Politechnika Wrocławska

Grafika komputerowa i komunikacja człowiek-komputer

Laboratorium nr 7

Potok graficzny oparty o shadery

Szymon Datko

szymon.datko@pwr.edu.pl

Wydział Informatyki i Telekomunikacji,
Politechnika Wrocławska

semestr zimowy 2022/2023

HR EXCELLENCE IN RESEARCH

Politechnika Wrocławska

Cel ćwiczenia

1. Poznanie elementów współczesnego potoku graficznego.
2. Nauczenie się, jak wykorzystywać jednostki cieniujące w OpenGL.
3. Nauczenie się, jak przekazywać dane wierzchołków do karty graficznej.
4. Zaznajomienie się z mechanizmem rysowania instancjowego.

Politechnika Wrocławska

W skrócie – dlaczego?

- Każde wywołanie `glVertex()`, `glColor()`, itd. angażuje procesor.
  - Wąskie gardło w momencie, kiedy obsługujemy złożoną scenę.
- Efektywność w grafice wymaga zrównoleglenia operacji przetwarzania.
- Jeszcze kolejny krok: podejście **Zero Driver Overhead**.
  - Zmniejszanie narzutu związanego z implementacjami, dostarczanymi w ramach sterowników przez producentów podzespołów graficznych.
- Skutki zmienionej koncepcji programów:
  - Większe możliwości i wydajność przetwarzania.
  - Wyższy próg wejścia i złożoność implementacji.

Politechnika Wrocławska

Współczesny potok graficzny

Kolejne etapy przetwarzania w aktualnej wersji OpenGL:

- Pobranie wierzchołków.
- **Shader wierzchołków**.
- **Shader sterowania teselacją**.
- Teselacja.
- **Shader wyliczenia teselacji**.
- **Shader geometrii**.
- Składanie prymitywów.
- Rasteryzacja.
- **Shader fragmentów**.
- Działania na buforze ramki.

Pogrubione elementy listy reprezentują etapy **programowalne**.

Shadery

- Programy uruchamiane równolegle na procesorze graficznym.
- Zwykle definiuje się je przy pomocy języka **GLSL**.
  - Inne języki mogą być dostępne jako rozszerzenia.
  - Wersja OpenGL 4.6 dopuściła także język **SPIR-V**.
- Obowiązują zasady podobne, jak przy typowych programach:
  - kod źródłowy umieszcza się w **obiekcie shadera** i kompiluje,
  - skompilowane kody łączy się w **obiekt programu** (linking),
  - domyślnie cały ten proces odbywa się w locie,
  - w dołączonym przykładowym programie warto prześledzić funkcję `compile_shaders()` oraz to, gdzie wykorzystywany jest jej wynik.
- Minimalny użyteczny potok musi zawierać:
  - shader wierzchołków lub shader obliczeniowy,
  - shader fragmentów[1].

[1] Tylko jeśli coś ma zostać wyświetlone na ekranie.

Politechnika Wrocławska

Rodzaje shaderów (1/2)

- **Shader wierzchołków**:
  - uruchamiany dla każdego wierzchołka wejściowego,
  - zazwyczaj służy do transformowania położenia wierzchołków.
- **Shader sterowania teselacją** / shader sterujący:
  - przyjmuje dane z shadera wierzchołków,
  - określa poziomy podziałów dla mechanizmu teselacji,
  - wytwarza nowy zbiór wierzchołków i współczynniki teselacji.
- **Shader wyliczenia teselacji** / shader wyliczenia:
  - zostaje uruchomiony dla każdego powstałego wierzchołka,
  - pozwala określić docelową pozycję tych wierzchołków.

Politechnika Wrocławska

Rodzaje shaderów (2/2)

- **Shader geometrii**:
  - uruchamiany raz dla każdego prymitywu,
  - pozwala tworzyć nowe prymitywy i zmieniać istniejące.
- **Shader fragmentów**:
  - uruchamiany dla każdego fragmentu (wyniku rasteryzacji),
  - stosowany do określenia wynikowego koloru piksela.
- **Shader obliczeniowy**:
  - element specjalnego, niezależnego potoku,
  - nie ma określonych wejść, wyjść, ani miejsca,
  - zazwyczaj stosowany do zadań niezwiązanych z rysowaniem.

Język GLSL (1/2)

- *OpenGL Shading Language*.
- Język o składni i działaniu podobnym do języka **C**.
- Dostosowany do potrzeb grafiki komputerowej:
  - wbudowane wektorowe i macierzowe typy danych,
    - elementy dostępne jak w zwykłej tablicy,
    - można odwoływać się także jak do pól w strukturach,
    - nazwy pól można sklejać, aby wydobyć nowy wektor,
    - kolejność i powtórzenia pól – bez znaczenia,
    - `vec4 color; return color.grb; // zwraca vec3`
  - wbudowane funkcje matematyczne i pomocnicze,
  - ...

Język GLSL (2/2)

- Dostosowany do potrzeb Grafiki Komputerowej:
  - ...
  - zaprojektowany na potrzeby wysokiego zrównoleglenia,
  - brak rekurencji i ograniczona różnorodność typów,
    - 32- i 64-bitowe liczby zmiennoprzecinkowe,
    - 32 bitowe całkowite (ze znakiem i bez),
    - wartości logiczne.
  - struktury definiuje się bezpośrednio je tworząc,
    - w języku nie ma słowa kluczowego `typedef`,
  - tablice mają wbudowaną metodę `length()`,
  - rozmiar tablicy można zapisać obok jej typu:
    - `float[5] var = float[5](1.0, 2.0, 3.0, 4.0, 5.0);`

Język GLSL – przykład shadera wierzchołków

```
#version 330 core

uniform mat4 mvp;
uniform float offset;

out MY_BLOCK {
    vec2 tc;
} vs_out;

void main(void) {
    const vec2[4] position = vec2[4](
        vec2(-0.5, -0.5),
        vec2( 0.5, -0.5),
        vec2(-0.5,  0.5),
        vec2( 0.5,  0.5)
    );

    vs_out.tc = (position[gl_VertexID].xy + vec2(offset, 0.5))
                * vec2(30.0, 1.0);

    gl_Position = mvp * vec4(position[gl_VertexID], 0.0, 1.0);
}
```

Politechnika Wrocławska

Dane w OpenGL (1/2)

- Część danych można zapisać bezpośrednio w shaderach,
  - nie są problemem także wyliczenia wewnątrz shaderów,
  - choć szybkie, jest to jednak rozwiązanie dosyć ograniczone.
- Najczęściej dane do shaderów przekazuje się z aplikacji.
  - W tym celu wykorzystuje się **bufory** oraz **tekstury**.
- **Bufory** stanowią ciągły fragment zaalokowanej pamięci karty graficznej.
  - Najpierw należy zadeklarować **nazwę**, jako odnośnik bufora.
  - Następnie utworzyć **magazyn danych** – zarezerwować pamięć.
  - Później należy zapisać w buforze dane, np. przez **mapowanie**.
  - Dalej następuje **dowiązanie** bufora do kontekstu OpenGL.
  - Miejsce dowiązania określa się fachowo jako **cel** (*target*), który opisuje w jaki sposób dane z bufora będą wykorzystane.

Politechnika Wrocławska

Dane w OpenGL (2/2)

- Dane można przekazać bezpośrednio do shadera wierzchołków za pośrednictwem tak zwanego **Vertex Array Object** (*VAO*).
  - W ramach *VAO* określa się wszystkie **atrybuty wierzchołków**.
  - Przechowuje referencje do obiektów bufora (np. *VBO*).
- Kolejne etapy potoku mogą otrzymać dane, jeśli zostaną one przekazane odpowiednio dalej z shadera wierzchołków.
  - Stosuje się w tym celu słowa kluczowe `in` i `out` w GLSL.
  - Nazwy przekazywanych zmiennych lub bloków muszą być takie same w sąsiadujących shaderach w potoku.
- Alternatywnie można skorzystać z danych typu `uniform`.
  - Takie dane dostępne są od razu we wszystkich shaderach.
  - Umożliwiają jednak wyłącznie odczyt danych, bez ich zmian.

Tworzenie buforów

- W języku **C**: najpierw należy zdefiniować zmienną typu `GLuint`.
- Zmienna ta stanowi odnośnik do bufora w pamięci karty graficznej.
- Dalej należy stworzyć tak zwaną nazwę bufora (identyfikator) i zapisać ją w utworzonej zmiennej za pomocą funkcji `glGenBuffers()`.
- Na koniec wypada zwolnić zasoby karty graficznej – `glDeleteBuffers()`.

```
1  buffer = None  # W języku C: GLuint buffer;
2
3
4  def startup():
5      ...
6      global buffer
7      buffer = glGenBuffers(1)  # W języku C: glGenBuffers(1, &buffer);
8
9
10 def shutdown():
11     glDeleteBuffers(1, buffer)  # W języku C: glDeleteBuffers(1, &buffer);
12     ...
```

Wypełnianie buforów danymi

- Najpierw należy określić przeznaczenie bufora – funkcją `glBindBuffer()`.
- Później można skopiować dane – służy do tego funkcja `glBufferData()`.

```
1| void startup() {  // W języku C
2|     ...
3|     static const GLfloat vertex_positions[] = {
4|         -0.25f,  0.25f, -0.25f,
5|         -0.25f, -0.25f, -0.25f,
6|          0.25f, -0.25f, -0.25f,
7|         ...
8|     };
9|     glBindBuffer(GL_ARRAY_BUFFER, buffer);
10|    glBufferData(GL_ARRAY_BUFFER, sizeof(vertex_positions),
11|                 vertex_positions, GL_STATIC_DRAW);
12| }
```

```
1| def startup():  # W języku Python
2|     ...
3|     vertex_positions = numpy.array([
4|         -0.25, +0.25, -0.25,
5|         -0.25, -0.25, -0.25,
6|         +0.25, -0.25, -0.25,
7|         ...
8|     ], dtype='float32')
9|     glBindBuffer(GL_ARRAY_BUFFER, buffer)
10|    glBufferData(GL_ARRAY_BUFFER, vertex_positions, GL_STATIC_DRAW)
```

Przekazanie danych do shadera wierzchołków

- Dane z bufora dowiązanego jako `GL_ARRAY_BUFFER` mogą zostać przekazane bezpośrednio na wejście potoku / shadera wierzchołków.
- Funkcja `glVertexAttribPointer()` określa sposób przekazywania danych.
  - W jaki sposób pamięć (zawartość bufora) zostaje podzielona i rozłożona pomiędzy wszystkie kolejno uruchomione instancje shadera wierzchołków.
- Mechanizm przekazu uaktywnia wywołanie `glEnableVertexAttribArray()`.
- Każda uruchomiona instancja shadera otrzymuje inny wycinek bufora!
  - Tutaj: każda otrzyma 3 kolejne liczby zmiennoprzecinkowe z bufora.

```
1| void startup() {  // W języku C
2|     ...
3|     glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
4|     glEnableVertexAttribArray(0);
5| }
```

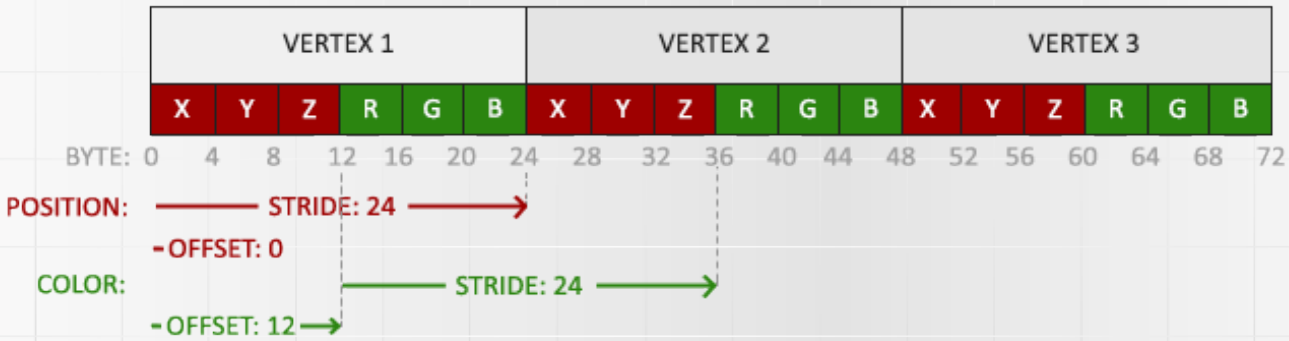
```
1| def startup():  # W języku Python
2|     ...
3|     glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, None)
4|     glEnableVertexAttribArray(0)
```

Szczegóły funkcji `glVertexAttribPointer()`

- Specyfikacja tej funkcji w języku **C**:

```
void glVertexAttribPointer( GLuint index, GLint size, GLenum type,
                            GLboolean normalized, GLsizei stride,
                            const void * pointer);
```

- `index` – numer zmiennej wejściowej shadera wierzchołków (pierwszy to 0),
  - to ta sama wartość co w funkcji `glEnableVertexAttribArray()`,
- `size` – liczba elementów przekazywanych z bufora do zmiennej (1, 2, 3 lub 4),
- `type` – typ (i rozmiar) pojedynczego elementu danych w buforze,
- `normalized` – czy dane całkowite mają być przeliczone do zakresu [0.0; 1.0],
- `stride` – co ile Bajtów w buforze znajdują się kolejne dane wejściowe,
  - wartość `0` oznacza gęste upakowanie (czyli ~ `size * type`),
- `pointer` – w którym miejscu bufora rozpoczynają się dane. (**Uwaga!**)
  - W bibliotece **PyOpenGL** należy zachować tutaj typ z języka **C**!
  - Poprawną wartością jest `None`, `0` lub np. `ctypes.c_void_p(12)`!



Źródło obrazka: https://learnopengl.com/Getting-started/Shaders

Wykorzystanie w ramach shadera wierzchołków

- Deklarujemy zmienną wejściową (`in`) typu `vec4` o nazwie `position`,
  - nazwa tej zmiennej nie ma znaczenia w tym miejscu potoku,
  - dobrą praktyką jest przypisanie indeksu przez kwalifikator `layout`:
    - unikamy wtedy automatycznego numerowania, które okazuje się niedeterministyczne (zależnie od sterownika i kolejności/nazw),
    - składnia: `layout(location = <indeks>) in <typ> <nazwa>`,
    - szczegóły: https://www.khronos.org/opengl/wiki/Vertex_Shader#Inputs;
  - ostatnia składowa będzie miała ustawioną wartość domyślną: 1.0.
- Dodatkowo zapowiadamy tutaj trzy zmienne `uniform` typu `mat4`.

```
1  def compile_shaders():
2      vertex_shader_source = """
3          #version 330 core
4
5          layout(location = 0) in vec4 position;
6
7          uniform mat4 M_matrix;
8          uniform mat4 V_matrix;
9          uniform mat4 P_matrix;
10
11         void main(void) {
12             gl_Position = P_matrix * V_matrix * M_matrix * position;
13         }
14     """
15     ...
```

Uzupełnienie zmiennych typu uniform

- Najpierw pobrać adres zmiennych z shadera: `glGetUniformLocation()`,
  - trzeba to zrobić w konkretnej skompilowanej instancji programu,
  - szukamy konkretnej nazwy zmiennej (kolejność nie ma znaczenia).
- Znalezione adresy można uzupełnić danymi: `glUniformMatrix4fv()`,
  - rozmiar pojedynczych danych określony jest w nazwie funkcji (`f`).

```
1| def render(time):
2|     ...
3|     M_matrix = glm.rotate(glm.mat4(1.0), time, glm.vec3(1.0, 1.0, 0.0))
4|     V_matrix = glm.lookAt(
5|         glm.vec3(0.0, 0.0, 1.0),
6|         glm.vec3(0.0, 0.0, 0.0),
7|         glm.vec3(0.0, 1.0, 0.0)
8|     )
9|
10|     glUseProgram(rendering_program)
11|     M_location = glGetUniformLocation(rendering_program, "M_matrix")
12|     V_location = glGetUniformLocation(rendering_program, "V_matrix")
13|     P_location = glGetUniformLocation(rendering_program, "P_matrix")
14|     glUniformMatrix4fv(M_location, 1, GL_FALSE, glm.value_ptr(M_matrix))
15|     glUniformMatrix4fv(V_location, 1, GL_FALSE, glm.value_ptr(V_matrix))
16|     glUniformMatrix4fv(P_location, 1, GL_FALSE, glm.value_ptr(P_matrix))
17|     ...
```

Wyświetlanie wielu obiektów

- Rysowanie następuje z chwilą wywołania funkcji `glDrawArrays()`.
- Aby narysować kilka kopii obiektu, wystarczy wywołać ją kilka razy.
- Żeby obiekty nie znajdowały się w tym samym miejscu należy dokonać zmian w macierzy transformacji – na przykład przesunąć kolejne kopie.
- Obciążamy CPU wywołaniami rysowania i obliczeniami, chociaż w pewnych szczególnych przypadkach można by tego uniknąć.
- Przykładowy kod:

```
1| def render(time):
2|     ...
3|
4|     for i in range(10):
5|         M_matrix = glm.translate(M_matrix, glm.vec3(1.0, 0.0, 0.0))
6|         glUniformMatrix4fv(M_location, 1, GL_FALSE, glm.value_ptr(M_matrix))
7|
8|         glDrawArrays(GL_TRIANGLES, 0, 36)
```

Mechanizm rysowania instancjowego

- Pozwala wygenerować wiele kopii tego samego obiektu w bardziej efektywny sposób – bezpośrednio na samej karcie graficznej.
- Aby narysować 10 kopii obiektu, należy wykonać zmianę funkcji rysującej:

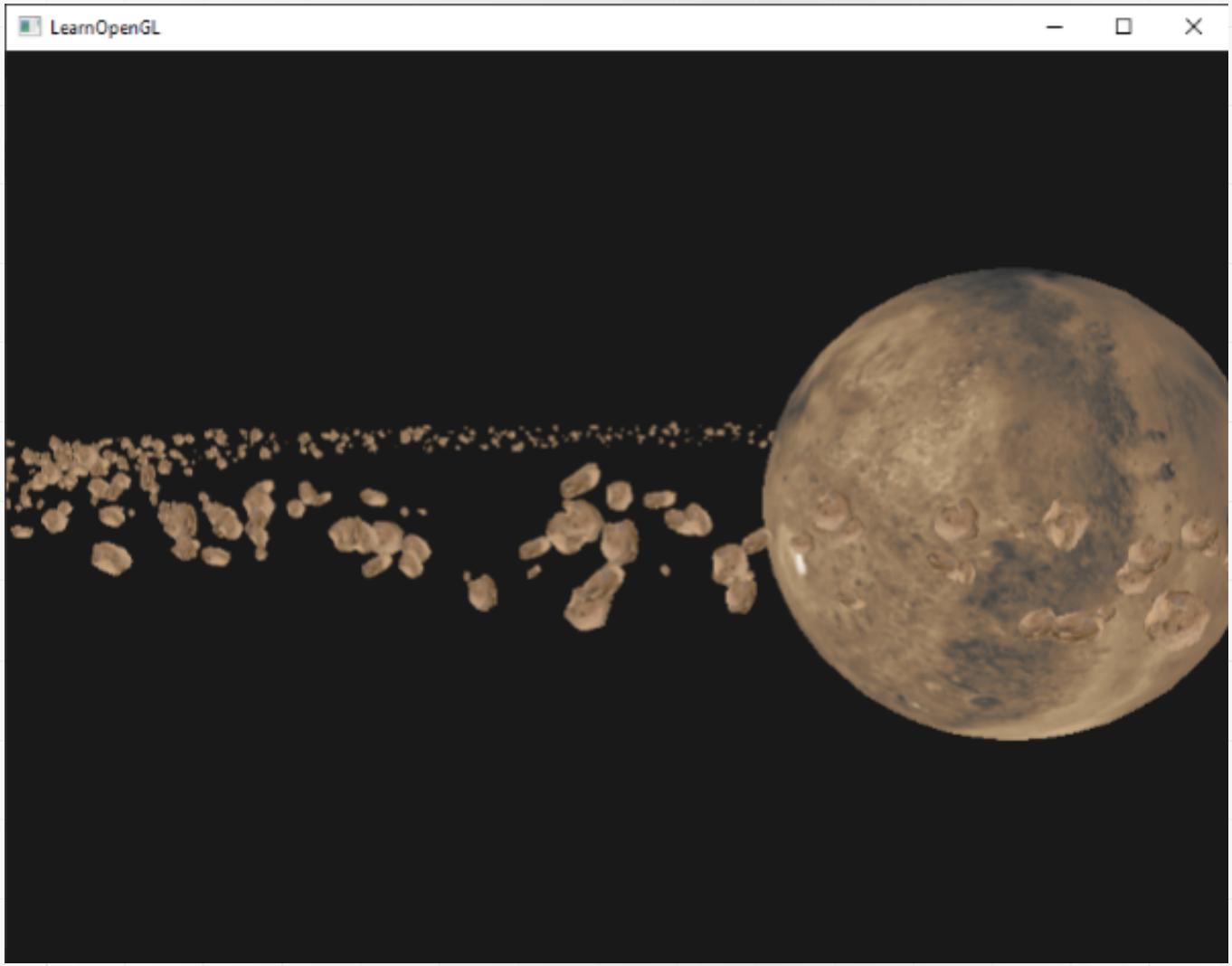
```
glDrawArrays(GL_TRIANGLES, 0, 36)
  ->
glDrawArraysInstanced(GL_TRIANGLES, 0, 36, 10)
            ^^^^^^^^^                   ^^^^
```

- Nowa zmienna `gl_InstanceID`, która numeruje instancje rysowanego obiektu, podobnie jak `gl_VertexID` numeruje instancje shadera.
- Transformację należy uwzględnić bezpośrednio w kodzie shadera,

```
gl_Position = P_matrix * V_matrix * M_matrix * position;
  ->
gl_Position = P_matrix * V_matrix * M_matrix * (
    positon + gl_InstanceID * vec4(1, 0, 0, 0)
);
```

Mechanizm rysowania instancjowego – przykład

– Ostatecznie tworzone obiekty nie muszą być całkowicie identyczne!



Źródło obrazka: https://learnopengl.com/Advanced-OpenGL/Instancing

Pozostałe nowości w przykładowym programie

- Konieczne jest zainstalowanie w systemie dwóch dodatkowych modułów.
  - `pip3 install --user numpy PyGLM`
  - Uwaga! Istnieje jeszcze inna biblioteka, o nazwie **glm**, która dostarcza zupełnie inne funkcje. Na potrzeby naszego kursu konieczna jest **PyGLM** (choć w kodzie obie działają jako `import glm`).
- Dodano funkcję `compile_shaders()`.
  - Zawiera ona kody źródłowe shadera wierzchołków i fragmentów, które są zapisane jako łańcuchy znaków.
  - Kody shaderów są kompilowane na karcie graficznej, a następnie scalane do obiektu programu, który można uruchomić na GPU.
  - Powyższy proces realizowany jest za każdym razem na nowo przez uruchomiony przez nas program w języku Python!
  - Funkcja zwraca obiekt programu do użycia w funkcji `render()`.
- W funkcji `main()` ustawiono minimalną wersję kontekstu OpenGL.

```
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE)
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3)
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3)
# Poniższą linijkę odkomentować w przypadku pracy w systemie macOS!
# glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE)
```

Politechnika Wrocławska

Koniec wprowadzenia.

Zadania do wykonania...

Zadania do wykonania (1)

Na ocenę **3.0** należy nacieszyć oko przykładowym programem i zmienić go.

Wskazówki:

- dla potwierdzenia, proszę zmienić tylko kolor wyświetlanego sześcianu,
- utworzyć nową zmienną **wyjściową** w shaderze wierzchołków,
  - `out vec4 vertex_color;`
- w funkcji `main()` shadera wierzchołków nadać jej wartość, np.
  - `vertex_color = vec4(0.2, 0.9, 0.1, 1.0);`
- utworzyć nową zmienną **wejściową** w shaderze fragmentów,
  - `in vec4 vertex_color;`
  - nazwa musi się pokrywać z wyjściem shadera wierzchołków (!),
- w funkcji `main()` shadera fragmentów przypisać przekazaną wartość,
  - `color = vertex_color;`
- zainstalować potrzebne biblioteki, jeśli ich brakuje,
  - `pip3 install --user numpy PyGLM`

Zadania do wykonania (2)

(po zrealizowaniu zadania poprzedniego)

Na ocenę **3.5** należy zmodyfikować kolory bryły w przykładowym programie.

Wskazówki:

- celem jest, aby każdy bok przykładowego sześcianu miał inny kolor,
- konieczne będzie zadeklarowanie dodatkowej zmiennej wejściowej (`in`) w shaderze wierzchołków oraz przekazanie jej do shadera fragmentów,
- zadanie można rozwiązać na dwa sposoby; konieczne będzie:
  - rozszerzenie tablicy `vertex_positions` o wartości kolorów, lub
  - zdefiniowanie nowej tablicy z kolorami oraz bufora danych;
- jeśli wybrano wariant z rozszerzeniem tablicy `vertex_positions`:
  - zmodyfikować pierwsze wywołanie `glVertexAttribPointer()`, aby uwzględnić przesunięcie kolejnych informacji w tablicy z danymi;
- przekazać drugą tablicę na wejście shadera wierzchołków, dodając nowe wywołania `glVertexAttribPointer()` oraz `glEnableVertexAttribArray()`.

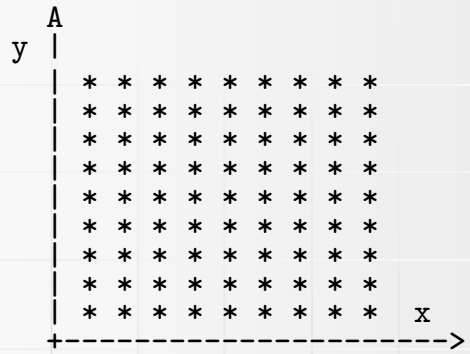
Zadania do wykonania (3)

(po zrealizowaniu zadania poprzedniego)

Na ocenę **4.0** należy stworzyć wiele kopii obiektu (klasycznie, na CPU).

Wskazówki:

- Należy utworzyć planszę, złożoną z wielu instancji, np. 10 x 10.
- Obiekty można rozmieścić, stosując translację wzdłuż osi X i Y.
- Cała zmiana powinna ograniczyć się wyłącznie do funkcji `render()`,
  - slajd 19 zawiera prawie wszystkie niezbędne elementy.
- Konieczne może być oddalenie kamery (`V_matrix`), aby zobaczyć efekt.
- Poglądowy widok na tworzoną scenę:



Zadania do wykonania (4)

(po zrealizowaniu zadania poprzedniego)

Na ocenę **4.5** należy wykorzystać mechanizm renderowania instancyjnego.

Wskazówki:

- Celem jest uzyskanie takiego samego efektu, jak w poprzednim zadaniu.
  - Tym razem implementacja będzie znacznie wydajniejsza.
  - Większość zmian obejmie kod shadera wierzchołków.
- Aby przetransformować cały obiekt, należy uzależnić transformacje wierzchołków tego obiektu od zmiennej `gl_InstanceID`.
  - Wszystkie wierzchołki danego obiektu należy poddać dokładnie temu samemu przekształceniu, aby obiekt zachował spójność.
  - Przydatna będzie funkcja moduł (operator `%`).
- Można zaimplementować funkcję, realizującą odpowiednią transformację.

Zadania do wykonania (5)

(po zrealizowaniu zadania poprzedniego)

Na ocenę **5.0** należy wprowadzić dodatkowe deformacje każdego obiektu.

Wskazówki:

- Deformacje zrealizować na poziomie shadera wierzchołków.
- Warto wykorzystać wykorzystać funkcje pseudolosowe do transformacji.
- W języku GLSL nie ma standardowo dostępnej funkcji `rand()` – należy ją zdefiniować jako dowolny ze znanych generatorów pseudolosowych.
- Przykładowe funkcje: https://en.wikipedia.org/wiki/List_of_random_number_generators.
- Uzależnić transformacje od zmiennej wbudowanej `gl_VertexID`.
- Uwzględnić także `gl_InstanceID`, aby każdy obiekt deformować inaczej.