

# Введение в OpenGL

Компьютерная графика

# Что говорит Wiki про OpenGL?

OpenGL (Open Graphics Library) — **спецификация**, определяющая **платформонезависимый** (независимый от языка программирования) **программный интерфейс** для написания приложений, использующих **двумерную и трёхмерную компьютерную графику**.

# Две стороны одной медали

- На базовом уровне, OpenGL — **это просто спецификация**, то есть документ, описывающий набор функций и их точное поведение.
- Производители оборудования на основе этой спецификации создают **реализации** — библиотеки функций, соответствующих набору функций спецификации.

# Текущее состояние

<b>Тип</b>	API
<b>Разработчик</b>	<a href="#">Silicon Graphics</a> , затем <a href="#">Khronos Group</a>
<b>Написана на</b>	C
<b>Операционная система</b>	Кроссплатформенное программное обеспечение
<b>Первый выпуск</b>	январь 1992
<b>Последняя версия</b>	4.6 (31 июля 2017)
<b>Состояние</b>	смотрите <a href="#">Vulkan</a>
<b>Лицензия</b>	Различные
<b>Сайт</b>	<a href="http://opengl.org">opengl.org</a> 

# Конкуренты и соратники

- Mantle – низкоуровневый графический и вычислительный API от AMD
- DirectX – набор API для Windows и Xbox
- Vulkan – другой графический API от Khronos Group
- Mesa (Mesa3D) — реализация OpenGL, Vulkan и других спецификаций графических API с открытым исходным кодом. Mesa переводит эти спецификации в драйверы графического оборудования конкретного производителя.

# Khronos Group: OpenGL и не только



# WebGL

- Web-based Graphics Library — программная библиотека для языка программирования JavaScript, позволяющая создавать на JavaScript интерактивную 3D-графику
- построена на основе OpenGL ES 2.0 и обеспечивает API для 3D-графики
- часть кода на WebGL может выполняться непосредственно на видеокартах
- использует HTML5-элемент canvas
- оперирует с DOM

# Vulkan

- кроссплатформенный API для 2D- и 3D-графики, впервые представленный Khronos Group в рамках конференции GDC 2015
- предназначен для обеспечения различных преимуществ по сравнению с другими API, включая его предшественника OpenGL
- предлагает более низкие накладные расходы, более непосредственный контроль над GPU, и с меньшей нагрузкой на CPU



# Vulkan и шейдеры

- OpenGL использует язык высокого уровня для написания шейдеров GLSL. Это заставляет каждого производителя OpenGL драйвера реализовать свой собственный компилятор для GLSL, выполняемый во время выполнения приложения, чтобы перевести шейдерные программы в исполняемый код для целевой платформы.
- Vulkan вместо этого обеспечивает промежуточный двоичный формат под названием SPIR-V (Standard Portable Intermediate Representation), аналогичный двоичному формату в который компилируются HLSL шейдеры на платформе DirectX.
- Это позволяет производить компиляцию шейдеров на этапе разработки.
- Также позволяет разработчикам приложений писать шейдеры на других языках, кроме GLSL.

# OpenGL — это

- программный интерфейс к графической аппаратуре
- около 250 отдельных команд (200 в самой OpenGL и 50+ в библиотеке утилит)
- обобщенный, независимый интерфейс, который может быть реализован для различного аппаратного обеспечения

# Основные графические операции OpenGL

- конструирует фигуры из геометрических примитивов
- позиционирует объекты в трехмерном пространстве и выбирает точку наблюдения
- вычисляет цвета для всех объектов
  - могут быть определены приложением
  - получены из расчета условий освещенности
  - вычислены при помощи текстур, наложенных на объекты
  - или из любой комбинации этих факторов
- выполняет растеризацию (растровую развертку)

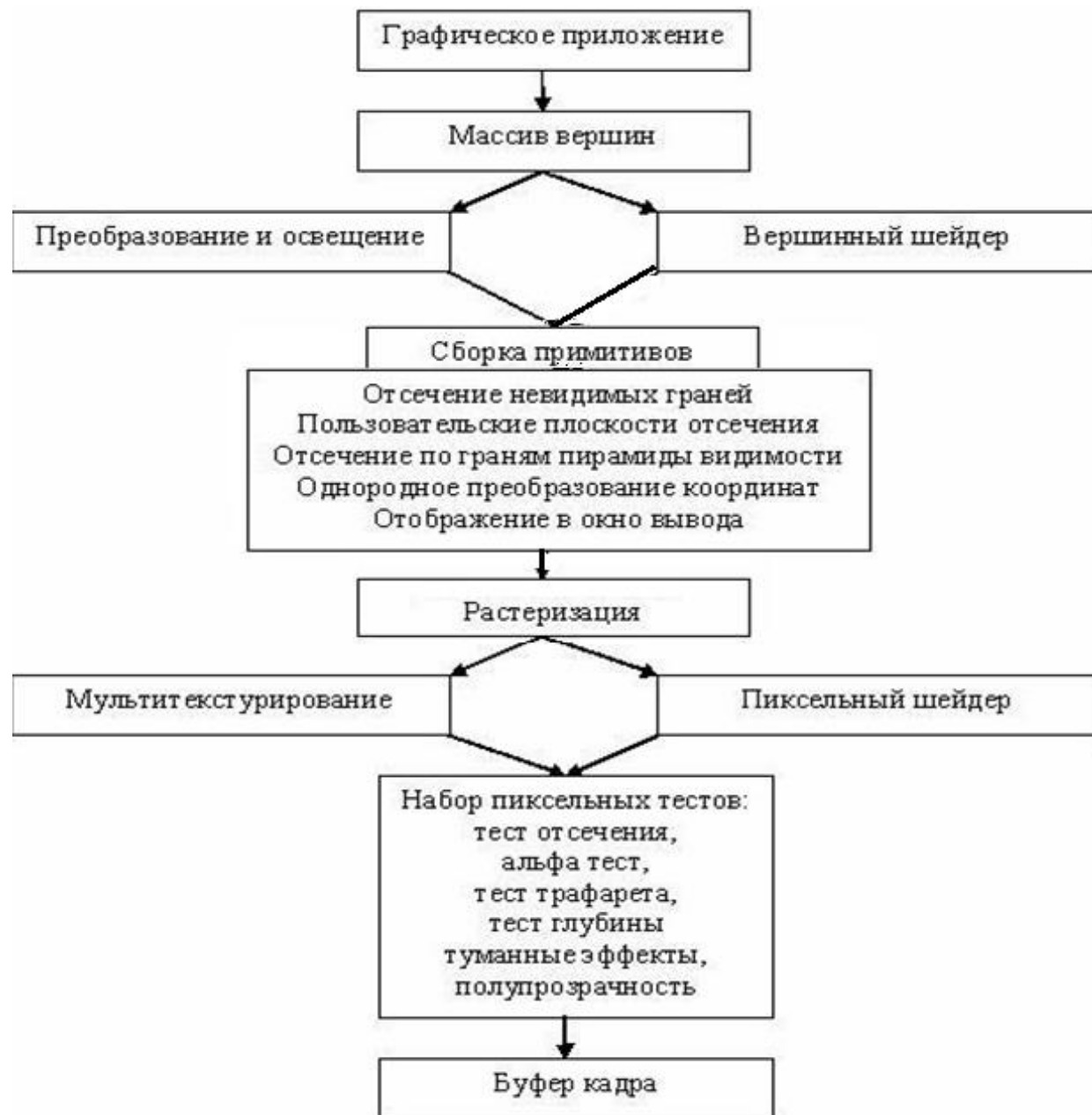
# Чего нет в OpenGL?

- функций для создания окон
- захвата пользовательского ввода
- высокоуровневых функций для описания моделей трёхмерных объектов

# Библиотеки для работы с окнами и не только

- GLEW
- SFML
- Freeglut + SOIL2
- GLFW

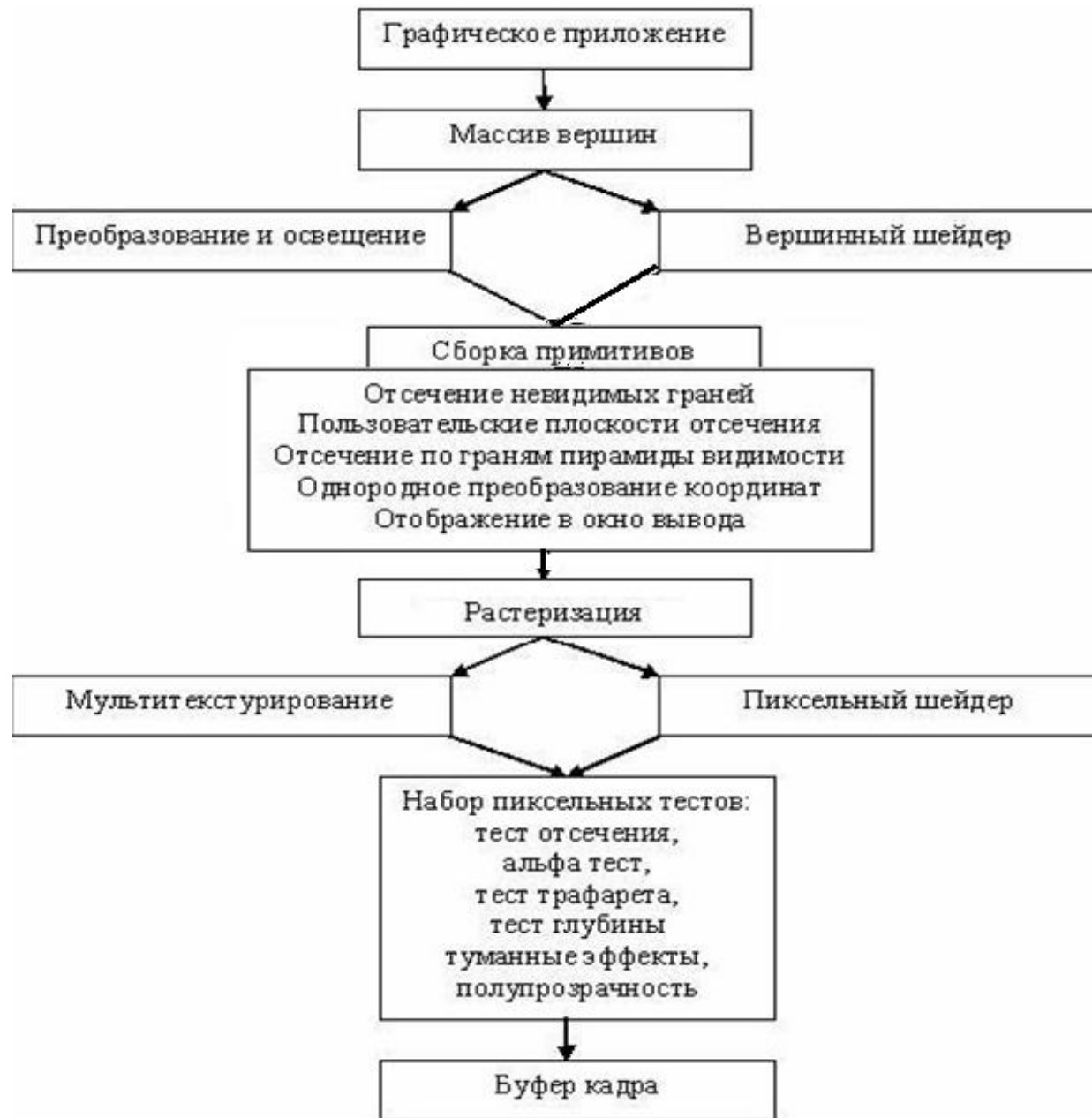
# Конвейер визуализации OpenGL



# Повершинные операции (per-vertex operations)

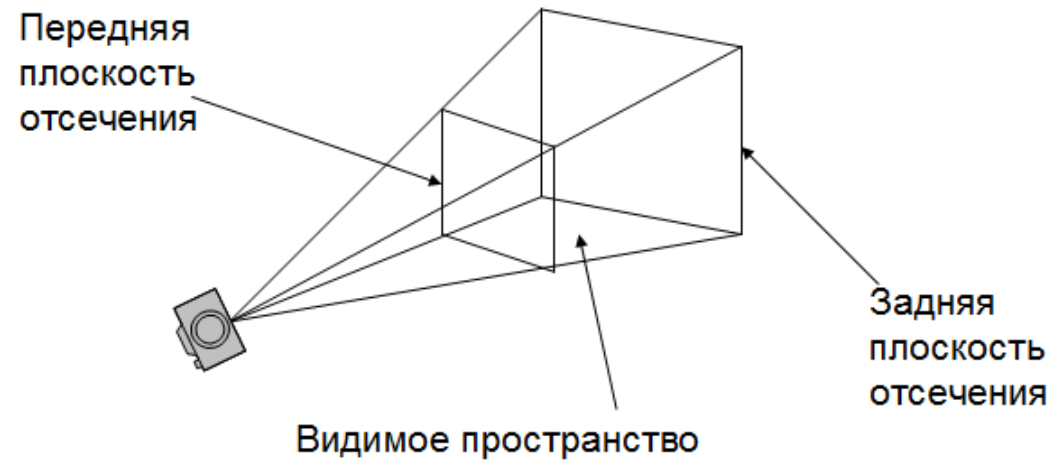
- В течение этого этапа вершины преобразуются в примитивы
  - Различные матричные преобразования
  - Пространственные координаты проецируются из 3D в экранные
- Если используется текстурирование, координаты текстуры могут быть сгенерированы и изменены на этом шаге
- Если используется освещение, вычисления, связанные с ним, производятся с использованием трансформированных вершин, нормалей поверхностей, позиций источников света, свойств материала, а также другой информации, позволяющей вычислить цветовую величину

# Конвейер визуализации OpenGL





# Сборка примитивов. Отсечение

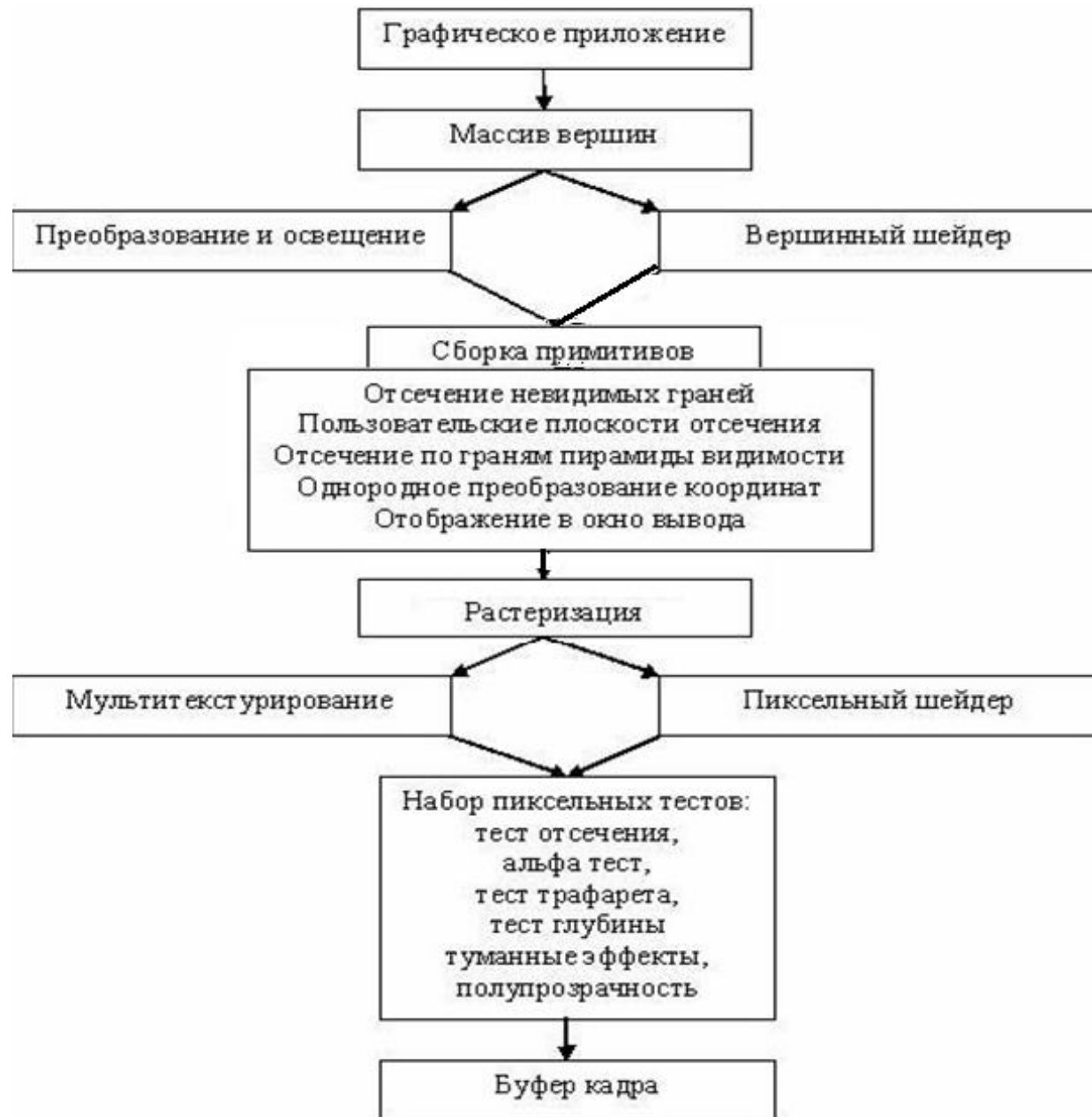


- Отсечение — большая часть сборки примитивов. Это уничтожение частей геометрии, выпадающих за полупространство, определенное плоскостью
  - Отсечение точек просто отвергает или не отвергает вершину
  - Отсечение линий или полигонов может добавить дополнительные вершины в зависимости от того, как именно линия или полигон отсекаются

# Сборка примитивов. Что дальше

- Перспективное разделение
- Операции с портом просмотра (viewport)
- Если включён режим, то выполняется тест на лицевые грани
- Результатом этого этапа являются завершённые примитивы, т.е. трансформированные и отсечённые вершины со связанными цветом, глубиной и, иногда, координатами текстуры

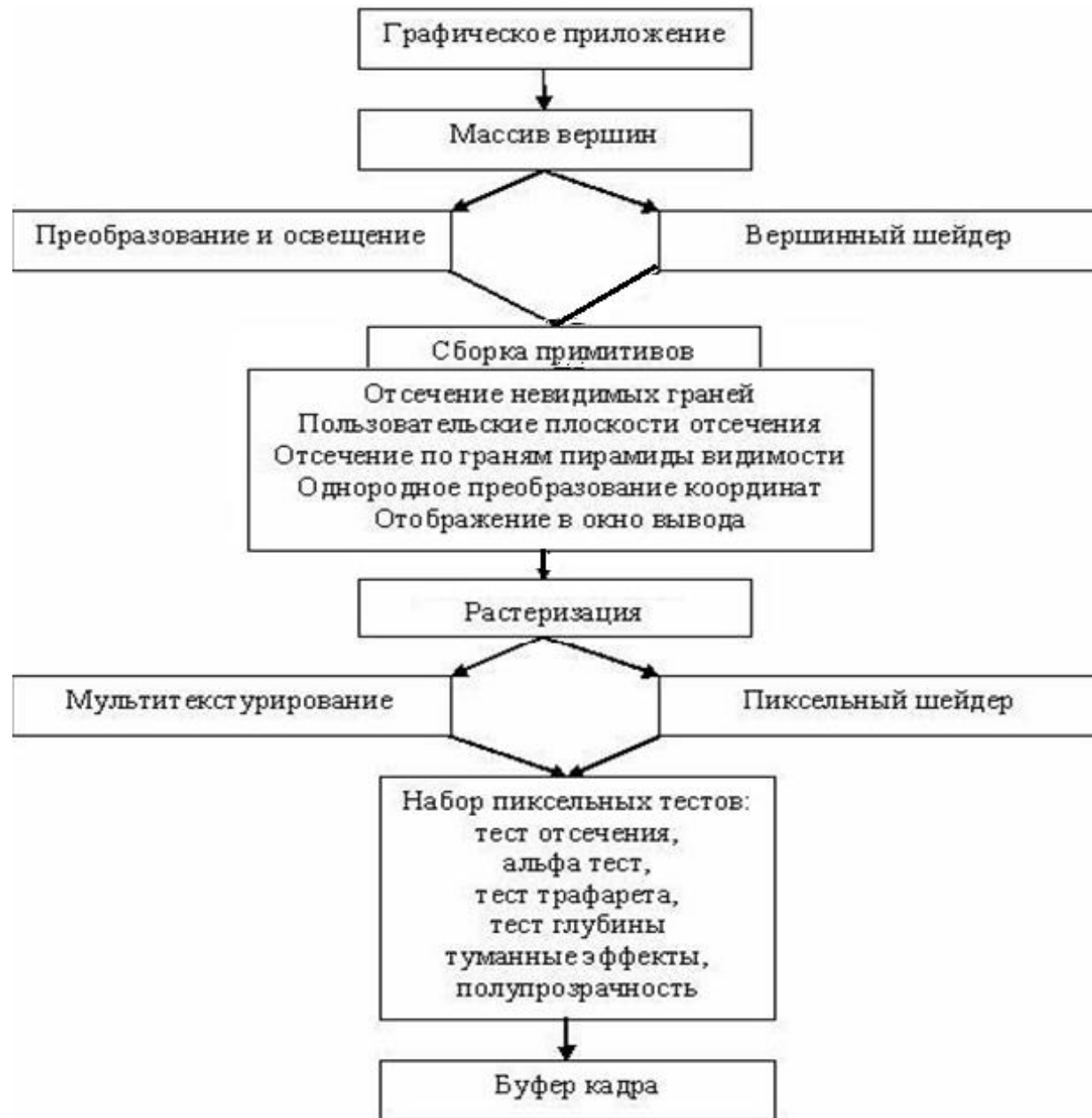
# Конвейер визуализации OpenGL



# Растеризация

- Растеризация – это процесс преобразования геометрических и пиксельных данных во фрагменты
- Каждый фрагмент соответствует пикселю в буфере кадра
- При развертке двух вершин в линию или вычислении внутренних пикселей полигона принимаются в расчет
  - шаблоны линий и полигонов
  - толщина линии
  - размер точек
  - модель заливки
  - вычисления связанные со сглаживанием

# Конвейер визуализации OpenGL



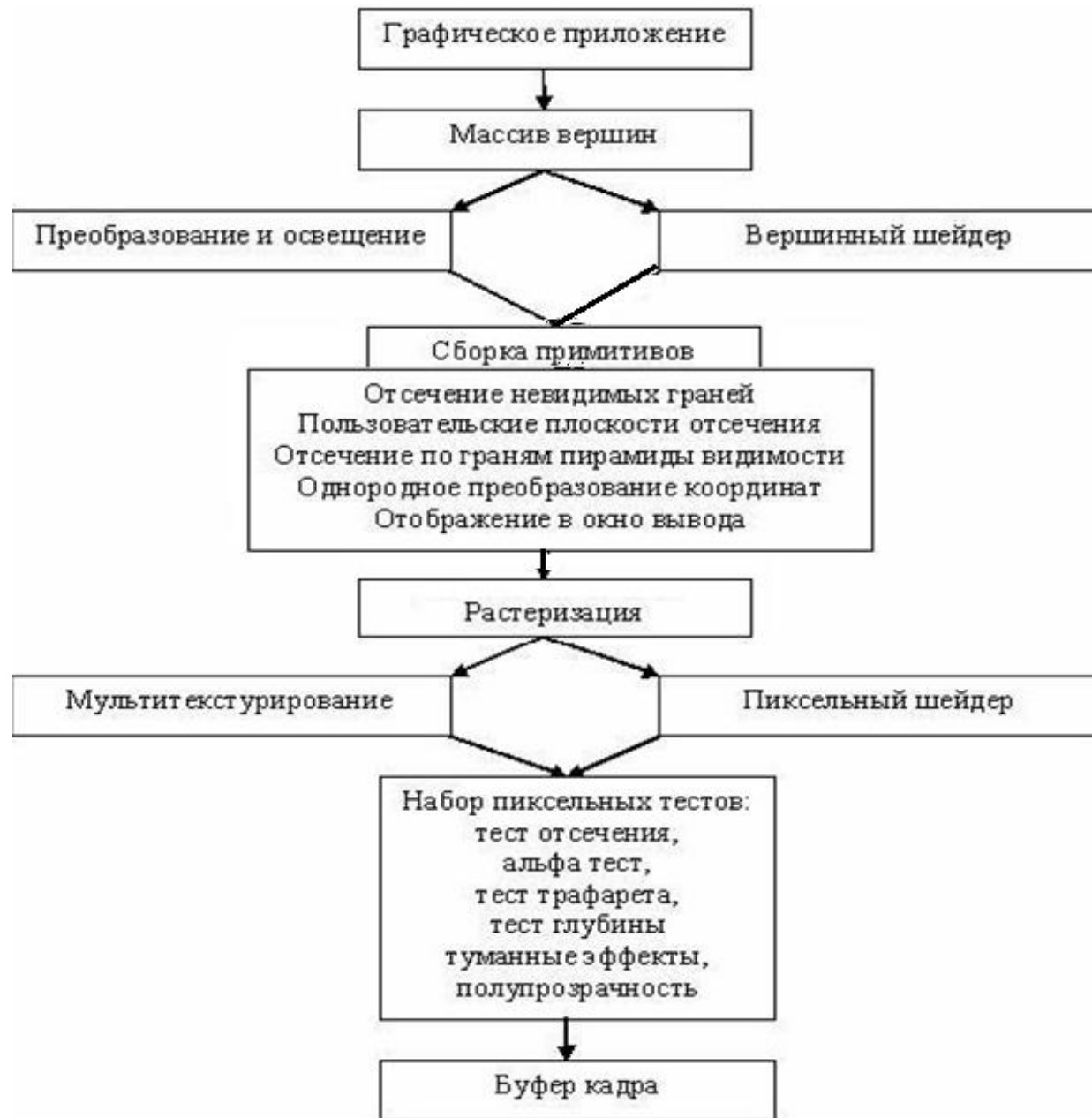
# Операции над фрагментами

- Могут изменить или даже выбросить некоторые фрагменты
- Операции могут быть включены или выключены
  - текстурирование
  - вычисления тумана
  - тест отреза (scissor test)
  - альфа-тест
  - тест трафарета (stencil test)
  - тест буфера глубины
- Если фрагмент не проходит один из включенных тестов, это может закончить его путь по конвейеру
- Далее могут быть произведены наложение, смешивание цветов (dithering), логические операции и маскирование с помощью битовой маски
- Наконец, фрагмент заносится в соответствующий буфер, где становится пикселем

# Наложение текстуры

- Если используется несколько изображений текстур, разумно поместить их в объекты текстуры, чтобы можно было легко переключаться между ними
- Некоторые реализации OpenGL могут иметь дополнительные ресурсы для ускорения операций с текстурами. Например, может существовать специализированная быстрая текстурная память

# Конвейер визуализации OpenGL

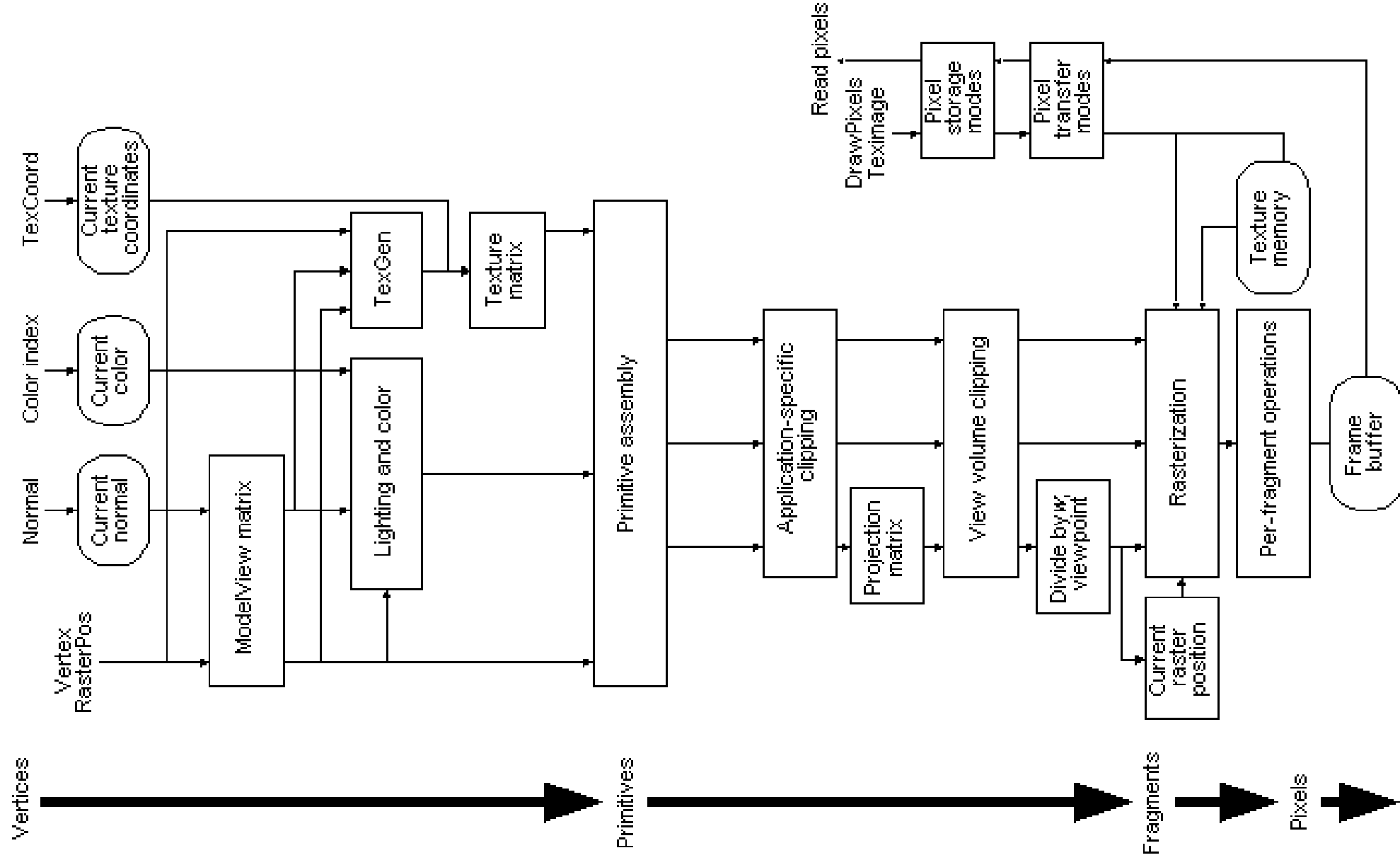




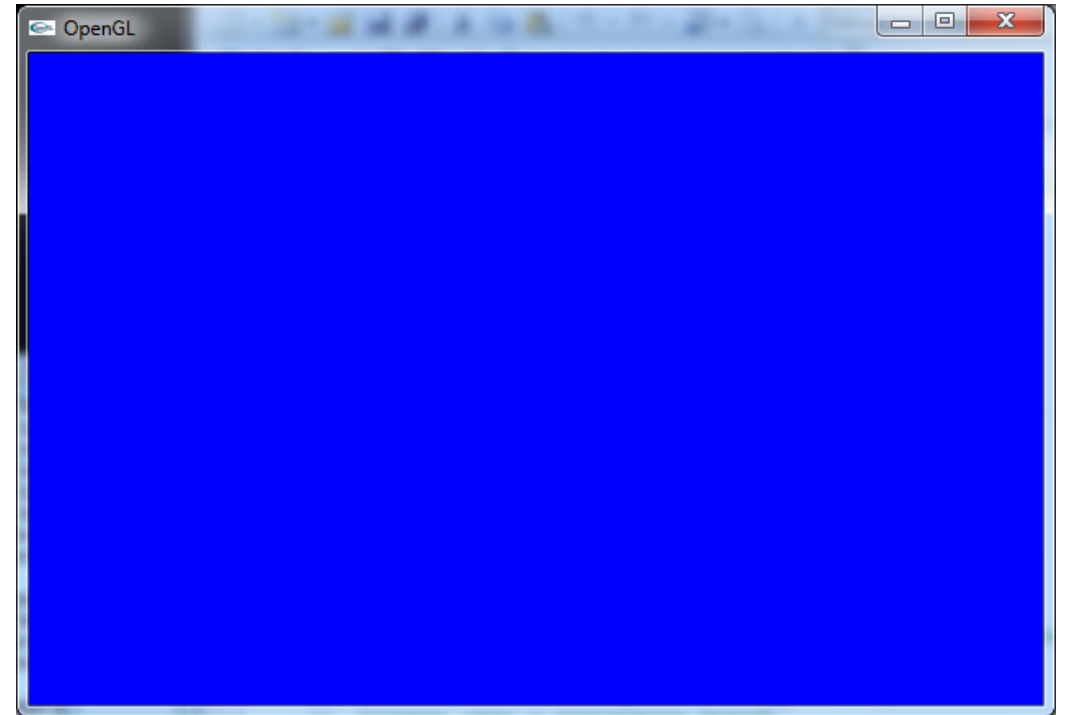
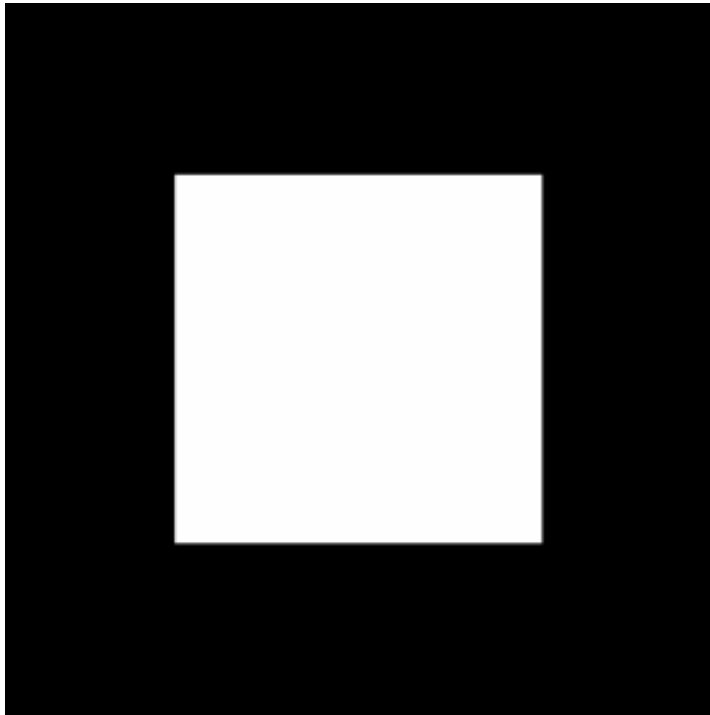
# Операции над пикселями

- Данные о пикселях следуют в конвейере OpenGL параллельным путем
- Массивы пиксельных данных из системной памяти распаковываются, т.е. преобразуются из какого-либо формата, в формат с необходимым числом компонент
- Далее данные масштабируются и обрабатываются пиксельными картами
- Результат сжимается и записывается в текстурную память или отправляется на этап растеризации
- Если пиксельные данные читаются из буфера кадра, над ними выполняются пиксельные операции (pixel-transfer operations)
- Существуют специальные операции копирования пикселей (pixel copy operations) для копирования данных из одной части буфера кадра в другие или из буфера кадра в текстурную память

# Конвейер визуализации OpenGL



На что может быть похожа программа?



# Старый vs новый стандарты OpenGL

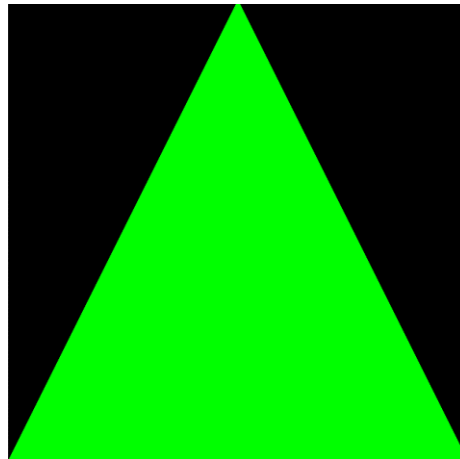
```
main() {
  Init();
  glClearColor(0.0,0.0,0.0);
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f(1.0,1.0,1.0);
  glOrtho(0.0,1.0,0.0,1.0,-1.0,1.0);
  glBegin(GL_POLYGON);
  glVertex3f(0.25,0.25,0.0);
  glVertex3f(0.75,0.25,0.0);
  glVertex3f(0.75,0.75,0.0);
  glVertex3f(0.25,0.75,0.0);
  glEnd();
  glFlush(); Redraw();
}
```

```
int main() { //всего порядка 225 строк
  InitWindow();
  glewInit();
  Init();
  while (window.isOpen()) {
    ...
    glViewport(0, 0, event.size.width, event.size.height);
    ...
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    Draw();
    window.display();
  }
  Release();
  return 0;
}
```

# Типы данных

Суффиксы	Тип данных	Типично соответствующий тип языка C	Тип, определенный в OpenGL
b	целое 8 бит	signed char	GLbyte
s	целое 16 бит	short	GLshort
i	целое 32 бита	int или long	GLint, GLsizei
f	число с плавающей точкой 32 бита	float	GLfloat, GLclampf
d	число с плавающей точкой 64 бита	double	GLdouble, GLclampd
ub	беззнаковое целое 8 бит	unsigned char	GLubyte, GLboolean
us	беззнаковое целое 16 бит	unsigned short	GLushort
ui	беззнаковое целое 32 бита	unsigned int или unsigned long	GLuint, GLenum, GLbitfield

# Первое приложение



На примере лабораторной, разработанной О. Арутюновым и С. Дуюновым

# Необходимые идентификаторы

```
// ID шейдерной программы  
GLuint Program;  
// ID атрибута  
GLuint Attrib_vertex;
```

```
// ID Vertex Buffer Object  
GLuint VBO;
```

# Структура для описания вершины

```
struct Vertex {  
    GLfloat x;  
    GLfloat y;  
};
```



# Вершинный и фрагментный шейдеры

```
// Исходный код вершинного шейдера
const char* VertexShaderSource = R"(
    #version 330 core
    in vec2 coord;
    void main() {
        gl_Position = vec4(coord, 0.0, 1.0);
    }
);
```

```
// Исходный код фрагментного шейдера
const char* FragShaderSource = R"(
    #version 330 core
    out vec4 color;
    void main() {
        color = vec4(0, 1, 0, 1);
    }
);
```

```

int main() {
    sf::Window window(sf::VideoMode(600, 600), "My OpenGL window", sf::Style::Default, sf::ContextSettings(24));
    window.setVerticalSyncEnabled(true);
    window.setActive(true);

    glewInit();
Init();

    while (window.isOpen()) {
        sf::Event event;
        while (window.pollEvent(event)) {
            if (event.type == sf::Event::Closed) { window.close(); }
            else if (event.type == sf::Event::Resized) { glViewport(0, 0, event.size.width, event.size.height); }
        }

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        Draw();

        window.display();
    }
    Release();
    return 0;
}

```

SFML

# Инициализация ресурсов

```
void Init() {  
  
    // Шейдеры  
    InitShader();  
  
    // Вершинный буфер  
    InitVBO();  
}
```

# Инициализация буфера вершин

```
void InitVBO() {
    glGenBuffers(1, &VBO);
    // Вершины нашего треугольника
    Vertex triangle[3] = {
        { -1.0f, -1.0f },
        { 0.0f, 1.0f },
        { 1.0f, -1.0f }
    };
    // Передаем вершины в буфер
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(triangle), triangle, GL_STATIC_DRAW);

    checkOpenGLError(); //Пример функции есть в лабораторной
    // Проверка ошибок OpenGL, если есть, то вывод в консоль тип ошибки
}
```

# InitShader. Вершинный шейдер

```
void InitShader() {  
  
    // Создаем вершинный шейдер  
    GLuint vShader = glCreateShader(GL_VERTEX_SHADER);  
  
    // Передаем исходный код  
    glShaderSource(vShader, 1, &VertexShaderSource, NULL);  
  
    // Компилируем шейдер  
    glCompileShader(vShader);  
  
    std::cout << "vertex shader \n";  
    // Функция печати лога шейдера  
    ShaderLog(vShader); //Пример функции есть в лабораторной  
  
    ...  
}
```

## InitShader. Фрагментный шейдер

...

```
// Создаем фрагментный шейдер  
GLuint fShader = glCreateShader(GL_FRAGMENT_SHADER);
```

```
// Передаем исходный код  
glShaderSource(fShader, 1, &FragShaderSource, NULL);
```

```
// Компилируем шейдер  
glCompileShader(fShader);
```

```
std::cout << "fragment shader \n";  
// Функция печати лога шейдера  
ShaderLog(fShader);
```

...

# Шейдерная программа

...

```
// Создаем программу и прикрепляем шейдеры к ней
```

```
Program = glCreateProgram();
```

```
glAttachShader(Program, vShader);
```

```
glAttachShader(Program, fShader);
```

```
// Линкуем шейдерную программу
```

```
glLinkProgram(Program);
```

```
// Проверяем статус сборки
```

```
int link_ok;
```

```
glGetProgramiv(Program, GL_LINK_STATUS, &link_ok);
```

```
if (!link_ok) {
```

```
    std::cout << "error attach shaders \n";
```

```
    return;
```

```
}
```

...

# Устанавливаем связь между параметрами в программе и шейдере

```
...
// Вытягиваем ID атрибута из собранной программы
const char* attr_name = "coord"; //имя в шейдере

Attrib_vertex = glGetAttribLocation(Program, attr_name);

if (Attrib_vertex == -1) {
    std::cout << "could not bind attrib " << attr_name << std::endl;
    return;
}

checkOpenGLerror();
}
```



# Ну а теперь рисуем

```
void Draw() {  
    glUseProgram(Program); // Устанавливаем шейдерную программу текущей  
  
    glEnableVertexAttribArray(Attrib_vertex); // Включаем массив атрибутов  
  
    glBindBuffer(GL_ARRAY_BUFFER, VBO); // Подключаем VBO  
  
    // сообщаем OpenGL как он должен интерпретировать вершинные данные.  
    glVertexAttribPointer(Attrib_vertex, 2, GL_FLOAT, GL_FALSE, 0, 0);  
    glBindBuffer(GL_ARRAY_BUFFER, 0); // Отключаем VBO  
  
    glDrawArrays(GL_TRIANGLES, 0, 3); // Передаем данные на видеокарту(рисуем)  
  
    glDisableVertexAttribArray(Attrib_vertex); // Отключаем массив атрибутов  
    glUseProgram(0); // Отключаем шейдерную программу  
  
    checkOpenGLError();  
}
```

```
glVertexAttribPointer(Attrib_vertex, 2, GL_FLOAT, GL_FALSE, 0, 0);
```

Первый аргумент описывает какой аргумент шейдера мы хотим настроить.

Второй аргумент описывает размер аргумента в шейдере.

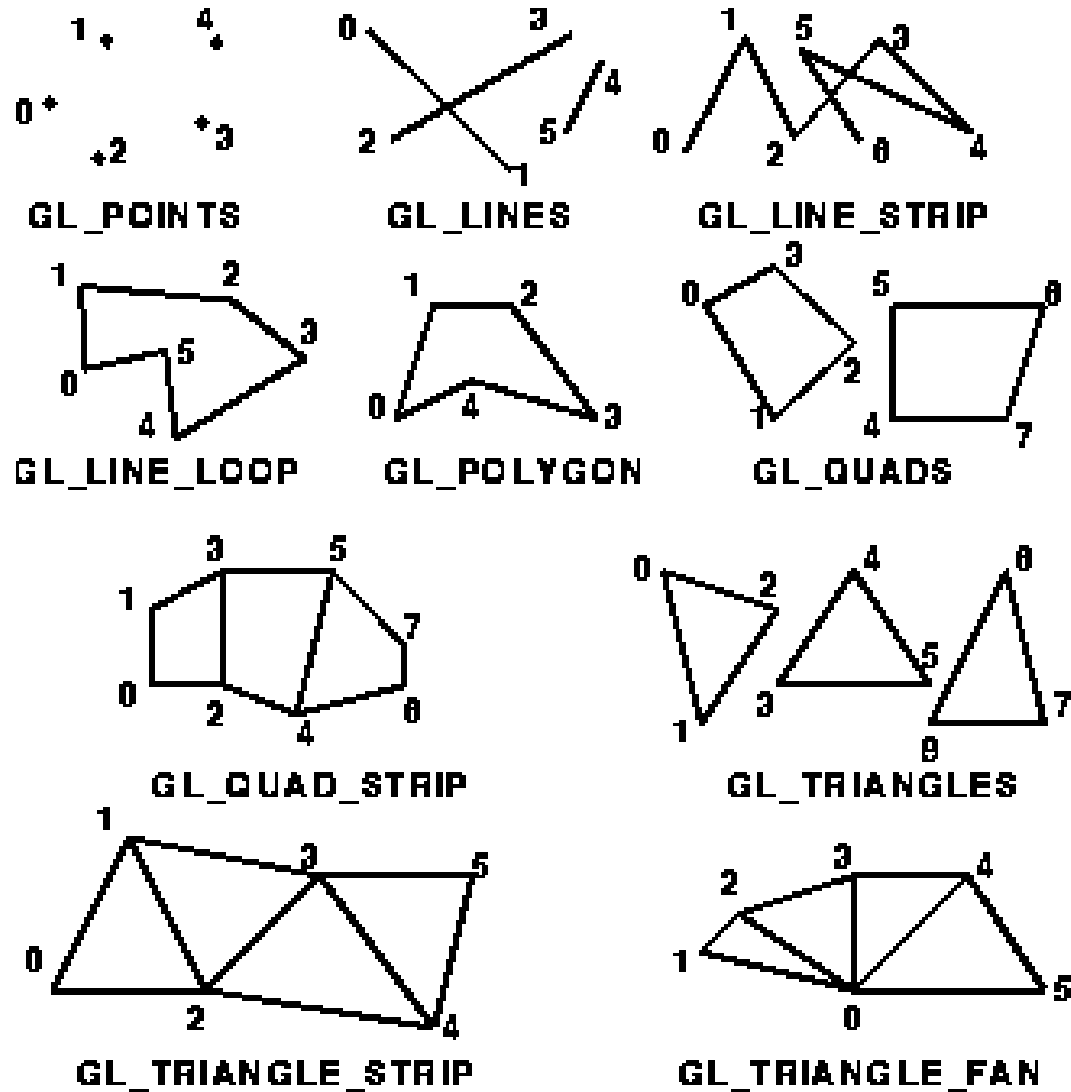
Третий аргумент описывает используемый тип данных.

Четвертый аргумент указывает необходимость нормализовать входные данные. Если указать `GL_TRUE`, то все данные будут расположены между 0 (-1 для знаковых значений) и 1.

Пятый аргумент называется шагом и описывает расстояние между наборами данных. Мы также можем указать шаг равный 0 и тогда OpenGL высчитает шаг (работает только с плотно упакованными наборами данных).

Последний параметр — смещение начала данных в буфере.

```
glDrawArrays(GL_TRIANGLES, 0, 3);
```



# Освобождение ресурсов

```
void Release() {  
  
    // Шейдеры  
    ReleaseShader();  
  
    // Вершинный буфер  
    ReleaseVBO();  
}  
  
// Освобождение шейдеров  
void ReleaseShader() {  
    // Передавая ноль, мы отключаем шейдерную программу  
    glUseProgram(0);  
    // Удаляем шейдерную программу  
    glDeleteProgram(Program);  
}  
  
// Освобождение буфера  
void ReleaseVBO() {  
    glBindBuffer(GL_ARRAY_BUFFER, 0);  
    glDeleteBuffers(1, &VBO);  
}
```

# Литература: Red Book=old Red Book+Orange Book

