Grafika komputerowa
Wykład 7

OpenGL - wprowadzenie

dr inż. Michał Chwesiuk
Michal.Chwesiuk@pw.edu.pl

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Warszawa, 12.04.2022 r.

Plan prezentacji

- Wprowadzenie
- Przygotowanie danych
- Programy cieniujące
- Rysowanie trójkąta
- Prymitywy
- Zmienne jednolite
- Transformacja 3D

Wprowadzenie

Open Graphics **L**ibrary

- API pozwalające na wykorzystanie akceleracji sprzętowej do renderowania grafiki czasu rzeczywistego.
- OpenGL nie jest biblioteką! Jest to zbiór funkcji i numerycznych stałych, które są interpretowane przez system operacyjny i/lub sterownik karty graficznej.
- Działanie OpenGL można rozumieć jako **maszynę stanu**, raz ustawiona wartość parametru systemu pozostaje taka sama do momentu jej zmiany.
- Główną zaletą jest wieloplatformowość.
- Konkurencja: Microsoft DirectX i Khronos Group Vulcan
- **Nie obejmuje obsługi wejścia i integracji z systemem operacyjnym w celu utworzenia kontekstu!**

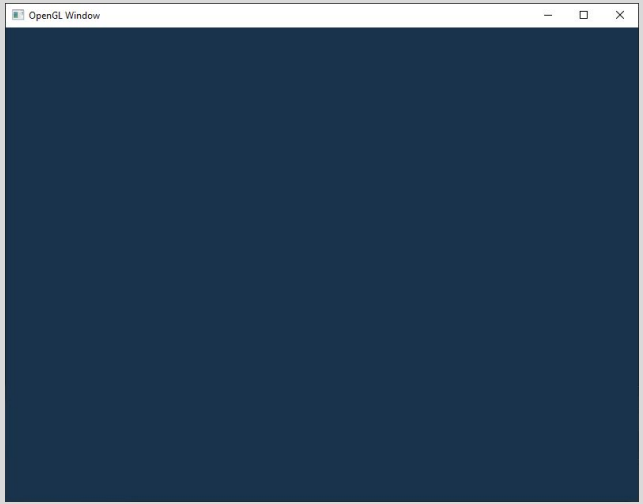
Biblioteki

- **GLFW**
  - Tworzenie okna, obsługa urządzeń wejścia
- **GLEW**
  - Wspomaganie obsługi OpenGL oraz jego rozszerzeń
- **Assimp**
  - Wczytywanie modeli 3D
- **GLM**
  - Operacje wektorowe i macierzowe
- **STB Image**
  - Wczytywanie plików graficznych w celu teksturowania obiektów

Przykład kodu OpenGL

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

int main()
{
   GLFWwindow* window;

   glfwInit();

   glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
   glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 0);

   window = glfwCreateWindow(800, 600, "OpenGL Window", NULL, NULL);

   glfwMakeContextCurrent(window);

   glewInit();

   while (!glfwWindowShouldClose(window))
   {
      glClearColor(0.1f, 0.2f, 0.3f, 1.0f);
      glClear(GL_COLOR_BUFFER_BIT);

      glfwSwapBuffers(window);
      glfwPollEvents();
   }
}
```

- Podstawowy kod w OpenGL otwierający puste okno.
- Nie zawiera renderingu żadnego obiektu.



Wersje OpenGL

- Na zajęciach będzie omówiony **OpenGL 3.0+**, który jest obecnie standardem (aktualna wersja to 4.6). Wersja ta wymaga od programisty **zdefiniowania potoku renderowania**.
  - Własne programy cieniujące (**shadery**).
  - Możliwość dowolnej obróbki pikseli zanim trafią do framebuffer'a.
  - Zaawansowane techniki grafiki czasu rzeczywistego (dynamiczne cienie, deferred shading, postprocessing).
- Alternatywą jest OpenGL do wersji 2.0+, która jest przestarzała, ale umożliwia łatwiejsze wejście do zagadnień renderingu grafiki komputerowej.
  - **Fixed Pipeline** - możliwość skorzystania z domyślnych shader'ów.
  - **Immediate Mode** - dostarczanie danych o wierzchołkach z poziomu CPU.

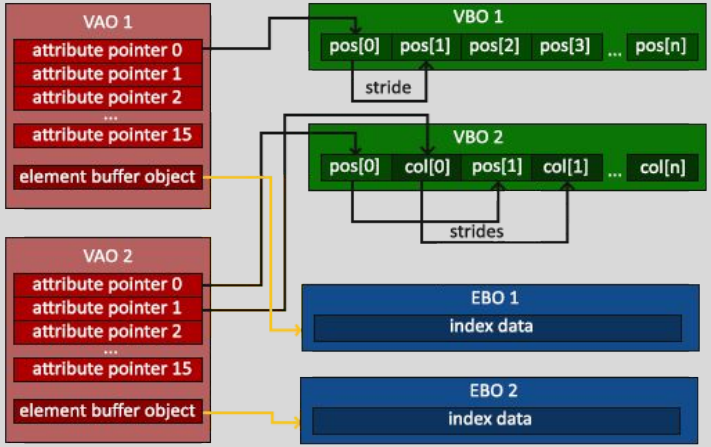
WYDZIAŁ ELEKTRONIKI I TECHNIK INFORMACYJNYCH

Vertex Buffer Object

- **VBO** jest to obiekt w OpenGL, który pozwala na załadowanie do pamięci GPU bufora danych.
- Dane przesłane do bufora dotyczą parametrów wierzchołków. Nazywame je **atrybutami**. Są to:
  - Pozycja
  - Kolor
  - Wektor
  - Współrzędna tekstury

```
GLuint VBO;

std::vector<glm::vec2> positions; // Przykładowe dane
positions.push_back(glm::vec2(-0.5f, -0.5f));
positions.push_back(glm::vec2(0.5f, -0.5f));
positions.push_back(glm::vec2(0.0f, 0.5f));

glGenBuffers(1, &VBO); // Generowanie VBO

glBindBuffer(GL_ARRAY_BUFFER, VBO); // Przypisanie aktywnego VBO

glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec2) * positions.size(),
    positions.data(), GL_STATIC_DRAW); // Kopiowanie danych do GPU
```

normalny



Vertex Array Object

- **VAO** jest to obiekt w OpenGL, który zawiera wszystkie dane o wierzchołkach.
- Do programisty należy wskazanie gdzie w obrębie struktury danych znajdują się atrybuty wierzchołka.

```
GLuint VAO;

glGenVertexArrays(1, &VAO); // Generowanie VAO

glBindVertexArray(VAO); // Przypisanie aktywnego VAO

glVertexAttribPointer
 (
    0, // Identyfikator atrybutu (layout w vertex shader)
    2, // Ilość danych w dla pojedynczego wierzchołka
    GL_FLOAT, // Typ danych
    GL_FALSE, // Czy normalizować przekazane dane
    sizeof(glm::vec2), // Wielkość struktury danych dla wierzchołka
    (void*) 0 // Offset dla wskazywanego atrybutu w obrębie wierzchołka
 )

glEnableVertexAttribArray(0); // Aktywowane atrybutu
```

- W Opengl zawiera się także **Element Buffer Object** (**EBO**), który pozwala wskazanie kolejności rysowania poszczególnych wierzchołków.

Wiele atrybutów w VAO

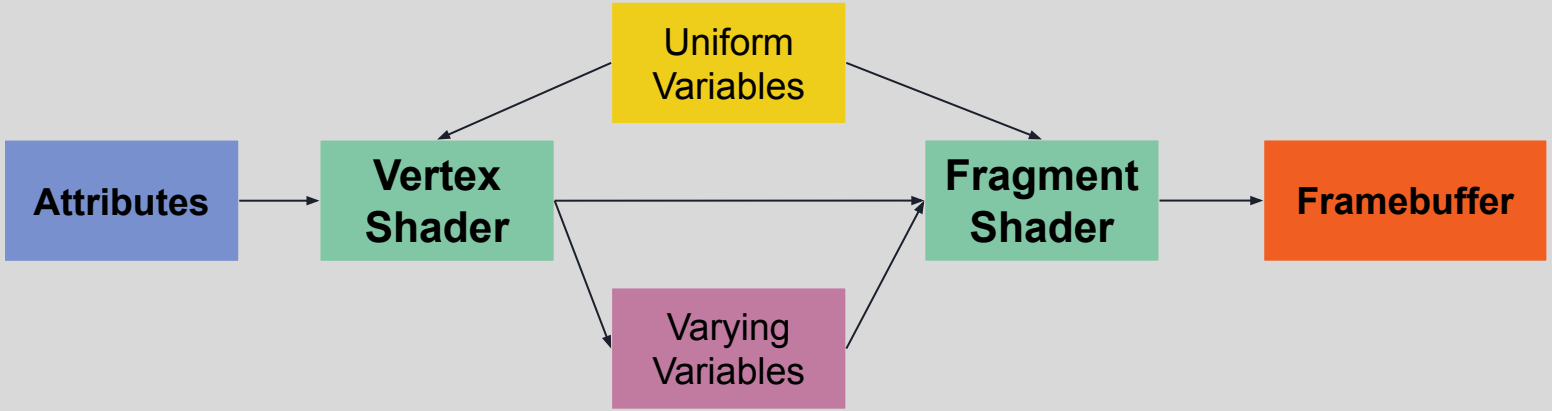

```
GLuint VBO;
GLuint VAO;

std::vector<glm::vec2> positions;
positions.push_back(glm::vec2(-0.5f, -0.5f));
positions.push_back(glm::vec2(0.5f, -0.5f));
positions.push_back(glm::vec2(0.0f, 0.5f));

std::vector<glm::vec3> colors;
colors.push_back(glm::vec3(1.0f, 0.0f, 0.0f));
colors.push_back(glm::vec3(0.0f, 1.0f, 0.0f));
colors.push_back(glm::vec3(0.0f, 0.0f, 1.0f));

struct Vertex
{
    glm::vec2 position;
    glm::vec3 color;
};

std::vector<Vertex> BufferData;

for (int i = 0; i < positions.size(); i++)
    BufferData.push_back({ positions[i], colors[i] });

glGenBuffers(1, &VBO);

glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(Vertex) * BufferData.size(),
  BufferData.data(), GL_STATIC_DRAW);

glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);

glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*) 0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)
  sizeof(glm::vec2));

glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
```

- Atrybuty wierzchołków mogą zawierać tj:
  - Pozycję
  - Kolor
  - Wektor normalny
  - Współrzędne tekstur
- Limit atrybutów jest określony przez stałą GL_MAX_VERTEX_ATTRIBS
- Dla trzeciego atrybutu należało by przekazać offset:

  (void*) (sizeof(glm::vec2) + sizeof(glm::vec3))

  i tak dalej ...
- Atrybuty można wyłączać za pomocą funkcji glDisableVertexArrayAttrib()

Programy cieniujące

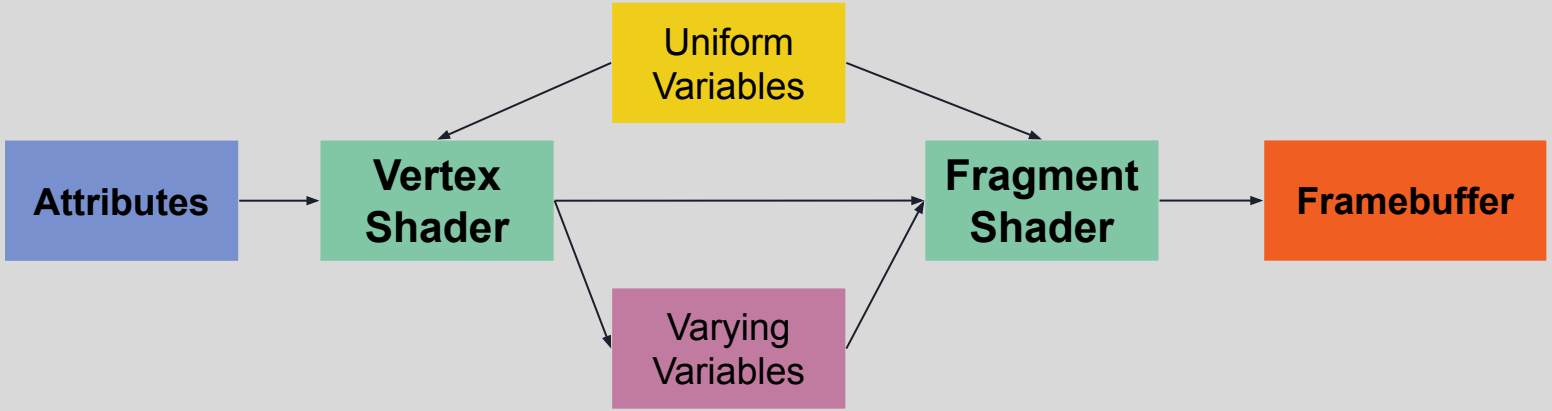


- **Program cieniujący** (ang. shader program) składa się z minimum dwóch shaderów:
  - **Vertex shader** - odpowiedzialny za wyliczenie danych dla danego wierzchołka.
  - **Fragment shader** - odpowiedzialny za wyliczenie danych dla danego fragmentu utożsamianego z pikselem.

- Możliwe jest dodanie opcjonalnych shaderów:
  - **Tessellation Control shader** i **Evaluation shader** - umożliwiają teselację.
  - **Geometry shader** - umożliwia operację na prymitywach.
  - **Compute shader** - wykorzystywany do obliczeń niezwiązanych z renderingiem.

- Shadery implementuje się w języku **GLSL** (OpenGL Shading Language).

Programy cieniujące




```
#version 330 core

layout (location = 0) in vec2 attrPosition;
layout (location = 1) in vec3 attrColor;

uniform float intensity;

out vec3 vColor;

void main()
{
    vColor = attrColor;

    gl_Position = vec4(attrPosition, 0.0, 1.0);
}
```

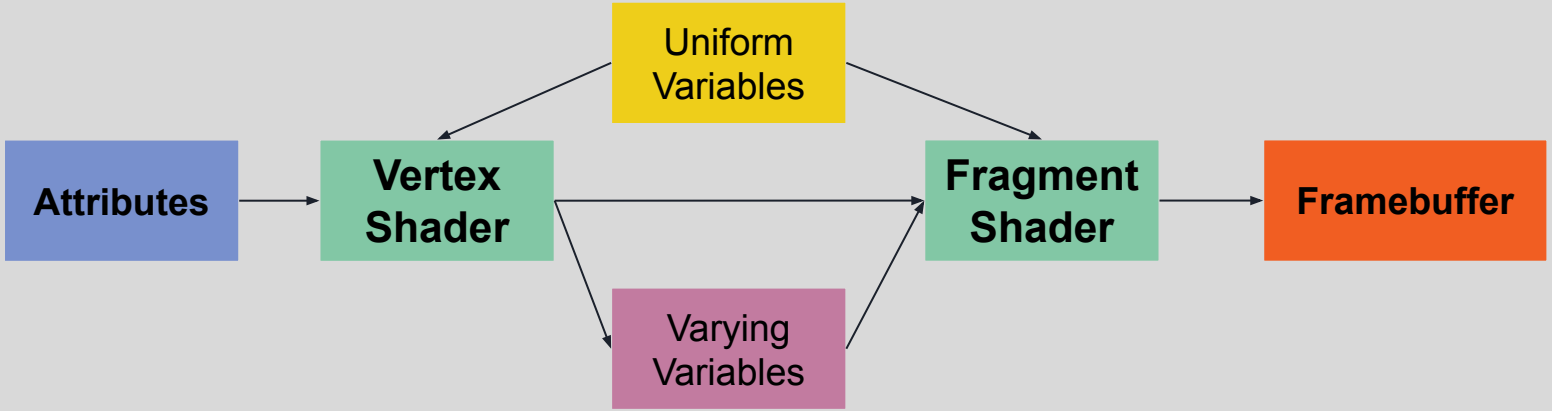
Vertex shader

```
#version 330 core

in vec3 vColor;

out vec4 FragColor;

uniform float intensity;

void main()
{
    FragColor = intensity * vec4(vColor, 1.0f);
}
```

Fragment shader

WYDZIAŁ ELEKTRONIKI I TECHNIK INFORMACYJNYCH

Programy cieniujące



- **Vertex shader ma za zadanie przypisać pozycję wierzchołka do zmienne gl_Position!**
- Zmienne Varying są **interpolowane** pomiędzy Vertex Shader a Fragment Shader.
- W GLSL możliwy jest **Swizzling**, to znaczy wolny dostęp do pól wektorów. Możliwe są takie operacje:
  - vec3 v1 = vec3(1.0, 2.0, 3.0);
  - vec2 v2 = vec3.yx; // (2.0, 1.0)
  - vec4 v3;
  - v3.wzxy = vec4(1.0, 2.0, 3.0, 4.0) // (3.0, 4.0, 2.0, 1.0)
- **Fragment shader może mieć więcej niż jedno wyjście**, stosowane do zaawansowanych technik.

Rysowanie trójkąta


```
GLuint VertexShader;
GLuint FragmentShader;

const char* vertexShaderSourceCStr =
  readShaderFile(VertexShaderFilePath).c_str();

const char* fragmentShaderSourceCStr =
  readShaderFile(FragmentShaderFilePath).c_str();

// Tworzenie i kompilowanie shader'ów
VertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(VertexShader, 1, &vertexShaderSourceCStr, NULL);
glCompileShader(VertexShader);

FragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(FragmentShader, 1, &fragmentShaderSourceCStr, NULL);
glCompileShader(FragmentShader);


// Linkowanie shader'ów
GLuint program;
program = glCreateProgram();

glAttachShader(program, VertexShader);
glAttachShader(program, FragmentShader);

glLinkProgram(program);
```

- Sprawdzenie stanu kompilacji shadera jest możliwe za pomocą funkcji:

glGetShaderiv(shader, GL_COMPILE_STATUS, &s);

- Sprawdzenie stanu linkowania programu jest możliwe za pomocą funkcji:

glGetProgramiv(program, GL_LINK_STATUS, &s);

- Log zawierający stan kompilacji, bądź linkowania można odczytać za pomocą funkcji:

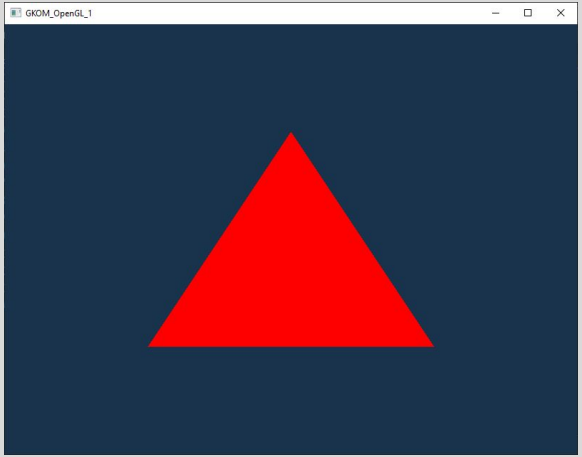
glGetShaderInfoLog()

glGetProgramInfoLog()

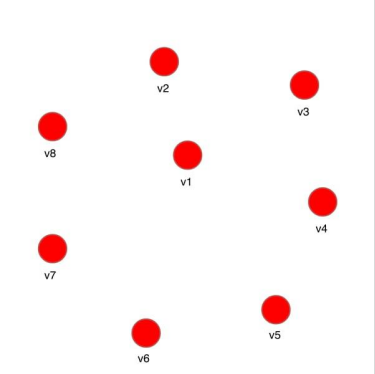
Rysowanie trójkąta


```
while (!glfwWindowShouldClose(window))
{
    glClearColor(0.1f, 0.2f, 0.3f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    glUseProgram(program);
    glBindVertexArray(VAO);
    glDrawArrays(GL_TRIANGLES, 0, 3);

    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

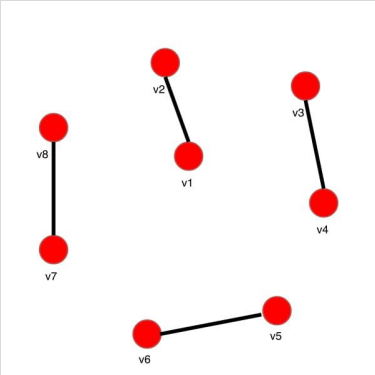
- glDrawArrays() przyjmuje:
  - **Prymityw**
  - Numer pierwszego wierzchołka
  - Ilość wierzchołków
- Wywołanie glUseProgram(0) wyłącza program cieniujący.



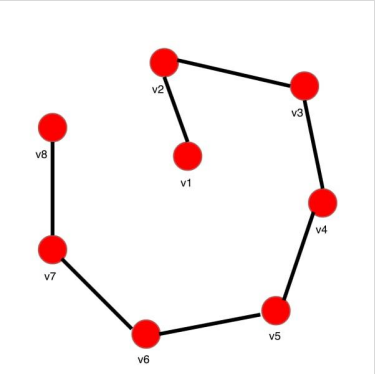
Prymitywy



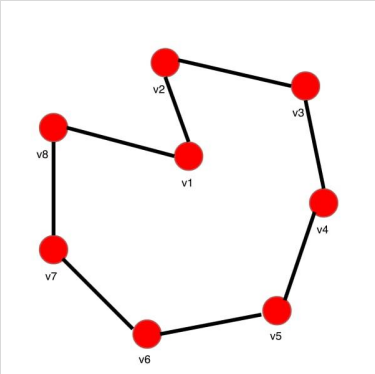
GL_POINTS



GL_LINES

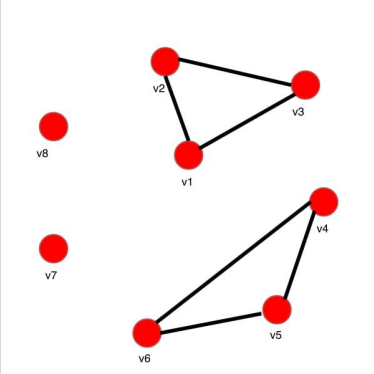


GL_LINE_STRIP

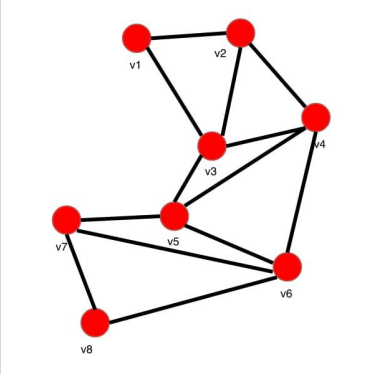


GL_LINE_LOOP

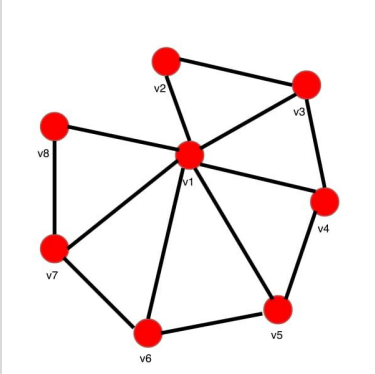
Prymitywy



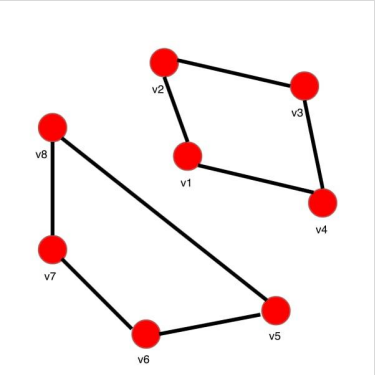
GL_TRIANGLES



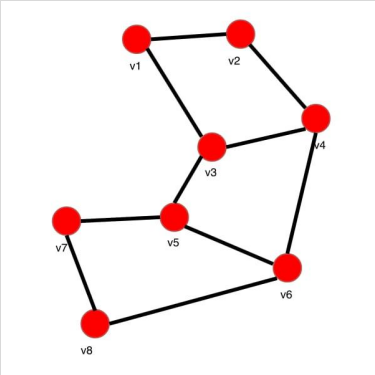
GL_TRIANGLE_STRIP



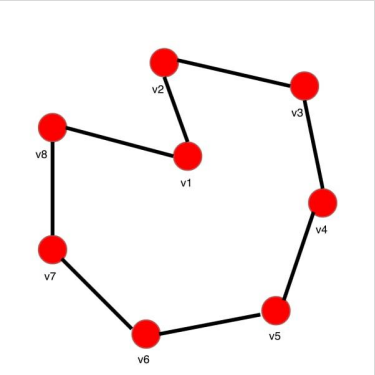
GL_TRIANGLE_LOOP



GL_QUADS



GL_TQUAD_STRIP



GL_POLYGON

Atrybuty i zmienne jednolite

- **Uniform variables** są **stałe dla wszystkich wierzchołka** w obrębie wywołania glDrawArrays().
- Aby uzyskać dostęp do zmiennej jednolitej należy **pobrać jej lokację**:

uniformVarLoc = glGetUniformLocation(program, name)

- **Dobrym zwyczajem jest pobranie lokacji zmiennej Uniform tylko raz!**
- Ustawienie wartości zmiennej jednolitej odbywa się przy wykorzystaniu **rodziny funkcji glUniform()**.

glUniform3f(uniformVarLoc , 1.0f, 2.0f, 3.0f)

- Rodzina funkcji glUniform():
  - glUniform*f(), glUniform*i(), glUniform*ui()
  - glUniform*fv(), glUniform*iv(), glUniform*uiv()
  - glUniformMatrix*fv(), glUniformMatrix*x*fv()
    - * oznacza wartość w przedziale <1, 4>, z pewnymi wyjątkami dla glUniformMatrix*x*fv

WYDZIAŁ ELEKTRONIKI I TECHNIK INFORMACYJNYCH

Transformacja 3D

- Celem zaprezentowania obiektu w środowisku 3D, w Vertex Shader należy **przemnożyć pozycję wierzchołka przez trzy macierze**:
  - **Projection Matrix** - macierz projekcji zawierająca parametry kamery (FoV, Far and Near Clipping)

    glm::perspective(glm::radians(FoV), windowResolution.x / windowResolution.y, nearZ, farZ)
  - **Camera Matrix** - macierz kamery zawierająca pozycjonowanie kamery (pozycja, kierunek, wektor góry)

    glm::lookAt(cameraPosition, cameraTarget, cameraUp)
  - **Model Matrix** - macierz obiektu zawierająca przekształcenia wskazanego obiektu (translacja, rotacja, skalowanie). Wyliczona jest poprzez mnożenie macierzy poszczególnych transformacji.
  - glm::rotate (m, angle, v)
  - glm::scale (m, v)
  - glm::translate (m, v)

WYDZIAŁ ELEKTRONIKI I TECHNIK INFORMACYJNYCH