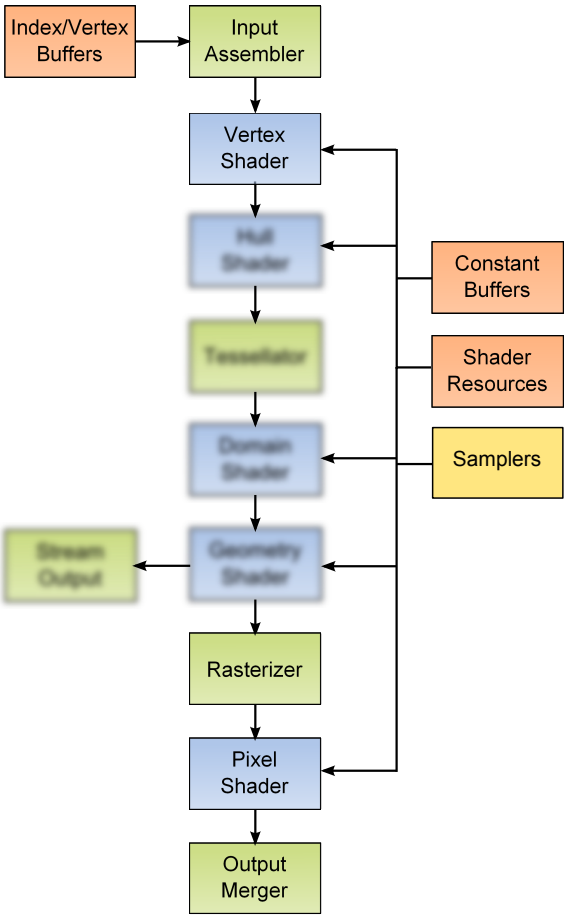
Programowanie w API Graficznych

Laboratorium

Direct3D 11 - ćwiczenie 3

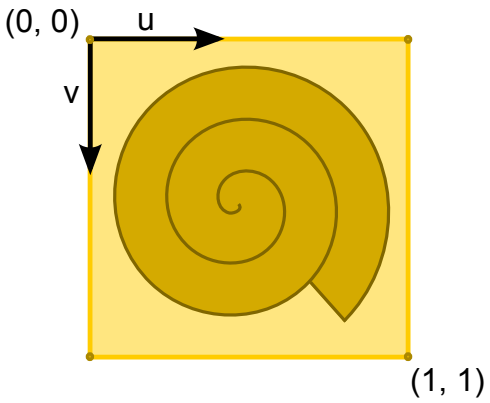
Jakub Stępień, 2012 (2014)



Rysunek 1: Stadia potoku D3D11 wykorzystane w ramach ćwiczenia 3

1 Cel ćwiczenia

Celem ćwiczenia jest poznanie procesu tworzenia i inicjowania tekstur (2D), jak również wiązanie ich do potoku poprzez odpowiedni widok zasobu oraz konfigurowanie *jednostek próbkujących*. Poza tym nauczysz się, jak próbkować teksturę z poziomu HLSL.
Mimo że sposób obsługi tekstur wyraźnie zmienił się od czasów D3D9, to nie jest to żadna nowa funkcjonalność specyficzna dla D3D11 czy nawet D3D10, a więc nadal trzymamy się *feature level* `D3D_FEATURE_LEVEL_9_1` i odpowiadających mu modeli shaderów.



Rysunek 2: Układ osi układu współrzędnych teksturowych (2D) dla Direct3D

2 Zadania

2.1 Zadanie 1 - współrzędne teksturowe

Zmodyfikuj kod tworzenia wierzchołków, *input layout*-u i shaderów tak, by bazował na wierzchołkach typu `Vertex_PosUV`.

2.1.1 Wierzchołki

Zacznij od zmodyfikowania `typedef`-u w linii siódmej pliku `renderwidget.cpp`. Teraz zaktualizuj generację wierzchołków, uwzględniając w konstruktorze dwa dodatkowe parametry - współrzędną u (poziomą) i v (pionową) w zakresie `[0.f, 1.f]`. Dla *przypomnienia* układ osi dla współrzędnych teksturowych Direct3D został przedstawiony na Rys. 2.

2.1.2 Input layout

Następnie w metodzie `dxCreateInputLayouts()` utwórz i dodaj do `vertexBufferElements` kolejną strukturę `D3D11_INPUT_ELEMENT_DESC`. Naturalna propozycja wyboru semantyki: `TEXCOORD`, indeks `0`. Pamiętaj o poprawnym ustawieniu składowej `AlignedByteOffset` (piąta z kolei).

2.1.3 Shadery

Teraz pozostaje już tylko modyfikacja kodu shadera - zmień strukturę wejściową vertex shadera, pamiętając o tym, by dobrze ustawić semantyki (zgodnie z pierwszymi dwoma parametrami `CreateInputLayout`); struktura wyjściowa vertex shadera (= wejściowa pixel shadera) musi teraz zawierać również współrzędne teksturowe wierzchołka/fragmentu.

2.1.4 Efekty?

No i co? No i nic, render powinien wyglądać bez zmian, bo przecież nic z tymi dodatkowymi informacjami *wkładanymi* do bufora wierzchołków nie robimy. Jeszcze niedawno efekty można byłoby sprawdzić w programie PIX i tak też robiliśmy na tym ćwiczeniu. Teraz jednak program ten nie jest oficjalnie wspierany i w zasadzie nie można na nim polegać, korzystając z DirectX 11. Jeżeli masz ochotę poeksperymentować, to spróbuj uruchomić debugger graficzny dostępny w Visual Studio 2012 i 2013. Interfejs w niczym nie przypomina PIXa, ale jest tam możliwość analizowanie kodu shadera w locie, które umożliwi Ci podejrzenie wprowadzonych przed siebie zmian "w akcji".

2.2 Zadanie 1 - tekstura i jej widok

W tym zadaniu utworzysz zasób typu `ID3D11Texture2D` (na bazie pliku z obrazkiem), a także jego widok (ang. *Shader Resource View*), który następnie wykorzystasz, wiążąc teksturę do stadium PS.

2.2.1 Tworzenie/ładowanie tekstury

W metodzie `dxCreateTextures()` uzupełnij jedyny brakujący argument procedury pomocniczej `D3DX11CreateTextureFromFile(...)`, wykorzystując zmienną `m_pTexture`[1]. Pozostałe parametry są już wprowadzone poprawnie.

2.2.2 Tworzenie widoku tekstury

Widoki zasobów shaderów tworzymy przy pomocy metody urządzenia `CreateShaderResourceView(...)` i struktury konfiguracyjnej typu `D3D11_SHADER_RESOURCE_VIEW_DESC` (ale sposób wykorzystania będzie taki sam dla wszystkich typów widoków).

Powinieneś się już domyślać, że Twoim zadaniem będzie odpowiednie ustawienie `ViewDimension` oraz pól odpowiadającej mu podstruktury (opis we wprowadzeniu) - `Texture2D`.

`Texture2D` to struktura typu `D3D11_TEX2D_SRV` i zawiera dwie składowe, które określają zakres mip-map kryjącej się za zasobem tekstury, które będą

[1]Po czym dziedziczy ID3D11Texture2D? W razie potrzeby zajrzyj do wprowadzeń lub zapytaj wuja G.

dostępne/widoczne dla korzystających z tekstury shaderów poprzez tworzony właśnie SRV:

- `MostDetailedMip`: określa indeks najbardziej dokładnego mip-poziomu, który będzie dostępny; `0` oznacza, że najbardziej dokładnym będzie faktyczny najdokładniejszy mip-poziom tekstury; wartości większe wykluczają kolejne poziomy dokładności
- `MipLevels`: określa, ile (począwszy od `MostDetailedMip`) kolejnych, coraz mniej dokładnych mip-poziomów tekstury będzie dostępnych[2]

Co istotne - korzystający z danego SRV shader/sampler będzie *widział* (stąd widok) zasób tak, jakby innych mip-poziomów nie było.

2.3 Wiązanie widoku tekstury

Przejdź do metody `dxConfigurePixelShaderStage()`: przywiążesz teraz do stadium PS utworzony widok tekstury, by móc później próbkować kryjącą się za nim teksturę w procedurze pixel shadera. Odkomentuj i odpowiednio uzupełnij argumenty `PSSetShaderResources(...)` tak, by opiewały na utworzony przed chwilą `m_pTextureSRV`.

3 Zadanie 2 - konfiguracja samplera

3.1 Tworzenie konfiguracji samplera

Przejdź do metody `dxCreateSamplerStates()` i wypełnij składowe:

- `AddressU`
- `AddressV`
- `AddressW`[3]

w taki sposób, by uzyskać adresowanie typu *wrap*. Jak działa taki typ adresowania?

Teraz możesz odkomentować `CreateSamplerState(...)`, by utworzyć odpowiednio skonfigurowany obiekt stanu samplera.

[2]MipLevels nie może być więc większy niż: ilośćMipPoziomówTekstury - MostDetailedMip. Jeżeli ustawimy go na -1, wszystkie mip-poziomy począwszy od MostDetailedMip będą dostępne w danym widoku.

[3]Po co konfigurować składową *w* skoro korzystamy z tekstury 2D? Bo API nie pozwoli nam zrobić inaczej :).

3.2 Wiązanie konfiguracji samplera

Ponownie przejdź do metody `dxConfigurePixelShaderStage()` - tym razem przywiążesz do stadium PS konfigurację samplera. Odkomentuj i odpowiednio uzupełnij argumenty `PSSetSamplers(...)` tak, by opiewały na utworzony przed chwilą `m_pSamplerState`.

4 Zadanie 3 - HLSL

Pozostało już tylko odpowiednie wykorzystanie utworzonych i przywiązanych do potoku obiektów. Zgodnie z wprowadzeniem:

- zadeklaruj teksturę (o dowolnej nazwie)
- zadeklaruj obiekt stanu samplera (o dowolnej nazwie)
- w kodzie pixel shadera zwróć kolor tekstury odpowiadający współrzędnym danego fragmentu/piksela