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)
```

- Нужно использовать:
  - `glCreateShader` – создаёт и возвращает ID шейдера
  - `glShaderSource` – задаёт исходный код шейдера (функция умеет принимать массив строк, но у нас она будет одна; в качестве длины можно указать `NULL`)
  - `glCompileShader` – компилирует шейдер
- Функция должна возвращать ID созданного шейдера (то, что вернула `glCreateShader`)
- Функция должна бросать исключение (например, `std::runtime_error`), если компиляция шейдера провалилась
  - `glGetShaderiv` – позволяет узнать параметры шейдера, в т.ч. статус компиляции (`GL_COMPILE_STATUS`)
- Заведите строковую константу с любым значением, создайте фрагментный шейдер (`shader_type = GL_FRAGMENT_SHADER`) с помощью вашей функции и убедитесь, что бросается исключение
- Это нужно делать где-то после `glewInit()` и до начала основного цикла `while (running)`

Задание 2

- Бросаемое исключение должно содержать текст ошибки компиляции шейдера
  - `glGetShaderiv` – позволяет узнать длину лога компиляции (`GL_INFO_LOG_LENGTH`)
  - `glGetShaderInfoLog` – позволяет получить сам текст компиляции (записывает его в указанное вами место в памяти!)
  - Можно завести статический массив `char info_log[1024];` и записать туда лог
  - Можно сначала узнать длину с помощью `glGetShaderiv`, создать переменную `std::string info_log(info_log_length, '\0');`, и записать лог в неё (через `info_log.data()`)

Задание 3

- Пишем настоящий фрагментный (пиксельный) шейдер
- Удобно использовать raw string literal из C++11:

```
const char fragment_source[] = R"(#version 330 core

layout (location = 0) out vec4 out_color;

void main()
{
    // vec4(R, G, B, A)
    out_color = vec4(1.0, 0.0, 0.0, 1.0);
}
)";
```

- (цвет можно выбрать любой)

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 (`shader_type = GL_VERTEX_SHADER`) с этим кодом
- Теперь у нас создано два шейдера: вершинный и фрагментный, каждый со своим кодом

Задание 5

- Пишем функцию создания шейдерной программы

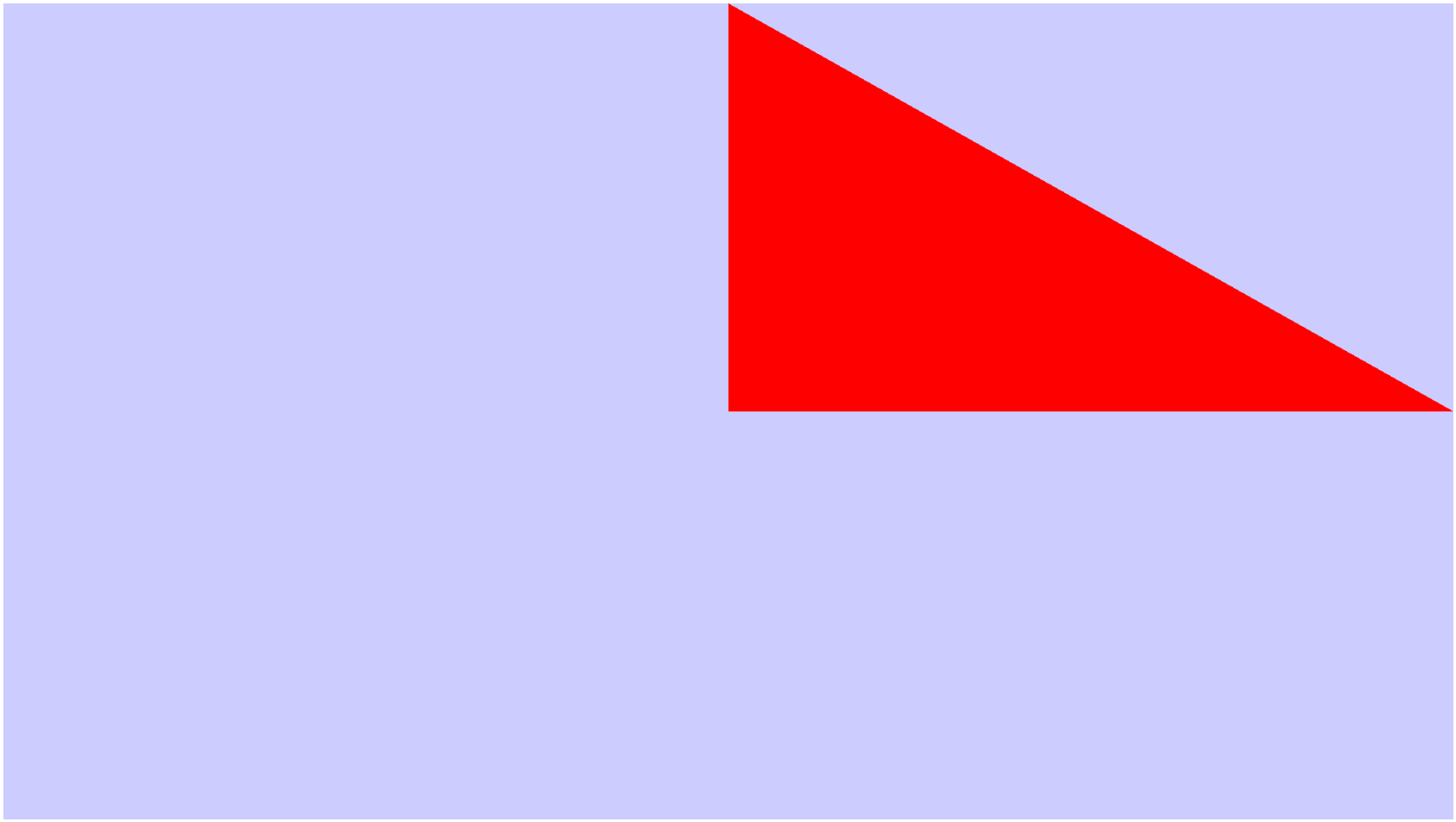
```
GLuint create_program(GLuint vertex_shader,
    GLuint fragment_shader)
```

- Программа = несколько скомпилированных шейдеров, слинкованных вместе
- Нужно использовать:
  - `glCreateProgram` – создаёт программу и возвращает её ID
  - `glAttachShader` – присоединяет шейдер к программе
  - `glLinkProgram` – линкует шейдеры (собирает программу из присоединённых шейдеров)
  - `glGetProgramiv` – 2 раза: получить статус (`GL_LINK_STATUS`) и длину лога линковки (`GL_INFO_LOG_LENGTH`)
  - `glGetProgramInfoLog` – получить текст лога линковки, если линковка не удалась
- Функция возвращает ID созданной программы (результат `glCreateProgram`)
- Аналогично `create_shader`, функция должна бросать исключение с текстом ошибки, если линковка программы не удалась
- Вызываем созданную функцию, чтобы создать программу, используя два созданных ранее шейдера

Задание 6

- Создаём Vertex Array Object
- VAO содержит информацию о входных данных - расположение данных о вершинах, типы аттрибутов
- В нашем случае нужен только номинально, но без него ничего не будет рисоваться
  - `glGenVertexArrays` – создаёт Vertex Array (функция умеет создавать сразу несколько VAO, поэтому принимает количество и указатель на массив; нам хватит одного VAO)
- Рисуем треугольник, используя созданные программу и VAO: где-то в теле основного цикла, после `glClear` и до `SDL_GL_SwapBuffers`:
  - `glUseProgram` – включаем использование созданной шейдерной программы
  - `glBindVertexArray` – включаем VAO
  - `glDrawArrays` – рисуем треугольники (`GL_TRIANGLES`), номер стартовой вершины – 0, количество вершин – 3

Задание 6



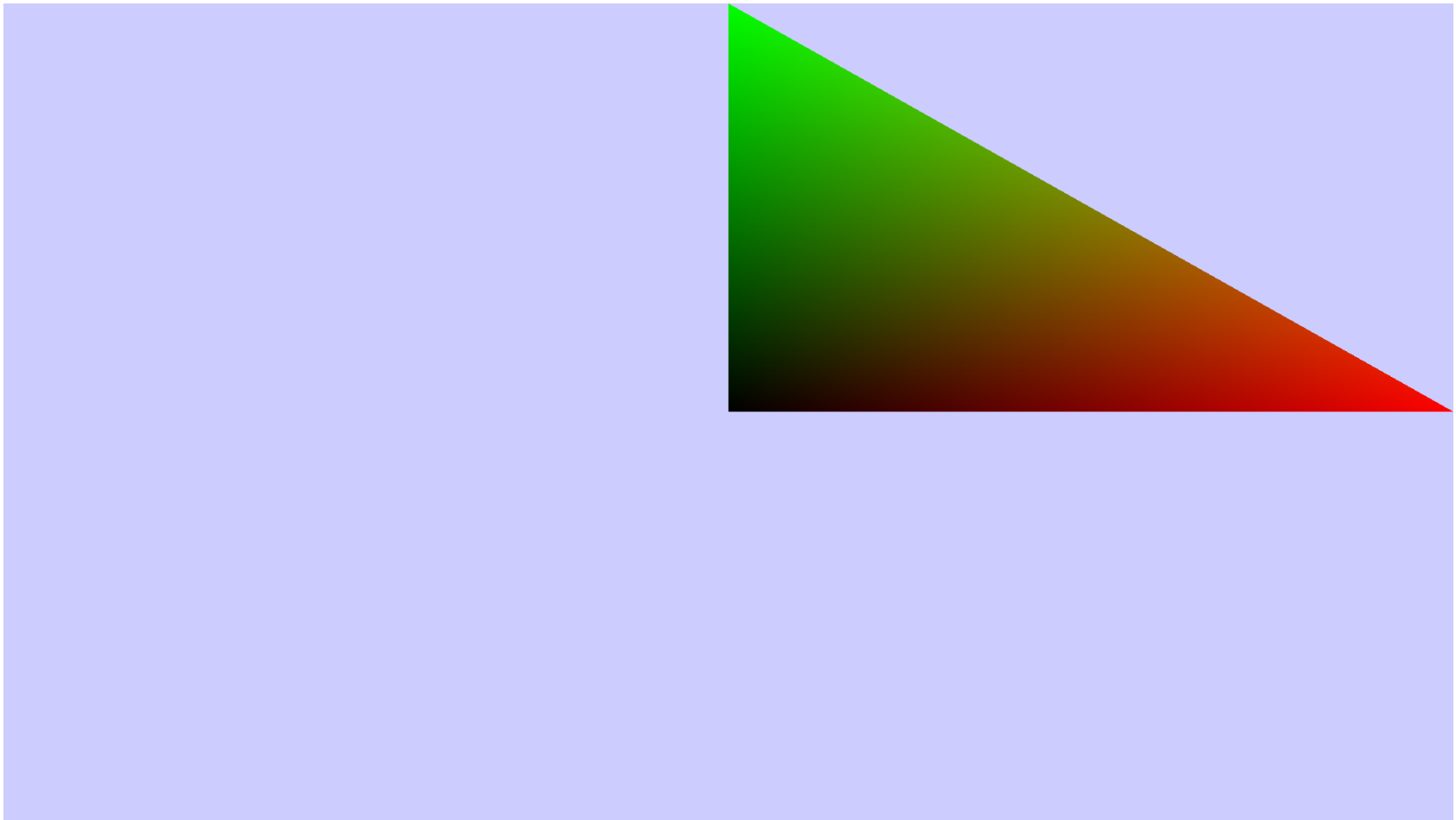
Задание 7

- 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 = vec4(VERTICES[gl_VertexID], 0.0, 1.0);
    color = ...; // что-нибудь, зависящее от gl_VertexID
                 // или от gl_Position
}

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

Задание 7



Задание 8

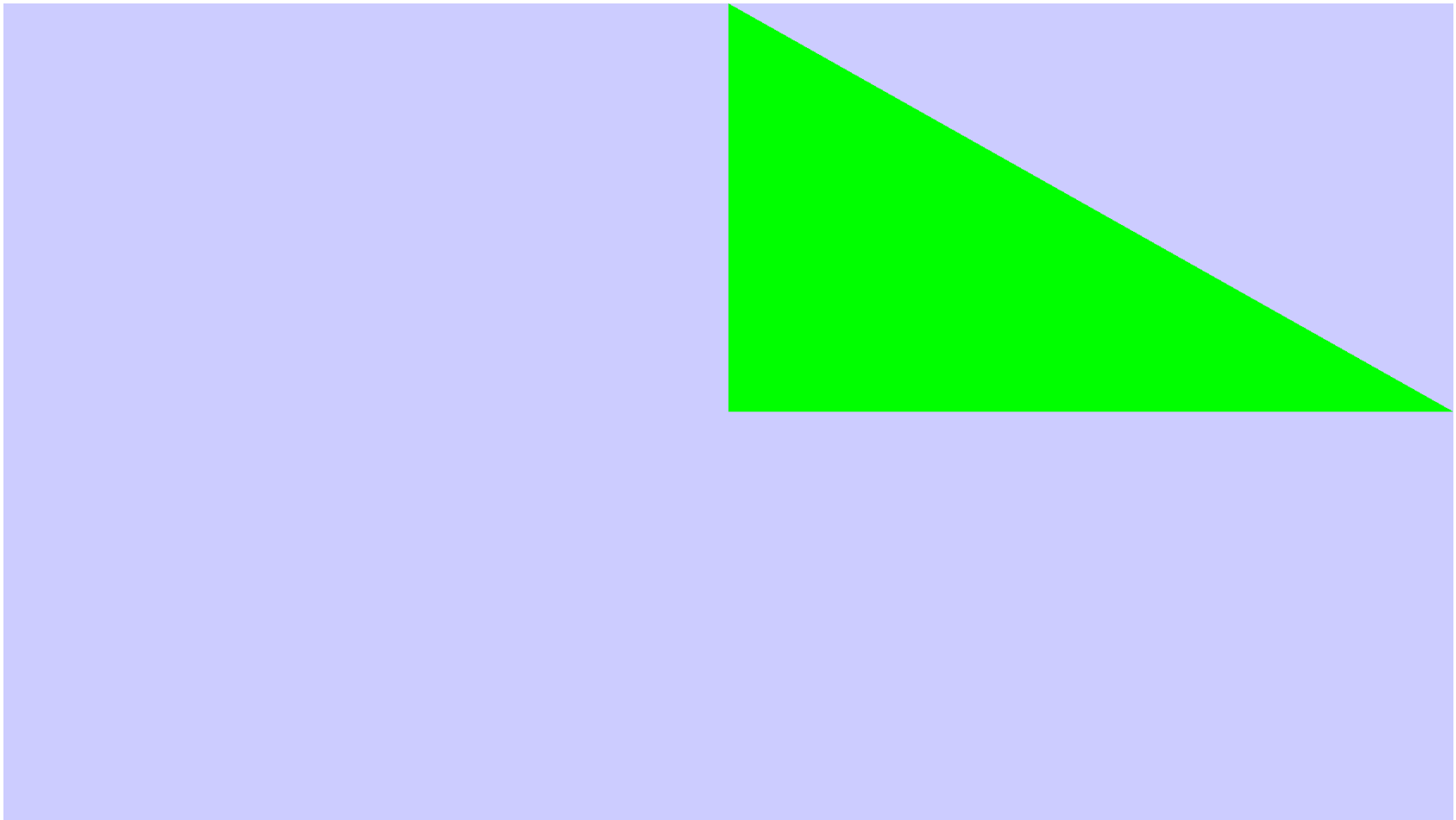
- Запрещаем интерполяцию переменных

```
flat out vec3 color;

flat in vec3 color;
```

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

Задание 8



Задание 9*

- Раскрашиваем треугольник в шахматном порядке
- Делается во фрагментном шейдере
- Нужно проинтерполировать какое-нибудь двумерное значение между вершинами
- Функция `floor` будет полезной

Задание 9