Programowanie w API Graficznych

Laboratorium

Direct3D 11 - wprowadzanie

Jakub Stępień, 2012

Spis treści

Rozdział 1

DirectX

1.1 Co to jest DirectX?

DirectX to zbiór funkcji API stworzony w celu zapewnienia programistom abstrakcyjnego interfejsu do sprzętu takiego jak karta graficzna czy dźwiękowa. Dzięki temu że kod aplikacji *kontaktuje się* z API DirectX, a nie bezpośrednio z warstwą sterowników, zadaniem dostosowywania się do różnego rodzaju urządzeń zainstalowanego na komputerze potencjalnego użytkownika zostali obciążeni producenci sprzętu, gdyż to oni muszą zapewnić zunifikowany dostęp do produkowanych urządzeń. Programiści muszą opanować *tylko* zunifikowane API.

1.2 DirectX a Direct3D

Na DirectX składa się wiele komponentów/bibliotek:

- Direct2D - grafika 2D
- Direct3D - grafika 3D
- DXGI - niskopoziomowe zadania (interfejs do fizycznych urządzeń i ich zasobów)
- DirectSound - dźwięk
- DirectWrite - tekst/czcionki
- ...

Dzięki takiemu podziałowi możemy linkować aplikację tylko z tymi modułami, które oferują poszukiwaną przez nas funkcjonalność. W ramach PwAG skupimy się na Direct3D.

Rozdział 2

Potok graficzny D3D11

2.1 Koncepcja potoku

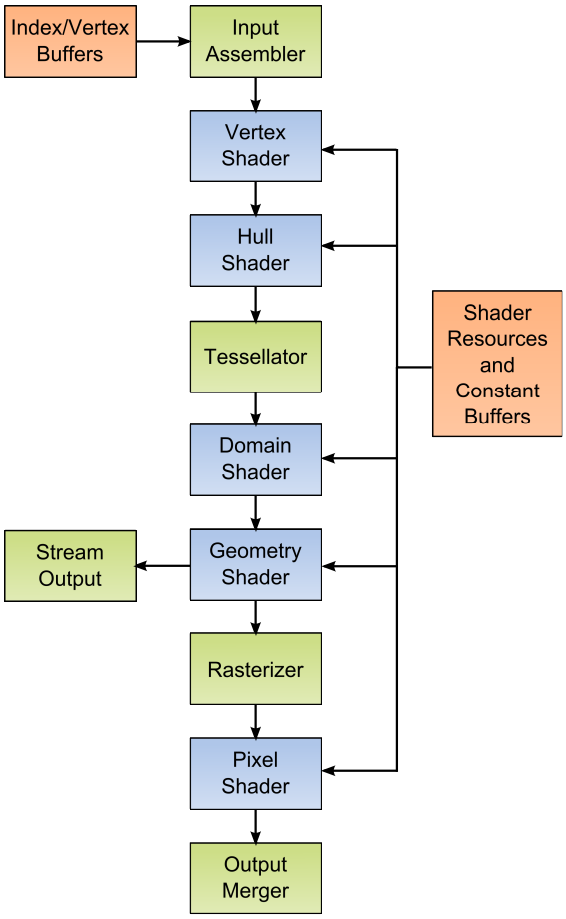
Potok przetwarzania graficznego można nazwać zalążkiem, z którego wyrosły architektury współczesnych chipsetów graficznych. Pierwsze akceleratory graficzne zapewniały sprzętowe przekształcenia wierzchołków, co umożliwiało przyspieszenie działania aplikacji 3D. Każda kolejna generacja sprzętu wprowadzała dodatkowe funkcjonalności, by umożliwić realizowanie coraz bardziej złożonego renderingu.
Podstawowym zadaniem potoku graficznego jest pobranie abstrakcyjnego opisu obiektu 3D i przekształcenie go w (płaski) obraz o formacie dostosowanym do prezentacji na wyznaczonym do tego oknie aplikacji.

2.2 Stadia potoku D3D11

Pełny potok Direct3D 11 jest przedstawiony na Rys. 2.1. Niebieskim kolorem zostały oznaczone jego programowalne stadia (ang. *programmable stages*), a zielonym nieprogramowalne/sztywne (ang. *fixed-function stages*).

2.2.1 Input Assembler (IA)

Input Assembler jest punktem wejścia potoku graficznego. Zadanie tego stadium polega na odczytywaniu danych wejściowych z odpowiednich zasobów i formowanie z nich wierzchołków dla dalszych stadiów przetwarzania. Poza tym IA grupuje sformowane wierzchołki w prymitywy geometryczne.



Rysunek 2.1: Potok graficzny Direct3D 11

2.2.2 Vertex Shader (VS)

Vertex Shader odczytuje sformowane już wierzchołki i niezależnie przetwarza każdy z nich. Jako że jest to stadium programowalne, jego dokładna rola w procesie przetwarzania jest pozostawiona programiście, ale tradycyjnie podstawowym zadaniem VS jest rzutowanie wierzchołków do *clip space*.

2.2.3 Hull Shader (HS) - Tessellator (TS) - Domain Shader (DS)

Te trzy stadia pojawiły się w D3D11 i pozwoliły na wzbogacenie potoku o (konfigurowalną) sprzętową teselację. Nie będziemy się w tej chwili na nich skupiać, żeby nie rozmyć na ich tle podstawowych funkcji potoku, ale w trakcie laboratoriów spotkamy się jeszcze z tymi stadiami.

2.2.4 Geometry Shader (GS)

To również stosunkowe nowe stadium potoku - pojawiło się wraz z D3D10. GS operuje na pełnych prymitywach geometrycznych (np. liniach, trójkątach): może konwertować prymityw jednego typu na inny, generować i wstrzykiwać do potoku *nowe* prymitywy (tzn. takie których nie wystawiła aplikacja).

2.2.5 Stream Output (SO)

To dość specyficzny element potoku i na razie nie będziemy go omawiać.

2.2.6 Rasterizer (RS)

Po opuszczeniu GS kończy się geometryczne przetwarzanie danych. Od tego momentu geometria jest rasteryzowana i przetwarzana jako *fragmenty*, czyli pule danych odpowiadające konkretnym pikselom *render targetu*. Informacje opisujące każdy taki fragment potencjalnie mogą być użyte w celu zaktualizowania koloru odpowiadającego im piksela. Atrybuty opisujące każdy fragment to głównie zinterpolowane dane, które oryginalnie (po wyjściu z VS) opisywały poszczególne wierzchołki. Zadaniem RS jest generowanie fragmentów na bazie dostarczonej mu geometrii.

2.2.7 Pixel Shader (PS)

Kiedy fragment zostanie już wygenerowany, jego atrybuty są wykorzystywane i przetwarzane przez Pixel Sader (zwany czasem Fragment Shaderem) w celu obliczenia odpowiadającego mu koloru dla każdego z *render targetów*. Poza określonymi obliczeniami, PS będzie zwykle w tym celu samplował kolor z odpowiednich tekstur. Poza kolorem PS *może* również modyfikować głębokość (odległość od kamery) przypisaną do danego fragmentu.

2.2.8 Output Merger (OM)

Output Merger przyjmuje kolor (i głębokość) fragmentów i po przeprowadzeniu szeregu operacji (testy głębokości i *stencil*, mieszanie kolorów) wykonuje zapis do odpowiednich zasobów wyjściowych (typowo jest to *render target* i bufor głębokości).

2.3 Konfiguracja (stadiów) potoku - obiekty stanu

W przypadku D3D9 zmiana stanu poszczególnych stadiów (sztywnego) potoku odbywała się przez wywołanie funkcji API dla każdego pojedynczego ustawienia. Wprowadzenie określonej konfiguracji potoku jako całości wymagało *wielu* wywołań API, co mogło prowadzić do problemów wydajnościowych. Wraz z D3D10 wprowadzone zostały tzw. obiekty stanu (ang. *state objects*), który grupują pojedyncze ustawienia w taki sposób, że jednym wywołaniem API konfigurujemy jedno stadium potoku (choć nie zawsze), co skutecznie ogranicza ilość wywołań API. D3D11 również korzysta z obiektów stanu do konfigurowanie nieprogramowalnych stadiów potoku (Rys. 2.2), czyli:

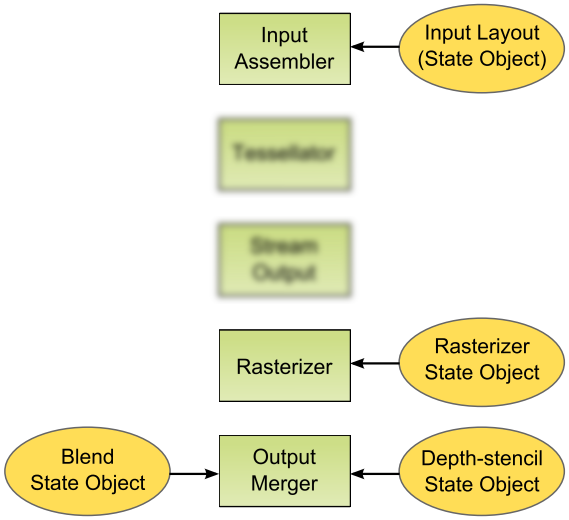
- Input Assembler - Input Layout (State Object)[1]
- Rasterizer - Rasterizer State Object
- Output Merger - Depth-stencil State Object i Blend State Object

Stadia Stream Output i Tessellator są konfigurowane w inny sposób, ale na razie nie będziemy ich omawiać.

2.4 Computation Pipeline

D3D11 wprowadza również stadium Compute Shader (CS), które umożliwia prowadzenie GPGPU. Technologia ta znana jest też pod nazwą DirectCompute. Jest to bardzo ciekawe i ważne rozszerzenie D3D, gdyż z definicji wykracza poza typowe programowanie graficzne, ale w tej chwili nie będziemy poświęcać mu więcej czasu.

[1]Nazywanie *input layout*-u obiektem stanu może być pewnym nadużyciem, ale spotkałem się z taką interpretacją i przypadła mi do gustu, gdyż pozwala na bardziej ogólne spojrzenie na obiekty stanu. Trzeba jednak pamiętać, że są pewne różnice: (i) konfigurując IA poza *input layout*-em musimy też określić topologię prymitywów; (ii) trochę inaczej wyglądają procedury API odpowiedzialne za tworzenie obiektów stanu i *input layout*-u.



Rysunek 2.2: Obiekty stanu wykorzystywane do konfiguracji sztywnych stadiów potoku

Rozdział 3

Zasoby po raz pierwszy

Sam potok przetwarzania nie jest w stanie wygenerować obrazu - musimy odpowiednim stadiom zapewnić różnego rodzaju zasoby, na bazie których będą one mogły pracować. W przypadku API D3D11 tak właśnie wygląda praca z potokiem, w czasach D3D9 było to mniej oczywiste z powodu mniej wyrazistego wyodrębnienia w nim poszczególnych stadiów i konieczności uwzględnienia w API obsługi sztywnego potoku przetwarzania. Mimo że możemy wiązać (ang. *bind*) z potokiem zasoby bardzo różnorodne, to są one zasadniczo albo **buforami**, albo **teksturami** i implementują interfejs `ID3D11Resource`.
By nie wprowadzać naraz zbyt wielu szczegółowych pojęć, ograniczymy się na razie do krótkiego omówienia podstawowej klasyfikacji zasobów. Odrobinę więcej powiemy tylko o buforach wierzchołków i indeksów (są stosunkowo proste i pojawiają się już na pierwszym ćwiczeniu) - inne rodzaje zasobów będą sukcesywnie wprowadzane w przyszłości.

3.1 Bufory

Zasób buforowy zapewnia jednowymiarowy liniowy blok pamięci do wykorzystania przez D3D11. Taki bufor może być skonfigurowany na wiele sposobów, ale zawsze będzie miał swoją podstawową liniową postać.

3.1.1 Bufory wierzchołków (VB)

Zadaniem bufora wierzchołków (ang. *vertex buffer*) jest przechowywanie wszystkich danych, które docelowo będą uformowane w wierzchołki (IA) i wprowadzone do potoku przetwarzania. Standardowo VB stanowi tablicę struktur opisujących kolejne wierzchołki, choć możliwe są bardziej złożone

konfiguracje (np. dane z kilku buforów jednocześnie, *instancing*).
Klasycznym miejscem wiązania buforów wierzchołków z potokiem jest Input Assembler[1] (zobacz też 3.4).

3.1.2 Bufory indeksów (IB)

Bufor indeksów (ang. *index buffer*) zapewnia możliwość pośredniego odnoszenia się do elementów bufora wierzchołków. Wyobraź sobie tablicę obiektów (VB), stanowiących dane i drugą tablicę (IB), zawierającą indeksy obiektów w VB oraz procedurę przetwarzającą, która potrafi jedynie *spacerować* wzdłuż tablicy danych i przetwarzać jeden element jednocześnie. Rozmieszczenie indeksów w IB decyduje o kolejności przetwarzania obiektów z VB, a jeżeli algorytm przetwarzający kilkakrotnie wykorzystuje te same dane z VB, nie ma konieczności kopiowania całych obiektów - wystarczy powtórne umieścić ich indeksy w IB, podczas gdy rozmiar VB pozostanie taki sam.
Korzystanie z IB nie jest obowiązkowe, można posługiwać się tylko buforami wierzchołków, ale ze względu na korzyści płynące z użycia IB, wykorzystanie ich jest bardzo powszechne.
Miejscem wiązania buforów indeksów z potokiem jest Input Assembler (zobacz też 3.4).

3.2 Tekstury

Określenie *tekstura* odnosi się do zasobu pamięciowego o charakterze *podobnym do obrazu*. To bardzo luźna definicja, gdyż istnieje wiele różnych typów tekstur, o różnych rozmiarach, topologiach itd. i wiele z nich wcale nie przypomina zwykłego dwuwymiarowego/prostokątnego obrazu. Elementem łączącym zasoby teksturowe z obrazami jest *piksel*, który jest tak samo zdefiniowany dla wszystkich typów tekstur. Piksel (zwany w przypadku tekstur *tekselem*) jest najmniejszym spójnym elementem tekstury. Każdy piksel/teksel reprezentowany jest przez maksymalnie cztery komponenty. Pomimo tego, że tekstury są również po prostu blokami pamięci dostępnymi dla GPU (tak jak zasoby buforowe), bardzo istotny jest fakt, iż chipsety graficzne zapewniają dodatkowy dedykowany osprzęt, który pozwala na wykonywanie określonych operacji na teksturach dużo wydajniej niż w przypadku buforów.

[1] Uwaga: Stream Output, ale o tym później.

3.3 Tworzenie zasobów

Nie ma sensu wchodzić w tym momencie w szczegóły procesu tworzenia poszczególnych rodzajów zasobów. Generalnie robimy to, korzystając z metod `ID3D11Device` zaczynających się od `Create` (np. `CreateBuffer(...)`). Wszystkie z tych metod przyjmują trzy parametry:

- wskaźnik do struktury opisującej opcje konfiguracyjne zasobu (zarówno wspólne jak i specyficzne dla różnych typów zasobów)
- wskaźnik do danych inicjujących zawartość zasobu (można przesłać `NULL` jeżeli nie chcemy tego robić)
- adres wskaźnika obiektu reprezentującego określony zasób (wyjście)

3.4 Wiązanie zasobów do potoku

O wiązaniu zasobów do potoku musimy pamiętać przy dwóch okazjach:

1. kiedy tworzymy zasób: jedną ze składowych struktury konfigurującej zasób są *flagi wiązania* (ang. *bind flags*) [2] (zobacz 3.3)
2. kiedy faktycznie wiążemy zasób do potoku

Jeżeli spodziewamy się, że dany zasób będzie wiązany do określonego stadium, musimy stworzyć go z odpowiednią flagą wiązania, *ale nie znaczy to, że tworząc ten zasób, przywiążemy go automatycznie do potoku!*

3.4.1 Bufory wierzchołków/indeksów

Flagi wiązania dla poznanych do tej pory zasobów:

- bufory wierzchołków: zawsze `D3D11_BIND_VERTEX_BUFFER`[3]
- bufory indeksów: zawsze `D3D11_BIND_INDEX_BUFFER`

Umożliwiają one późniejsze wiązanie buforów z IA, korzystając z metod kontekstu urządzenia: `IASetVertexBuffer(...)` i `IASetIndexBuffer(...)`.

[2] Jest to wspólna składowa - występuje we wszystkich strukturach konfiguracyjnych odpowiadających różnym rodzajom zasobów.

[3] Ponownie - uwaga na Stream Output

Rozdział 4

Trochę o HLSL-u

4.1 High Level Shading Language

Dla zachowania standardów zaczniemy od kilku słów wprowadzenia, mimo że tak naprawdę miałeś już styczność z wysokopoziomowymi językami programowania kart graficznych na wcześniejszych przedmiotach (HLSL/GLSL).
High Level Shading Language (HLSL) to język wywiedziony z C/C++, pozbawiony jednak wskaźników i związanych z nimi operacji oraz *template*-ów. Służy on do tworzenia programów cieniowania (ang. *shaders*), które są wykonywane przez jednostki obliczeniowe programowalnych stadiów potoku.
HLSL pojawił się wraz z D3D9 (D3D8 umożliwiał pisanie shaderów w tzw. asemblerze graficznym) i jest w dalszym ciągu rozwijany, by nadążyć za wymogami rynku. Począwszy od D3D10 nie ma już sztywnego potoku przetwarzania i każdy, kto chce renderować przy pomocy Direct3D, musi opanować ten język.
Również wraz z D3D10 została wprowadzona koncepcja zunifikowanego modelu shaderów (ang. *Unified Shader Model*), czyli wyposażenia wszystkich stadiów programowalnych w ten sam zestaw instrukcji. Łączy się to naturalnie z tendencją producentów kart graficznych do wyposażania ich w uniwersalne jednostki obliczeniowe (ang. *Unified Shader Architecture/Core*) dynamicznie przydzielane konkretnym programowalnym stadiom potoku.

4.2 HLSL - semantyki

Semantyki (ang. *semantics*) [1] w HLSL-u to metadane dołączane do definicji zmiennych. Semantyki towarzyszą HLSL od początku istnienia, ale wraz z

[1] Tak, beznadziejne tłumaczenie. Jestem otwarty na sugestie lepszych.

D3D10 zostały trochę (zdaniem autora) uporządkowane. W obecnej wersji ich rola jest trojaka:

1. identyfikują zmienne w komunikacji pomiędzy *stykającymi się* ze sobą stadiami lub aplikacją a wejściem potoku
2. pozwalają shaderom na żądanie i przyjmowanie z potoku specjalnych danych generowanych poza programowalnymi stadiami
3. pozwalają shaderom na przesyłanie specjalnych danych do potoku

Te *specjalne dane* brzmią na razie pewnie dość tajemniczo - nie martw się, wszystko wyjaśni się w swoim czasie. Teraz zapamiętaj tylko, że takim danym opowiadają semantyki z prefiksem `SV_` od *system value*.
Przed semantykami nie ma ucieczki, będą nam towarzyszyć od najprostszych shaderów, gdy będziemy określać parametry wejściowe/wyjściowe dla ich procedur głównych (ang. entry points). Przykłady możesz zobaczyć na listingu 4.1.

```
float4 inPos : POSITION;        //przypadek 1
float4 outPos : SV_POSITION;    //przypadek 3
```

Listing 4.1: Przykładowe semantyki

4.3 Zasoby a HLSL

Po stronie HLSL mamy *odbicie* obiektów zasobów, którymi posługujemy się na poziomie aplikacji/API. Obiekty te nazywane są czasami obiektami zasobów HLSL (ang. *HLSL resource objects*). Każdy zasób, do którego chcemy mieć bezpośredni dostęp z kodu shaderów, musi mieć zdefiniowany w HLSL *odpowiedni* obiekt zasobu. Zauważ jednak, że nie wszystkie typy zasobów mogą być bezpośrednio wykorzystywane/widoczne w kodzie shaderów - poznane już bufory wierzchołków/indeksów mogą służyć za przykład; dla tych typów nie istnieją w związku z tym odpowiadające im rodzaje obiektów zasobów HLSL.

Rozdział 5

Zasoby po raz drugi

5.1 Bufory stałych

W poprzednich rozdziałach tego opracowania i w ramach pierwszego ćwiczenia laboratoryjnego miałeś okazję trochę poznać już bufory wierzchołków i indeksów. Teraz do tej grupy dorzucimy bufor stałych (and. *constant buffer*), w skrócie *cbuffer*. Ma on nie tylko inna rolę (o której za chwilę), ale i strukturę - bufory wierzchołków/indeksów można intuicyjnie wyobrazić sobie jako tablicę (powtarzany wielokrotnie określony typ danej); dla każdego bufora stałych również określimy zawartość podstawowego elementu, ale wystąpi on tylko raz. Można więc powiedzieć, że bufory stałych mają formę struktury.
Rola *cbuffer*-ów polega na dostarczaniu shaderom (stadiom programowalnym) zmiennych, które nie ulegają zmianie w trakcie pojedynczego żądania rysowania (ang. *Draw Call*). Brzmi to może mgliście, ale na dobrą sprawę nas interesuje tylko tyle, że w kodzie shadera definiujemy zmienne, a aplikacja ustawia ich wartości wykorzystując *cbuffer* i odpowiednie wywołania API, po czym uruchamia potok przy pomocy *Draw Call*.
Bufory stałych zostały wprowadzone w D3D10, ale nie wniosły żadnej rewolucyjnej funkcjonalności. Różnica w porównaniu do D3D9 jest taka, że nie ustawiamy poszczególnych zmiennych z osobna, tylko całymi grupami. Tak naprawdę zmienne już w ogóle nie występują osobno, definiujemy je jako element określonego bufora stałych i z poziomu aplikacji ustawiamy cały bufor[1].

[1]Co ciekawe - jeżeli w kodzie shadera stworzymy sobie *luźną* zmienną, niejawnie zostanie ona i tak przypisana do domyślnie utworzonego bufora stałych o nazwie $Globals.

5.2 Bufor stałych - aplikacja

5.2.1 Tworzenie

Bufory stałych, jak wszystkie zasoby buforowe, tworzone są przy pomocy `ID3D11Device::CreateBuffer(...)`. Jak wszystkie metody tworzenia zasobów, metody `CreateBuffer` przyjmują trzy argumenty (zobacz 3.3).

Jedyną flagą wiązania, jakiej możemy użyć tworząc bufor stałych, jest `D3D11_BIND_CONSTANT_BUFFER` - pozwoli ona nam wiązać bufor z każdym z programowalnych stadiów potoku (zobacz 3.4 i 6.2.3).

5.2.2 Modyfikacja zawartości

Podstawową metodą modyfikowania zawartości buforów stałych (i nie tylko) z poziomu aplikacji/procesora jest para metod kontekstu urządzenia:

- `ID3D11DeviceContext::Map(...)`
- `ID3D11DeviceContext::Unmap(...)`

Najpierw wywołujemy metodę `Map()`, dzięki czemu uzyskujemy wskaźnik, do którego będziemy zapisywać [2]. Od tego momentu GPU nie może korzystać z zasobu, a my z poziomu aplikacji/CPU możemy swobodnie zmodyfikować zmapowany obszar pamięci (choćby przy pomocy `memcpy`). Po zakończeniu wywołujemy `Unmap()`, które unieważni mapowanie i odblokuje dostęp do zasobu dla GPU[3].

5.2.3 Wiązanie do stadiów potoku

Bufory stałych wiążemy do konkretnych stadiów potoku wywołując jedną z metod:

- `ID3D11DeviceContext::VSSetConstantBuffers(...)`
- `ID3D11DeviceContext::HSSetConstantBuffers(...)`
- `ID3D11DeviceContext::DSSetConstantBuffers(...)`
- `ID3D11DeviceContext::GSSetConstantBuffers(...)`

[2] Możliwy jest również odczyt - decyduje o tym parametr określający typ mapowania.

[3] Jeżeli zajrzałeś do dokumentacji, mogło zaskoczyć cię pojawiające się określenie *subresource* - na razie się tym nie przejmuj.

- `ID3D11DeviceContext::PSSetConstantBuffers(...)`

Nazwy mówią chyba same za siebie.

5.3 Bufor stałych - HLSL

W przeciwieństwie do poznanych do tej pory buforów wierzchołków i indeksów, bufor stałych jest zasobem, który jest bezpośrednio dostępny z poziomu kodu shaderów, czas więc poznać sposób, w jaki się do niego należy odnosić w HLSL-u. Bufory stałych do kodu shaderów wprowadzamy przy pomocy słowa kluczowego `cbuffer`, po którym następuje definicja struktury:

```
cbuffer VSConstants
{
  float4x4 ViewProjMatrix;      //float4x4 <=> matrix
  float3 LightDirection;
};
```

Listing 5.1: Bufor stałych w HLSL-u

Nazwy struktury/bufora (opcjonalne) są wykorzystywane jedynie do identyfikacji buforów przez aplikację[4], a w ramach samego HLSL do poszczególnych zmiennych odnosimy się bezpośrednio, tak jakby były dostępne globalnie.

[4]Poprzez mechanizm *refleksji shaderów* (ang. *Shader Reflection*); alternatywnie możemy w HLSL-u przypisywać cbuffer jawnie do określonego rejestru i na tej podstawie identyfikować go z poziomu aplikacji.

Rozdział 6

Zasoby po raz drugi

6.1 Bufory stałych

W poprzednich rozdziałach tego opracowania i w ramach pierwszego ćwiczenia laboratoryjnego miałeś okazję trochę poznać już bufory wierzchołków i indeksów. Teraz do tej grupy dorzucimy bufor stałych (and. *constant buffer*), w skrócie *cbuffer*. Ma on nie tylko inna rolę (o której za chwilę), ale i strukturę - bufory wierzchołków/indeksów można intuicyjnie wyobrazić sobie jako tablicę (powtarzany wielokrotnie określony typ danej); dla każdego bufora stałych również określimy zawartość podstawowego elementu, ale wystąpi on tylko raz. Można więc powiedzieć, że bufory stałych mają formę struktury.
Rola *cbuffer*-ów polega na dostarczaniu shaderom (stadiom programowalnym) zmiennych, które nie ulegają zmianie w trakcie pojedynczego żądania rysowania (ang. *Draw Call*). Brzmi to może mgliście, ale na dobrą sprawę nas interesuje tylko tyle, że w kodzie shadera definiujemy zmienne, a aplikacja ustawia ich wartości wykorzystując *cbuffer* i odpowiednie wywołania API, po czym uruchamia potok przy pomocy *Draw Call*.
Bufory stałych zostały wprowadzone w D3D10, ale nie wniosły żadnej rewolucyjnej funkcjonalności. Różnica w porównaniu do D3D9 jest taka, że nie ustawiamy poszczególnych zmiennych z osobna, tylko całymi grupami. Tak naprawdę zmienne już w ogóle nie występują osobno, definiujemy je jako element określonego bufora stałych i z poziomu aplikacji ustawiamy cały bufor[1].

[1]Co ciekawe - jeżeli w kodzie shadera stworzymy sobie *luźną* zmienną, niejawnie zostanie ona i tak przypisana do domyślnie utworzonego bufora stałych o nazwie $Globals.

6.2 Bufor stałych - aplikacja

6.2.1 Tworzenie

Bufory stałych, jak wszystkie zasoby buforowe, tworzone są przy pomocy `ID3D11Device::CreateBuffer(...)`. Jak wszystkie metody tworzenia zasobów, metody `CreateBuffer` przyjmują trzy argumenty (zobacz 3.3).

Jedyną flagą wiązania, jakiej możemy użyć tworząc bufor stałych, jest `D3D11_BIND_CONSTANT_BUFFER` - pozwoli ona nam wiązać bufor z każdym z programowalnych stadiów potoku (zobacz 3.4 i 6.2.3).

6.2.2 Modyfikacja zawartości

Podstawową metodą modyfikowania zawartości buforów stałych (i nie tylko) z poziomu aplikacji/procesora jest para metod kontekstu urządzenia:

- `ID3D11DeviceContext::Map(...)`
- `ID3D11DeviceContext::Unmap(...)`

Najpierw wywołujemy metodę `Map()`, dzięki czemu uzyskujemy wskaźnik, do którego będziemy zapisywać [2]. Od tego momentu GPU nie może korzystać z zasobu, a my z poziomu aplikacji/CPU możemy swobodnie zmodyfikować zmapowany obszar pamięci (choćby przy pomocy `memcpy`). Po zakończeniu wywołujemy `Unmap()`, które unieważni mapowanie i odblokuje dostęp do zasobu dla GPU[3].

6.2.3 Wiązanie do stadiów potoku

Bufory stałych wiążemy do konkretnych stadiów potoku wywołując jedną z metod:

- `ID3D11DeviceContext::VSSetConstantBuffers(...)`
- `ID3D11DeviceContext::HSSetConstantBuffers(...)`
- `ID3D11DeviceContext::DSSetConstantBuffers(...)`
- `ID3D11DeviceContext::GSSetConstantBuffers(...)`

[2] Możliwy jest również odczyt - decyduje o tym parametr określający typ mapowania.

[3] Jeżeli zajrzałeś do dokumentacji, mogło zaskoczyć cię pojawiające się określenie *subresource* - na razie się tym nie przejmuj.

- `ID3D11DeviceContext::PSSetConstantBuffers(...)`

Nazwy mówią chyba same za siebie.

6.3 Bufor stałych - HLSL

W przeciwieństwie do poznanych do tej pory buforów wierzchołków i indeksów, bufor stałych jest zasobem, który jest bezpośrednio dostępny z poziomu kodu shaderów, czas więc poznać sposób, w jaki się do niego należy odnosić w HLSL-u. Bufory stałych do kodu shaderów wprowadzamy przy pomocy słowa kluczowego `cbuffer`, po którym następuje definicja struktury:

```
cbuffer VSConstants
{
  float4x4 ViewProjMatrix;      //float4x4 <=> matrix
  float3 LightDirection;
};
```

Listing 6.1: Bufor stałych w HLSL-u

Nazwy struktury/bufora (opcjonalne) są wykorzystywane jedynie do identyfikacji buforów przez aplikację[4], a w ramach samego HLSL do poszczególnych zmiennych odnosimy się bezpośrednio, tak jakby były dostępne globalnie.

[4]Poprzez mechanizm *refleksji shaderów* (ang. *Shader Reflection*); alternatywnie możemy w HLSL-u przypisywać cbuffer jawnie do określonego rejestru i na tej podstawie identyfikować go z poziomu aplikacji.

Rozdział 7

Widoki zasobów

Do tej pory spotkałeś się z typami zasobów, które były wiązane do stadiów potoku *bezpośrednio*, czyli w momencie wiązania po prostu korzystaliśmy ze wskaźnika do `ID3D11Buffer`. Ten sposób wiązania jest wystarczający dla zasobów przywiązanych do potoku jako:

- bufory wierzchołków
- bufory indeksów
- bufory stałych
- bufory w stadium Stream Ouput[1]

Dla pozostałych typów wiązań (a są jeszcze cztery) musimy dodatkowo stworzyć obiekt *widoku zasobu* (ang. *Resource View*). I teraz oczywiście pytanie - po co mi to? Widoki zasobów są czymś w rodzaju *wrapper*-a na zasób, przez który stadia potoku widzą go w jakiejś specyficznej odsłonie:

- mogą widzieć tylko fragment pamięci związanej z zasobem
- mogą widzieć zasób po uwzględnieniu dodatkowych opcji konfiguracyjnych, które nie wpływają na sam zasób
- mogą widzieć ten sam zasób jako pamięć do odczytu lub odczytu i zapisu

Widoki zasobów wprowadzamy pokrótce teraz, bo w następnym rozdziale będą omawiane tekstury, których *nie można* wiązać z zasobem bez pośrednictwa widoku.

[1]Tak, to znowu ten nieomówiony dotąd Stream Output.

7.1 Typy widoków zasobów

W D3D11 mamy cztery typy widoków zasobów, z których każdy odpowiada miejscu wiązania kryjącego się za nim zasobu:

- widoki *render targetu* (ang. *Render Target Views*) - `RTV(s)`
- widoki bufora *depth-stencil* (ang. *Depth stencil Views*) - `DSV(s)`
- widoki zasobów shaderów (ang. *Shader Resource Views*) - `SRV(s)`
- widoki dostępu R/W (ang. *Unordered Access Views*) - `UAV(s)`

Nie będziemy ich tutaj szczegółowo omawiać, dowiesz się o nich więcej w odpowiednim czasie.

7.2 Tworzenie widoków zasobów

Nie zdziwi Cię pewnie fakt, że by utworzyć widok zasobu, należy wypełnić odpowiednią strukturę konfiguracyjną i przesłać jej wskaźnik jako parametr metody urządzenia o nazwie zaczynającej się od `Create`.

Wszystkie struktury konfiguracyjne widoków zasobów zawierają następujące składowe (choć mogą mieć ich więcej):

- `Format` - format na jaki zostanie *zrzutowany* kryjący się za tworzonym widokiem zasób w momencie odczytu
- `ViewDimension` - brzmi podejrzanie, ale określa po prostu typ zasobu, kryjącego się za tworzonym właśnie widokiem (w ten sposób określimy, czy widok będzie *pokazywał* teksturę 1D, 2D czy 3D albo może bufor)
- unia *podstruktur*[2] reprezentujących szczegółowe opcje konfiguracyjne dla danego typu zasobu

Unia podstruktur może brzmieć trochę groźnie, ale sprowadza się to do tego, że w zależności od zawartości składowej `ViewDimension`, pewien blok pamięci struktury (owa unia) jest interpretowany jako podstruktura specyficzna dla określonego typu zasobu. Jest to chyba wygodniejsze rozwiązanie niż nasuwające się na myśl alternatywy:

- osobna struktura dla każdego typu zasobu

[2]Nie ma czegoś takiego, są to zwykłe struktury. Chodzi tylko o odróżnienie od głównej struktury.

- nieużywanie grupowania na podstruktury i wrzucenie wszystkich reprezentowanych przez nie opcji do jednego wora

7.3 Widoki zasobów shaderów (SRV)

Widoki zasobów shaderów umożliwiają dostęp *tylko do odczytu* do zasobów buforowych i teksturowych we wszystkich programowalnych stadiach potoku (shaderach). W aplikacji widoki zasobów shaderów to obiekty `ID3D11ShaderResourceView`.

SRV tworzymy przy pomocy metody `CreateShaderRessourceView(...)` i struktury konfiguracyjnej `D3D11_SHADER_RESOURCE_VIEW_DESC` - jej opisaną wcześniej składową `ViewDimension` ustawiamy na:

- `D3D11_SRV_DIMENSION_BUFFER` dla buforów
- `D3D11_SRV_DIMENSION_TEXTURE1D` dla tekstur 1D
- `D3D11_SRV_DIMENSION_TEXTURE2D` dla tekstur 2D
- `D3D11_SRV_DIMENSION_TEXTURE3D` dla tekstur 3D

a następnie ustawiamy pola podstruktury o nazwie, odpowiednio:

- `Buffer`
- `Texture1D`
- `Texture2D`
- `Texture3D`

Możliwości jest *dużo* więcej, ale nie ma sensu sobie nimi w tej chwili zaprzątać głowy.

Rozdział 8

Zasoby po raz trzeci

8.1 Tekstury!

Podstawowe typy buforów za nami (oj tak, jest ich więcej), czas przyjrzeć się teksturom. Przybliżać ich zastosowań chyba nie trzeba, jest ich multum - od tradycyjnych (kolorowanie modeli) po bardzo zaskakujące.

My zaczniemy od tego najbardziej naturalnego przypadku: wykorzystanie tekstur w programowalnych stadiach przetwarzania (shaderach) jako pamięć *read-only*. Czemu nie używać w tym celu buforów? Przecież choćby znane nam już bufory stałych to też pamięć *read-only* dostępna dla shaderów. Odpowiedź jest prosta: tekstury z definicji lepiej od buforów spisują się w przypadku samplowania/filtrowania[1] i *indeksowanego* dostępu, a przecież właśnie to mamy na myśli, mówiąc o najbardziej naturalnym ich zastosowaniu[2].

W najogólniejszym podziale wyróżniamy:

- tekstury jednowymiarowe (`ID3D11Texture1D`)
- tekstury dwuwymiarowe (`ID3D11Texture2D`)
- tekstury trójwymiarowe (`ID3D11Texture3D`)

Tekstury, podobnie jak zasoby buforowe, dziedziczą po `ID3D11Resource`.

[1] Chodzi o interpolację wartości/koloru kilku sąsiadujących elementów.

[2] Bufory stałych są za to lepiej przystosowane do częstszego zapisu przez CPU, oferują dostęp o niższej latencji pamięci pod warunkiem respektowania określonych zasad użytkowania.

8.2 Tekstury - aplikacja

8.2.1 Tworzenie

By utworzyć zasób teksturowy, musimy wywołać jedną z trzech metod urządzenia o prawie identycznych nazwach:

- tekstury jednowymiarowe: `CreateTexture1D(...)`
- tekstury dwuwymiarowe: `CreateTexture2D(...)`
- tekstury trójwymiarowe: `CreateTexture3D(...)`

Jak wszystkie metody tworzenia zasobów, `CreateTexture*D` przyjmuje trzy argumenty (zobacz 3.3).

Tekstury, ze względu na wspominaną wyżej mnogość zastosowań, możemy wiązać do potoku na różne sposoby, ale dla wybranego na początek sposobu ich wykorzystania mamy na szczęście tylko jedną opcję - umożliwienie wiązania tekstury jako zasób shadera (ang. *Shader Resource*) przy użyciu flagi `D3D11_BIND_SHADER_RESOURCE`[3].

Jak zwykle - pozostałe dwa argumenty metod `CreateTexture*D` to struktura konfigurująca (którą w tym wprowadzeniu pomijamy, by zachować jego ogólność/przejrzystość) i dane potrzebne do zainicjowania zasobu. Dla buforów stałych problemu nie było w ogóle, bo i tak aktualizowaliśmy ich zawartość, korzystając z `Map` i `Unmap`; w przypadku buforów wierzchołków/indeksów wypełnimy po prostu odpowiednie tablice i przekazywaliśmy adres pierwszego ich elementu; dla tekstur można by zrobić podobnie, ale byłoby to dużo bardziej frustrujące zajęcie, dlatego w trakcie laboratorium w celu wczytania tekstury z pliku skorzystamy z procedur biblioteki pomocniczej `D3DX`.

8.2.2 Wiązanie do stadiów potoku

Tekstury, w przeciwieństwie do poznanych do tej pory rodzajów buforów, nigdy nie są wiązane do potoku bezpośrednio - zawsze za pośrednictwem widoku zasobu. Jako że omawiamy na razie tylko przypadek użycia tekstur jako *shader resource*, skorzystamy z widoku typu *shader resource view*. Metody kontekstu urządzenia służące do wiązania zasobów/widoków tego typu to:

- `VSSetShaderResources(...)`

[3]Nie jest to flaga/sposób wiązania zarezerwowany dla tekstur!

- `HSSetShaderResources(...)`
- `DSSetShaderResources(...)`
- `GSSetShaderResources(...)`
- `PSSetShaderResources(...)`

Ponownie, nazwy mówią same za siebie.

8.2.3 Obiekty stanu samplera

Trzeba niestety w tym miejscu wprowadzić dodatkowe pojęcie - obiekt stanu samplera (ang. *Sampler State Object*). Zacznijmy od tego, jak taki obiekt ma się do pojęć, z którymi spotkaliśmy się wcześniej: zajrzyj do podrozdziału 2.3, gdzie mówiliśmy o obiektach stanu wykorzystywanych do konfigurowania nieprogramowalnych stadiów potoku. Rola obiektu stanu samplera, sposób jego tworzenia i użycia są podobne, tylko że taki obiekt nie konfiguruje konkretnego stadium. Obiekt stanu samplera zawiera wszystkie parametry potrzebne, gdy shader próbkuje/sampluje teksturę, czyli przede wszystkim:

- sposób adresowania
- metoda filtrowania

Podobnie do innych obiektów stanu, *sampler state object* tworzymy przy pomocy metody urządzenia `CreateSamplerState(...)`, podając dwa parametry:

- wskaźnik do struktury konfigurującej (wejście)
- adres wskaźnika do `ID3D11SamplerState` (wyjście)

Jako że obiekty stanu samplera nie konfigurują żadnego stadium, przypisujemy je do potoku na zasadzie podobnej do poznanych już tekstur czy buforów stałych:

- `VSSetSamplers(...)`
- `HSSetSamplers(...)`
- `DSSetSamplers(...)`
- `GSSetSamplers(...)`
- `PSSetSamplers(...)`

Również liczba mnoga (*...Samplers*) ujawnia, że nie jest to konfiguracja stadium jako takiego, a raczej wyposażenie go w zgrupowane pule parametrów, które może wykorzystać lub nie. Do każdego programowalnego stadium możemy jednocześnie przywiązać maksymalnie *szesnaście* obiektów stanu samplera.

8.3 Tekstury - HLSL

8.3.1 Tekstury

Tekstury do kodu shaderów wprowadzamy przy pomocy słowa kluczowego `Texture*D`, po którym następuje nazwa tekstury:

```
Texture1D SepiaLookupTex;
Texture2D MeshTex;
```

Listing 8.1: Tekstury w HLSL-u

Podobnie jak w przypadku buforów stałych możemy jawnie przypisać teksturę do określonego rejestru, co pozwoli na łatwe identyfikowanie jej z poziomu aplikacji (albo możemy zdać się na mechanizm refleksji shaderów).

8.3.2 Obiekty stanu samplera

Samplery do kodu shaderów wprowadzamy przy pomocy słowa kluczowego `SamplerState`, po którym następuje nazwa danej konfiguracji samplera:

```
SamplerState BasicSamplerCfg;
```

Listing 8.2: Obiekt stanu samplera w HLSL-u

Identyfikacja rejestru obiektów stanu samplera działa tak samo jak dla tekstur i buforów stałych.

8.3.3 Próbkowanie tekstury

By próbkować teksturę najczęściej użyjemy procedury `Sample(...)`:

```
float4 color = MeshTex.Sample(BasicSamplerCfg, TexCoord);
```

Listing 8.3: Obiekt stanu samplera w HLSL-u

Jak widzisz - ta sama tekstura może korzystać z różnych konfiguracji samplera, po prostu zmieniając parametr. Może wydawać się to oczywiste, ale starsze wersje D3D nie pozwalały na taką elastyczność, co dodatkowo ograniczało ilość tekstur, których można było użyć w shaderze.

Rozdział 9

Wyjście potoku graficznego

Do tej pory szerokim łukiem omijaliśmy temat utrwalania efektów pracy potoku graficznego w danej klatce, a mianowicie zapisu:

- informacji o wynikowym kolorze dla każdego piksela do *render targetu*
- informacji o wynikowej głębokości dla każdego piksela do bufora *depth-stencil*[1]

Są to określone obszary pamięci przeznaczone do pracy potoku, czyli znane Ci już zasoby pamięciowe, a dokładniej - tekstury[2].

9.1 Render target

9.1.1 Swap chain

Swap chain, zwany *czasem* buforem wymiany, to zbiór bloków pamięci wykorzystywanych do przechowywania rezultatów renderingu, zanim zostaną one zaprezentowane. Bloki te (nazywane niestety buforami) zawierają zależną od formatu informację na temat koloru każdego piksela i dzielą się na tzw. bufory tylne (ang, *back buffers*) i bufor przedni (ang. *front buffer*). Zawartość bufora przedniego to dokładnie to, co widzimy na wyjściu/monitorze, natomiast wyniki obecnej pracy potoku generującego kolejną klatkę są umieszczane w jednym z buforów tylnych. Typową konfiguracją jest jeden back buffer i jeden front buffer (tzw. podwójne buforowanie), dlatego też, bez większego uszczerbku dla wiedzy czytelnika, w dalszej części tego rozdziału zakładamy taką właśnie konfigurację. W D3D11 łańcuch wymiany reprezentowany jest przez `IDXGISwapChain`.

[1] Sam *stencil* sobie na razie pominiemy.

[2] A dlaczego nie zasoby buforowe? Jak myślisz?

9.1.2 Render target

Render target to po prostu obszar pamięci, do którego w trakcie renderingu danej klatki potok graficzny zapisuje informacje o kolorze dla każdego piksela. Standardowo będzie to bufor tylny, który *jest za nas tworzony* na skutek wywołania funkcji `D3D11CreateDeviceAndSwapChain(...)`. Dzięki niej za jednym zamachem uzyskujemy dostęp do urządzenia, jego kontekstu oraz *swap chain*-u (który reprezentuje jeden lub więcej buforów tylnych).

Nie wywołujemy jawnie żadnej z metod `CreateTexture*D()`, więc by móc w aplikacji operować jakimś obiektem, który reprezentuje bufor tylny, użyjemy `IDXGISwapChain::GetBuffer(...)`, co pozwali nam na zainicjowanie wskaźnika na `ID3D11Texture*D`.

9.1.3 Render Target View

Jako że mamy do czynienia z teksturą, musimy skorzystać z odpowiedniego widoku zasobu, by móc przekazać ją do dyspozycji potoku graficznego. Skoro chcemy, by mógł on do tego zasobu renderować, skorzystamy z metody urządzenia `CreateRenderTargetView(...)`, przekazując jako parametr teksturę, reprezentującą bufor tylny łańcucha wymiany, a odbierając zainicjowany wskaźnik na `ID3D11RenderTargetView`.

Utworzony w ten sposób RTV wiążemy do potoku (a dokładniej do stadium OM) przy pomocy metody kontekstu urządzenia `OMSetRenderTargets`. Gdy uznamy, że praca potoku w danej klatce jest skończona, wywołujemy metodę `IDXGISwapChain::Present(...)`, która przekazuje dane z bufora tylnego do przedniego.

9.1.4 Render target $\neq$ bufor tylny

Jednoznaczne utożsamianie bufora tylnego łańcucha wymiany z render targetem jest sporym uproszczeniem. W określonych okolicznościach pożądany może być rendering do tekstury utworzonej całkowicie niezależnie od łańcucha wymiany (przy pomocy `CreateTexture*D` z odpowiednią flagą wiązania). Na bazie takiej tekstury również tworzymy RTV i wiążemy go do OM. Ta technika jest przydatne, jeżeli np.:

- chcemy uzyskać widoki z różnych kamer (wszelkie lustra, fajnym przykładem są portale w grach takich jak *Portal*)
- implementujemy *deferred rendering*

- tworzymy efekty post-procesowe

Choć można sobie wyobrazić sytuację, gdy zatrudniamy potok do wygenerowania obrazu bez wykorzystywania go do jakiegokolwiek renderingu, standardowo taki render target po wypełnieniu koniec końców jest w jakiś sposób wykorzystywany (zwykle jako tekstura w shaderach) w trakcie renderingu do bufora tylnego, bo tylko z niego metodą `Present(...)` możemy *przesłać* obraz do bufora przedniego.

9.2 Bufor depth-stencil

Bufor depth-stencil to popularny bufor głębokości czy też bufor Z. Ten zasób tworzy samodzielnie przy pomocy `CreateTexture*D` z odpowiednią flagą wiązania.

9.3 Depth-stencil View

I znowu, jako że mamy do czynienia z teksturą, musimy dla niej utworzyć odpowiedni widok zasobu, tym razem będzie to `ID3D11DepthStencilView`, do którego wskaźnik zainicjujemy przy pomocy metody urządzenia `CreateDepthStencilView(...)`. Wiązanie do potoku załatwiamy łącznie z render targetem/targetami, wywołując `OMSetRenderTargets(...)`.

Rozdział 10

Shadery od strony aplikacji

Kolejnym tematem, który do tej pory został przemilczany, jest zarządzanie shaderami od strony aplikacji. W standardowych scenariuszach użycia nie jest to szczególnie złożone zagadnienie, sprowadza się do trzech kroków:

- kompilacja kodu shadera
- utworzenie obiektu reprezentującego shader (ang. *Shader Object*)
- przywiązanie obiektu shadera do potoku

10.1 Kompilacja

Kompilację funkcji shaderów możemy przeprowadzić na podstawie pliku/bufora tekstowego w trakcie działania aplikacji korzystając z funkcji, odpowiednio:

- `D3DCompile(...)`
- `D3DCompileFromFile(...)`

Poza samym kodem źródłowym musimy przekazać przede wszystkim informacje takie jak nazwa procedury głównej shadera (ang. *shader entry point*), profil modelu shadera, jaki chcemy utworzyć[1] (ang. *target profile*). W rezultacie uruchomienia tych funkcji otrzymujemy bajtkod shadera w formie obiektu `D3DBlob`.

Alternatywnie, możemy prekompilować shadery *offline*, przy pomocy dostarczanego z SDK DirectX-a programiku `fxc`, i w trakcie działania aplikacji

[1] Najlepiej wyjaśni to chyba przykład: Shader Model 3.0 udostępnia profile vs_3_0, ps_3_0.

wczytywać już tylko gotowy bajtkod. `fxc` może być też wykorzystywany do niskopoziomowej optymalizacji/debugowania/diagnostyki dzięki możliwości wygenerowania względnie czytelnego kodu w asemblerze graficznym.

10.2 Tworzenie obiektu shadera

Gdy dysponujemy już bajtkodem, utworzenie obiektu shadera sprowadza się do wywołania jednej z następujących metod urządzenia:

- `CreateVertexShader(...)`
- `CreateHullShader(...)`
- `CreateDomainShader(...)`
- `CreateGeometryShader(...)`
- `CreatePixelShader(...)`

Dobre wieści są takie, że z reguły nie musimy już dostarczyć żadnych parametrów poza buforem z bajtkodem i jego rozmiarem.

10.3 Wiązanie do potoku

Proces wiązania do potoku również nie wymaga od programisty alokowania dużej ilości swojej cennej pamięci długotrwałej, wystarczy użyć jednej z metod kontekstu urządzenia:

- `VSSetShader(...)`
- `HSSetShader(...)`
- `DSSetShader(...)`
- `GSSetShader(...)`
- `PSSetShader(...)`

W znakomitej większości przypadków jedynym parametrem będzie odpowiedni obiekt shadera.

Rozdział 11

Geometry shader

Geometry shader, podobnie jak vertex shader, operuje na wierzchołkach czyli na danych potoku przed rasteryzacją. W przeciwieństwie jednak do VS, którego przetwarzanie ogranicza się do jednego wierzchołka, GS otrzymuje do przetworzenia w każdym wywołaniu tablicę zawierającą wszystkie wierzchołki tworzące określony typ prymitywu geometrycznego:

- dla punktu jeden wierzchołek
- dla odcinka/linii dwa wierzchołki
- dla trójkąta trzy wierzchołki
- dla linii z sąsiedztwem cztery wierzchołki[1]
- dla trójkątów z sąsiedztwem sześć wierzchołków

W obrębie programu cieniowania geometrii mamy więc zupełnie inny widok danych niż w przypadku VS. Ponadto, GS jest wyposażony w kilka możliwości, których nie można spotkać w innych miejscach potoku: z poziomu procedur GS możemy (i) usuwać i/lub dodawać wierzchołki/prymitywy; (ii) dokonywać konwersji typu prymitywów; (iii) w połączeniu ze stadium Stream Output możemy skierować strumień geometrii *do* buforów wierzchołków. Po zakończeniu pracy, GS umieszcza zaakceptowane/wygenerowane wierzchołki do wyjściowego strumienia:

- punktów
- linii/odcinków

[1] Nie wiesz, o co chodzi z tym sąsiedztwem? Na razie się tym nie przejmuj.

- trójkątów[2]

11.1 Geometry shader - aplikacja

Od strony aplikacji zarządzanie geometry shaderem nie różni się niczym od znanych już vertex i pixel shaderów, wystarczy więc zajrzeć do rozdziału 10. Opisany w poprzednich rozdziałach schemat wiązania do programowalnych stadiów buforów stałych, SRV i konfiguracji samplerów również pozostaje dla GS bez zmian.

11.2 Geometry shader - HLSL

Najlepiej omówić to zagadnienie na prostych przykładach.

```
[maxvertexcount(3)]
void GSMain(triangle PSInput input[3], inout TriangleStream<
   PSInput> outputStream)
{
  for(int i = 0; i < 3; i++)
    outputStream.Append(input[i]);

}
```

Listing 11.1: Prosty (forwardujący) geometry shader

```
[maxvertexcount(1)]
void GSMain(line PSInput input[2], inout PointStream<PSInput>
    outputStream)
{
  for(int i = 0; i < 2; i++)
    outputStream.Append(input[i]);

}
```

Listing 11.2: Prosty (konwertujący) geometry shader

Popatrz na geometry shadery na listingach 11.1 i 11.2: pierwszy z nich przekazuje po prostu wejście na wyjście, nie wprowadzając żadnych modyfikacji, a drugi zmienia jedynie typ prymitywu geometrycznego z linii na punkty.

Idąc od góry mamy:

[2]Generowane strumienie linii i trójkątów interpretowane są nie jako listy prymitywów, ale ich *strip*-y, czyli takie ciągi wierzchołków, w których każdy kolejny element tworzy nowy prymityw.

- `maxvertexcount(n)` - gdzie `n` to maksymalna liczba wierzchołków, którą może wygenerować pojedyncze wywołanie danego GS[3]
- nagłówek procedury głównej GS z dwoma parametrami: tablicą wierzchołków wejściowych poprzedzoną deklaracją ich typu i obiektem strumienia wierzchołków wyjściowych odpowiedniego typu; wyniki przetwarzanie zapisywane są do strumienia wyjściowego, więc nie trzeba nic zwracać (`void`)
- ciało procedury, w którym użyta jest metoda strumienia wyjściowego `Append`, której działania tłumaczyć chyba nie trzeba

Jeżeli jakiegoś wierzchołka nie wprowadzimy do strumienia wyjściowego, nie zostanie on przekazany do rasteryzacji; jeżeli utworzymy nowy wierzchołek i wprowadzimy go do strumienia wyjściowego - staje się on dla kolejnych stadiów pełnoprawnym elementem przetwarzanej geometrii.

[3]Pojedyncze wywołanie GS nie może wygenerować więcej niż 1024 wartości skalarnych.

Rozdział 12

Teselacja w potoku Direct3D 11

Teselcja w potoku Direct3D 11 oznacza trzy nowe stadia przetwarzania (Rys. 12.1), dwa programowalne:

- Hull Shader
- Domain Shader

i jedno sztywne:

- Tessellator (po polsku - teselator)

Niezależnie jednak od tego, czy korzystamy ze stadiów teselacyjnych czy nie, potok przetwarzania niezmiennie zaczyna swoją pracę w Input Assemblerze i Vertex Shaderze, dlatego na początku wyjaśnijmy, jak z nich korzystać, by skutecznie zaprząc do pracy teselację.

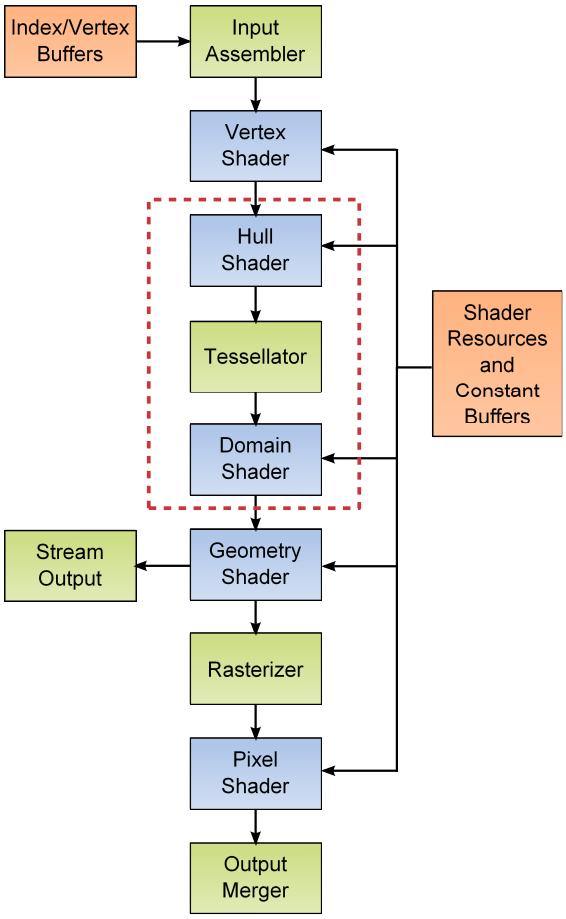
12.0.1 Input Assembler

Jeżeli chcemy korzystać ze stadiów teselacyjnych, musimy używać odpowiednich rodzajów topologii prymitywów, mianowicie:

`D3D11_PRIMITIVE_TOPOLOGY_n_CONTROL_POINT_PATCHLIST`,

gdzie `n` to liczba od `1` do `32` włącznie. Oznacza to, że możemy wykorzystywać płaty[1] (ang. *patch*) opisane nawet 32 punktami kontrolnymi/wierzchołkami. Poza tym musimy oczywiście dostarczyć punkty kontrolne w buforach wierzchołków i indeksów, ustawić pożądaną deklarację zawartości wierzchołka, a praca IA będzie tradycyjnie polegała na formowaniu wierzchołków i aranżowaniu ich w prymitywy (w tym przypadku - w płaty).

[1] Płat parametryczny to uogólnienie krzywej parametrycznej.



Rysunek 12.1: Potok graficzny Direct3D 11 z zaznaczonymi stadiami teselacyjnymi

12.0.2 Vertex Shader

Z niskopoziomowego punktu widzenia, praca vertex shadera nie różni się w przypadku przetwarzania punktów kontrolnych od znanego już nam przetwarzania wierzchołków. Przypomnij sobie jednak, że tradycyjnie rola tego stadium polegała na dostarczaniu zrzutowanych na płaszczyznę wierzchołków tworzących geometrię (przekazywanych w dół potoku jako `SV_Position`). Biorąc pod uwagę fakt, że wierzchołki zostaną dopiero wygenerowane w dalszych stadiach potoku, ta rola zanika. Praktyczne implementacje technik teselacyjnych wykorzystują VS do skinningu w animacji szkieletowej, przesuwania/obracania geometrii w świecie - krótko mówiąc do operacji, które można wykonać na ogólnej/abstrakcyjnej reprezentacji geometrii (przed teselacją) w celu ograniczenia ilości obliczeń (mniej elementów do przetworze-

nia).

12.0.3 Hull Shader

Przetworzone w VS punkty kontrolne i złożone z nich płaty trafiają teraz do stadia programowalnego zwanego Hull Shader. Składają się na niego dwie funkcje:

- funkcja samego shadera wywoływana dla każdego punktu kontrolnego
- funkcja wywoływana dla każdego płatu (zwana *Patch Constant Function*).

Poza dostarczeniem ciał tych funkcji, programista musi także ustawić pulę atrybutów[2] konfigurujących stadia teselacyjne. Atrybuty te muszą być umieszczone przed kodem shadera, gdyż są używane również do walidacji jego zawartości.

Dla każdego płatu

Funkcja wywoływana dla każdego płatu ma za zadanie ustalenie poziomu jego teselacji (czyli podziału który dokona się w teselatorze). Osobno określamy poziom teselacji dla krawędzi i wnętrza płatu. Dzięki tej funkcji możliwe jest zaimplementowanie algorytmów dynamicznego LOD (poziom szczegółowości)[3] na GPU.

Dla każdego punktu kontrolnego

Funkcja samego shadera jest wołana dla każdego *wyjściowego* punktu kontrolnego danego płatu[4]. Ilość ta może pokrywać się z ilością punktów przychodzących w ramach płatu z Vertex Shadera, może być większa lub mniejsza. Rola hull shadera będzie więc polegać na różnego rodzaju konwersjach dokonywanych na punktach kontrolnych.

[2]Składniowo wygląda to tak samo jak ustawianie atrybutu maxvertexcount dla Geometry Shadera.

[3]Dalsze od kamery lub po prostu mniejsze płaty zwykle wymagają mniej geometrii.

[4]Ilość tych punktów definiujemy przy pomocy jednego z wyżej wspomnianych atrybutów, ale (w przeciwieństwie do Geometry Shadera) nie generujemy ich potem sami: tutaj zamawiamy ich określoną ilość, po czym operaujemy na każdym z osobna w programie shadera.

12.0.4 Teselator

Kolejny krok to sam teselator, stadium nieprogramowalne i niekonfigurowalne z poziomu aplikacji, o jego pracy decydują w pełni atrybuty stadium hull shadera i poziomy teselacji definiowane przez jego *patch constant function*. Co istotne - teselator *nie widzi* punktów kontrolnych, operuje zupełnie niezależnie od nich. Rola tego stadium polega na generowaniu lokalizacji, które w Domain Shaderze posłużą do próbkowania powierzchni, której fragment opisywany jest przez dany płat.

Próbkowania? Posłużmy się znowu niżej-wymiarową analogią do krzywych. Musisz sobie zdać sprawę, że to, co nazywamy *rysowaniem krzywej* na podstawie punktów kontrolnych, sprowadza się do próbkowania faktycznej krzywej w określonych punktach na jej długości zmapowanej do przedziału $[0, 1]$. Rola teselatora polega właśnie na generowaniu tych współrzędnych do próbkowania z gęstością wynikającą ze zdefiniowanych poziomów teselacji.

Co istotne, teselator działa w zupełnym oderwaniu od faktycznej geometrii, z czego wynika, że trafniejszym jest nazywanie go stadium *generującym* (a nie *przetwarzającym*).

12.0.5 Domain Shader

Ostatnim ze stadiów teselacyjnych jest Domain Shader, który pracuje w oparciu o punkty kontrolne wygenerowane w Hull Shaderze oraz współrzędne próbkowania pochodzące z teselatora. DS jest wywoływany raz dla każdego punktu próbkowania wygenerowanego przez teselator, a jego zadaniem jest utworzenie na jego bazie wierzchołka, odpowiednio próbkując powierzchnię opisaną przez punkty kontrolne.

Rozdział 13

Wygładzanie przy pomocy teselacji

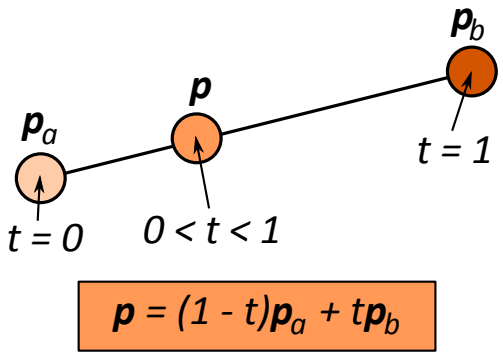
W poprzednim rozdziale zahaczyliśmy o temat wykorzystywania teselacji do wygładzania, ale ćwiczenie na laboratorium ograniczyło się do korzystania z zagęszczonej geometrii w celu wytłoczenia w niej mapy zniekształceń (ang. *displacement map*). Miałeś w takim razie do czynienia z płaską teselacją, która na każdym trójkącie (czyli na jego płaszczyźnie) opisującym powierzchnie modelu tworzy dodatkowe wierzchołki, żeby (mówiąc kolokwialnie) teksele mapy zniekształceń miały jakieś wierzchołki do przesuwania.

W przypadku teselacji wygładzającej, modyfikujemy (w miarę potrzeb) płaszczyzny opisujące wejściowy model, zamieniając je na powierzchnie niepłaskie w celu wygładzenia[1].

Zaproponowano już sporo metod teselacji z wygładzaniem, o różnym poziomie jakości, złożoności obliczeniowej (i realizowalności na GPU) oraz oczywiście trudności implementacji. Dzisiaj zapoznasz się (a w trakcie laboratorium zaimplementujesz) chyba najmniej skomplikowaną i jedną z tańszych obliczeniowo spośród nich - tzw. *teselację Phonga* opisaną w **(Boubekeur, T. i Alexa, M.: Phong Tessellation, SIGGRAPH Asia 2008)**.

Rozdział ten jest o tyle nietypowy, że praktycznie nie wspomina o Direct3D, a skupia się na opisie samej metody teselacji Phonga. Instrukcja laboratoryjna będzie skonstruowana komplementarnie: skupi się na API, nie opisując już samej metody. Do rzeczy.

[1] Koniec końców i tak mamy do czynienia z liniowym przybliżeniem krzywizn, ale mniej dostrzegalnym i bliższym wyglądowi modelu zamierzonemu przez grafika/modelarza.



Rysunek 13.1: Interpolacja liniowa

13.1 Interpolacja barycentryczna

Współrzędne i interpolacja barycentryczna są wygodną metodą opisu położenia punktu na powierzchni trójkąta[2]. Żeby ją dobrze pojąć, przyjrzyjmy się najpierw zwykłej interpolacji liniowej wzdłuż odcinka opisanego przez dwa wierzchołki, $\boldsymbol{p}_a$ i $\boldsymbol{p}_b$ (Rys. 13.1). Położenie punktu $\boldsymbol{p}$ jest dane jego znormalizowaną odległością t od $\boldsymbol{p}_a$; jednocześnie $1 - t$ oznacza oczywiście jego znormalizowaną odległość od $\boldsymbol{p}_b$. Patrząc na wzór z Rys. 13.1 można dojść do wniosku, że t opisuje procentową zawartość $\boldsymbol{p}_b$ w $\boldsymbol{p}$, a $1 - t$ - procentową zawartość $\boldsymbol{p}_a$ w $\boldsymbol{p}$.

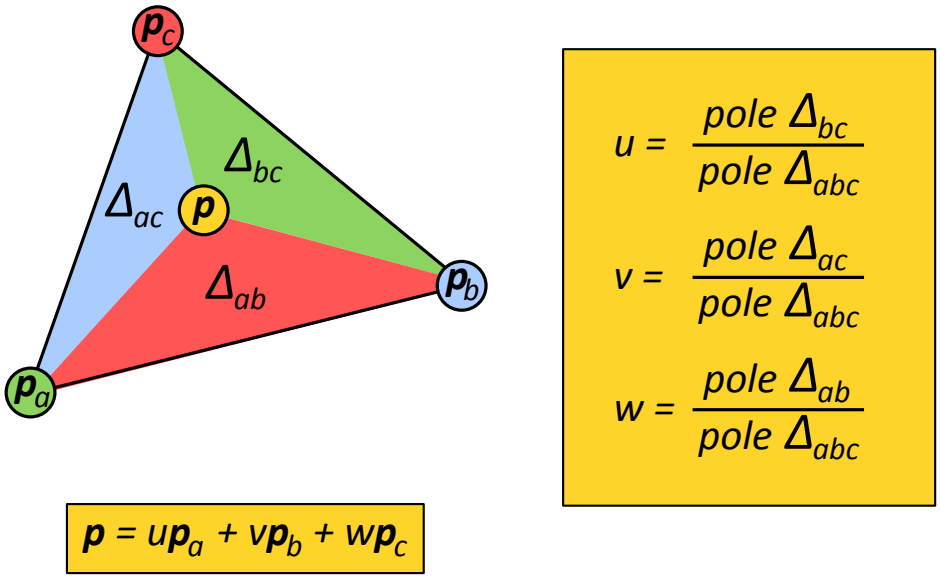
W przypadku trójkąta (opisanego przez wierzchołki trzy $\boldsymbol{p}_a$, $\boldsymbol{p}_b$ i $\boldsymbol{p}_c$) i współrzędnych barycentrycznych rozwiązanie jest podobne, z tym że zamiast odległości stosujemy pola powierzchni (Rys. 13.2): u opisuje procentową zawartość $\boldsymbol{p}_a$ w $\boldsymbol{p}$, itd.

W przeciwieństwie do interpolacji liniowej, dla której mieliśmy minimalną reprezentacją przy pomocy jednego parametru t, tutaj mamy do czynienia z reprezentacją nadmiarową/redundantną, gdyż (dzięki normalizacji) każdy z trzech parametrów można obliczyć na podstawie dwóch pozostałych, np. $w = 1 - u - v$.

13.2 Cieniowanie Phonga

Społeczność graficzna zawdzięcza doktoratowi pana Bui Tuong Phonga dwa znane pod jego nazwiskiem rozwiązania:

[2]Zagadnie jest szersze, ale my ograniczamy się do trójkątów.



Rysunek 13.2: Interpolacja barycentryczna

- model oświetlenia Phonga (ang. *Phong illumination model*)
- cieniowanie Phonga (ang. *Phong shading*)

Pierwsze z nich powinno Ci być już znane z GK i/lub ZTPGK (model oświetlenia lokalnego z rozbiciem na składowe *ambient*, *diffuse* i *specular*).

Drugie z wyżej wymienionych rozwiązań Phonga dotyczy oświetlenia/kolorowania poszczególnych fragmentów/pikseli renderowanej sceny: zamiast obliczać kolory wierzchołków w Vertex Shaderze, a potem interpolować je na powierzchni trójkątów (cieniowanie Gourauda), interpolujemy wektory normalne wierzchołków i dopiero w Pixel Shaderze wykonujemy obliczenia koloru dla konkretnych fragmentów/pikseli[3].

Względnie wysoki koszt obliczeniowy cieniowania Phonga nie jest już dzisiaj wielkim problemem, stało się ono więc (w różnych odmianach) pewnym bazowym standardem, dzięki któremu możemy uzyskać złudzenie gładkich powierzchni, mimo tego że opisująca je geometria jest w rzeczywistości dosyć prosta i *kanciasta*.

[3]Model oświetlenia Phonga i cieniowanie Phonga mogą, ale nie muszą być stosowane razem/jednocześnie.

13.2.1 Artefakty

Cieniowanie Phonga prowadzi do wizualnego dysonansu[4] szczególnie widocznego, gdy patrzymy na kontury modelu, na których nie da się przy pomocy sprytnego modelu oświetlenia zamaskować prawdziwej natury geometrii. By temu zaradzić, trzeba by ją jakoś ulepszyć/zmodyfikować jeszcze przed rasteryzacją i tu właśnie wracamy do tematu technik wygładzania przy pomocy teselacji.

13.3 Teselacja (wygładzanie) Phonga

Deklarowanym założeniem autorów teselacji Phonga było utworzenie takiej techniki wygładzania geometrii, która byłaby prosta w implementacji, niedroga obliczeniowo, możliwa do bezbolesnego wprowadzenia do istniejących rendererów i działałaby na istniejących modelach[5].

13.3.1 Płaska teselacja trójkątów

Płaska teselacja trójkątów opiera się na interpolacji barycentrycznej czyli:

$$\boldsymbol{p} = (u\ v\ w)(\boldsymbol{p}_i\ \boldsymbol{p}_j\ \boldsymbol{p}_k)^T = u\boldsymbol{p}_i + v\boldsymbol{p}_j + w\boldsymbol{p}_k,$$

gdzie $\boldsymbol{p}$ to zinterpolowany/wygenerowany wierzchołek, $\boldsymbol{p}_i, \boldsymbol{p}_j, \boldsymbol{p}_k$ wierzchołki oryginalne, a u, v, w to współrzędne barycentryczne.

Cieniowanie Phonga sprowadza się do bardzo podobnej operacji:

$$\boldsymbol{n} = (u\ v\ w)(\boldsymbol{n}_i\ \boldsymbol{n}_j\ \boldsymbol{n}_k)^T = u\boldsymbol{n}_i + v\boldsymbol{n}_j + w\boldsymbol{n}_k,$$

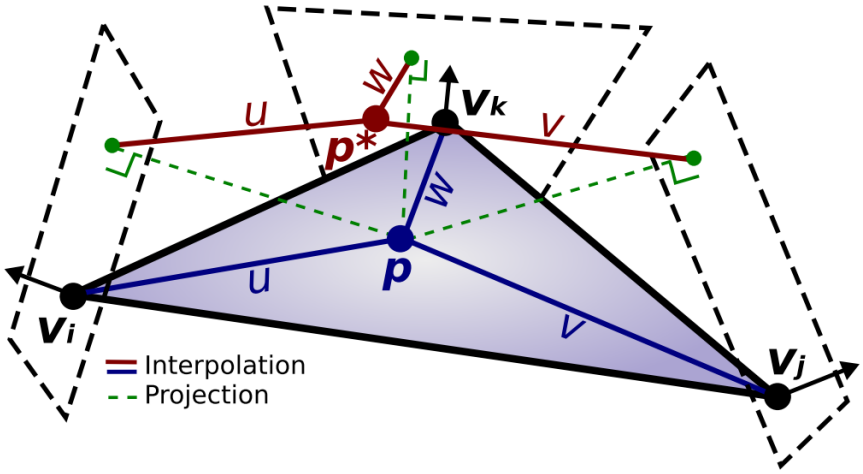
tyle że następuje po niej jeszcze normalizacja (czyli dzielnie $\boldsymbol{n}$ przez jego długość).

13.3.2 Teselacja Phonga

Koncepcja teselacji Phonga polega na takim wygięciu powierzchni trójkąta wejściowej geometrii, by jej przebieg odpowiadał zinterpolowanym na jej przestrzeni wektorom normalnym. Uzyskiwane jest w następujących krokach:

[4] Podobnie jak wiele innych *oszczędnych* technik stosowanych w interaktywnej grafice komputerowej

[5] Niektóre techniki teselacji wymagają modyfikacji modeli, co oznacza dodatkową pracę grafików/modelarzy, a to z kolei trwa i kosztuje.



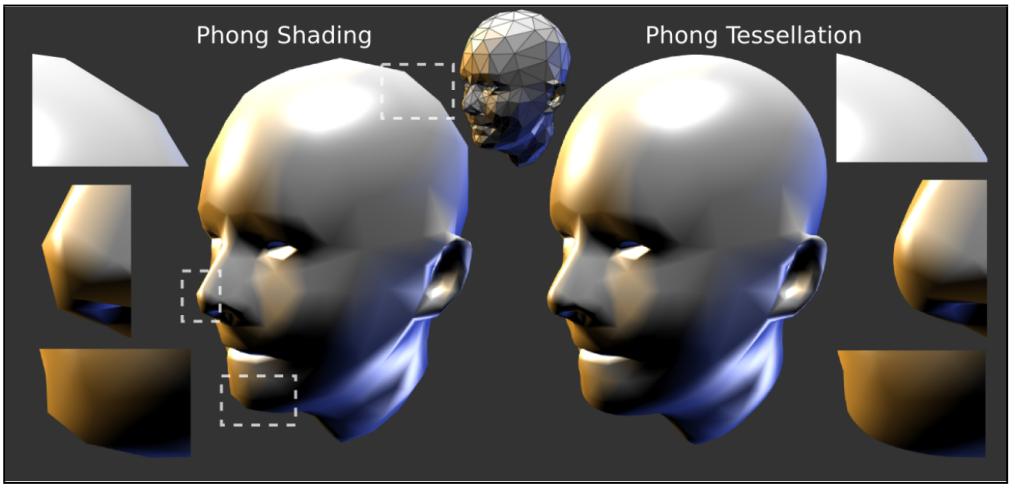
Rysunek 13.3: Rzutowanie wygenerowanego wierzchołka $\boldsymbol{p}$ na płaszczyzny styczne [Tamy Boubekeur i Marc Alexa]

1. płaska teselacja trójkątów
2. dla każdego powstałego wierzchołka $\boldsymbol{p}$
   - rzutowanie prostopadłe $\boldsymbol{p}$ kolejno na trzy płaszczyzny styczne do powierzchni modelu w oryginalnych wierzchołkach (Rys. 13.3)
   - interpolacja barycentryczna tych trzech rzutów prostopadłych
3. cieniowanie Phonga[6]

Pewnie się teraz zastanawiasz, o co chodzi z tymi *płaszczyznami stycznymi do powierzchni modelu w oryginalnych wierzchołkach*. Koncepcyjnie jest to dość proste, trzeba tylko wziąć pod uwagę kilka rzeczy:

- każdy wektor w przestrzeni 3D może posłużyć do określenia orientacji pewnej (prostopadłej do niego) płaszczyzny w tej przestrzeni, ale takich płaszczyzn jest nieskończenie wiele (równoległych do siebie); jeżeli jednak określimy dodatkowo punkt, przez który płaszczyzna ma przechodzić, będziemy mieli pewność, że mówimy o jakiejś konkretnej płaszczyźnie
- zakładamy, że dla każdego z wierzchołków jakiegoś wejściowego trójkąta modelu mamy określona normalną; jest więc punkt (wierzchołek) i wektor → mamy konkretną płaszczyznę

[6]Nie jest to już co prawda element teselacji, ale można potraktować ten punkt jako integralną część procesu wygładzania.



Rysunek 13.4: Teselacja Phonga w akcji - w naturalny sposób uzupełnia cieniowanie Phonga [Tamy Boubekeur i Marc Alexa]

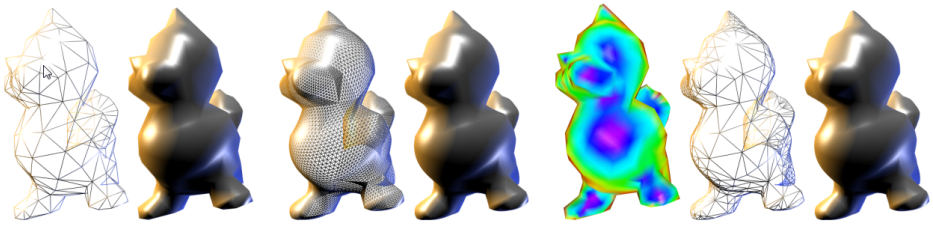
- idea wektorów normalnych przekazywanych w ramach geometrii modelu jest taka, że są one prostopadłe do jego powierzchni w punkcie reprezentowanym przez powiązany z daną normalną wierzchołek
- kierunkiem prostopadłym do normalnego jest kierunek styczny, jeżeli więc w każdym wierzchołku wejściowego trójkąta wyobrazimy sobie prostopadłą do jego wektora normalnego płaszczyznę, możemy nazwać ją płaszczyzną styczną do powierzchni modelu w danym wierzchołku

To wszystko oznacza również, że powierzchnia modelu jest przybliżona określoną ilością płaszczyzn: płaszczyzny te byłyby tożsame z powierzchniami trójkątów siatki, ale że traktujemy je (trójkąty) poniekąd oddzielnie od normalnych (normalne są określane *per-vertex*), mamy więc dokładniejszy-/gęstszy opis powierzchni modelu! Teselacja Phonga interpoluje informację o orientacji powierzchni modelu w generowanych wierzchołkach na podstawie orientacji danej dla wierzchołków wejściowych poprzez ich normalne.

Rzut prostopadły $\pi(\boldsymbol{q})$ punktu $\boldsymbol{q}$ na płaszczyznę prostopadłą do (znormalizowanego) wektora $\boldsymbol{n}$, przechodzącą przez punkt $\boldsymbol{p}$ przeprowadzimy w następujący sposób:

$$\pi(\boldsymbol{q}) = \boldsymbol{q} + \boldsymbol{n}((\boldsymbol{p} - \boldsymbol{q}) \cdot \boldsymbol{n})) = \boldsymbol{q} + \boldsymbol{n}((\boldsymbol{p} - \boldsymbol{q})\boldsymbol{n}^T)),$$

gdzie $\cdot$ oznacza iloczyn skalarny.



(a) Input (b) Uniform (c) Adaptive View-Dependent

Rysunek 13.5: Porównanie modelu po samym cieniowaniu Phonga (a), jednolitej teselacji Phonga (b) i teselacji Phonga skupionej na konturach (c) [Tamy Boubekeur i Marc Alexa]

13.3.3 Optymalizacja

Jako że niedoskonałości cieniowania Phonga są widoczne głównie na konturach modeli, dość naturalną optymalizacją teselacji Phonga jest prowadzenie jej tylko na trójkątach, które w danym momencie tworzą kontur. Sami autorzy proponują prostą miarę siły teselacji, polegającą na ocenie kąta między kierunkiem patrzenia kamery, a normalnymi wierzchołków tworzących dany trójkąt: im kąt bliższy 90°, tym teselacja powinna być silniejsza, by uzyskać lepsze wygładzenie (Rys. 13.5).

Źródła

- Zink, J., Pettineo, M., Hoxley, J.: Practical Rendering & Computation with Direct3D 11. CRC Press, 2011.
- Sherrod, A., Jones, W.: Beginning DirectX 11 Game Programming. Course Technology, 2011.
- `http://msdn.microsoft.com/`