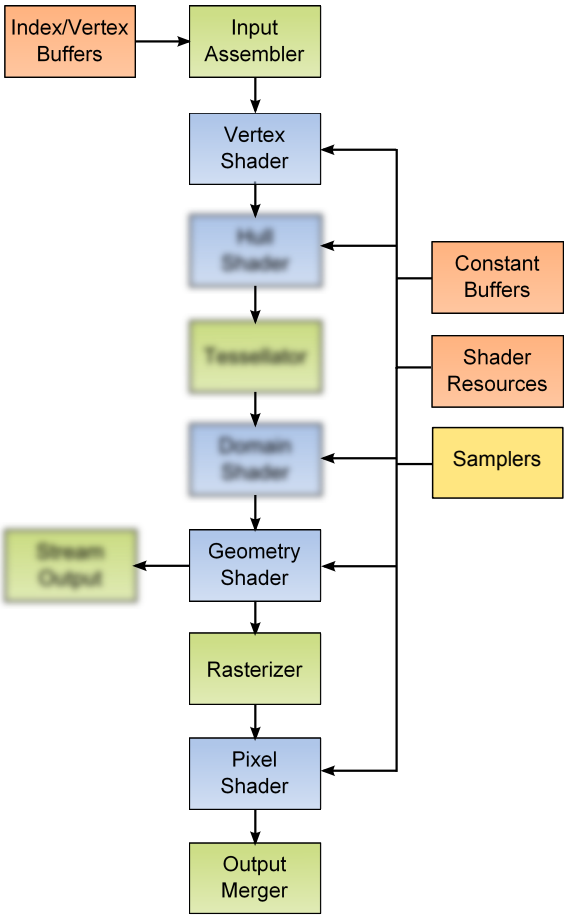
Programowanie w API Graficznych

Laboratorium

Direct3D 11 - ćwiczenie 4

Jakub Stępień, 2012 (2014)



Rysunek 1: Stadia potoku D3D11 wykorzystane w ramach ćwiczenia 4

1 Cel ćwiczenia

Celem ćwiczenia jest poznanie procesu tworzenia i inicjowania bufora głębokości, a także zapoznanie się z shaderami geometrycznymi zarówno od strony aplikacji, jak i z poziomu HLSL-a.

Pierwsza część ćwiczenia (dot. bufora *depth-stencil*) nadal nie wymaga funkcjonalności powyżej D3D w wersji 9.1, ale shadery geometryczne już przez nią wspierane nie są - w końcu zahaczamy o D3D 10.0.

2 Zadania

2.1 Zadanie 1 - bufor głębokości

Uruchom aplikację - zamiast wyeksploatowanego już prostokąta, zobaczysz sześcian. No, powiedzmy, bo wyraźnie coś z nim jest z nim nie tak. W tym zadaniu twoim celem będzie doprowadzenie do poprawnego renderingu sześcianu. W tym celu będziesz musiał utworzyć, skonfigurować i przywiązać do potoku bufor głębokości (na razie rendering odbywa się bez niego: poszukaj wywołania `OMSetRenderTargets(...)` - trzeci argument to właśnie wskaźnik widoku bufora *depth-stencil*).

2.1.1 Tekstura dla bufora

Bufor głębokości to po prostu blok pamięci, czyli zasób pamięciowy, więc pierwszym krokiem, jaki musimy zrobić, będzie zarezerwowanie sobie odpowiednio dużo pamięci. W tym celu utworzymy teksturę 2D (pomimo zwyczajowej nazwy *bufor głębokości*), pamiętając o jej docelowym zastosowaniu, gdy będziemy określać flagi wiązania.

Przejdź do `dxCreateResourcesAndStateObjectsAndShaders()` i poszukaj metody odpowiedzialnej za utworzenie zasobów teksturowych dla bufora *depth-stencil*, a następnie przejdź do niej. Zgodnie ze znanym schematem, najpierw wypełniasz strukturę konfiguracyjną, po czym tworzysz zasób przy pomocy metody urządzenia `CreateTexture2D(...)`. Uzupełnij odpowiednio flagę wiązania i kod tworzący zasób.

2.1.2 Widok zasobu/tekstury

Następnie w metodzie `dxCreateDepthStencilViews()` odkomentuj kod wypełniania struktury konfiguracyjnej i określ sposób, w jaki zasób ukrywający się za tworzonym właśnie widokiem będzie interpretowany (czyt. ustaw odpowiednio składową `Format` struktury konfiguracyjnej). Teraz odkomentuj kod odpowiedzialny za utworzenie widoku.

Pamiętasz z poprzedniego ćwiczenia znaczenie składowej `ViewDimension`? Jaką rolę pełni w strukturze konfiguracyjnej widoku zasobu unia?

2.1.3 Obiekt stanu

Teraz przejdź do `dxCreateDepthStencilStates()`: tutaj tworzony jest obiekt stanu, konfigurujący te aspekty pracy stadium *Output Merger*, które wiążą

się z testami *depth-stencil*[1] - nas interesują teraz tylko testy głębokości (czyli samo *depth* ;)).

Włącz testy głębokości oraz wyłącz testy *stencil*. Teraz odkomentuj kod odpowiedzialny za utworzenie obiektu stanu bufora *depth-stencil*.

2.1.4 Konfiguracja stadium OM

Poszukaj w `dxRender()` metody konfigurującej stadium OM i przejdź do niej. Musisz:

- użyć `OMSetDepthStencilState(...)`, by ustawić utworzony przed chwilą obiekt stanu stadium OM (drugi parametr metody zignoruj)
- uzupełnić wywołanie `OMSetRenderTargets(...)` o utworzony wcześniej widok bufora głębokości (wskaźnik)

2.1.5 Widać coś? Nie bardzo?

Bufor głębokości zwykle musi być co klatkę czyszczony, tak jak *render target*. Użyj `ClearDepthStencilView(...)` według opisu zamieszczonego na platformie (msdn). Nie wiesz, gdzie jej użyć? Znajdź w `dxRender()` analogiczne wywołanie dla *render target*-u.

2.2 Zadanie 2 - Geometry Shader

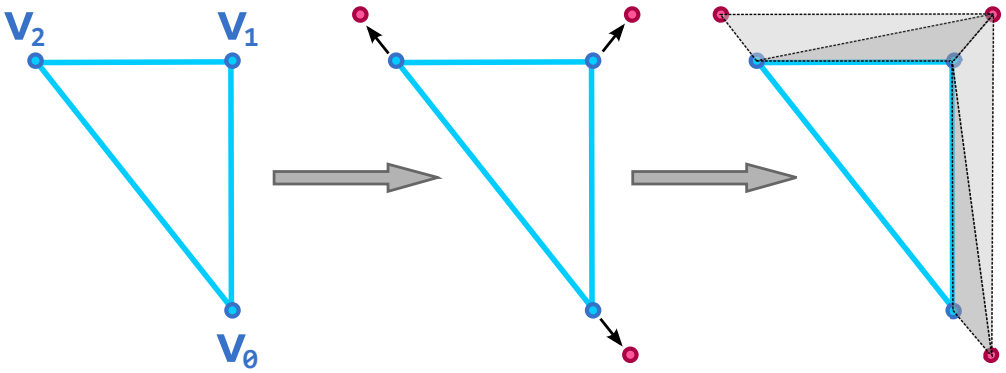
W tym zadaniu skompilujesz geometry shader, utworzysz reprezentujący go obiekt i przywiążesz go do potoku.

2.2.1 Kompilacja i tworzenie obiektu shadera

Przejdź do `dxCreateShaderObjects()` i odkomentuj linie odpowiedzialne za kompilację geometry shadera oraz jego obiektu. Przy wywołaniu kompilacji uzupełnij tekstowy identyfikator profilu modelu shadera (ang. *target profile*) (opis wszystkich profili znajdziesz na platformie).

Przebuduj i uruchom aplikację. Co się dzieje? Odszukaj w zakładce `Output` komunikatu o błędzie ("D3D11: ERROR..."). Czy coś Ci mówi jego treść?

[1] OM poza tym konfigurujemy innym obiektem stanu - *blend state object*, określającym ustawienia związane z mieszaniem kolorów/alfa



Rysunek 2: Dodawanie krawędzi w GS - koncepcja

2.2.2 Feature level

Problem polega na tym, że mamy ustawiony za niski poziom funkcjonalności, ograniczający nas DirectX 9.1, który nie wspiera geometry shaderów. Przejdź do `dxInitialize()` i odszukaj tablicę zawierającą poziomy funkcjonalności tolerowane przez naszą aplikację. Zamień jedyny jej element na taki, by wpisany przez Ciebie przed chwilą profil modelu shadera był wspierany (znowu może Ci się przydać wspomniana wcześniej lista profili kompilatora).

2.2.3 Wiązanie do potoku

Wiązanie do potoku i przypisywanie zasobów to nic nowego. Zobacz tylko, jak to wygląda w `dxConfigureGeometryShaderStage(...)`.

Flaga `m_UseGeometryShader` spięta jest z check-boxem, który włącza/wyłącza geometry shader (`Use GS`). Uruchom aplikację i włącz to stadium. Przykładowy shader jest już gotowy, powinieneś teraz zobaczyć efekty jego działania.

2.3 Zadanie 3 - krawędzie generowane w GS

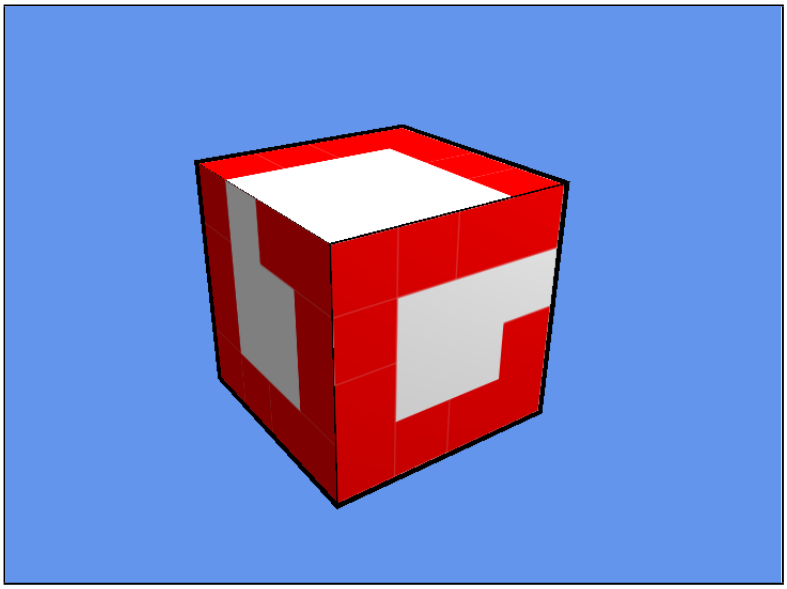
W trzecim zadaniu zmierzysz się ze stadium GS na poziomie HLSL-a. Celem jest dorysowanie do sześcianu poligonalnych (czworokątnych ;)) krawędzi, które nadadzą mu troszkę bardziej rysunkowy charakter.

Koncepcja rozwiązania tego zadania przedstawiona jest na Rys. 2:

- każdy z wierzchołków trójkąta generuje nowy wierzchołek
- trzy nowe wierzchołki w sumie z trzema oryginalnymi budują 4 trójkąty = 2 czworokąty = 2 krawędzie

Efekt Twojej pracy powinien wyglądać mniej więcej jak na Rys. 3. Poniżej znajdziesz wytyczne, jak to osiągnąć:

- wejście do GS to trójkąt (`triangle`)
- wyjście z GS to *strip* trójkątów (`TriangleStream`)
- zacznij od prostego forwardowania wierzchołków, które przyszły z VS; potrzebujemy ich, żeby poza krawędziami wyrenderować też sam oteksturowany sześcian
- 3 forwardowane wierzchołki + ich 3 kopie dla krawędzi + 3 wygenerowane daje 9 wierzchołków; taką wartość ustaw dla `maxvertexcount`
- kopiuj wierzchołki poprzez proste przypisanie, np.:
  `GSOutput v = input[0];`
- do generowania nowego wierzchołka użyj przygotowanej procedury `extrudeVertex(...)`, podając jako parametr oryginalny wierzchołek, np.: `GSOutput v = extrudeVertex(input[0]);`
- pamiętaj o konstruowaniu trójkątów w kolejności przeciwnej do ruchu wskazówek zegara (tak skonfigurowany jest przez aplikację potok)
- zastosowanie *strip*-u oznacza, że każdy kolejny wierzchołek daje nowy trójkąt (zbudowany z dwóch poprzednich i tego nowego wierzchołka)
- w momencie dorzucania kolejnych wierzchołków do stripu, nie musisz już troszczyć się o wskazówki zegara, potok sam odpowiednio zinterpretuje dane
- dla każdego wierzchołka wygenerowanego w GS (czyli nie uwzględnia to wierzchołków forwardowanych) ustaw zmienną `colMult` na same zera (podstawowe wierzchołki mają tam jedynki; `colMult` jest wykorzystane w PS, by odróżnić krawędzie od bazowego sześcianu)



Rysunek 3: Sześcian z dodatkowymi krawędziami