Wprowadzenie do Direct3D

Paweł Aszklar
P.Aszklar@mini.pw.edu.pl

Warszawa, 28 lutego 2019

Część I
Inicjalizacja Direct3D

Direct3D jest biblioteką opartą o interfejsy COM[1]. Całe API Direct3D 11 dostępne jest (bezpośrednio lub pośrednio) poprzez interfejs `ID3D11Device`[2] (z późniejszymi zmianami w pochodnych interfejsach `ID3D11Device1`, …, `ID3D11Device5`). Nieco myląco nazwane *urządzenie* reprezentuje konkretną implementację API, która może być sprzętowa, oparta o konkretne kartę GPU (zwaną *adapterem*), lub całkowicie software'owa. Interfejs ten umożliwia, m.in. pozyskiwanie zasobów, często rezydujących na karcie graficznej (w przypadku implementacji sprzętowych).

Warto tu jeszcze wspomnieć, że możemy korzystać z API w wersji 11 nawet jeżeli dostępna w systemie implementacja nie jest z nim w pełni zgodna. Przy tworzeniu urządzenia możemy określić interesujący nas *poziom funkcjonalności* (ang. *feature level*). Podając niższy poziom, deklarujemy, że korzystać będziemy tylko z funkcjonalności, która dostępna była już w poprzedniej wersji Direct3D (np. wersji 9, 10, itp.). Dzięki temu program uruchomi się nawet na sprzęcie niezgodnym z DirectX 11, lecz próba wywołania funkcji z nowszego API zakończy się niepowodzeniem.

To czego powyższy interfejs nie obsługuje, to faktyczne renderowanie. Dostęp do tej funkcjonalności umożliwia interfejs `ID3D11DeviceContext`[3] (oraz pochodne `ID3D11DeviceContext1`, …, `ID3D11DeviceContext4` udostępniające nowsze funkcjonalności). Reprezentuje on *kontekst* potoku renderowania, tj. jego aktualne parametry, wejście/wyjście itp. Jeden kontekst, zwany *kontekstem bezpośrednim* tworzony jest razem z urządzeniem. On

[1] *Component Object Model* https://docs.microsoft.com/en-gb/windows/desktop/com/component-object-model--com--portal

[2] https://docs.microsoft.com/en-gb/windows/desktop/api/d3d11/nn-d3d11-id3d11device

[3] https://docs.microsoft.com/en-gb/windows/desktop/api/d3d11/nn-d3d11-id3d11devicecontext

jeden może faktycznie zlecać rysowanie za pomocą potoku renderowania. Można jednak tworzyć dodatkowe, tzw. *konteksty odroczone*, które mogą zbierać sekwencje komend niezależnie od innych kontekstów, które mogą być potem odtworzone na kontekście bezpośrednim. Ułatwia to znacznie implementacje aplikacji wielowątkowych.

Za wyświetlanie rezultatu renderowania w oknie aplikacji odpowiada interfejs `IDXGISwapChain`[4] (wraz z nowszymi funkcjami dostępnymi w pochodnych iterfejsach `IDXGISwapChain1`, …, `IDXGISwapChain4`). Reprezentuje on zestaw buforów obrazu, jeden *przedni* wyświetlany w oknie, oraz jeden lub więcej buforów tylnych, które mogą służyć za wyjście potoku renderowania. Sam moduł *DXGI* (ang. *DirectX Graphics Infrastructure*) nie jest bezpośrednio częścią Direct3D, lecz odpowiada za współdzielenie zasobów przez różne moduły DirectX, czy przez różne procesy (w tym menadżer okien Windows), itp.

Najprostszym sposobem uzyskania powyższych interfejsów jest funkcja `D3D11CreateDeviceAndSwapChain`[5]. Jej użycie zademonstrujemy modyfikując prostą aplikację wyświetlającą jedno puste okno WinAPI w celu przygotowania do renderowania z użyciem API Direct3D.

Wykonanie:

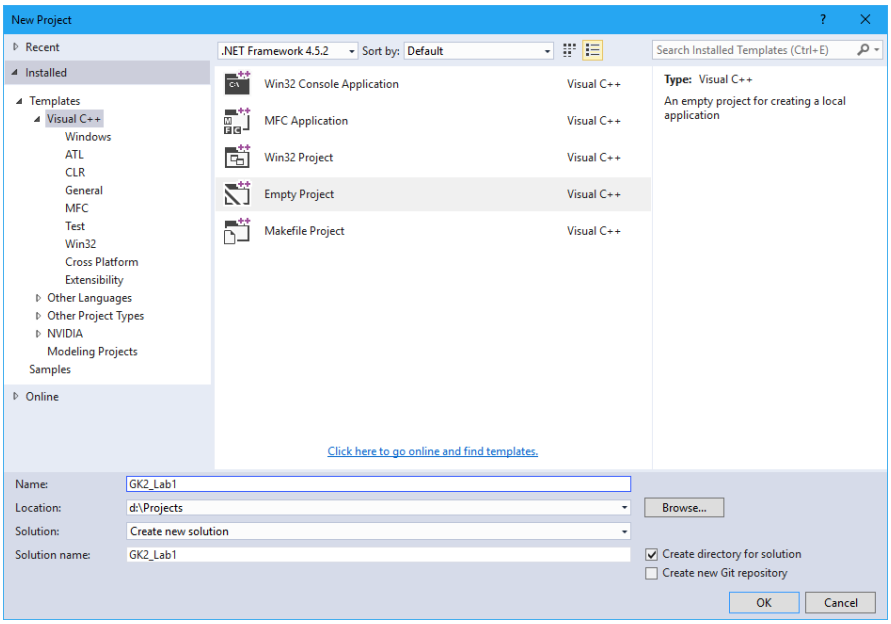
1. Pobierz archiwum z plikami źródłowymi: `main.cpp`, `dxptr.h`, `exceptions.h`, `exceptions.cpp`, `window.h`, `window.cpp`, `windowApplication.h`, `windowApplication.cpp`.

   Zapoznaj się z ich zawartością. Większość to standardowy kod odpowiedzialny za wyświetlenie i obsługę okna aplikach. Zdefiniowany jest też typ `dx_ptr`. Jest to inteligentny wskaźnik, który pomoże nam zadbać o poprawne zwolnienie interfejsów COM.

2. Utwórz nowy pusty projekt C++ w Visual Studio.

[4] https://docs.microsoft.com/en-us/windows/desktop/api/DXGI/nn-dxgi-idxgiswapchain

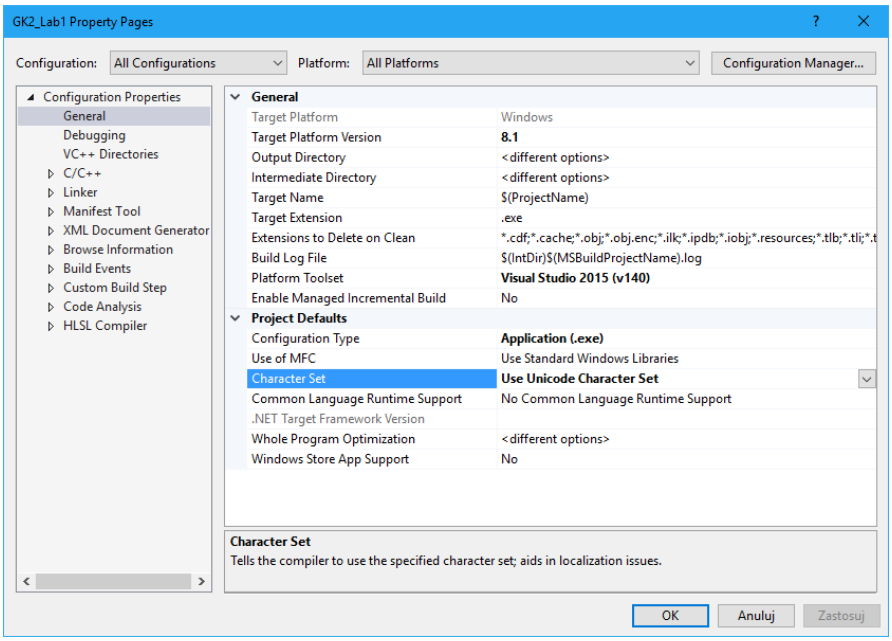
[5] https://docs.microsoft.com/en-us/windows/desktop/api/D3D11/nf-d3d11-d3d11createdeviceandswapchain



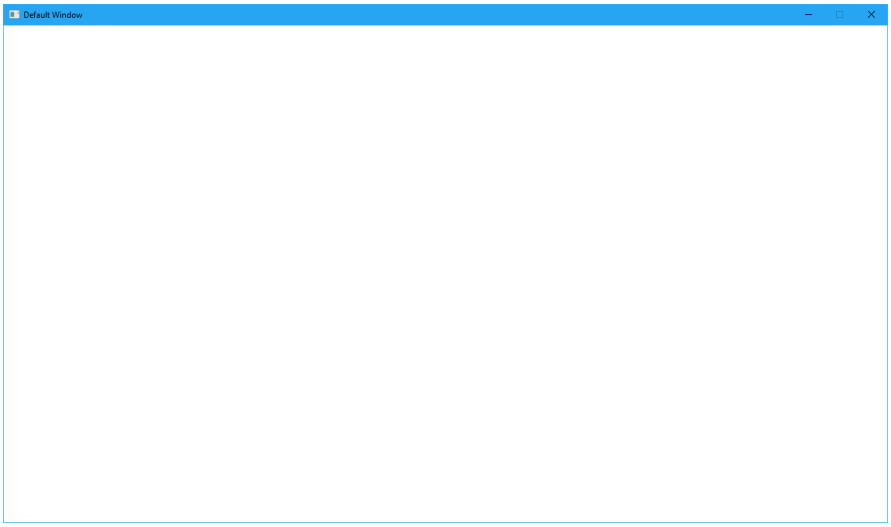
3. Rozpakuj archiwum do katalogu projektu i dodaj pliki do projektu w Visual Studio (*Project* → *Add Existing Item*).

4. Uruchom projekt. W wyniku powinno pojawić się puste białe okno programu.

5. Aby naprawić niepoprawnie wyświetlający się tytuł okna, w ustawieniach projektu (*Project* → *Properties* → *Configuration Properties* → *General*) zmień opcję *Character Set* na *Use Unicode Character Set*. Ustawienie to warto zmienić dla wszystkich konfiguracji (Debug/Release) i platform (x86/x64).



6. Po ponownym uruchomieniu okno wyglądać powinno następująco:



7. Wiele funkcji tworzących różnego rodzaju zasoby przyjmuje ich parametry w postaci struktur. Struktury te zazwyczaj mają wiele pól i niestety nie definiują sensownych konstruktorów przypisujących im domyślne, poprawne wartości. Dzięki dziedziczeniu możemy jednak takie konstruktory dodać sami.

   Dodaj do projektu plik nagłówkowy `dxStructures.h`, w którym przechowywać będziemy takie struktury. Dodaj na początek definicję struktury `SwapChainDescription`:

```
#pragma once
#include <d3d11.h>
struct SwapChainDescription : DXGI_SWAP_CHAIN_DESC {
    SwapChainDescription(HWND wndHwnd, SIZE wndSize);
};
```

Konstruktor pozwoli nam `DXGI_SWAP_CHAIN_DESC`, która opisuje parametry tworzonego interfejsu `IDXGISwapChain`.

8. Dodaj do projektu plik źródłowy **`dxStructures.cpp`** i umieść w nim definicję konstruktora `SwapChainDescription`:

```
#include "dxStructures.h"
SwapChainDescription::SwapChainDescription(HWND wndHwnd,
    SIZE wndSize) : DXGI_SWAP_CHAIN_DESC{} {
    BufferDesc.Width = wndSize.cx;
    BufferDesc.Height = wndSize.cy;
    BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
// BufferDesc.RefreshRate.Numerator = 0;
    BufferDesc.RefreshRate.Denominator = 1;
// BufferDesc.ScanlineOrdering =
//   DXGI_MODE_SCANLINE_ORDER_UNSPECIFIED /*0*/;
// BufferDesc.Scaling = DXGI_MODE_SCALING_UNSPECIFIED /*0*/;
// SampleDesc.Quality = 0;
    SampleDesc.Count = 1;
    BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
    BufferCount = 1;
    OutputWindow = wndHwnd;
    Windowed = true;
// SwapEffect = DXGI_SWAP_EFFECT_DISCARD /*0*/;
// Flags = 0;
}
```

Warto przyjrzeć się wartościom przypisywanym do niektórych pól (w komentarzach wymienione są pola, których wartość domyślna wynosi 0 i nie wymaga zmiany):

- pierwsze dwie wartości to wymiary buforów. W trybie okienkowym powinny się zgadzać z wymiarami wnętrza okna lub kontrolki w której wyświetlany będzie obraz (tzw. *client area*). W trybie okienkowym definiują pożądaną rozdzielczość ekranu.
- kolejna określa typ danych przechowywanych w każdym pikselu. Domyślnie zastosujemy 4 kanały (RGBA), a każdy kanał będzie 8-bitową liczbą całkowitą bez znaku. Końcówka `NORM` oznacza, że z punktu widzenia potoku renderowania Direct3D wartości kanału będą jednak widoczne jako liczby zmienno-przecinkowe z

zakresu $[0, 1]$ i przy każdym odczycie lub zapisie nastąpi automatyczne skalowanie (z/do zakresu $[0, 255]$) i konwersja typów.

- Pole `BufferUsage` określa w jaki sposób będziemy korzystać z buforów. Przypisana wartość mówi, że będą one użyte wyłącznie jako wyjście potoku renderowania. Jeżeli chcielibyśmy móc z nich czytać, wartość ta powinna być zmieniona.
- `BufferCount` określa ilość buforów. W trybie full-screen powinniśmy uwzględnić przedni bufor. W trybie okienkowym buforem przednim jest okno programu, więc liczymy tylko bufory tylne.
- `OutputWindow` określa uchwyt okna, bądź kontrolki w której wyświetlany będzie obraz w trybie okienkowym.
- `Windowed` określa czy pracować będziemy w trybie okienkowym (wartość `true`) czy też full-screen (wartość `false`).

9. Dodaj do projektu plik nagłówkowy **dxDevice.h**. Zgodnie z poniższym kodem zdefiniuj w nim klasę `DxDevice`. Posłuży nam ona do przechowywania trzech wspomnianych we wstępie interfejsów. Później rozszerzymy ją o funkcje pomocnicze, opakowujące wywołania analogicznych funkcji interfejsu `ID3D11Device`[6], które zapewnią nam wygodniejsze zestawy parametrów i sprawdzanie błędów.

```
#pragma once
#include "dxptr.h"
#include "window.h"
#include "dxStructures.h"
class DxDevice {
public:
    explicit DxDevice(const mini::Window& window);
    const mini::dx_ptr<ID3D11DeviceContext>& context() const
    { return m_context; }
    const mini::dx_ptr<IDXGISwapChain>& swapChain() const
    { return m_swapChain; }
    ID3D11Device* operator->() const { return m_device.get(); }
private:
    mini::dx_ptr<ID3D11Device> m_device;
    mini::dx_ptr<ID3D11DeviceContext> m_context;
    mini::dx_ptr<IDXGISwapChain> m_swapChain;
};
```

[6]Nie korzystamy z nowszych interfejsów pochodnych, gdyż ich funkcjonalność nie jest nam potrzebna, a interfejs bazowy jest wspierany przez więcej wersji systemu Windows. Ponadto ich inicjalizacja jest bardziej złożona, gdyż stworzyć należy interfejs bazowy i dopiero potem odpytać go[7]czy udostępnia interfejs pochodny.

[7]https://docs.microsoft.com/en-gb/windows/desktop/api/unknwn/nf-unknwn-iunknown-queryinterface(q_)

10. Możemy przejść do inicjalizacji potrzebnych nam interfejsów używając funkcji `D3D11CreateDeviceAndSwapChain`. Dodaj do projektu plik źródłowy `dxDevice.cpp` i zaimplementuj w nim konstruktor klasy `DxDevice`:

```
#include "dxDevice.h"
#include "exceptions.h"
DxDevice::DxDevice(const Window& window) {
    SwapChainDescription desc {
        window.getHandle(), window.getClientSize() };
    ID3D11Device* d = nullptr;
    ID3D11DeviceContext* dc = nullptr;
    IDXGISwapChain* sc = nullptr;
    auto hr = D3D11CreateDeviceAndSwapChain(nullptr,
        D3D_DRIVER_TYPE_HARDWARE, nullptr,
        D3D11_CREATE_DEVICE_DEBUG, nullptr, 0,
        D3D11_SDK_VERSION, &desc, &sc, &d, nullptr, &dc);
    m_device.reset(d);
    m_swapChain.reset(sc);
    m_context.reset(dc);
    if (FAILED(hr)) THROW_WINAPI;
}
```

Funkcja `D3D11CreateDeviceAndSwapChain` przyjmuje następujące parametry:

(a) Adapter, który użyty będzie przez implementację sprzętową. `nullptr` powoduje użycie domyślnego adaptera.

(b) Typ implementacji – tu chcemy uzyskać dostęp do sprzętowej.

(c) Wskaźnik do biblioteki implementującej API software'owe – parametr użyty jest tylko, jeżeli w poprzednim przekazaliśmy wartość `D3D_DRIVER_TYPE_SOFTWARE`.

(d) Kombinacja flag `D3D11_CREATE_DEVICE_FLAG`[8]. Wartość którą przekazaliśmy włącza warstwę debugowania Direct3D, która zawiera m.in. dodatkową kontrolę poprawności parametrów.

(e) Tablica poziomów funkcjonalności, w których nasz program może pracować. Urządzenie stworzone zostanie z najwyższym z wymienionych, jakie wspiera dostępna implementacja API. Wartość `nullptr` oznacza poziom domyślny[9].

(f) Ilość elementów w tablicy z poprzedniego parametru.

[8] https://docs.microsoft.com/en-gb/windows/desktop/api/d3d11/ne-d3d11-d3d11_create_device_flag

[9] Najwyższy dostępny poziom funkcjonalności pośród: 11.0, 10.1, 10.0, 9.3, 9.2, 9.1. Lista domyślnych wersji nie zawiera 11.1. Aby ją uzyskać należy przekazać tablicę zawierającą `D3D_FEATURE_LEVEL_11_1`.

(g) Wersja SDK użyta przy kompilacji programu.

(h) Parametry interfejsu `IDXGISwapChain`.

(i) Parametr wyjściowy interfejsu `IDXGISwapChain`.

(j) Parametr wyjściowy interfejsu `ID3D11Device`.

(k) Opcjonalny parametr wyjściowy, w którym zapisany zostanie poziom funkcjonalności, z którym urządzenie zostało utworzone (patrz. punkt 10e).

(l) Parametr wyjściowy interfejsu `ID3D11DeviceContext`.

11. Urządzenie tworzyć i przechowywać będziemy klasie aplikacji. Na potrzeby późniejszych przykładów zmodyfikować powinniśmy też główną pętlę programu. Dodaj do projektu plik nagłówkowy **dxApplication.h** i stwórz w nim klasę `DxApplication` dziedziczącą po `WindowApplication`:

```
#pragma once
#include "windowApplication.h"
#include "dxDevice.h"
class DxApplication : public mini::WindowApplication {
public:
    explicit DxApplication(HINSTANCE hInstance);
protected:
    int MainLoop() override;
private:
    void Render();
    void Update();

    DxDevice m_device;
};
```

12. Dodaj do projektu plik źródłowy **dxApplication.cpp** i zaimplementuj w nim konstruktor oraz metodę `MainLoop` zgodnie z poniższym kodem. Pozostałe metody pozostaw puste.

```
#include "dxApplication.h"
using namespace mini;
DxApplication::DxApplication(HINSTANCE hInstance)
    : WindowApplication(hInstance), m_device(m_window) { }
int DxApplication::MainLoop() {
    MSG msg{};
    do {
        if (PeekMessage(&msg, nullptr, 0,0, PM_REMOVE)) {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        else {
            Update();
```

```
14            Render();
15            m_device.swapChain()->Present(0, 0);
16        }
17    } while (msg.message != WM_QUIT);
18    return msg.wParam;
19 }
20 void DxApplication::Update() { }
21 void DxApplication::Render() { }
```

Zauważ, że w porównaniu do oryginalnej metody `MainLoop` z klasy bazowej, nie czekamy bezczynnie na wiadomości o zdarzeniach okna. Zamiast tego w przypadku braku zdarzeń, nieustannie wykonujemy kroki aktualizacji i rysowania. Przydatne będzie to, gdy wyświetlane będą animacje, które nie zależą od interakcji użytkownika.

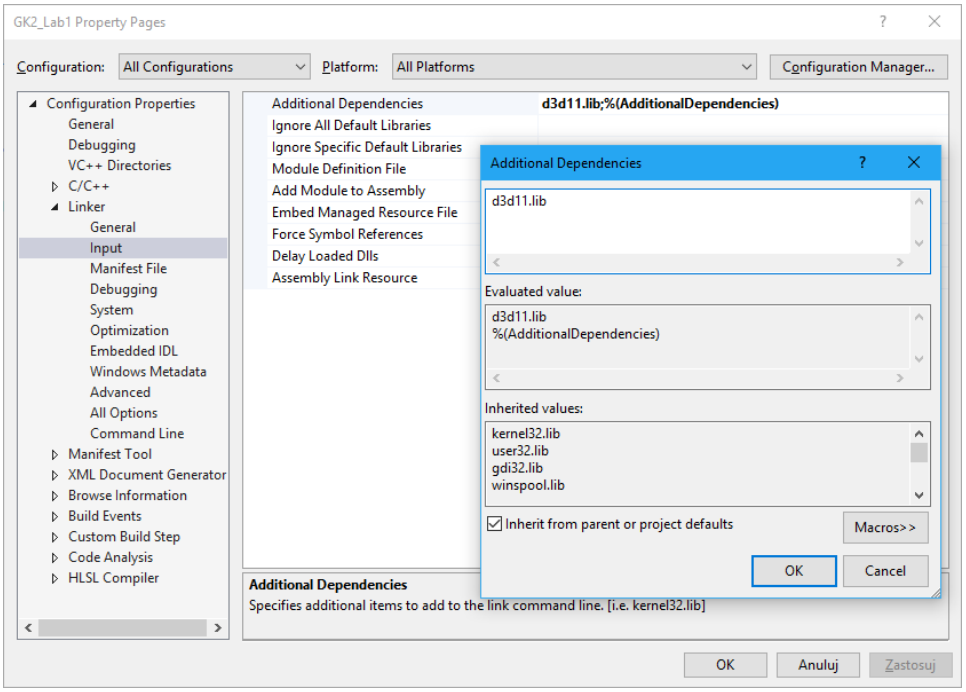
Ostatnim krokiem po rysowaniu jest przeniesienie zawartości tylnego bufora na powierzchnię okna. Służy do tego metoda `Present`[10] interfejsu `IDXGISwapChain`.

13. Ostatnia zmiana polega na podmianie funkcji `main` typu zmiennej `app` na `DxApplication`.

14. Próba kompilacji zakończy się niepowodzeniem, ze względu na brak dołączonej biblioteki. W ustawieniach projektu (*Project* → *Properties* → *Configuration Properties* → *Linker* → *Input*) należy do pola *Additional Dependencies* dodać bibliotekę `d3d11.lib`. (Rys. 1)

15. Po powyższej zmianie program powinien się uruchomić. Efektem wprowadzonych zmian jest to, że okno aplikacji jest teraz czarne.

[10] https://docs.microsoft.com/en-us/windows/desktop/api/DXGI/nf-dxgi-idxgiswapchain-present



Rysunek 1: Dołączanie biblioteki `d3d11.lib`

Część II
Czyszczenie okna

Dane wejściowe i wynikowe potoku renderowania w Direct3D przechowywane są w postaci zasobów. Istnieją ich dwa główne typu: bufory i tekstury.

Bufory to zwykłe blok pamięci. Można o nich myśleć jako o jednowymiarowych tablicach.

Tekstury są tworem bardziej złożonym. Istnieją warianty jedno-, dwulub trójwymiarowe. Wariant 3D reprezentuje pojedynczy obraz, warianty 1D i 2D mogą posiadać całą tablicę warstw (tj. obrazów o takich samych wymiarach). Dostępny jest też specjalny wariant zwany teksturą sześcienną – jest to w praktyce tekstura 2D o 6 warstwach odpowiadających ścianom sześcianu. Każdy typ tekstury ponadto zawierać może tzw. *mipmapy*, czyli wersje obrazu (lub warstw obrazu) o coraz mniejszych rozdzielczościach.[11]

Nie zawsze można użyć zasobów bezpośrednio. Niektóre zastosowania wymagają tworzenia tzw. *widoków*, które określają w jaki sposób dane zawarte w zasobie sa interpretowane. Dla przykładu kontekst oczekuje, że wyjście potoku renderowania zapisane będzie w obrazie. Widok zasobu docelowego dla tekstury pozwala nam np. wybrać poziom mipmapy zostanie w tym celu użyty. Widok może też modyfikować typ danych czytanych/zapisywanych z/do zasobu. Jeżeli stworzony został on bez typu (format z końcówką `TYPELESS`, porównaj z p. 8 w cz. I – tam podaliśmy format buforów z typem), każdy widok zasobu może określać inny, kompatybilny[12], typ.

Bufory obrazu interfejsu `IDXGISwapChain` nie są początkowo powiązane z kontekstem potoku renderowania. Aby móc narysować coś na tylnym buforze, musimy uzyskać do niego dostęp w postaci tekstury dwuwymiarowej (będzie ona miała tylko jedną warstwę i jeden poziom mipmapy) i utworzyć dla niej *widok bufora docelowego* (ang. *render target view*).

Wykonanie:

1. Dodaj do definicji klasy `DxDevice` pomocniczą metodę tworzenia widoku bufora docelowego dla tekstury dwuwymiarowej:

```
5  class DxDevice {
6  public:
       ...
13     mini::dx_ptr<ID3D11RenderTargetView>
14     CreateRenderTargetView(
15         const mini::dx_ptr<ID3D11Texture2D>& texture) const;
       ...
20 };
```

[11] Więcej o typach tekstur: https://docs.microsoft.com/en-gb/windows/desktop/direct3d11/overviews-direct3d-11-resources-textures-intro

[12] Zgadzać musi się liczba kanałów i ilość bitów na kanał.

2. Jej implementację, zilustrowaną poniżej, umieść w pliku `dxDevice.cpp`.

```
21 mini::dx_ptr<ID3D11RenderTargetView>
22 DxDevice::CreateRenderTargetView(
23     const mini::dx_ptr<ID3D11Texture2D>& texture) const {
24     ID3D11RenderTargetView* temp = nullptr;
25     auto hr = m_device->CreateRenderTargetView(texture.get(),
26         nullptr, &temp);
27     mini::dx_ptr<ID3D11RenderTargetView> result{ temp };
28     if (FAILED(hr)) THROW_WINAPI;
29     return result;
30 }
```

   Drugi parametr metody `CreateRenderTargetView` pozwala określić poziom mipmapy do którego renderowany będzie obraz. Przekazanie `nullptr` używa najwyższej rozdzielczości. W naszym przypadku bufor tylny i tak ma tylko jeden poziom mipmapy.

3. Do klasy `DxApplication` dodaj pole `m_backBuffer`, w którym zapamiętamy interfejs widoku bufora docelowego:

```
4  class DxApplication : public mini::WindowApplication {
       ...
9  private:
       ...
14     mini::dx_ptr<ID3D11RenderTargetView> m_backBuffer;
15 }
```

4. W konstruktorze `DxApplication` uzyskaj teksturę bufora tylnego i utwórz dla niej widok, który zapisany zostanie w stworzonym przed chwilą polu:

```
3  DxApplication::DxApplication(HINSTANCE hInstance)
4      : ... {
5      ID3D11Texture2D *temp = nullptr;
6      m_device.swapChain()->GetBuffer(0,
7          __uuidof(ID3D11Texture2D),
8          reinterpret_cast<void**>(&temp));
9      const dx_ptr<ID3D11Texture2D> backTexture{ temp };
10     m_backBuffer =
11         m_device.CreateRenderTargetView(backTexture);
12 }
```

5. Czyszczenie okna wykonać można w metodzie `Render`. Posłuży nam do tego metoda `ClearRenderTargetView` kontekstu:

```
22 void DxApplication::Render() {
```

```
23      const float clearColor[] = { 0.5f, 0.5f, 1.0f, 1.0f };
24      m_device.context()->ClearRenderTargetView(
25          m_backBuffer.get(), clearColor);
26  }
```

6. Po uruchomieniu program powinien wyświetlić okno wypełnione niebieskim kolorem

Część III
Rysowanie trójkąta

W celu narysowania czegokolwiek musimy:

- Powiązać bufor głębokości oraz tzw. *viewport* z obecnym *Render Target*em.
- Stworzyć bufor wierzchołków trójkąta
- Stworzyć oraz wczytać *Pixel* i *Vertex Shader*
- Stworzyć *Input Layout* wiążący atrybuty wierzchołka z wejściem *Vertex Shader*a
- Powiązać wszystko powyższe z kontekstem urządzenia
- Wywołać metodę rysującą.

1. Przed rozpoczęciem tego etapu pobierz archiwum i nadpisz pliki `dxStructures.h`, `dxStructures.cpp` i `dxDevice.h`, `dxDevice.cpp`. Archiwum zawiera cały dotychczasowy projekt, więc w razie problemów z poprzednimi częściami, można go użyć do kontynuowania zadania.

2. W klasie `DxApplication` dodać należy pola na nowe obiekty (kod 1).

```
4  class DxApplication : public mini::WindowApplication {
       ...
9  private:
       ...
15     mini::dx_ptr<ID3D11DepthStencilView> m_depthBuffer;
16     mini::dx_ptr<ID3D11Buffer> m_vertexBuffer;
17     mini::dx_ptr<ID3D11VertexShader> m_vertexShader;
18     mini::dx_ptr<ID3D11PixelShader> m_pixelShader;
19     mini::dx_ptr<ID3D11InputLayout> m_layout;
20 };
```

Listing 1: Modyfikacja definicji klasy `DxApplication`

3. Zacznijmy od bufora głębokości. W Direct3D jako bufora głębokości używamy tekstury 2D o rozdzielczości takiej samej jak *Render Target* i pewnych specyficznych parametrach.

   Parametry tekstury dwuwymiarowej przekazujemy za pomocą zmiennej typu `D3D11_TEXTURE2D_DESC`[13]. Dla ułatwienia jego inicjalizacji

[13] https://docs.microsoft.com/en-us/windows/desktop/api/d3d11/ns-d3d11-d3d11_texture2d_desc

w pliku `dxstructures.h` zdefiniowana jest dziedzicząca struktura `Texture2DDescription`. Zajrzyj do jej konstruktora i zobacz jakie wartości domyślne mają jej pola przy tworzeniu zwykłej tekstury koloru. Znajdziesz sporo podobieństw do struktury parametrów *Swap Chain*u z p. 8.

Nie wszystkie domyślne wartości są odpowiednie dla bufora głębokości:

(a) Potrzebny jest nam tylko poziom mipmapy najwyższej rozdzielczości (pole `MipLevels` równe 1),

(b) Wymagany jest inny format. Bufor głębokości zamiast koloru RGBA, przechowuje pojedynczą wartość głębokości. W tym celu użyć możemy formatu `DXGI_FORMAT_D32_FLOAT`. Jedną z innych możliwości jest wydzielenie dodatkowego kanału na tzw. bufor szablonu podając format `DXGI_FORMAT_D24_UNORM_S8_UINT`. Z bufora szablonu korzystać będziemy na późniejszych zajęciach.

(c) Tekstury tej nie będziemy używać do tworzenia widoku zasobu shadera, lecz jako bufora głębokości. Tak więc pole `BindFlags` powinno mieć wartość `D3D11_BIND_DEPTH_STENCIL`.

Strukturę `Texture2DDescription` rozszerzyć należy o statyczną metodę `DepthStencilDescription` (kod 2), dokona wymienionych powyżej zmian (kod 3).

```
14 struct Texture2DDescription : D3D11_TEXTURE2D_DESC
15 {
16     Texture2DDescription(UINT width, UINT height);
17     static Texture2DDescription DepthStencilDescription(
18         UINT width, UINT height);
19 };
```

Listing 2: Modyfikacja definicji struktury `Texture2DDescription`

4. Naszą nową metodę wykorzystamy do stworzenia tekstury 2D[14]. Jednakże, tak jak w przypadku *Render Target*u, nie możemy jej bezpośrednio podłączyć do kontekstu jako bufora głębokości. W tym celu potrzebne jest jeszcze utworzenie widoku bufora głębokości i szablonu (ang. *Depth-Stencil View* - widok się tak nazywa, nawet gdy tekstura nie zawiera bufora szablonu).

   Posłuży nam do tego metoda `CreateDepthStencilView`[15] urządzenia.

[14] https://docs.microsoft.com/en-us/windows/desktop/api/d3d11/nf-d3d11-id3d11device-createtexture2d

[15] https://docs.microsoft.com/en-us/windows/desktop/api/d3d11/nf-d3d11-id3d11device-createdepthstencilview

```
49 Texture2DDescription
50 Texture2DDescription::DepthStencilDescription(UINT width,
51     UINT height) {
52     Texture2DDescription desc(width, height);
53     desc.MipLevels = 1;
54     desc.Format = DXGI_FORMAT_D24_UNORM_S8_UINT;
55     desc.BindFlags = D3D11_BIND_DEPTH_STENCIL;
56     return desc;
57 }
```

Listing 3: Definicja statycznej metody `DepthStencilDescription` klasy `Texture2DDescription`

Klasę `DxDevice` należy rozszerzyć o metodę `CreateDepthStencilView` (kod 4), w której umieścimy implementację (kod 5).

```
8  class DxDevice
9  {
10 public:
       ...
24     mini::dx_ptr<ID3D11DepthStencilView>
25     CreateDepthStencilView(SIZE size) const;
       ...
31 };
```

Listing 4: Modyfikacja definicji klasy `DxDevice`

```
116 dx_ptr<ID3D11DepthStencilView>
117 DxDevice::CreateDepthStencilView(SIZE size) const {
118     auto desc = Texture2DDescription::DepthStencilDescription(
119         size.cx, size.cy);
120     dx_ptr<ID3D11Texture2D> texture = CreateTexture(desc);
121     return CreateDepthStencilView(texture);
122 }
```

Listing 5: Metoda `CreateDepthStencilView` klasy `DxDevice`

5. Pozycje obiektów (trójkąty, linie, itp.) rysowanych przez potok renderowania wyrażone są w trójwymiarowym tzw. znormalizowanym układzie współrzędnych urządzenia (NDC, ang. *Normalized Device Coordinates*) i obcięte do prostopadłościanu $[-1,1]\times[-1,1]\times[0,1]$. Współrzędne x i y odpowiadają za pozycję na ekranie, współrzędna z trafia do bufora głębokości.

Viewport odpowiada za odpowiednie przeskalowanie tych współrzędnych (w tym odwrócenie osi Y) zanim trafią do bufora wyjściowego. Parametry opisuje struktura `D3D11_VIEWPORT`[16]. W pliku `dxStructures.h` znaleźć możesz pochodną strukturę `Viewport`, której konstruktor wypełnia pola domyślnymi wartościami.

Pola `TopLeftX`, `TopLeftY`, `Width` i `Height` określają fragment *Render Target*u na którym rysowane będą renderowane obiekty.

6. W konstruktorze `DxApplication` utworzyć należy bufor głębokości, *viewport* i powiązać je (oraz *Render Target* — zauważ, że wcześniej tego jeszcze nie zrobiliśmy) z kontekstem potoku renderowania[17]. Zmiany te prezentuje kod 6.

```
3  DxApplication::DxApplication(HINSTANCE hInstance)
4  : ... {
       ...
13     SIZE wndSize = m_window.getClientSize();
14     m_depthBuffer = m_device.CreateDepthStencilView(wndSize);
15     auto backBuffer = m_backBuffer.get();
16     m_device.context()->OMSetRenderTargets(1,
17        &backBuffer, m_depthBuffer.get());
18     Viewport viewport{ wndSize };
19     m_device.context()->RSSetViewports(1, &viewport);
20 }
```

Listing 6: Modyfikacja konstruktora klasy `DxApplication`

7. Wszystkie dane przekazywane są do pamięci karty graficznej za pomocą buforów. Ich tworzenie jest jednak trochę bardziej złożone, niż alokacja pamięci RAM, gdyż określić musimy nie tylko rozmiar, ale też sposób w jaki bufor będzie wykorzystywany. Do określenia parametrów służy struktura `D3D11_BUFFER_DESC`[18], dla której również mamy strukturę pochodną `BufferDescription` definiują pomocniczy konstruktor nadający polom domyślne wartości.

   Pierwszym buforem, który stworzymy będzie bufor wierzchołków. Musimy ustawić dla niego pole `BindFlags` na `D3D11_BIND_VERTEX_BUFFER`, aby mógł być on użyty jako dane wejściowe potoku renderowania.

[16] https://docs.microsoft.com/en-gb/windows/desktop/api/d3d11/ns-d3d11-d3d11_viewport

[17] https://docs.microsoft.com/en-us/windows/desktop/api/d3d11/nf-d3d11-id3d11devicecontext-omsetrendertargets, https://docs.microsoft.com/en-us/windows/desktop/api/d3d11/nf-d3d11-id3d11devicecontext-rssetviewports

[18] https://docs.microsoft.com/en-us/windows/desktop/api/d3d11/ns-d3d11-d3d11_buffer_desc

Strukturę `BufferDescription` rozszerzyć należy o pomocniczą statyczną metodę `VertexBufferDescription` (kod 7).

```
21 struct BufferDescription : D3D11_BUFFER_DESC
22 {
23     BufferDescription(UINT bindFlags, size_t byteWidth);
24     static BufferDescription
25     VertexBufferDescription(size_t byteWidth)
26     { return { D3D11_BIND_VERTEX_BUFFER, byteWidth }; }
27 };
```

Listing 7: Modyfikacja definicji struktury `BufferDescription`

8. Do definicji klasy `DxDevice` dodaj metodę `CreateVertexBuffer`, która stworzy[19] bufor wierzchołków i przekopiuje dane z podanego wektora. (Kod 8)

```
8  class DxDevice
9  {
10 public:
       ...
25     template<class T> mini::dx_ptr<ID3D11Buffer>
26     CreateVertexBuffer(const std::vector<T>& vertices) const {
27         auto desc = BufferDescription::VertexBufferDescription(
28             vertices.size() * sizeof(T));
29         return CreateBuffer(reinterpret_cast<const void*>(
30             vertices.data()), desc);
31     }
       ...
37 };
```

Listing 8: Metoda `CreateVertexBuffer` klasy `DxDevice`

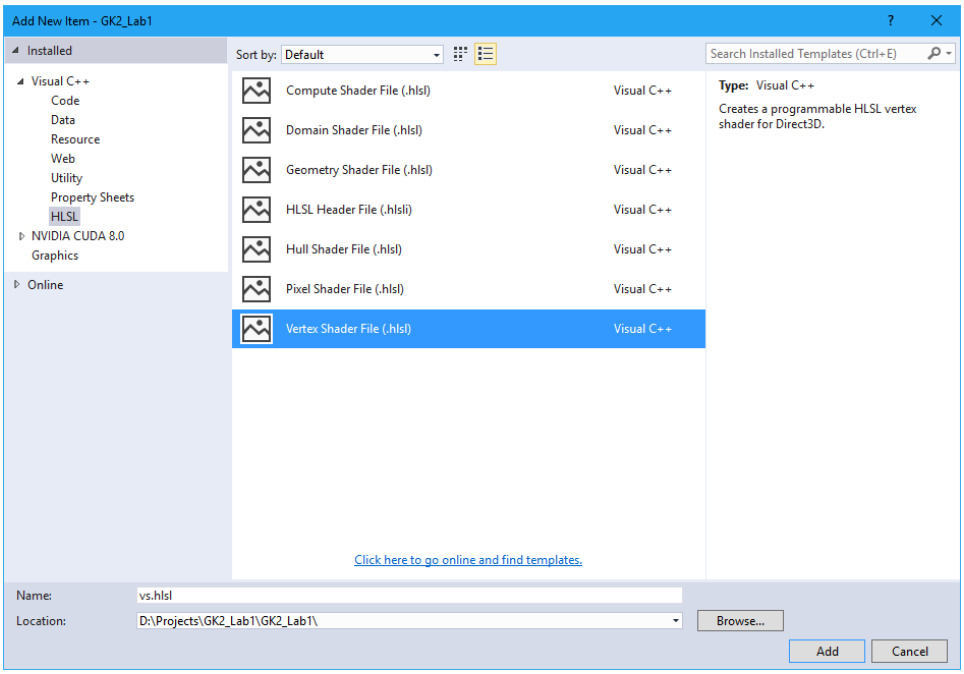
9. Programowany potok renderowania, jak sama nazwa wskazuje zawiera etapy, które możemy sami zaprogramować: shadery wierzchołków i pikseli (ang. *Vertex* i *Pixel Shader*).

   Pierwszy z nich odpowiada przede wszystkim za przekształcenia pozycji wierzchołków. Wartości zwrócone traktowane są jako wyrażone w przestrzeni afinicznej i po automatycznej normalizacji (podzieleniu przez współrzędną W) określają pozycję w NDC.

   Drugi wywoływany jest dla każdego piksela wewnątrz rysowanego obiektu i ma za zadanie wyznaczenie koloru danego piksela.

[19] https://docs.microsoft.com/en-us/windows/desktop/api/d3d11/nf-d3d11-id3d11device-createbuffer

Shadery w Direct3D piszemy w języku HLSL. Jego składnia jest bardzo podobna do C++. Do projektu należy dodać nowe pliki *Vertex* i *Pixel Shader*a (*Project* → *Add New Item* → *Visual C++* → *HLSL*) o nazwach `vs.hlsl` (rys. 2) oraz `ps.hlsl`.



Rysunek 2: Tworzenie pliku *Vertex Shader*a

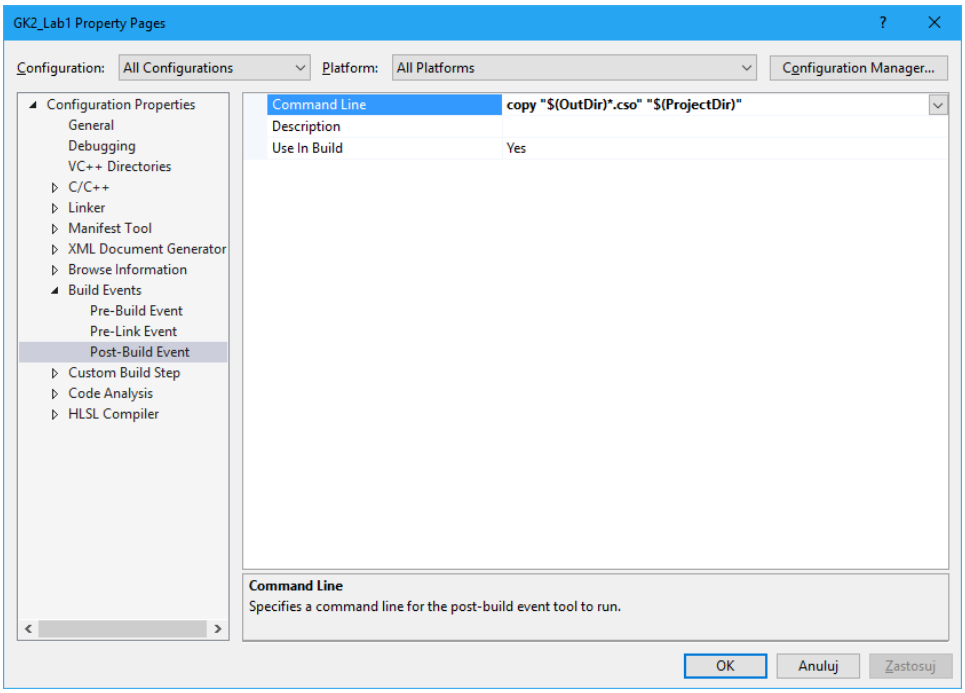
10. Pliki `.hlsl` są kompilowane, a wynik umieszczany w katalogu wyjściowym projektu. Jednakże łatwiej będzie nam je wczytać, jeżeli przekopiujemy je sobie do folderu głównego. W tym celu w ustawieniach projektu (*Project* → *Properties* → *Configuration Properties* → *Build Events* → *Post-Build Event*) w polu *Command Line* (rys. 3) należy umieścić instrukcję:
`copy "$(OutDir)*.cso" "$(ProjectDir)"`

11. Pozycji wierzchołków póki co nie będziemy modyfikować. Rozszerzymy je jednak z dwu-wymiarowych do wektorów afinicznych.

    Każdy piksel pokolorujemy stałym, różowawym kolorem.

    Modyfikację plików `vs.hlsl` oraz `ps.hlsl` ilustrują kod 9 oraz 10.

12. Wracając do kodu programu, stwórzmy w klasie `DxApplication` pomocniczą metodę `CreateTriangleVertices`, która zwróci wektor pozycji (`std::vector<XMFLOAT2>`) trzech wierzchołków trójkąta. Jej implementację pozostawiam jako ćwiczenie.



Rysunek 3: Tworzenie pliku *Vertex Shadera*

```
1 float4 main( float2 pos : POSITION ) : SV_POSITION {
2    return float4(pos, 0.0f, 1.0f);
3 }
```

Listing 9: Kod shadera wierzchołków (`vs.hlsl`)

13. Pozostał nam jeszcze jeden element układanki. Jako że na wejściu potoku renderowania możemy mieć wiele buforów wierzchołków, a każdy z nich zawierać może jeden lub więcej atrybutów wierzchołka, musimy określić w jaki sposób atrybuty mapowane są na parametry funkcji `main` shadera wierzchołków.

    Atrybuty do parametrów mapowane są po tzw. semantyce składającej się z nazwy oraz indeksu. Zauważ, że po parametrze `main` w `vs.hlsl` podaliśmy nazwę semantyki (`POSITION`[2021]).

    Opis mapowania pojedynczego parametru/atrybutu umieszczamy w strukturze `D3D11_INPUT_ELEMENT_DESC`[22]. Zastanówmy się jak w tej

[20] Indeks pominęliśmy, gdyż domyślnie użyta jest wartość 0.

[21] Nazwy semantyk są w miarę dowolne, poza tymi rozpoczynającymi się od przedrostka `SV_`, gdyż te interpretowane są przez potok renderowania.

[22] https://docs.microsoft.com/en-us/windows/desktop/api/d3d11/ns-d3d11-d3d11_input_element_desc

```
float4 main() : SV_TARGET {
    return float4(1.0f, 0.5f, 0.5f, 1.0f);
}
```

Listing 10: Kod shadera pikseli (`ps.hlsl`)

strukturze opisać mapowanie pozycji (zapisanej w tablicy elementów typu `XMFLOAT2`) na parametr `pos` funkcji `main` shadera wierzchołków.

(a) Pole `SemanticName` zawiera nazwę semantyki jako ciąg znaków — w naszym przypadku to `"POSITION"`.

(b) `SemanticIndex` to indeks semantyki — 0,

(c) `Format` to format danych atrybutu w buforze wierzchołków. Klasa `XMFLOAT2` reprezentuje wektor 2D współrzędnych typu `float` — odpowiadający format danych to `DXGI_FORMAT_R32G32_FLOAT`,

(d) `InputSlot` to indeks bufora wierzchołków, z którego brany jest atrybut — użyjemy tylko jednego bufora, który przypniemy do indeksu 0,

(e) `AlignedByteOffset` określa offset w bajtach w strukturze wierzchołka w danym buforze wierzchołków do jego początku pola atrybutu — u nas wierzchołek zawiera tylko pole pozycji, więc offset wynosi 0,

(f) `InputSlotClass` musimy wynieść `D3D11_INPUT_PER_VERTEX_DATA`. Druga z możliwych wartości umożliwia skorzystanie z zaawansowanej techniki tzw. *instancing*u geometrii, o której opowiemy sobie kiedy indziej.

(g) `InstanceDataStepRate` też odnosi się do tej techniki — skoro z niej nie korzystamy, ustawiamy 0.

W klasie `DxDevice` mamy już pomocniczą metodę `CreateInputLayout`, która z tablicy elementów `D3D11_INPUT_ELEMENT_DESC` stworzy tzw. *Input Layout* pasujący do danego shadera wierzchołków[23].

14. W konstruktorze klasy `DxApplication` należy stworzyć bufor wierzchołków trójkąta, wczytać i stworzyć shadery wierzchołków i pikseli oraz stworzyć łączący *Input Layout*. Zmiany prezentuje kod 11.

15. Aby poprawnie narysować trójkąt, w metodzie `Render` klasy `DxApplication` należy dodatkowo:

[23] tudzież każdego innego shadera wierzchołków o tej samej ilości, kolejności, typie i semantyce parametrów

```
#include "dxApplication.h"
#include <DirectXMath.h>
using namespace mini;
using namespace std;
using namespace DirectX;
DxApplication::DxApplication(HINSTANCE hInstance)
: ... {
    ...
    const auto vsBytes = DxDevice::LoadByteCode(L"vs.cso");
    const auto psBytes = DxDevice::LoadByteCode(L"ps.cso");
    m_vertexShader = m_device.CreateVertexShader(vsBytes);
    m_pixelShader = m_device.CreatePixelShader(psBytes);
    const auto vertices = CreateTriangleVertices();
    m_vertexBuffer = m_device.CreateVertexBuffer(vertices);
    vector<D3D11_INPUT_ELEMENT_DESC> elements {
        { "POSITION", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 0,
          D3D11_INPUT_PER_VERTEX_DATA, 0 }
    };
    m_layout = m_device.CreateInputLayout(elements, vsBytes);
}
```

Listing 11: Modyfikacja konstruktora klasy `DxApplication`

(a) Używając metody kontekstu `ClearDepthStencilView`[24] wyczyścić bufor głębokości (jako drugi parametr podajemy kombinację flag `D3D11_CLEAR_DEPTH` oraz `D3D11_CLEAR_STENCIL` aby wyczyścić oba kanały naszego bufora; trzeci i czwarty parametr to wartości jakie wpisane zostaną do obu kanałów);

(b) powiązać z potokiem renderowania bufor wierzchołków — bufor nie ma informacji o tym jakiego rozmiaru dane przechowuje, dlatego poza tablicą buforów do funkcji `IASetVertexBuffers`[25] kontekstu przekazać musimy również tablicę rozmiarów (w bajtach) elementu w każdym z buforów, oraz tablicę offsetów (również w bajtach) do początku pierwszego elementu — dla naszego bufora rozmiar elementu to `sizeof(XMFLOAT2)`, a offset to oczywiście 0;

(c) powiązać shadery[26], *Input Layout*[27] z kontekstem;

[24] https://docs.microsoft.com/en-us/windows/desktop/api/d3d11/nf-d3d11-id3d11devicecontext-cleardepthstencilview

[25] https://docs.microsoft.com/en-us/windows/desktop/api/d3d11/nf-d3d11-id3d11devicecontext-iasetvertexbuffers

[26] https://docs.microsoft.com/en-us/windows/desktop/api/d3d11/nf-d3d11-id3d11devicecontext-vssetshader,https://docs.microsoft.com/en-us/windows/desktop/api/d3d11/nf-d3d11-id3d11devicecontext-pssetshader

[27] https://docs.microsoft.com/en-us/windows/desktop/api/d3d11/nf-d3d11-id3d11devicecontext-iasetinputlayout

(d) Zmienić typ rysowanej przez potok renderowania geometrii za pomocą metody `IASetPrimitiveTopology`[28] kontekstu na listę trójkątów — stała `D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST`;

(e) wywołać funkcję renderowania[29] podając liczbę wierzchołków do narysowania oraz indeks wierzchołka startowego.

Zmiany te prezentuje kod 12.

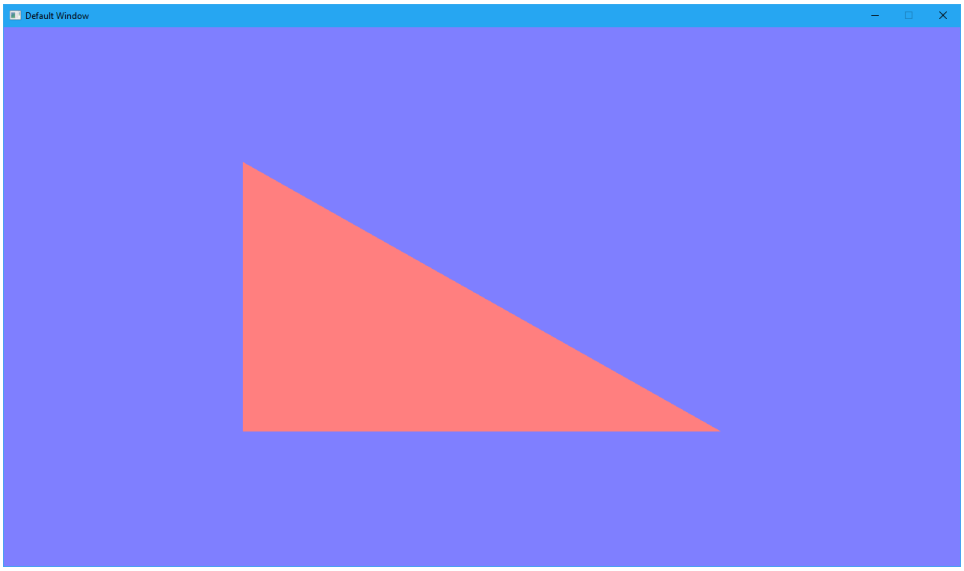
```
52 void DxApplication::Render() {
       ...
56     m_device.context()->ClearDepthStencilView(
57        m_depthBuffer.get(),
58        D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);
59     m_device.context()->VSSetShader(
60        m_vertexShader.get(), nullptr, 0);
61     m_device.context()->PSSetShader(
62        m_pixelShader.get(), nullptr, 0);
63     m_device.context()->IASetInputLayout(m_layout.get());
64     m_device.context()->IASetPrimitiveTopology(
65        D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
66     ID3D11Buffer* vbs[] = { m_vertexBuffer.get() };
67     UINT strides[] = { sizeof(XMFLOAT2) };
68     UINT offsets[] = { 0 };
69     m_device.context()->IASetVertexBuffers(
70        0, 1, vbs, strides, offsets);
71     m_device.context()->Draw(3, 0);
72 }
```

Listing 12: Modyfikacja metody `Render` klasy `DxApplication`

16. Po uruchomieniu programu okno wyglądać powinno jak na rys. 4.

[28] https://docs.microsoft.com/en-us/windows/desktop/api/d3d11/nf-d3d11-id3d11devicecontext-iasetprimitivetopology

[29] https://docs.microsoft.com/en-us/windows/desktop/api/d3d11/nf-d3d11-id3d11devicecontext-draw



Rysunek 4: Oczekiwany końcowy wygląd okna programu

Część IV
Obracająca się kostka

1 Tworzenie modelu

W tej części postaramy się narysować trójwymiarowy model sześcianu z każdą ze ścian w innym kolorze. Jego powierzchnia składa się z 6 kwadratów, a każdy z nich można złożyć z dwóch trójkątów. Aby narysować je wszystkie musielibyśmy stworzyć bufor wierzchołków zawierający 36 elementów, jednak wierzchołki sześcianu posiadają tylko 8 unikalnych pozycji. Widać więc, że wiele pozycji w buforze by się w ten sposób powtarzało.

Istnieje metoda unikania duplikowania wierzchołków w buforze, każda unikalna kombinacja atrybutów wierzchołka umieszczona musi być przynajmniej raz. Jeżeli chcemy by ściany były kolorowe, informację tą musimy zapisać w wierzchołkach w postaci dodatkowego atrybutu koloru. Tak więc wierzchołki będące częścią różnych ścian, nawet jeśli mają tą samą pozycje, różnić się będą kolorem — takiej duplikacji nie da się łatwo uniknąć.

Możemy jednak zredukować liczbę wierzchołków tworzących jedną ścianę (mających ten sam kolor) z 6 (dwa trójkąty) do 4, redukując rozmiar bufora do 24 elementów. Oszczędność może nie jest duża, ale pozwoli nam zilustrować rysowanie z użyciem tzw. bufora indeksów.

W tym przypadku wierzchołki nie będą tworzyć trójkątów na podstawie ich kolejności w buforze. Elementy bufora wierzchołków można więc ustawić dowolnie. Musimy jednak stworzyć jeszcze bufor indeksów mówiący, które wierzchołki składają się na każdy trójkąt. Trzy kolejne elementy definiują trójkąt wskazując indeksy jego wierzchołków w buforze wierzchołków.

Stworzymy sześcian o krawędziach długości 1, równoległych do osi układu współrzędnych (układ w którym definiujemy początkowe położenia modelu zwany jest jego układem lokalnym).

Wykonanie:

1. Strukturę `BufferDescription` rozszerz o pomocniczą statyczną metodę `IndexBufferDescription` (kod 13).

```
21 struct BufferDescription : D3D11_BUFFER_DESC
22 {
       ...
27     static BufferDescription
28     IndexBufferDescription(size_t byteWidth)
29     { return { D3D11_BIND_INDEX_BUFFER, byteWidth }; }
30 };
```

Listing 13: Modyfikacja definicji struktury `BufferDescription`

2. Do definicji klasy `DxDevice` dodaj metodę `CreateIndexBuffer`, która stworzy bufor indeksów i wypełni go danymi z wektora. (Kod 14)

```
8  class DxDevice
9  {
10 public:
       ...
32     template<class T> mini::dx_ptr<ID3D11Buffer>
33     CreateIndexBuffer(const std::vector<T>& indices) const {
34         auto desc = BufferDescription::IndexBufferDescription(
35             indices.size() * sizeof(T));
36         return CreateBuffer(reinterpret_cast<const void*>(
37             indices.data()), desc);
38     }
       ...
44 };
```

Listing 14: Metoda `CreateIndexBuffer` klasy `DxDevice`

3. Oba atrybuty wierzchołków będziemy przechowywać w jednym buforze. Dla ułatwienia ich definicji i zwiększenia czytelności kodu stworzymy pomocniczą strukturę `VertexPositionColor`, której definicję umieścimy w klasie `DxApplication`.

   Definicję klasy rozszerzymy przy okazji o metody `CreateCubeVertices` i `CreateCubeIndices`, które stworzą model kostki, oraz pole bufora indeksów. Zmiany prezentuje kod 15.

```
4  class DxApplication : public mini::WindowApplication {
       ...
9  private:
10     struct VertexPositionColor {
11         DirectX::XMFLOAT3 position, color;
12     };
13     static std::vector<VertexPositionColor> CreateCubeVertices();
14     static std::vector<unsigned short> CreateCubeIndices();
15     mini::dx_ptr<ID3D11Buffer> m_indexBuffer;
       ...
27 };
```

Listing 15: Modyfikacja definicji klasy `DxApplication`

4. Częściową implementację tworzenia modelu prezentuje kod 16. Dodanie kodu tworzącego pozostałe ściany sześcianu pozostawione zostało jako ćwiczenia.

Przy dodawaniu kolejnych indeksów zwrócić uwagę należy na kolejność wierzchołków w trójkącie. Ze względu na obcinanie ścian tylnych, które domyślnie jest włączone, dla trójkąta obserwowanego od strony widocznej kolejność wierzchołków na ekranie powinna być zgodna z ruchem wskazówek zegara[30].

```
74 vector<DxApplication::VertexPositionColor>
75 DxApplication::CreateCubeVertices() {
76     return {
77     //Front Face
78     { { -0.5f, -0.5f, -0.5f }, { 1.0f, 0.0f, 0.0f } },
79     { { +0.5f, -0.5f, -0.5f }, { 1.0f, 0.0f, 0.0f } },
80     { { +0.5f, +0.5f, -0.5f }, { 1.0f, 0.0f, 0.0f } },
81     { { -0.5f, +0.5f, -0.5f }, { 1.0f, 0.0f, 0.0f } },
       ...
110    };
111 }
112 vector<unsigned short> DxApplication::CreateCubeIndices() {
113     return {
114         0,2,1, 0,3,2,
            ...
119     };
120 }
```

Listing 16: Tworzenie modelu sześcianu

5. W konstruktorze `DxApplication` musimy:

   (a) Zainicjalizować bufor wierzchołków wierzchołkami sześcianu zamiast trójkąta;

   (b) stworzyć bufor indeksów kostki;

   (c) zmodyfikować *Input Layout* tak by pasował do naszego nowego modelu — dla atrybutu koloru użyjemy semantyki `"COLOR"`.

   Zmiany prezentuje kod 17.

6. Do rysowania modelu wykorzystującego bufor indeksów w metodzie `Render` klasy `DxApplication` musimy zamiast metody `Draw` kontekstu użyć musimy `DrawIndexed`[31]. Wymaga ona przypięcia[32] do kontekstu bufora wierzchołków, a jej parametrami są: ilość wierzchołków, indeks

[30] https://docs.microsoft.com/en-gb/windows/desktop/api/d3d11/ns-d3d11-d3d11_rasterizer_desc#remarks

[31] https://docs.microsoft.com/en-us/windows/desktop/api/d3d11/nf-d3d11-id3d11devicecontext-drawindexed

[32] https://docs.microsoft.com/en-us/windows/desktop/api/d3d11/nf-d3d11-id3d11devicecontext-iasetindexbuffer

```
7  DxApplication::DxApplication(HINSTANCE hInstance)
8  : ... {
       ...
27     const auto vertices = CreateCubeVertices();
28     m_vertexBuffer = m_device.CreateVertexBuffer(vertices);
29     const auto indices = CreateCubeIndices();
30     m_indexBuffer = m_device.CreateIndexBuffer(indices);
31     vector<D3D11_INPUT_ELEMENT_DESC> elements {
32        { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
33        D3D11_INPUT_PER_VERTEX_DATA, 0 },
34        { "COLOR", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0,
35           offsetof(VertexPositionColor, color),
36           D3D11_INPUT_PER_VERTEX_DATA, 0 }
37     };
38     m_layout = m_device.CreateInputLayout(elements, vsBytes);
39 }
```

Listing 17: Modyfikacja konstruktora klasy `DxApplication`

pierwszego indeksu w buforze od którego funkcja ma zacząć, oraz *offset* w buforze wierzchołków (tzn. wartość dodana do każdego indeksu przy odwoływaniu się do wierzchołków).

Ostatnią zmianą jest zaktualizowanie rozmiaru elementu w buforze wierzchołków podawanego do `IASetVertexBuffers`.

Powyższe modyfikacje prezentuje kod 18.

```
57 void DxApplication::Render() {
       ...
71     ID3D11Buffer* vbs[] = { m_vertexBuffer.get() };
72     UINT strides[] = { sizeof(VertexPositionColor) };
73     UINT offsets[] = { 0 };
74     m_device.context()->IASetVertexBuffers(
75        0, 1, vbs, strides, offsets);
76     m_device.context()->IASetIndexBuffer(m_indexBuffer.get(),
77        DXGI_FORMAT_R16_UINT, 0);
78     m_device.context()->DrawIndexed(36, 0, 0);
79 }
```

Listing 18: Modyfikacja metody `Render` klasy `DxApplication`

7. Jeśli uruchomimy program w tym momencie, w oknie wyświetli się różowy trójkąt o wymiarach równych połowie wysokości i szerokości okna. Aby użyć kolorów zapisanych w wierzchołkach musimy zmodyfikować shadery.

Shader wierzchołków powinien przyjmować dodatkowy parametr koloru, który zwrócony musi być wraz z pozycją. Jako, że w HLSL jak w C/C++ funkcje zwracać mogą tylko jedną wartość, musimy zdefiniować strukturę zawierającą oba zwracane atrybuty. Semantyki atrybutów podaje się w takim przypadku w definicji struktury przy odpowiednich polach. Podobną strukturę możemy zdefiniować dla parametrów funkcji `main` shadera — zwiększa to czytelność kodu zwłaszcza gdy wierzchołki mają dużo atrybutów wejściowych. Zmiany w shaderze wierzchołków prezentuje kod 19.

```
struct VSIn {
    float3 pos : POSITION;
    float3 col : COLOR;
};
struct VSOut {
    float4 pos : SV_POSITION;
    float4 col : COLOR;
};
VSOut main( VSIn i )
{
    VSOut o;
    o.pos = float4(i.pos, 1.0f);
    o.col = float4(i.col, 1.0f);
    return o;
}
```

Listing 19: Kod shadera wierzchołków (`vs.hlsl`)

8. W shaderze pikseli możemy użyć wartości koloru, która interpolowana zostanie automatycznie po powierzchni trójkąta na podstawie kolorów zwróconych z shadera wierzchołków dla jego rogów.

   W tym celu musimy dodać do funkcji `main` shadera pikseli pewne parametry. Ich kolejność, lub kolejność pól w strukturze będącej typem parametru, musi być taka sama jak na wyjściu shadera wierzchołków. Nie musimy przyjmować na wejściu wszystkich atrybutów wyjściowych shadera wierzchołków, lecz jeśli interesuje nas pewien atrybut, musimy przyjąć też wszystkie go poprzedzające.

   Zauważ, że podanie semantyki dla wszystkich atrybutów wejściowych i wyjściowych shaderów jest wymagane, nie są one używane do ich mapowania pomiędzy kolejnymi etapami potoku renderowania. Takie mapowanie musiałoby być robione w trakcie działania programu (shadery w potoku możemy dowolnie i nie zależnie od siebie zmieniać) co powodowałoby zbyt duży narzut. Z tego powodu mapowanie po

semantykach zostało usunięte z Direct3D od wersji 10.[33]

Wymóg semantyk pozostał na potrzeby tzw. efektów[34], które definiują stan potoku renderowania (w tym shadery) kompilowany z jednego pliku. Mając kod sąsiadujących ze sobą shaderów kompilator może zapewnić mapowanie po semantykach statycznie podczas kompilacji. Jednakże od wersji Direct3D 11 efekty dostępne są jako oddzielna biblioteka[35] udostępniana głównie na potrzeby portowania starszych aplikacji, a flaga kompilatora HLSL do kompilacji efektów oznaczona jest jako przestarzała.

Mając to wszystko na uwadze, jako typ parametru shadera wierzchołków najlepiej użyć tej samej struktury `VSOut`, jak prezentuje to kod 20.

```
1 struct VSOut {
2    float4 pos : SV_POSITION;
3    float4 col : COLOR;
4 };
5 float4 main(VSOut i) : SV_TARGET
6 {
7    return i.col;
8 }
```

Listing 20: Kod shadera pikseli (`ps.hlsl`)

Zamiast kopiować kod, można do projektu dodać plik nagłówkowy HLSL (`.hlsli`) w którym umieszczone będą struktury współdzielone i dołączyć go do obu shaderów (dyrektywa `#include`). Taka opcjonalna zmiana pozostawiona jest jako ćwiczenie.

2 Transformacje

Program uruchomiony po wykonaniu powyższych kroków wyświetli puste okno. Zauważ, że przed zmianą shadera wierzchołków współrzędna z pozycji wierzchołków była ignorowana. Obecnie ściana przednia kostki wyjechała poza prostopadłościan obcinania w NDC, ściany górna, dolna, lewa i prawa, mimo że obcięta tylko częściowo, są prostopadłe do płaszczyzny XY i przez to niewidoczne, natomiast ściana tylna obserwowana jest od strony niewidocznej i, jak sama nazwa wskazuje, nie zdała testu obcinania ścian tylnych.

[33] https://docs.microsoft.com/en-us/windows/desktop/direct3dhlsl/dx-graphics-hlsl-signatures

[34] https://docs.microsoft.com/en-us/windows/desktop/direct3d11/d3d11-graphics-programming-guide-effects

[35] https://github.com/Microsoft/FX11

Do sceny musimy dodać pewne transformacje, które zapewnią, że:

1. ma poprawne proporcje (wcześniej, mimo że ściany są kwadratami, na ekranie rysował nam się prostokąt);
2. uzyskamy efekt iluzji głębi poprzez rzutowanie perspektywiczne;
3. kostka obraca się w czasie wokół osi prostopadłej do ściany górnej przechodzącej przez jej środek.

Transformacje wyrażać będziemy macierzami 4×4. Pierwsze dwa punkty zapewni nam macierz rzutowania perspektywicznego. Wyobrazić możemy sobie, że naszą trójwymiarową scenę obserwujemy z pewnego punktu. Dla ułatwienia powiążmy go z pewnym układem współrzędnych (zwanych układem kamery). Na scenę spoglądamy wzdłuż osi Z układu w kierunku dodatnim. Ponadto patrzymy na nią przez pewne prostokątne okno prostopadłe do osi Z, przecinające tą oś w swoim środku. Krawędzie okna są prostopadłe do osi X i Y a jego proporcje są takie same jak proporcje bufora na którym będziemy rysować. Okno ma pewną ustaloną odległość od początku układu (zwaną odległością przedniej płaszczyzny obcinania). Wymiary okna nie są dane wprost. Rozpatrzymy stożek o wierzchołku w początku układu którego ściany przechodzą przez krawędzie okna (zwany stożkiem widzenia). Wysokość okna wyznaczyć można z kąta dwuściennego pomiędzy górną i dolną ścianą stożka (zwanego kątem widzenia w pionie). Mając wysokość i współczynnik proporcji wyznaczyć można też szerokość. Za oknem w pewnej odległości umieszczamy drugą równoległą płaszczyznę zwaną tylną płaszczyzną obcinania.

Macierz perspektywy P, sparametryzowana:

- kątem pola widzenia w pionie,
- współczynnikiem proporcji okna,
- odległością od początku układu bliższej i dalszej płaszczyzny obcinania

przekształci (w połączeniu z normalizacją współrzędnych afinicznych) ścięty stożka widzenia na prostokąta obcinania NDC.

Przekształcenie perspektywiczne P będzie ostatnim, którym poddane zostaną pozycje wierzchołków. Zanim to jednak nastąpi musimy kostkę odpowiednio ustawić wewnątrz stożka widzenia.

Zaczynamy od kostki o środku w początku układu i ścianach prostopadłych do osi. Jako oś Y pokrywa się z osią obrotu naszej docelowej animacji, tą transformację (R_Y) wykonamy najpierw.

Obróconą kostkę pochylimy trochę (obracając wokół osi X — macierz transformacji R_X) tak by spojrzeć częściowo na ścianę górną.

W ostatnim kroku przesuniemy obróconą kostkę wzdłuż osi Z (macierz transformacji T_Z) tak by znalazła się gdzieś wewnątrz stożka widzenia.

Ostatecznie każda pozycja przekształcona zostanie przez iloczyn macierzy:

$$M_{MVP} = PT_Z R_X R_Y$$

Tradycyjnie macierz M_{MVP} reprezentuje się jako iloraz trzech macierzy $M_{MVP} = M_P M_V M_M$. Podział podstawowych transformacji pomiędzy te trzy składowe jest w pewnym stopniu umowny, ale zależy głównie od tego jak często ich wartości będą zmieniane:

1. Macierz M_P zazwyczaj reprezentuje wyłącznie transformację rzutowania. Zmienia się ona bardzo rzadko lub wcale. Innym powodem wyróżnienia jest to, że jest to jedyne przekształcenie, które nie jest afiniczne.

2. Macierz M_C definiuje przekształcenie do układu kamery z pośredniego układu zwanego często układem świata lub układem globalnym. Wydziela ona transformacje afiniczne które stosowane są do każdego modelu. Pozwala to traktować kamerę jako oddzielny obiekt na scenie, którym możemy poruszać, np. pod wpływem akcji użytkownika. Zwykle zmienia się ona co najwyżej raz na klatkę.

3. Macierz M_M zwana macierzą modelu lub macierzą świata zawiera transformacje specyficzne dla danego modelu. Jeżeli rysujemy wiele obiektów, każdy posiadać będzie macierz świata definiującą jego położenie w układzie świata (tj. w relacji do pozostałych modeli). Z punktu widzenia modelu zazwyczaj się ona nie zmienia, chyba że obiekt jest animowany. Jednakże z punktu widzenia shadera wierzchołków, odpowiadającego za przeprowadzanie transformacji, zawartość macierzy modelu zmienia się wielokrotnie na klatkę, za każdym razem gdy rysujemy inny model.

Na potrzeby kolejnych etapów w naszym projekcie dokonamy następującego podziału:

- $M_P = P$ — macierz rzutowania perspektywicznego, która będzie stała.
- $M_C = T_Z R_X$ — macierz kamery zawierająca afiniczne przekształcenia, które (póki co) się nie zmieniają.
- $M_M = R_Y$ — macierz modelu odpowiadająca za animację kostki.

Wykonanie:

1. Zaczniemy od modyfikacji shadera wierzchołków, aby pozycja przekształcana była przez macierz $M_M VP$. Będzie ona się zmieniać w czasie więc musimy mieć sposób na przekazanie jej z kodu programu do shadera. W tym celu posłużymy się buforem stałych[36]. Podobnie jak

[36] https://docs.microsoft.com/en-gb/windows/desktop/direct3dhlsl/dx-graphics-hlsl-constants

bufory wierzchołków, bufory stałych są przypinane do konkretnych slotów. Kod shadera definiuje w których slotach oczekiwane są jakie dane.

Dodajmy więc do shadera deklarację bufor stałych przypisaną do slotu 0 (`register(b0)`) zawierający jedną macierz `MVP`. Przez macierz przemnożymy pozycję wierzchołka. Zauważ, że mimo umieszczenia jej deklaracji w buforze stałych, identyfikator macierzy jest dostępny globalnie (nie wymaga np. poprzedzenia go nazwą bufora). Zmiany prezentuje kod 21.

```
1  cbuffer transformations : register(b0) {
2      matrix MVP;
3  }
   ...
12 VSOut main( VSIn i )
13 {
14     VSOut o;
15     o.pos = mul(MVP, float4(i.pos, 1.0f));
       ...
18 }
```

Listing 21: Modyfikacja kodu shadera wierzchołków (`vs.hlsl`)

Umieszczanie zmiennych globalnych w buforach stałych nie jest wymagane. Zmienne globalne zadeklarowane poza buforami (o ile nie mają kwalifikatora `static`) umieszczane są wszystkie w tworzonym automatycznie globalnym buforze stałych.

Podzielenie ich na bufory ma jednak tą zaletę, że aktualizacja danych w buforze wymaga wymiany pomiędzy procesorem i kartą graficzną wszystkich znajdujących się w nim zmiennych. Aby zmniejszyć ilość transferowanych danych należy grupować zmienne które wymagać aktualizacji będą w podobnych interwałach czasu.

O ile jawne podawanie slotu dla bufora nie jest konieczne, zawsze warto to robić. Dzięki temu indeks zostaje dla niego zarezerwowany nawet jeżeli nie jest on używany, tzn. każda zmienna z bufora nie jest użyta w programie shadera lub jeśli pojawia się w kodzie, to jej wartość nie wpływa na rezultat funkcji `main`, przez co zostaje wyoptymalizowana.

Buforom, które nie mają jawnie podanego slotu (dotyczy to też bufora globalnego), jest on przydzielany automatycznie. Bufory takie rozpatrywane są w kolejności w jakiej pojawiają się w kodzie. Bufor globalny jest wyjątkiem, rozpatrywany jest zawsze jako pierwszy niezależnie od tego gdzie zdefiniowana jest pierwsza zmienna globalna. Jeżeli bufor nie jest użyty, nie dostaje on slotu. W przeciwnym wypadku przypi-

sany zostaje mu pierwszy wolny, który nie jest zarezerwowany przez inne bufory.

Jak łatwo się domyślić drobne zmiany w kodzie shadera mogą w znaczny i nieoczywisty sposób zmienić przydział slotów, a to spowoduje konieczność odzwierciedlenia tych zmian w kodzie programu. Jest to też kolejna zaleta deklarowania zmiennych globalnych w jawnych buforach — buforowi globalnemu nie da się ręcznie przydzielić slotu.

2. W kodzie programu stworzyć musimy bufor, który powiążemy z shaderem. Zacznijmy od dodania do struktury `BufferDescription` o pomocniczą metodę `ConstantBufferDescription` (kod 22).

```
21 struct BufferDescription : D3D11_BUFFER_DESC
22 {
       ...
30     static BufferDescription
31     ConstantBufferDescription(size_t byteWidth);
32 };
```

Listing 22: Modyfikacja definicji struktury `BufferDescription`

Poza odpowiednią modyfikacją pola `BindFlags` musimy umożliwić zapis do bufora z poziomu procesora (`D3D11_CPU_ACCESS_WRITE` w polu `CPUAccessFlags`). Ponadto możemy zaznaczyć, że zawartość bufora będzie często aktualizowana podając `D3D11_USAGE_DYNAMIC` w polu `Usage`. Zmiany prezentuje kod 23.

```
70 BufferDescription
71 BufferDescription::ConstantBufferDescription(size_t byteWidth)
72 {
73     BufferDescription desc{
74         D3D11_BIND_CONSTANT_BUFFER, byteWidth };
75     desc.Usage = D3D11_USAGE_DYNAMIC;
76     desc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
77     return desc;
78 }
```

Listing 23: Definicja metody `ConstantBufferDescription` struktury `BufferDescription`

3. Do definicji klasy `DxDevice` dodaj metodę `CreateConstantBuffer`, która stworzy bufor stałych zdolny pomieścić zmienną podanego typu. (Kod 24)

```
8  class DxDevice
9  {
10 public:
       ...
39     template<typename T>
40     mini::dx_ptr<ID3D11Buffer> CreateConstantBuffer() const
41     {
42         BufferDescription desc =
43             BufferDescription::ConstantBufferDescription(
44                 sizeof(T));
45         return CreateBuffer(nullptr, desc);
46     }
       ...
52 };
```

Listing 24: Metoda `CreateIndexBuffer` klasy `DxDevice`

4. Do klasy `DxApplication` dodajmy pola do przechowywania aktualnych wartości macierzy M_P, M_C i M_M, oraz pole bufora stałych tak, jak ilustruje to kod 25.

```
4  #include <DirectXMath.h>
5  class DxApplication : public mini::WindowApplication {
       ...
9  private:
       ...
26     DirectX::XMFLOAT4X4 m_modelMtx, m_viewMtx, m_projMtx;
27     mini::dx_ptr<ID3D11Buffer> m_cbMVP;
28 };
```

Listing 25: Modyfikacja definicji klasy `DxApplication`

5. W konstruktorze `DxApplication` wyznaczyć należy początkowe wartości macierzy transformacji i stworzyć bufor stałych (kod 26).

   Biblioteka DirectXMath zapewnia nam zoptymalizowane typy i funkcje do operacji na macierzach i wektorach. Wykorzystują one wektorowe instrukcje procesora, które niestety mają dość nietypowe wymagania co do wyrównania adresów ich parametrów. Dlatego biblioteka operuje głównie na typach `XMMATRIX` i `XMVECTOR`.

   Ze względu na wspomniane wymogi wyrównania, typy te nie za bardzo nadają się na pola struktur. Do takich zastosowań przeznaczone są typy takie jak poznane przez nas już `XMFLOAT2` i `XMFLOAT3` dla wektorów, czy `XMFLOAT4X4` dla macierzy. Istnieje spora liczba ich wariantów w zależności od tego ile elementów mają one zawierać.

```
7 DxApplication::DxApplication(HINSTANCE hInstance)
8 : ... {
...
39    XMStoreFloat4x4(&m_modelMtx, XMMatrixIdentity());
40    XMStoreFloat4x4(&m_viewMtx,
41       XMMatrixRotationX(XMConvertToRadians(-30))*
42          XMMatrixTranslation(0.0f, 0.0f, 10.0f));
43    XMStoreFloat4x4(&m_projMtx, XMMatrixPerspectiveFovLH(
44       XMConvertToRadians(45),
45       static_cast<float>(wndSize.cx) / wndSize.cy,
46       0.1f, 100.0f));
47    m_cbMVP = m_device.CreateConstantBuffer<XMFLOAT4X4>();
48 }
```

Listing 26: Modyfikacja konstruktora klasy `DxApplication`

DirectXMath wyewoluowało ze starszej biblioteki XNA Math, w której wszystkie identyfikatory istniały w globalnej przestrzeni nazw. Stąd nazwy funkcji są nadzwyczaj długie. Ponadto nie wykorzystuje ona operatorów konwersji i przypisania do przenoszenia wartości pomiędzy zmiennymi typów `XMMATRIX` i `XMVECTOR` a typami `XMFLOATXXX`. Zamiast tego definiuje szereg funkcje `XMStoreFloatXXX` i `XMLoadFloatXXX`.

W macierzy modelu zapisana na początku będzie macierz identycznościowa — zmienimy ją później.

W macierzy kamery zapiszemy iloczyn $T_Z R_Y$, gdzie R_Y jest pierwszą transformacją — obrotem o 30° wokół osi Y — po której następuje T_Z — translacja o 10 jednostek wzdłuż osi Z. Zauważ, że w notacji matematycznej kolejność transformacji postępuje od prawej do lewej. W DirectXMath, aby zachować kolejność transformacji, kolejność mnożenia macierzy musi zostać odwrócona!

Macierz rzutowania zainicjalizujemy na podstawie 4 opisanych wcześniej parametrów perspektywy: kąta pola widzenia w pionie, współczynnika proporcji tylnego bufora, oraz odległości bliższej i dalszej płaszczyzny obcinania.

Typy `XMMATRIX` i `XMFLOAT4X4` pod względem rozmiaru i rozmieszczenia są identyczne i są kompatybilne się z typem `matrix` w kodzie shadera. Oba mogą być więc użyte do wypełnienia bufora stałych. Jeżeli nasz bufor stałych zawierać powinien więcej zmiennych, przydatne byłoby stworzenie struktury o polach, których typy i kolejność odpowiadają zmiennym w buforze. To może jednak nie wystarczyć, gdyż reguły wyrównania pól w strukturze różnią się od tych w kodzie shadera![37]

[37] https://docs.microsoft.com/en-gb/windows/desktop/direct3dhlsl/dx-graphics-hlsl-packing-rules

6. Stworzony bufor stałych przypisać musimy do kontekstu. Dla przypomnienia użyć powinniśmy slotu 0 buforów stałych shadera wierzchołków. Zmiany wprowadzimy w metodzie `Render` klasy `DxApplication` (kod 27).

```
57 void DxApplication::Render() {
       ...
61     ID3D11Buffer* cbs[] = { m_cbMVP.get() };
62     m_device.context()->VSSetConstantBuffers(0, 1, cbs);
       ...
81 }
```

Listing 27: Modyfikacja metody `Render` klasy `DxApplication`

7. Za animację obrotu kostki, wymnożenie macierzy modelu, kamery i rzutowania, oraz zaktualizowanie zawartości bufora stałych odpowiadać będzie metoda `Update` klasy `DxApplication`, której implementację prezentuje kod 28.

```
82 void DxApplication::Update()
83 {
84     XMStoreFloat4x4(&m_modelMtx, XMLoadFloat4x4(&m_modelMtx) *
85         XMMatrixRotationY(0.0001f));
86     D3D11_MAPPED_SUBRESOURCE res;
87     m_device.context()->Map(m_cbMVP.get(), 0,
88         D3D11_MAP_WRITE_DISCARD, 0, &res);
89     XMMATRIX mvp = XMLoadFloat4x4(&m_modelMtx) *
90         XMLoadFloat4x4(&m_viewMtx) * XMLoadFloat4x4(&m_projMtx);
91     memcpy(res.pData, &mvp, sizeof(XMMATRIX));
92     m_device.context()->Unmap(m_cbMVP.get(), 0);
93 }
```

Listing 28: Metoda `Update` klasy `DxApplication`

Na razie do macierzy modelu domnożymy mały obrót wokół osi Y. O ile wystarczy to dla naszego przykładu, takie rozwiązanie nie jest najlepsze z dwóch powodów.

Po pierwsze nie jest ono stabilne numerycznie — poprawa wymagałaby pamiętania kumulatywnego konta obrotu i tworzenia macierzy modelu za każdym razem od nowa.

Po drugie zmiany zależą od szybkości działania programu. Stały obrót przy każdej klatce spowoduje, że im szybciej nasz program rysuje klatki, tym szybciej kostka się obraca. Lepszym rozwiązaniem byłoby zdefiniowanie stałej prędkości obrotowej. Przemnażając ją przez czas

który upłynął od poprzedniej klatki animacji, uzyskalibyśmy kąt obrotu.

Nowa macierz modelu, po wymnożeniu przez macierze kamery i rzutowania trafić musi do bufora stałych na karcie graficznej. Aktualizacja zawartości bufora nie jest jednak prosta.

Najpierw musimy uzyskać wskaźnik do danych bufora poprzez jego zmapowanie[38]. Operacja ta uniemożliwia dostęp do bufora z poziomu GPU do czasu odmapowania.[39] Flaga `D3D11_MAP_WRITE_DISCARD` sugeruje, że nadpiszemy cały bufor i nie będziemy odczytywać poprzednio zapisanych tam wartości.[40]

Dane możemy potem przekopiować (zwykłym `memcpy`) po czym musimy zwolnić bufor.

Istnieje inny sposób kopiowania danych do pamięci karty graficznej: metoda `UpdateSubresource`[41] kontekstu. Używana powinna być ona jednak wtedy gdy aktualizacje występują rzadko i gdy mamy pewność, że nie nastąpi konflikt w próbie dostępu do zasobu pomiędzy kartą graficzną i procesorem. Z tego powodu użycie tej metody z buforami stworzonymi z flagą `D3D11_USAGE_DYNAMIC` w polu `Usage` jest niemożliwe.

8. Po uruchomieniu programu na ekranie zobaczyć powinniśmy obracający się kolorowy sześcian, jak zaprezentowano na rys. 5.

Ćwiczenie 1. Zmodyfikuj sposób wyznaczania macierzy modelu w celu rozwiązania dwóch problemów wymienionych w punkcie 7. Program powinien przechowywać kumulatywny kąt zwiększany w metodzie `Update` klasy `DxApplication` o wartość prędkości kątowej (stałej $\frac{\pi}{4}\mathrm{s}^{-1}$), przemnożonej przez upływ czasu od poprzedniej klatki. Macierz modelu tworzona powinna być od nowa na podstawie sumarycznego kąta. Rozdzielczość standardowych funkcji do odczytywania znaczników czasu może być zbyt niska na nasze potrzeby (czas pomiędzy klatkami wynosić może <1ms). Zalecane jest użycie zegara wysokiej rozdzielczości[42] — funkcje `QueryPerformanceCounter`[43] i

[38] https://docs.microsoft.com/en-gb/windows/desktop/api/d3d11/nf-d3d11-id3d11devicecontext-map

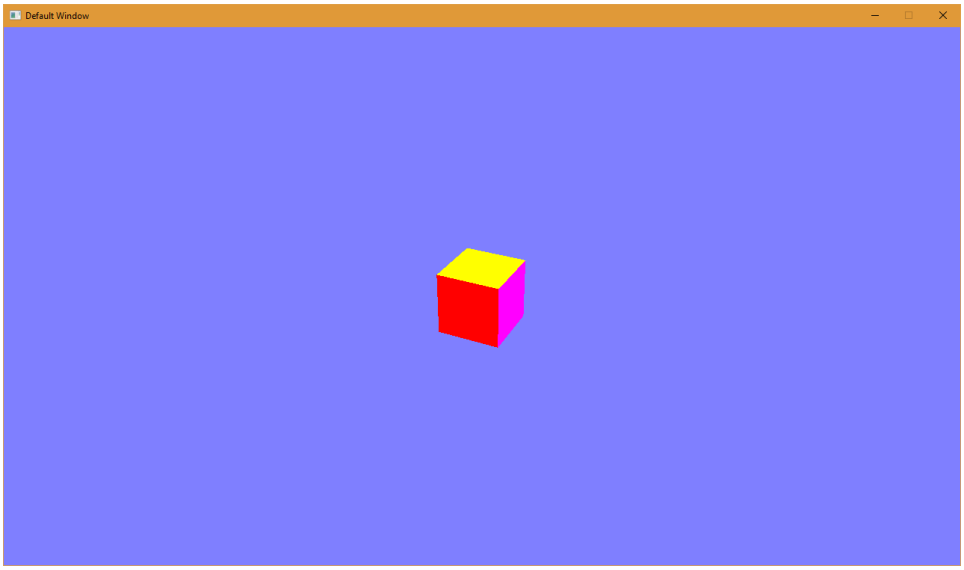
[39] Może być tu wymagane również zmapowanie adresu w pamięci RAM karty graficznej do przestrzeni adresów procesora.

[40] Jeśli GPU używa bufora nie będziemy wtedy czekać na jego zwolnienie — sterownik karty graficznej stworzy nowy bufor do którego zapiszemy dane, a zostanie on zamieniony ze starym gdy GPU go zwolni.

[41] https://docs.microsoft.com/en-gb/windows/desktop/api/d3d11/nf-d3d11-id3d11devicecontext-updatesubresource

[42] https://docs.microsoft.com/en-gb/windows/desktop/sysinfo/acquiring-high-resolution-time-stamps, https://docs.microsoft.com/en-us/windows/desktop/dxtecharts/game-timing-and-multicore-processors

[43] https://msdn.microsoft.com/library/windows/desktop/ms644904



Rysunek 5: Oczekiwany końcowy wygląd okna programu

`QueryPerformanceFrequency`[44]

Ćwiczenie 2. Dodaj możliwość kontrolowania nachylenia kamery i jej odległości od kostki za pomocą myszy. Przesuwanie myszą w pionie z wciśniętym lewym klawiszem powinno zmieniać kąt nachylenia (macierz R_X — kąt nie powinien wyjść poza zakres $[-\pi, \pi]$). Przesuwając mysz w pionie z wciśniętym modyfikowana powinna być odległość (macierz T_Z, wartość przesunięcia nie powinna wyjść poza zakres $[0, 50]$). W celu obsługi zdarzeń myszy przedefiniuj w klasie `DxApplication` wirtualną metodę `ProcessMessage` z klasy bazowej. Otrzymuje ona wszystkie wiadomości systemowe przychodzące do okna aplikacji.

Ćwiczenie 3. Narysuj na ekranie drugą kostkę, która nie będzie się obracać. Zamiast tego przesunięta będzie względem pierwszej o -10 jednostek wzdłuż osi X. Skorzystaj z istniejącego modelu, lecz zdefiniuj drugą macierz modelu reprezentującą translację. Model narysuj ponownie w metodzie `Render` klasy `DxApplication` wywołując `DrawIndexed` drugi raz. W tym wypadku jednak aktualizacja zawartości bufora stałych nie może być w metodzie `Update`, lecz przed każdym wywołaniem `DrawIndexed`.

[44] https://msdn.microsoft.com/library/windows/desktop/ms644905