WMM – Lab 8
Generowanie grafiki z wykorzystaniem popularnej biblioteki graficznej

Autor: Łukasz Dąbała, Michał Chwesiuk

1 Wstęp

Celem laboratorium 8 jest zapoznanie się z biblioteką graficzną OpenGL oraz podstawami programowania na GPU. W tym celu do zrealizowania będą 3 ćwiczenia poruszające zagadnienia wyświetlania grafiki z jej użyciem.

Przydatne linki:

1. Dokumentacja OpenGL - https://www.khronos.org/registry/OpenGL-Refpages/gl4/

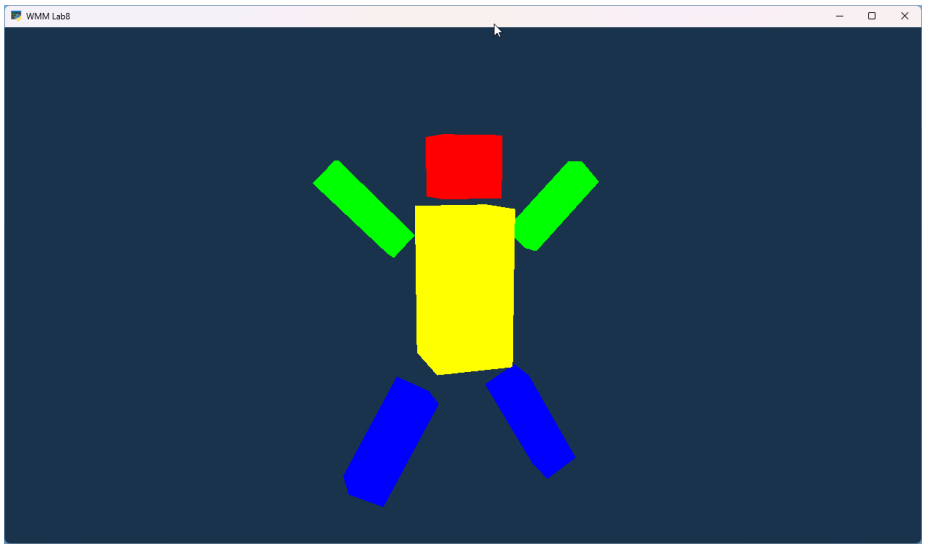
2 Wymagania

W celu realizacji laboratorium niezbędny jest interpreter języka Python w wersji 3.11.3. Aby ułatwić też sobie pracę, preferowanym narzędziem do pisania kodu będzie środowisko programistyczne PyCharm. Ze względu na pracę z shaderami, przydatna może być też wtyczka do tego programu: GLSL Support, która dodaje podstawowe kolorowanie składni.

W plikach dołączonych do zadania, znajduje się kod stanowiący podstawę do modyfikacji w trakcie trwania laboratorium. W środku archiwum znajdują się pliki Pythonowe oraz wzorcowe shadery, które będą mogły stanowić bazę do wykonania ćwiczeń. Sam program umożliwia wczytanie shaderów z plików *.frag* (*fragment shader*) i *.vert* (*vertex shader*) oraz wyświetlenie podstawowego okna. Aby ułatwić instalację niezbędnych pakietów w paczce znajduje się plik *requirements.txt*.

3 Sprawozdanie i kod

Kod tworzony w ramach laboratorium powinien być napisany w sposób zrozumiały oraz powinien być skomentowany. Komentarze w kodzie powinny tłumaczyć co dzieje się w danym miejscu np. *obliczanie wektora z danego wierzchołka w kierunku źródła światła*. Do opisu działania można wykorzystać zarówno język polski i angielski.



Rysunek 1: Przykładowe złożenie transformacji dla obiektów w cel ułożenia z nich robota.

Dodatkowo należy stworzyć sprawozdanie, które będzie dokumentować działanie stworzonych programów. W opisie każdego zadania znajdują się również elementy, które powinny znaleźć się w sprawozdaniu.

4 Zadanie 1: Składanie transformacji (2.0 pkt)

Celem tego zadania jest złożenie transformacji obiektów w scenie 3D. Efektem końcowym wykonanego zadania powinna być scena zawierająca robota złożonego z trójwymiarowych sześcianów, tak jak na rysunku 1. Zadanie powinno zostać wykonane w klasie $RobotWindow$, która znajduje się w pliku $robot\_window.py$.

Pierwszym etapem zadania jest wygenerowanie buforu pamięci znajdującego się w pamięci GPU (Vertex Array Object - VAO), który będzie zawierał dane geometrii przedstawianego sześcianu. Funkcja, która realizuje ten etap jest zawarta w dostarczonym kodzie, i znajduje się ona w pliku $models.py$ pod nazwą $load\_cube()$.

Kolejnym etapem jest napisanie programu cieniującego (ang. *shader program*). Programy cieniujące zaimplementowane do realizacji zadania powinny znaleźć się w folderze $resources/shaders/robot$, jako pliki $robot.vert$ i $robot.frag$. W odróżnieniu od shader'ów zawartych w dostarczonym projekcie ($passthrough.vert$ i $passthrough.frag$), które wyłącznie przenosiły pozycję dwuwymiarowych wierzchołków do kolejnych etapów renderingu. Shader'y potrzebne do wykonania tego zadania będą miały za zadanie przekształcenie trójwymiarowych wierzchołków do dwuwymiarowej przestrzeni obrazu wyświetlanego na ekranie. Tą operacje realizuje się za pomocą formuły:

$$gl\_Position = P * V * M * vert_{in}$$

P, V, M oznaczają kolejno macierz projekcji, widoku i modelu o rozmiarze 4x4, które w $GLSL$ są typu $mat4$. $vert_{in}$ oznacza wejściowy trójwymiarowy wierzchołek w postaci jednorodnej, a $gl\_Position$ oznacza wierzchołek przekształcony do przestrzeni obrazu przekazywany do kolejnych etapów renderingu.

Następnie, należy skonstruować macierze, które będziemy przekazywać do programu cieniującego. W tym celu należy wykorzystać pakiet *pyrr*. Przydatne będą następujące metody:

1. `Matrix44.perspective_projection`

   - macierz projekcji perspektywicznej

2. `Matrix44.look_at`

   - macierz w celu skierowania wzroku na dany punkt

Macierz modelu należy skonstruować z pod-macierzy realizujących poszczególne przekształcenia geometryczne:

1. `Matrix44.from_translation`

   - macierz translacji tworzona z wektora

2. `Matrix44.from_scale`

   - macierz skalowania tworzona z wektora skali dla poszczególnych osi

3. `Matrix44.from_eulers`

   - macierz obrotu tworzona z podanych kątów Eulera

4. `Matrix44.from_x_rotation`

   `Matrix44.from_y_rotation`

   `Matrix44.from_z_rotation`

   - macierze obrotu wokół odpowiednich osi tworzona z podanej rotacji

Generowanie macierzy powinno następować za każdym razem, gdy chcemy generować ramkę obrazu (w funkcji $render()$).

Wygenerowane macierze w kodzie Python należy przekazać do shader'a. W tym celu zadeklarować zmienne w kodzie shader'a ze słowem kluczowym $uniform$. Następnie, należy pobrać lokacje (identyfikator) tych zmiennych w przygotowanej funkcji $init\_shaders\_variables()$. Przykład:

$$self.uniform\_location = self.program['variable\_name']$$

Mając lokacje, możemy przekazać wartości do zmiennej $uniform$ w programie cieniującym:

$$self.uniform\_location.write(vector\_variable)$$

$$self.uniform\_location.write(matrix\_variable.astype('f4'))$$

Po implementacji przekształceń geometrycznych, możemy wygenerować kilka sześcianów o różnych pozycjach, rozmiarach, i rotacjach, w taki sposób, by przypominały robota. Przykładowa konfiguracja: Podstawowe kształty, które dostarczone są w ramach zasobów do laboratorium należy poddać modyfikacjom:

1. głowa - translacja o wektor $t = (0, 5, 0)$, skalowanie we wszystkich osiach ze współczynnikiem 1,5.
2. ciało - translacja o wektor $t = (0, 2, 0)$, skalowanie w osiach XYZ o wektor $s = (2, 4, 2)$
3. ręce - translacja o wektor $t = (-/+2.5, 4, 0)$, rotacja o kąt $-/+45^o$ wokół osi z, skalowanie ze współczynnikami $s = (0.75, 2.5, 0.75)$
4. nogi - translacja o wektor $t = (-/+2, -2, 0)$, rotacja o kąt $+/-30^o$ wokół osi z, skalowanie ze współczynnikami $s = (1, 3, 1)$

Jak można zauważyć ma rysunku 1, obiekty są również pokolorowane na różne sposoby, w związku z tym należy zmodyfikować shader'y.

1. **vertex shader** - przekształcenie geometryczne
2. **fragment shader** - dodanie koloru, który jest identyczny dla wszystkich fragmentów ($uniform\ vec3$)

W sprawozdaniu należy zawrzeć wizualizację stworzonego robota w innym kolorze lub przekształceniu niż ten pokazany w instrukcji.

5 Zadanie 2: Cieniowanie (1.0 pkt)

Celem tego zadania jest implementacja modelu cieniowania Phonga. W instrukcji będzie podany sposób implementacji składowych ambient i diffuse, który należy rozbudować o składową specular. Parametry światła mogą być zawarte w kodzie, natomiast parametry materiału (obiektu) powinny być przekazywane do zmiennych $uniform$. Zadanie powinno zostać wykonane w klasie $PhongWindow$, która znajduje się w pliku $phong\_window.py$. Programy cieniujące powinny znaleźć się w katalogu $resources/shaders/phong$ w plikach $phong.vert$ i $phong.frag$.

Większość obliczeń cieniowania Phonga będzie wykonywana w fragment shaderze. Cześć danych, tj. pozycja wierzchołka i wektor normalny, znajduje się w

vertex shaderze, stąd w pierwszej kolejności musimy te dane przekazać do fragment shadera.

W vertex shaderze deklarujemy zmienną wyjściową dla pozycji wierzchołka:

$$out\ vec3\ v\_position;$$

W fragment shaderze deklarujemy odpowiadającą zmienną wejściową:

$$in\ vec3\ v\_position;$$

W celu przekazania pozycji wierzchołka do przetworzenia przez proces rasteryzacji i fragment shadera, przypisujemy ją do zadeklarowanej zmiennej w funkcji $main()$ vertex shadera:

$$v\_position = (M * vec4(in\_position, 1.0)).xyz;$$

Tak samo należy postąpić z wektorem normalnym:

$$v\_normal = (M * vec4(in\_normal, 0.0)).xyz;$$

W fragment shaderze należy zadeklarować zmienne przechowujące parametry światła, parametry materiału i pozycje kamery:

$$const\ vec3\ light\_position = vec3(0.0, 7.0, -15.0);$$
$$const\ vec3\ light\_ambient = vec3(0.1, 0.1, 0.1);$$
$$const\ vec3\ light\_diffuse = vec3(1.0, 1.0, 1.0);$$
$$const\ vec3\ light\_specular = vec3(1.0, 1.0, 1.0);$$

$$uniform\ vec3\ material\_ambient;$$
$$uniform\ vec3\ material\_diffuse;$$
$$uniform\ float\ material\_shininess;$$

$$uniform\ vec3\ camera\_position;$$

Należy upewnić się, że pozycja kamery przekazywana do programu cieniującego jest zgodna z wektorem pozycji kamery wykorzystanej do generacji macierzy widoku (parametr eye w $Matrix44.look\_at()$).

Składową ambient liczymy za pomocą poniższej formuły:

$$vec3\ ambient = light\_ambient * material\_ambient;$$

Do składowej diffuse musimy policzyć cosinus kąta między wektorem normalnym, a wektorem światła (wektorem od powierzchni do źródła światła):

$$vec3\ N = normalize(v\_normal);$$

$$vec3\ L = normalize(light\_position - v\_position);$$
$$float\ cosNL = clamp(dot(N, L), 0.0, 1.0);$$

Samą składową diffuse wyliczamy za pomocą poniższej formuły:

$$vec3\ diffuse = light\_diffuse * material\_diffuse * cosNL$$

Końcowy kolor obiekty wyliczamy za pomocą poniższej formuły:

$$vec3\ phong\_color = clamp(ambient + diffuse, 0.0, 1.0);$$
$$f\_color = vec4(phong\_color, 1.0);$$

Do kodu należy dodać obliczenia związane ze składową specular. GLSL oferuje szereg funkcji, które ułatwią wykonanie tego zadania:

1. `transpose(V/M)`

   - transpozycja macierzy M/wektora V

2. `inverse(M)`

   - odwrotność macierzy M

3. `normalize(V)`

   - normalizacja wektora V

4. `reflect(I, N)`

   - oblicza promień odbity od powierzchni z wykorzystaniem promienia padającego I oraz wektora normalnego N

5. `dot(V1, V2)`

   - iloczyn skalarny wektorów V1 i V2

6. `pow(a, b)`

   - podnosi liczbę a do potęgi b

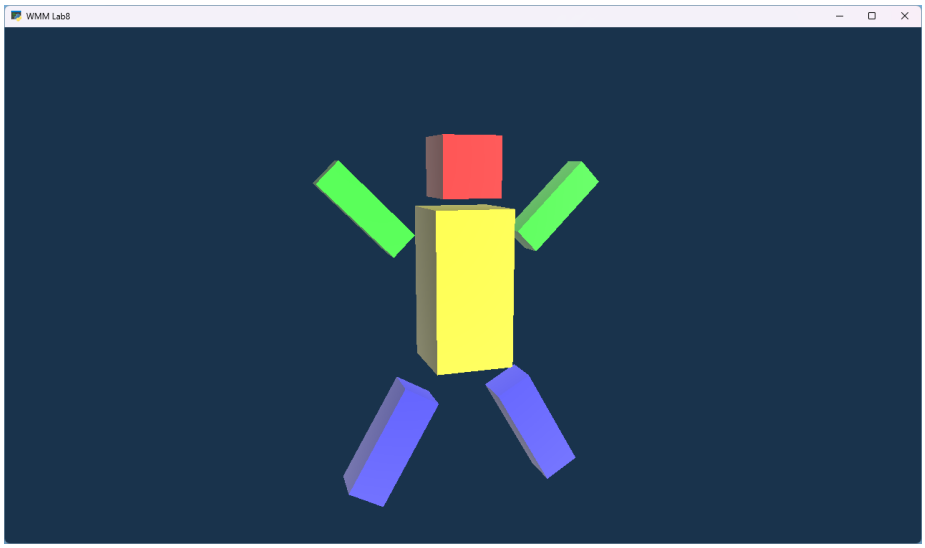
7. `max(a, b)`

   - wartość maksymalna z dwóch wartości a i b

8. `clamp(v, a, b)`

   - jeżeli wartość v jest mniejsza niż a, zwróć a, jeżeli wartość v jest większa niż b, zwróć b

W sprawozdaniu należy zawrzeć wizualizacje dla różnych kombinacji parametrów wejściowych (położenie źródła światła, różna połyskliwość obiektu, różne kolory obiektu oraz światła). Przykładowa wizualizacja znajduje się na rysunku 2.



Rysunek 2: Przykład renderu robota z zaaplikowanym oświetleniem Phonga, dla parametru połyskliwości (*shininess*) równej 50.

6 Zadanie 3: Generowanie geometrii (2.0 pkt)

Celem zadania jest implementacja własnych brył geometrycznych. Zadanie zostanie zaliczone, gdy w renderze robota znajdować się będą dwie bryły geometryczne niezamieszczone w dostarczonym przykładzie. Zadanie powinno zostać wykonane w klasie *ShapeWindow*, która znajduje się w pliku *shape_window.py*. Dobór brył jest dowolny, jako przykład można przyjąć:

1. Bryła nr 1 - Ostrosłup, wstawiony w miejsce głowy
2. Bryła nr 2 - Walec, wstawiony w miejsce tułowia

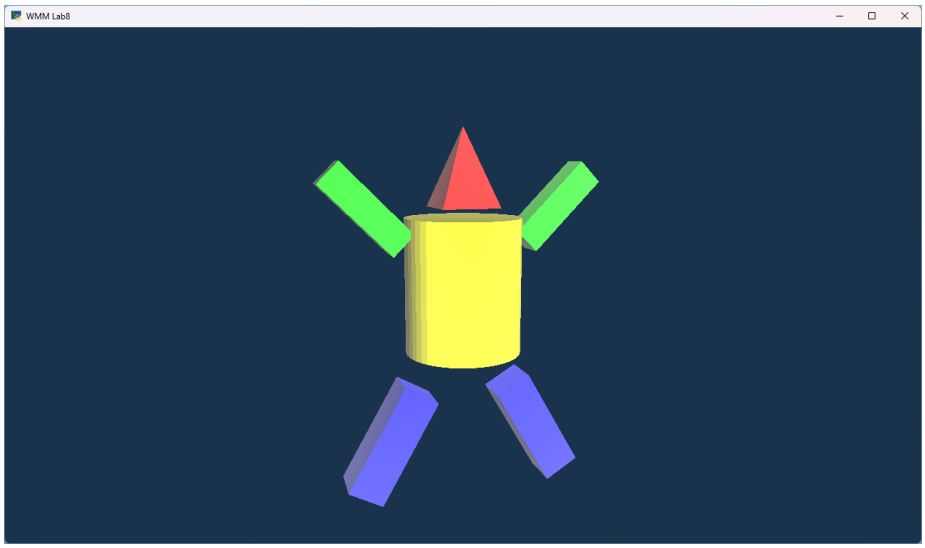
Aby stworzyć własną geometrie, należy wygenerować tablicę pozycji, które będą zawierać kolejne wierzchołki bryły. Prócz tego, należy wygenerować także tablicę wektorów normalnych, który jest wymagany do obliczeń związanych z modelem cieniowania Phonga.

Kolejnym krokiem jest deklaracja pamięci w GPU do przechowania danych o wierzchołkach. Do tego służy Vertex Array Object (VAO). Jego deklaracja jest zawarta w bibliotece moderngl:

$$from\ moderngl\_window.opengl.vao\ import\ VAO$$

$$vao = VAO()$$

Wypełnienie przestrzeni pamięci występuje za pomocą funkcji buffer:



Rysunek 3: Robot z dodatkowymi bryłami geometrycznymi w miejscach głowy i tułowia.

$$vao.buffer(positions, "3f", ["in\_position"])$$

$$vao.buffer(normals, "3f", ["in\_normal"])$$

Należy pamiętać, że funkcja $buffer()$ przyjmuje wartości tablicę jednowymiarową float'ów typu $numpy\ array$, którą możemy przygotować za pomocą formuły:

$$buffer\_data = numpy.array(buffer\_data, dtype = numpy.float32).flatten()$$

Aby korzystać ze stworzonego VAO do renderingu, należy wygenerować jego instacje dla stworzonego wcześniej programu cieniującego:

$$vao.instance(program)$$

Podczas realizowania zadania, można wspomóc się funkcją $load\_cube()$ zawartą w dołączonym projekcie, która generuje VAO i wypełnia go danymi wierzchołków sześcianu. W kodzie zamieszczona jest także funkcja $generate\_normals\_triangles()$, która generuje wektory normalne, przy czym należy zaznaczyć, że zakłada ona, że kolejność wierzchołków jest zgodna z prymitywem $GL\_TRIANGLES$.

Wiele podstawowych brył posiada gładkie powierzchnie bądź okręgi w podstawie. Przykładem jest stożek i walec. Aby wygenerować taką siatkę, można posłużyć się kodem do generowania siatki okręgu przy użyciu współrzędnych biegunowych:

```
positions = []

circleCenter = [2.0, 5.0]
radius = 3.0
segments = 50

for segment in range(segments):
    theta0 = 2.0 * math.pi * (segments + 0) / segments
    theta1 = 2.0 * math.pi * (segments + 1) / segments

    X0 = circleCenter[0] + radius * cos(theta0)
    Y0 = circleCenter[1] + radius * sin(theta0)

    X1 = circleCenter[0] + radius * cos(theta1)
    Y1 = circleCenter[1] + radius * sin(theta1)

    positions.append(circleCenter)
    positions.append([X0, Y0])
    positions.append([X1, Y1])
```

Podpowiedź: bryły nie muszą być zbudowane wyłącznie z jednego VAO, ani być renderowane jednym "render callem".