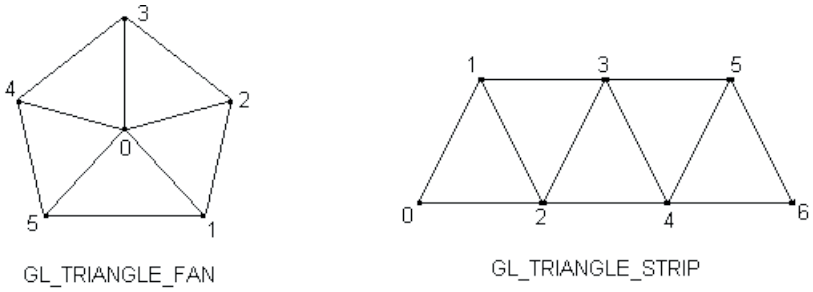
1 Podstawy programowania shaderów

Celem niniejszych ćwiczeń jest przybliżenie tematyki związanej z przetwarzaniem fragmentów i wierzchołków z wykorzystaniem programów cieniujących (ang. shaders). Podczas ćwiczeń przestawiony zostanie sposób renderowania czworokątów, który odróżnia się od dostępnych we wcześniejszych wersjach OpenGL. Ponadto omówione zostaną podstawy działania shaderów oraz wykorzystanie zmiennych jednorodnych.

1.1 Omne trinum prefectum

Zanim przystąpimy do prowadzenia badań nad mechanizmem shaderów, warto przytoczyć stare łacińskie przysłowie [1], wychwalające liczbę trzy. Poza aspektami filozoficznymi i numerologicznymi trójka wyznacza optymalne wymogi konstrukcyjne w wielu zastosowaniach technicznych. Kratownice i belkowania w architekturze są składane z trójkątów, gdyż sprzyjają one rozkładaniu naprężeń na całą długość elementu zamiast skupiać je punktowo. Trójstanowa logika jest podstawą działania współczesnych, elektronicznych systemów cyfrowych. Trójkąty używa się w grafice komputerowej jako podstawowe elementy w renderowaniu obiektów między innymi dlatego, że każdy wielokąt może być zdekomponowany na zbiór przystających trójkątów. Trójkąt jest najmniejszym niepodzielnym dalej prymitywem ograniczającym daną powierzchnię. W związku z tym większość współczesnych silników gier, oraz szerzej narzędzi sprzyjających komputerowemu renderowaniu grafiki opiera się na przetwarzaniu trójkątów. W OpenGL w wersji 3.3. rezygnowano z obsługi prymitywów typu QUAD (czworokąt), oraz polygon (Wielokąt wypukły) wymuszając na programiście opisanie geometrii renderowanych powierzchni wyłącznie poprzez zastosowanie trójkątów. Do tworzenia powierzchni złożonych z przystających do siebie trójkątów utrzymano obsługę prymitywów `GL_TRIANGLE_FAN` oraz `GL_TRIANGLE_STRIP`. Interpretacje graficzną w/w obiektów prezentuje rys. 1.



Rysunek 1: Prymitywy zapewniające renderowanie przystających trójkątów

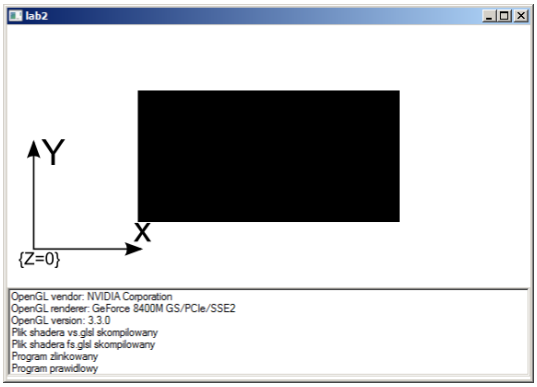
Wierzchołki prymitywu oznaczono kolejnymi liczbami naturalnymi, które reprezentują porządek zdefiniowania ich w tablicach atrybutów wierzchołków. Pierwsze trzy wierzchołki $0, 1, 2$ wyznaczają trójkąt startowy, każdy następny $n - ty$ wierzchołek powoduje doklejenie do obiektu trójkąta o bokach rozpiętych między $n, n-1, n-2$. Prymitywy `GL_TRIANGLE_FAN` tworzą obiekty o symetrii środkowej i sprzyjają generowaniu np. wielokątów foremnych. Prymitywy `GL_TRIANGLE_STRIP` pozwalają natomiast budować obiekty o charakterze wstęgowym.

1.1.1 Przebieg ćwiczenia

1. Przygotuj nowy projekt programistyczny i podłącz do niego pliki zawarte w materiałach do bieżących ćwiczeń

[1] dosł. wszystko, co złożone z trzech jest doskonałe

2. Uzupełnij kod metody `PrepareObjects`, aby zapewnić rysowanie czworokąta leżącego w płaszczyźnie XY dla $Z = 0$, rozpiętego między punktami $(-0.5, -0.5, 0.0)$ a $(0.5, 0.5, 0.0)$ z użyciem maksymalnie 4 zdefiniowanych wierzchołków. Przykład realizacji zadania prezentuje rys. 2



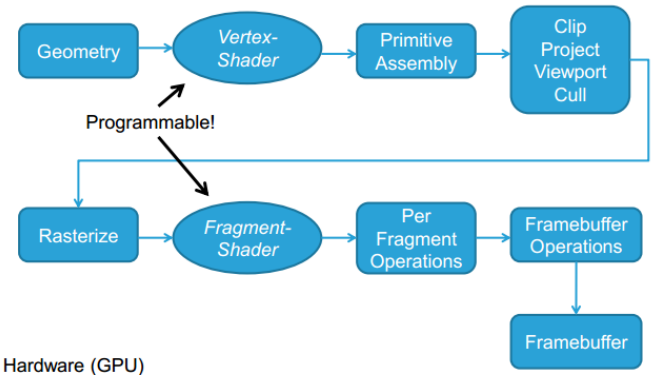
Rysunek 2: Realizacja czworokata.

1.2 Idea shaderów

Shader jest niedużym programem, który może być uruchamiany bezpośrednio na procesorze karty graficznej. Zadaniem tego programu jest wykonanie określonej operacji na wierzchołkach dostarczanych przez aplikację. Operacja ta może wiązać się z pobraniem i przetworzeniem atrybutów wierzchołka oraz przesłaniem ich dalej do kolejnych bloków w potoku renderowania grafiki OpenGL. Ze względu na charakter działań wyróżniamy trzy podstawowe typy shaderów:

- wierzchołków - przetwarzają tylko i wyłącznie atrybuty wierzchołka, w najprostszym ujęciu wykonują przepisanie atrybutu pozycji wierzchołka
- fragmentów - przetwarzają obszary ograniczone zbiorami wierzchołków nadając im np. kolor, cieniowanie, teksturę
- geometrii - pozwalają wykonać przekształcenie geometryczne zapewniające np. obrót całej grupy wierzchołków, lub przekształcenie używanego układu odniesienia

Odrębną grupą shaderów są shadery tesselacji, które wywodzą się z wprowadzonych uprzednio w OpenGL 2.0 obiektów tesselatorów. Tesselacja wielokątów polega na podziale obszaru wielokąta ograniczonego zadanym zbiorem wierzchołków na zbiór przystających do siebie trójkątów. Programy shaderów są wyrażane jako pliki tekstowe opisane w składni języka *OpenGL Shading Language* (GLSL). Od wersji 3.3. OpenGL obowiązują skojarzone z nim rewizje wersji GLSL. Oznacza to, że dla karty graficznej wspierającej renderowanie OpenGL na poziomie wersji 3.3. możliwe jest używanie shaderów w wersji GLSL 3.3. Nie wszystkie elementy składniowe języka GLSL są kompatybilne w przód. Kompatybilność wsteczna jest jednakże zachowywana. Składnia języka shaderów jest oparta na składni języka C++. Szczegóły omawia specyfikacja języka GLSL w określonej wersji. Pliki shaderów podlegają kompilacji i linkowaniu. Możliwe jest przeprowadzenie tych operacji jednorazowo, po starcie aplikacji używającej OpenGL. Obecnie spróbujemy wykonać kilka eksperymentów z shaderami włączonymi w aplikację. Na rysunku 3 zaprezentowano rozwinięcie potoku renderowania OpenGL z uwzględnieniem kolejności wykonania shaderów.



Rysunek 3: Potok renderowania OpenGL z uwzględnieniem kolejności wykonania shaderów.

1.2.1 Przebieg ćwiczenia

1. Zapoznaj się ze sposobem implementacji sposobu włączania shaderów w potok przetwarzania grafiki. Służą temu metody `PreparePrograms`, która buduje obiekt programu. W programie zawiera się shadery przetwarzające przy pomocy funkcji `glAttachShader`. Po załadowaniu obiektów shaderów program jest linkowany `glLinkProgram()` i jest sprawdzana jego poprawność (m. in. kompatybilność z kartą graficzną) `glValidateProgram`. Dla każdego z tych etapów możliwe jest pobranie statusu operacji oraz pobranie logu z komunikatami o ewentualnych błędach przetwarzania. Błędy te są prezentowane użytkownikowi z użyciem funkcji `PrintLog()`. Tworzenie i linkowanie obiektu programu jest rozwiązane w sposób typowy dla tego typu aplikacji. Warto zwrócić uwagę na fakt, że `PreparePrograms()` ładuje shadery z zewnętrznych plików. Tworzenie shaderów jest zaimplementowane w funkcji `LoadShader`, która tworzy obiekt shadera określonego typu, podłącza do niego treść kodu źródłowego pozyskiwanego z pliku. Po załadowaniu treści pliku, kod shadera jest kompilowany `glCompileShader()` a informacje o statusie operacji kompilowania są prezentowane w oknie loga.

2. Zapoznaj się z treścią shadera wierzchołków `vs.glsl`. W bieżącej implementacji pozwala ona na przepisanie atrybutu 0 wierzchołka z przychodzącego z tablicy VOA do wbudowanej w potok OpenGL zmiennej `gl_Position`. Warto zwrócić uwagę, że atrybuty położenia są przechowywane jako wektory $[x, y, z, 1.0]$ Rozszerzanie wektora pozycji do postaci 4-o elementowej znajdzie swoje wytłumaczenie przy omawianiu macierzowych przekształceń geometrycznych

3. Zapoznaj się z treścią shadera fragmentów `fs.glsl` Na bieżącym etapie kursu będziemy przyjmowali, że jedyną funkcją shadera fragmentów, jest pokolorowanie fragmentu prymitywu. W związku z tym zadaniem shadera jest zwrócenie wartości wektora, zawierającego trójkę RGB koloru. Tutaj również następuje wyskalowanie wektora do postaci 4-o elementowej RGBA. Znaczenie ostatniego parametru (A) zostanie omówione później.

4. Zmień wartości elementów wektora `outputColor` w shaderze fragmentów. Zobacz jak wpłynie to na barwę renderowanego obiektu.

Obecnie zmodyfikujemy shadery tak, aby kolorowały obiekt przy pomocy atrybutów koloru skojarzonego z każdym z wierzchołków. W tym celu należy wprowadzić nowe wartości atrubutów do VAO opisującego obiekt, oraz zapewnić przypisywanie tych atrybutów przez shadery.

1.2.2 Przebieg ćwiczenia

1. W sekcji atrybutów prywatnych klasy Scena (plik `scene.h` zmodyfikuj zmienne opisujące liczbę tworzonych buforów VBOs. Potrzebne będą dwa bufory zawierające wartości dwóch rodzajów atrybutów (pozycja i kolor).

```
#define VAO_cnt 1 // liczba tablic wierzcholkow obiektow (= liczba obiektow)
#define VBO_cnt 2 // liczba buforow skojarzonych z VAOs
GLuint VAOs[VAO_cnt];
GLuint VBOs[VBO_cnt];
```

2. Rozbuduj kod funkcji `PrepareObjects` o tworzenie i ładowanie atrybutu koloru wierzchołka do VAO

```
float col_tab[12]; // tablica kolorow (4 wierzcholki)
	// wypelnij kolorami (r,g,b)
	col_tab[0] = 1.0f; col_tab[1] = 0.0f; col_tab[2] = 0.0f; // czerwony
	col_tab[3] = 0.0f; col_tab[4] = 1.0f; col_tab[5] = 0.0f; // zielony
	col_tab[6] = 0.0f; col_tab[7] = 0.0f; col_tab[8] = 1.0f; // niebieski
	col_tab[9] = 1.0f; col_tab[10] = 1.0f; col_tab[11] = 0.0f; // zolty

	// podlacz pierwszy bufor VBOs
	glBindBuffer(GL_ARRAY_BUFFER, VBOs[1]);
	// wypelnij bufor wspolrzednymi wierzcholka
	glBufferData(GL_ARRAY_BUFFER, sizeof(col_tab), col_tab, GL_STATIC_DRAW);
	// wybierz atrybut indeksie 1 (wskazany w shaderze)
	glEnableVertexAttribArray(1);
	// powiaz dane z bufora ze wskazanym atrybutem
	glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);
```

3. Rozbuduj kod shadera wierzchołków do postaci z listingu

```
// definicja wersji
#version 330
// zmienna wejsciowa inPosition zbierana z VAO jako atrybut 0
layout (location = 0) in vec3 inPosition;
// zmienna wejsciowa inColor zbierania z VAO jako atrybut 1
layout (location = 1) in vec3 inColor;
// zmienna wyjsciowa do shadera fragmentow
out vec3 kolorek;
void main()
{
	// przypisz pozycje do zmiennej wbudowanej OpenGL
	gl_Position = vec4(inPosition, 1.0);
	// przepisz kolor na wyjscie z shadera
	kolorek = inColor;
}
```

4. Rozbuduj kod shadera fragmentów do postaci z listingu. Zauważ, że przekazanie wartości atrybutu kolor zachodzi za pośrednictwem NAZWY zmiennej wyjściowej dla shadera wierzchołków a wejściowej dla shadera fragmetów. W omawianych przykładach jest to zmienna `kolorek`

```
#version 330

in vec3 kolorek; // zmienna wejsciowa
out vec4 outputColor; // zmienna wyjsciowa

void main()
{
       outputColor = vec4(kolorek, 1.0);
}
```

5. Skompiluj i uruchom program. Zwróć uwagę, że zmiana koloru wierzchołka w trakcie renderowania prymitywu skutkuje płynnym cieniowaniem fragmentów.

1.3 Zmienne jednorodne

Dotychczas omówione przykłady ukazują OpenGL jako narzędzie, które może wyrenderować poprawną grafikę pod warunkiem dostarczenia jej od razu mnóstwa danych opisujących wierzchołki i ich atrybuty. Tymczasem poprzez zastosowanie programów shaderów możliwa jest redukcja ilości atrybutów przekazywanych na wejście potoku przetwarzania. Język GLSL oferuje szereg wbudowanych funkcji matematycznych, które pozwalają np. w sposób zautomatyzowany naliczać współrzędne kolorów wierzchołków. Istnieje zatem możliwość uzyskiwania efektu kolorowania obiektu w zależności od jego położenia. To jest preludium do całej klasy algorytmów zapewniających realistyczną zmianę cieniowania przemieszczanego obiektu. Warto podkreślić, że modyfikacje zapisane w programie shadera wpływają na cały obiekt, który ten shader przetwarza. A z drugiej strony wykonanie shadera może być skalowane na przetwarzanie wielordzeniowe. Daje to zatem szerokie możliwości adaptacji efektu graficznego do potrzeb. Obecnie omówimy technikę przekazywania danych do shaderów z aplikacji z modyfikacją atrybutów tablic wierzchołków. Jest to technika oparta o tzw. zmienne jednorodne (ang. Uniform Variables) Mianowicie istnieje możliwość zapisania wartości z poziomu aplikacji OpenGL do zmiennej istniejącej w shaderze. Tą technikę zilustrujemy przykładem, który pozwoli przemieszczać wyrenderowany uprzednio czworokąt z użyciem klawiszy strzałkowych klawiatury.

1.3.1 Przebieg ćwiczenia

Ćwiczenie zostanie zrealizowane poprzez wprowadzenie dodatkowych atrybutów prywatnych klasy scena, które reprezentować będą względne przesunięcie renderowanego obiektu. Wartości tych atrybutów `dX` oraz `dY` będą modyfikowane w odpowiedzi na komunikat systemowy związany z naciskaniem klawiszy klawiatury. Na koniec wartości atrybutów zostaną przekazane go shadera wierzchołków, aby ten zmodyfikował otrzymywane wartości pozycji obiektu o przyrosty `dX` oraz `dY`

1. Wprowadź definicje atrybutów prywatnych do klasy scena (plik `scene.h`)

```
private:

float dX; // przesuniecie obiektu po X
float dY; // przesuniecie obiektu po Y
```

2. W pliku `scene.cpp` odszukaj definicję konstruktora klasy zapewnij w nim wyzerowanie wartości zmiennych `dX` oraz `dY`

3. W pliku `main.cpp` odszukaj kod obsługi kolejki komunikatów Windows. Rozbuduj kod obsługujący komunikat `WM_KEYDOWN`

```
            case WM_KEYDOWN: // nacisniecie klawisza
            {
                    POINT cPos;
                    GetCursorPos(&cPos);
                    SC->KeyPressed(wParam,cPos.x,cPos.y);
                    SC->Draw();
                     SwapBuffers(hDC);
                    return 0;
            }
```

Kod ten powoduje wywołanie metody `KeyPressed` na instancji klasy Scene. Do wywołania metody przekazywany jest kod naciśniętego klawisza oraz współrzędne kursora myszy. W kolejnych linijkach wywołana jest metoda renderowania widoku sceny. Zapewni to aktualizację widoku sceny po każdorazowym naciśnięciu klawiszy.

4. Odszukaj definicję metody `KeyPressed` w pliku `scene.cpp`.

5. Przy pomocy procedury diagnostycznej `PrintLog` wyświetl wartości kodów klawiszy przekazywanych do metody

```
void Scene::KeyPressed(unsigned char key, int x, int y)
{
       if (key == ESCAPE) ThrowException("Zatrzymaj program");

       sprintf(_msg,"%d",key);
       PrintLog(_msg);
}
```

6. Rozbuduj kod metody `KeyPressed` aby zmieniała ona wartości zmiennych `dX` oraz `dY` o np. 0.1 po naciśnięciu klawiszy strzałkowych. Zapewnij ergonomiczne przypisanie funkcji klawiszy tzn. klawisz w górę powoduje zwiększenie dY o 0.1, klawisz w dół zmniejszenie o 0.1. Podobnie dla klawiszy prawo/lewo i dX

7. Na końcu metody `KeyPressed` wprowadź kod, który przekazuje zmienne `dX` oraz `dY` do obiektu programu tworzonego w OpenGL[2]

```
void Scene::KeyPressed(unsigned char key, int x, int y)
{
       if (key == ESCAPE) ThrowException("Zatrzymaj program");

       // ... tutaj kod obslugi zmiennych dX dy ...

       // pobierz polozenie zmiennej ze shadera pod dX_loc
       GLint dX_loc = glGetUniformLocation(program, "dX");
       // podstaw wartsc pod dX_loc (spowoduje nadpisanie zmiennej w shaderze)
       glUniform1f(dX_loc,dX);
       GLint dY_loc = glGetUniformLocation(program, "dY");
       glUniform1f(dY_loc,dY);
}
```

[2]Przypominamy, że program zawiera zbiór załączonych i skompilowanych shaderów

8. Rozbuduj kod shadera wierzchołków do postaci z listingu poniżej

```
// definicja wersji
#version 330
// zmienna wejsciowa inPosition zbierana z VAO jako atrybut 0
layout (location = 0) in vec3 inPosition;
// zmienna wejsciowa inColor zbierania z VAO jako atrybut 1
layout (location = 1) in vec3 inColor;
// zmienna wyjsciowa do shadera fragmentow
out vec3 kolorek;
uniform float dX;
uniform float dY;
void main()
{
	// przypisz pozycje do zmiennej wbudowanej OpenGL
	gl_Position = vec4(inPosition.x+dX,inPosition.y+dY,inPosition.z, 1.0);
	// przepisz kolor na wyjscie z shadera
	kolorek = inColor;
}
```

9. Skompiluj i uruchom program. Na tym etapie powinna istnieć możliwość sterowania położeniem czworokąta (opisanego poprzez 12 współrzędnych w VAO[0]) za pośrednictwem shadera i dwóch zmiennych.

1.4 Zadanie

Zapoznaj się ze specyfikacją GLSL dla wersji 3.3. lub wyższej. Zwróć uwagę na sposób definiowania zmiennych wektorowych i macierzowych.