

Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

| | | | |
|--|--|--------------------|-------------------------------|
| Grafika komputerowa i komunikacja człowiek-komputer | | | |
| Temat: | Laboratorium 7 – Potok graficzny oparty o shadery | | |
| Termin zajęć: | środa TN 14:15 – 17:15 semestr zimowy 2023/2024 | | |
| Autor: | Eryk Mika 264451 | Prowadzący: | Dr inż. arch. Tomasz Zamojski |

1. Cel laboratorium

Celem laboratorium było poznanie elementów współczesnego potoku graficznego. Analizowano wykorzystanie jednostek cieniujących w OpenGL oraz sposoby przekazywania danych wierzchołków do karty graficznej. Zaznajomiono się z mechanizmem rysowania instancyjnego.

2. Kod wspólny dla wszystkich zadań

Rozwiązania zadań z tego laboratorium są znacznie odmienne od rozwiązań zadań z poprzednich ćwiczeń ze względu na wykorzystanie wstawek języka *OpenGL Shading Language* (GLSL). Posiada on składnię podobną do języka C i jest dostosowany do potrzeb grafiki komputerowej. Programy napisane w języku GLSL – **shadery** – są wykonywane równolegle na procesorze graficznym (GPU) i są one linkowane (łączone) z programem wykonywanym „klasycznie” na procesorze (CPU).

Kod realizujący podstawowy potok graficzny z wykorzystaniem shaderów wierzchołków oraz fragmentów został dostarczony przez Prowadzącego. Shader wierzchołków (*vertex_shader_source*) jest uruchamiany dla każdego wierzchołka z osobna i jest zwykle użyty do transformacji jego położenia. Shader fragmentów (*fragment_shader_source*) jest uruchamiany dla każdego wynikowego fragmentu rasteryzacji – rysowania – użyty jest do określenia koloru piksela. Shadery umieszczone są wewnątrz funkcji *compile_shaders()* (Rysunek 2.1).

```
24     vertex_shader_source = ""
25         #version 330 core
26
27         layout(location = 0) in vec4 position;
28
29         uniform mat4 M_matrix;
30         uniform mat4 V_matrix;
31         uniform mat4 P_matrix;
32
33         void main(void) {
34             gl_Position = P_matrix * V_matrix * M_matrix * position;
35         }
36     ""
37
38     fragment_shader_source = ""
39         #version 330 core
40
41         out vec4 color;
42
43         void main(void) {
44             color = vec4(0.7, 0.7, 0.7, 1.0);
45         }
46     ""
```

Rysunek 2.1 Kod shaderów wewnątrz funkcji *compile_shaders()*

Dane mogą być zapisywane bezpośrednio w shaderach – mogą być typu *uniform* – są one dostępne we wszystkich shaderach, oraz typu *in/out* – są one wtedy przekazywane odpowiednio do/z danych shaderów w potoku. Możliwa jest także wymiana danych z głównym programem. W tym celu stosuje się, między innymi, bufor. Są one deklarowane z określoną nazwą, zapisywane są w nich dane, a następnie następuje dowiązanie bufora do kontekstu *OpenGL*.

Najważniejszym buforem wykorzystywanym w rozwiązaniach wszystkich zadań jest bufor *vertex_buffer* będący odniesieniem do macierzy/tablicy *vertex_positions*, która przechowuje informacje o położeniach wierzchołków. Za pomocą funkcji *glBindBuffer()* określa się przeznaczenie bufora, a następnie dane do niego są kopiowane z listy *vertex_positions* za pomocą funkcji *glBufferData()*. Ostatecznie, wywołanie funkcji *glVertexAttribPointer()* określa, w jaki sposób zawartość bufora ma być rozdzielona pomiędzy kolejno uruchamiane instancje shadera. Wywołanie funkcji *glEnableVertexAttribArray()* powoduje przekazywanie wycinków bufora do shaderów¹ (Rysunek 2.2). Tablica w linii 106. została zwinięta ze względu na jej znaczny rozmiar.

```
106 > vertex_positions = numpy.array([ ...
155
156 vertex_buffer = glGenBuffers(1)
157 glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer)
158 glBufferData(GL_ARRAY_BUFFER, vertex_positions, GL_STATIC_DRAW)
159
160 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, ctypes.c_void_p(0))
161 glEnableVertexAttribArray(0)
```

Rysunek 2.2 Obsługa bufora związanego z *vertex_positions* w funkcji *startup()*

Wewnątrz funkcji *render()* następuje przekazanie zmiennych z głównego programu do shaderów. Następuje powiązanie lokalizacji w programie ze zmiennymi typu *uniform* w przedstawionych shaderach (Rysunek 2.3).

```
188 M_location = glGetUniformLocation(rendering_program, "M_matrix")
189 V_location = glGetUniformLocation(rendering_program, "V_matrix")
190 P_location = glGetUniformLocation(rendering_program, "P_matrix")
191 glUniformMatrix4fv(M_location, 1, GL_FALSE, glm.value_ptr(M_matrix))
192 glUniformMatrix4fv(V_location, 1, GL_FALSE, glm.value_ptr(V_matrix))
193 glUniformMatrix4fv(P_location, 1, GL_FALSE, glm.value_ptr(P_matrix))
```

Rysunek 2.3 Fragment funkcji *render()* – powiązanie lokalizacji zmiennych w głównym programie oraz shaderach

¹ <https://learnopengl.com/Getting-started/Shader>

3. Zadanie na ocenę 3.0 – skrypt *lab3_0.py*

W zadaniu tym należało zaobserwować działanie dostarczonego skryptu oraz go zmodyfikować – zmienić kolor wyświetlanego sześcianu.

W celu realizacji zadania zmodyfikowano kod shaderów – wierzchołków oraz fragmentów – w sposób opisany w instrukcji do laboratorium. W shaderze wierzchołków utworzono nową zmienną *out vec4 vertex_color*, której wartość została przypisana wewnątrz funkcji *main()* – poprzez przypisanie wektora wartości *vec4* (Rysunek 3.1). W shaderze fragmentów utworzono natomiast nową zmienną wejściową *in vec4 vertex_color*, której nazwa pokrywa się z nazwą zmiennej wyjściowej z shadera wierzchołków. W funkcji *main()* shadera fragmentów następuje przypisanie tej wartości do zmiennej *color* (Rysunek 3.2).

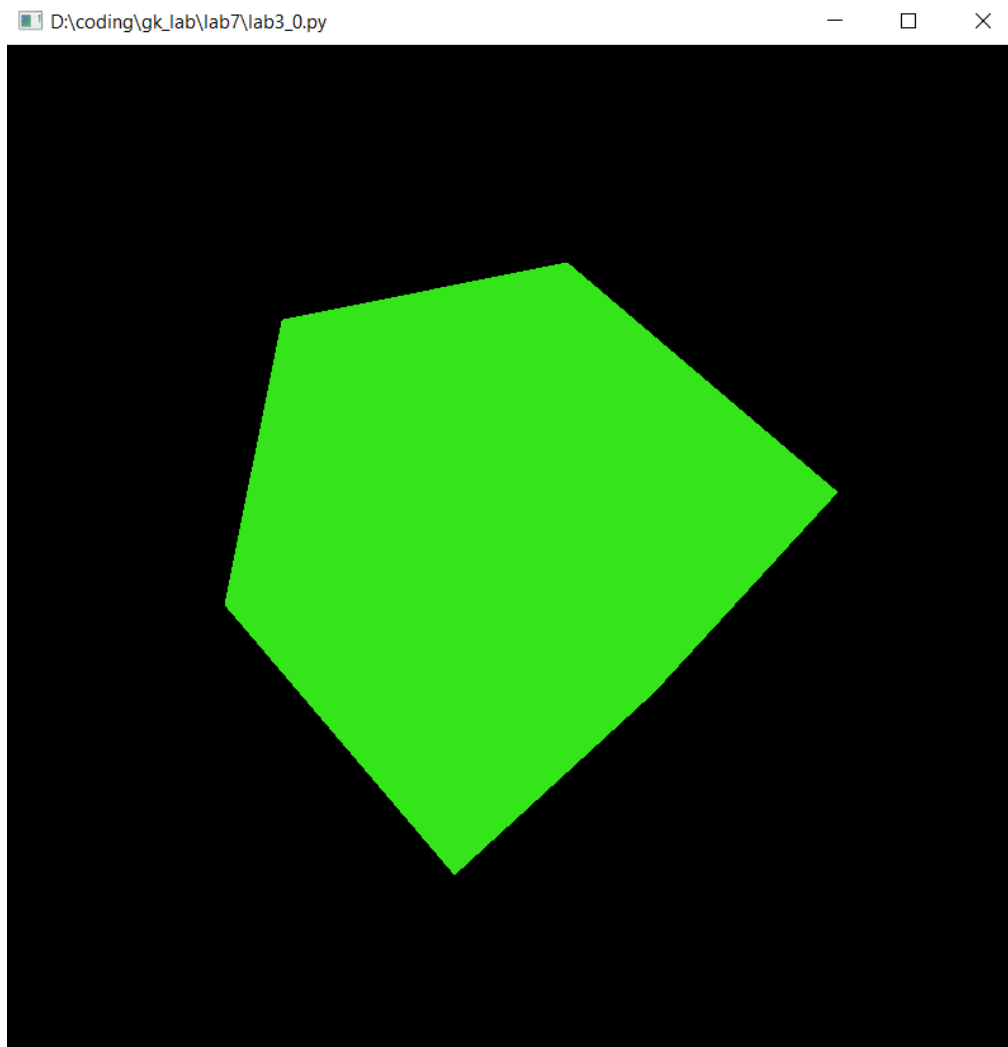
```
24 vertex_shader_source = """
25     #version 330 core
26
27     layout(location = 0) in vec4 position;
28
29     uniform mat4 M_matrix;
30     uniform mat4 V_matrix;
31     uniform mat4 P_matrix;
32     out vec4 vertex_color;
33
34     void main(void) {
35         gl_Position = P_matrix * V_matrix * M_matrix * position;
36         vertex_color = vec4(0.2, 0.9, 0.1, 1.0);
37     }
38 """
```

Rysunek 3.1 Shader wierzchołków dla zadania na ocenę 3.0

```
40 fragment_shader_source = """
41     #version 330 core
42
43     in vec4 vertex_color;
44     out vec4 color;
45
46     void main(void) {
47         color = vertex_color;
48     }
49 """
```

Rysunek 3.2 Shader fragmentów dla zadania na ocenę 3.0

Efekt działania skryptu przedstawia Rysunek 3.1.



Rysunek 3.1 Efekt działania skryptu na ocenę 3.0

4. Zadanie na ocenę 3.5 – skrypt *lab3_5.py*

W zadaniu tym należało zmodyfikować kolory bryły z poprzedniego zadania tak, aby każdy bok sześcianu miał inny kolor.

W celu realizacji zadania w shaderze wierzchołków została zadeklarowana nowa zmienna *in vec4 color*, która jest następnie przekazywana do shadera fragmentów poprzez zmienną typu *out – vertex_color* (Rysunek 4.2). Zmienna *color* jest określana z indeksem 1, dzięki czemu można ją powiązać z danymi z głównego programu².

² https://www.khronos.org/opengl/wiki/Vertex_Shader#Inputs

```

25 vertex_shader_source = ""
26     #version 330 core
27
28     layout(location = 0) in vec4 position;
29     layout(location = 1) in vec4 color;
30
31     uniform mat4 M_matrix;
32     uniform mat4 V_matrix;
33     uniform mat4 P_matrix;
34     out vec4 vertex_color;
35
36     void main(void) {
37         gl_Position = P_matrix * V_matrix * M_matrix * position;
38         vertex_color = color;
39     }
40 ""
41
42 fragment_shader_source = ""
43     #version 330 core
44
45     out vec4 frag_color;
46     in vec4 vertex_color;
47
48     void main(void) {
49         frag_color = vec4(vertex_color);
50     }
51 ""

```

Rysunek 4.2 Kod shaderów dla zadania na ocenę 3.5

W głównym kodzie programu – wewnątrz funkcji *startup()* zmodyfikowana została tablica *vertex_positions*. W pierwotnej wersji definiuje ona położenia wierzchołków trójkątów tworzących poszczególne boki sześcianu – po 2 trójkąty na każdy bok. W związku z tym 6 kolejnych wierszy w tej tablicy powiązane jest z jednym bokiem sześcianu. Zastosowana modyfikacja polega na dopisaniu w każdym wierszu 3 liczb określających składowe RGB koloru danej ściany – takich samych dla 6 kolejnych wierszy (Rysunek 4.3).

```

106     vertex_positions = numpy.array([
107         -0.25, +0.25, -0.25, 0.0, 0.0, 1.0,
108         -0.25, -0.25, -0.25, 0.0, 0.0, 1.0,
109         +0.25, -0.25, -0.25, 0.0, 0.0, 1.0,
110
111         +0.25, -0.25, -0.25, 0.0, 0.0, 1.0,
112         +0.25, +0.25, -0.25, 0.0, 0.0, 1.0,
113         -0.25, +0.25, -0.25, 0.0, 0.0, 1.0,
114
115         +0.25, -0.25, -0.25, 1.0, 1.0, 0.0,
116         +0.25, -0.25, +0.25, 1.0, 1.0, 0.0,
117         +0.25, +0.25, -0.25, 1.0, 1.0, 0.0,
118
119         +0.25, -0.25, +0.25, 1.0, 1.0, 0.0,
120         +0.25, +0.25, +0.25, 1.0, 1.0, 0.0,
121         +0.25, +0.25, -0.25, 1.0, 1.0, 0.0,
122
123         +0.25, -0.25, +0.25, 0.604, 0.075, 0.639,
124         -0.25, -0.25, +0.25, 0.604, 0.075, 0.639,
125         +0.25, +0.25, +0.25, 0.604, 0.075, 0.639,
126
127         -0.25, -0.25, +0.25, 0.604, 0.075, 0.639,
128         -0.25, +0.25, +0.25, 0.604, 0.075, 0.639,
129         +0.25, +0.25, +0.25, 0.604, 0.075, 0.639,

```

Rysunek 4.3 Fragment tablicy `vertex_positions` dla zadania na ocenę 3.5

Następnie, w sposób opisany wcześniej, tworzony jest bufor dla koloru. Następuje powiązanie z wartościami z tablicy `vertex_positions`. W wywołaniach funkcji `glVertexAttribPointer()` zmodyfikowano argument `stride` na 24 bajty – aby uwzględnić przesunięcie wywołane dodaniem nowych wartości do tablicy (32-bitowe liczby typu `float`, czyli 4-bajtowe liczby, 4 bajty x 3 = 24 bajty). Jednocześnie, dla bufora koloru ustawiono wskaźnik rozpoczęcia danych na 12 bajtów (Rysunek 4.4).

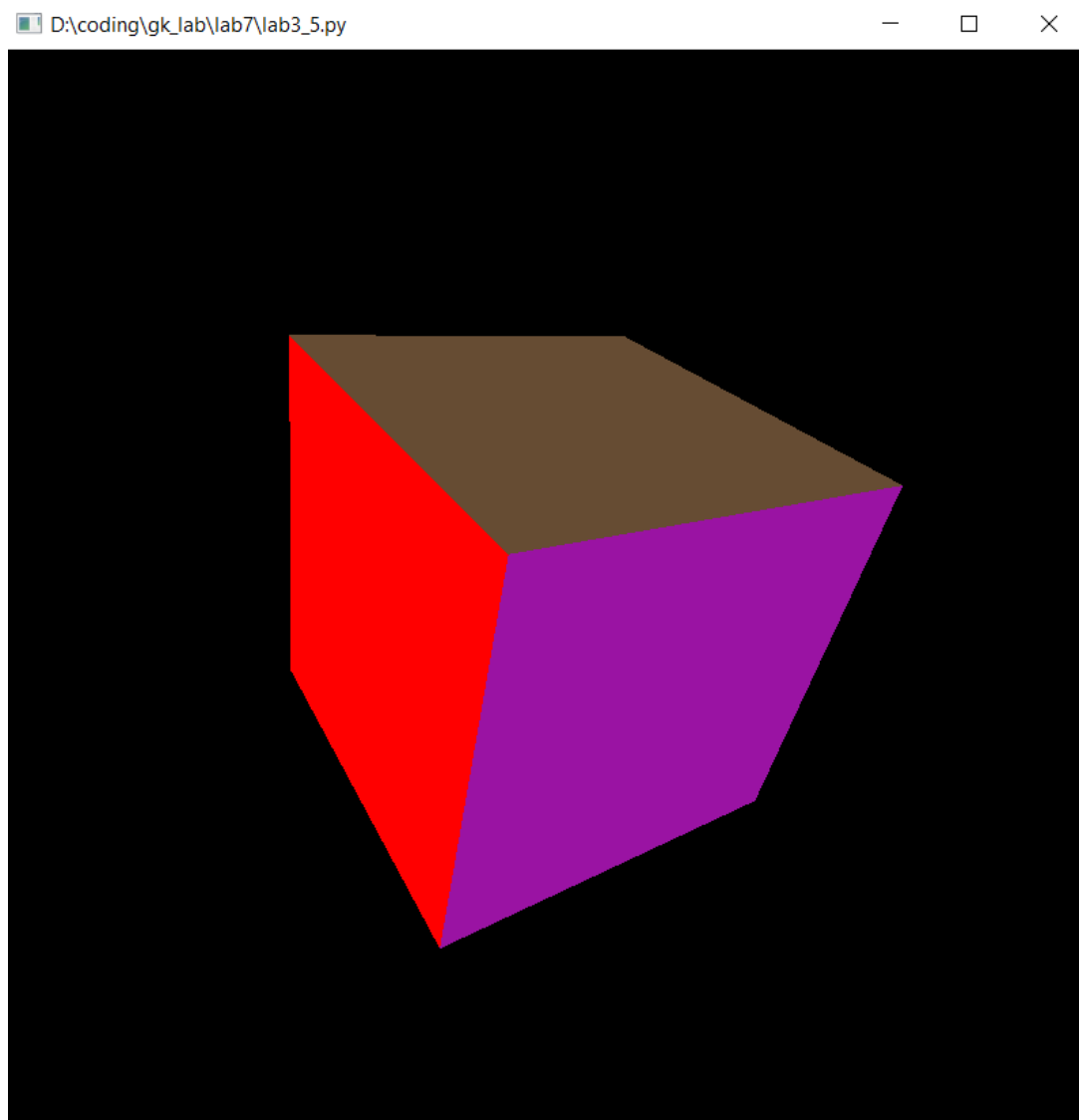
```

156     vertex_buffer = glGenBuffers(1)
157     color_buffer = glGenBuffers(1)
158     glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer)
159     glBufferData(GL_ARRAY_BUFFER, vertex_positions, GL_STATIC_DRAW)
160     glBindBuffer(GL_ARRAY_BUFFER, color_buffer)
161     glBufferData(GL_ARRAY_BUFFER, vertex_positions, GL_STATIC_DRAW)
162
163     glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 24, ctypes.c_void_p(0))
164     glEnableVertexAttribArray(0)
165     glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 24, ctypes.c_void_p(12))
166     glEnableVertexAttribArray(1)

```

Rysunek 4.4 Powiązanie buforów dla zadania na ocenę 3.5

Efekt działania skryptu przedstawia Rysunek 4.5.



Rysunek 4.5 Efekt działania skryptu na ocenę 3.5