

Grafika Komputerowa. Podstawy OpenGL

Aleksander Denisiuk

Polsko-Japońska Akademia Technik Komputerowych

Wydział Informatyki w Gdańsku

ul. Targ Drzewny 9/11

80-894 Gdańsk

denisiuk@pja.edu.pl

Historia

Szablon aplikacji

Hello, triangle!

Najnowsza wersja tego dokumentu dostępna jest pod adresem

<http://users.pja.edu.pl/~denisjuk>

Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

Historia

Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

- 1981: Silicon Graphics
- Koniec lat 1980: IRIS GL
(Integrated Raster Imaging
System Graphical Library) —
proprietarna
- 1992: OpenGL (Open Graphics
Library) — otwarta



SiliconGraphics
Computer Systems



Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

- Brak jekiegokolwiek kodu, sama specyfikacja
 - ☐ każdy producent (sprzętu jak i oprogramowania) może implemetować na swojej platformie
- Rozszerzenia (**NV_**, **AGL_**)
 - ☐ brak standardowego mechanizmu rozszerzeń
 - ☐ szczególny problem na Windows, gdzie nagłówki OpenGL są nadal w wersji 1.1

Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

- 1992: ARB (OpenGL Architecture Review Board)
 - ☐ testy zgodności implementacji
- Wiodący standard grafiki w czasie rzeczywistym
 - ☐ jedyny wieloplatformowy

Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

- 1993: Windows NT: rywal Uniksowi w sieci
 - ☐ brak biblioteki graficznej
 - ☐ decyzja zaimplementować OpenGL



- 1994: Windows NT 3.5, implementacja OpenGL
 - ☐ prowizoryczna
 - ☐ brak optymalizacji i przyśpieszenia sprzętowego
 - ☐ Ostrzeżenie **KB121282**:
“OpenGL Screen Savers May Degrade Server Performance”

Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

- Microsoft: własne 3W API dla gier pod Windows

- ☐ WinG (translacja poleceń do GDI)
- ☐ 1995: Microsoft nabywa RenderMorphics

- Reality Lab



- ☐ 1996: DirectX

- DirectDraw
- DirectInput
- DirectPlay
- DirectSound
- Direct3D

- Direct3D: bardzo niewygonna
- implementacja OpenGL 1.1 w Windows 95 oraz NT 4.0

Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

- Microsoft: OpenGL do grafiki „profesjonalnej” (CAD). Do programowania gier lepiej nadaje się DirectX



- 1996: John Carmack (id Software) — port Quake na OpenGL dla Windows, **porównanie OpenGL a Direct3D**
- 1997: Alex St. John — **odpowiedź w obronie Direct3D**

Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

- Microsoft: OpenGL jest profesjonalną biblioteką i nie będzie wspierana przez “zwykłe” karty graficzne
- SGI: **Odpowiedź: porównanie OpenGL a Direct3D** pod kątem inżynierskim, nie marketologowym
- Direct3D w wersji 5 został porównywalnym z OpenGL

Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

- Sterowniki implementujące OpenGL na Windows NT zostały oparte o MCD (Mini-Client Driver)
- Microsoft nie pozwoliła na licencjonowanie MCD na Windows 95
- SGI: Installable Client Driver (ICD)
- Producenci sprzętu: nowe sterowniki
- Producenci gier: gry pod OpenGL

Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

- Koniec lat 1990: OpenGL standardem 3W grafiki

- ☐ CAD, Quake 2, Unreal, Half-Life

- Pierwsze dedykowane 3W karty graficzne

- ☐ ATI 3D Rage, S3 ViRGE

- ☐ Voodoo Graphics (3Dfx Interactive)

- Glide API

- ☐ NVIDIA

- GeForce 256

- GPU (Graphics Processing Unit)

- T&L (Transform & Lighting)



- Początek 2000: NVIDIA GeForce 2, ATI Radeon 7000

- ☐ OpenGL, Direct3D

Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

- Na początku lat 2000 wydajność GPU wzrasta wykładniczo
- CPU nie jest potrzebny do renderowania 3W grafiki w czasie rzeczywistym
- CPU nawet jest *wązkim gardłem* w procesie renderowania
 - potrzebne są nowe metody, żeby obejść CPU

Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

- *Immediate mode*: program przekazuje ciąg poleceń do GPU
- *Nowa metoda*: obiekt bufora
 - ☐ tworzony i przechowywany w pamięci GPU
 - ☐ OpenGL: *VAO (Vertex Array Object)*, *VBO (Vertex Buffer Object)*
 - ☐ Direct3D: *Vertex Buffer*

Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

■ 2000: Microsoft, Direct3D 8.0

- ☐ vertex shaders
- ☐ pixel shaders
- ☐ Asembler GPU

■ 2003: Direct3D 9.0

- ☐ High-Level Shader Language (HLSL)

Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

- Brak wsparcia shaderów
- Dopiero w 2004: OpenGL 2.0, OpenGL Shading Language (GLSL)
- 2004–2006: dominacja Direct3D 9.0, zaledwie kilka gier na OpenGL
- 2005: Xbox 360 ze wsparciem Direct3D 9.0
- żadnych wiadomości od ARB

Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

- 2006:
 - ☐ OpenGL 2.1
 - ☐ Direct3D 10.0 (razem z Windows Vista)
- Zmiana kierunku rozwoju sprzętu:
 - ☐ eliminacja immediate mode, ustalonych funkcji
 - ☐ co raz bardziej programowalne GPU
- ARB, SGI: brak odpowiedzi

Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

- 2006, SIGGRAPH: zarządzać rozwojem OpenGL będzie Khronos Group
 - ☐ konsorcjum producentów (sprzętu i oprogramowania)
 - ☐ rozwój otwartych standardów
 - OpenGL
 - COLLADA

Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

- Miały zostać opublikowane w 2007: latem oraz w październiku
 - ☐ dociągnąć do Direct3D 10.0
 - wyeliminować immediate mode
 - działać tylko na buforach i shaderach
 - Technical Sub-Groups
 - ☐ Longs Peak
 - kompatybilna z ówczesnym sprzętem
 - kompatybilna wstecz ze starym OpenGL
 - ☐ Mt. Evans
 - złamie kompatybilność wstecz
 - będzie podstawą dla przyszłości
- 30 października 2007: trzeba poczekać

Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

- Lipiec 2008
- Nie było to *Longs Peak*
 - ☐ został immediate mode
 - ☐ nie został włączony w specyfikację nowy model obiektowy
 - ☐ nie było planów, co włączyć w przyszłe wersje
 - ☐ trochę nowych możliwości
 - ☐ deprecation model
 - wszystkie metody, związane z immediate mode zostały zaznaczone jako przestarzałe
 - pełna kompatybilność ze starymi metodami

Historia

Początek OpenGL
OpenGL a Windows
Ewolucja kart
graficznych
VAO
Shadery
Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

- Liczne protesty społeczności
- Programiści Windows zaczęli porzucać OpenGL na rzecz Direct3D
- Czyżby wojna API została przegrana?

Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

- Okazało się, że nie jest tak źle :-)
 - ☐ OpenGL 3.0 miał cechy, których nie było w Direct3D 10.0:
 - dostęp do nowych funkcji w Windows XP
- Marzec 2009: OpenGL 3.1
 - ☐ eliminacja immediate mode
- Czerwiec 2009: OpenGL 3.2
 - ☐ Geometry Shaders
 - ☐ dorównanie z Direct3D 10.0

Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

- *Context* — obiekt pośredniczy pomiędzy programem a urządzeniem OpenGL. Przy tworzeniu kontekstu:
 - ☐ główny numer wersji
 - ☐ drugorzędny numer wersji
 - ☐ opcjonalne parametry
- Każda wersja OpenGL (powyżej 3.0) ma część funkcji *deprecated*, które mogą zostać usunięte w następnych wersjach
 - ☐ Core Profile flag
 - ☐ Compatibility Profile flag
 - ☐ Forward Compatible flag
 - ☐ Debug flag

Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

- 11 marca 2010
 - ☐ OpenGL 4.0
 - współczone GPU
 - Direct3D 11
 - Teselacja
 - 64-bitowa dokładność obliczeń
 - ☐ OpenGL 3.3
 - kompatybilna ze starym sprzętem
 - jak najwięcej możliwości openGL 4.0
- W przykładach będzie używany openGL nowszy niż 3.3, *Core profile*
 - ☐ w szczególności, żadnych `glBegin`, `glEnd`, `glVertex3f`, `glColor3f`
 - ☐ wszystkie przykłady pójdą na 3.3

Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

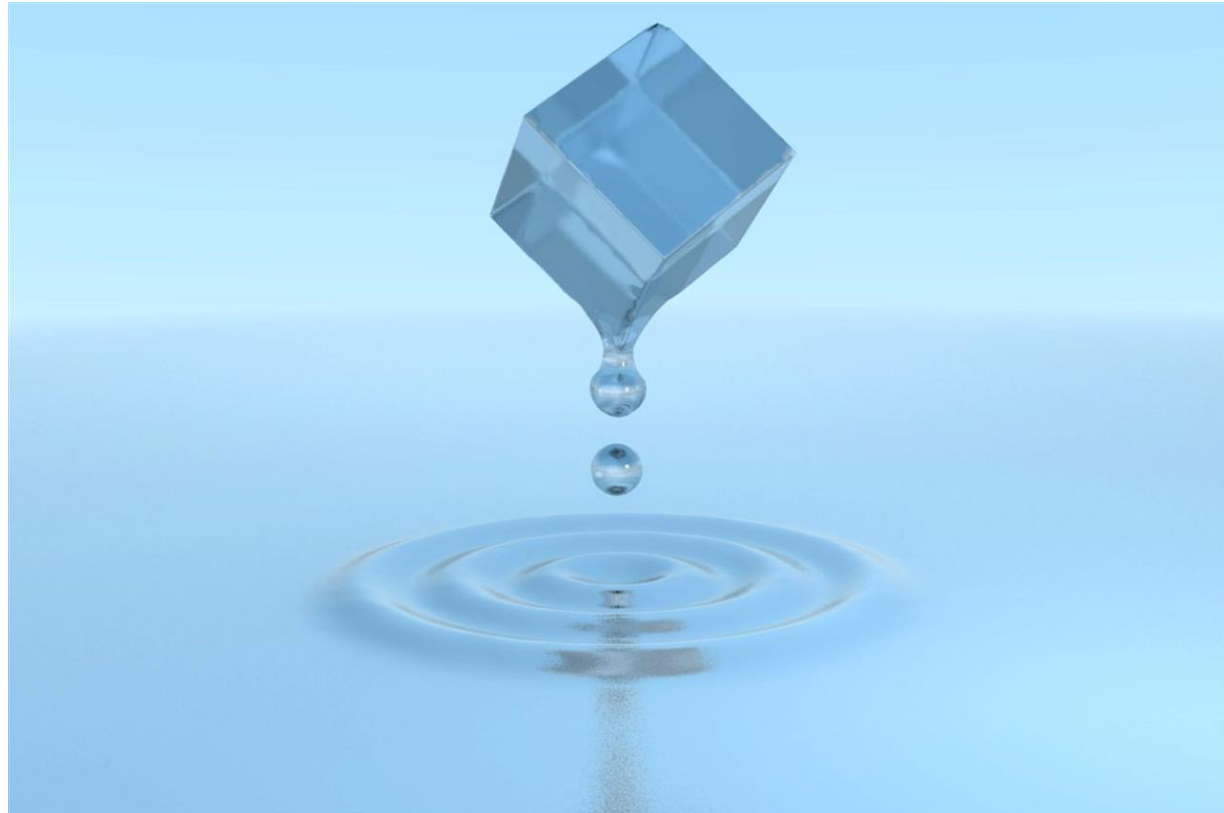
- 26 czerwca 2010 — OpenGL 4.1
 - ☐ nowa wersja GLSL
 - ☐ zupełna zgodność z OpenGL ES
- 8 sierpnia 2011 — OpenGL 4.2
- 6 sierpnia 2012 — OpenGL 4.3
- 22 lipca 2013 — OpenGL 4.4
- 11 sierpnia 2014 — OpenGL 4.5
- 16 lutego 2016 — Vulkan 1.0
- 31 lipca 2017 — OpenGL 4.6

Historia

- Początek OpenGL
- OpenGL a Windows
- Ewolucja kart graficznych
- VAO
- Shadery
- Rozwój OpenGL

Szablon aplikacji

Hello, triangle!



Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!



Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

- Wieloplatformowość
 - ☐ Windows, Linux, MacOS
 - Windows, Linux, MacOS
 - iPhone, Android
 - ☐ OpenGL ES
- WebGL

Wymagania sprzętowe. OpenGL 4

Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

- NVIDIA GPU — GeForce GTX 400
- AMD/ATI GPU —Radeon HD 5000
- najnowsze sterowniki
- prawie każda karta wyprodukowana po roku 2006

Historia

Początek OpenGL

OpenGL a Windows

Ewolucja kart
graficznych

VAO

Shadery

Rozwój OpenGL

Szablon aplikacji

Hello, triangle!

■ GLFW

■ GLEW

Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

Szablon aplikacji

Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

- 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

Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

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

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

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

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

Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

- 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);
```

Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

■ 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`

Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

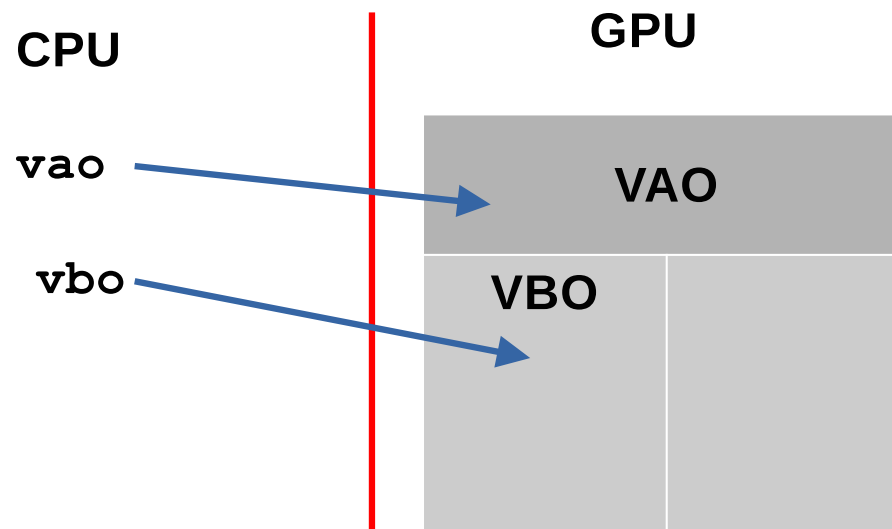
```
GLenum glew_init_result;  
glewExperimental = GL_TRUE;  
glew_init_result = glewInit();
```

- 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

- 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



Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

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

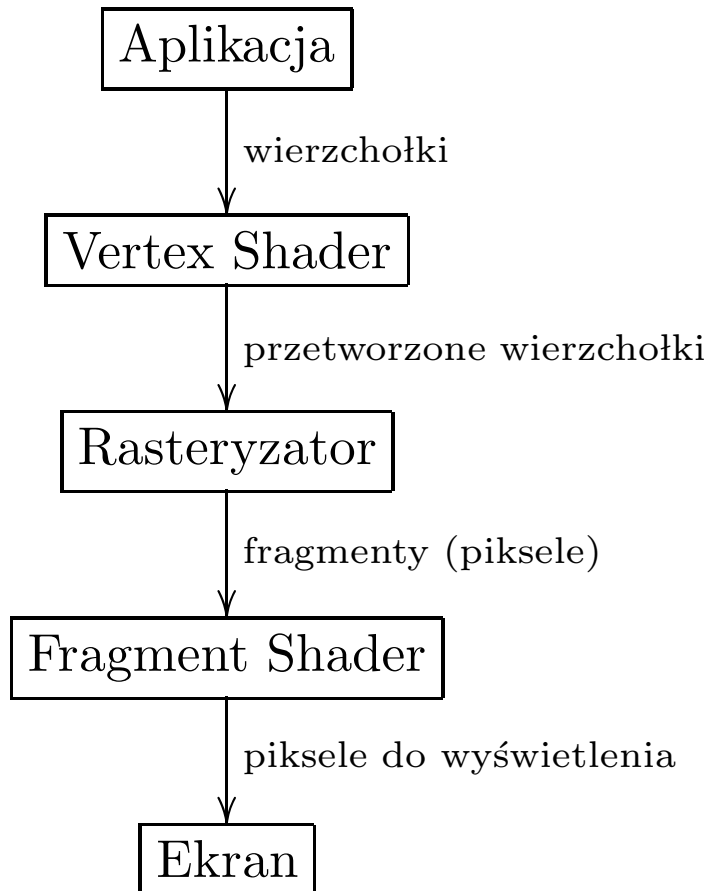
```
glGenBuffers(1, &vbo);  
glBindBuffer(GL_ARRAY_BUFFER, vbo)
```

- 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



Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

Wykorzystanie VBO w shaderze wierzchołków

Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

- 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;
```


Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

■ 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);
```

Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

■ Za pomocą GLFW:

- ☐ `glfwSetWindowSizeCallback` zmiana rozmiarów okna
- ☐ `glfwSetKeyCallback` zdarzenia klawiatury
- ☐ `glfwSetMouseButtonCallback` zdarzenia myszki
- ☐ etc

Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

```
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

Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

```
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);
```

Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

GL_POINTS, GL_LINES, GL_LINE_STRIP,
GL_LINE_LOOP, GL_TRIANGLE_STRIP,
GL_TRIANGLE_FAN

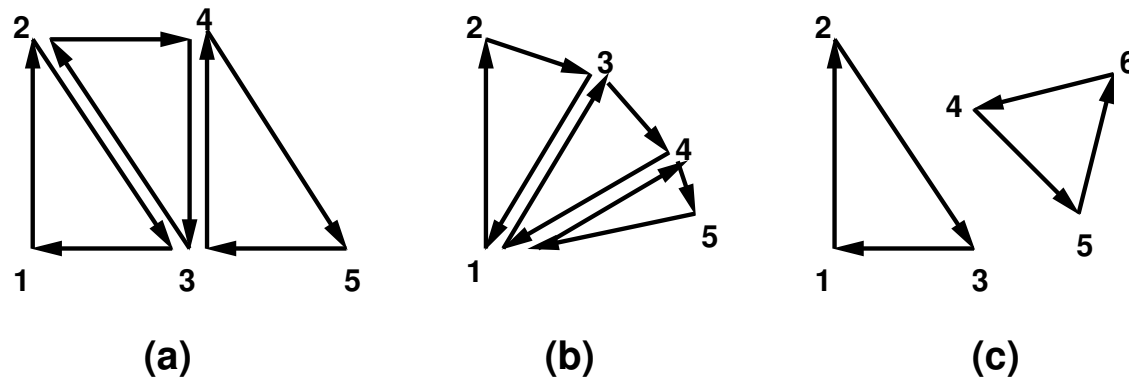


Figure 2.3. (a) A triangle strip. (b) A triangle fan. (c) Independent triangles. The numbers give the sequencing of the vertices in order within the vertex arrays. Note that in (a) and (b) triangle edge ordering is determined by the first triangle, while in (c) the order of each triangle's edges is independent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

Usuwanie programu i shaderów

Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

```
glUseProgram(0);
```

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

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

```
glDeleteProgram(program);
```

Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

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

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

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

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

Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

- 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);
```


Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

- 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);
```

Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

- 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

Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

- W OpenGL, od wersji 4.3 można zarejestrować callback na zdarzenie **error**

- ☐ inicjalizować kontekst z flagą **GLUT_DEBUG**
- ☐ 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**

Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

```
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

Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

```
glEnable(GL_DEBUG_OUTPUT_SYNCHRONOUS);
```

Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

- Można ograniczyć ilość emitowanych komunikatów:

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

Historia

Szablon aplikacji

Czynności
standardowe

Nagłówki

Kontekst

Dostęp do rozszerzeń

VAO

Kompilacja

Rejestracja callbacków

Renderowanie

Czyszczenie pamięci

Debugowanie

Hello, triangle!

```
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;
```

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

Hello, triangle!

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb



Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
#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;
```

```
}
```

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
#version 430 core
```

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

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

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

- `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](#)

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

- 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

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
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);
}
```

[Historia](#)[Szablon aplikacji](#)[Hello, triangle!](#)[Render trójkąta](#)[Shadery](#)[Klasy C++](#)[Założenie](#)[main.cpp](#)[Window](#)[Program](#)[Triangle](#)[Błędy OpenGL](#)[Makefile dla Linux](#)[gdb](#)

```
int main(void){
    window.Initialize(kMajorGLVersion,
                      kMinorGLVersion);
    glfwSetWindowSizeCallback(window, Resize);
    glfwSetKeyCallback(window, KeyEvent);

    window.Run();
    glfwTerminate();
    exit(EXIT_SUCCESS);
}
```

Klasa Window, dane publiczne

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
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_;
```


Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
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();
```

```
};
```

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

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

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
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);  
}
```

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
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
```

.....

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

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

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

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

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

Profil DEBUG — rejestracja callbacka na error

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
#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

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

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

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


Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

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

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
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;
        }
    }
}
```

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

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

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
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

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
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

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

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

Inicjalizacja. Kompilacja i linkowanie

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
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);
}
```

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

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

.....

Kompilacja shadera. Weryfikacja wyniku

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
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

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
.....  
    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;  
}
```

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
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

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
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

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
Program::~Program(){
    glUseProgram(0);

    glDetachShader(program_, vertex_shader_);
    glDetachShader(program_, fragment_shader_);

    glDeleteShader(fragment_shader_);
    glDeleteShader(vertex_shader_);

    glDeleteProgram(program_);
}
```

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
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

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
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
    };
}
```

.....

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

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


Inicjalizacja. VBO dla wierzchołków

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
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

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
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);
```

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

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

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
void Triangle::Draw(Program program){
    glUseProgram(program);
    glBindVertexArray(vao_);

    glDrawArrays(GL_TRIANGLES, 0, 3);

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

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
Triangle::~Triangle(){
    glDisableVertexAttribArray(1);
    glDisableVertexAttribArray(0);

    glBindBuffer(GL_ARRAY_BUFFER, 0);

    glDeleteBuffers(1, &color_buffer_);
    glDeleteBuffers(1, &vertex_buffer_);

    glBindVertexArray(0);
    glDeleteVertexArrays(1, &vao_);
}
```

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
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;
```

[Historia](#)[Szablon aplikacji](#)[Hello, triangle!](#)[Render trójkąta](#)[Shadery](#)[Klasy C++](#)[Założenie](#)[main.cpp](#)[Window](#)[Program](#)[Triangle](#)[Błędy OpenGL](#)[Makefile dla Linux](#)[gdb](#)

```
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;
```

}

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
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

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

gdb

```
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
```

Historia

Szablon aplikacji

Hello, triangle!

Render trójkąta

Shadery

Klasy C++

Założenie

main.cpp

Window

Program

Triangle

Błędy OpenGL

Makefile dla Linux

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](#)