<table>
<tr><td>Politechnika Wrocławska</td><td>Autor: Piotr Józefek 272311</td><td>Wydział Informatyki i Telekomunikacji
Rok akademicki: 3</td></tr>
<tr><td colspan="3">Grafika komputerowa i komunikacja człowiek – komputer</td></tr>
<tr><td>Data ćwiczenia:
21.01.2024</td><td rowspan="2">Temat ćwiczenia laboratoryjnego:
Potok graficzny oparty o shadery</td><td>Ocena:</td></tr>
<tr><td>Nr ćwiczenia:
7</td><td>Prowadzący:
dr inż. arch. Tomasz Zamojski</td></tr>
</table>

1. Wstęp

Współczesna grafika komputerowa, stojąc w obliczu potrzeby efektywnego renderowania coraz bardziej złożonych scen, dąży do zrównoleglenia procesów i minimalizacji narzutu sterowników. Zmiana paradygmatu programowania w tym obszarze, choć podnosząca poprzeczkę złożoności implementacji, otwiera drzwi do większej wydajności i możliwości. Współczesny potok graficzny OpenGL, składający się z etapów takich jak pobieranie wierzchołków, shadery (wierzchołków, teselacji, geometrii, fragmentów), teselacja, rasteryzacja i operacje na buforze ramki, oferuje programistom szerokie możliwości manipulacji renderingiem. Shadery, programy uruchamiane na GPU, pozwalają na równoległe przetwarzanie danych i są zazwyczaj tworzone w języku GLSL. Wyróżnia się różne typy shaderów, z których każdy odpowiada za inny etap potoku graficznego. GLSL, język programowania shaderów, dostosowany do potrzeb grafiki, oferuje wbudowane typy danych wektorowych i macierzowych oraz funkcje matematyczne. Dane do shaderów przekazywane są za pomocą buforów i tekstur, a Vertex Array Object (VAO) służy do zarządzania atrybutami wierzchołków. Dodatkowo, dane typu uniform pozwalają na przekazywanie wartości do shaderów. Tworzenie buforów obejmuje definicję identyfikatora bufora, alokację pamięci na GPU i ewentualne zwolnienie zasobów. Dane do bufora wprowadzane są po jego bindowaniu i określeniu przeznaczenia. Przekazywanie danych do shadera wierzchołków odbywa się za pomocą funkcji glVertexAttribPointer i glEnableVertexAttribArray. W shaderze wierzchołków definiuje się zmienne wejściowe i uniform, które następnie są uzupełniane danymi z poziomu aplikacji. Wyświetlanie wielu obiektów może być realizowane poprzez wielokrotne wywołanie funkcji rysującej z odpowiednimi transformacjami lub za pomocą mechanizmu instancjonowania, który pozwala na efektywne generowanie wielu kopii obiektu. Współczesne programy graficzne często korzystają z bibliotek takich jak NumPy i PyGLM oraz implementują funkcje do kompilacji shaderów w czasie rzeczywistym, co zwiększa elastyczność i możliwości programisty.

2. Cel ćwiczenia

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

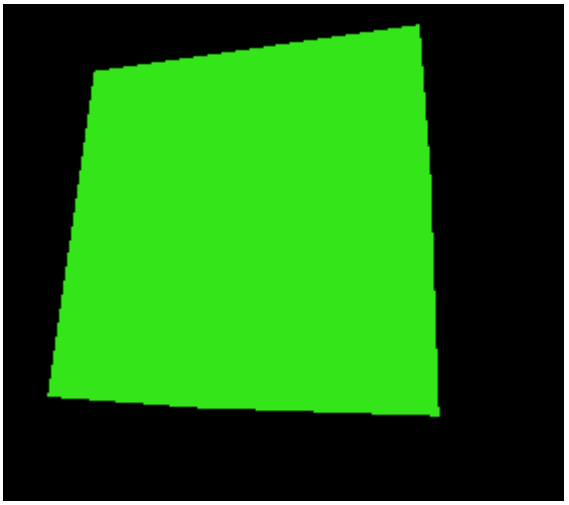
3. Wykonane zadania

Udało się zrealizować wszystkie zadania.

4. Prezentacja i omówienie funkcjonalności

4.1 Wyswietlenie szescianu za pomocą shaderów

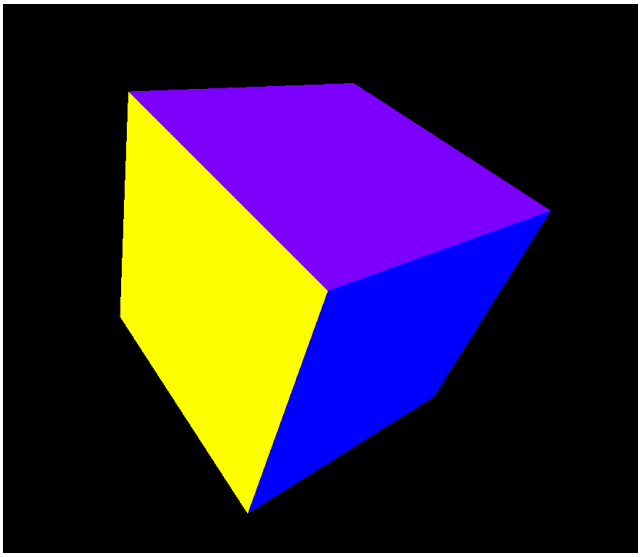
Celem zadania było zdefiniowanie i skompilowanie shaderów wierzchołków i fragmentów, a następnie przekazanie danych wierzchołkowych do potoku graficznego OpenGL. Funkcja compile_shaders() zawiera kod źródłowy shadera wierzchołków i fragmentów. Shader wierzchołków (vertex_shader_source) transformuje pozycje wierzchołków, mnożąc je przez macierze Modelu (M_matrix), Widoku (V_matrix) i Projekcji (P_matrix). Dodatkowo, shader wierzchołków ustawia kolor wierzchołka na zielony. Shader fragmentów (fragment_shader_source) odbiera kolor wierzchołka przekazany z shadera wierzchołków i ustawia go jako kolor fragmentu, co w efekcie powoduje, że obiekt jest renderowany na zielono. Oba shadery są kompilowane, a następnie łączone w jeden program shaderowy, który jest zwracany przez funkcję. Funkcja startup() przeprowadza inicjalizację OpenGL, w tym pobiera informacje o wersji OpenGL i GLSL. Następnie, ustawia viewport, włącza test głębi i wywołuje funkcję compile_shaders() w celu utworzenia i skompilowania programu shaderowego. Kolejnym krokiem jest wygenerowanie i związanie Vertex Array Object (VAO), który służy do przechowywania informacji o atrybutach wierzchołków. Dane wierzchołkowe, reprezentowane przez tablicę vertex_positions, są przesyłane do bufora wierzchołków (VBO) i przechowywane na karcie graficznej. Na koniec, funkcja glVertexAttribPointer konfiguruje sposób interpretacji danych wierzchołkowych, a funkcja glEnableVertexAttribArray włącza przekazywanie danych z bufora do shadera wierzchołków.



Rys 1 Wyświetlenie renderowanego sześciany

4.2 Modyfikacja kolorów bryły

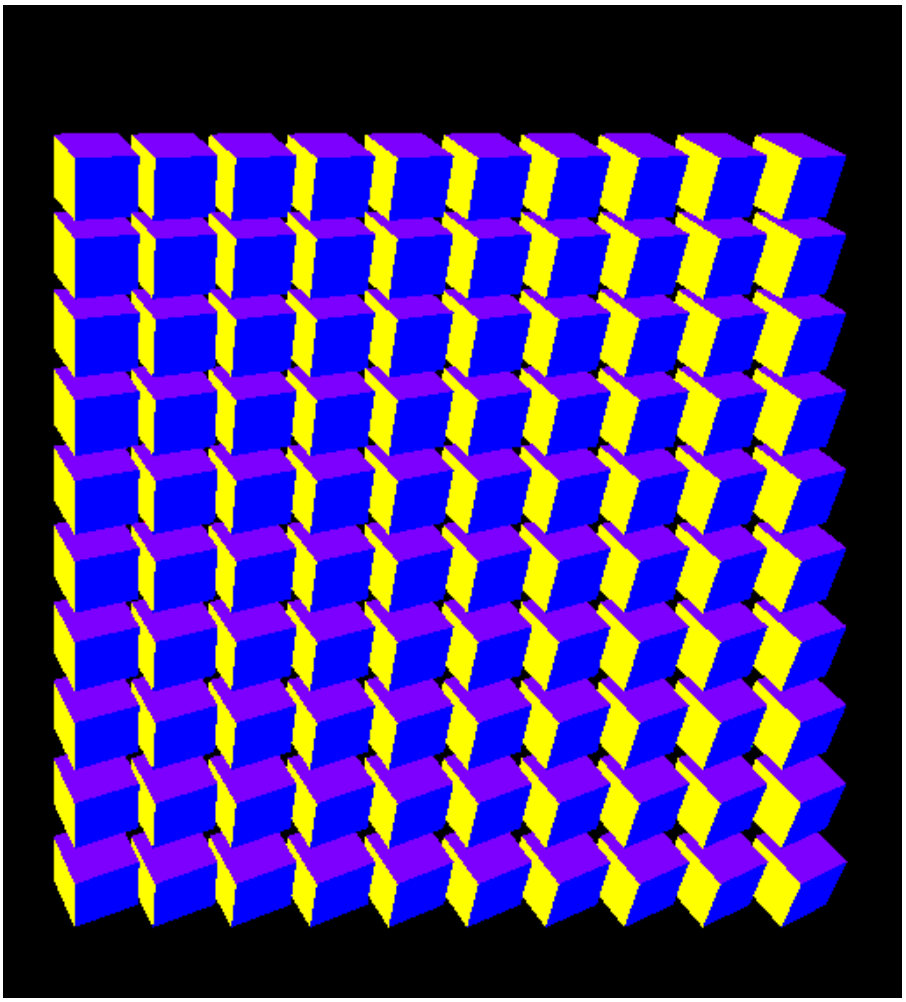
Celem tego kodu jest narysowanie kolorowego sześcianu 3D za pomocą OpenGL i shaderów. Kod został zmodyfikowany w celu dodania obsługi kolorów wierzchołków, gdzie każdy wierzchołek może mieć przypisany inny kolor. Funkcja compile_shaders() definiuje i kompiluje shadery. W shaderze wierzchołków, do atrybutu position (lokalizacja 0) i color (lokalizacja 1) są pobierane dane o pozycji i kolorze wierzchołka. Pozycja wierzchołka jest transformowana przez macierze Modelu, Widoku i Projekcji, a kolor wierzchołka jest przekazywany do shadera fragmentów. Shader fragmentów odbiera kolor wierzchołka i ustawia go jako kolor fragmentu, dodając kanał alfa z wartością 1.0 (pełna widoczność). Funkcja ta tworzy, kompiluje i linkuje program shaderowy, a następnie zwraca jego identyfikator. Funkcja startup() inicjalizuje OpenGL. Tworzy i kompiluje program shaderowy za pomocą funkcji compile_shaders(). Następnie generuje i wiąże Vertex Array Object (VAO). Dane wierzchołkowe, wraz z danymi o kolorach wierzchołków, są przesyłane do buforów (VBO). Funkcja glVertexAttribPointer konfiguruje sposób interpretacji danych pozycji (atrybut 0) i kolorów (atrybut 1) wierzchołków. Włączane jest również przekazywanie danych z buforów do shaderów za pomocą funkcji glEnableVertexAttribArray. Dodatkowo, włączany jest test głębi (glEnable(GL_DEPTH_TEST)) i ustawiana funkcja testu głębi na GL_LESS, co zapewnia, że obiekty są rysowane w poprawnej kolejności, uwzględniając ich odległość od kamery. Każda ściana sześcianu ma przypisany inny kolor, co jest widoczne po uruchomieniu programu.



Rys 2 Wyświetlenie renderowanego sześciany z różnokolorowymi ścianami

4.3 Wiele kopi obiektu

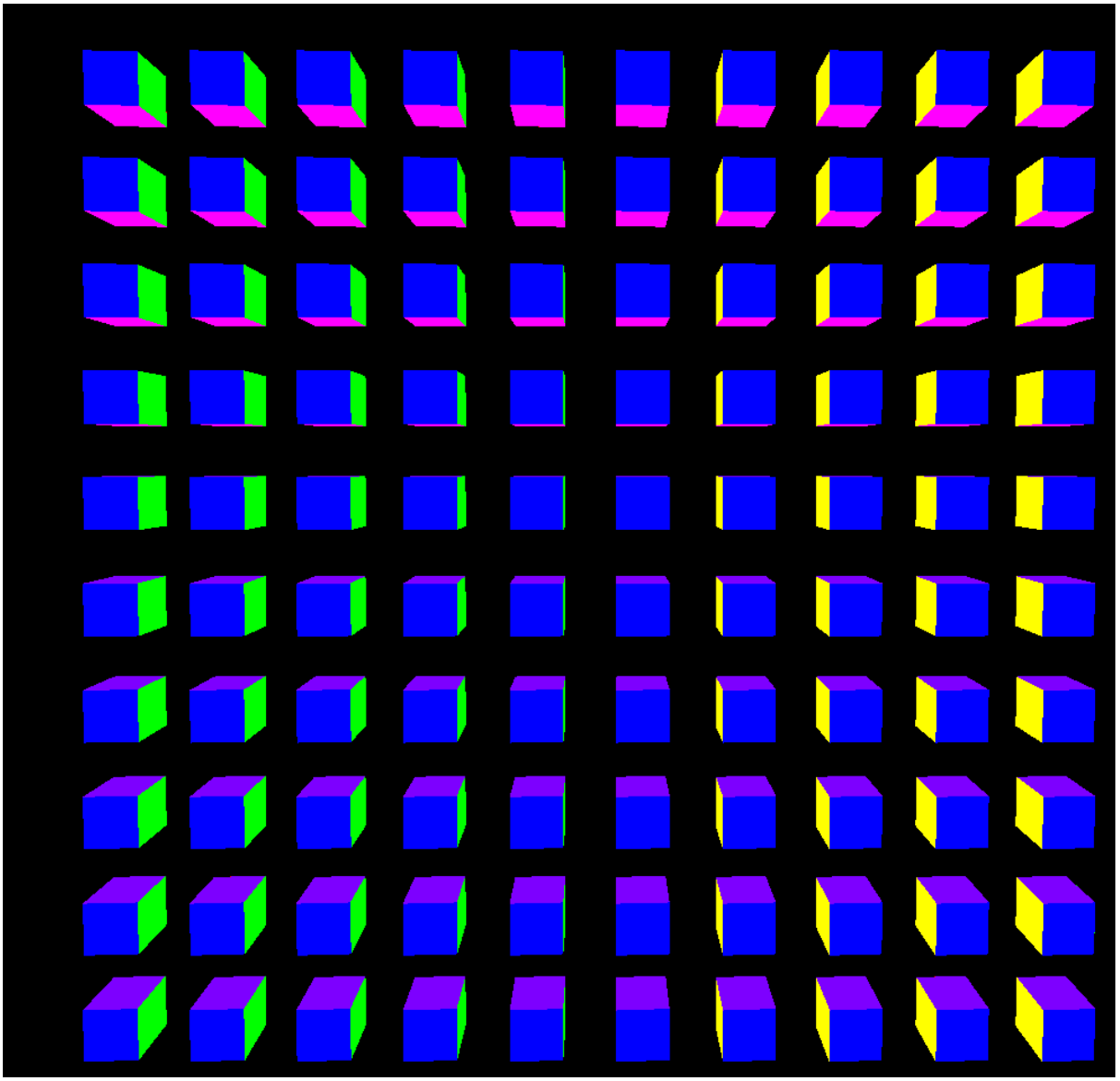
Celem zadania było stworzenie renderowanej siatki sześcianów 3D. Każdy sześcian jest obracany, a kamera jest ustawiona w określonej pozycji. Funkcja render(time) odpowiada za rysowanie sceny w każdej klatce. Na początku funkcja czyści bufor kolorów i bufor głębi (glClear). Następnie aktywuje program shaderowy (glUseProgram). Macierz widoku (V_matrix) jest tworzona za pomocą funkcji glm.lookAt(). Kamera jest umieszczona w punkcie (0, 0, 10), patrzy na punkt (0, 0, 0), a wektor "góry" jest skierowany w kierunku (0, 1, 0). Macierze projekcji (P_matrix) i widoku (V_matrix) są przekazywane do shadera za pomocą funkcji glUniformMatrix4fv(). Pętla iteruje po siatce 10x10. Wewnątrz pętli tworzona jest macierz modelu (M_matrix). Macierz ta najpierw zawiera transformację translacji, a potem rotacji. Każdy sześcian jest przesuwany o wektor (i * spacing, j * spacing, 0), gdzie i i j to indeksy w siatce, a spacing to odległość między sześcianami. Sześciany są obracane wokół osi (1, 1, 0) z prędkością proporcjonalną do czasu. Macierz modelu jest przekazywana do shadera. Funkcja glDrawArrays(GL_TRIANGLES, 0, 36) rysuje sześcian. Sześcian jest rysowany za pomocą 36 wierzchołków, które są połączone w trójkąty. Funkcja glFlush() przesyła polecenia rysowania do karty graficznej.



Rys 3 Wyświetlenie renderowanego sześciany z różnokolorowymi ścianami

4.4 Mechanizm renderowania instancyjnego

W funkcji render(time) w porównaniu z poprzednim zadaniem , kluczową zmianą jest użycie instancjonowania za pomocą glDrawArraysInstanced. Zamiast rysować każdy sześcian oddzielnie w pętli, jak to miało miejsce w zadaniu poprzednim , teraz funkcja glDrawArraysInstanced rysuje 100 instancji sześcianu (ostatni argument to 100) za pomocą jednego wywołania. To znacznie poprawia wydajność, szczególnie przy dużej liczbie obiektów. Dodatkowo, macierze widoku (V_matrix) i projekcji (P_matrix) są ustawiane tylko raz na początku funkcji render, a nie w pętli dla każdego sześcianu, co również optymalizuje rendering. Do shadera wierzchołków przekazywany jest również czas (time), co umożliwia animację obrotu. W shaderze wierzchołków (vertex_shader_source) wprowadzono kilka istotnych zmian. Dodano atrybut layout(location = 1) in vec4 color; do odbierania koloru dla każdego wierzchołka. Dodano również uniformy uniform mat4 V_matrix;, uniform mat4 P_matrix; i uniform float time; do odbierania macierzy widoku, projekcji i czasu. Kluczowa zmiana to obliczanie pozycji każdego sześcianu na podstawie gl_InstanceID. int x = gl_InstanceID % 10 - 5; i int y = gl_InstanceID / 10 - 5; obliczają pozycję x i y sześcianu w siatce 10x10. Macierz translacji translation jest tworzona na podstawie tych wartości. Dodatkowo, funkcja rotate oblicza macierz rotacji na podstawie czasu i osi obrotu. Macierz modelu M_matrix jest teraz wynikiem połączenia translacji i rotacji: mat4 M_matrix = translation * rotation;. Ostatecznie, pozycja wierzchołka jest obliczana jako gl_Position = P_matrix * V_matrix * M_matrix * position;. Kolor wierzchołka jest przekazywany do shadera fragmentów za pomocą vertex_color = color;. Shader fragmentów (fragment_shader_source) pozostaje bez zmian i jedynie przypisuje kolor wierzchołka do piksela.

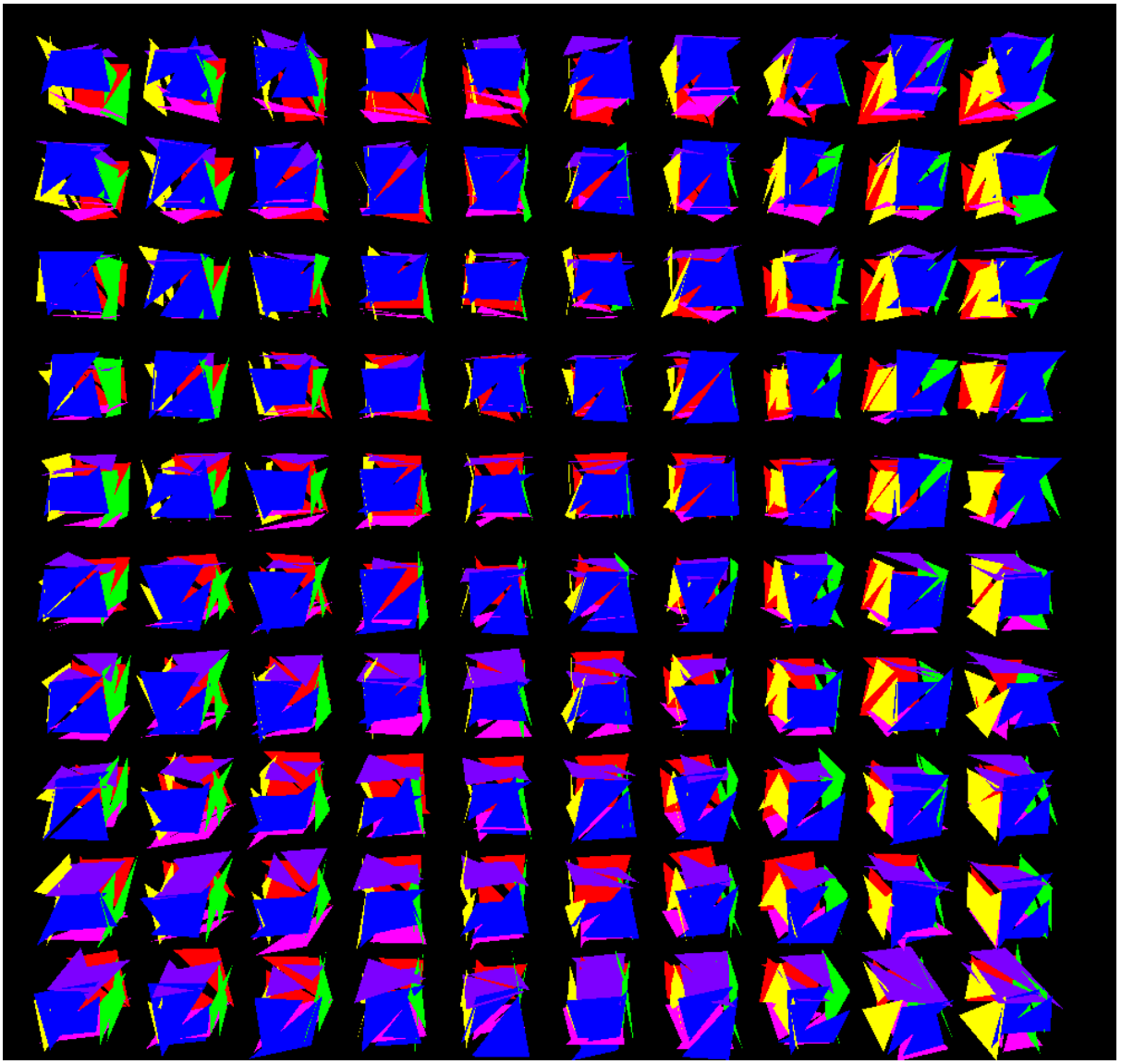


Rys 4 Wyświetlenie renderowania instancyjnego

4.5 Wprowadzenie dodatkowych deformacji każdego obiektu

Aby umożliwić deformację obiektów na poziomie wierzchołków, shader wierzchołków został zmodyfikowany poprzez dodanie generatora liczb pseudolosowych i algorytmu deformacji, który bazuje na identyfikatorach instancji i wierzchołków. Zaimplementowano funkcję xorshift jako generator liczb pseudolosowych. Funkcja ta, oparta na algorytmie Xorshift, generuje sekwencję liczb, które przybliżają wartości losowe. Ponieważ język GLSL nie ma wbudowanej funkcji rand(), konieczne było zaimplementowanie własnego generatora liczb pseudolosowych, aby móc generować wartości losowe na poziomie shadera. W funkcji main shadera wierzchołków, po obliczeniu macierzy modelu M_matrix, zaimplementowano algorytm deformacji. Algorytm ten wykorzystuje wbudowane zmienne gl_InstanceID (identyfikator instancji) oraz gl_VertexID (identyfikator wierzchołka), aby zapewnić unikalną deformację dla każdego wierzchołka

każdego obiektu. Wartości gl_InstanceID i gl_VertexID są konwertowane na typ uint (liczba całkowita bez znaku) i używane do wygenerowania stanu początkowego dla generatora liczb pseudolosowych. Stan ten jest obliczany poprzez pomnożenie gl_InstanceID przez stałą (w tym przypadku 100) i dodanie gl_VertexID. Takie podejście gwarantuje, że każdy wierzchołek każdej instancji obiektu będzie miał unikalny stan początkowy, co przekłada się na unikalną sekwencję liczb pseudolosowych generowanych dla każdego wierzchołka. Na podstawie wygenerowanego stanu funkcja xorshift jest wywoływana trzykrotnie, aby uzyskać trzy wartości pseudolosowe, które są następnie normalizowane do zakresu od -0.1 do 0.1. Wartości te reprezentują przesunięcia w osiach x, y i z, które będą dodawane do pozycji wierzchołka. Do pozycji każdego wierzchołka, pobranej z atrybutu position, dodawane są wygenerowane przesunięcia, tworząc zdeformowaną pozycję deformedPosition. Ostatecznie, pozycja wierzchołka gl_Position jest obliczana na podstawie macierzy projekcji (P_matrix), macierzy widoku (V_matrix), macierzy modelu (M_matrix) oraz zdeformowanej pozycji wierzchołka (deformedPosition). Kolor wierzchołka, pobrany z atrybutu color, jest przekazywany do shadera fragmentów. W wyniku tych zmian każdy sześcian jest renderowany z unikalnymi deformacjami, bazując na identyfikatorze instancji i wierzchołka, co prowadzi do wizualnie zróżnicowanego renderowania.



Rys 4 Wyświetlenie renderowania instancyjnego z deformacjami

5. Wnioski

- Udało się zrealizować wszystkie zadania

6. Literatura

- Szymon Datko [GK] Lab7 :: Potok graficzny oparty o shadery