Programowanie w API Graficznych

Laboratorium

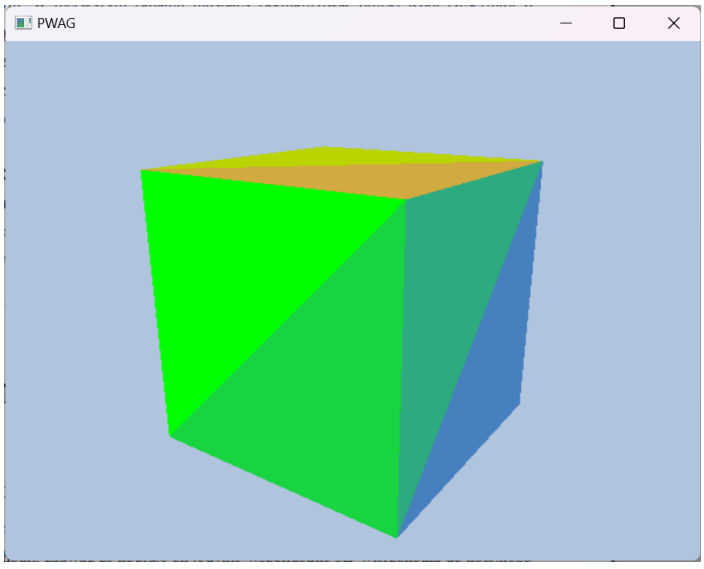
DirectX 12 – ćwiczenie 3

1 Cele ćwiczenia

W tym ćwiczeniu zapoznasz się z procesem tworzenia tekstur w DirectX. Jest to złożony proces i poza znajomością DirectX wymaga także znajomości bibliotek graficznych umożliwiających wczytanie pliku graficznego (BMP, JPG, PNG..) do pamięci RAM, dlatego zadanie to zostało podzielone na trzy części. W pierwszej części tego ćwiczenia omówione zostanie podpinanie tekstury do potoku renderującego oraz wykorzystanie jej w programie cieniującym piksele. Następnie omówione zostaną samplery – są to obiekty odpowiedzialne za próbkowanie tekstury. W ostatnim zadaniu nauczysz się tworzyć zasoby tekstury z dowolnego pliku graficznego i ładować je do pamięci karty graficznej.

2 Zadania

Zaraz po uruchomieniu aplikacji powinieneś zobaczyć kolorowy sześcian. Wygląda on znajomo, ponieważ podobny obiekt stworzyłeś w poprzednim zadaniu. W tym zadaniu wykorzystano kod z poprzedniego ćwiczenia, zmianie uległ jedynie kod odpowiedzialny za tworzenie geometrii modelu 3D, w poprzednim zadaniu każdy wierzchołek sześcianu posiadał przypisane mu współrzędne 3D i kolor, w tym zadaniu do każdego wierzchołka przypisano współrzędne 3D oraz współrzędne tekstury. Twoim zadaniem będzie wczytanie tekstury do pamięci karty graficznej i naniesienie jej na wspomniany sześcian.



2.1 Nakładanie tekstury na obiekt 3D

W tym zadaniu nauczysz się wczytywać tekstury DDS do pamięci karty graficznej i próbkować je w programie cieniującym piksele. W przypadku plików DDS programiści Microsoftu udostępnili funkcje odpowiedzialne za wczytanie pliku graficznego do pamięci karty graficznej,

dzięki temu w tym zadaniu skupisz się jedynie na konfiguracji potoku renderującego i napisaniu prostego programu cieniującego nakładającego teksturę na obiekt 3D.

2.1.1 Root signature

Tekstura, podobnie jak stworzony przez ciebie w poprzednim zadaniu bufor stałych, jest zasobem, a więc także wymaga utworzenia widoku zasobu. Tworzenie zasobu należy rozpocząć od zaktualizowania obiektu Root Signature, przejdź do funkcji `RenderWidget::BuildRootSignature()`. Z poprzednich ćwiczeń powinieneś już znać proces tworzenia obiektu Root Signature, w pierwszym kroku należy dodać parametry, które opisują jego strukturę (tzw. Root Parameters). Poniżej umieszczono strukturę opisującą obiekt Root Signature zawierający tylko bufor stałych.

```
CD3DX12_ROOT_PARAMETER slotRootParameter[1];
slotRootParameter[0].InitAsConstantBufferView(0);
```

Twoim zadaniem jest dodanie informacji o zasobie (ang. shader resource view), który będzie reprezentował teksturę, do obiektu Root Signature. Istnieje wiele sposobów na to, w tym zadaniu nauczysz się dodawać parametry Root Signature za pomocą struktury CD3DX12_DESCRIPTOR_RANGE, zwanej dalej deskryptorem, pozwala ona na tworzenie grup zasobów. Utwórz deskryptor i dodaj jeden parametr reprezentujący widok zasobów (shader resource view):

```
CD3DX12_DESCRIPTOR_RANGE resourceTable;
resourceTable.Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 1, 0);
```

Pierwszy argument określa typ zasobu, drugi argument określa ilość zasobów, a ostatni rejestr bazowy w programie cieniującym. Tekstura przypisana do takiego widoku będzie dostępna w programie cieniującym pod rejestrem numer zero (t0):

```
Texture2D gTexture : register(t0);
```

Jeżeli będziesz potrzebować większej ilości tekstur to jedyne co musisz zrobić to dopisać kolejny parametr do tak utworzonego deskryptora, a następnie przypisać mu wolny indeks rejestru tekstury w programie cieniującym. W kolejnym kroku dodaj deskryptor do listy parametrów. Służy do tego funkcja InitAsDescriptorTable, pierwszy argument określa ilość zasobów opisanych przez deskryptor, w tym przypadku jest to tylko jedna tekstura. Drugi argument jest wskaźnikiem do deskryptora, a trzeci określa widoczność zasobów w potoku renderującym.

```
CD3DX12_DESCRIPTOR_RANGE resourceTable;
resourceTable.Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 1, 0);
```

```
	CD3DX12_ROOT_PARAMETER slotRootParameter[2];
	slotRootParameter[0].InitAsConstantBufferView(0);
	slotRootParameter[1].InitAsDescriptorTable(1, &resourceTable,
D3D12_SHADER_VISIBILITY_PIXEL);
```

2.1.2 Sampler

Sampler jest obiektem odpowiedzialnym za próbkowanie tekstury. Pozwala on wybrać sposób filtrowania tekstury, oraz określić co się stanie jeżeli współrzędne próbkowania wyjdą poza zakres tekstury. Utwórz jeden deskryptor samplera:

```
	const CD3DX12_STATIC_SAMPLER_DESC sampler(
		0, // shaderRegister
		D3D12_FILTER_MIN_MAG_MIP_POINT,   // filter
		D3D12_TEXTURE_ADDRESS_MODE_WRAP,  // addressU
		D3D12_TEXTURE_ADDRESS_MODE_WRAP,  // addressV
		D3D12_TEXTURE_ADDRESS_MODE_WRAP); // addressW
```

Następnie przypisz deskryptor zasobu oraz deskryptor sampler do obiektu Root Signature:

```
	CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(2, slotRootParameter, 1, &sampler,
		D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);
```

Projekt powinien się poprawnie kompilować i rysować kolorowy sześcian, na tym etapie jeszcze bez tekstury.

2.1.3 Widok zasobu (shader resource view)

Posiadasz już poprawnie skonfigurowany obiekt Root Signature, następnym krokiem jest utworzenie zasobu tekstury i przypisanego mu widoku zasobu (shader resource view). Przejdź do funkcji `RenderWidget::LoadDDSTexture`. Funkcja ta robi trzy rzeczy:

- Tworzy zasób DirectX i ładuje do niego teksturę. DirectX 12 jest interfejsem niskopoziomowym, a przez to nie posiada funkcji do ładowania plików graficznych. Ty, jako programista jesteś za to odpowiedzialny! W przykładowym kodzie udostępnionym przez programistów DirectX znajdziesz funkcję `DirectX::CreateDDSTextureFromFile12`, w prosty sposób pozwala ona wczytać plik graficzny o formacie DDS do pamięci karty graficznej, skorzystamy z niej w tym zadaniu.
- Pobiera uchwyt do sterty SRV (shader resource view). W tym przypadku mowa jest o stercie pamięci karty graficznej, znajdują się w niej widoki do zasobów karty graficznej.
- Tworzy widok zasobu, będzie on ci potrzebny do przypisania tekstury do potoku renderującego.

Funkcja `RenderWidget::LoadDDSTexture` nie została jeszcze użyta w aplikacji, użyj jej do wczytania tekstury, wykorzystaj jeden z załączonych do projektu plików graficznych DDS:

gg_logo_sama_kostka.dds lub WoodCrate.dds. Nie zapomnij odkomentować i uzupełnić ostatnią linię w kodzie funkcji `LoadDDSTexture`! Jeżeli wykonałeś poprawnie zadanie to tekstura zostanie poprawnie wczytana do pamięci karty graficznej, nie jest ona jeszcze wykorzystana w kodzie programu cieniującego wiec póki co nie zobaczysz żadnych wizualnych zmian w aplikacji.

2.1.4 Przypisanie tekstury do potoku renderującego

Ostatnim etapem jest przypisanie tekstury do potoku renderującego i wykorzystanie jej w programie cieniującym. W pierwszym zadaniu musiałeś zaktualizować obiekt Root Signature o element reprezentujący teksturę(shader resource view), wykorzystałeś do tego DescriptorTable, dlatego teraz powinieneś użyć funkcji `SetGraphicsRootDescriptorTable` do podpięcia tekstury do potoku renderującego. Przejdź do funkcji `RenderWidget::Draw()` i przypisz teksturę do potoku. Jeżeli program się poprawnie kompiluje to przejdź do kolejnego zadanie.

2.1.5 Program cieniujący

Teraz, kiedy wszystkie potrzebny zasoby zostały już stworzone i przypięte do potoku renderującego możesz rozpocząć pisanie programu cieniującego, którego zadaniem będzie naniesie tekstury na wyświetlany sześcian. Na początek proponuję ci dokładnie zapoznać się ze strukturą danych umieszczonych w buforach wierzchołków i indeksów oraz z strukturą samego wierzchołka – InputLayout, kod odpowiedzialny za generowanie sześcianu znajdziesz w funkcji `CreateBoxGeometry()`, zwróć uwagę na dodatkowe współrzędne tekstur.

Pora napisać program cieniujący, otwórz plik shaders.fx. Przyjrzyj się programowi cieniującemu wierzchołki, zauważ że kopiuje on jedynie współrzędne 3D wierzchołka do kolejnego etapu potoku renderującego, popraw kod tak aby współrzędne tekstury także zostały skopiowane do kolejnego etapu. Program cieniujący piksele wymaga całkowitego przepisania, w pierwszej kolejności zwróć uwagę jakie dane są przekazywane z poprzedniego etapu. Wcześniej utworzyłeś teksturę oraz sampler, zadeklaruj je teraz w programie cieniującym i wykorzystaj do naniesienia tekstury na sześcian, służy do tego funkcja Sample(). Deklaracja tekstury i samplera w HLSL:

```
Texture2D    NazwaTekstury : register(numer rejestru);
SamplerState NazwaSamplera  : register(numer rejestru);
```

Jeżeli poprawnie wykonałeś wszystkie kroki w zadaniu, to powinieneś zobaczyć sześcian z naniesioną na niego wybraną przez ciebie teksturę, tak jak na obrazku poniżej. W przeciwnym wypadku sprawdź za pomocą debuggera graficznego (np. Render Doc), czy w pamięci karty graficznej znajduje się poprawnie załadowana tekstura.

2.2 Próbkowanie tekstury

W tym zadaniu poznasz lepiej możliwości samplera. Służy on do próbkowania tekstury, jako programista możesz zmodyfikować szereg jego parametrów, w tym metodę filtrowania. Zanim jednak zaczniemy modyfikować kod samplera to powinieneś poznać sposoby wczytywania pozostałych popularnych plików graficznych, takich jak PNG, JPG, BMP, etc... Jak już zostało to wcześniej wspomniane, DirectX12 to api niskopoziomowe i nie posiada żadnych funkcji odpowiedzialnych za wczytywanie tekstur. Jako programista sam musisz napisać ten kod, możesz skorzystać z kilku gotowych projektów, jak na przykład OpenCV, QT czy SDL Image. Samo wczytanie pliku graficznego do pamięci RAM to dopiero połowa sukcesu, musisz następnie utworzyć odpowiednie zasoby w karcie graficznej i załadować do nich teksturę. Jest to proces dość złożony i wybiega poza zakres tego laboratorium, dlatego funkcjonalność ta została już napisana za ciebie – otwórz funkcję `RenderWidget::LoadTexture`. Na początku plik graficzny jest wczytywany do pamięci RAM przy pomocy funkcji `DirectXHelper::LoadTextureToBuffer`, możesz ją otworzyć, korzysta ona z udostępnionego prze Microsoft api WIC (Windows Imaging Component), jest to domyślnie zainstalowane w systemie Windows api graficzne, pozwala ono wczytywać i przetwarzać pliki graficzne. Prześledź kod tej funkcji i zapoznaj się procesem wczytywania tekstury do pamięci RAM. Wróć do funkcji LoadTexture, w dalszej jej części tworzony jest desktryptor tekstury (`D3D12_RESOURCE_DESC textureDesc`) oraz dwa zasoby odpowiedzialne za wczytanie tekstury do karty graficznej, przeanalizuj dokładnie kod tej funkcji. Czy wiesz dlaczego DirectX12 korzysta z dwóch zasobów w celu wczytania tekstury do pamięci karty graficznej? Spróbuj wczytać inne tekstury dołączone do tego projektu.

2.2.1 Filtrowanie

Te ćwiczenie powinieneś wykonać w trybie pełnoekranowym – otwórz plik main.cpp i zmień wartość odpowiedniego argumentu funkcji `System::Initialize`, po ponownym uruchomieniu aplikacja uruchomi się w trybie pełnoekranowym, możesz ją zamknąć klikając klawisz esc.

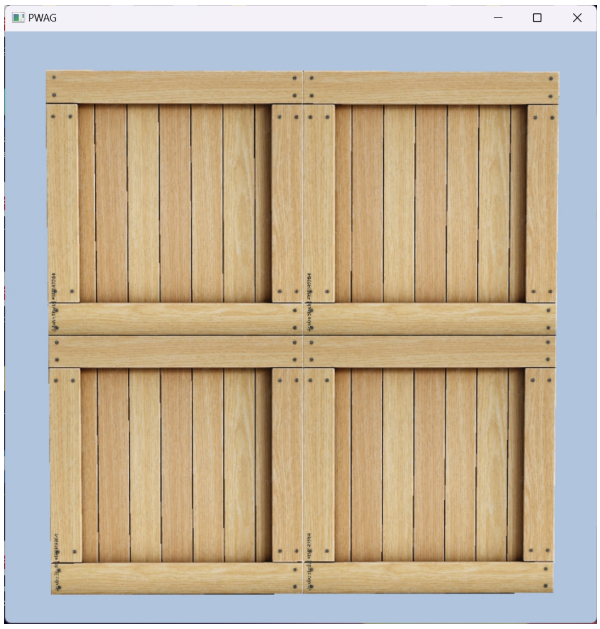
Twoim zadaniem jest zmodyfikowanie samplera, tak aby nałożona tekstura była filtrowana za pomocą filtru:

- `D3D12_FILTER_MIN_MAG_MIP_POINT`
- `D3D12_FILTER_MIN_LINEAR_MAG_MIP_POINT`
- `D3D12_FILTER_ANISOTROPIC`

Przyjrzyj się sześcianowi pod różnymi kątami oraz sprawdź kilka różnych tekstur. Jak technika filtrowania wpływa na wyświetlaną teksturę?

2.2.2 Współrzędne tekstury

Otwórz funkcję `CreateBoxGeometry()`, jak zapewne pamiętasz, odpowiada ona za wygenerowanie współrzędnych wyświetlanego sześcianu. Zmodyfikuj współrzędne tekstury tak aby uzyskać następujący obraz:



Podobny efekt możesz uzyskać manipulując rozmiarem współrzędnych tekstury w programie cieniującym wierzchołki. Jak zapewne zauważyłeś, niektóre współrzędne tekstury wyszły poza jej zakres (zakres współrzędnych tekstury to <0.0, 1.0>), o tym co zostanie wyświetlone poza zakresem decyduje sampler, zmodyfikuj ustawienia samplera i sprawdź jak poniższe tryby adresowania wpływają na renderowany obraz:

- `D3D12_TEXTURE_ADDRESS_MODE_WRAP`

- D3D12_TEXTURE_ADDRESS_MODE_MIRROR
- D3D12_TEXTURE_ADDRESS_MODE_CLAMP
- D3D12_TEXTURE_ADDRESS_MODE_BORDER
- D3D12_TEXTURE_ADDRESS_MODE_MIRROR_ONCE

2.3 Dwie tekstury

Twoim ostatnim zadaniem jest zmodyfikowanie kodu tak, aby wczytał dwie tekstury jednocześnie. Następnie, w programie cieniującym piksele pomnożysz kolory obu tekstur (tzw. blendowanie) przez siebie. Nie jest to proste zadanie, rozpocznij je od zmodyfikowania obiektu RootSignature, dodaj do niego informację o kolejnej teksturze:

```
CD3DX12_DESCRIPTOR_RANGE resourceTable[2];
resourceTable[0].Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 1, 0);
resourceTable[1].Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, ?, ?);
```

Skoro chcemy wyświetlić dwie tekstury, to potrzebny nam będzie także drugi widok zasobu (shader resource view). Wyświetl funkcje `RenderWidget::LoadDDSTexture`, tutaj napotkasz na pewien problem – w jaki sposób umieścić drugą teksturę na stercie (m_srvDescriptorHeap)?
Odpowiedź: należy pobrać uchwyt do tej sterty i go inkrementować, robi się to w analogiczny sposób jak inkrmenentacja wskaźniki w C++. Spójrz na poniższy kod:

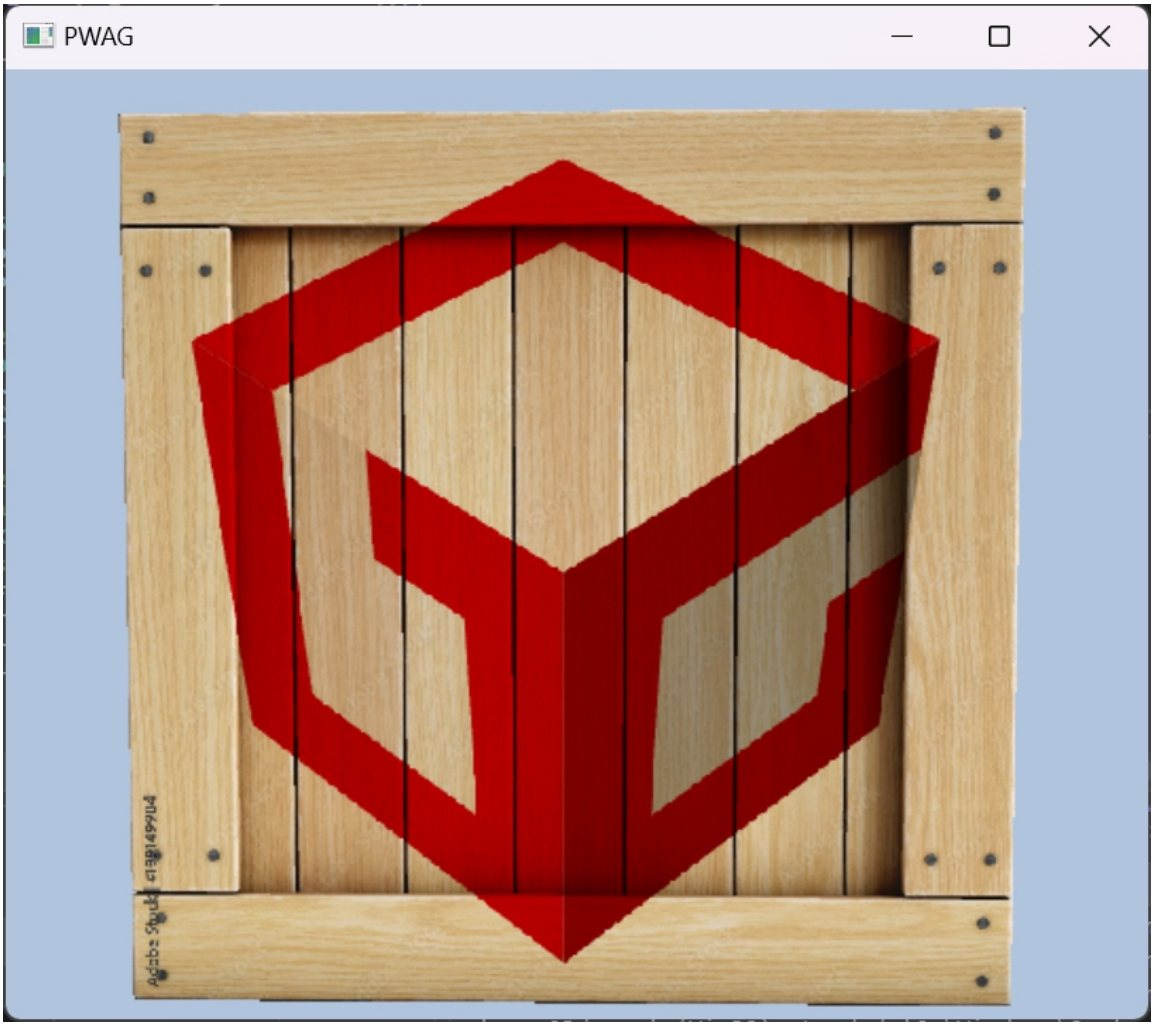
```
CD3DX12_CPU_DESCRIPTOR_HANDLE hDescriptor(m_srvDescriptorHeap-
>GetCPUDescriptorHandleForHeapStart());
	static auto size = m_dxDevice-
>GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV);
	hDescriptor.Offset(1, size);
```

Funkcja GetCPUDescriptorHandleForHeapStart() zwraca uchwyt sterty, aktualnie znajduje się na niej widok zasobu pierwszej tekstury, aby go nie nadpisać przesuwamy uchwyt za pomocą funkcji Offset. Teraz wskazuje on na niezapisany obszar pamięci, możemy więc za jego pomocą utworzyć drugi widok zasobu:

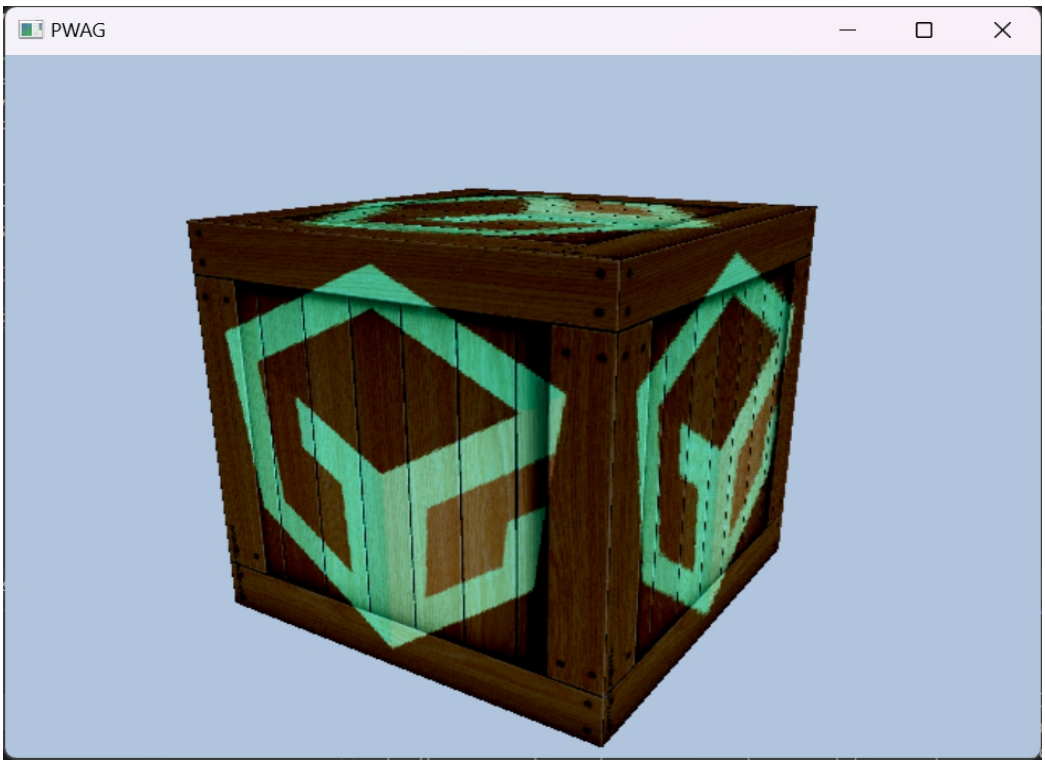
```
	// 2. Get the descriptor heap handle
	CD3DX12_CPU_DESCRIPTOR_HANDLE hDescriptor(m_srvDescriptorHeap-
>GetCPUDescriptorHandleForHeapStart());
	static auto size = m_dxDevice-
>GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV);
	hDescriptor.Offset(1, size);

	// 3. Shader resouce view
	D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
	srvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;
	[additional code was removed]
	m_dxDevice->CreateShaderResourceView(m_ddsTextureResource.Get(), &srvDesc,
hDescriptor);
```

Nie zapomnij zwiększyć rozmiaru sterty m_`srvDescriptorHeap`, kod za to odpowiedzialny znajduje się w funkcji `RenderWidget::CreateDescriptorHeaps()`. Sugeruję stworzenie drugiej funkcji, np LoadTexture2(...), która będzie odpowiedzialna za załadowanie drugiej tekstury. Jak już zauważyłeś, w przeciwieństwie do sterty pamięci RAM, w przypadku kart graficznych to ty odpowiadasz za zarządzanie pamięcią, jeżeli tworzona przez ciebie aplikacja potrzebuje kilku tekstur jednocześnie, to dla każdej z nich musisz utworzyć widok zasobu i zaalokować dla niego wolne miejsce na stercie. Pozostało ci już tylko zmodyfikowanie programu cieniującego, pomnóż przez siebie kolory spróbowane z obu tekstur. Jeżeli poprawnie wykonałeś wszystkie kroki to powinieneś uzyskać efekt podobny do poniższego:



Ja w tym przykładzie wykorzystałem tekstury WoodCrate3.jpg i gg_logo_sama_kostka.dds. Obie tekstury pomnożyłem przez siebie, możesz wypróbować inne operacje matematyczne aby uzyskać ciekawe efekty.



Ciekawe efekty uzyskasz modyfikując także współrzędne jednej z tekstur.