AMCBRIDGE

OpenGL, створення вікна, зображення примітивів

Курс з 3D-програмування «Від трикутника до сцени».

Лекція 1.

Нові терміни та позначення

Vertex

GLEW

EBO

Shader

GLSL

VAO

OpenGL

GLFW

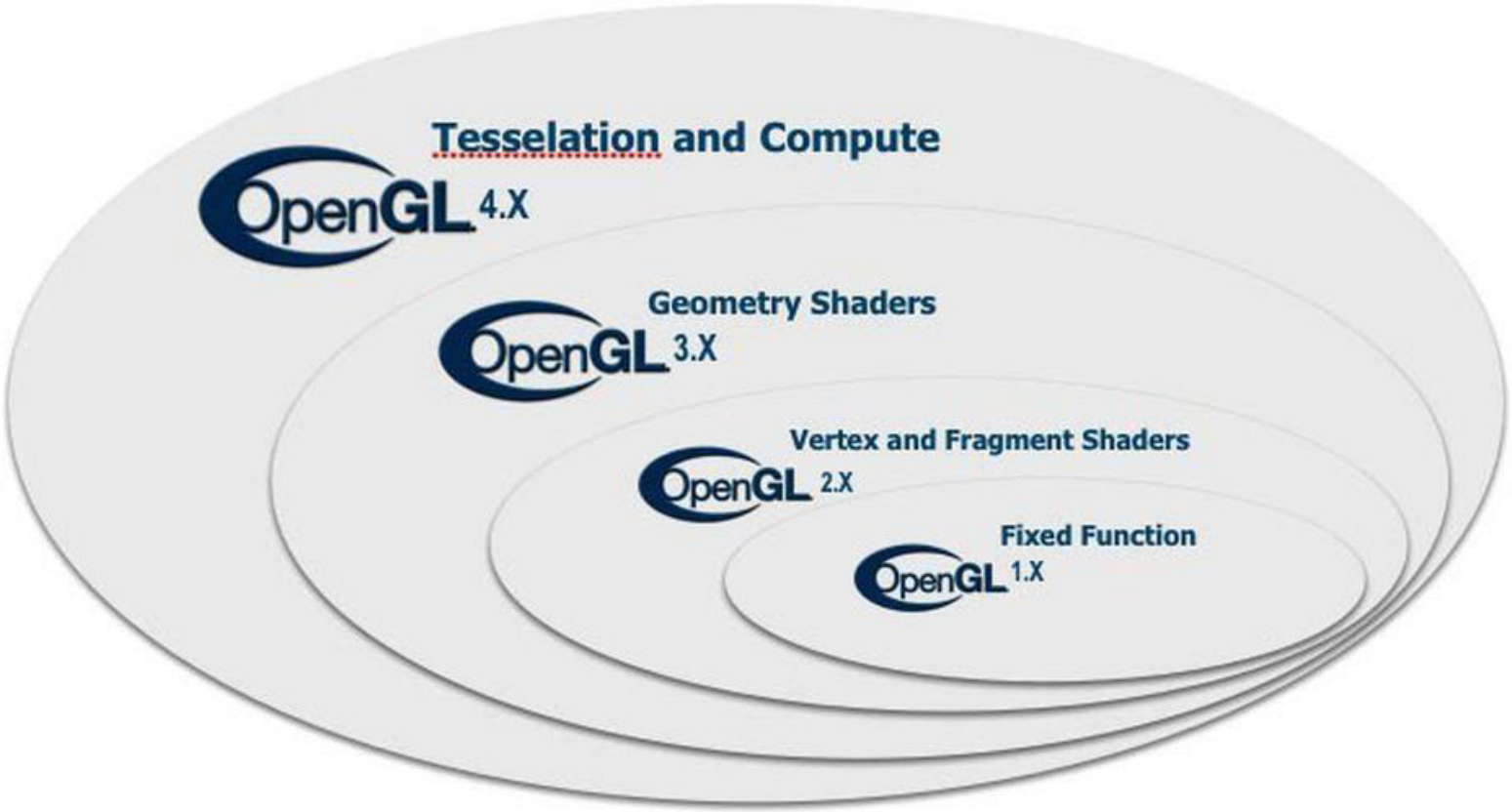
VBO

glManyFunctionsInCamelRegister

Що ми сьогодні розглянемо:

- Бібліотеки OpenGL та їх підключення
- Структура програми
- Створення вікна для рисування
- Нормалізовані координати
- Вершини та їх атрибути
- Зображення трикутника з нуля
- Шейдери
- Робота з об’єктами у графічній пам’яті

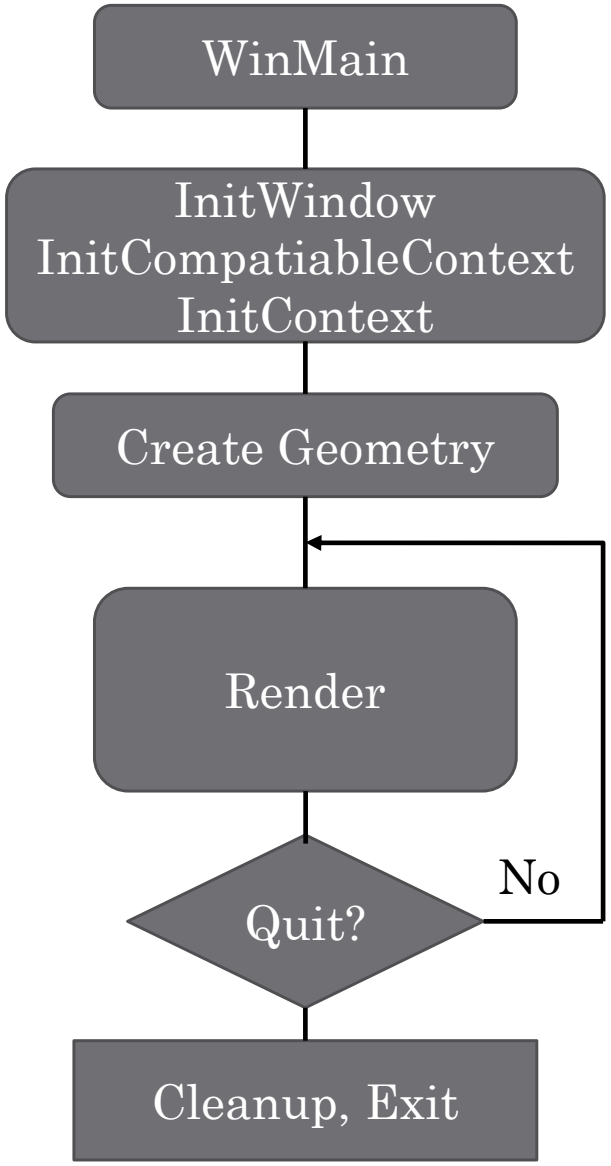
OpenGL – Open Graphics Library



Перш ніж створювати програму

- **!Оновіть драйвер відеокарти!**
- Встановіть середовище програмування
- Скачайте пакети бібліотечних файлів GLFW, GLEW
- Підключіть папки include та lib скачаних пакетів у вашому середовищі розробки
- Починаємо роботу!

Архітектура програми




```
#include <libraryheaders>

int main()
{
    createWindow(title, width, height);
    createOpenGLContext(settings);

    while (windowOpen)
    {
        while (event = newEvent())
            handleEvent(event);

        updateScene();

        drawGraphics();
        presentGraphics();
    }

    return 0;
}
```

1. Створення вікна

Створення вікна

- 1. Підключення бібліотек GLEW та GLFW

```
#include <iostream>

// GLEW
#define GLEW_STATIC
#include <GL/glew.h>

// GLFW
#include <GLFW/glfw3.h>
```

2. Функція main для створення вікна:

```
int main()
{
    return 0;
}
```

3. Для використання GLFW потрібно ініціалізувати перед запуском програми і знищити, після припинення роботи

```
glfwInit();
...
glfwTerminate();
```

Створення вікна (продовж.)

4.Створення і налаштування параметрів вікна:

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);

glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);

GLFWwindow* window = glfwCreateWindow(800, 600, "OpenGL", nullptr, nullptr); // Windowed
GLFWwindow* window =
    glfwCreateWindow(800, 600, "OpenGL", glfwGetPrimaryMonitor(), nullptr); // Fullscreen
```

Перевірка,
чи створилося вікно:

```
if (window == nullptr)
{
    std::cout << "Failed to create GLFW window" << std::endl;
    glfwTerminate();
    return -1;
}
```

5. Після створення вікна, потрібно активувати контекст OpenGL:

```
glfwMakeContextCurrent(window);
```

Створення вікна (продовж.)

6. Для того, аби OpenGL зміг використовувати сучасні графічні функції ініціалізуємо GLEW:

```
glewExperimental = GL_TRUE;
glewInit();
```

```
// Initialize GLEW to setup the OpenGL Function pointers
if (glewInit() != GLEW_OK)
{
    std::cout << "Failed to initialize GLEW" << std::endl;
    return -1;
}
```

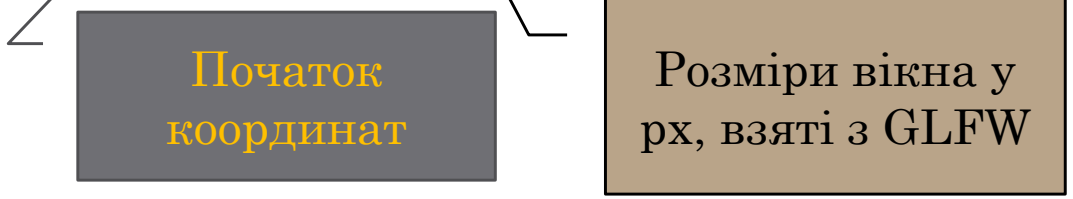
7. Після створення вікна, потрібно активувати контекст OpenGL:

```
glfwMakeContextCurrent(window);
```

Створення вікна (продовж.)

8. Перш ніж розпочати рендеринг, потрібно повідомити OpenGL розмір вікна для відображення (Viewport). Саме відносно нього потім масштабуються координати та інші парамерти графіки. Використовуємо для цього функціюкцию **glViewport**.

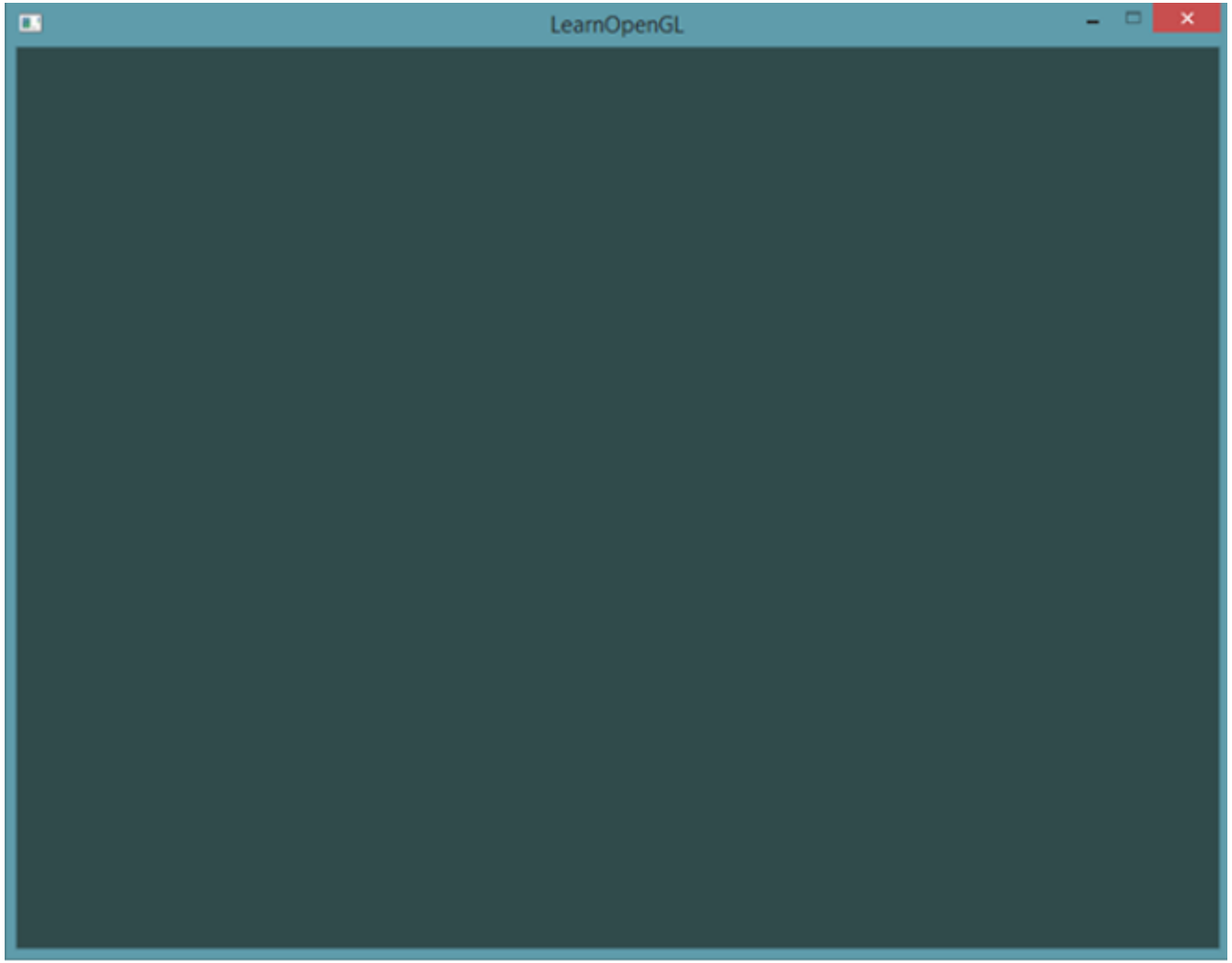
```
// Define the viewport dimensions
int width, height;
glfwGetFramebufferSize(window, &width, &height);
glViewport(0, 0, width, height);
```



Підготовка графічного рушія та рендерінг вікна

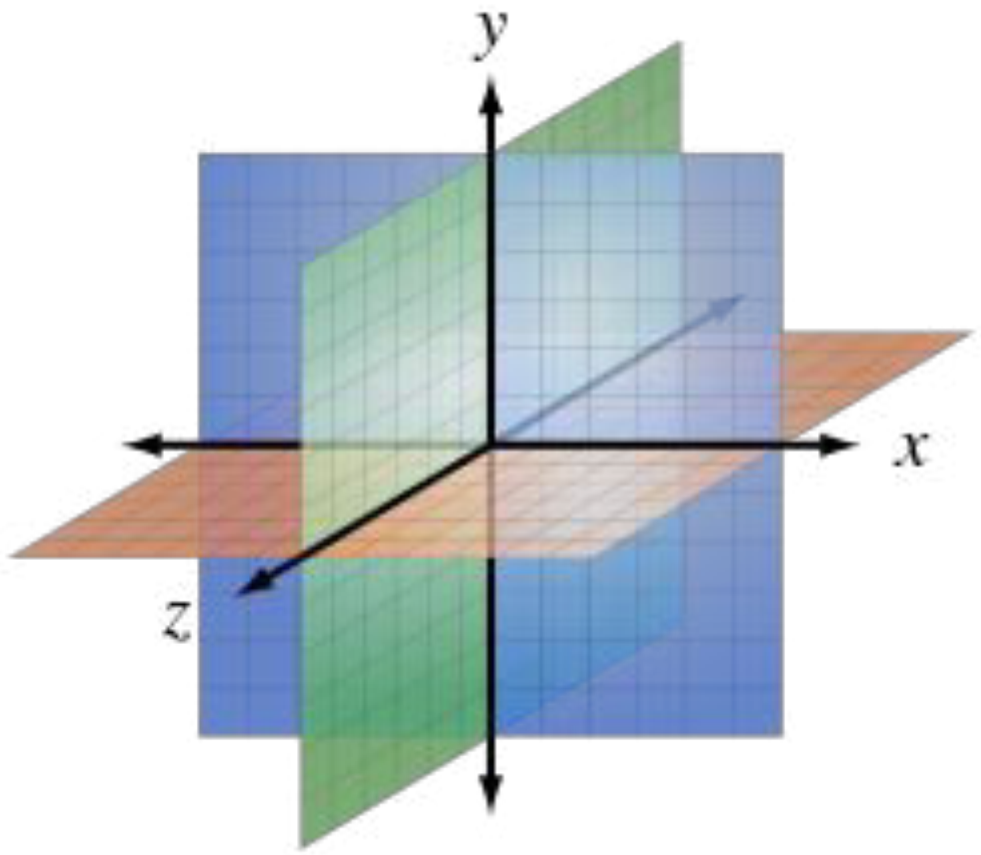
```
    // Game loop
    while (!glfwWindowShouldClose(window))
    {
        // Check if any events have been activiated (key pressed, mouse moved etc.)
        glfwPollEvents();

        // Render
        // Clear the colorbuffer
        glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);

        // Swap the screen buffers
        glfwSwapBuffers(window);
    }

    // Terminate GLFW, clearing any resources allocated by GLFW.
    glfwTerminate();
    return 0;
}
```

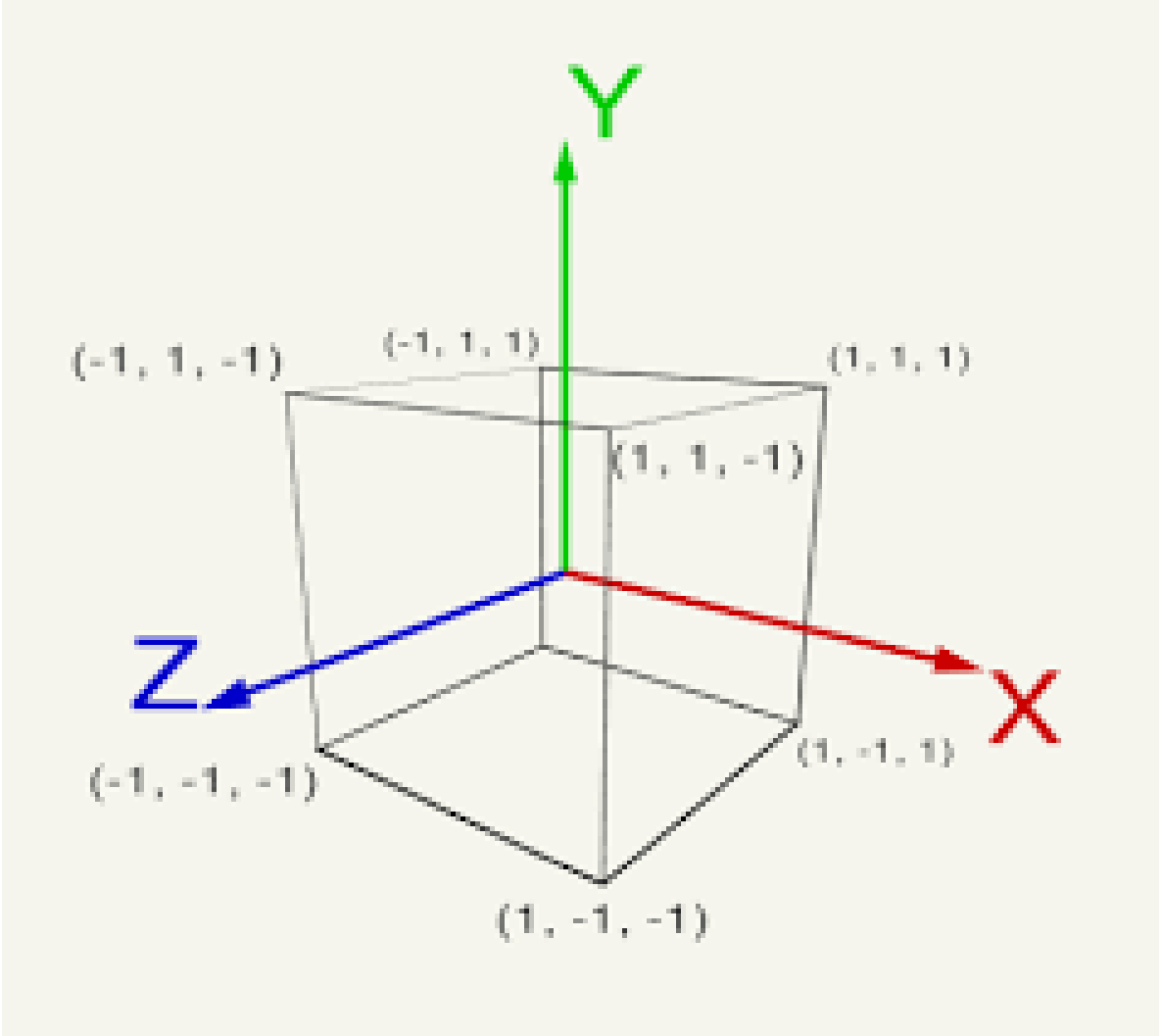
Перше вікно OpenGL



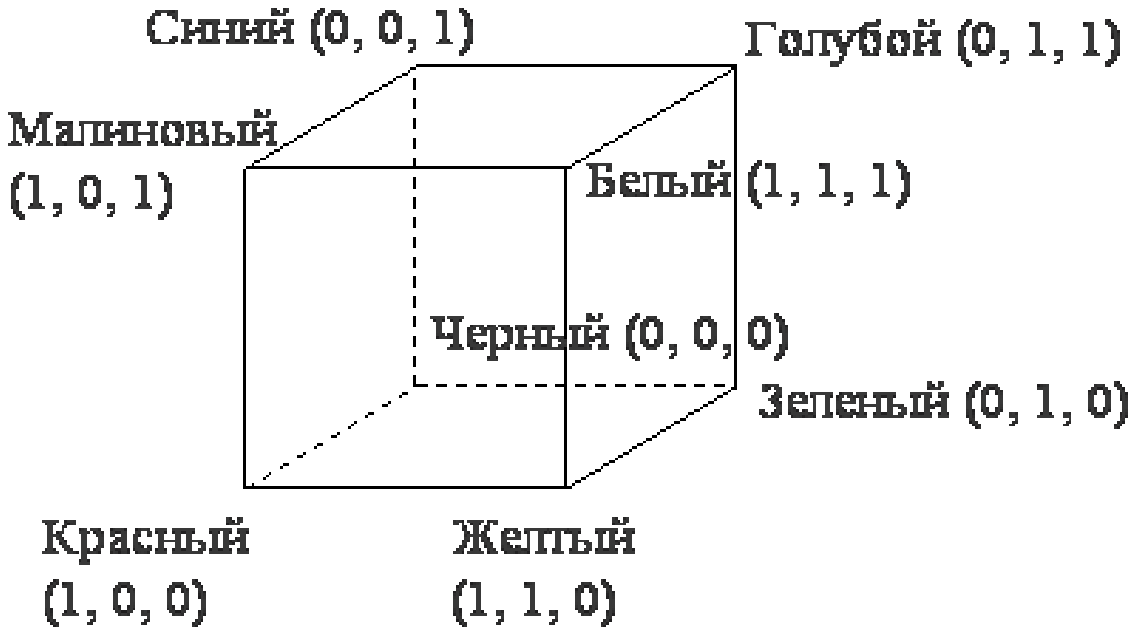

2. Параметри графіки

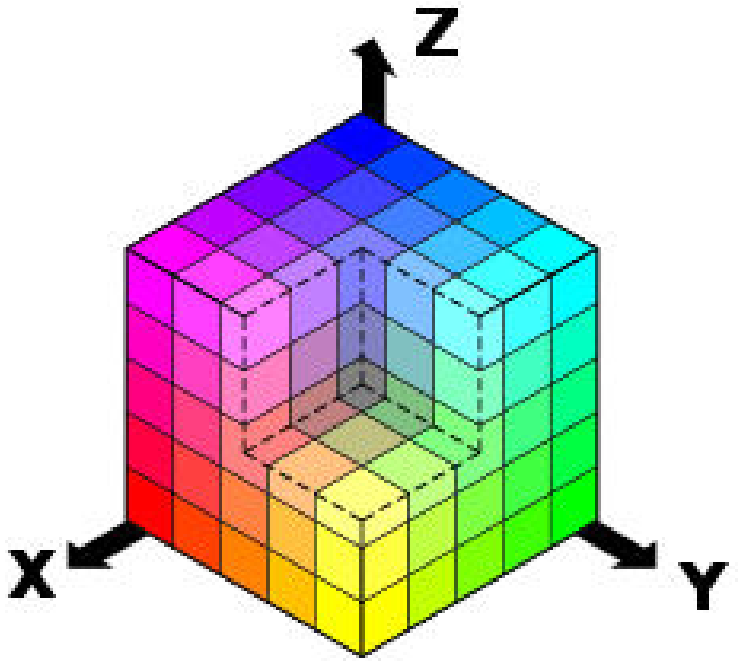
Нормалізовані координати пристрою



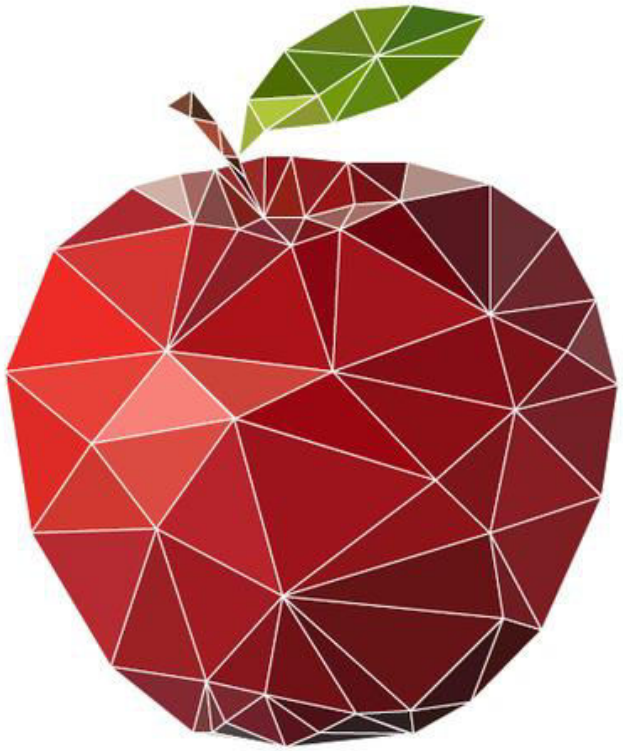


Кольорова модель RGB



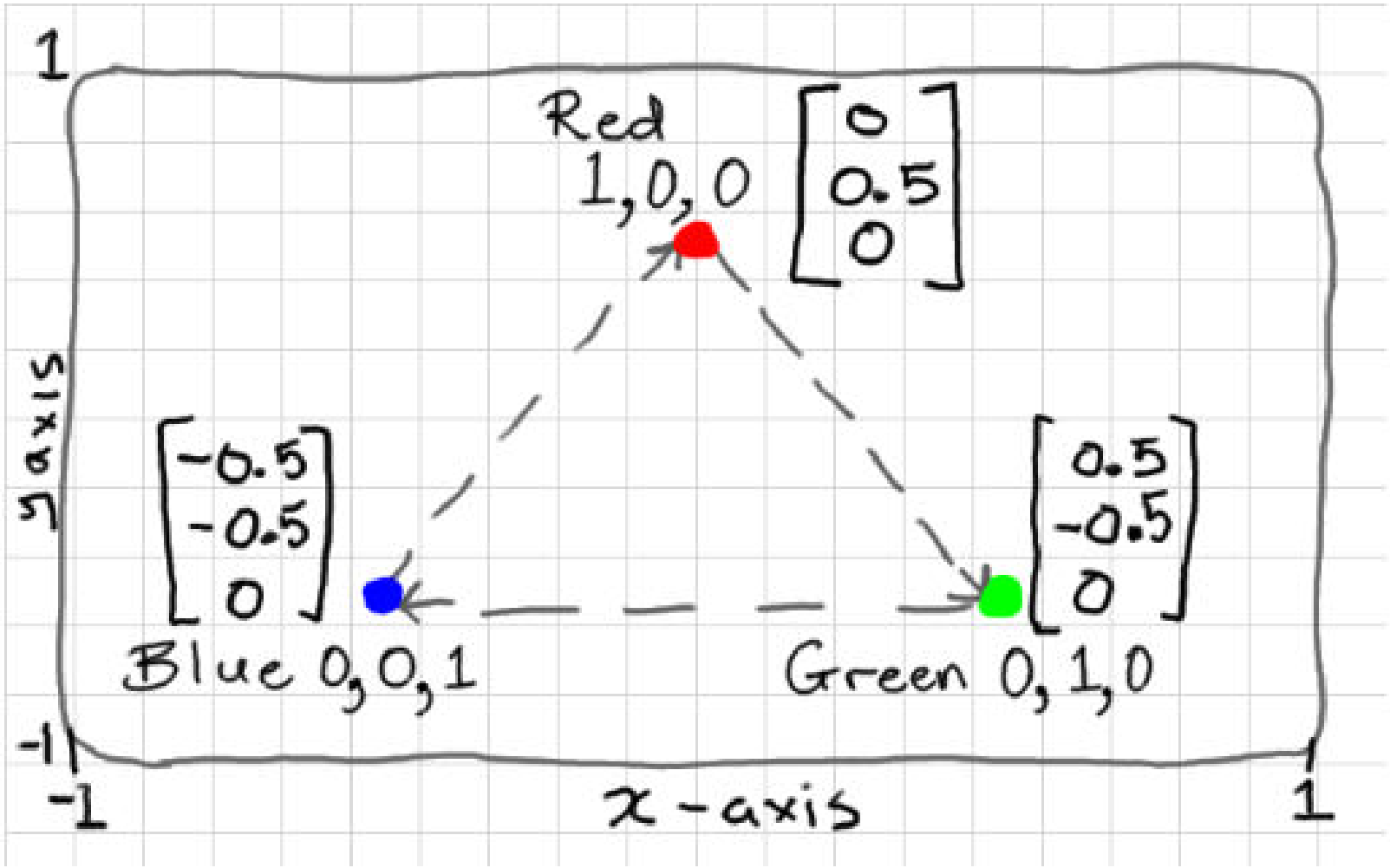


OpenGL працює з різними моделями, заданими за допомогою полігонів

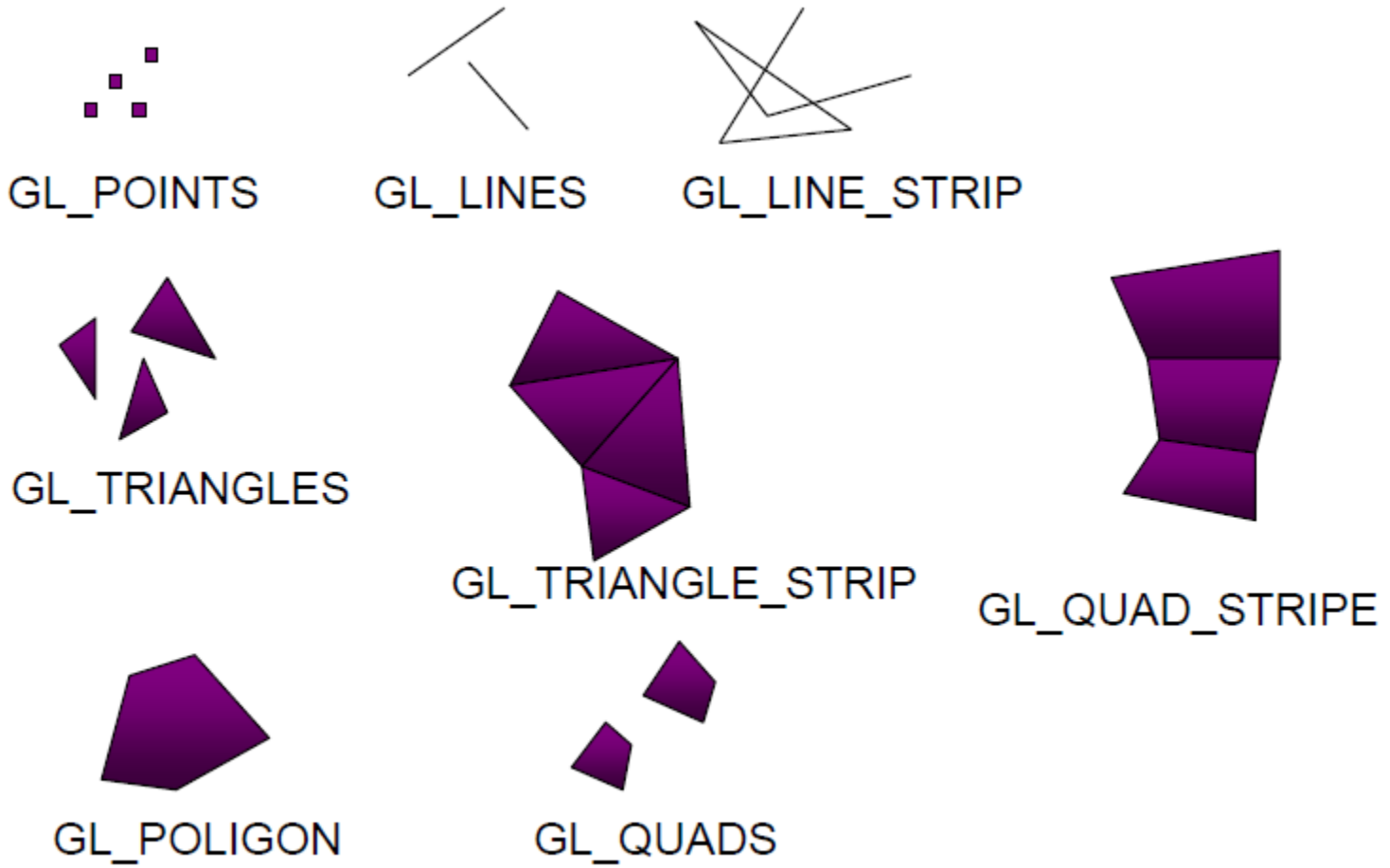


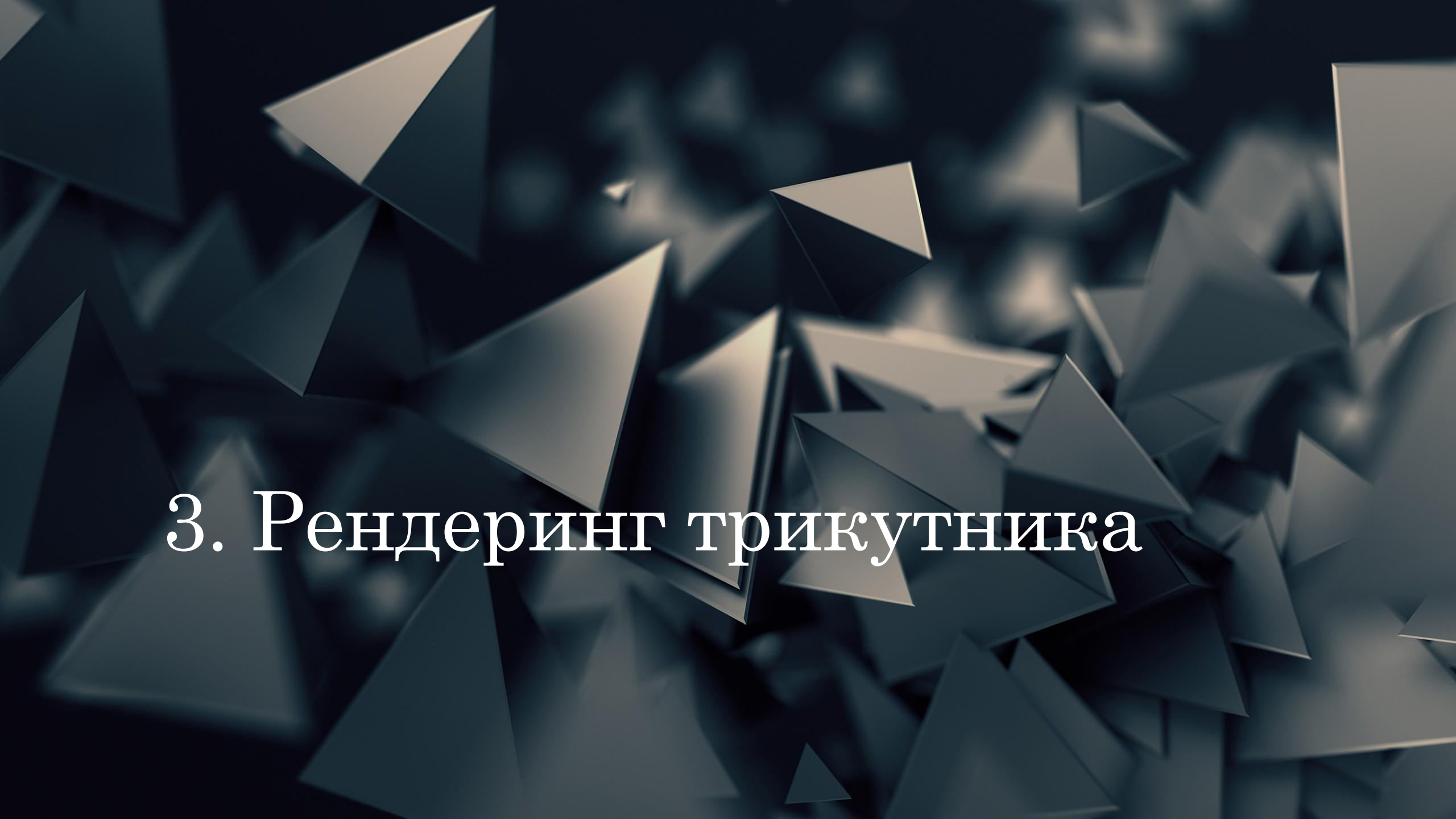
- Поверхня наближається за допомогою полігональних граней (face, polygon)
- Границі граней описуються ребрами (edge)
- Ребро обмежується вершинами (vertex)

Вершина (Vertex)



OpenGL може по-різному об’єднувати вершини у полігони



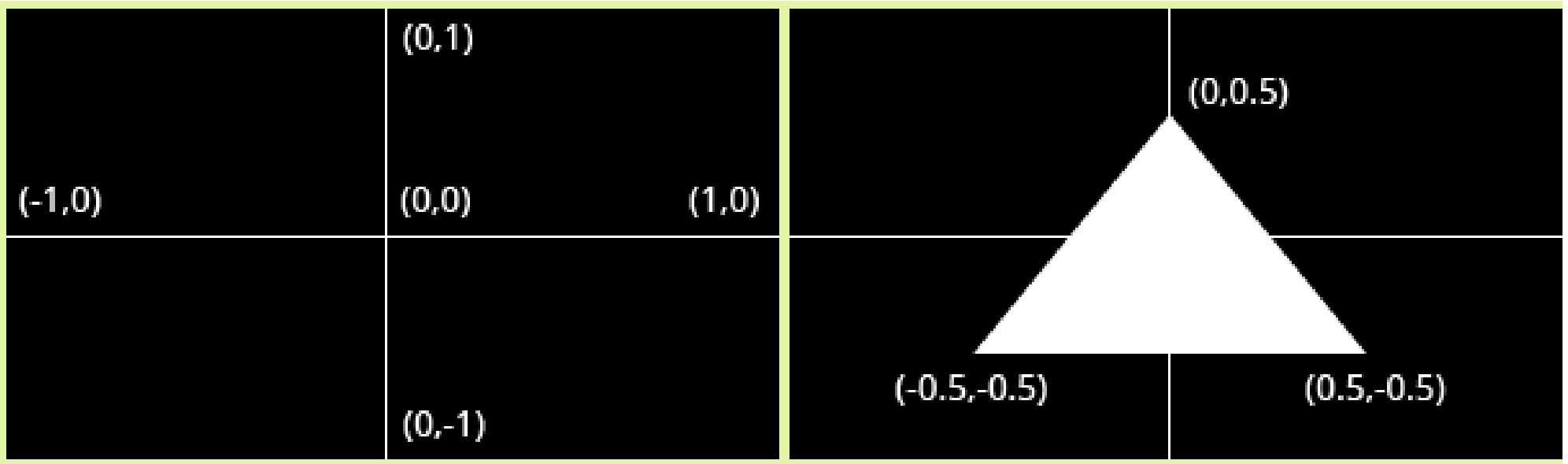


3. Рендеринг трикутника

Трикутник: old school

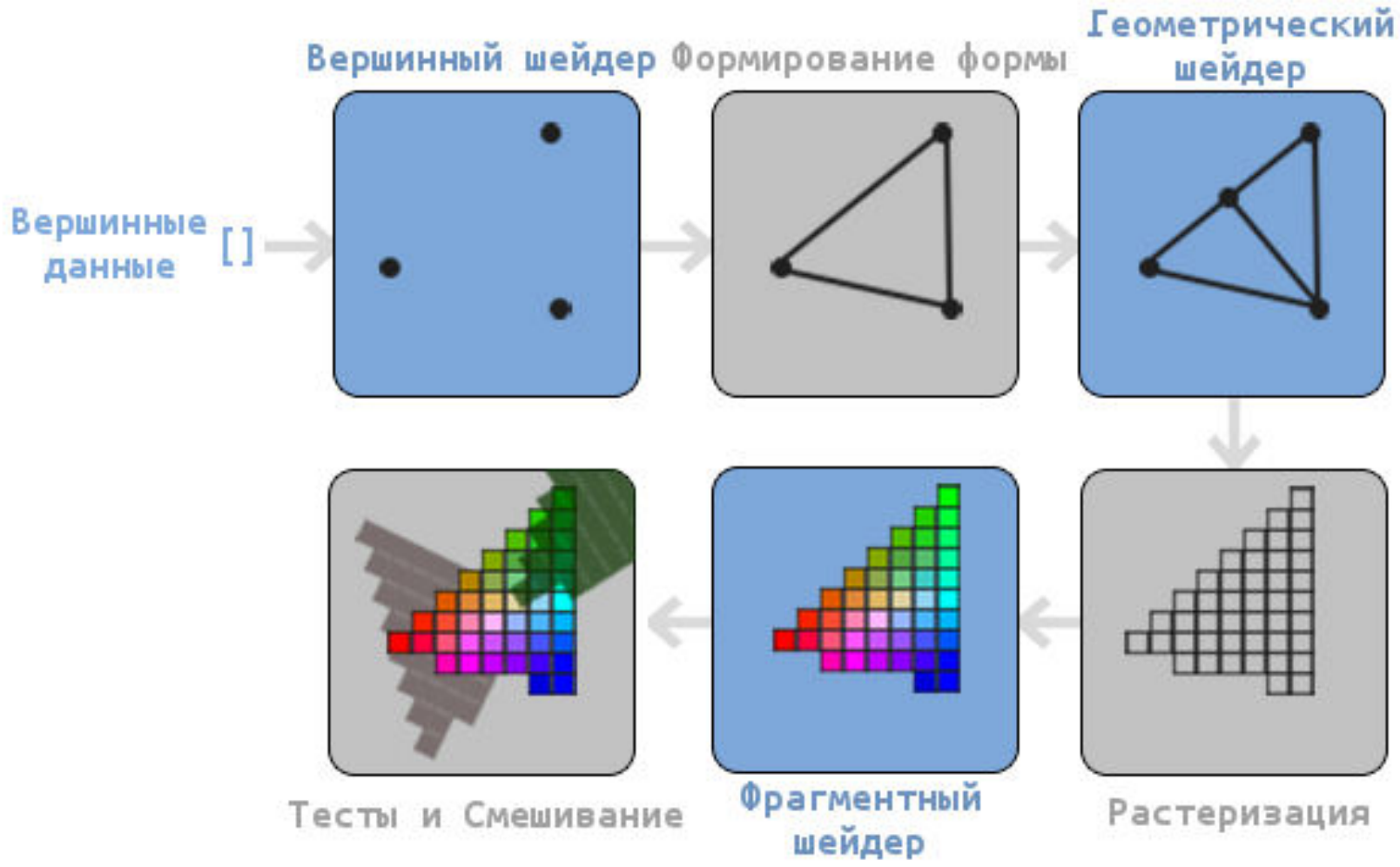
```
glBegin(GL_TRIANGLES);
   glColor3f(1.0f, 0.0f, 0.0f);
   glVertex3f(-1.0f, -1.0f, -2.0f);

   glColor3f(0.0f, 1.0f, 0.0f);
   glVertex3f( 0.0f,  1.0f, -2.0f);

   glColor3f(0.0f, 0.0f, 1.0f);
   glVertex3f( 1.0f, -1.0f, -2.0f);
glEnd();
```

Параметри трикутника

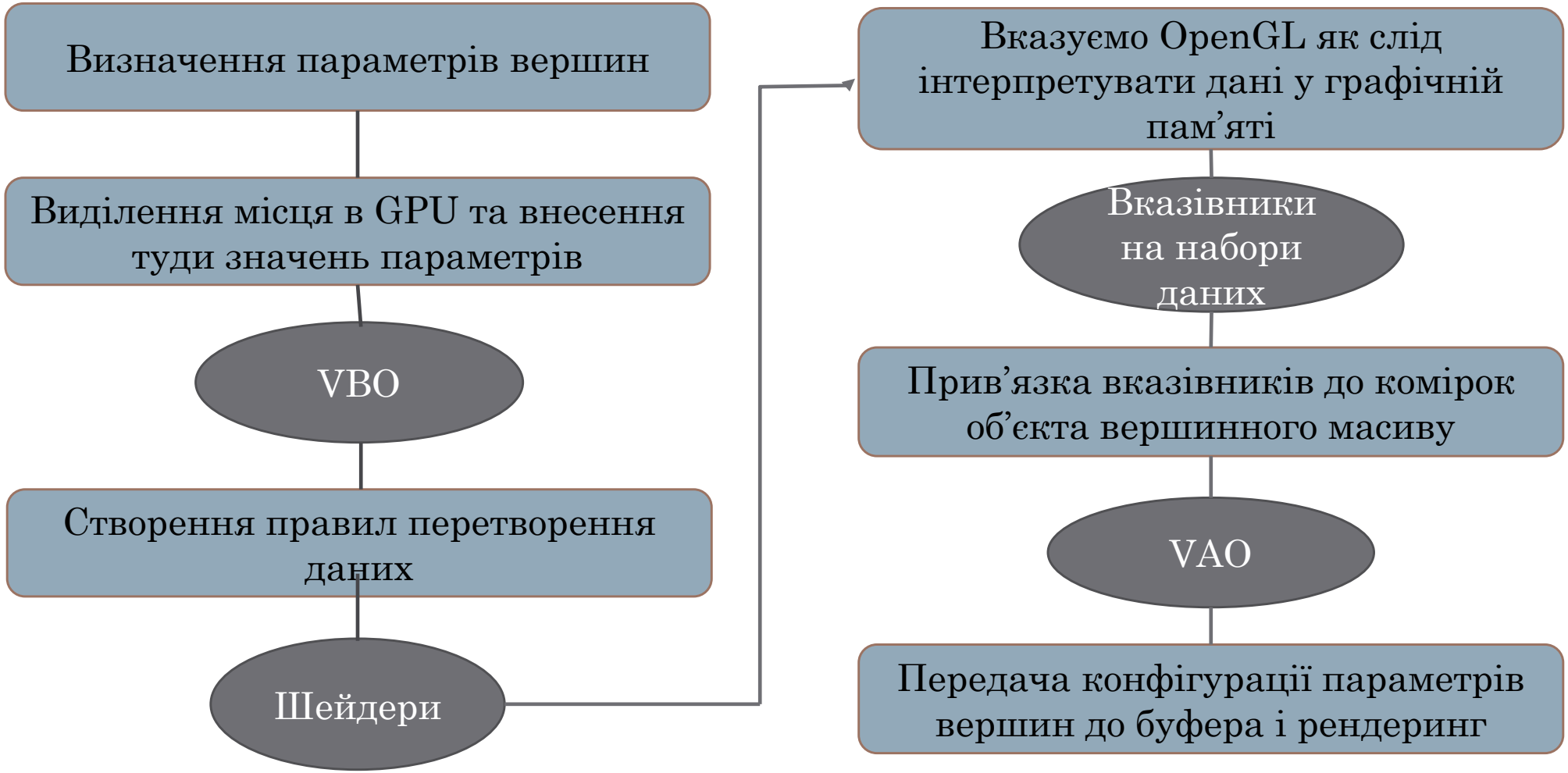



```
GLfloat vertices[] = { -0.5f, -0.5f, 0.0f,        //ліва вершина
                        0.5f, -0.5f, 0.0f,        // права вершина
                        0.0f, 0.5f,  0.0f         // верхня вершина
};
```

Графічний конвеєр



Процес рендерингу



Загрузка даних вершини до відеокарти

1. Створюємо VBO:

```
GLuint vbo;
glGenBuffers(1, &vbo); // Generate 1 buffer
```

2. Створєюмо буфер масивів:

```
glBindBuffer(GL_ARRAY_BUFFER, vbo);
```

3. Копіюємо дані вершин до буфера масивів:

```
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

Шейдер

- Шейдери — це невеликі програми, яі виконуються на графічному прискорювачі (GPU), для кожної конкретної ланки графічного конвейєра.
- Програми, які перетворюють входи у виходи.
- Програмуються на C-подібній мові GLSL (OpenGL Shading Language), яка адаптована для використання у графіці і надає функції для роботи з векторами і матрицями.
- Вершинний, геометричний, фрагментний шейдери

Вектор

- Vector в GLSL — це контейнер, який містить від 1 до 4 значень довільного базового типу.
- Оголошення контейнера vector (n — это кількість елементів вектора):

vecn — вектор (масив) з n значень типу float
bvecn —n значень типу boolean
ivecn — n значень типу integer
uvecn — вектор n значень типу unsigned integer
dvecn — n значень типу double.

Компоненти вектора

- Компоненти вектора мають назви (x, y, z, w) і до них можна звертатися окремо.
- Інші позначення компонентів для кольорів (r, g, b, a) або текстур (s, t, p, q).

```
vec2 someVec;
vec4 differentVec = someVec.xyxx;
vec3 anotherVec = differentVec.zyw;
vec4 otherVec = someVec.xxxx + anotherVec.yxzy;
```

```
vec2 vect = vec2(0.5, 0.7);
vec4 result = vec4(vect, 0.0, 0.0);
vec4 otherResult = vec4(result.xyz, 1.0);
```

Структура шейдера

```
#version version_number
in type in_variable_name;
in type in_variable_name;

out type out_variable_name;

uniform type uniform_name;

void main()
{
  // process input(s) and do some weird graphics stuff
  ...
  // output processed stuff to output variable
  out_variable_name = weird_stuff_we_processed;
}
```

Вершинний шейдер (Vertex shader)

- Версія шейдера = версія OpenGL (330 = 3.3, 420 = 4.2)
- layout (location=0) – положення першого вхідного параметра
- in vec3 aPos – вхідний параметр тривимірний вектор координат вершин
- gl_Position – визначена стандартна змінна типу vec4, яка є вихідним параметром вершинного шейдера

```
#version 330 core
layout (location = 0) in vec3 aPos;

void main()
{
    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
}
```

Фрагментний шейдер (Fragment shader)

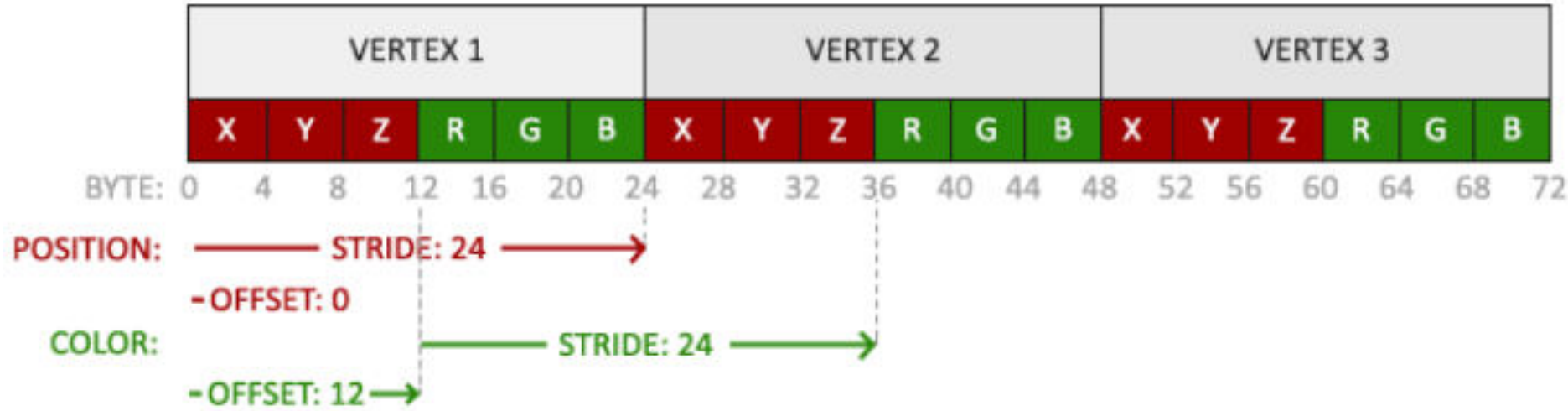
```
#version 330 core
out vec4 FragColor;

void main()
{
    FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);
}
```

Поєднання координат та кольорів вершин

```
GLfloat cvertices[] = { -0.5f, -0.5f, 0.0f,   1.0f, 0.0f, 0.0f,  //ліва вершина
                         0.5f, -0.5f, 0.0f,   0.0f, 1.0f, 0.0f,  // права вершина
                         0.0f, 0.5f,  0.0f,   0.0f, 0.0f, 1.0f   //верхня вершина
};
```

x y z
координати

R G B
кольори



Поєднання координат та кольорів вершин у шейдерах

Вершинний шейдер:

```
# version 330
layout (location=0) in vec3 inPosition;
layout (location=1) in vec3 inColor;
out vec3 ourColor;
void main()
{     gl_Position=vec4(inPosition, 1.0);
       ourColor=inColor;
}
```

Фрагментний шейдер

```
# version 330
in vec3 ourColor;
out vec4 color
void main()
{
        color=vec4(ourColor, 1.0f);
}
```

Збереження шейдерів

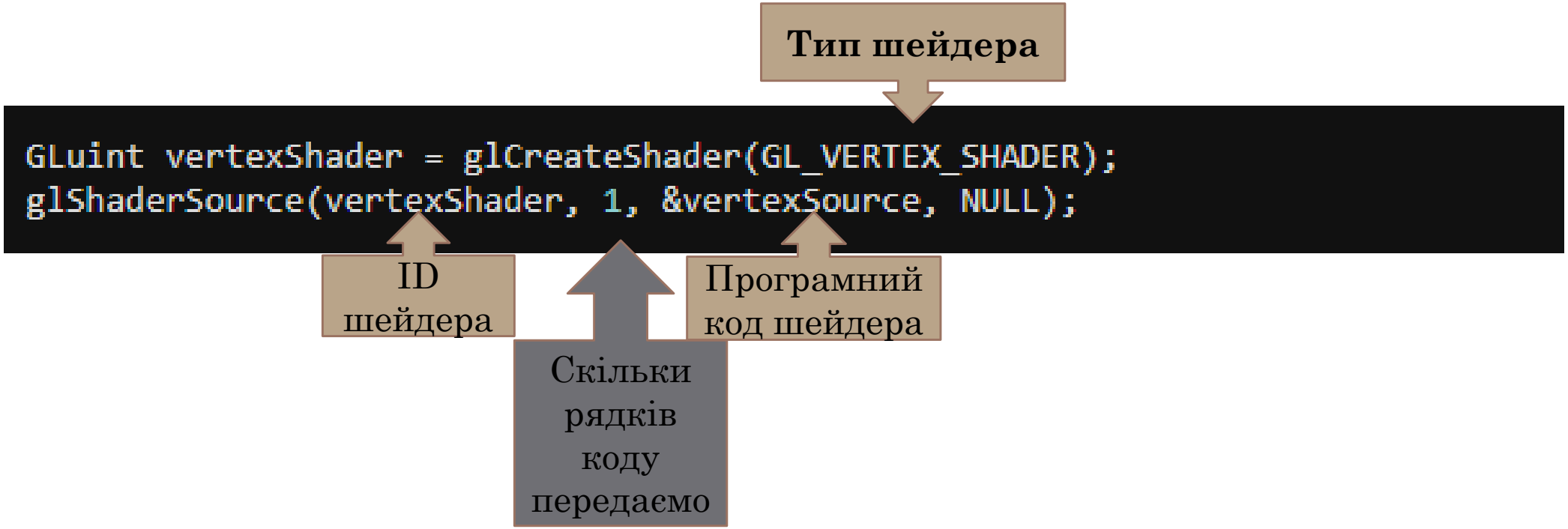
Код шейдерів зберігається у окремі іменованих рядках.

```
const char *vertexShaderSource = "#version 330 core\n"
    "layout (location = 0) in vec3 aPos;\n"
    "void main()\n"
    "{\n"
    "   gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n"
    "}\0";
const char *fragmentShaderSource = "#version 330 core\n"
    "out vec4 FragColor;\n"
    "void main()\n"
    "{\n"
    "   FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);\n"
    "}\n\0";
```

```
const char* vertexSource = R"glsl(
    #version 150 core

    in vec2 position;

    void main()
    {
        gl_Position = vec4(position, 0.0, 1.0);
    }
)glsl";
```

Створення об’єкта шейдера

- Створюємо об’єкт шейдера за допомогою **glCreateShader**, ідентифікатором якого буде змінна *vertexShader* типу **Gluint** і завантажуємо дані цього об’єкту через функцію **glShaderSource**:



Збір шейдера

- Компіляція вершинного шейдера:

```
glCompileShader(vertexShader);
```

- Якщо компіляція пройшла невдало, наприклад через синтаксичну помилку, повідомлення про це не видаватиметься.

- Перевірка успішності компіляції:

```
int  success;
char infoLog[512];
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
```

- Якщо статус true, то компіляція пройшла успішно, інакше в infoLog зберігатиметься log повідомлення про помилку (перші 511 символів):

```
if(!success)
{
    glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
}
```

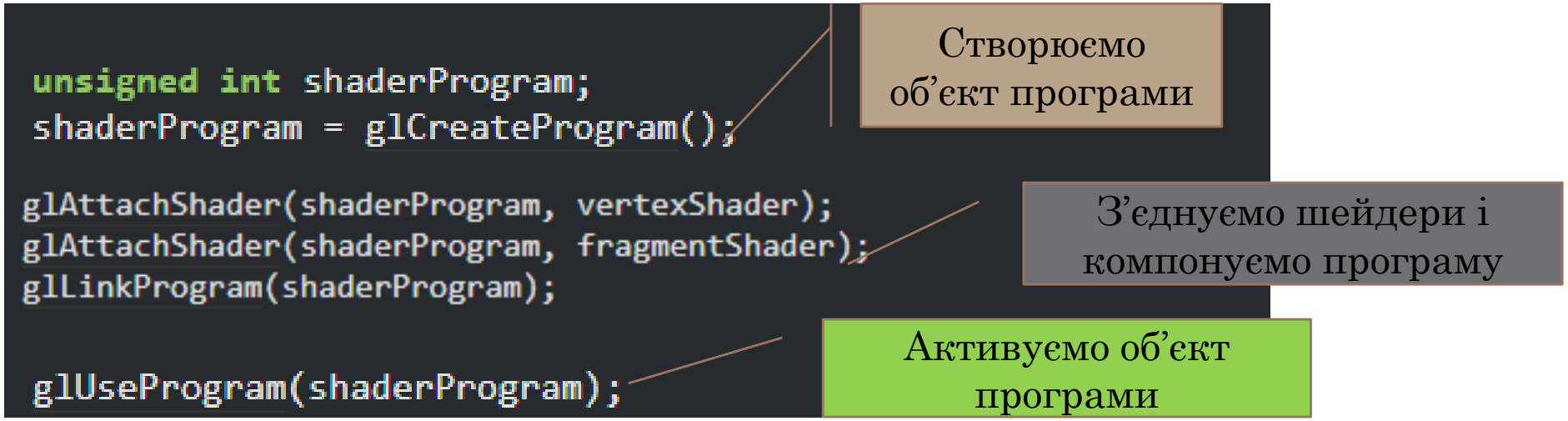
Компіляція фрагментного шейдера

- Тип шейдера: GL_FRAGMENT_SHADER
- Назва рядка коду: fragmentSource

```
GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentSource, NULL);
glCompileShader(fragmentShader);
```

Шейдерна програма

- Об’єкт шейдерної програми зв’язує разом різні шейдери
- Активується разом з шейдерами щоразу, коли потрібно рендерити (зобразити) графічні об’єкти.
- При компоновці шейдерів у програму, вихідні дані одного шейдера передаються на вхід наступного.
- Якщо виходи і входи не збігаються можуть виникати помилки компоновки.



Проміжні підсумки:

- Ми направили вхідні дані вершин до GPU (VBO)
- Вказали GPU як слід обробляти ці дані у вершинному та фрагментному шейдерах.
- Проте, OpenGL поки що не знає як інтерпретувати дані вершин у пам’яті і як їх пов’язати з атрибутами шейдера.
- Ми підкажемо OpenGL як це зробити.

Формат вершинного буфера (VBO)

- Значення кожної координати має тип float і зберігається на 4 байтах;
- Позиція кожної вержини формується з 3-х значень;
- Між наборами координат 3-х вершин немає прогалин (щильно упакований буфер);
- Перше значення, яке передається, - це початок буфера.

Специфікація вершинних даних

```
                        1   2     3          4                5                6
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
```

1 параметр вказує який аргумент шейдера ми хочемо налаштувати (0 позначає аргумент, для якого *layout (location = 0)*.

2 – розмір аргумента в шейдері, оскільки ми використовували **vec3**, то зазначаємо 3.

3 – тип даних у **vec3** - GL_FLOAT

4 – Чи потрібно нормалізувати координати

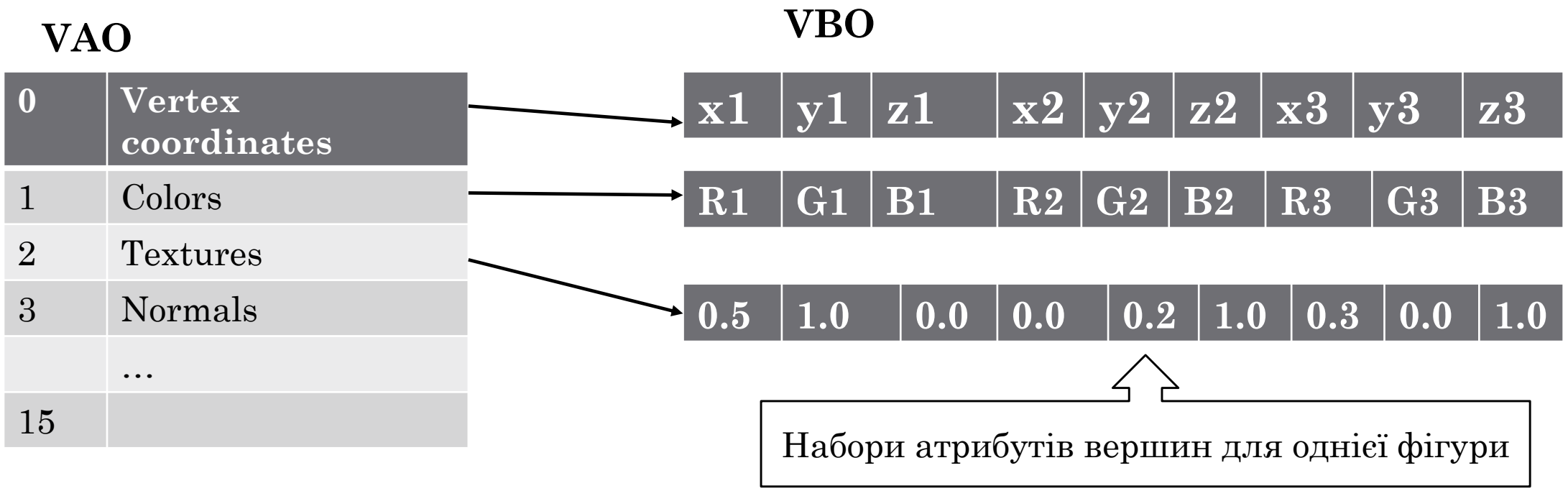
5 – крок, описує відстань між наборами даних

6 – зміщення початку даних буфера.

Зображення об’єкта в OpenGL

```
// 0. copy our vertices array in a buffer for OpenGL to use
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 1. then set the vertex attributes pointers
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// 2. use our shader program when we want to render an object
glUseProgram(shaderProgram);
// 3. now draw the object
someOpenGLFunctionThatDrawsOurTriangle();
```

Vertex Array Object (VAO)



Vertex Array Object (VAO)

```
GLfloat data[] = {
  // Position  // Color          // TexCoords
  1.0f, 0.0f,  0.5f, 0.5f, 0.5f, 0.0f, 0.5f,
  0.0f, 1.0f,  0.2f, 0.8f, 0.0f, 0.0f, 1.0f
};
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE,
        7 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
        7 * sizeof(GLfloat), (GLvoid*)(2 * sizeof(GLfloat)));
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE,
        7 * sizeof(GLfloat), (GLvoid*)(5 * sizeof(GLfloat)));
```

VAO[0], location 0

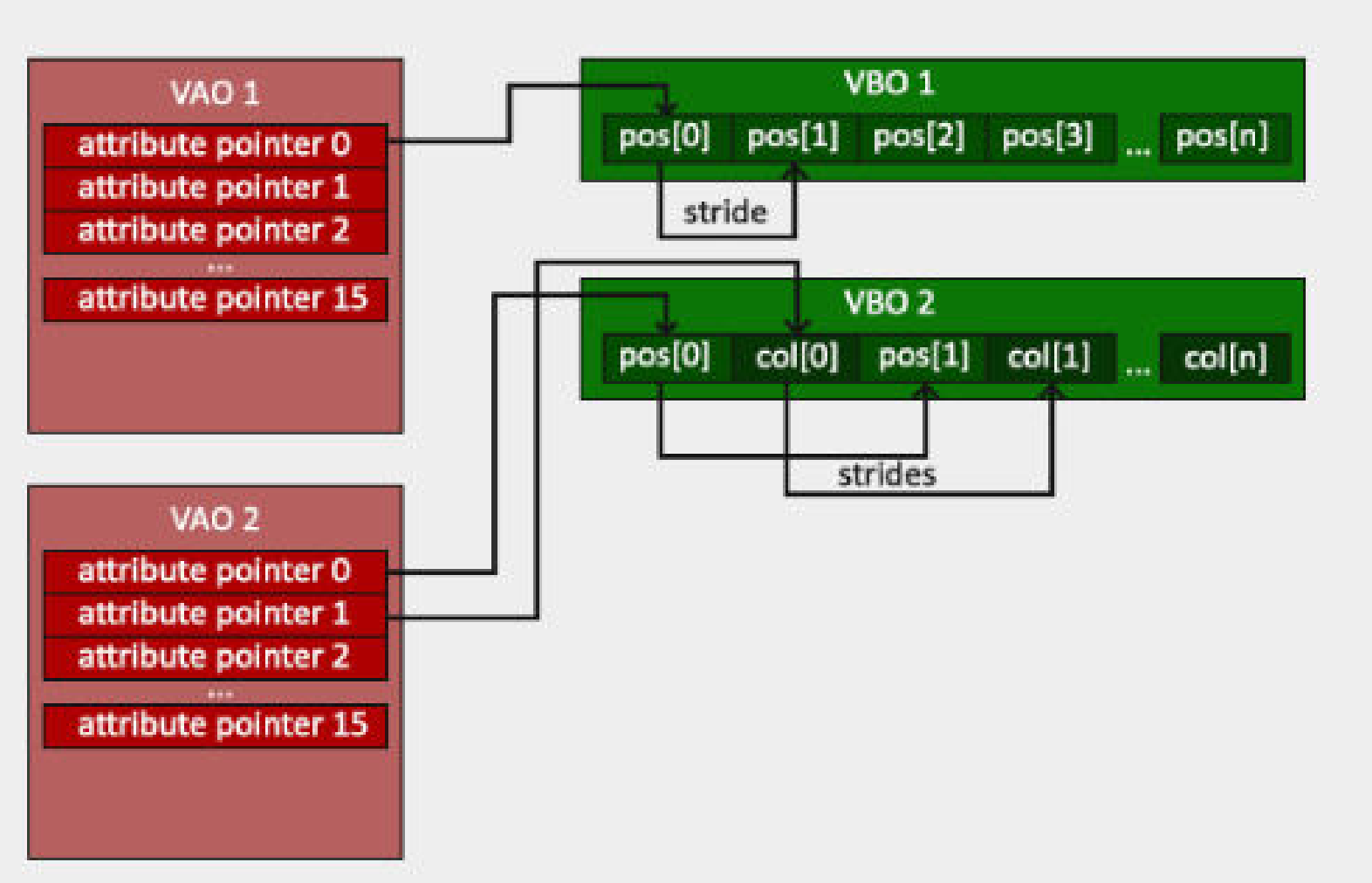
VAO[1], location 1

VAO[2], location 2

Функція **glEnableVertexAttribArray**(id) – активує відповідну комірку вершинного масиву (VAO), що відповідає деякому атрибуту вершини (0 – для набору координат, 1 – кольорів, 2 - текстур).

Вказівник, який повертає функція **glVertexAttribPointer**, прив’язується до комірки VAO з відповідним індексом.

VAO vs. VBO


VAO 1
attribute pointer 0
attribute pointer 1
attribute pointer 2
...
attribute pointer 15
VAO 2
attribute pointer 0
attribute pointer 1
attribute pointer 2
...
attribute pointer 15
VBO 1
pos[0]
pos[1]
pos[2]
pos[3]
...
pos[n]
stride
VBO 2
pos[0]
col[0]
pos[1]
col[1]
...
col[n]
strides

Використання VAO і VBO

- Створюємо VAO і зв’язуємо його за допомогою glBindVertexArray.
- Виділяємо місце у GPU і переносимо туди дані вершин (VBO).
- Одержуємо вказівник на визначений набір однотипних атрибутів вершин і прив’язуємо його до комірки VAO, попередньо її активувавши.

```
unsigned int VAO;
glGenVertexArrays(1, &VAO);
```

```
// ..:: Initialization code (done once (unless your object frequently changes)) :: ..
// 1. bind Vertex Array Object
glBindVertexArray(VAO);
// 2. copy our vertices array in a buffer for OpenGL to use
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3. then set our vertex attributes pointers
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);


[...]
```

Початок розміщення параметрів для рендерингу в пам’яті:

Якщо маємо такий вершинний шейдер і зібрану програму shaderProgram,

```
#version 330 core
layout (location = 0) in vec3 aPos;

void main()
{
    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
}
```

То аби передати початкове положення однотипних атрибутів вершини створюємо вказівник:

GLint positionAttribute = glGetAttribLocation(shaderProgram, «aPos");

glVertexAttribPointer(positionAttribute, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)0);

Here comes the Triangle

Для зображення примітивів OpenGL надає нам функцію glDrawArray, яка рисує фігуру за допомогою поточного шейдера. Конфігурація параметрів вершин передається через наперед визначений VAO.

glDrawArray(режим, поч_ID_VAO, кількість_вершин);

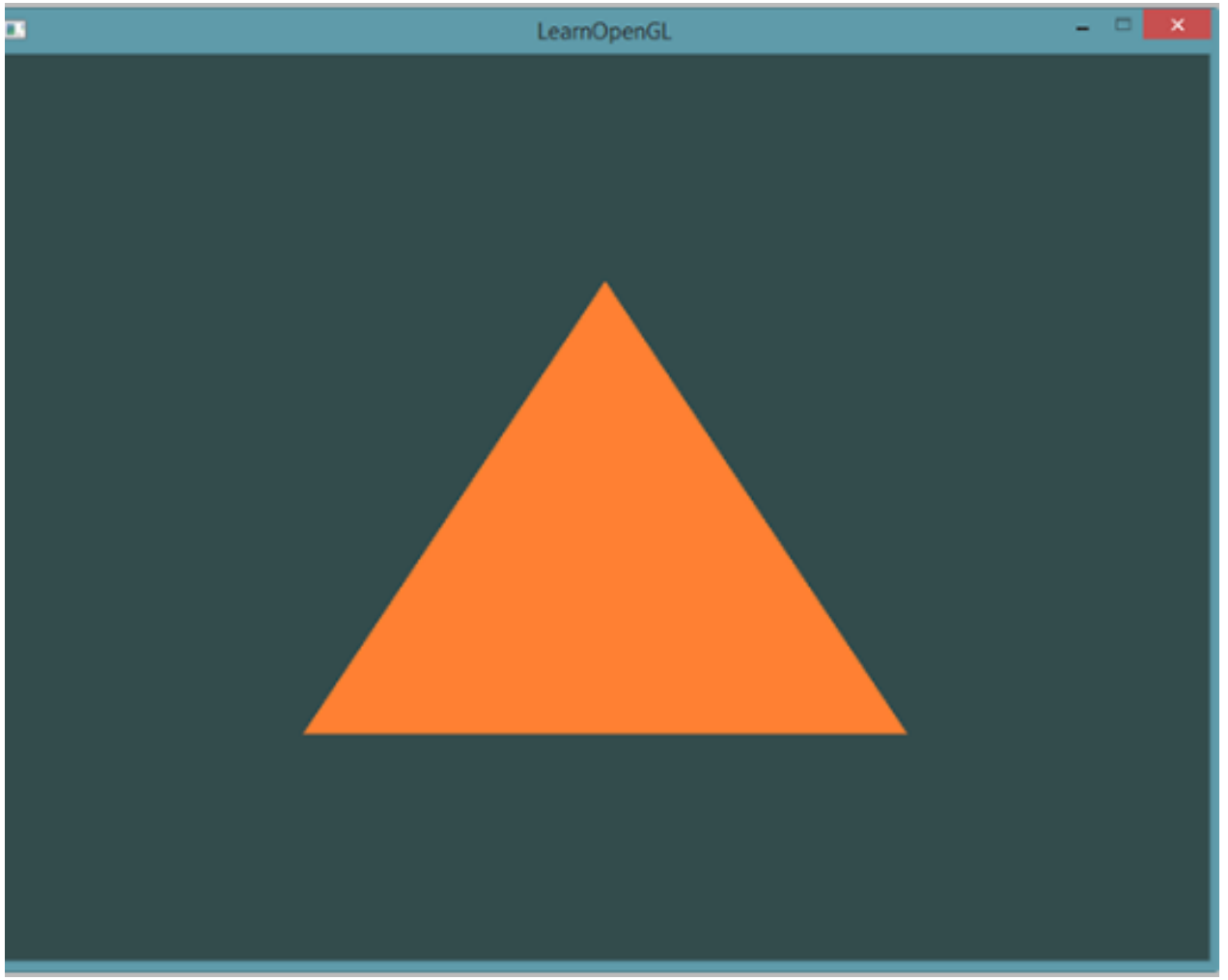
```
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Колір трикутника задається статично у фрагментному шейдері.

Рендеринг трикутника виконується у ігровому циклі (game loop)

```
// Game loop
while (!glfwWindowShouldClose(window))
{
    // Check if any events have been activiated
    glfwPollEvents();

    // Render
    // Clear the colorbuffer
    glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    // Draw our first triangle
    glUseProgram(shaderProgram);
    glBindVertexArray(VAO);
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glBindVertexArray(0);

    // Swap the screen buffers
    glfwSwapBuffers(window);
}
```

Here comes the Triangle



Різновиди трикутника

- Спробуйте замість функції glDrawArrays(GL_TRIANGLES, 0, 6); використати:

```
glEnable(GL_PROGRAM_POINT_SIZE);

glDrawArrays(GL_POINTS, 0, 6);
```

- Яким буде результат?

Доповніть код такими командами:

GL_FILL

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

glLineWidth(3.0f);

glDrawArrays(GL_TRIANGLES, 0, 6);
```

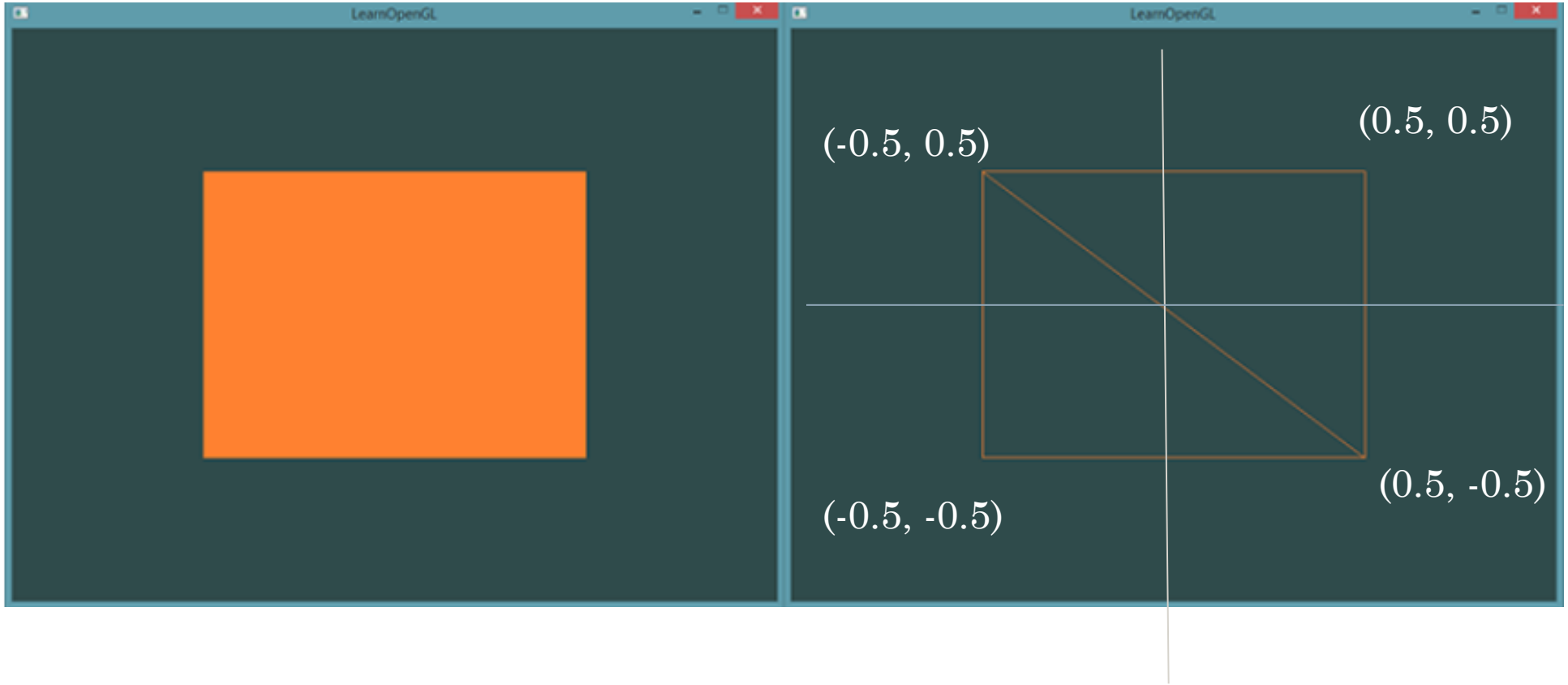
Прямокутник з двох трикутників

- Glfloat vertices [] = {
  - // Перший трикутник
  - 0.5f, 0.5f, 0.0f, // Верхня права вершина
  - 0.5f, -0.5f, 0.0f, //Нижня права вершина
  - -0.5f, 0.5f, 0.0f, //Верхня ліва вершина

  - //Другий трикутник
  - 0.5f, -0.5f, 0.0f, //Нижня права вершина
  - -0.5f, -0.5f, 0.0f, //Нижня ліва вершина
  - -0.5f, 0.5f, 0.0f //Верхня ліва вершина

  };

Прямокутник з двох трикутників



Element Buffer Object

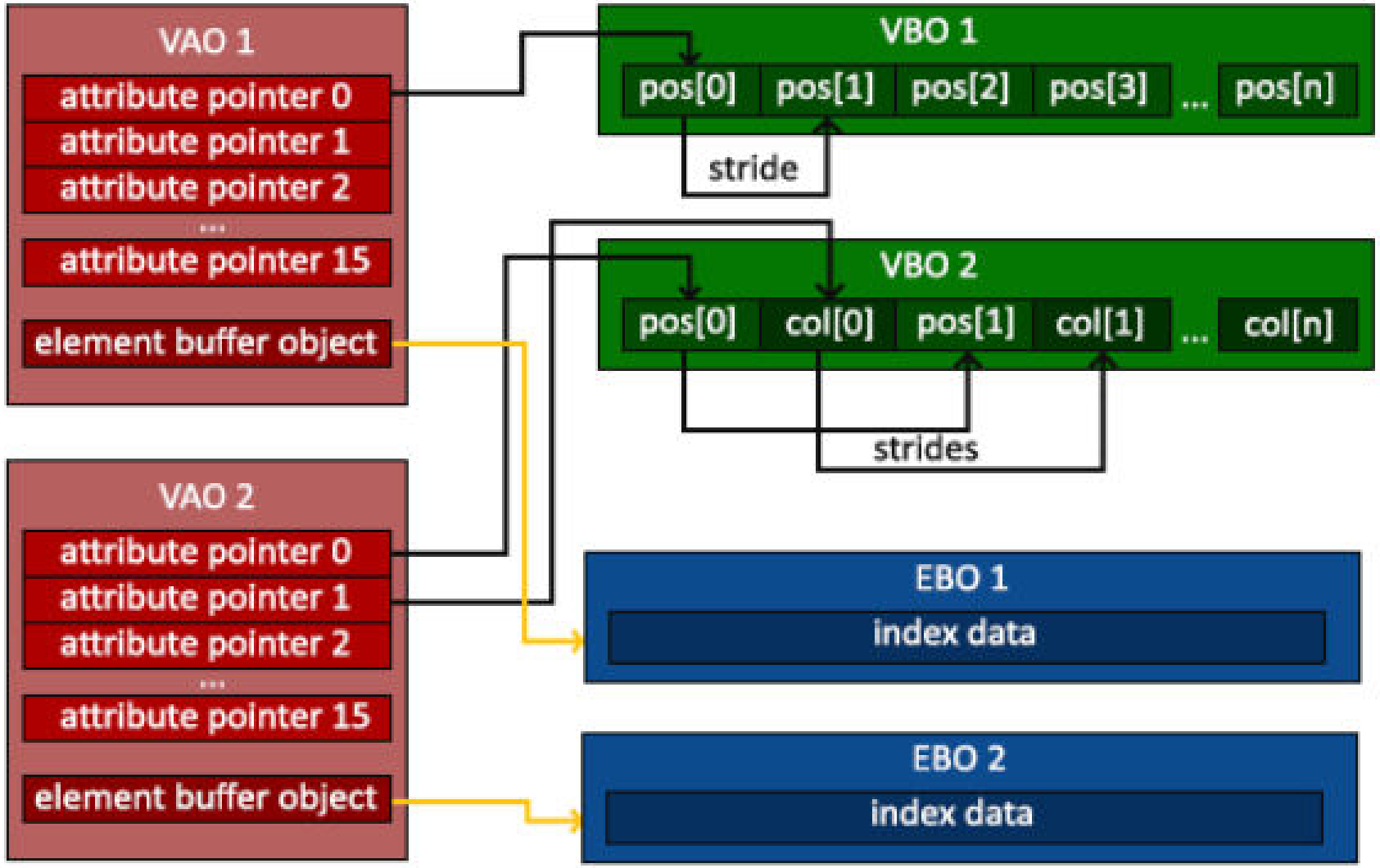
Це буфер на зразок VBO, який зберігає індекси унікальних координат, за якими OpenGL здійснюватиме рисування.

```
Glfloat vertices [] = {
        0.5f, 0.5f, 0.0f,       // 0
        0.5f, -0.5f, 0.0f,      // 1
        -0.5f, -0.5f, 0.0f,     // 2
        -0.5f, 0.5f, 0.0f       // 3
        };
```

```
Gluint indices[] ={
0, 1, 3, //Перший трикутник
1, 2, 3 //Другий трикутник
};
```

Передача даних у графічну пам’ять

```
GLuint VBO, VAO, EBO;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);
glGenBuffers(1, &EBO);
// Bind the Vertex Array Object first, then bind and set vertex buffer(s) and attri
glBindVertexArray(VAO);

glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);

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


VAO 1
attribute pointer 0
attribute pointer 1
attribute pointer 2
...
attribute pointer 15
element buffer object
VBO 1
pos[0]
pos[1]
pos[2]
pos[3]
...
pos[n]
stride
VBO 2
pos[0]
col[0]
pos[1]
col[1]
...
col[n]
strides
VAO 2
attribute pointer 0
attribute pointer 1
attribute pointer 2
...
attribute pointer 15
element buffer object
EBO 1
index data
EBO 2
index data

Рендеринг прямокутника через EBO

```
// Draw our first triangle
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
//glDrawArrays(GL_TRIANGLES, 0, 6);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
glBindVertexArray(0);

// Swap the screen buffers
glfwSwapBuffers(window);
```

Домашнє завдання:

1. Створити вікно та кольоровий трикутник

2. Розібратися, що таке EBO.

3. Зобразити прямокутник, який складається з двох трикутників, з використанням EBO.

4. Створіть два та більше трикутників, кожен свого розміру та кольору.

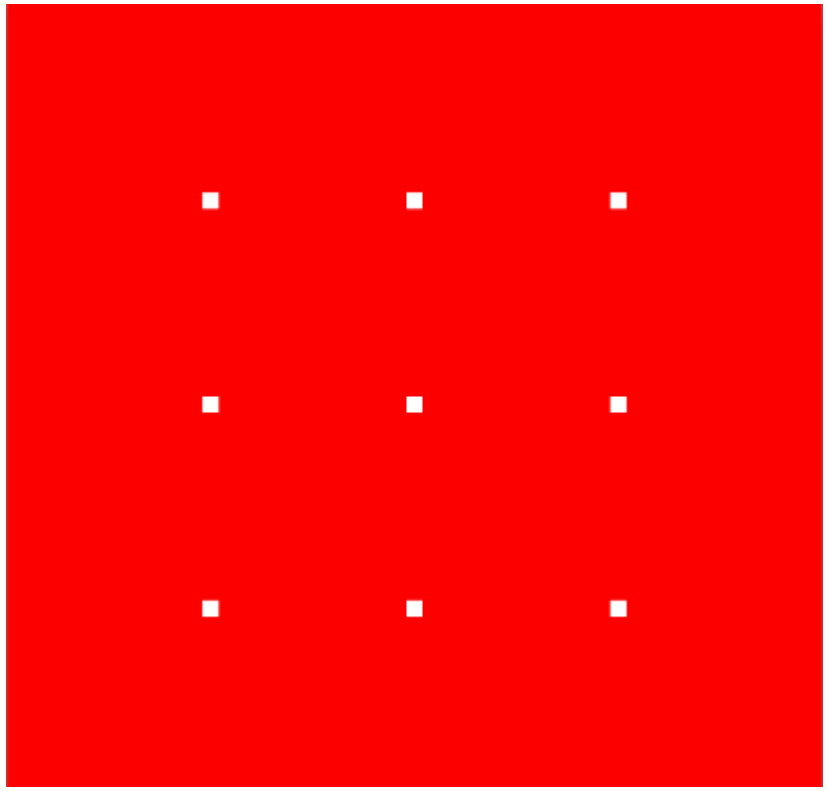
5. Ознайомитись з матеріалами та вправами Tutorial Learn OpenGL, уроки 1.1. – 1.5.

Додаткові завдання:

- A. Перевернути трикутник у вершинному шейдері.
- B. У фрагментному шейдері замініть на протилежний колір фігури, який отримується з вершинного шейдера.
- C. На наступному слайді дано набір координат вершин. Які фігури з трикутників можна зобразити з їх допомогою?
- Створіть декілька фігур, використавши для них різні режими рисування (суцільна заливка, каркас, точки).
- Задайте колір фігур: у масиві параметрів вершин, як Uniform, заповніть фігури суцільним кольором, окремим кольором для кожної вершини, випадковою комбінацією кольорів.
- Використовуйте шейдери, VBO, VAO, EBO.

- Найоригінальніші фігури будуть окремо відзначені!

```
Glfloat vertices_pos [12] =     {0.0, 0.0,
                                 0.5, 0.0,
                                 0.5, 0.5,
                                 0.0, 0.0,
                                 0.0, 0.5,
                                 -0.5, 0.5,
                                 0.0, 0.0,
                                 -0.5, 0.0,
                                 -0.5, -0.5,
                                 0.0, 0.0,
                                 0.0, -0.5,
                                 0.5, -0.5
};
```



Довідкові джерела:

1. https://habr.com/post/311808/

2. https://habr.com/post/313380/

3. https://learnopengl.com/Getting-started/Hello-Triangle

4. https://open.gl/drawing

5. Уроки по OpenGL: https://triplepointfive.github.io/ogltutor/

6. OpenGL на C#. Быстрый старт.
https://www.youtube.com/watch?v=Ih7gO5TU6dU

7. OpenGL 3D Game Tutorial in Java:

https://www.youtube.com/watch?v=VS8wlS9hF8E&list=PLRIWtICgwaX0u7Rf9zkZhLoLuZVfUksDP

Keep calm and learn OpenGL