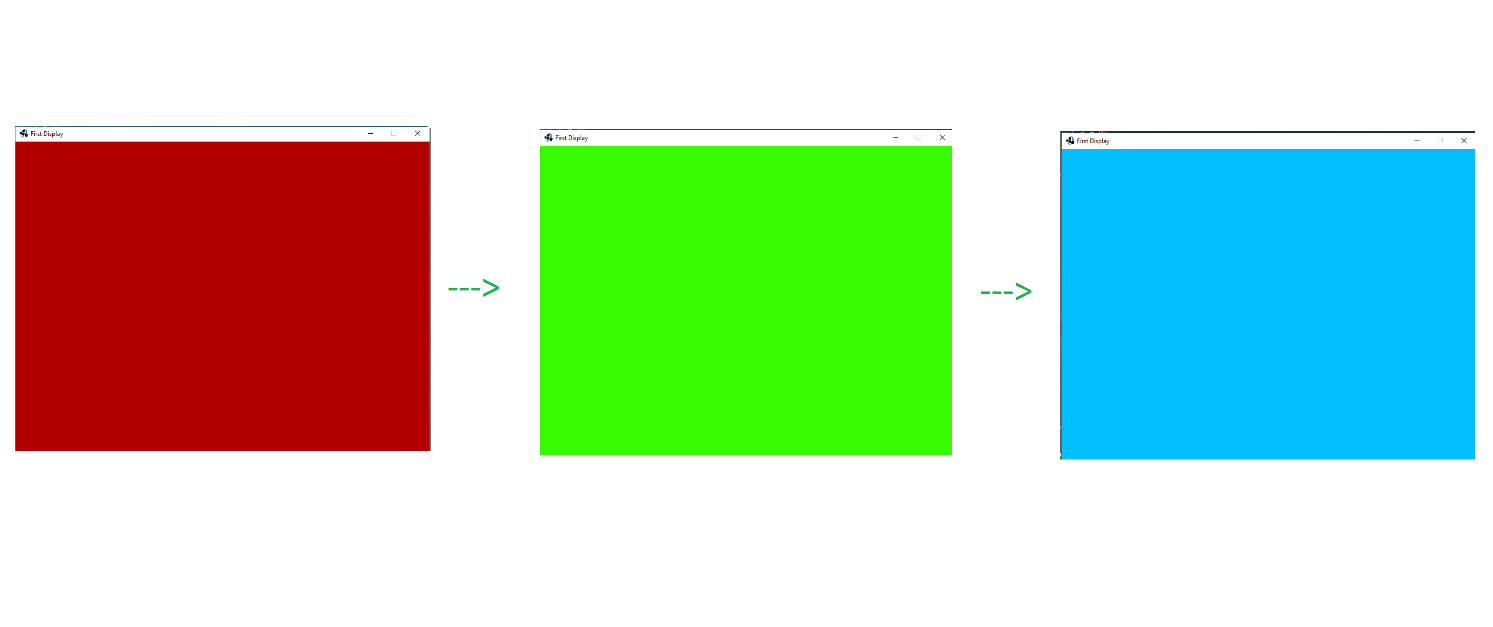
Не буду здесь рассказывать, по какому принципу формируются цвета с помощью трех значений, разберитесь с этим сами (гуглите по теме RGB прямо сейчас, потому что дальше будет фокус-покус)!

**Изменим метод start() нашей программы:**



Запустите программу.

Если вы все сделали правильно, то теперь цвет дисплея будет плавно меняться с течением времени, плавно перетекая с одного цвета в другой:



**Как это работает?**

Создаем 3 переменные – r, g, b (красный, зеленый, голубой, которые формируют определенный цвет). Вначале каждого цикла while каждая из этих переменных увеличивается, на маленькое значение (причем эти значения различны для всех 3ех переменных. Попробуйте потом поэкспериментировать и увеличивать переменных на одинаковое число).

Затем эти переменные передаем в метод очистки экрана цветом, оборачивая их в синус. Получается, что для каждой итерации цикла в этот метод передается плавно-изменяющийся цвет.

Но зачем оборачивать r, g, b в синус? Дело в том, что из-за увеличения r, g, b на каждой итерации цикла, их значения перевалят единицу (они будут считаться как 1.0f, 1.0f, 1.0f), а потом вовсе достигнут переполнения и станут отрицательными. Оборачивание в синус решает эту проблему – синус принимает любые числа (как большие, так и отрицательные) и возвращает значение от -1 до 1.

**Обязательно попробуйте поэкспериментировать:**

1. Попробуйте не оборачивать r, g, b в синус т.е. передать их напрямую и проанализируйте результат.
2. Попробуйте увеличивать значения r, g, b вначале цикла как на маленькие числа, так и на числа на порядок больше (например, добавлять 0.2, 0.6, 0.9) и оцените быстроту изменения цвета.
3. Попробуйте в вызов метода **Display.sync()** передавать очень низкие значения: 1, 2, 3, 4, 5… и оцените быстроту изменения цвета т.е. смены изображений (вы фактически можете сделать так, чтобы в секунду происходила только одна итерация цикла while).

**Ввод с мыши и клавиатуры в LWJGL 2. Input.**

Информация, представленная здесь – выполненный мною перевод документации lwjgl 2 по разделу «input».

**Источники:**

<http://wiki.lwjgl.org/wiki/LWJGL_Basics_2_(Input).html>

Также, где-то снабдил информацию дополнительными пояснениями, внес уточнения.

LWJGL может обрабатывать ввод с помощью собственных классов **Keyboard** и **Mouse**.

Для получения обновлений нажатий клавиш или клика требуется обращение (с помощью метода poll()) к классам **Keyboard** и **Mouse**.

В LWJGL это делается автоматически при вызове метода Display.update().

**Мышь.**

Положение курсора мыши на дисплее можно узнать, вызвав методы:

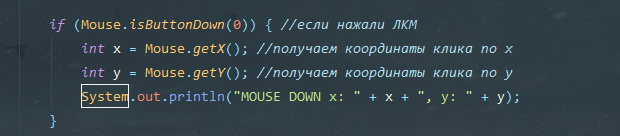


Позиция курсора с координатами (0, 0) соответствует левому, нижнему углу дисплея (в традиционном стиле OpenGL).

Метод **Mouse.isButtonDown(int button)** вернет true, если кнопка мыши нажата, или false, если нет. Кнопки мыши пронумерованы от 0 до buttonCount.



Пример:



**Клавиатура.**

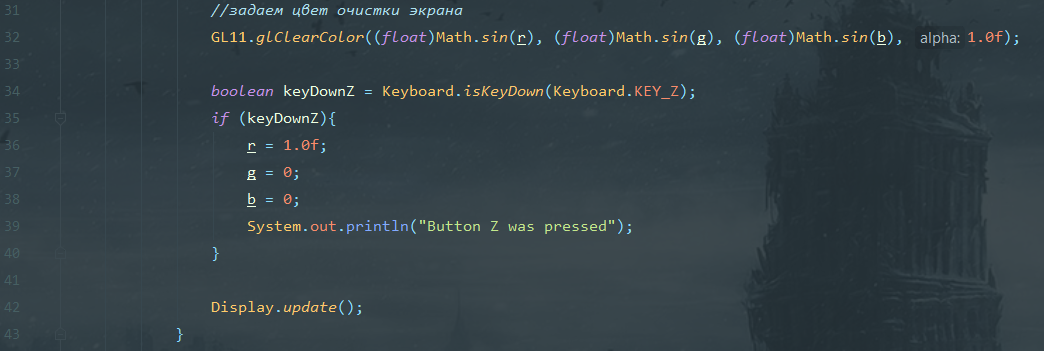
Метод **Keyboard.isKeyDown(int key)** вернет true, если соответствующая клавиша нажата, или false, если нет.



Ссылку на полный список клавиш клавиатуры найдете ниже, в конце главы.

**Внесем изменения в предыдущую программу-радугу по переливанию цвета для демонстрации разобранного метода клавиатуры.**

Добавьте часть кода:



Запустите программу.

Теперь при нажатии клавиши «z» цвет дисплея будет сбрасываться на красный и продолжать плавное переливание цвета.

Как упражнение, дополнительно добавьте такую же обработку для клавиш «x», «c», которые сбрасывают цвет дисплея на зеленый и синий соответственно.

**Буфер событий.**

Использование методов **isKeyDown()** или **isButtonDown()** может работать хорошо, однако можно пропустить (не зафиксировать) нажатие клавиш или кнопок мыши, если запрос о нажатии (вызов метода **poll()** под капотом) выполняется долго.

Напомню, что метод **poll()**, который запрашивает состояние клавиатуры или мыши вызывается каждый раз при вызове **Display.update().**

Чтобы решить эту проблему предлагается использовать буфер событий.

Этот буфер содержит все события от клавиш и кнопок мыши, полученных дисплеем.

Доступ к событиям в буфере осуществляется с помощью методов **getEvent\*()**, где вместо звездочки подставляются слова, характеризующие события, например, **Keyboard.getEventKey().**

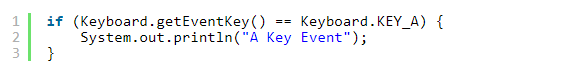
Метод **next()** используется для перебирания буфера событий.

Каждый раз, когда вызывается метод **next()**, он вытаскивает следующее событие из буфера в качестве клавиши/кнопки мыши. После перебора всех событий (клавиш/кнопок мыши) он возвращает false.

Например, буфер клавиатуры можно перебирать так:

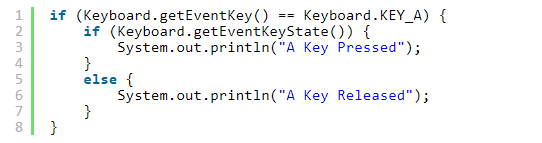


Метод **Keyboard.getEventKey()** возвращает клавишу, которая сгенерировала событие.



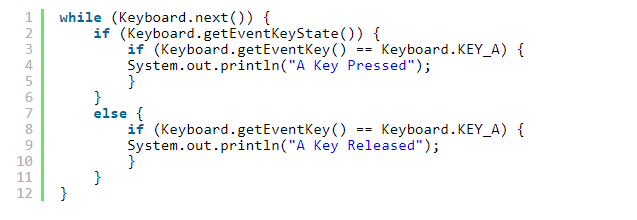
Метод **Keyboard.getEventState()** вернет true, если клавиша события была нажата, или false, если она была отпущена.

Используя два вышеупомянутых метода, можно определить, какая клавиша сгенерировала событие и была ли она нажата или отпущена:



Таким способом можно определить зажата ли клавиша сейчас.

В приведенном ниже примере проверяются события для клавиши A, хранящиеся в буфере.



Так, если запустить этот код в бесконечном цикле рендеринга, то, если мы

1. Зажмем клавишу А, то в консоль выведется «A Key Pressed» (один раз, пока мы ее не отпустим). Далее, спустя 5 сек, если мы отпустим ее, то в консоль выведется «A Key Released».
2. Если клавишу A нажать и сразу же опустить, то почти мгновенно последовательно выведется

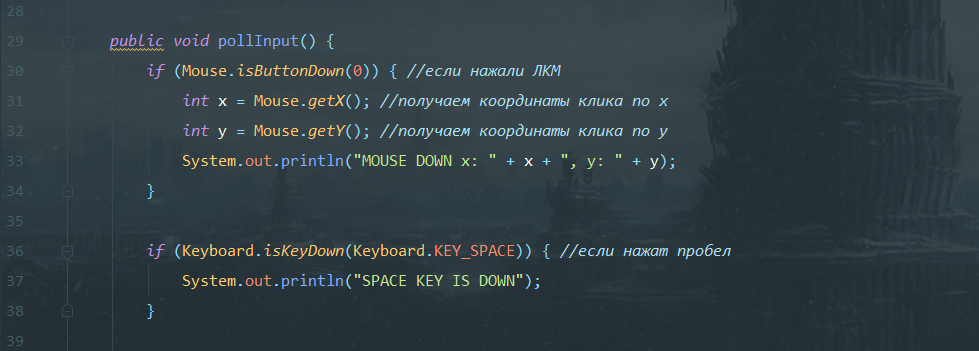
«A Key Pressed

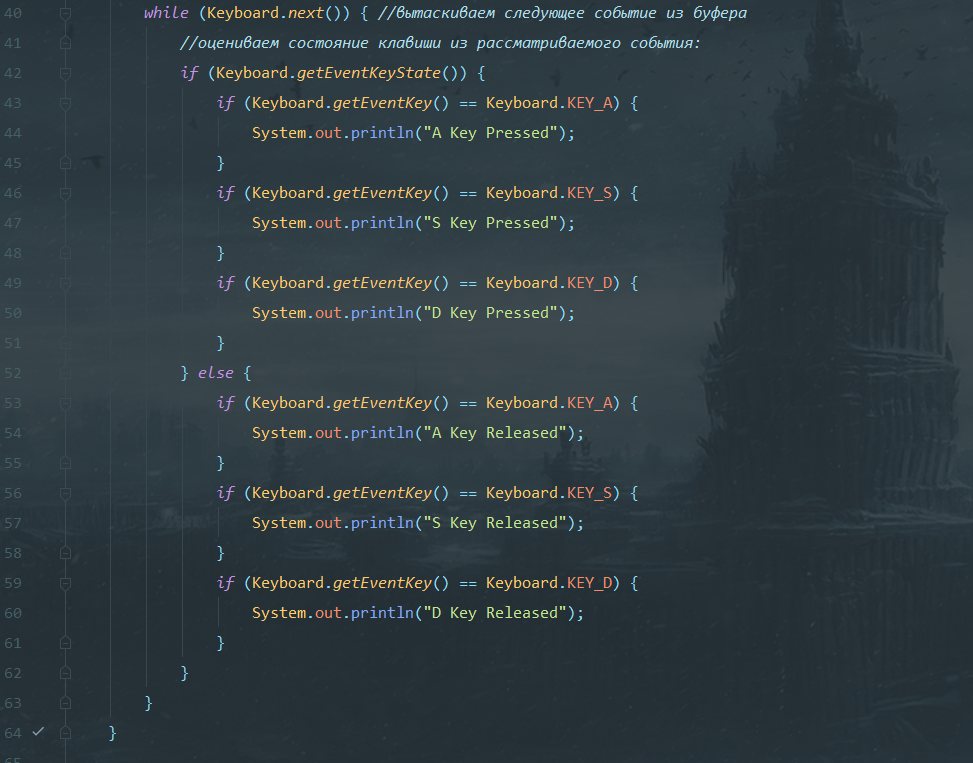
A Key Released»

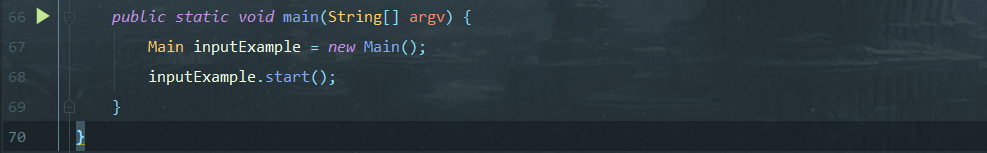
1. Если в результате выполнения кода вывелось только «A Key Pressed», то это означает, что клавиша все еще зажата. Можно в положительную ветку if написать код, который выполнит какую-нибудь логику, если спустя n сек времени «A Key Released» не появился (т.е. клавишу зажали надолго).

**Пример из документации с моими комментариями.**









В этом примере я обернул в try-catch весь код метода **start().**

Попробуйте запустить этот пример и покликать ЛКМ по дисплею. Понажимайте «пробел», клавиши «a», «s», «d» в разных порядках и одновременно. Проанализируйте выводимый результат в консоль.

**Дополнительные ссылки на документацию.**

**Полный список клавиш клавиатуры можно найти здесь:**

Здесь собраны все константы из всех классов, которые используются в LWJGL (в том числе и константы для клавиатуры, с их числовыми значениями):

<http://legacy.lwjgl.org/javadoc/constant-values.html>

Или погугля, введя в поиск что-то вроде «javadoc list of Keyboard keys lwjgl», если ссылки не будут работать. Список там огромный, на пару десятков страниц, поэтому не стал его сюда добавлять.

**Документация по классу Keyboard** (здесь вы также найдете константы клавиш, а также дополнительные методы этого класса):

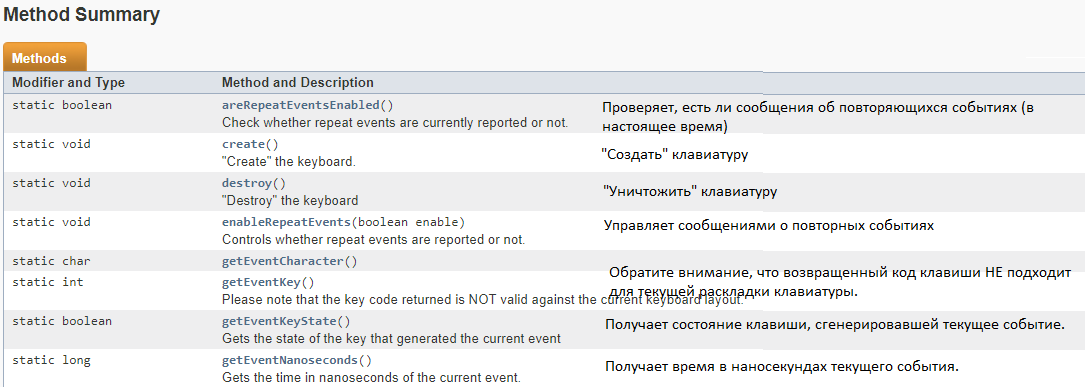
<http://legacy.lwjgl.org/javadoc/org/lwjgl/input/Keyboard.html>

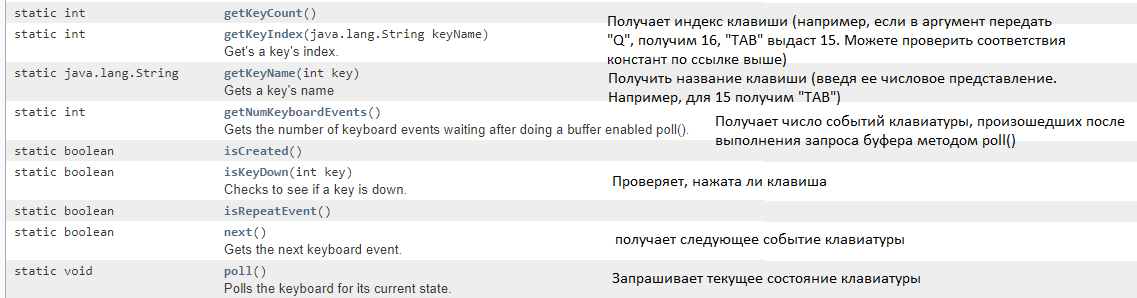
**Документация по классу Mouse** (здесь вы найдете константы, а также дополнительные методы этого класса):

<http://legacy.lwjgl.org/javadoc/org/lwjgl/input/Mouse.html>

Методы класса Keyboard и Mouse, пожалуй, переведу (чтобы были под рукой).

**Keyboard:**





**Mouse:**



