Politechnika Poznańska
Wydział Informatyki
Instytut Informatyki

Praca dyplomowa inżynierska

PROJEKT I IMPLEMENTACJA GRY KOMPUTEROWEJ Z WYKORZYSTANIEM TECHNOLOGII MICROSOFT DIRECTX 11

Krzysztof Marciniak, 106574
Piotr Przybysz, 106602
Mikołaj Szychowiak, 106580
Ryszard Wojtkowiak, 106609

Promotor
dr inż.Witold Andrzejewski

Poznań, 2015 r.

Tutaj przychodzi karta pracy dyplomowej;
oryginał wstawiamy do wersji dla archiwum PP, w pozostałych kopiach wstawiamy ksero.

Spis treści

Rozdział 1

Wstęp

1.1 Cel i zakres pracy

Rozdział 2

Przegląd zagadnień teoretycznych

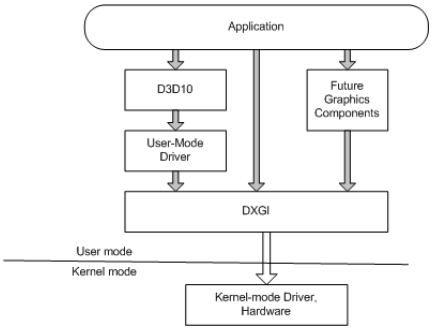
2.1 Zarys działania Microsoft DirectX API

2.1.1 Struktura i podstawowe pojęcia

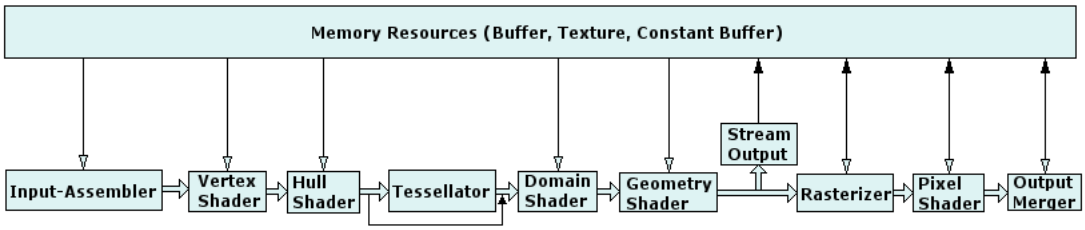
Microsoft DirectX to interfejs programistyczny (API, ang. *application programming interface*) do tworzenia aplikacji multimedialnych. Składają się na nie przede wszystkim:

- DirectDraw, Direct2D/Direct3D (komponenty odpowiedzialne za rysowanie grafiki),
- DirectSound, DirectMusic (obsługa dźwięków),
- DirectInput (obsługa wejścia - myszy, klawiatury, kontrolerów itp.).

DirectX pozwala na tworzenie aplikacji w trzech językach - C#, Visual Basic oraz C++, przy czym w praktyce spotyka się głównie aplikacje stworzone w C# oraz C++. Jak widać na rysunku 2.1, struktura DirectX jest wielopoziomowa i, poza odwołaniami do API oraz komponentami jak na przykład Direct3D 11 (D3D11), obejmuje także DXGI (Infrastruktura Graficzna DirectX, ang. *Microsoft DirectX Graphics Infrastructure*), które – jako najniższa warstwa – komunikuje się bezpośrednio ze sterownikiem znajdującym się w przestrzeni jądra systemu operacyjnego. Wykorzystanie komponentów COM (ang. *Component Object Model*) pozwala na łatwą rozbudowę oraz jasny podział funkcjonalności.
Podstawowymi i koniecznymi do zrozumienia działania DirectX są koncepcje urządzenia (ang. *device*) oraz jego kontekstu (ang. *device context*).
Urządzenie, reprezentowane w wersji 11. przez interfejs ID3D11Device, reprezentuje kartę graficzną i może służyć do tworzenia zasobów oraz pobierania informacji o jej możliwościach (ang. *capabilities*), tj. oferowanych przez nią funkcjonalnościach takich jak np. obsługa podwójnej precyzji w programach cieniujących.
Kontekst urządzenia z kolei, jak wskazuje nazwa, określa kontekst użycia urządzenia, co jednoznacznie pokazuje, iż do prawidłowej pracy z urządzeniem powiązany powinien być co najmniej jeden kontekst. Pozwala on głównie na ustawienie stanu w procesie renderingu oraz ustawienie prawidłowych komend wykorzystujących zasoby karty graficznej do wygenerowania obrazu (bezpośrednio na ekran lub do pośredniczącej tekstury, która może zostać wykorzystana później).



Rysunek 2.1: Architektura DirectX



Rysunek 2.2: Proces generowania obrazu w DirectX

Wyróżniamy dwa rodzaje kontekstów: bezpośredni/zwykły (*forward*) oraz opóźniony (*deferred*). Bezpośredni wykonuje komendy od razu gdy zostają wywołane, podczas gdy opóźniony pozwala na zapisanie ich na odpowiedniej liście, dzięki czemu mogą zostać wykonane później (jest to przydatne zwłaszcza w przypadku aplikacji wielowątkowych).

2.1.2 Proces renderingu

Przebieg wygenerowania grafiki (renderingu) opisywany przez tzw. *pipeline* jest złożony z wielu etapów (ang. *stage*), z których część może być konfigurowana jedynie przez wywołanie odpowiednich komend API poprzedzonych prefiksem będącym skrótem nazwy danego etapu (np. *IA* dla etapu *Input-Assembler*), a część opisywana jest przez programy cieniujące (ang. *shader*).
W pipeline DirectX 11 wyróżnia się następujące etapy:

- Input-Assembler - odpowiada za wczytanie wierzchołków w sposób opisany przez prymityw (trójkąt, czworokąt itp.),
- Vertex Shader - cieniowanie wierzchołków, ustalanie dla nich wartości początkowych zmiennych jak np. kolor, wektor normalny itp. ,
- Hull Shader - pierwszy etap teselacji (zagęszczania siatki), przygotowuje siatkę do zagęszczenia przez ustalenie punktów kontrolnych,
- Tessellator - zagęszcza siatkę wprowadzając dodatkowe prymitywy zastępujące podstawowy i zwraca nowe współrzędne teksturowania,
- Domain Shader - generuje nowe pozycje wierzchołków na podstawie dwóch poprzednich etapów,
- Geometry Shader - pozwala zastąpić prymityw wejściowy innym prymitywem,
- Rasterizer - odpowiada za śpłaszczenieóbrazu tak, aby mógł zostać wyświetlony na ekranie,
- Pixel Shader - pozwala zmieniać kolor piksela w wynikowym obrazie,
- Output Merger - generuje końcowy obraz łącząc w odpowiedni sposób (opisany przez komendy takie jak np. OMSetRenderTargets) informacje z poprzedniego etapu.

Wszystkie te etapy przedstawione zostały na Rys. 2.2. Strzałki określają czy dany etap korzysta jedynie z odczytu danych z karty, czy też pozwala na ich zapis (lub oba równocześnie).

Ostatnim zasługującym na uwagę jest fakt, iż w większości aplikacji wraz z DirectX wykorzystywana jest technologia WinAPI pozwalająca na tworzenie aplikacji graficznych pod platformę Windows. Głównym jej mechanizmem jest pętla komunikatów, których odebranie warunkuje sposób przetwarzania informacji w aplikacji (np. wciśnięcie klawisza powoduje przesunięcie modelu). Komunikaty mogą zostać odczytane w sposób blokujący (przez funkcję *GetMessage*) lub nieblokujący (*PeekMessage*). Z oczywistych względów (tj. narzutu czasowego), w aplikacjach generujących obraz w czasie rzeczywistym wykorzystywana jest wyłącznie funkcja nieblokująca.

2.2 Deferred Shading

Podstawowym sposobem obliczania oświetlenia, uwzględniając opisany wcześniej proces generacji obrazu, jest wyliczenie oświetlenia bezpośrednio dla każdego obiektu. Wartości takie jak wektory normalne, współrzędne teksturowania i inne używane we wspomnianym procesie są wczytywane w shaderze wierzchołków, a następnie interpolowane we fragment/pixel shaderze. W przypadku wielu źródeł oświetlenia (na przykład 200 lub więcej), dla każdego obiektu (niezależnie od tego, czy jest widoczny) należy sprawdzić wszystkie źródła światła, co oznacza złożoność

$$O(\ liczba\_pikseli\_per\_obiekt\ *\ liczba\_zrodel\_swiatla\ ) \tag{2.1}$$

co z kolei w wielu przypadkach jest nie do zaakceptowania (z 2.1 wynika, iż zwiększanie liczby obiektów zmniejsza liczbę źródeł światła, które możemy użyć).
Jeśli jednak proces obliczania oświetlania przesuniemy do oddzielnego etapu następującego po wyliczeniu tego, które obiekty są aktualnie widoczne, otrzymamy złożoność

$$O(\ liczba\_pikseli\ *\ liczba\_zrodel\_swiatla\ ) \tag{2.2}$$

Jak widać z 2.2, podejście to nie wprowadza już zależności między liczbą obiektów a liczbą źródeł światła. Pozwala to uzyskać o wiele lepszą wydajność w przypadku scen z wieloma złożonymi obiektami oraz złożonym oświetleniem. Możliwe jest również wprowadzenie wielu uprawnień, jak na przykład podział obrazu na wiele "kafelek"(ang. *tile*), z których każda może zostać obliczona przez wątki na karcie graficznej - technika ta nosi nazwę *tiled deferred rendering* i jest powszechnie stosowana w popularnych silnikach graficznych.
Uzyskanie tego efektu wymaga utworzenia kilku tekstur pośredniczących, które łącznie mają nazwę *G-Buffera*. Jest on wykorzystywany do przetrzymywania efektów pośrednich procesu renderingu oraz wyliczenia obrazu końcowego. Format G-Buffera zmienia się w zależności od zastosowania oraz algorytmu obliczania oświetlenia i osoby odpowiadającej za jego implementację, jednak najczęściej wykorzystywane są tekstury:

- wektorów normalnych,
- koloru/albedo,
- głębokości.

Pomimo wielu zalet, deferred shading ma jednak swoje wady. Obliczanie oświetlenia z gotowego obrazu uniemożliwia łatwe obliczenie kolorów w przypadku przezroczystych obiektów, zaś wyświetlanie obrazu końcowego w postaci tekstury wymaga wykorzystania bardziej skomplikowanych algorytmów obliczania antyaliasingu, co z kolei obciąża kartę graficzną. Mimo to deferred shading, w zmodyfikowanych odmianach, jest powszechnie wykorzystywany w wielu popularnych grach komputerowych oraz innych aplikacjach renderujących obraz w czasie rzeczywistym.

2.3 Podpowierzchniowe Rozproszenie Światła (Subsurface Scattering)

2.4 Depth of Field

Rozdział 3

Przegląd narzędzi

Jedną z najważniejszych kwestii podczas tworzenia gry komputerowej jest prawidłowy dobór narzędzi, ponieważ decyduje to nie tylko o komforcie pracy, ale także o jakości końcowego produktu. Podczas tego procesu szczególny nacisk powinien zostać położony na doborze silnika graficznego, między innymi ze względu na dużą dywersyfikację narzędzi należących do tej kategorii oraz mnogość funkcjonalności przez nie oferowanych.

Ze względu na specyfikę tego rozdziału, w kolejnych punktach opisane zostaną przetestowane rozwiązania wraz z krótkim podsumowaniem w formie listy jego zalet i wad. Warto w tym miejscu również wspomnieć, iż wszystkie analizowane narzędzia oferują możliwość wykorzystania DirectX 11 API, choć niekoniecznie bezpośrednio.

3.1 Własny silnik graficzny w języku C++

Proces doboru narzędzi rozpoczęto od zaprojektowania i stworzenia własnego silnika graficznego w języku C++ w wersji 11 z wykorzystaniem środowiska Microsoft Visual Studio 2012 na platformę Windows. Pozwoliło to nie tylko na praktyczne wykorzystanie umiejętności nabytych podczas uczestnictwa w zajęciach z Inżynierii Oprogramowania, ale także poznać w praktyce wykorzystanie Microsoft DirectX API w wersji 11 na najniższym dostępnym poziomie. Tworzenie tego rodzaju oprogramowania wymaga jednak nie tylko odpowiedniej ilości czasu, ale także dobrego zaprojektowania interakcji między klasami oraz zrozumienia wielu zagadnień z zakresu grafiki komputerowej, w większości takich, które wykraczają poza program przedmiotu Grafika Komputerowa i Wizualizacja. Z ostatnich dwóch powodów (oraz faktu, iż tworzenie silnika nie było tematem pracy inżynierskiej), najpierw ograniczono rozwój oprogramowania do jednej osoby, a następnie zrezygnowano z wykorzystania go w pracy, uzasadniając tę decyzję wysoką czasochłonnością wytwarzania owego narzędzia.

Zalety:

- większe możliwości w zakresie wykorzystania DirectX API
- lepsza znajomość możliwości oferowanych przez oprogramowanie
- brak kosztów

Wady

- czasochłonność
- wysoki próg wejścia (znajomość m.in. C++ oraz DirectX API)
- konieczność wytworzenia edytora poziomów i dodatnia odpowiednich funkcjonalności

3.2 Unreal Engine 4

Unreal Engine 4 (UE4) jest jednym z najpopularniejszych silników graficznych dostępnych na rynku, co jest złożeniem wielu czynników. Pierwszym z nich jest z pewnością niska cena płatnej licencji (subskrypcja miesięczna to koszt x$) oraz darmowy dostęp dla studentów zarówno w ramach licencji edukacyjnej (należy w tym wypadku zgłosić chęć wydania licencji w ramach

przedmiotu prowadzonego na uczelni) jak i w ramach GitHub Developers Pack (należy jedynie potwierdzić studencki adres e-mail oraz wykorzystać [ang. *redeem*] licencję dostępną na odpowiedniej podstronie serwisu GitHub w formie kodu [ang. *serial code*]). Drugim jest jakość generowanych (renderowanych) obrazów – wykorzystanie algorytmu Voxel Cone Tracing [na pewno ten?] (algorytm rozwiązywania zagadnienia globalnego oświetlenia w czasie rzeczywistym) pozwala uzyskać niemal fotorealistyczną grafikę, jednak kosztem wysokich wymagań sprzętowych. Unreal Engine, zarówno w wersji 3 jak i 4, oferuje dostęp do kodu źródłowego w języku C++, co - po poznaniu API udostępnianego przez twórców - pozwala szybko i wygodnie rozwijać logikę gry. Umiejętność programowania nie jest jednak wymagana do tego ze względu na obecność mechanizmu blueprintów (znaleźć tłumaczenie), który pozwala tworzyć kod z wykorzystaniem bloków oferowanych bezpośrednio w silniku graficznym (sprawdzić czy dokładnie tak jest). Ze względu na wysokie wymagania sprzętowe (brak możliwości uruchomienia na komputerach laboratoryjnych oraz komputerach 75% zespołu) ostatecznie odrzucono to rozwiązanie.

Zalety:

- tania (darmowa) licencja
- wygodny edytor i dostęp do API w języku C++
- niemal fotorealistyczna grafika

Wady:

- konieczność poznania API
- wysokie wymagania sprzętowe

3.3 Unity

3.4 OGRE

3.5 CryEngine 3

3.6 Unreal Engine 3

3.7 Unreal Development Kit (UDK)

3.8 Autodesk Maya

3.9 Autodesk 3D Studio Max

3.10 Blender

3.11 Adobe Flash - Scaleform

Rozdział 4

Praca własna

Proces twórczy został podzielony na dwa główne etapy: projektowanie i implementację. Ponieważ tworzenie gry wymaga przygotowania oraz doboru odpowiedniego środowiska, etapy te zostały poprzedzone analizą problemu oraz burzą mózgów na której zarysowały się wstępne wymagania funkcjonalne. Zdefiniowane zostały również główne wymagania pozafunkcjonalne, takie jak docelowa platforma obsługująca grę. W celu zapewnienia spójnej wizji gry wśród członków zespołu, postanowiono skonstruować dokument, zawierający opis wszystkich decyzji podjętych podczas tworzenia projektu oraz wyjaśnienia dotyczące wszystkich wymagań funkcjonalnych oraz pozafunkcjonalnych – Game Design Document (GDD), stanowiący jednocześnie załącznik 1 do niniejszej p racy. Pomysły zebrane podczas burzy mózgów zostały poddane wnikliwej analizie, co pozwoliło na stworzenie ogólna wersji projektu.

4.1 Projektowanie

4.1.1 Projekt silnika graficznego

4.1.2 Modele 3D

4.1.3 Projekt poziomu

4.2 Implementacja

4.2.1 Wykorzystanie technologii DirectX 11 w UDK

4.2.2 Zarządzanie projektem

W celu sprawnej organizacji pracy w zespole wykorzystano metody zarządzania projektami.

Początkowo użyto modelu kaskadowego. Jego zaletą jest sekwencyjność, pozwalająca oddzielić procesy analizy problemu, projektowania, implementacji oraz testowania i późniejszego utrzymania projektu. Metoda Waterfall, wykorzystująca ten model, nie jest jednak pozbawiona wad. Dotyczą one głównie dużych projektów, a więc nie miały miejsca w przypadku oprogramowania tworzonego w czteroosobowym zespole programistów. Korzystając z tej metody oszczędza się czas na planowaniu, faza ta zajmuje zaledwie 25% czasu pracy nad projektem. Rezultat końcowy zostaje ustalony jeszcze przed rozpoczęciem implementacji, podobnie jak poszczególne przyrosty implementacji. Każdy przyrost miał trwać 2 tygodnie i zawierał określone zadania. Rezultatem końcowym był produkt posiadający wartość biznesową, w tym przypadku - w pełni działająca gra. Co istotne, wartość biznesową produkt miał zyskać dopiero w przedostatnim przyroście.

W niedługim czasie po zaplanowaniu prac nad projektem okazało się, że metoda ta nie jest wystarczająca, zaczęło się pojawiać opóźnienie w pracach, które mogło spowodować pogorszenie jakości produktu. Zdecydowano się więc skorzystać z metod zwinnych. Metodyki Agile charakteryzują się stałą jakością, a sterowane są zakresem. W omówionej wcześniej metodzie Waterfall zakres był stały, zmieniała się jedynie jakość, a chęć utrzymania wysokiej jakości powodowała opóźnienie względem planu.

Ostatecznie zastosowano metodykę zwinną zbliżoną do metody Scrum. Zadania podzielono na część dotyczącą rozpoznania danego zagadnienia i część implementacyjną. Realizowane były tygodniowe sprinty (przyrosty). Na koniec każdego z nich otrzymywany był prototyp posiadający pewną funkcjonalność. Ze względu na wcześniejszy nieudany eksperyment z metodą Waterfall

wartość biznesową projekt zyskał dopiero po kliku przyrostach. Oznacza to, że pierwsze efekty nie były satysfakcjonujące, jednak prezentowały postęp prac. Każdy sprint rozpoczynał się spotkaniem zespołu, na którym omówiono pozostałe zadania oraz zaplanowano jakie funkcjonalności będą implementowane w kolejnym przyroście. W efekcie udało się zachować jakość wykonania, modyfikując nieznacznie zakres dostępnych funkcjonalności.

Rozdział 5

Podsumowanie

Rozdział 6

Literatura

Rozdział 7

Dodatki

Literatura

Instytut Informatyki, Wydział Informatyki
Politechnika Poznańska

Skład przy użyciu systemu LaTeX.

BibTeX:

```
@mastersthesis{ key,
    author = "Krzysztof Marciniak \and Piotr Przybysz \and Mikołaj Szychowiak \and
Ryszard Wojtkowiak",
    title = "{Projekt i implementacja gry komputerowej z wykorzystaniem technologii
Microsoft DirectX 11}",
    school = "Poznan University of Technology",
    address = "Pozna{\'n}, Poland",
    year = "2015",
}
```