Лабораторные:

1. Заставить работать
2. Нарисовать кубик VBO
3. Заставить вращаться
4. Остановиться/Вращаться при нажатии на пробел
5. Сделать текстуру
6. Простой шейдер

Презентацию

glfwGetTime()

1. Функции перерисовки

2. Размеры окна

3. Ивенты

4. Вершиины, Полигоны, ВБО - В Памяти

5. Текстуры, сэмпл

6. Преобразования Глобальные Локальные

7. uniform – однородная, varying - варьируя

Следующий пример демонстрирует создание окна на первом мониторе с заданным разрешением если не FULL\_SCREEN и полноэкранный режим в противном случае ☺

if (!FULL\_SCREEN) {

window = glfwCreateWindow(640, 480, "Lesson 01", NULL, NULL);

} else {

GLFWmonitor\* monitor = glfwGetPrimaryMonitor();

const GLFWvidmode\* mode = glfwGetVideoMode(monitor);

glfwWindowHint(GLFW\_RED\_BITS, mode->redBits);

glfwWindowHint(GLFW\_GREEN\_BITS, mode->greenBits);

glfwWindowHint(GLFW\_BLUE\_BITS, mode->blueBits);

glfwWindowHint(GLFW\_REFRESH\_RATE, mode->refreshRate);

window = glfwCreateWindow(mode->width, mode->height, "Lesson 01", monitor, NULL);

}

Библиотека GLFW позволяет устанавливать различные *callback*-функции

Событие изменения размера заданного окна

glfwSetFramebufferSizeCallback (window, resize);

void resize (GLFWwindow \* window, int w, int h)

{

. . .

}

В GLFW есть всего несколько функций, которые работают с сообщениями. Первая из них - **glfwPollEvents** - берет сообщения из очереди и обрабатывает их. Если в очереди нет ни одного сообщения, то управление немедленно возвращается. При этом основной цикл обработки сообщений обычно выглядит следующим образом:

while (!glfwWindowShouldClose(window))

{

Render ();

glfwSwapBuffers ();

glfwPollEvents ();

}

Подобно библиотеке GLUT можно задать функции-обработчики для ряда событий, таких как изменение размера окна, ввод с клавиатуры, события от мыши и т.п. Весь ввод с клавиатуры GLFW делит на две категории - *key events* и *character events*. Первые связаны с физическими клавишами клавиатуры (нажатие и отпускание клавиш), а вторые - с генерацией символов клавиатурой (подобно тому как это сделано в Windows).

Функция- обработчик нажатий и отпусканий клавишь имеет следующий вид:

void keyCallback (GLFWwindow \* window, int key, int scancode, int action, int mode)

{

. . .

}

Параметр *key* задает клавишу на клавиатуре, например GLFW\_KEY\_SPACE или GLFW\_KEY\_ESCAPE. Если соответствующая клавиша неизвестна GLFW, то этот параметр будет равен GLFW\_KEY\_UNKNOWN.

Параметр *action* определяет произошедшее с клавишей событие и равен GLFW\_PRESS, GLFW\_RELEASE или GLFW\_REPEAT.

Параметр *scancode* идентифицирующее клавишу на клавиатуре, его значение зависит от платформы.

Параметр *mode* - это набор битовых флагов, задающих состояние клавиш-модификторов. Этими клавишами являются GLFW\_MOD\_SHIFT, GLFW\_MOD\_CONTROL, GLFW\_MOD\_ALT и GLFW\_MOD\_SUPER.

Обработчик *key events* задается при помощи функции **glfwSetKeyCallback**:

glfwSetKeyCallback ( window, keyCallback );

**Vertex Buffer Object** (VBO) — особенность [OpenGL](https://ru.wikipedia.org/wiki/OpenGL), обеспечивающая методы выгрузки данных ([вершин](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D1%80%D1%88%D0%B8%D0%BD%D0%B0_%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE%D1%83%D0%B3%D0%BE%D0%BB%D1%8C%D0%BD%D0%B8%D0%BA%D0%B0), [вектора нормали](https://ru.wikipedia.org/wiki/%D0%9D%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D1%8C), цветов, и так далее.) в видеоустройство для не оперативного режима рендеринга. VBO дали существенный прирост производительности над непосредственным режимом визуализации, в первую очередь, потому что данные находятся в памяти видеоустройства, а не в оперативной памяти и поэтому она может быть отрендерена непосредственно видеоустройством.

float vertices[] = {

-0.5f, -0.5f, -0.5f, 0.0f, 0.0f,

0.5f, -0.5f, -0.5f, 1.0f, 0.0f,

0.5f, 0.5f, -0.5f, 1.0f, 1.0f,

0.5f, 0.5f, -0.5f, 1.0f, 1.0f,

-0.5f, 0.5f, -0.5f, 0.0f, 1.0f,

-0.5f, -0.5f, -0.5f, 0.0f, 0.0f,

…

};

Процедура [glGenBuffers](http://docs.gl/gl3/glGenBuffers) возвращает заданное количество неиспользуемых в настоящее время айдишников VBO. Как бы генерирует id

Далее процедурой [glBindBuffer](http://docs.gl/gl3/glBindBuffer) мы связываем (bind) полученное на предыдущем шаге имя с определенной точкой связывания (binding point). Мы хотим сохранить в буфере координаты вершин, и должны использовать для этого точку связывания GL\_ARRAY\_BUFFER. Обратите внимание, здесь снова ни о каком создании буферов речи пока не идет. Мне лично проще всего думать о glBindBuffer, как о команде «ок, сейчас мы будем что-то делать вот с этим VBO».

Наконец, при вызове [glBufferData](http://docs.gl/gl3/glBufferData) происходит выделение памяти и загрузка в нее наших данных. Последний аргумент процедуры называется usage и задает предполагаемый паттерн использования буфера. Часть STATIC говорит о том, что мы не собираемся модифицировать данные, а часть DRAW — о том, что данные будут использованы для отрисовки чего-то. Этот аргумент не приводит к каким-то действительным ограничениям на использование буфера. Но он является подсказкой для реализации OpenGL, и может существенно влиять на производительность приложения. Так что, лучше указывать в нем что-то правдоподобное. Заметьте, что первым аргументом указывается binding point, а не конкретный VBO. То есть, выбор самого VBO был сделан на предыдущем шаге.

Когда VBO больше не нужен, его можно удалить следующим образом:

glDeleteBuffers(1, &vbo);

Итак, мы сохранили данные в буфере. Зачем, спрашивается, нужны еще какие-то VAO?

### VAO

[Vertex Arrays Object (VAO)](https://www.opengl.org/wiki/Vertex_Specification#Vertex_Array_Object) — это такая штука, которая говорит OpenGL, какую часть VBO следует использовать в последующих командах. Чтобы было понятнее, представьте, что 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.

glGenVertexArrays(1, &VAO);

glGenBuffers(1, &VBO);

glBindVertexArray(VAO);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO);

glBufferData(GL\_ARRAY\_BUFFER, sizeof (vertices), vertices, GL\_STATIC\_DRAW);

// position attribute

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 5 \* sizeof (float), (void\*) 0);

glEnableVertexAttribArray(0);

// texture coord attribute

glVertexAttribPointer(1, 2, GL\_FLOAT, GL\_FALSE, 5 \* sizeof (float), (void\*) (3 \* sizeof (float)));

glEnableVertexAttribArray(1);

GLM (OpenGL Mathematics — математика для OpenGL) — библиотека для OpenGL, предоставляющая программисту на C++ структуры и функции, позволяющие использовать данные для OpenGL[источник не указан 1408 дней].

Одна из особенностей GLM состоит в том, что его реализация основана на спецификации GLSL (OpenGL Shading Language)[1].

Исходный код GLM распространяется под MIT License.

При использовании glm :: translate (X, vec3) , вы умножаете

X \* glm :: translate (Identity, vec3)

Это значит сначала перевести, затем X

**glm::rotate – поворот на угол относительно вектора**

**GLSL** (OpenGL Shading Language, Graphics Library Shader Language) — язык высокого уровня для [программирования](https://ru.wikipedia.org/wiki/%D0%9F%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%D0%B5) [шейдеров](https://ru.wikipedia.org/wiki/%D0%A8%D0%B5%D0%B9%D0%B4%D0%B5%D1%80). Разработан для выполнения математики, которая обычно требуется для выполнения [растеризации](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D1%81%D1%82%D0%B5%D1%80%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F) графики. Синтаксис языка базируется на языке программирования [ANSI C](https://ru.wikipedia.org/wiki/ANSI_C), однако, из-за его специфической направленности, из него были исключены многие возможности, для упрощения языка и повышения производительности. В язык включены дополнительные функции и типы данных, например для работы с векторами и матрицами.

Основное преимущество GLSL перед другими шейдерными языками — переносимость кода между платформами и ОС.

Язык **GLSL** используется в [OpenGL](https://ru.wikipedia.org/wiki/OpenGL), в [OpenGL ES](https://ru.wikipedia.org/wiki/OpenGL_ES) и [WebGL](https://ru.wikipedia.org/wiki/WebGL) используется язык **GLSL ES** (OpenGL ES Shading Language).

<https://learnopengl.com/>

<https://www.glfw.org/docs/latest/window_guide.html#window_size>

<https://github.com/JoeyDeVries/LearnOpenGL>

<https://www.youtube.com/watch?v=WMiggUPst-Q>

<https://github.com/KnightDanila/GraphicProjects_OpenGL_Shaders_GLSL>