

Informatyka praktyczna. Szablon Aplikacji w OpenGL

Aleksander Denisiuk
Uniwersytet Warmińsko-Mazurski
Olsztyn, ul. Słoneczna 54
aleksander.denisiuk@uwm.edu.pl

Szablon Aplikacji w OpenGL

Najnowsza wersja tego dokumentu dostępna jest pod adresem

<http://wmii.uwm.edu.pl/~denisjuk/uwm>

Czynności

Czynności standardowe

- Utworzenie kontekstu — biblioteka `GLFW` (jedna z możliwości)
- Dostęp do *rozszerzeń OpenGL* — biblioteka `GLEW` (jedna z możliwości)
- Utworzenie VAO
- Kompilacja programu cieniującego z shaderów
- Rejestracja *callbacków* (funkcji wywołania zwrotnego) na zdarzenia okna (zmiana rozmiaru, renderowanie, etc), klawiatury, myszki, etc — biblioteka `GLFW` (jedna z możliwości)
- Renderowanie
- Czyszczenie pamięci GPU (z VAO, programów, etc)
- Debugowanie

Dołączenie bibliotek graficznych

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

- Kolejność ma znaczenie
- Nie ma potrzeby w sposób jawny dołączać

```
#include <GL/gl.h>
```

Utworzenie kontekstu

- Inicjalizacja GLFW

```
glfwInit();
```

- Ustawienie wersji OpenGL i innych parametrów

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR,  
               major_gl_version);
```

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR,  
               minor_gl_version);
```

```
glfwWindowHint(GLFW_OPENGL_DEBUG_CONTEXT, GLFW_TRUE);
```

Ustawienia i utworzenie okna

- Utworzenie okna i odpowiedniego kontekstu

```
window = glfwCreateWindow(width, height, title,  
                           monitor, share_window);
```

```
if (!window) {  
    std::cerr << "ERROR";  
    glfwTerminate();  
    exit(EXIT_FAILURE);  
}
```

```
glfwMakeContextCurrent(window);
```

- ◆ w razie powodzenia `window` będzie dodatnią liczbą
- ◆ w trybie okienkowym `monitor == NULL`
- ◆ jeżeli kontekst nie jest współdzielony, `share_window == NULL`

Dostęp do rozszerzeń

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

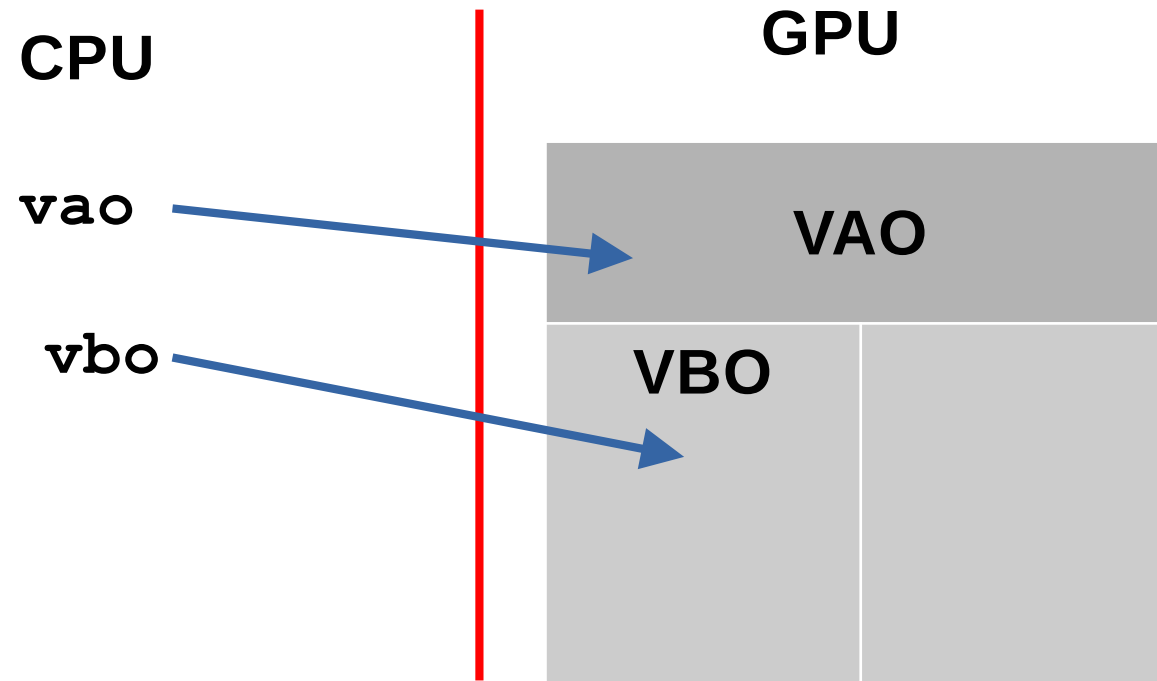
- Zmienna `glew_init_result` jest zdefiniowana w module `GLEW`
- W razie niepowodzenia zmienna `glew_init_result` będzie zawierała kod błędu (wartość inną, niż `GLEW_OK`)
- Komunikat o błędzie można otrzymać poprzez funkcję `glewGetErrorString(glew_init_result)`
- Czasami po `glewInit()` generuje się błąd OpenGL `GL_INVALID_ENUM` — zignorować
- `GLEW` a `Wayland`: generuje się błąd `Unknown error` o kodzie 4 (`GLEW_ERROR_NO_GLX_DISPLAY`) — zignorować

Utworzenie VAO

- Zmienna `vao` typu `GLuint` jest *identyfikatorem* VAO

```
glGenVertexArrays(1, &vao);  
glBindVertexArray(vao);
```

- Dowiązanie VAO robi go aktywnym. Wszystkie kolejne działania będą dotyczyć bieżącego VAO
- W szczególności, można utworzyć VBO



Utworzenie VBO

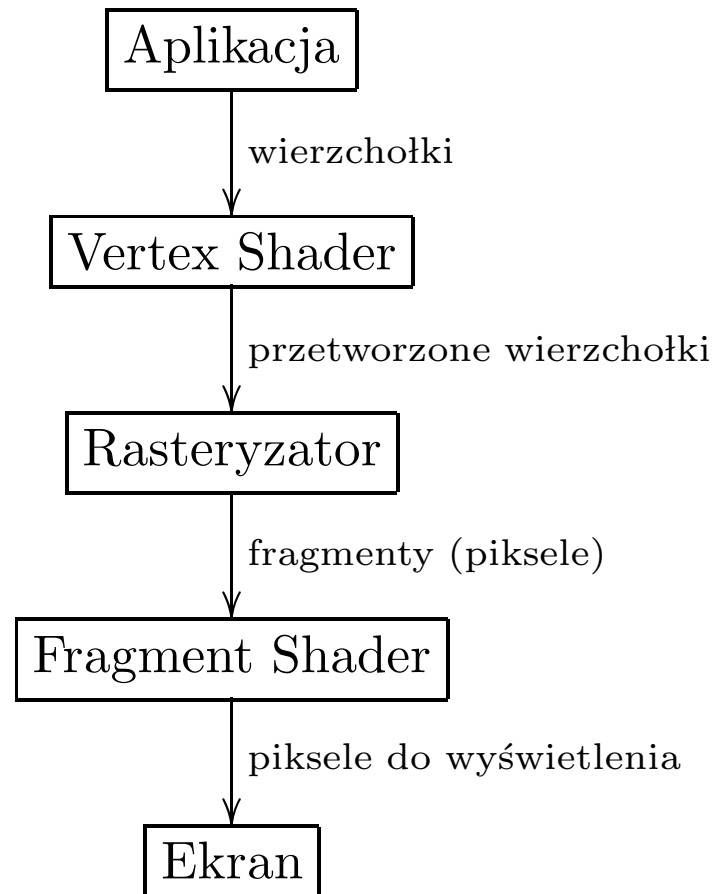
- Zmienna `vbo` typu `GLuint` jest *identyfikatorem* VBO

```
glGenBuffers(1, &vbo);  
glBindBuffer(GL_ARRAY_BUFFER, vbo)
```

- Dowiązanie VBO robi go aktywnym. Wszystkie kolejne działania będą dotyczyć bieżącego VBO
- W szczególności, można wysłać dane do VBO

```
glBufferData(GL_ARRAY_BUFFER, size, dane,  
             GL_STATIC_DRAW);
```

Uproszczony potok renderingu



Wykorzystanie VBO w shaderze wierzchołków

- Definicja parametru wejściowego dla shadera:

```
glVertexAttribPointer(index, size, type,  
                      normalized, stride, pointer);  
glEnableVertexAttribArray(index);
```

- Na przykład:

```
glVertexAttribPointer(0, 4, GL_FLOAT,  
                     GL_FALSE, 0, 0);  
glEnableVertexAttribArray(0);
```

- W shaderze wierzchołków:

```
layout (location=index) in type variable;
```

- Na przykład:

```
layout (location=0) in vec4 in_position;
```

Kompilacja programu

- Kompilacja shadera:

```
GLuint shader = glCreateShader(type);  
glShaderSource(shader, 1, &source, NULL);  
glCompileShader(shader);
```

- ◆ `type` to `GL_VERTEX_SHADER` bądź `GL_FRAGMENT_SHADER`
- ◆ `source` jest tekstem kodu shadera w pamięci CPU

- Linkowanie programu:

```
GLuint program = glCreateProgram();  
glAttachShader(program, vertex_shader);  
glAttachShader(program, fragment_shader);  
glLinkProgram(program);
```

Rejestracja callbacków

- Za pomocą GLFW:
 - ◆ `glfwSetWindowSizeCallback` zmiana rozmiarów okna
 - ◆ `glfwSetKeyCallback` zdarzenia klawiatury
 - ◆ `glfwSetMouseButtonCallback` zdarzenia myszki
 - ◆ etc

Pętla główna

```
while (!glfwWindowShouldClose(window)) {  
    // polecenia renderowania OpenGL  
    glfwSwapBuffers(window);  
    glfwPollEvents();  
}
```

- `glfwPollEvents()` — opracować wszystkie zdarzenia (o ile są) i ponownie renderować okno
- `glfwWaitEvents()` — czekać na zdarzenia i dopiero wówczas renderować okno
- `glfwWaitEventsTimeout(sec)` — czekać na zdarzenia albo `sec` sekund i dopiero wówczas renderować okno

Renderowanie OpenGL

```
glUseProgram(program);  
glBindVertexArray(vao);
```

```
glDrawArrays(mode, first, count);
```

```
glBindVertexArray(0);  
glUseProgram(0);
```

- **na przykład:**

```
glUseProgram(program);  
glBindVertexArray(vao);
```

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

```
glBindVertexArray(0);  
glUseProgram(0);
```

Tryby renderowania

GL_POINTS, GL_LINES, GL_LINE_STRIP,
GL_LINE_LOOP, GL_TRIANGLE_STRIP,
GL_TRIANGLE_FAN

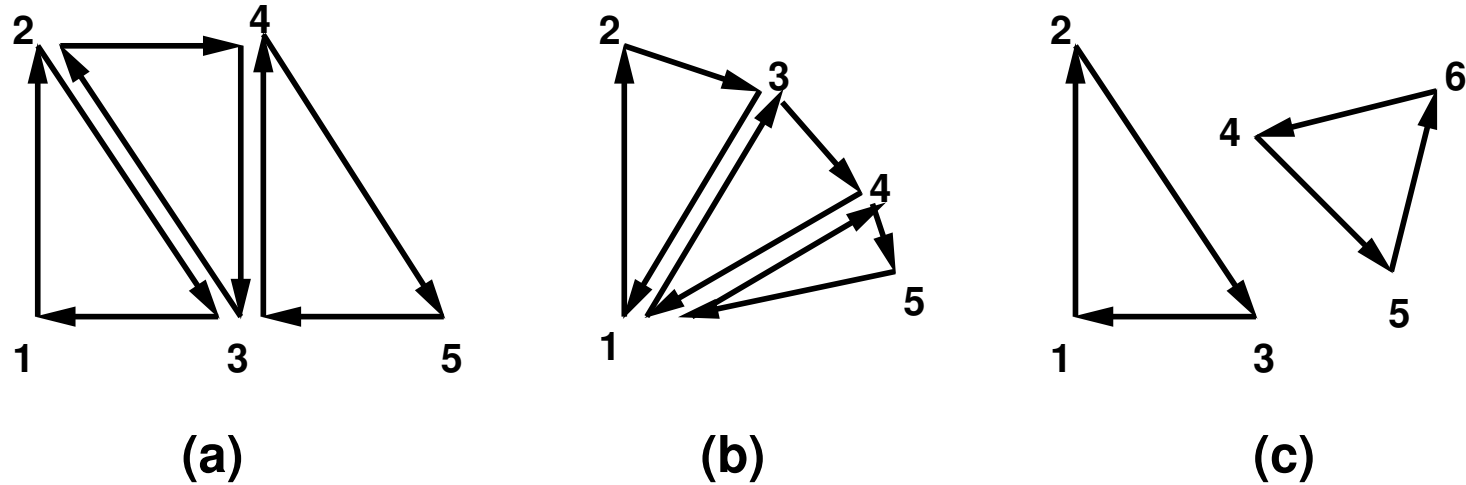


Figure 2.3. (a) A triangle strip. (b) A triangle fan. (c) Independent triangles. The numbers give the sequencing of the vertices in order within the vertex arrays. Note that in (a) and (b) triangle edge ordering is determined by the first triangle, while in (c) the order of each triangle's edges is independent of the other triangles.

Usuwanie programu i shaderów

```
glUseProgram(0);
```

```
glDetachShader(program, vertex_shader);  
glDetachShader(program, fragment_shader);
```

```
glDeleteShader(fragment_shader);  
glDeleteShader(vertex_shader);
```

```
glDeleteProgram(program);
```

Uzuwanie VBO i VAO

```
glDisableVertexAttribArray(1);  
glDisableVertexAttribArray(0);
```

```
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

```
glDeleteBuffers(1, &color_buffer);  
glDeleteBuffers(1, &vertex_buffer);
```

```
glBindVertexArray(0);  
glDeleteVertexArrays(1, &vao);
```

Błędy kompilacji shadera

- Zmienna `compiled` ma typ `GLint`, zawiera kod błędu (lub zero):

```
glGetShaderiv(shader, GL_COMPILE_STATUS,  
               &compiled);
```

- Zmienna `log_size` ma typ `GLint`, zawiera rozmiar komunikatu o błędzie (łącznie z `'\0'`):

```
glGetShaderiv(shader, GL_INFO_LOG_LENGTH,  
               &logSize);
```

- Zmienna `log_msg` ma typ `char[log_size]`, zawiera komunikat o błędzie:

```
glGetShaderInfoLog(shader, log_size, NULL,  
                   log_msg);
```

Błędy linkowania programu

- Zmienna `linked` ma typ `GLint`, zawiera kod błędu (lub zero):

```
glGetProgramiv(program, GL_LINK_STATUS,  
                &linked);
```

- Zmienna `log_size` ma typ `GLint`, zawiera rozmiar komunikatu o błędzie (łącznie z `'\0'`):

```
glGetProgramiv(program, GL_INFO_LOG_LENGTH,  
                &log_size);
```

- Zmienna `log_msg` ma typ `char[log_size]`, zawiera komunikat o błędzie:

```
glGetProgramInfoLog(program, log_size, NULL,  
                    log_msg);
```

Błędy OpenGL

- Funkcja `glGetError` zwraca kod błędu, na przykład:

```
GLenum error_code = glGetError();
```

- Kody błędów:

```
GL_NO_ERROR, GL_INVALID_OPERATION,  
GL_INVALID_ENUM, GL_INVALID_VALUE,  
GL_INVALID_FRAMEBUFFER_OPERATION,  
GL_OUT_OF_MEMORY.
```

- Można wywołać po każdym działaniu graficznym

Wykorzystanie callbacka

- W OpenGL, od wersji 4.3 można zarejestrować callback na zdarzenie `error`
 - ◆ inicjalizować kontekst z flagą `GLFW_OPENGL_DEBUG_CONTEXT`
 - ◆ ustawić callback za pomocą funkcji

```
void glDebugMessageCallback (
    DEBUGPROC callback, void * user_param);
```
- callback powinien być funkcją o prototypie

```
typedef void (APIENTRY *DEBUGPROC) (
    GLenum source, GLenum type,
    GLuint id, GLenum severity,
    GLsizei length,
    const GLchar *message,
    void *user_param);
```

 - ◆ możliwe, że zamiast `APIENTRY` powinno być `GLAPIENTRY`

Przykładowy callback

```
void GLAPIENTRY OpenglCallbackFunction(
    GLenum /*source*/, GLenum type, GLuint id,
    GLenum /*severity*/, GLsizei /*length*/,
    const GLchar* message, void* /*user_param*/) {
    cout << "message: " << message << endl;
    cout << "type: ";
    switch (type) {
        case GL_DEBUG_TYPE_ERROR:
            cout << "ERROR";
            break;
        case GL_DEBUG_TYPE_OTHER:
            cout << "OTHER";
            break;
    }
    cout << endl << "id: " << id << endl;
}
```

Synchroniczne emitowanie komunikatów

```
glEnable (GL_DEBUG_OUTPUT_SYNCHRONOUS) ;
```

Filtracja komunikatów

- Można ograniczyć ilość emitowanych komunikatów:

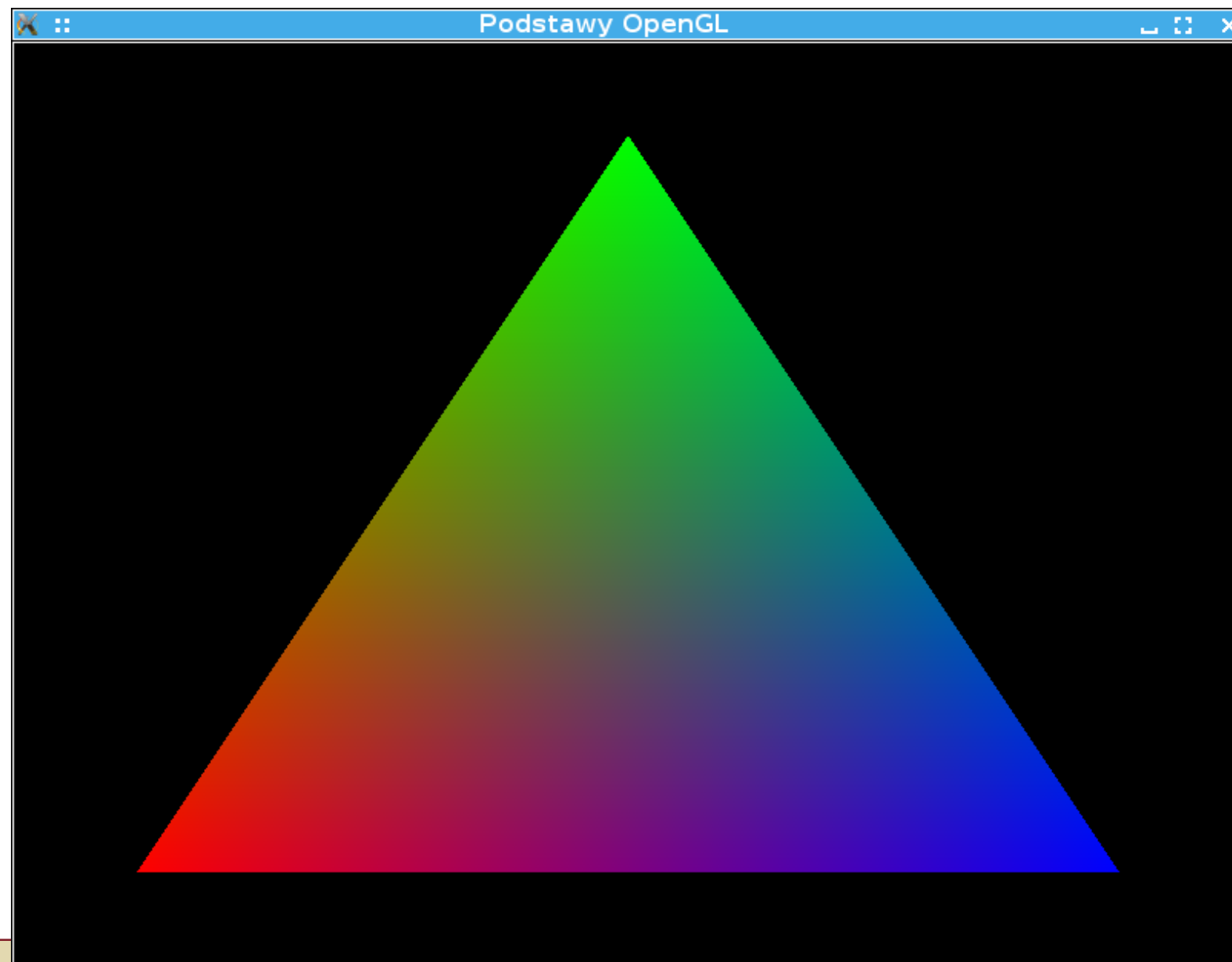
```
void glDebugMessageControl(GLenum source,  
                           GLenum type,  
                           GLenum severity,  
                           GLsizei count,  
                           const GLuint *ids,  
                           GLboolean enabled);
```

Przykład

```
if (glDebugMessageCallback) {  
    glEnable (GL_DEBUG_OUTPUT_SYNCHRONOUS);  
    glDebugMessageCallback (OpenglCallback, NULL);  
    GLuint unused_ids = 0;  
    glDebugMessageControl (GL_DONT_CARE,  
        GL_DONT_CARE,  
        GL_DONT_CARE,  
        0,  
        &unused_ids,  
        GL_FALSE);  
}  
else  
    std::cout << "glDebugMessageCallback\  
        not available" << std::endl;
```

Hello, Trianlge!

Render trójkąta



Shader Wierzchołków

```
#version 430 core
```

```
layout(location=0) in vec4 in_position;  
layout(location=1) in vec4 in_color;  
out vec4 frag_color;
```

```
void main(void) {  
    gl_Position = in_position;  
    frag_color = in_color;  
}
```

Shader Fragmentów

```
#version 430 core
```

```
layout (location = 0) out vec4 color;  
in vec4 frag_color;
```

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

Klasy C++

- `Window` — odpowiada za kontekst
- `Triangle` — model: VAO, renderowanie
- `Program` — program: kompilacja shaderów
- `glerror.h` — definicje, związane z debugowaniem
- Kod przeważnie jest zgodny z [Google C++ Style Guide](#)

Założenie

- Obiekty geometryczne oraz programy definiujemy jako zmienne (*nie dynamiczne* klasy `Window`)
- Kod, odpowiadający za zwolnienie zasobów karty graficznej umieszczamy w destruktorach klas-programów i klas-obiektów geometrycznych
 - ✦ w taki sposób, zostaną one odpalone po zakończeniu pacy, w wyniku odpalania standardowego destruktora obiektu klasy `Window`
- Inicjalizację obiektów geometrycznych i programów umieszczamy w funkcjach `Initialize` odpowiednich klas

Globalny obiekt window i callbacki

```
const int kMajorGLVersion = 4;
const int kMinorGLVersion = 3;

static Window window("Postawy OpenGL", 800, 600);

void Resize (GLFWwindow* /*window*/,
             int new_width, int new_height) {
    window.Resize(new_width, new_height);
}

void KeyEvent (GLFWwindow* /*window*/, int key,
              int scancode, int action, int mods) {
    window.KeyEvent(key, scancode, action, mods);
}
```

main

```
int main(void) {  
    window.Initialize(kMajorGLVersion,  
                      kMinorGLVersion);  
    glfwSetWindowSizeCallback(window, Resize);  
    glfwSetKeyCallback(window, KeyEvent);  
  
    window.Run();  
    glfwTerminate();  
    exit(EXIT_SUCCESS);  
}
```

Klasa Window, dane publiczne

```
class Window{
public:
    Window(const char*, int, int);
    void Initialize(int major_gl_version,
                  int minor_gl_version);
    void Resize(int new_width, int new_height);
    void KeyEvent(int key, int scancode,
                 int action, int mods);
    void Run(void);
    operator GLFWwindow*(){return window_;}
private:
    GLFWwindow* window_;
```

Klasa Window. Dane prywatne

```
private:
    int width_;
    int height_;
    const char* title_;
    GLFWwindow* window_;

    Triangle triangle_;

    Program program_;

    void InitGlfwOrDie (int major_gl_version,
                      int minor_gl_version);

    void InitGlewOrDie ();
    void InitModels ();
    void InitPrograms ();
};
```

Konstruktor

```
Window::Window(const char * title,  
               int width, int height) {  
    title_ = title;  
    width_ = width;  
    height_ = height;  
}
```

Inicjalizacja

```
void Window::Initialize(  
    int major_gl_version, int minor_gl_version) {  
  
    InitGlfwOrDie(major_gl_version,  
                  minor_gl_version);  
    InitGlewOrDie();  
    std::cout  
        << "OpenGL initialized: OpenGL version: "  
        << glGetString(GL_VERSION)  
        << " GLSL version: "  
        << glGetString(GL_SHADING_LANGUAGE_VERSION)  
        << std::endl;  
    InitModels();  
    InitPrograms();  
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);  
}
```

Inicjalizacja kontekstu

```
void Window::InitGlfwOrDie(int major_gl_version,
                           int minor_gl_version) {
    if ( !glfwInit() ) {
        std::cerr << "ERROR..." << std::endl;
        exit(EXIT_FAILURE);
    }
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR,
                   major_gl_version);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR,
                   minor_gl_version);
#ifdef DEBUG
    glfwWindowHint(GLFW_OPENGL_DEBUG_CONTEXT,
                   GLFW_TRUE);
#endif
    .....
}
```

Inicjalizacja kontekstu. Okno

```
.....  
window_ = glfwCreateWindow(width_,  
                             height_, title_, nullptr, nullptr);  
if (!window_) {  
    std::cerr << "ERROR..." << std::endl;  
    glfwTerminate();  
    exit(EXIT_FAILURE);  
}  
glfwMakeContextCurrent(window_);  
}
```

Dostęp do rozszerzeń

```
void Window::InitGlewOrDie() {
    GLenum glew_init_result;
    glewExperimental = GL_TRUE;
    glew_init_result = glewInit();

    if (GLEW_OK != glew_init_result) {
        if (GLEW_ERROR_NO_GLX_DISPLAY != glew_init_result
            || GLFW_PLATFORM_WAYLAND != glfwGetPlatform()) {
            std::cerr << "Glew ERROR: "
                << glewGetErrorString(glew_init_result)
                << std::endl;
            glfwTerminate();
            exit(EXIT_FAILURE);
        }
    }
}
```

Profil *DEBUG* — rejestracja callbacka na error

```
#ifdef DEBUG
    if (glDebugMessageCallback) {
        std::cout << "Register debug callback";
        glEnable(GL_DEBUG_OUTPUT_SYNCHRONOUS);
        glDebugMessageCallback((GLDEBUGPROC)
                                OpenglCallbackFunction, NULL);
        GLuint unused_ids = 0;
        glDebugMessageControl(GL_DONT_CARE,
                              GL_DONT_CARE,
                              GL_DONT_CARE,
                              0,
                              &unused_ids,
                              GL_FALSE);
    }
    else
        std::cout << "glDebugMessageCallback not available";
#endif
```

Inicjalizacja modelu i programu

```
void Window::InitModels () {  
    triangle_.Initialize();  
}
```

```
void Window::InitPrograms () {  
    program_.Initialize();  
}
```

Resize

```
void Window::Resize(int new_width,  
                    int new_height) {  
    width_ = new_width;  
    height_ = new_height;  
    glViewport(0, 0, width_, height_);  
}
```

KeyEvent

```
void Window::KeyEvent(int key,
    int /*scancode*/, int action, int /*mods*/) {
    if(action == GLFW_RELEASE) {
        switch (key) {
            case GLFW_KEY_ESCAPE:
                glfwSetWindowShouldClose(window_,
                                           GLFW_TRUE);

                break;
            default:
                break;
        }
    }
}
```

Renderowanie (Run)

```
void Window::Run(void) {  
    while (!glfwWindowShouldClose(window_)) {  
        glClear(GL_COLOR_BUFFER_BIT  
            /*/ GL_DEPTH_BUFFER_BIT*/);  
        triangle_.Draw(program_);  
        glfwSwapBuffers(window_);  
        glfwWaitEvents();  
    }  
}
```

Klasa Program

```
class Program{
public:
    void Initialize();
    // to be used in glUseProgram()
    operator GLuint() const {return program_;}
    ~Program();
private:
    GLuint program_;
    GLuint vertex_shader_;
    GLuint fragment_shader_;
    GLuint CompileShaderOrDie(const char* source,
                             GLenum type);
    GLuint LinkProgramOrDie(GLint vertex_shader,
                           GLint fragment_shader);
};
```

Inicjalizacja. Kod shadera wierzchołków

```
void Program::Initialize() {
    const GLchar* kVertexShaderText =
    {
        "#version 430 core\n" \

        "layout(location=0) in vec4 in_position;\n" \
        "layout(location=1) in vec4 in_color;\n" \
        "out vec4 frag_color;\n" \

        "void main(void) \n" \
        "{\n" \
        "    gl_Position = in_position;\n" \
        "    frag_color = in_color;\n" \
        "}\n"
    };
};
```

Inicjalizacja. Kod shadera fragmentów

```
const GLchar* kFragmentShaderText =  
{  
    "#version 430 core\n"  
  
    "layout (location = 0) out vec4 color;\n"  
    "in vec4 frag_color;\n"  
  
    "void main(void) \n"  
    "{ \n"  
        color = frag_color;\n"  
    } \n"  
};
```

Inicjalizacja. Kompilacja i linkowanie

```
vertex_shader_ = CompileShaderOrDie(  
    kVertexShaderText,  
    GL_VERTEX_SHADER);  
fragment_shader_ = CompileShaderOrDie(  
    kFragmentShaderText,  
    GL_FRAGMENT_SHADER);  
program_ = LinkProgramOrDie(vertex_shader_,  
    fragment_shader_);  
  
glUseProgram(program_);  
//      some actions on the created program  
//      will be placed here  
glUseProgram(0);  
}
```

Kompilacja shadera

```
GLuint Program::CompileShaderOrDie  
    (const char * source, GLenum type){  
    GLuint shader = glCreateShader(type);  
    glShaderSource(shader, 1, &source, NULL);  
    glCompileShader(shader);
```

.....

Kompilacja shadera. Weryfikacja wyniku

```
GLint  compiled;
glGetShaderiv(shader, GL_COMPILE_STATUS,
               &compiled);

if (!compiled) {
    switch(type) {
        case GL_VERTEX_SHADER:
            std::cerr << "vertex ";
            break;
        case GL_FRAGMENT_SHADER:
            std::cerr << "fragment ";
            break;
    }
    std::cerr << "shader is failed to compile:";
```

Kompilacja shadera. Weryfikacja wyniku,cd

```
.....  
    GLint  log_size;  
    glGetShaderiv(shader, GL_INFO_LOG_LENGTH,  
                                                           &log_size);  
  
    char* log_msg = new char[log_size];  
    glGetShaderInfoLog(shader, log_size, NULL,  
                                                                log_msg);  
  
    std::cerr << log_msg << std::endl;  
    delete [] log_msg;  
    glfwTerminate();  
    exit( EXIT_FAILURE );  
}  
return shader;  
}
```

Linkowanie programu.

```
GLuint Program::LinkProgramOrDie(  
    GLint vertex_shader, GLint fragment_shader) {  
    GLuint new_program = glCreateProgram();  
    glAttachShader(new_program, vertex_shader);  
    glAttachShader(new_program, fragment_shader);  
    glLinkProgram(new_program);  
}
```

.....

Linkowanie programu. Weryfikacja wyniku

```
GLint  linked;
glGetProgramiv(new_program, GL_LINK_STATUS,
               &linked);

if ( !linked ) {
    std::cerr << "Shader program failed to link";
    GLint  log_size;
    glGetProgramiv(new_program,
                  GL_INFO_LOG_LENGTH, &log_size);
    char* log_msg = new char[log_size];
    glGetProgramInfoLog(new_program, log_size,
                       NULL, log_msg);
    std::cerr << log_msg << std::endl;
    delete [] log_msg;
    glfwTerminate();
    exit( EXIT_FAILURE );
}
return new_program;
}
```

Usuwanie programu z pamięci GPU

```
Program::~Program() {  
    glUseProgram(0);  
  
    glDetachShader(program_, vertex_shader_);  
    glDetachShader(program_, fragment_shader_);  
  
    glDeleteShader(fragment_shader_);  
    glDeleteShader(vertex_shader_);  
  
    glDeleteProgram(program_);  
}
```

Klasa Triangle

```
class Triangle{
public:
    void Initialize();
    void ~Triangle();
    void Draw(const Program & program);
private:
    GLuint vao_;
    GLuint vertex_buffer_;
    GLuint color_buffer_;
};
```

Inicjalizacja. Wierzchołki i kolory

```
void Triangle::Initialize() {  
    const GLfloat kVertices[] = {  
        -0.8f, -0.8f, 0.0f, 1.0f,  
         0.0f,  0.8f, 0.0f, 1.0f,  
         0.8f, -0.8f, 0.0f, 1.0f  
    };  
  
    const GLfloat kColors[] = {  
        1.0f, 0.0f, 0.0f, 1.0f,  
        0.0f, 1.0f, 0.0f, 1.0f,  
        0.0f, 0.0f, 1.0f, 1.0f  
    };  
};
```

.....

Inicjalizacja. VAO

```
glGenVertexArrays(1, &vao_);  
glBindVertexArray(vao_);
```

Inicjalizacja. VBO dla wierzchołków

```
glGenBuffers(1, &vertex_buffer_);  
glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer_);  
glBufferData(GL_ARRAY_BUFFER, sizeof(kVertices),  
             kVertices, GL_STATIC_DRAW);  
glVertexAttribPointer(0, 4, GL_FLOAT,  
                     GL_FALSE, 0, 0);  
glEnableVertexAttribArray(0);
```

Inicjalizacja. VBO dla kolorów

```
glGenBuffers(1, &color_buffer_);  
glBindBuffer(GL_ARRAY_BUFFER, color_buffer_);  
glBufferData(GL_ARRAY_BUFFER, sizeof(kColors),  
             kColors, GL_STATIC_DRAW);  
glVertexAttribPointer(1, 4, GL_FLOAT,  
                      GL_FALSE, 0, 0);  
glEnableVertexAttribArray(1);
```

Inicjalizacja. Zakończenie

```
glBindBuffer(GL_ARRAY_BUFFER, 0);  
glBindVertexArray(0);  
}
```

Renderowanie

```
void Triangle::Draw(Program program) {  
    glUseProgram(program);  
    glBindVertexArray(vao_);  
  
    glDrawArrays(GL_TRIANGLES, 0, 3);  
  
    glBindVertexArray(0);  
    glUseProgram(0);  
}
```

Usuwanie z pamięci GPU

```
Triangle::~Triangle() {  
    glDisableVertexAttribArray(1);  
    glDisableVertexAttribArray(0);  
  
    glBindBuffer(GL_ARRAY_BUFFER, 0);  
  
    glDeleteBuffers(1, &color_buffer_);  
    glDeleteBuffers(1, &vertex_buffer_);  
  
    glBindVertexArray(0);  
    glDeleteVertexArrays(1, &vao_);  
}
```

glDebugMessageCallback

```
void GLAPIENTRY OpenglCallbackFunction(
    GLenum /*source*/, GLenum type, GLuint id,
    GLenum severity, GLsizei /*length*/,
    const GLchar* message, void* /*user_param*/) {
    cout << "----opengl-callback-start--" << endl;
    cout << "message: " << message << endl;
    cout << "type: ";
    switch (type) {
        case GL_DEBUG_TYPE_ERROR:
            cout << "ERROR";
            break;
        .....

        case GL_DEBUG_TYPE_OTHER:
            cout << "OTHER";
            break;
    }
    cout << endl;
```

glDebugMessageCallback, cd

```
cout << "id: " << id << endl;
cout << "severity: ";
switch (severity) {
    case GL_DEBUG_SEVERITY_LOW:
        cout << "LOW";
        break;
    case GL_DEBUG_SEVERITY_MEDIUM:
        cout << "MEDIUM";
        break;
    case GL_DEBUG_SEVERITY_HIGH:
        cout << "HIGH";
        break;
    case GL_DEBUG_SEVERITY_NOTIFICATION:
        cout << "NOTIFICATION";
        break;
}
cout << endl;
```

}

Makefile dla Linux

```
CC = g++
LIBS = -lX11 -lglfw -lGL -lGLU -lGLEW -lm
SOURCES = $(wildcard *.cpp)
OBJECTS = $(SOURCES:.cpp=.o)
EXECUTABLE = triangle
CFLAGS=-c -Wall -DDEBUG -g3 -fpermissive -std=c++11 -MMD

all: $(SOURCES) $(EXECUTABLE)
$(EXECUTABLE): $(OBJECTS)
    $(CC) $(OBJECTS) -o $@ $(LIBS)
clean:
    rm -f $(EXECUTABLE) $(OBJECTS)
    rm -f $(SOURCES:%.cpp=%.d)
.cpp.o:
    $(CC) $(CFLAGS) $< -o $@
-include $(SOURCES:%.cpp=%.d)
```

Makefile dla Windows i Linux

```
ifeq ($(OS),Windows_NT)
# -l:pełna forma biblioteki bez autodopasowywania lib*.a
#W związku z czym trzeba to ustawić jawnie
LIBS =-lglfw3 -l:glew32.dll -lopengl32 -lm -lglu32 -lgdi32
EXECUTABLE = triangle.exe
LDFLAGS=-Wl,--subsystem,windows
else
LIBS = -lX11 -lglfw -lGL -lGLU -lGLEW -lm
EXECUTABLE = triangle
endif
```

gdb

- uruchomienie programu w gdb

```
$ gdb ./triangle
```

- ustawienie przerywania

```
(gdb) break OpenGLCallbackFunction
```

- uruchomienie programu

```
(gdb) run
```

- stos wywołań

```
(gdb) backtrace
```

- wyjście z callbacka

```
(gdb) finish
```

- RMS's gdb Debugger Tutorial