6 Programowanie potoku renderującego

Po zapoznaniu się z fundamentalnymi elementami języka możemy przejść do bardziej konkretnych zagadnień. Najpierw zostanie przedstawiony podstawowy program zawierający wszystkie elementy potoku. Omówiony zostanie przepływ danych między etapami i wszystkie podstawowe elementy niezbędne do prawidłowego funkcjonowania programu. Następnie opisane zostaną wszystkie elementy charakterystyczne dla każdego z shaderów tak, aby wyjaśnić, czym każdy z nich dysponuje, czego potrzebuje do pracy i co jest w stanie osiągnąć, tym razem z bardziej praktycznej strony.

6.1. Przykładowy program zawierający wszystkie podstawowe shadery

Najprostsza implementacja każdego z shaderów w stylu klasycznego *"hello world"* wymaga odrobinę więcej pracy niż w przypadku języków programowania operujących na CPU. Nie jest to jednak zadanie specjalnie wymagające. Wystarczy kilkanaście linii kodu, aby uzyskać podstawowe rezultaty, jak wyświetlenie kolorowego modelu 3D złożonego z trójkątów. Dokładnie takie zadanie realizuje program pokazany na listingach 6.1–6.5. Pod każdym z nich można odnaleźć krótki opis powiązanego etapu, który jest bardziej wstępnym uświadomieniem zasad i mechanizmów występujących w każdym z etapów niż dogłębną analizą poszczególnych fragmentów kodu. Dokładny opis wszystkich etapów można natomiast odnaleźć w poświęconych im podrozdziałach.

Listing 6.1. Przykładowa implementacja shadera wierzchołków

```
#version 400
layout(location = 0) in vec3 position;
layout(location = 1) in vec4 color;

out Attributes
{
```

```
   vec3 position;
   vec4 color;
} att_out;

void main()
{
   att_out.position = position;
   att_out.color = color;
}
```

Powyższy shader wierzchołków składa się z trzech głównych części, którymi są: zmienne wejściowe, zmienne wyjściowe oraz funkcja `main()`. Zmienne wejściowe opatrzone kwalifikatorem `in`, poprzedzone dodatkowo kwalifikatorem `layout` są atrybutami aktualnie przetwarzanego wierzchołka, które są pobierane z bufora danych VBO przygotowanego i przywiązanego do kontekstu graficznego jeszcze przed wywołaniem funkcji rysującej. W tym przykładzie zastosowano jedynie pozycję oraz kolor wierzchołka. Należy mieć na uwadze, że zmienne wejściowe shaderów są dostępne tylko do odczytu. Zmienne wyjściowe z kolei są opatrzone kwalifikatorem `out` i są zawarte w bloku interfejsu o nazwie `Attributes`, który służy jako kontener do przekazywania danych między etapami. W samej definicji funkcji `main()` następuje jedynie proste przekazanie danych wejściowych do wyjściowych.

Shader wierzchołków w tej postaci jest tylko pośrednikiem w przekazywaniu danych. Nic nie stoi jednak na przeszkodzie, aby wzbogacić jego funkcjonalność. Można np. dostarczyć do programu macierze przekształceń i za ich pomocą przenieść wierzchołki z przestrzeni obiektu do przestrzeni świata albo widoku.

Listing 6.2. Przykładowa implementacja shadera kontroli teselacji

```
#version 400
layout(vertices = 3) out;

in Attributes
{
   vec3 position;
   vec4 color;
} att_in[3];

out Attributes
{
   vec3 position;
   vec4 color;
} att_out[3];

#define ID gl_InvocationID
```

```
void main()
{
   att_out[ID].position = att_in[ID].position;
   att_out[ID].color    = att_in[ID].color;

   if(ID == 0)
   {
      gl_TessLevelInner[0] = 3;
      gl_TessLevelOuter[0] = 3;
      gl_TessLevelOuter[1] = 3;
      gl_TessLevelOuter[2] = 3;
   }
}
```

Shader kontroli teselacji składa się z nieco większej liczby podstawowych elementów, o których trzeba pamiętać. Na początku należy określić liczbę jego wywołań, przypadających na pojedynczy płat za pomocą wyjściowego kwalifikatora `layout`. Często zrównuje się ją z rozmiarem samego płata, który jest ustalany z poziomu aplikacji. W tym przykładzie zarówno rozmiar płata, jak i liczbę uruchomień shadera ustawiliśmy na 3 (`vertices = 3`), co spowoduje, że każdemu wywołaniu shadera zostanie zlecone przetwarzanie jednego wierzchołka w płacie. Informacji o tym, który wierzchołek jest aktualnie przetwarzany, dostarcza zmienna wbudowana `gl_InvocationID`.

Kolejnymi ważnymi elementami są bloki interfejsu, tym razem zarówno wejściowe, jak i wyjściowe, a ponadto występujące w formie stablicowanej. Dla zmiennych wejściowych rozmiarem tablicy bloku jest liczba wierzchołków w płacie, natomiast dla zmiennych wyjściowych – liczba wywołań shadera. Ponownie, dane w niezmienionej formie przenosimy do następnego etapu.

Interesującym fragmentem kodu jest blok wewnątrz instrukcji warunkowej, w którym następuje określenie stopnia teselacji. Renderujemy trójkąty, dlatego zagęszczenie ustalamy osobno dla trzech pierwszych elementów tablicy `gl_TessLevelOuter` odpowiadających trzem krawędziom i dla pierwszego elementu tablicy `gl_TessLevelInner`, decydującego o teselacji wewnętrznej.

Listing 6.3. Przykładowa implementacja shadera ewaluacji teselacji

```
#version 400
layout(triangles, equal_spacing, ccw) in;

in Attributes
{
   vec3 position;
   vec4 color;
```

```
} att_in[];
out Attributes
{
   vec3 position;
   vec4 color;
} att_out;

void main()
{
   att_out.position = gl_TessCoord.x * att_in[0].position +
                      gl_TessCoord.y * att_in[1].position +
                      gl_TessCoord.z * att_in[2].position;

   att_out.color = gl_TessCoord.x * att_in[0].color +
                   gl_TessCoord.y * att_in[1].color +
                   gl_TessCoord.z * att_in[2].color;
}
```

Na wstępie shadera ewaluacji teselacji znajduje się wejściowy kwalifikator `layout`. Za jego pomocą zostaje określony obiekt poddawany teselacji, rozstaw nowo powstałych wierzchołków i kierunek nawijania w przypadku renderowania trójkątów. W powyższym przykładzie obiektem podstawowym jest trójkąt (`triangles`), który dzielimy w sposób jednorodny (`equal_spacing`) i którego porządek wierzchołków określamy na przeciwny do ruchu wskazówek zegara (`ccw` – w rozwinięciu: ang. *counter clock wise*). W przypadku korzystania z mechanizmu teselacji w potoku, to właśnie shader ewaluacji teselacji jest pierwszym etapem, w którym dokonuje się decyzji o tym, jaki prymityw zostanie wyrenderowany. Można go jednak jeszcze zmienić w opcjonalnym shaderze geometrii.

Zmienne wejściowe oraz wyjściowe ponownie pojawiają się w formie bloków, jednak tym razem jedynie blok wejściowy jest stablicowany. Dostarcza on dane ze wszystkich wywołań shadera kontroli teselacji dla aktualnie przetwarzanego płatu. Wyjściowy blok jest pojedynczą instancją, ponieważ shader ewaluacji teselacji jest wykonywany indywidualnie i niezależnie dla każdego nowo powstałego wierzchołka prymitywu zadeklarowanego w wejściowym kwalifikatorze `layout`.

W funkcji `main()` następuje obliczenie pozycji i wartości koloru aktualnie przetwarzanego wierzchołka. W efekcie teselacji pojawiają się nowe wierzchołki, dlatego ich pozycja i wartości koloru muszą zostać odpowiednio wyliczone. Każde wywołanie shadera ewaluacji teselacji dostarcza za pomocą zmiennej wbudowanej `gl_TessCoord` lokalne współrzędne służące do interpolacji atrybutów. W przypadku trójkąta są to współrzędne barycentryczne. Znając zatem bazową pozycję i wartości koloru wierzchołków przetwarzanego prymitywu, można dokonać ich prostej interpolacji.

Listing 6.4. Przykładowa implementacja shadera geometrii

```
#version 400
layout(triangles, invocations = 1) in;
layout(triangle_strip, max_vertices = 3) out;
in Attributes
{
   vec3 position;
   vec4 color;
} att_in[];

out vec4 color;

void main()
{
   for(int i = 0; i < 3; i++)
   {
      color = att_in[i].color;
      gl_Position = vec4(att_in[i].position, 1.0);
      EmitVertex();
   }
   EndPrimitive();
}
```

Na początku shadera geometrii występują dwa kwalifikatory `layout`. Pierwszy dotyczy interfejsu wejścia i określa rodzaj wejściowego prymitywu oraz liczbę wywołań shadera przypadających na pojedynczy prymityw trafiający do niego na wejściu. W naszym przypadku z poprzedniego etapu otrzymujemy trójkąt (`triangles`) będący jednym z wielu utworzonych w procesie teselacji i sam shader wywołujemy tylko raz dla każdego takiego trójkąta. Drugi wyjściowy z kolei decyduje o rodzaju tworzonego prymitywu oraz pozwala określić maksymalną liczbę wierzchołków, które mogą zostać wyemitowane w pojedynczym wywołaniu shadera. Tutaj podobnie jako obiekt wyjściowy ustalamy trójkąt (`triangle_strip`), który jest tworzony raz na wywołanie shadera przez wyemitowanie trzech tworzących go wierzchołków (`max_vertices=3`).

Rozmiar bloku danych wejściowych zależy od rodzaju wejściowego prymitywu. Jeżeli otrzymujemy trójkąt, to tablica będzie przechowywać trzy elementy. Pewną zmianą w stosunku do wcześniejszych etapów jest brak bloku wyjściowego, który został zastąpiony pojedynczym atrybutem z kwalifikatorem `out`. Pozycja wierzchołków nie będzie nam już potrzebna w shaderze fragmentów, więc mogliśmy sobie pozwolić na zastosowanie tej skrótowej formy. Nic nie stoi jednak na przeszkodzie, aby zastosować blok interfejsu z pojedynczą zmienną.

W funkcji `main()` znajduje się pętla przechodząca po wszystkich wierzchołkach, w której wykonywane są trzy instrukcje. Najpierw następuje przypisanie

wartości koloru wejściowego do wyjściowego. Następnie pozycja wierzchołka zostaje zapisana do zmiennej wbudowanej `gl_Position`. To na jej podstawie następuje pozycjonowanie obiektu na ekranie i zapis do niej jest obowiązkowy, jeżeli chcemy zobaczyć poprawnie wyrenderowane obiekty. Zwykle zapisuje się do niej pozycje w przestrzeni ograniczonej (ang. *clip space*), czyli po przemnożeniu przez macierz projekcji. Następnie, gdy w zmiennej `gl_Position` znajduje się już odpowiednia wartość pozycji, wierzchołek zostaje wyemitowany za pomocą funkcji `EmitVertex()`. Po trzykrotnym wykonaniu tych kroków jest wywoływana funkcja `EndPrimitive()`, która kończy emisję prymitywu. Tak przetworzony obiekt trafia następnie do etapu rasteryzacji, po czym przechodzi do ostatniego programowalnego etapu w potoku.

Listing 6.5. Przykładowa implementacja shadera fragmentów

```
#version 400

in vec4 color;
out vec4 fragment_color;

void main()
{
   fragment_color = color;
}
```

Powyższy shader fragmentów jest bardzo prostym i krótkim programem. Otrzymuje on jedynie kolor z shadera geometrii, który jest automatycznie interpolowany na całej powierzchni prymitywu. Jego wartość jest zapisywana do zmiennej wyjściowej o nazwie `fragment_color`, która reprezentuje pojedynczy fragment/piksel bufora ramki.