Лекция 12. Работа с аппаратно-ускоренной графикой

Илья Макаров

ИТМО JB

7 декабря 2021
Санкт-Петербург

Замечания по домашке

- Лишние заголовочные файлы и лишние объявления функций.
- Неиспользуемый (закомментированный) код.
- Приведения в стиле C.
- Отсутствие квалификаторов `const`.
- Ошибки при выборе типов для хранения значений.
- Отсутствие (или неправильное) использование атрибутов.
- Отсутствие проверок compile-time (`static_assert`) и run-time (`assert`).

Замечания по домашке

- Многократное открытие файлов.
- Отсутствие виртуальных деструкторов.
- Потеря точности при многократном округлении.
- Требование на фиксированный порядок аргументов.
- Использование сырых указателей и утечки памяти.
- Неправильный результат фильтрации.
- Необработанные исключения.

Предлагаемое решение

```
struct Color {
    std::uint8_t r = 0;
    std::uint8_t g = 0;
    std::uint8_t b = 0;
};

class Bitmap {
public:
    Bitmap(
        std::size_t width,
        std::size_t height,
        std::vector<std::uint8_t> && data);
public:
    void set_pixel(
        std::size_t x,
        std::size_t x,
        Color color);
    const Color & get_pixel(
        std::size_t x,
        std::size_t y) const;
private:
    std::size_t width_;
    std::size_t height_;
    std::vector<Color> data_;
};
```

Предлагаемое решение

```
[[nodiscard]] Bitmap read_image(
    const std::filesystem::path & image_path) {
    // check file exists
    // read headers
    // fill bitmap content
}

void write_image(
    const Bitmap & bitmap,
    const std::filesystem::path & image_path) {
    // create headers
    // write headers
    // write bitmap content as bytes
}

[[nodiscard]] Bitmap blur_image(
    const Bitmap & bitmap,
    const size_t & kernel_size) {
    // create kernel
    // apply convolution
}
```

Предлагаемое решение

```
class ArgParser
{
public:
    ArgParser(int argc, char ** argv) {
        // fill hash table
    }
public:
    [[nodiscard]] std::string get_argument(std::string_view arg_name){
        // try find arg by name
        // throws on error
    }
private:
    std::unordered_map<std::string, std::string> names_and_values_;
};
```

Предлагаемое решение

```
int main(int argc, char **argv) {
    try {
        ArgParser parser(argc, argv);
        const std::filesystem input_image = parser.get_argument("-i");
        const std::filesystem output_image = parser.get_argument("-o");
        const std::size_t kernel_radius = std::stoi(parser.get_argument("-r"));

        const auto input_bitmap = read_image(input_image);
        const auto blured_bitmap = blur(input_bitmap, kernel_radius);
        write_image(blured_bitmap, output_image);
        return 0;
    }
    catch (const std::exception & error) {
        std::cerr << error.what() << std::endl;
        return -1;
    }
}
```

Qt

- Коллекция библиотек, расширяющих функциональность C++.
- Для использования в коммерческих продуктах (и статической линковки) нужна лицензия.

Qt

Основные модули:

- Core
- Widgets
- Network
- Test
- QML
- SQL
- …

Qt интеграция

Исходный код (или предсобранные бинарные файлы):

- `https://qt.io`;
- либо пакетный менеджер (например, `https://conan.io`).

Интеграция в систему сборки:

```
cmake_minimum_required(VERSION 3.15 FATAL_ERROR)

project(gl_demo)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTORCC ON)
set(CMAKE_AUTOUIC ON)

add_executable(${PROJECT_NAME} main.cpp GLWindow.h GLWindow.cpp shaders.qrc)

find_package(Qt5 COMPONENTS Widgets REQUIRED)
target_link_libraries(${PROJECT_NAME} PRIVATE Qt5::Widgets)
```

Краткий рассказ про OpenGL

- Рассказ про **OpenGL Context**.
- Расширения (extensions).
- OpenGL **Core** vs **Compatibility** profile.
- **Вершинный** и **индексный** буферы OpenGL.
- Шейдерные программы: **Vertex** и **Fragment**.
- **Атрибуты** (attributes) и **юниформы** (uniforms).

Где искать ответы на вопросы

- Вопросы и ответы `https://stackoverflow.com`.
- Спецификация на `https://www.khronos.org/registry/OpenGL/specs/gl/glspec33.core.pdf/`.
- Все спецификации + расширения `https://www.khronos.org/registry/OpenGL/index_gl.php`
- Уроки на `https://learnopengl.com/`.
- Переводы части уроков на `https://habr.com/ru/post/310790/`.

Почему OpenGL 3.3?

- Все основные функции уже реализованы.
- Новые версии только добавляют полезные возможности, не изменяя принципов работы с графическим API.
- **3.3** — версия, которая запустится на практически любом графическом ускорителе.
- Дополнительные возможности можно получить с помощью расширений.

Расширения OpenGL

- Производители графических карт могут дополнять функционал графического API.
- Популярные, как правило, попадают в новые версии API.
- Существует механизм для работы с расширениями в run-time. Выглядит это примерно так:

```
if (extension exists) {
    // call new API
}
else {
    // use old one
}
```

Core VS Compatibility

- Использование **Core**-профиля заставляет нас пользоваться современными и актуальными практиками при разработке графических приложений.
- **Compatibility**-профиль сохраняет совместимость.

Замечание: на некоторых платформах **Compatibility** не доступен (например, на платформах Apple).

OpenGL Context и его состояние

- OpenGL по своей сути — это **большой конечный автомат**: набор переменных, определяющий некоторое состояние.
- Перед отрисовкой следующего кадра мы задаем необходимое **состояние**, которое говорит OpenGL, **как нужно рисовать**.
- Под состоянием графического конвейера обычно имеют ввиду состояние контекста.

Основные шаги для отрисовки

- Подготовливаем сцену (в геометрическом смысле).
- Создаем и заполняем необходимые OpenGL-буферы, копируем их в видеопамять.
- Задаем необходимое состояние контекста (как рисуем + куда рисуем).
- Делаем вызов отрисовки.
- **ВАЖНО**: освобождаем не нужные более нам ресурсы.

Далее рассмотрим эти шаги подробнее.

Подготовка сцены

- Обновляем объекты в сцене.
- Выполняем предварительные вычисления видимости, т.е. определяем, что попало в сцену, чтобы не рисовать лишние объекты.
- Возможно, обрабатываем пользовательские события.
- …

В результате должны получить набор объектов C++, которые будем использовать для формирования буферов в видеопамяти.

Буферы OpenGL

- OpenGL — преимущественно C -библиотека, поэтому работа со всеми объектам осуществляется в стиле C.
- Создать объект в видеопамяти значит заполнить некоторую C -структуру, которая в дальнейшем будет записана в видеопамять.

Буферы OpenGL

Общая схема работа с объектами в OpenGL:

```
GLuint object_id = 0;
glGenObject(1, &object_id);
glBindObject(object_id);
glSetObjectStateOrData(...);
glBindObject(0);
```

Вершинный буфер

- Содержит геометрию вершин и их атрибуты.
- По сути — набор байтов.
- Чтобы OpenGL смог их корректно прочитать, нужно задать правильные смещения.
- Такие буферы обычно называют **VBO** (Vertex Buffer Object).

Создание вершинного буфера

- Зададим геометрию:

```
GLfloat vertices[] = { // треугольник из 3 вершин
    -0.5f, -0.5f, 0.0f,
    0.5f, -0.5f, 0.0f,
    0.0f,  0.5f, 0.0f
};
```

- Создадим и заполним объект в видеопамяти:

```
GLuint vbo;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
             vertices, GL_STATIC_DRAW);
```

Создание вершинного буфера

Режимы отрисовки.

- GL_STATIC_DRAW — данные редко изменяются в памяти.
- GL_DYNAMIC_DRAW — данные изменяются часто.
- GL_STREAM_DRAW — данные изменяются на каждый кадр.

Это позволяет оптимально использовать видеопамять.

Разметка вершинных атрибутов

- Формат вершин:

- Разметка:

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
                      3 * sizeof(GLfloat), 0);
glEnableVertexAttribArray(0);
```

Разметка вершинных атрибутов

Аргументы **glVertexAttribPointer**:

1. Какой аргумент шейдера хотим настроить (location).
2. Размер аргумента шейдера, в нашем случае vec3.
3. Тип данных — GL_FLOAT.
4. Нужно ли нормализовать данные.
5. Размер вершины.
6. Смещение от начала буфера.

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
                      3 * sizeof(GLfloat), 0);
glEnableVertexAttribArray(0);
```

Индексный буфер

- Содержит индексы вершин из вершинного буфера.
- Такие буферы обычно называют **IBO** (Index Buffer Object) или **EBO** (Element Buffer Object).

Создание индексного буфера

- Зададим индексы:

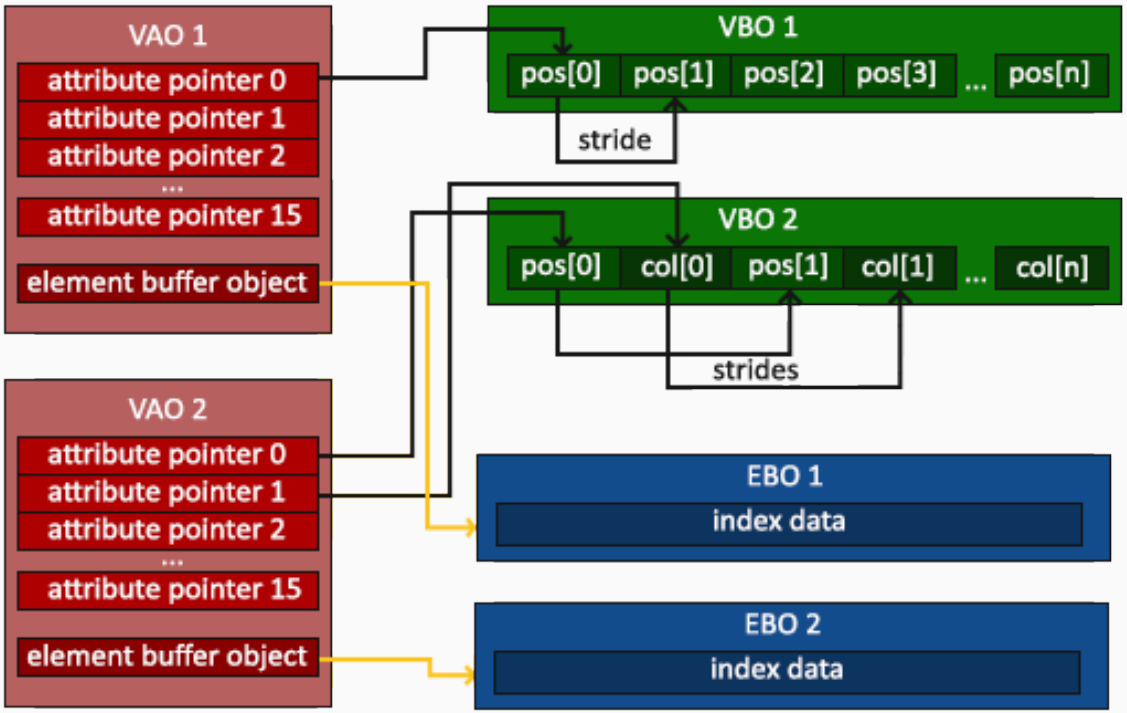
```
GLuint indices[] = { // треугольник из 3 вершин
    0, 1, 2
};
```

- Создадим и заполним объект в видеопамяти:

```
GLuint ibo;
glGenBuffers(1, &ibo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER,
             sizeof(indices), indices,
             GL_STATIC_DRAW);
```

Vertex Array Object

Некоторая обертка, позволяющая быстро переиспользовать VBO и IBO.



Шейдерные программы

- Соотвествуют специальным программируемым частям конвейера.
- Пишутся на C-подобном языке — **GLSL**.
- Требуют дополнительной компиляции.
- Вершинный шейдер обрабатывает каждую вершину.
- Фрагментный шейдер — каждый фрагмент (в каком-то смысле пиксель).

Вершинный шейдер

Пример вершинного шейдера:

```
#version 330 core

layout (location = 0) in vec3 vertex_position;

void main()
{
    gl_Position = vec4(vertex_position.xyz, 1.0);
}
```

Вершинный шейдер

Создание вершинного шейдера:

```
GLuint vs = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vs, 1, &vs_source, 0);
glCompileShader(vs);
```

Фрагментный шейдер

Пример фрагментного шейдера:

```
#version 330 core

out vec4 fragment_color;

void main()
{
    fragment_color = vec4(1.f, 0.f, 0.f, 1.f);
}
```

Фрагментный шейдер

Создание фрагментного шейдера:

```
GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fs, 1, &fs_source, 0);
glCompileShader(fs);
```

Шейдерная программа

Соотвествует набору шейдеров.

```
GLuint sp = glCreateProgram();
glAttachShader(sp, vs);
glAttachShader(sp, fs);
glLinkProgram(sp);
```

Замечание: если шейдер не задан, то он заменяется шейдером по умолчанию.

Подготовка к отрисовке

Задаем буферы для отрисовки:

```
glBindVertexArray(vao);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
                 vertices, GL_STATIC_DRAW);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
                         3 * sizeof(GLfloat), 0);
glEnableVertexAttribArray(0);
glBindVertexArray(0);
```

Отрисовка

Сначала зададим шейдерную программу и VAO:

```
glUseProgram(sp);
glBindVertexArray(vao);
```

Отрисуем треугольник:

```
glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_INT, 0)
glBindVertexArray(0);
```

Unifroms

- Юниформы — некоторые глобальные для всей шейдерной программы переменные.
- Задаются из пользовательского кода на C++.
- Используются в шейдере при помощи ключевого слова **uniform**. Например:

```
#version 330 core
out vec4 color;

uniform vec4 u_color;

void main()
{
    color = u_color;
}
```

Unifroms

Как задать uniform из кода?

- Сначала узнаем расположение:

```
GLint color_location;
color_location = glGetUniformLocation(sp,
                                      "u_color");
```

- Потом задаем значение:

```
glUseProgram(sp);
glUniform4f(color_location, 1.f, 0.f, 0.f, 1.f);
```

Обработка ошибок

- Обработка ошибок происходит в стиле C.
- Для получения статусов используются функции вида **glGet***. Например, для шейдеров:

```
GLint status;
glGetShaderiv(vs, GL_COMPILE_STATUS, &status);
```

- Также существует функция **glGetError**, которая возвращает информацию об ошибках.

Корректное управление ресурсами

- Все созданные OpenGL объекты должны быть освобождены вручную.
- Инициализация объектов должна производиться единожды.
- Заполнять и модифицировать эти объекты можно бесконечно много раз.
- Заполнять данные нужно только по мере их изменения.