Mgr inż. Zbigniew ROSIEK[1]

JĘZYK C# I BIBLIOTEKA DIRECTX W PROCESIE WSPOMAGANIA TWORZENIA GIER NA PLATFORMIE MS WINDOWS

Streszczenie

Tworzenie grafiki komputerowej na potrzeby symulacji oraz gier jest zadaniem dość trudnym w porównaniu z innymi dziedzinami programowania. Wymagana jest znaczna wiedza z zakresu matematyki i fizyki oraz dostęp do odpowiednich bibliotek takich jak np. DirectX. W artykule przedstawiono historię i istotne elementy grafiki komputerowej w odniesieniu do platformy MS Windows. Została pokrótce scharakteryzowana biblioteka DirectX. Użycie elementów biblioteki przedstawiono za pomocą prostego przykładu. Biblioteka została zastosowana w środowisku NET. Framework. Przykładową aplikację napisano w języku C#. Podano jego główne cechy i porównano z innymi popularnymi językami (C++, Java) w celu wykazania, że jest on nowoczesnym, wygodnym i prostym w użyciu narzędziem do tworzenia aplikacji, również graficznych.

Abstract

Designing computer graphics for simulations and games is quite a difficult task in comparison with other domains of software design. Not only is considerable knowledge required in the field of mathematics and physics, but also the access to proper libraries, such as DirectX, is vital. The article describes the history and the most important features of computer graphics with reference to the MS Windows platform. The DirectX library is briefly described. The usage of the library is demonstrated on a simple example. The library is used in .NET Framework environment. Example application is written in C# language. The main features of C# are shown and the language is compared with other frequently used programming languages, such as C++ and Java, to show that C# is a modern, simple and convenient tool for creating applications, also graphical ones.

1. WPŁYW ROZWOJU KOMPUTERÓW OSOBISTYCH I ICH SYSTEMÓW OPERACYJNYCH NA ROZWÓJ NARZĘDZI I TECHNOLOGII TWORZENIA GIER KOMPUTEROWYCH

Za początek historii komputerów domowych przyjmuje się rok 1976, kiedy to Steve Jobs i Steve Wozniak stworzyli komputer Apple 1. Sprzedawany był on w cenie 666 dolarów za sztukę. Rozwój komputerów w latach 80 pozwolił każdemu na zakup własnego komputera.

[1] Mgr inż. Zbigniew Rosiek jest wykładowcą w Warszawskiej Wyższej Szkole Informatyki.

Wprowadzenie w 1980 przez firmę IBM komputera PC/XT wykorzystującego 16 bitowy mikroprocesor Intela 8086 (pracującego z częstotliwością 4.77 MHz) zapoczątkowało erę dzisiejszych domowych komputerów PC. Nie minęło wiele czasu, kiedy komputery osobiste zaczęły być wykorzystywane do rozrywki, na przykład jako swoisty „nośnik" gier.

Olbrzymi wpływ na rozwój komputerów i aplikacji miało wprowadzenie w 1986 roku napędu CD-ROM, a później DVD. Dzięki nośnikom CD możliwe stało się gromadzenie i szybkie przenoszenie dużych ilości danych, które w przypadku tradycyjnych dyskietek zajmowałyby kilkadziesiąt lub kilkaset nośników.

Oczywiście wraz z rozwojem komputerów osobistych następował rozwój ich systemów operacyjnych. W najbardziej popularnych komputerach osobistych opartych na procesorach firm Intel i AMD od początku dominowały systemy operacyjne firmy Microsoft.

Istotnymi etapami rozwoju systemów operacyjnych Microsofta było pojawienie systemów DOS, Windows 3.1, Windows NT oraz Windows 95. Zasadnicze zmiany w tych systemach dotyczyły możliwości bezpiecznego dzielenia zasobów komputera pomiędzy wielu użytkowników lub wiele aplikacji. Obsługę z prawdziwego zdarzenia wielu użytkowników (procesów) system Windows zaczął zapewniać w zasadzie w wersjach 95 i NT. Ta tzw. wieloprogramowość z wywłaszczaniem osiągnęła dobry poziom w wersjach 2000 oraz XP. Toteż tworzenie poważniejszych gier komputerowych na platformę systemu MS Windows mogło rozpocząć się w momencie pojawienia się tych systemów.

1.1. Windows oczami programisty (Win 95 -> XP)

System operacyjny Windows jest zbudowany ze współpracujących ze sobą części zarządzających m.in. pamięcią, współpracą z użytkownikami, urządzeniami wejścia/wyjścia. Z punktu widzenia programisty istotne jest, w jaki sposób aplikacja może wykorzystywać zasoby sprzęty udostępniane przez system operacyjny. To, czego potrzebuje programista, to informacje, o tym w jaki sposób aplikacja ma się komunikować z systemem plików, korzystać z pamięci, urządzeniami graficznymi itd.

Windows zbudowany jest warstwowo. Tylko najniższe warstwy operują na poziomie sprzętu. Oznacza to, że nie można się bezpośrednio odwołać do pamięci ekranu, działać na strukturze dysku, sterować ruchem głowicy drukarki. Zamiast tego programista ma do dyspozycji pewien ściśle określony zbiór **funkcji, stałych i typów danych,** za pomocą, których program może się komunikować z systemem. Zbiór tych funkcji i typów danych nazywamy **interfejsem programowania** (Application Programming Interface – **API).**

Dzięki takiej organizacji systemu programista nie musi się martwić np. o model karty graficznej zainstalowanej w komputerze, bowiem z jego punktu widzenia oprogramowanie każdej karty graficznej wygląda tak samo. To system operacyjny za pomocą odpowiedniego sterownika, dostarczonego przez producenta karty bierze na siebie trud realizacji „zachcianek programisty". Interfejs programisty w aktualnych wersjach systemu Windows to **głównie Win32API.** W ramach **Win32API** istnieją grupy interfejsów o określonym przeznaczeniu. Za obsługę grafiki odpowiada **GDI (Grafic Device Interface).** Dostarcza on funkcji i struktur danych, które mogą być wykorzystane do tworzenia efektów graficznych na urządzeniach wyjściowych takich jak monitory czy drukarki. GDI pozwala rysować kształty takie jak linie, krzywe oraz figury zamknięte, pozwala także na rysowanie tekstu.

1.2. Przyczyny powstania biblioteki Direct-X

Gdy podstawowym systemem operacyjnym dla komputerów klasy PC był DOS twórcy gier mieli bezpośredni dostęp do przerwań, kart graficznych, dźwiękowych, urządzeń wejścia/wyjścia. Gdy pojawił się Windows 3.1, bezpośredni dostęp do urządzeń został zastąpiony przez stosunkowo wygodne, ale niezwykle wolne i mało wydajne funkcje (GDI dla funkcji graficznych). Twórcy gier nie stosowali tychże funkcji w swoich produktach. Jednak doskonałe do tej pory środowisko DOS również zaczęło sprawiać problemy. Niezliczone konfiguracje komputerów wynikające z pojawiania się coraz to nowych rodzajów urządzeń we/wy, ich wariantów i modeli sprawiły, że większość czasu zespoły programistów spędzały nad zapewnieniem kompatybilności aplikacji ze sprzętem, niż nad tworzeniem właściwego kodu.

Wdrożony w 1995 roku nowy system operacyjny Windows 95 do obsługi grafiki oferował interfejs GDI nie akceptowany przez twórców gier. Wymagane było nowe podejście do tworzenia skomplikowanej grafiki, stworzenie mechanizmu posiadającego zalety GDI polegające na ustalonym interfejsie dla wszystkich kart graficznych oraz uwolnieniu się od wad GDI wynikających ze skomplikowanej i powolnej wymiany informacji pomiędzy aplikacją a kartą graficzną poprzez kolejne warstwy systemu operacyjnego.

Wybawieniem miało się stać wprowadzenie bibliotek DirectX wraz z ich specyficzną filozofią polegającą na omijania zbędnych elementów systemu operacyjnego i „dogadywaniu" się bezpośrednio z kartą graficzną spełniającą standardy pewnego interfejsu. Oczywiście DirectX nie jest jedynym narzędziem umożliwiającym tworzenie aplikacji obsługujących skomplikowaną grafikę, innym popularnym narzędziem jest chociażby OpenGL.

1.3. GDI a DirectX

Głównym celem projektantów pakietu DirectX było przyciągnięcie programistów zajmujących się oprogramowaniem rozrywkowym do platformy Windows**.**

DirectX jest warstwą oprogramowania oferującą abstrakcyjne podejście do obsługi grafiki, dźwięku, urządzeń wejścia, sieci itp., niezależnie od bieżącej konfiguracji sprzętu komputera. Jest znacznie szybsza od standardowej biblioteki Windows GDI, gdyż do *hardware'u* dostaje się „opłotkami i na skróty", jednak w pewien standardowy sposób. Jej dewiza brzmi:

„Wiem z kim mam współpracować, korzystam z ustalonych zasad, mój kontrahent też je zna, po co nam pośrednictwo przyciężkiego pośrednika Windows'a, zresztą on się na to zgadza i będzie wdzięczny, gdyż i tak ma dość własnych problemów".

1.4. Co można zrobić za pomocą DirectX?

Rozwój biblioteki DirectX jest bardzo dynamiczny. Średnio raz na kwartał pojawia się jej nowa wersja. Przeszła ona długą i nie zawsze usłaną różami drogę. Obecnie najpopularniejsza jest wersja 9.0, w której dostępne są podane poniżej biblioteki:

Direct3D – służąca do rysowania obiektów graficznych na ekranie,
DirectSound – obsługująca dźwięk przestrzenny w aplikacjach,
DirectInput – dostęp do urządzeń wejściowych (np.: klawiatura, mysz, joystick) umożliwia także obsługę urządzeń z Force Feedback (z funkcją sprzężenia zwrotnego),
DirectPlay – odpowiedzialna za usługi sieciowe,
AudioVideoPlayback – komponent dostarczający obsługę plików multimedialnych i filmów.

Oczywiście najważniejszym elementem DirectX jest Direct3D. Dostarcza on interfejs programowania akceleracji grafiki zaimplementowanych prawie we wszystkich kartach graficznych, które posiadają swoje sterowniki Direct3D. Oznacza to, iż jedna aplikacja może być uruchamiana na różnych konfiguracjach sprzętowych. W praktyce nie jest to jednak takie proste. Często akcelerator 3D na karcie graficznej wspiera tylko podzbiór cech dostępnych w DirectX, który jak to już zostało wcześniej powiedziane, rozwija się bardzo dynamicznie. Z tego też powodu dość powszechne jest „nie chodzenie" nowych gier, wykorzystujących nowe możliwości DirectX, na komputerach ze starszymi kartami graficznymi.

Należy podkreślić, że biblioteki DirectX pod nazwą Software Development Kit (SDK) są udostępniane przez firmę Microsoft wszystkim legalnym użytkownikom systemu operacyjnego MS Windows bezpłatnie, co biorąc pod uwagę zaawansowa-

nie biblioteki, jej ciągły rozwój, a więc niewątpliwie związane z tym wysokie koszty, zmienia powszechnie funkcjonujący obraz firmy Microsoft jako monopolisty, zmuszającego użytkowników swoich produktów do wysokich opłat za coraz to nowe wersje swoich produktów.

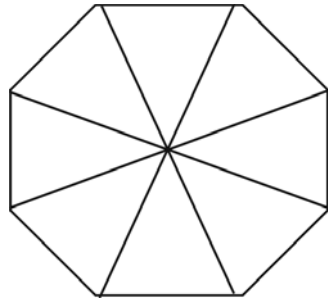
2. PODSTAWY GRAFIKI KOMPUTEROWEJ

Tworzenie zawansowanej grafiki komputerowej nie jest zadaniem prostym. Wymagana jest dość szeroka wiedza z zakresu matematyki i fizyki oraz uzdolnienia plastyczne. Bez tych umiejętności korzystanie z biblioteki DirectX nie jest możliwe. Należy też ostrzec potencjalnych jej użytkowników, że ze względu na jej olbrzymie zasoby obiektów i realizowanych funkcji oraz ubogą literaturę aktualnych wersji w języku polskim osiągnięcie pierwszych efektów nie przychodzi szybko i po drodze często zdarzają się sytuacje „krok do przodu, dwa kroki do tyłu". W dalszej części tego rozdziału omówione zostaną najbardziej istotne elementy grafiki komputerowej**.**

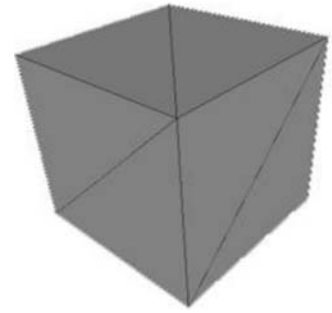
2.1. Obiekty 3D

Wszystkie obiekty w dzisiejszej grafice 3D tworzonej w czasie rzeczywistym składają się z wierzchołków. Wierzchołek jest punktem w przestrzeni, opisanym najmniej trzema zmiennymi x,y,z. Definiując trzy wierzchołki i łącząc je między sobą otrzymamy trójkątną powierzchnię, która jest budulcem używanym do tworzenia powierzchni trójwymiarowych. Takie elementarne trójkąty zwane są w DirectX werteksami (rys. 1). Z odpowiednio dużej ilości małych werteksów można zbudować (narysować) dowolną figurę lub bryłę geometryczną (rys. 2, 3).

Rys. 1. Werteks



Rys. 2. Ośmiokąt zbudowany w oparciu o 8 trójkątów równoramiennych



Rys. 3. Sześcian składający się z 12 trójkątów

2.1.1. Obliczenia i transformacje w trzech wymiarach – co trzeba wiedzieć?

Aby zbudować złożoną bryłę geometryczną konieczna jest dobra znajomość wymienionych poniżej zagadnień z zakresu matematyki:

- Dwu i trzywymiarowe układy współrzędnych (kartezjański, biegunowy, lewoskrętny, prawoskrętny itd.);
- Podstawy trygonometrii;
- Wektory i elementy rachunku wektorowego;
- Macierze i algebra liniowa;
- Liczby zespolone;

Biorąc pod uwagę dynamikę gier, a więc konieczność „rysowania" szeregu zjawisk fizycznych przydatna jest znajomość rachunku całkowego i różniczkowego jak również rachunku prawdopodobieństwa.

Oczywiście DirectX dostarcza szeregu funkcji realizujących czy ułatwiających korzystanie z wymienionych elementów matematyki, ale aby z czegoś skorzystać trzeba najpierw wiedzieć do czego to służy.

2.2. Kolory

Powszechnie stosowanym rozwiązaniem w interfejsie GDI jest operowanie kombinacjami kolorów: czerwonego (red), zielonego (green) oraz niebieskiego (blue) – w skrócie RGB. Każdy z kolorów dysponuje 256 odcieniami (0 – brak koloru, 255 – maksymalne nasycenie). W DirekctX precyzja kolorów jest większa o określana jest w przedziale 0,00 do 1,00. Dodatkowo wprowadzony został parametr werteksów określający ich przeźroczystość (0,00 – całkowicie przeźroczysty, 1.00 – całkowicie nieprzeźroczysty).

2.3. Tekstury

Ogólna definicja opisuje ten termin jako właściwość powierzchni odbieraną poprzez zmysł dotyku. W grafice komputerowej tekstura to mapa bitowa użyta do pokrycia obiektów trójwymiarowych i nadania im możliwie realistycznego wyglądu. Chcąc stworzyć ścianę z cegieł nie należy tworzyć oddzielnie sześciennej cegły złożonej z ośmiu wierzchołków i powielania jej aż do powstania muru. Wygenerowanie takiej struktury zajmowałoby zbyt wiele czasu procesora i nie było by efektywne. Prawidłowe rozwiązanie to stworzenie jednego dużego sześcianu symulującego mur i pokrycie go odpowiednią teksturą. Łącząc bitmapę odpowiadających za kolor oraz

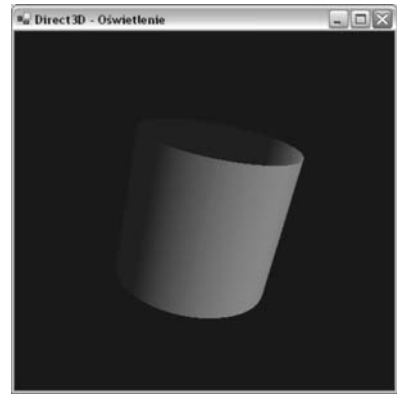
opisującą wypukłości powierzchni można otrzymać bardzo realistyczny obraz. DirectX dysponuje funkcjami umożliwiającymi „obleczenie" bryły w odpowiednią teksturę. Przykład walca powleczonego teksturą zaczerpnięty z tutoriala DirectX przedstawia rys. 4.

2.4. Sekrety oświetlenia

W chwili obecnej nie istnieją powszechnie dostępne urządzenia do tworzenia obrazu trójwymiarowego w 3 wymiarach. Jest on „udawany" przez odpowiednie operowanie światłem na powierzchni 2 wymiarowej.

Wiedza o właściwościach światła jest zagadnieniem bardzo szerokim, niejeden malarz, czy fotografik dałby wiele za poznanie wszystkich jego tajemnic. Dlaczego cień jest taki a nie inny, dlaczego w pewnym miejscu jest „czarniejszy", a w innym „bledszy", dlaczego czasem powierzchnia obiektu jest w świetle matowa a czasem błyszczy, kiedy tak jest? Tą wiedzę można zdobyć na wykładach z fizyki, z pewnością nie ma jej w dokumentacji programowej DirectX, który jednak daje wiele funkcji umożliwiających tworzenie efektów świetlnych. W DirectX wyróżniane są dwa rodzaje oświetlenia: rozproszone i bezpośrednie. Światło rozproszone oświetla obiekty równomiernie, natomiast światło bezpośrednie umożliwia tworzenie przeróżnych cieni. Jednym słowem światło bezpośrednie posiada zawsze określone za pomocą współrzędnych x, y, z źródło. Przykład demonstrujący zastosowanie odpowiednich funkcji DirectX służących do tworzenia efektu światła bezpośredniego pokazuje rys. 5 (przykład zaczerpnięto z tutoriala DirectX). Należy tu podkreślić, że zarówno zagadnienie teksturowania oraz oświetlenia komplikuje się, gdy nasz obiekt ma się poruszać, co w istocie jest bardzo istotnym elementem wszelkich animacji.

Rys. 4



Rys. 5

2.5. Prymitywy

Rysowanie złożonych brył z trójkątów elementarnych związane jest z koniecznością dysponowania dostatecznie dużą pamięcią na zapamiętanie ich współrzędnych, zapamiętywać musimy też informacje o oświetleniu i kolorze poszczególnych trójkątów. Jeżeli do tego dochodzi efekt ruchu, to zmiany współrzędnych poszczególnych wierzchołków trójkątów są dość częste, a więc występuje znaczny ruch danych. Aby jakoś przeciwdziałać tym niekorzystnym zjawiskom zastosowano w DirectX pewne mechanizmy służące do zminimalizowania ilości zapamiętywanych współrzędnych wierzchołków trójkątów, są to tzw. prymitywy, należą do nich:

- lista punktów;
- lista linii;
- wstęga linii;
- lista trójkątów;
- wstęga trójkątów;
- wachlarz trójkątów.

Szczególnie ciekawe i efektywne są wstęga trójkątów i wachlarz trójkątów. W obu tych prymitywach wykorzystywane jest zjawisko pokrywania się niektórych punktów w sąsiadujących ze sobą w danej bryle trójkątów. We wstędze (trójkąt obok trójkąta) rysując nowy trójkąt korzystamy zawsze z dwu wierzchołków poprzedniego. Dzięki temu do narysowania z pomocą wstęgi trójkątów n trójkątów potrzebnych jest tylko n+2 wierzchołków. Oczywiście nie zawsze można użyć wstęgi. Do kreślenia zaokrąglonych kształtów (czasza) przydatny jest wachlarz. Tutaj wykorzystywany jest efekt wspólnego wierzchołka trójkątów tworzących wielokąt z trójkątów równoramiennych. Wierzchołek ten pokrywa się ze środkiem okręgu opisanego na utworzonym z trójkątów wielokącie.

2.6. Renderowanie kadru

Zjawisko animacji jest powszechnie znane i polega na wyświetlaniu z określoną częstotliwością (20-30 klatek /sek.) kolejnych klatek filmu zawierających odpowiednio zmieniony obraz obiektu. A więc tworząc grafikę komputerową należy w aplikacji utworzyć bufor na zapamiętanie obrazu klatki w określonym formacie. Proces „wypełniania" bufora danymi o obrazie nazywamy renderowaniem. W DirectX stosowany jest system 2 buforów, podczas gdy z jednego obraz jest wyświetlany, drugi „napełniany" jest danymi. Mechanizm ten umożliwia tworzenie animacji bardzo płynnej, pozbawionej efektu migotania.

3. NAJPOPULARNIEJSZE ŚRODOWISKA PROGRAMISTYCZNE DO TWORZENIA GIER KOMPUTEROWYCH

Języki programowania, podobnie jak sprzęt komputerowy czy systemy operacyjne, ulegają ciągłej ewolucji. Powstają nowe języki, ale inne, do niedawna uważane za coś absolutnie doskonałego tracą świeżość, a potem popadają w zapomnienie, tak było z Algolem, Fortranem, Cobolem. Bardzo popularny do niedawna Pascal i jego wersja obiektowa ObjectPascal stosowana w pakiecie Delphi firmy Borland tracą wielu dotychczasowych zwolenników na rzecz języków C++, Java czy coraz popularniejszego środowiska .NET.

Obecnie nadal najpopularniejszym językiem, w którym pisane są gry komputerowe jest C++, ale widać coraz bardziej, że poprawiająca się stale wydajność komputerów osobistych sprawia, że coraz częściej programiści gier decydują się na wygodne i stale rozwijane środowisko .NET, a w szczególności specjalnie dla niego opracowany język C#.

3.1. Porównanie C# z innymi językami

C#, Java oraz C++ mają wspólne korzenie, stąd C# ma z Javą i C++ zdecydowanie więcej elementów wspólnych niż z innymi językami. C# zajmuje wśród współczesnych języków obiektowych szczególne miejsce. Łączy, bowiem w sobie bardzo wysoką wydajność (co stawia go obok C++) z niezwykle eleganckim i przyjaznym modelem obiektowym.

3.1.1. C# a C++

C++ jest próbą zbudowania języka obiektowego na bazie składni języka C. C# jest od początku do końca zaprojektowany jako język obiektowy. Kilka ważniejszych różnic między C# a C++:

- System typów jest w C# o wiele mocniejszy niż w C++. Typy są śledzone dynamicznie, to znaczy, że nawet podczas wykonywania programu nie ma możliwości konwersji pomiędzy wartościami niezgodnych typów;
- W C# nie ma plików nagłówkowych, a kolejność klas w projekcie nie ma znaczenia;
- W C# testowanie warunków wymaga wyrażenia bool. W C++ można zamiennie wykorzystywać wyrażenie typu int, np. if (1);
- W C# nie ma jawnej destrukcji obiektów. Niszczeniem obiektów zajmuje się odśmiecacz.

- W C# nie ma szablonów. Jednorodny model obiektowy pozwala pisać kod bardziej elegancki niż za pomocą szablonów C++.
- Przekazywanie błędów w C# odbywa się za pomocą wyjątków. Ta reguła jest konsekwentnie stosowana.

Niewątpliwie język C++ jest w chwili obecnej językiem bardzo popularnym, obficie opisanym w literaturze, umożliwiającym budowanie w sposób w miarę wygodny efektywnego kodu. Biorąc jednak pod uwagę dynamiczny wzrost dostępnych pamięci operacyjnych oraz wydajności procesorów relacja nakład pracy – efekt w wielu zastosowaniach stawać się będzie nieuchronnie coraz bardziej korzystna dla języka C#, który przecież powstał m.in. jako reakcja na słabości C++. Biorąc dodatkowo pod uwagę, że C# jest udanym produktem Microsoft, wygodnym i przyjaznym w programowaniu, mającym dzięki filozofii .NET Framework dostęp do wielu bibliotek. Dostępna jest w nim również biblioteka DirectX, a więc jest rzeczą oczywistą, że warto pisać w C# aplikację wymagające zaawansowanej grafiki. Istotnym mankamentem jest w chwili obecnej brak ciekawej literatury w języku polskim dotyczącej tworzenia aplikacji w C# z wykorzystaniem DirectX. Trzeba posiłkować się literaturą dotyczącą C++ lub nawet VisualBasica.

3.1.2. C# a Java

Istotną zaletą języka Java jest jego znaczne rozpowszechnienie, dzięki filozofii maszyny wirtualnej na wielu platformach sprzętowych i systemowych. Jest on powszechnie wykorzystywany w dużych instalacjach sieciowych jak również w telefonach komórkowych lub różnego rodzaju urządzeniach umożliwiających zdalny dostęp poprzez Internet. C# jest językiem młodszym (prace nad nim Microsoft podjął po rezygnacji z udziału w rozwoju Javy), jego środowisko, aczkolwiek może dzięki filozofii kompilatora JIT być osadzone na różnych platformach systemowych, zdecydowanie najlepiej rozpowszechniło się na platformie MS Windows, czemu nie należy się dziwić. Język Java, chociaż bardzo wygodny w wielu zastosowaniach, nie jest i raczej nie będzie stosowany do tworzenia zaawansowanej grafiki. W tym wypadku to C# ma nad nim zdecydowaną przewagę dzięki możliwości korzystania z bogatych bibliotek DirectX. Również środowisko programisty jest w przypadku wersji okienkowej C# zdecydowanie bardziej przyjazne niż ma to miejsce w Javie. Jest ono rozwinięciem, wzbogaconym o szereg nowych rozwiązań, wygodnego interfejsu programisty stworzonego w pakiecie Delphi.

4. JAK TO SIĘ ROBI W C# – PROSTY PRZYKŁAD APLIKACJI WYKORZYSTUJĄCEJ DIRECTX

Jak to zostało wcześniej napisane, zrozumienie filozofii tworzenia grafiki za pomocą DirectX wymaga pewnej wiedzy matematycznej i pewnej liczby wykonanych samodzielnie aplikacji (nic tak nie uczy narzędzia jak samodzielne próby jego użycia). Należy sobie zdawać sprawę z tego, że DirectX nie jest narzędziem do tworzenia gier, lecz biblioteką umożliwiającą dość sprawne tworzenie interfejsu graficznego oraz audio. Na to, aby powstała gra komputerowa nie wystarczy wykonanie samej grafiki, na grę składa się wiele innych elementów, których nie da się oczywiście omówić w krótkim artykule.

Nasza aplikacja będzie wyświetlała obracający się sześcian pokryty teksturą. Oczywiście zanim przystąpimy do pracy należy na komputerze zainstalować dostępny na stronie Microsoftu DirectX 9 Update SDK, gdzie znajdziemy wyczerpującą dokumentację i przydatne narzędzia.

Korzystanie z klas DirectX w .NET jest możliwe po uprzednim załączeniu do stworzonego projektu odpowiednich bibliotek (nie są to oczywiście wszystkie biblioteki DirectX, a tylko potrzebne w naszej aplikacji):

```
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;
using Colour = System.Drawing.Color;
```

Direct3D.Device jest obiektem, od którego wywodzą się wszystkie inne obiekty graficzne. Jest on odpowiedzialny między innymi za ich transformację, oświetlenie czy renderowanie.

Pierwszym krokiem w budowaniu każdej aplikacji korzystającej z Direct3D jest stworzenie obiektu klasy *Device*. W naszej aplikacji będzie się nazywał *dev*:

```
private Device dev = null;
```

Kod metody powołującej do życia obiekt klasy Direct3D.Device przedstawiony jest poniżej:

```
private void tworzDX()
{
    PresentParameters initParams = new PresentParameters();
    initParams.Windowed = true;
    initParams.SwapEffect = SwapEffect.Copy;
```

```
    dev = new Device(0,DeviceType.Hardware, this,
            CreateFlags.HardwareVertexProcessing,initParams);
}
```

Windowed – odpowiada za wybór trybu wyświetlania: pełnoekranowy *(false)*, w oknie (*true*),

SwapEffect – określa mechanizm postępowania z buforami obrazu w trakcie wyświetlania kolejnych klatek animacji.

Przygotowanie do wyświetlenia każdej klatki animacji nazywamy renderowaniem klatki. Poniżej podany został szkielet metody służącej do przygotowania i wyświetlenia pojedynczej klatki. Jako pierwsze możemy ustalić tło, na którym obracać się będzie nasz sześcian. Do tego celu użyjemy metody *Clear*. Metody *BeginScene* oraz *EndScene* mają za zadanie odpowiednio rozpoczęcie i zakończenie tworzenia obrazu kadru w buforze. Metoda *Present* odpowiada za wyświetlenie w oknie lub na pełnym ekranie zapisanego wcześniej w buforze obrazu.

```
private void renderuj()
{
    dev.Clear(ClearFlags.Target,Colour. LightSteelBlue,0,0);
    dev.BeginScene();
     dev.EndScene();
    dev.Present();
}
```

Nadszedł czas na utworzenie struktury umożliwiającej przechowywanie współrzędnych wierzchołków sześcianu. W tym celu użyjemy jednej z kilku struktur do tego przeznaczonych – *PositionTextured*. W jej skład wchodzą współrzędne x,y-,z wierzchołka, jego kolor oraz tzw. para współrzędnych tekstury u,v. Zrealizujemy to za pomocą metody tworzVerteksy. Jak łatwo policzyć sześcian składa się z 12 trójkątów, a te mają po 3 wierzchołki – razem 36 punktów. Oczywiście można byłoby zastosować wstęgę, ale w naszym przykładzie tego nie zrobimy.

```
private void tworzVerteksy ()
{
CustomVertex.PositionTextured[] szescian = new
CustomVertex.PositionTextured[36];
```

```
// A
szescian [0] = new CustomVertex.PositionTextured(-1.0f, 1.0f, 1.0f, 0.0f, 0.0f);
// B
v[1] = new CustomVertex.PositionTextured(-1.0f, -1.0f, 1.0f, 0.0f, 1.0f);
// C
szescian [2] = new CustomVertex.PositionTextured(1.0f, 1.0f, 1.0f, 1.0f, 0.0f);
...
...
dev.VertexFormat = CustomVertex.PositionTextured.Format;
}
```

Dwa ostatnie parametry każdego wierzchołka określają sposób nałożenia tekstury na powierzchnię każdego z werteksów. Ostatni wiersz metody służy do ustalenia, w jaki sposób będą renderowane poszczególne wierzchołki. Mając zdefiniowaną strukturę naszego sześcianu czas pomyśleć o utworzeniu bufora przechowującego wierzchołki o nazwie *BuforVerteksow*. Dodajmy, zatem pole do naszej klasy:

```
VertexBuffer BuforVerteksow = null;
```

Teraz w metodzie *tworzVerteksy* możemy dopisać:

```
BuforVerteksow = new VertexBuffer(szescian [0].GetType(), szescian.Length,dev,
Usage.WriteOnly,CustomVertex.PositionColored.Format,Pool.Default);
```

Pierwszy parametr określa typ wierzchołków przechowywanych w *VertexBuffer*, w następnych określana jest ich liczba oraz obiekt *Direct3D.Device*, z jakim bufor ma być skojarzony. Kolejny argument definiuje sposób użycia bufora jako „tylko do zapisu" (*Usage.WriteOnly*), co czasem zwiększa efektywność przetwarzania. Należy jeszcze określić format wierzchołków a także obszar pamięci, w którym będzie przechowywany bufor – *Pool.Default* wskazuje na VRAM lub obszar pamięci dostępny poprzez port AGP.

Aby DirectX mógł przetworzyć zawartość bufora, trzeba go powiązać ze strumieniem danych obiektu *Direct3D.Device*. Do tego przeznaczona jest metoda *SetStreamSource*. W metodzie *renderuj* należy więc dopisać po wywołaniu funkcji *Clear*:

```
dev.SetStreamSource(0, BuforVerteksow, 0);
```

Parametrami są tutaj: numer strumienia (0 jest w tym przypadku pierwszym i jedynym strumieniem), adres bufora wierzchołków oraz przesunięcie (w bajtach) w tym buforze określające miejsce, od którego DirectX ma rozpocząć rysowanie. Nasza metoda *renderuj* mająca za zadanie przygotowanie pojedynczej klatki animacji pomału się zapełnia. Teraz pozostał nam jeszcze pokryć sześcian teksturą. Oczywiście musimy dodać do projektu bitmapę z jakąś teksturą. W tym celu w menu *Project* za pomocą funkcji *Add Existing Item* wskazujemy odpowiedni plik (w naszym przypadku jest to „tekstura1.bmp"), a następnie we właściwościach bitmapy zmieniamy *Build Action* na *Embedded Resource*. Teraz deklarujemy w naszej klasie kolejne pole do obsłużenia tekstur:

Texture tekstura = null;
a następnie za pomocą klasy *TextureLoader* tworzymy teksturę:

tekstura = TextureLoader.FromFile(dev,"world.bmp");

Teraz należy powiedzieć DirectX, aby używał tej bitmapy w trakcie renderowania prymitywów. Skorzystamy w tym celu z funkcji SetTexture (pierwszy argument tej metody jest poziomem używanej tekstury, poziomów może być maksymalnie 8):

dev.SetTexture(0,tekstura);

W tej chwili dysponujemy informacjami dotyczącymi wyglądu i położenia sześcianu, czas określić sposób wyświetlania naszej bryły. W pierwszym kroku trzeba określić perspektywę, z jakiej będziemy oglądać obiekt oraz położenie w przestrzeni kamery (punktu obserwacji). Perspektywę określamy poprzez metodę *Matrix.PerspectiveFovLH* (LH oznacza lewoskrętny układ współrzędnych, obowiązujący w DirectX):

dev.Transform.Projection = Matrix.PerspectiveFovLH((float)Math.PI /4,
this.Width / this.Height, 0.0f,50.0f);

kolejne argumenty to:
- Pole widzenia wyrażone w radianach;
- Format obrazu określony jako stosunek szerokości do długości;
- Odległość pierwszego i ostatniego planu.

Określimy teraz położenie kamery za pomocą metody *Matrix.LookAtLH* przekazując jako parametry trzy wektory określające:

- położenie kamery w przestrzeni,
- lokalizację miejsca, na które skierowany jest obiektyw,
- kierunek „do góry" w przestrzeni:

```
dev.Transform.View = Matrix.LookAtLH(new Vector3(-2,2,3.0f), new
Vector3(0,0,0),new Vector3(0,1,0));
```

Teraz nastąpił moment, w którym należy określić oświetlenia sześcianu:

```
dev.RenderState.Lighting = false;
```

Nasza bryła ma już określony kształt, jest „oklejona" teksturą, dysponujemy danymi o punkcie, z którego jest ona obserwowana. Czas na ostatni element naszej aplikacji – wprawienie bryły w ruch. Oczywiście można ją poruszać na wiele sposobów, jednym z nich jest, w tym celu skorzystamy z metody *Matrix.RotationYawPitchRoll*.

```
rAngle += 0.07f;
dev.Transform.World = Matrix.RotationYawPitchRoll(rAngle
/ (float)Math.PI, rAngle /(float)Math.PI * 2.0f, rAngle / (float)Math.PI);
```

Teraz należy umieścić między metodami *BeginScene* a *EndScene* podane poniżej wywołanie metody *DrawPrimitives*:

```
dev.DrawPrimitives(PrimitiveType.TriangleList, 0, 12);
```

Nasza aplikacja jest w zasadzie gotowa. Należy pamiętać, że podczas zmiany rozmiaru okna aplikacji automatycznie resetowany jest obiekt *Direct3D.Device*, czego wynikiem jest wyczyszczenie buforów obrazu. Jeżeli więc chcemy, aby sześcian zachował swoje położenie podczas skalowania okienka trzeba obsłużyć wyjątek na taką okoliczność. Dopiszmy więc do metody *tworzDX* po utworzeniu obiektu *dev*:

```
dev.DeviceReset+=new EventHandler(OnDeviceReset);
```

Funkcja OnDeviceReset obsługująca zdarzenie wygląda natomiast tak:

```
private void OnDeviceReset(object sender, EventArgs e)
{
    createVertices();
     render();
}
```

Widać na przedstawionym powyżej w sposób skrótowy prostym przykładzie, że wykonanie bardziej skomplikowanych aplikacji bez opanowania wymienionego wcześniej zakresu wiedzy z matematyki i fizyki, a następnie samodzielnym wykonaniu prostych aplikacji, jest raczej niemożliwe. Programowanie grafiki wydaje się dziedziną programowania zdecydowanie bardziej skomplikowaną niż np. tworzenie baz danych, czy prostych stron internetowych i to nie dlatego, że DirectX jest skomplikowany, lecz dlatego, że zjawiska realizowane przez funkcje DirectX nie są proste.

5. ZAKOŃCZENIE

W artykule starałem się w sposób niezbyt skomplikowany przedstawić najistotniejsze zagadnienia związane z tworzeniem grafiki komputerowej na potrzeby gier komputerowych. Grafikę tę tworzy się za pomocą określonych narzędzi, specyficznych dla określonych platform sprzętowych i systemowych. Niewątpliwie najpopularniejszym obecnie systemem operacyjnym dla komputerów domowych, a więc tych, na których korzysta się z gier, jest system Windows firmy Microsoft. O firmie i jej systemie operacyjnym Windows można mówić dobrze lub źle, ale codzienna rzeczywistość w sposób jednoznaczny udowadnia, że przysłowiowy Kowalski ma w domu komputer z systemem operacyjnym tej firmy. Toteż nie należy się dziwić, że większość gier komputerowych pisana jest pod kątem uruchamiania ich pod kontrolą systemu Windows. Microsoft zdając sobie sprawę ze znaczenia rynku „graczy" dokłada wiele starań by twórcom gier udostępniać coraz to nowsze wersje biblioteki DirectX, dającej możliwość szybkiego tworzenia grafiki na potrzeby gier. W artykule zasygnalizowałem coraz większą popularność środowiska .NET, a w szczególności języka C#, który zaczyna wypierać bardzo do niedawna popularny język C++. Biblioteka DirectX doskonale współpracuje z tym językiem i coraz chętniej jest ten tandem wykorzystywany przez programistów w tworzeniu grafiki i animacji w grach komputerowych. Pomimo ograniczania się do informacji nieraz najbardziej podstawowych i tak nie udało mi się napisać o szeregu istotnych spraw związanych z tematem artykułu. Po prostu tworzenie gier komputerowych jest procesem złożonym,

w którym niezbędna jest duża wiedza teoretyczna z różnych dziedzin matematyki i fizyki, a także znaczne doświadczenie. Jeżeli więc czytelnik tego artykułu będzie po jego przeczytaniu czuł niedosyt, że nie znalazł w nim opisu procesu tworzenia gry komputerowej z wykorzystaniem DirectX to zapraszam do przestudiowania literatury podanej poniżej.

Literatura

[1] Zasoby MSDN, DirectX SDK.
[2] Bruno Migiel, Teixeira de Sousa, Programowanie gier komputerowych, wyd. Helion, Gliwice 2003.
[3] Kelly Dempski, DirectX Rendering w czasie rzeczywistym, wyd. Helion, Gliwice 2003.
[4] Mason Mc Cuskey, Programowanie gier w DirectX, wyd. MIKOM, Warszaw 2003.
[5] Wayne S. Freeze, Visual Basic i DirectX Programowanie gier w Windows, wyd. Helion, Gliwice 2004.