Programowanie w API Graficznych

Laboratorium

DirectX 12 – ćwiczenie 1

1 Cele ćwiczenia

Celem ćwiczenia jest zapoznanie się z procesem konfigurowania podstawowego potoku graficznego w DirectX i następnie wykorzystania go do narysowania trójkąta w oknie aplikacji. Cały proces jest bardzo złożony, praktycznie każdy błąd w konfiguracji uniemożliwi korzystania z DirectX, powinieneś także znać interfejs programistyczny WinAPI i wiedzieć jak za jego pomocą utworzyć okno aplikacji w systemie Windows. Dlatego twoim zadaniem będzie uzupełnianie luk w istniejącej aplikacji, dzięki czemu zrozumiesz jak tworzy się kluczowe obiekty i zasoby DirectX i będziesz w przyszłości w stanie zrobić to samodzielnie. W trakcie wykonywania zadania nauczysz się także pisać, kompilować proste programy cieniujące oraz tworzyć bufory wierzchołków.

1.1 Struktura projektu

Wszystkie zadania należy wykonać w załączonej solucji pwag01.sln. Solucja do poprawnego działania wymaga co najmniej IDE Visual Studio 2022 oraz systemu operacyjnego Windows 10. W skład solucji wchodzi tylko jeden projekt– pwag01.vcxproj, generuje on plik wykonywalny(.exe), projekt został skonfigurowany tak aby uruchomić się jako aplikacja okienkowa Windows (flaga /SUBSYTEM:WINDOWS). Projekt został dostosowany do uruchomienia aplikacji korzystających z DirectX 12, wszystkie wymagane pliki nagłówkowe i statyczne biblioteki DirectX zostały już włączone do projektu, można to sprawdzić w ustawieniach projektu:

- pliki nagłówkowe: Properties->Configuration Properties->VC++ Directories->Include Directories
- lokalizacja bibliotek: Properties->Configuration Properties->VC++ Directories->Library Directories
- biblioteki: Properties->Linker->Input->Additional Dependencies

Wykonywanie aplikacji rozpoczyna się od funkcji `int WINAPI WinMain()`, możesz ją znaleść w pliku main.cpp. W pierwszej kolejności tworzony jest obiekt klasy System, odpowiada on za stworzenie i wyświetlenie głównego okna aplikacji oraz obsługę pętli komunikatów, wykorzystywane jest do tego niskopoziomowe api systemu windows WinAPI. Klasa System została zadeklarowana w plikach System.h i System.cpp, przyjrzyj się funkcji `System::Initialize`(), pozwala ona określić rozdzielczość wyświetlanego obrazu oraz wybrać czy aplikacja zostanie uruchomiona w oknie lub na pełnym ekranie. Aplikację można zamknąć w standardowy sposób bądź klikając klawisz escape.

Kod odpowiedzialny za rysowanie obrazu został umieszczony w osobnej klasie – RenderWidget (deklaracja w plikach RenderWidget.h i RenderWidget.cpp), zostanie ona

szczegółowo omówiona w kolejnym rozdziale. Kolejnym istotnym plikem jest shader.fx, znajduje się w nim kod programów cieniujących (ang. shaders), plik tez zostanie skompilowany przez DirectX na etapie uruchamiania aplikacji. W projekcie znajdują się też dwa pomocnicze pliki, w pierwszym z nich – common.h dołączono pliki nagłówkowe directx oraz stl. W drugim – GeometryHelper znajdują się funkcje pomocnicze, w tym funkcja odpowiedzialna za kompilację programów cieniujących oraz funkcja odpowiedzialna za generowanie geometry trójkąta.

Ostatnim, niewspomnianym jeszcze plikiem jest d3dx12.h, został on napisany przez programistów Mircrosoftu i zawiera wiele przydatnych funkcji pomocniczych. Nie jest on wymagany do uruchomienia aplikacji DirectX i nie jest częścią windowsowego SDK, dlatego należy go osobno pobrać z repozytorium DirectX-Header: https://github.com/microsoft/DirectX-Headers

2 Zadania

2.1 Konfiguaracja DirectX

Proces konfiguracji potoku renderującego można prześledzić wewnątrz funkcji `RenderWidget::Initialize`. W pierwszym kroku aktywowana jest warstwa debugowa DirectX, służy do tego funkcja `ID3D12Debug::EnableDebugLayer`, jak pewnie zauważyłeś, funkcja ta jest umieszczona w bloku makra `if defined(DEBUG)`, a więc kod ten zostanie dołączony do kodu źródłowego jedynie w konfiguracji DEBUG. Warstwa debugowa DirectX zapewni dodatkowe, bardziej szczegółowe informacje o błędach w konfiguracji DirectX, możesz je znaleźć w zakładce Output w IDE Visual Studio, poniżej umieszczono przykładowy komunikat o błędzie. W tym przypadku, bez aktywnej warstwy debugowej DirectX zwrócił by jedynie numer błędu – E_INVALIDARG.

```
D3D12 ERROR: ID3D12Device::CreateCommittedResource: D3D12_RESOURCE_DESC::Alignment is invalid. The value is 421.
```

Wydruk 1. Przykładowy błąd wygenerowany przez warstwę debugową DirectX

Następnie wywoływana jest funkcja `CreateDXDeviceAndFactory()`, odpowiada ona za utworzenie dwóch kluczowych obiektów:

- `IDXGIFactory4 m_dxgiFactory` – odpowiada za utworzenie obiektu IDXGISwapChain, ID3D12Device oraz udostępnia informacje o sprzęcie (karty graficzne, monitory, dostępna pamięć gpu, itp..) jaki jest dostępny na danym komputerze
- `ID3D12Device m_dxDevice` – umożliwia utworzenie wszystkich pozostałych obiektów

niezbędnych do skonfigurowania potoku renderującego

W kolejnym kroku (funkcja `RenderWidget::CreateCommandObjects()`) tworzone są obiekty odpowiedzialne za kontrolę listy rozkazów, w DirectX 12 rozkazy można umieścić we wspólnej liście, a następnie przesłać je razem do kraty graficznej i wykonać. W poprzednich wersjach DirectX rozkazy były indywidualnie przesyłane do karty graficznej i wykonywane, powodowało to dodatkowy narzut czasowy i negatywnie wpływało na czas generowania obrazu.

Funkcja `CraeteSwapChain()` odpowiada za utworzenie obiektu `IDXGISwapChain m_swapChain`. Łańcuch wymiany (ang. swap chain) odpowiada za obsługę buforów w których znajduje się wyrendereowany obraz. Typowa aplikacja 3D powinna posiadać przynajmniej dwa bufory – bufor tylni (ang. back buffer) i bufor przedni (ang. front buffer), w buforze przednim powinien znajdować się aktualnie wyświetlany obraz, a w buforze tylnym aktualnie rysowany obraz. W momencie gdy rysowanie obrazu zostanie zakończone to oba bufory zamieniają się miejscami.

Kolejna funkcja – `CreateDescriptorHeaps()`, jak wskazuje nazwa, odpowiada za utworzenie obiektu *descriptor heap*, obiekt ten służy do przechowywania widoków zasobów (*resource view*), w przypadku tej aplikacji korzystamy jedynie z dwóch widoków – *RenderTargetView* oraz *DepthStencilView.*

Zaimplementowanie pozostałych funkcji – `BuildRootSignature()`, `CompileShaders()`, `CrateGraphicPipeline()`, `LoadGeometry()` oraz `CreateGraphicPipeline()` jest celem tego ćwiczenia.

Do tej pory wszystkie utworzone obiekty i zasoby były statyczne - nie zmieniały swojego stanu ponieważ nie były w żaden sposób powiązane z rozmiarem okna aplikacji. Pozostałe zasoby, jak na przykład bufor głębokości, bufor obrazu czy *viewport*, których parametry są bezpośrednio zależne od rozmiaru wyświetlanego obrazu są tworzone w funkcji `RenderWidget::Resize`(), funkcja ta jest też wywoływana za każdym razem gdy zmianie ulegnie rozmiar okna aplikacji. Wewnątrz tej funkcji znajdziesz, kolejno, funkcję `ReziseSwapChain()`, która zmienia rozmiar obu buforów łańcucha wymiany. Funkcje `CreateRenderTargetView()` i `CreateDepthStencilView()`, które tworzą obiekt *render target* (jest to zasób odpowiedzialny za renderowanie obrazu), bufor głębokości oraz bufor szablonowy. W ostatnim kroku aktualizowany jest *viewport* (obiekt reprezentuje prostokąt na który rzutowana jest scena 3D) oraz *scisor rectangle* (prostokąt określający obszar przycinania).

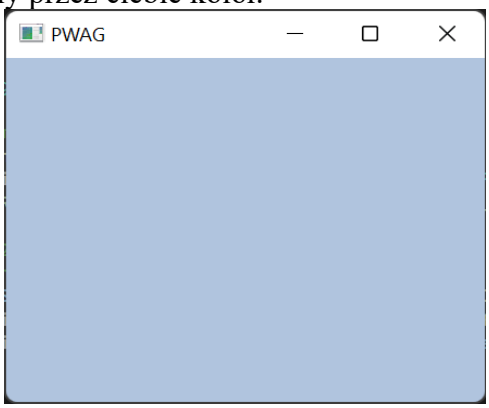
Trzecią istotną dla działania aplikacji funkcją jest `RenderWidget::Draw()`, jest ona odpowiedzialna za podpięcie wszystkich wymaganych zasobów do potoku renderujacego, narysowanie obrazu i wyświetlenie go w oknie aplikacji. Na tym etapie w funkcji znajduje się

jedynie kod inicjalizujący obiekt listy komend oraz kod odpowiedzialny za obsługę łańcucha wymiany , twoim zadaniem będzie napisanie brakującej funkcjonalności.

W pierwszym kroku powinieneś sprawdzić kiedy dokładnie jest wywoływana funkcja `Draw()` oraz co decyduje o tym kiedy oraz jak często obraz będzie renderowany.

2.1.1 Inicjalizacja

Proces renderowania należy rozpocząć od „wyczyszczenia" wcześniej wykorzystanych zasobów, przejdź więc do funkcji `RenderWidget::Draw()`, jest ona odpowiedzialna za rysowanie każdej klatki i tam właśnie napiszesz odpowiedzialny za to kod. W aktualnie programowanej aplikacji dysponujemy dwoma buforami obrazu, każdą klatkę chcemy rysować na jednolitym tle, a więc będziesz musiał wypełnić bufor tylny jednolitym kolorem (np. `DirectX::Colors::LightSteelBlue`), służy do tego funkcja `ClearRenderTargetView()`, jednym z argumentów wymaganych przez tą funkcję jest tzw. widok celu renderowania(ang. Render-target view) bufora tylnego, możesz go uzyskać za pomocą funkcji `GetCurrentBackBufferView`(). W DirectX obiekty typu widok reprezentują zasoby karty graficznej. Podobnie postąp z buforem głębokości oraz buforem szablonów (`ClearDepthStencilView()`), wykorzystaj widok bufora szablonu i głębokości - `depthStencilViewHandle`. Oba bufory należy wypełnić wartościami z przedziału od 0 do 1. W przypadku bufora głębokości wartość 0 reprezentuje piksele umieszczone z przodu, natomiast wartość 1 obiekty umieszczone na samym końcu bufora(maksymalnie oddalone), jakimi wartościami powinien być wypełniony bufor głębokości na początku? Jakimi wartościami powinien zostać wstępnie bufor szablonowy? Po uruchomieniu aplikacji powinieneś zauważyć, że zmienił się kolor tła na wybrany przez ciebie kolor.



2.1.2 Tworzenie obiektu root signature i programów cieniujących

W kolejnym kroku utworzysz obiekt *root signature*, który potrzebny będzie do narysowania trójkąta w ekranie aplikacji, będziesz musiał także napisać programy cieniujące wierzchołki i piksele, następnie skompilować je i podpiąć pod potok renderujący.

- Root signature – określa jakie zasoby zostaną przypięte do potoku renderującego przed

wywołanie funkcji renderującej obraz. *Root signature* musi być kompatybilny z programami cieniującymi jakie zostaną przypisane do potoku renderującego (musi on dostarczyć wszystkie zasoby wymagane przez program cieniujący). *Root signature* jest reprezentowany w Direct3D poprzez interfejs `ID3D12RootSingature`, kod odpowiedzialny za jego utworzenie został umieszczony w funkcji `BuildRootSignature()`. Funkcja ta jest niekompletna, twoim zadaniem jest poprawienie tego kodu, w pierwszej kolejności powinieneś poprawić kod tworzący obiekt `CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc`, jest on odpowiedzialny za zdefiniowanie tego jakie zasoby zostaną przypisane do obiektu *root signature*. W przypadku tej aplikacji nie tworzymy żadnych dodatkowych zasobów, a więc będziesz musiał utworzyć pusty root signature. Następnie odkomentuj i popraw funkcję `CreateRootSignature()`, która utworzy wspomniany już obiekt *root signature*. Na koniec uruchom aplikację i upewnij się, że uruchamia się ona bez żadnych błędów.

- Vertex shader -program cieniujący wierzchołki. Celem obecnego zadania jest poprawne skonfigurowanie DirectX i wyświetlenie zapisanego w buforze wierzchołków trójkąta, a więc program cieniujący wierzchołki powinien tylko przekazać wierzchołki zapisane w w buforze wierzchołków do kolejnego etapu potoku renderującego. Kod programów cieniujących znajduje się w pliku shader.fx, twoim zadaniem jest napisanie kodu programu cieniującego wierzchołki, który tylko skopiuje wierzchołki z argumentu wejściowego (`VertexIn vin`) do kolejnego etapu i skompilowanie go (patrz funkcja `CompileShaders()`).

- Pixel shader – program cieniujący piksele. Napisz prosty program cieniujący piksele, który zwraca kolor różny od koloru tła, tak aby rysowany trójkąt był widoczny

- Pipeline State – ostatnim krokiem jest utworzenie obiektu `ID3D12PipelineState m_pipelineState`. Odpowiada on za konfigurację potoku renderującego – za jego pomocą podepniesz programy cieniujące, określisz typ geometrii w buforze wierzchołków, *input layout* i wiele innych kluczowych parametrów potoku. Przejdź do funkcji `CreateGraphicPipeline`(), to tutaj tworzony jest obiekt stanu potoku renderującego, kod podobnie jak w poprzednich zadaniach jest niekompletny. Popraw go uzupełniając obiekt *inputLaout* tak aby odpowiadał on strukturze danych przekazywanych do programu cieniującego wierzchołki. Następnie zmień typ geometrii na listę trójkątów (`PrimitiveTopologyType`). Na koniec odkomentuj kod przypisujący programy cieniujące i funkcję `CreateGraphicsPipelineState()`. Jeżeli poprawnie wykonałeś powyższe kroki to program powinien uruchomić się bez żadnych błędów.

2.2 Potok renderujący

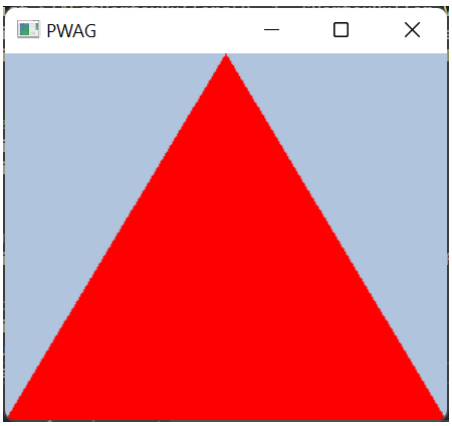
2.2.1 Bufor wierzchołków i bufor indeksów

Bufor indeksów i bufor wierzchołków jest tworzony wewnątrz funkcji `LoadGeometry()`.

Otwórz tę funkcję, jak zapewne zauważyłeś jest niekompletna, w pierwszym kroku wywoływana jest funkcja tworząca geometrię trójkąta – `Geometry::CreateTriangleGeometry()`, ta funkcja także jest niekompletna, brakuje w niej współrzędnych wierzchołków trójkąta, uzupełnij je. W jakich przedziałach mogą znajdować się poszczególne współrzędne? Następnie odkomentuj funkcje `LoadVertexBuffer()` oraz `LoadIndexBuffer()`, dla odmiany te funkcje są kompletne i nie trzeba w nich nic uzupełnić, otwórz je i przeanalizuj ich kod, zapoznaj się z procesem ładowania wierzchołków i indeksów do zasobów karty graficznej. Czy wiesz który obiekt w kodzie aplikacji reprezentuje bufor wierzchołków i bufor indeksów?

2.2.2 Rysowanie trójkąta w oknie aplikacji

Na tym etapie potok renderujący jest już poprawnie skompilowany i utworzone zostały wszystkie potrzebne do rysowani zasoby. Otwórz więc funkcję Draw(), jest ona niekompletna, twoim zadaniem będzie podpięcie zasobów do potoku graficznego i finalnie narysowanie trójkąta w oknie aplikacji. Potok renderujący wymaga obiektu root signature, utworzyłeś już go wcześniej, teraz wystraczy tylko przypisać go do potoku za pomocą funkcji `SetGraphicRootSignature()`. Następnie skonfiguruj etap *InputAssembly*, musisz podpiąć utworzone wcześniej bufory wierzchołków (znajduje się on w obiekcie `VertexBuffer`) i indeksów (w obiekcie `IndexBuffer`) oraz określić strukturę geometrii w buforze wierzchołków. Następnie odkomentuj i uzupełnij funkcje odpowiedzialne za konfigurację etapu rasteryzacji. Ostatni etap potoku renderującego – *output merger stage* jest już skonfigurowany, postaraj się zapoznać z działaniem funkcji `OMSetRenderTargets()`. Ostatnim krokiem jest uzupełnienie funkcji `DrawIndexedInstanced()`, uruchomi ona potok renderujący i jeżeli wszystkie poprzednie kroki wykonałeś poprawnie to stworzona przez ciebie aplikacja narysuje trójkąt.



Zapewne zauważyłeś, że wszystkie napisane przez ciebie rozkazy nie są od razu wykonywane lecz zapisywane w liście rozkazów, aby je wykonać należy należy wykonać funkcję

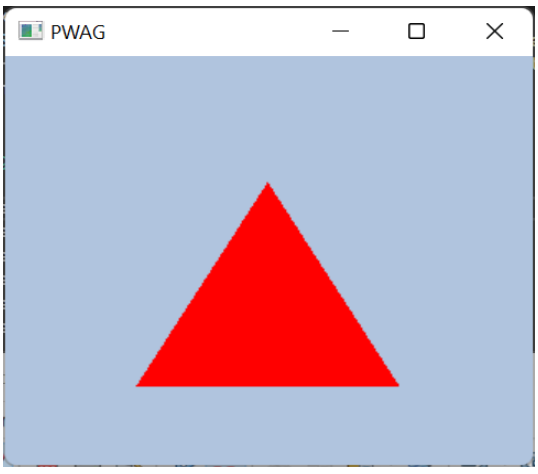
`ExecuteCommandList()`. Wywołana przez ciebie funkcja rysuje trójkąt w buforze tylnym, aby wyświetlić rysowany obraz należy zamienić bufory miejscami, kod ten został już za ciebie napisany i nie wymaga poprawek. Warto jednak abyś się z nim zaznajomił. Zwróć uwagę na funkcję `m_swapChain->Present()`, odpowiada ona za wyświetlenie zawartości bufora tylnego na ekranie, w kolejnym kroku zamieniamy jest bufor przedni na tylny.

```
        // Indicate a state transition on the resource usage.
        m_commandList->ResourceBarrier(1,
&CD3DX12_RESOURCE_BARRIER::Transition(GetCurrentBackBuffer(),
                D3D12_RESOURCE_STATE_RENDER_TARGET, D3D12_RESOURCE_STATE_PRESENT));

        ExecuteCommandList();

        // swap the back and front buffers
        ThrowIfFailed(m_swapChain->Present(0, 0));
        m_currBackBuffer = (m_currBackBuffer + 1) % SwapChainBufferCount;

        FlushCommandQueue();
```

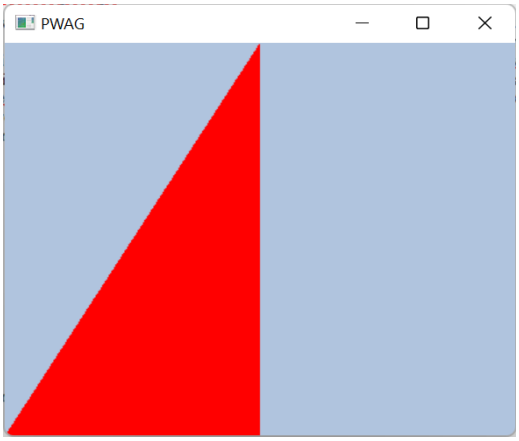
Wydruk 2. Zamiana bufora przedniego i tylnego

2.3 Rasteryzacja

W tym zadaniu wrócisz do konfiguracji etapu rasteryzacji, nie był on do tej pory szczegółowo omówiony i nie musiałeś poprawiać kodu odpowiedzialnego za ten etap. Teraz, gdy już udało ci się narysować obraz za pomocą DirectX warto abyś zobaczył jak modyfikacje niektórych parametrów etapu rasteryzacji wpływają na generowany obraz. Otwórz funkcję `UpdateViewport()`, wewnątrz znajdziesz kod odpowiedzialny za aktualizację wspomnianych we wstępie do zadania obiektów `m_screenViewport` oraz `m_scissorRect`. Określają one obszar rasteryzacji i przycinania geometrii, sprawdź jak zmiany współrzędnych tych obiektów wpływają na generowany obraz. Zmodyfikuj współrzędne obiektu `m_screenViewport` tak aby zmienić rozmiar oraz położenie trójkąta:

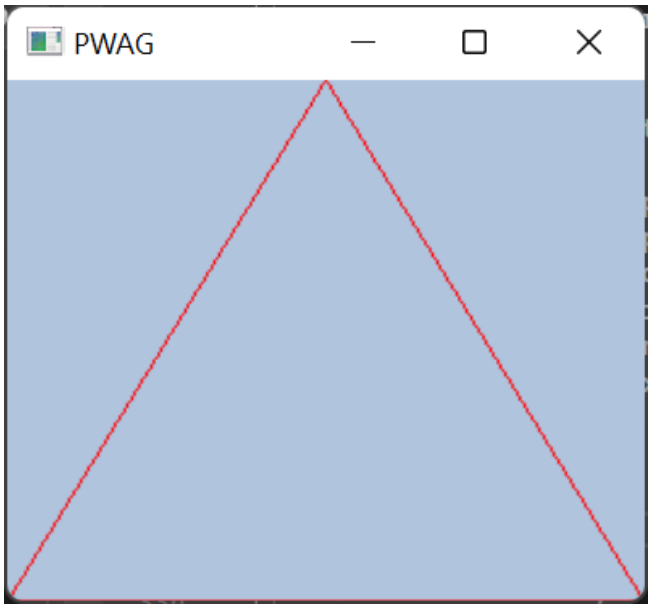


Zmodyfikuj współrzędne `m_scissorRect` tak aby przyciąć trójkąt:



W jaki sposób obiekty typu `D3D12_VIEWPORT` oraz `D3D12_RECT` wpływają na generowany obraz, jaka jest ich rola w potoku graficznym?

Przejdź do funkcji `CreateGraphicPipeline()` i zmodyfikuj strukturę `RasterizerState` tak aby wyświetlała jedynie obrys trójkąta:



W tej samej strukturze ustaw zmienną `FrontCounterClockwise` na *true*. Aplikacja przestanie wyświetlać trójkąt, dlaczego? Zmodyfikuj kolejność wierzchołków w buforze indeksów tak aby trójkąt znów był widoczny, jaki wpływ ma kolejność wierzchołków na widoczność obiektów 3D? Ustaw `FrontCounterClockwise` ponownie na *false*, trójkąt nie powinien być widoczny. Następnie ustaw zmienną `CullMode` na `D3D12_CULL_MODE_NONE`, dlaczego trójkąt jest znowu widoczny? Jak na proces rasteryzacji wpływają zmienne `FillMode`, `CullMode` oraz `FrontCounterClockwise`?