

Компьютерная графика

Практика 1: Рисуем треугольник

2021

Задание 1

- ▶ Пишем вспомогательную функцию для создания шейдера

```
GLuint create_shader(GLenum shader_type,  
    const char * shader_source)
```

Задание 1

- ▶ Пишем вспомогательную функцию для создания шейдера

```
GLuint create_shader(GLenum shader_type,  
                    const char * shader_source)
```

- ▶ `glCreateShader`
- ▶ `glShaderSource`
- ▶ `glCompileShader`

Задание 1

- ▶ Пишем вспомогательную функцию для создания шейдера

```
GLuint create_shader(GLenum shader_type,  
                    const char * shader_source)
```

- ▶ `glCreateShader`
 - ▶ `glShaderSource`
 - ▶ `glCompileShader`
- ▶ Функция должна бросать исключение (например, `std::runtime_error`), если компиляция шейдера провалилась
 - ▶ `glGetShaderiv`

Задание 1

- ▶ Пишем вспомогательную функцию для создания шейдера

```
GLuint create_shader(GLenum shader_type,
                    const char * shader_source)
```

 - ▶ `glCreateShader`
 - ▶ `glShaderSource`
 - ▶ `glCompileShader`
- ▶ Функция должна бросать исключение (например, `std::runtime_error`), если компиляция шейдера провалилась
 - ▶ `glGetShaderiv`
- ▶ Заведите строковую константу с любым значением, создайте фрагментный шейдер (`GL_FRAGMENT_SHADER`) с помощью вашей функции и убедитесь, что бросается исключение

Задание 2

- ▶ Бросаемое исключение должно содержать текст ошибки компиляции шейдера

Задание 2

- ▶ Бросаемое исключение должно содержать текст ошибки компиляции шейдера
 - ▶ `glGetShaderiv`
 - ▶ `glGetShaderInfoLog`
 - ▶ `std::string info_log(info_log_length, '\0')`

Задание 3

- ▶ Пишем настоящий фрагментный (пиксельный) шейдер

Задание 3

- ▶ Пишем настоящий фрагментный (пиксельный) шейдер

```
R"#version 330 core
```

```
layout (location = 0) out vec4 out_color;
```

```
void main()
```

```
{
```

```
    out_color = vec4(1.0, 0.0, 0.0, 1.0);
```

```
}
```

```
);
```

Задание 4

- ▶ Пишем вершинный шейдер

Задание 4

- ▶ Пишем вершинный шейдер

```
R"#version 330 core
```

```
const vec2 VERTICES[3] = vec2[3](  
    vec2(0.0, 0.0),  
    vec2(1.0, 0.0),  
    vec2(0.0, 1.0)  
);
```

```
void main()  
{  
    gl_Position = vec4(VERTICES[gl_VertexID], 0.0, 1.0);  
}  
)";
```

Задание 4

- ▶ Пишем вершинный шейдер

```
R"#version 330 core
```

```
const vec2 VERTICES[3] = vec2[3](  
    vec2(0.0, 0.0),  
    vec2(1.0, 0.0),  
    vec2(0.0, 1.0)  
);
```

```
void main()  
{  
    gl_Position = vec4(VERTICES[gl_VertexID], 0.0, 1.0);  
}  
)";
```

- ▶ Создаём вершинный шейдер (GL_VERTEX_SHADER) с этим кодом

Задание 5

- ▶ Пишем функцию создания шейдерной программы
- ▶ Программа = несколько скомпилированных шейдеров, слинкованных вместе

```
GLuint create_program(GLuint vertex_shader,  
                     GLuint fragment_shader)
```

Задание 5

- ▶ Пишем функцию создания шейдерной программы
- ▶ Программа = несколько скомпилированных шейдеров, слinkованных вместе

```
GLuint create_program(GLuint vertex_shader,  
                     GLuint fragment_shader)
```

- ▶ `glCreateProgram`
- ▶ `glAttachShader`
- ▶ `glLinkProgram`
- ▶ `glGetProgramiv` (2 паза)
- ▶ `glGetProgramInfoLog`

Задание 5

- ▶ Пишем функцию создания шейдерной программы
- ▶ Программа = несколько скомпилированных шейдеров, слinkованных вместе

```
GLuint create_program(GLuint vertex_shader,  
                     GLuint fragment_shader)
```

- ▶ `glCreateProgram`
 - ▶ `glAttachShader`
 - ▶ `glLinkProgram`
 - ▶ `glGetProgramiv` (2 раза)
 - ▶ `glGetProgramInfoLog`
- ▶ Создаём программу, используя два созданных ранее шейдера

Задание 6

- ▶ Создаём Vertex Array Object
- ▶ VAO содержит информацию о входных данных -
расположение данных о вершинах, типы атттрибутов
- ▶ В нашем случае нужен только номинально

Задание 6

- ▶ Создаём Vertex Array Object
- ▶ VAO содержит информацию о входных данных -
расположение данных о вершинах, типы атттрибутов
- ▶ В нашем случае нужен только номинально
 - ▶ `glGenVertexArrays`

Задание 6

- ▶ Создаём Vertex Array Object
- ▶ VAO содержит информацию о входных данных -
расположение данных о вершинах, типы атрибутов
- ▶ В нашем случае нужен только номинально
 - ▶ `glGenVertexArrays`
- ▶ Рисуем треугольник (`GL_TRIANGLES`), используя созданные
программу и VAO

Задание 6

- ▶ Создаём Vertex Array Object
- ▶ VAO содержит информацию о входных данных -
расположение данных о вершинах, типы атрибутов
- ▶ В нашем случае нужен только номинально
 - ▶ `glGenVertexArrays`
- ▶ Рисуем треугольник (`GL_TRIANGLES`), используя созданные
программу и VAO
 - ▶ `glUseProgram`
 - ▶ `glBindVertexArray`
 - ▶ `glDrawArrays`

Задание 7

- ▶ Добавляем градиентное закрашивание
- ▶ Из вершинного шейдера во фрагментный можно передавать данные: они будут интерполироваться между вершинами

Задание 7

- ▶ Добавляем градиентное закрашивание
- ▶ Из вершинного шейдера во фрагментный можно передавать данные: они будут интерполироваться между вершинами

В вершинном шейдере:

```
out vec3 color;  
void main()  
{  
    gl_Position = ...  
    color = ...  
}
```

Во фрагментном шейдере:

```
in vec3 color;  
void main()  
{  
    out_color = vec4(color, 1.0);  
}
```

Задание 8

- ▶ Запрещаем интерполяцию переменных

Задание 8

- ▶ Запрещаем интерполяцию переменных

```
flat out vec3 color;
```

```
flat in vec3 color;
```

Задание 8

- ▶ Запрещаем интерполяцию переменных

```
flat out vec3 color;
```

```
flat in vec3 color;
```

- ▶ Будет использоваться значение в последней вершине
- ▶ Можно настроить с помощью `glProvokingVertex`