Na początek mała uwaga. Większość z Was, czytając te artykuły, zauważy zapewne, że jest to w dużej mierze tłumaczenie dokumentacji DirectX-a umieszczonej w SDK. Nie ukrywam, że tak jest w istocie. Ponieważ wielu z Was ma problemy z językiem angielskim (wstydźcie się :-P), więc mam nadzieję, że te artykuły Wam trochę pomogą. Nie będzie to tłumaczenie dosłowne i od czasu do czasu będę wtrącał coś od siebie, ale postaram się aby w całości i w dużej mierze przypominało to to, co możemy znaleźć w oryginalnej dokumentacji.

Układ współrzędnych (ang. *coordinate system*)

W naszych bojach często będziemy się spotykać z pojęciem układu współrzędnych. Czym on jest sam w sobie, to mam nadzieję, że nie trzeba nikomu tłumaczyć, bo tego chyba nauczyli każdego w szkole podstawowej. Istnieją różne układy współrzednych, które są stosowane w zależności od tego, jakie są potrzeby i jakie dane są dostępne. W naszym świecie 3D będziemy mieli do czynienia z tzw. kartezjańskim układem współrzędnych (ang. *cartesian coordinate system*). Jest on nam dobrze znany (mam nadzieję) ze szkoły, ale dla większego porządku wprowadźmy sobie jednak taką uściśloną definicję encyklopedyczną.
Kartezjański układ współrzednych to taki, który na płaszczyźnie tworzą dwie, a w przestrzeni trzy wzajemnie prostopadłe proste - noszą one nazwę osi układu, a miejsce ich przecięcia jest jego początkiem (środkiem). Położenie punktu określa się przez podanie odległości od poczatku układu do punktów otrzymanych przez rzutowanie prostopadłe danego punktu na poszczególne osie. Brzmi to trochę mętnie, ale chyba rozumiemy wszyscy na czym to polega. Aplikacje generujące grafikę 3D używają zasadniczo dwóch rodzajów układów współrzędnych kartezjańskich: prawo- i lewoskrętnego (ang. *right-handed*, *left-handed*). W obu przypadkach oś *X* (ang. *x-axis*) jest położona poziomo i jest skierowana w prawo. Oś *Y* (ang. *y-axis*) jest osią pionową i jest skierowana w górę. Różnica istnieje w kierunku osi *Z* (ang. *z-axis*). Oś ta jest zawsze położona prostopadle do powierzchni ekranu, jeśli dla uproszczenia założymy, że jest on całkowicie płaski a jej kierunek możemy określić na podstawie ułożenia lewej lub prawej ręki. Dla lewoskrętnego układu współrzędnych, jeśli wszystkie palce lewej ręki skierujemy w dodatnią stronę osi *X* i zagniemy je, tak aby pokazywały w kierunku dodatniej osi *Y*, to odchylony kciuk pokaże nam dodatni kierunek osi *Z* (w tym przypadku "w głąb" monitora). Dla prawoskrętnego układu robimy to samo, tylko z wykorzystaniem prawej ręki. Wszystkie nasze przykłady będą używać lewoskrętnego układu współrzędnych. Jeśli kogoś interesuje (lub będzie musiał wykorzystać) układ prawoskretny, to będzie musiał dokonać kilku przeróbek, ale o tym może poczytać dokładniej w dokumentacji zamieszczonej w SDK.

Cieniowanie (ang. *shading*)

Nasze trójwymiarowe sceny będziemy tworzyć w oparciu o wierzchołki umieszczone w przestrzeni i zbudowane na podstawie trojkątów. Nie ma takiej figury, której nie dałoby się zbudować z trójkątów. Wszystkie programy do obróbki grafiki 3D takie jak: 3D Studio Max, LightWave itp. używają trójkąta jako podstawowego budulca. Takim właśnie najmniejszym klockiem trójkąt będzie i dla nas, i właśnie całymi zbiorami trójkątów będziemy operować w naszych aplikacjach. Trójkąty można łączyć w większe zbiory, które będą zbudowane w specyficzny sposób (tzw. pasy lub wachlarze), na to też przyjdzie czas. Z trójkątów będziemy tworzyć wszelkie powierzchnie o praktycznie dowolnym kształcie. Większość kart z akceleracją jest zoptymalizowana właśnie do renderingu trójkątów. Jednak zastosowanie samych trójkątów nie spowoduje jeszcze, że wszystko będzie wyglądać pięknie i gładko. Wymyślono więc coś takiego jak techniki cieniowania obiektów. Czym jest cieniowanie? Jest to, mówiąc w uproszczeniu, sposób nakładania kolorów na nasze trójkąty. Jeśli nasz trójkąt pomalujemy w odpowiedni sposób, wykorzystując cieniowanie, możemy uzyskać na jego powierzchni np. odbicie światła. Dzięki zastosowaniu takich technik, możemy oglądać piękne i gładkie kule, które tak naprawdę składają się tylko z płaskich trójkątów. Jeśli obiekt 3D jest dobrze zaprojektowany, to nawet zastosowanie małej liczby trójkątów i odpowiedniej techniki cieniowania bardzo często pozwala osiagnąć zadowalające efekty. Musimy przecież pamiętać, że zwiększenie liczby wierzchołków na scenie (no a co za tym idzie oczywiście liczby trójkątów także), powoduje wzrost ilości obliczeń, co, w końcowym rozrachunku, może "zadusić" naszą kartę grafiki. Cieniowanie obiektów, to spora matematyka, ale nas powinno interesować tylko to, że DirectX Graphics daje nam do ręki gotowe funkcje, które będą cieniować obiekty za nas, my tylko będziemy decydować o rodzaju cieniowania. Mamy do dyspozycji dwa rodzaje cieniowania - płaskie (ang. *flat*) i tzw. Gourauda. Cieniowanie płaskie polega, mówiąc w skrócie na tym, że każdy trójkąt jest malowany w takim kolorze, jaki posiada pierwszy z jego wierzchołków. Jeśli dla przykładu wyrenderujemy sobie kulę za pomocą płaskiego cieniowania, to będziemy mogli zobaczyć poszczególne trójkąty z jakich jest zbudowana. Druga metoda - Gourauda (nazwana tak pewnie na cześć wynalazcy) powoduje, że kolor trójkąta jest aproksymowany, czyli dobierany na podstawie oświetlenia i wektora normalnego dla każdego wierzchołka oddzielnie i wnętrze trójkąta nie jest już jednolite. Dlatego też jeśli wyrenderujemy kulkę w ten sposób, to otrzymamy zupełnie gładką powierzchnię bez załamań. Ważną rzeczą, o której należy jeszcze wspomnieć, są wyżej wymienione tak zwane wektory normalne. Są to wektory skierowane prostopadle do powierzchni, na ktorej jest położony dany trojkat. Każdy trójkąt ma swoją normalną (wektor normalny). Wektory te będą nam potrzebne przy malowaniu oświetlonych obiektów, ponieważ bez nich będzie niemożliwe osiągnięcie określonych efektów. Oczywiście na wszystko przyjdzie czas i na pewno się wyjaśni dokładniej na przykładach. Zawsze też polecam przeczytanie dokumentacji zamieszczonej w SDK.

Macierze i transformacje (ang. *matrices and transformations*)

Wiadomo powszechnie, że grafika 3D wymaga sporej ilości obliczeń. Ci, którzy pisali dawno temu animacje 3D w Pascalu czy C, korzystając ze starego dobrego BGI, wiedzą, ile trzeba było rozwiązać układów równań, aby móc osiagnąć w końcu upragniony cel. Ludzie projektujący biblioteki OpenGL-a i DirectX-a wpadli jednak na pomysł, że układy równań bardzo

dobrze rozwiązuje się metodą macierzową. A ponieważ operacje na macierzach, za pomocą programow komputerowych, to sama przyjemność i wygoda, zastosowano ten sposób i tutaj. Do pracy z grafiką 3D będzie nam niestety potrzebna podstawowa wiedza o działaniach na macierzach, takich jak ich wypełnianie, dodawanie i mnożenie. Wszystko, co będziemy robić, wcześniej czy później sprowadzi się do operacji na macierzach (wypełniania ich odpowiednimi wartościami). DirectX oczywiście oferuje wiele ułatwień w posługiwaniu się macierzami, począwszy od struktur je reprezentujących, kończąc na przeciążonych operatorach, które za jednym zamachem pomnożą czy dodadzą nam macierze. Nie będziemy więc musieli pisać takich funkcji od nowa. Macierze pomogą w stworzeniu naszego swiata, ustawieniu w nim obiektów i kamer. Jednak najważniejszym celem, do którego użyjemy macierzy, będą transformacje. Mamy trzy podstawowe rodzaje tranformacji, dzięki którym możemy zrobić z obiektem prawie wszystko, co nam się tylko podoba. Są to: przesunięcie, obrót i skalowanie. Oczywiście każda z nich opisuje się odpowiednią macierzą i tylko od wypełnienia ich odpowiednimi wartościami zależy, jak będzie się zachowywał nasz obiekt na scenie. Odpowiednio składając takie przekształcenia, będziemy mogli dokonywać sporo skomplikowanych wyczynów, które w połączeniu ze swiatłem i teksturami dadzą naprawdę rewelacyjny efekt. Należy jednak w tym miejscu zaznaczyć, a o czym będziemy się mogli jeszcze nie raz przekonać, o kolejności działań na macierzach. Mają one to do siebie, że musimy dokładnie wiedzieć, co robimy z naszymi macierzami. Pomimo że mnożenie czy dodawanie zwykłych liczb jest przemienne, to w przypadku macierzy niestety już tak nie jest. I jeśli np. zachce nam się złożyć w jedno przekształcenie macierz reprezentującą obrót i przesunięcie, to inny wynik otrzymamy jeśli pomnożymy przez macierz obrotu macierz przesunięcia niż gdybyśmy zrobili to odwrotnie... tak więc pamiętajmy od teraz - uważamy na kolejność!

Bufory i powierzchnie (ang. *buffers and surfaces*)

Kolejną ważną rzeczą, z jaką musimy się oswoić, jeśli pragniemy animować, jest tak zwane przełączanie stron. Jest to technika, która pomoże nam wprawić w ruch nasze modele w wirtualnym świecie. Od początków animacji rysunkowej filmy te powstawały zawsze w ten sam sposób. Rysowano na kolejnych karteczkach poszczególne fazy ruchu postaci czy przedmiotu. Następnie "przeglądano" te kartki w bardzo szybkim tempie. Uzyskiwano co? No tak, animację czyli, w najwiekszym skrócie, wrażenie ruchu. Taką samą technikę zastosujemy dla naszych potrzeb komputerowych. Będziemy tworzyli serię obrazków, które będą zawierać kolejne fazy ruchu a następnie będziemy je wyświetlać klatka po klatce. Oczywiście, w naszym przypadku, ułatwimy sobie znacznie pracę, ponieważ mamy w ręku jedno z najwspanialszych narzędzi, jakie wymyślił człowiek, i grzechem byłoby go nie wykorzystać. Tylko, że w naszym przypadku zmienią się trochę pojęcia, którymi będziemy operować. Przede wszystkim nasze kartki, na których będziemy rysować. My będziemy je nazywać buforami (bez skojarzeń proszę ;-). Będą to fragmenty pamięci, czy to naszej karty graficznej, czy systemowej, w których będziemy umieszczać dane, które po wyświetleniu na ekranie pozwolą na pełną ekstazę 3D :-). W DirectX, w zależności od rodzaju aplikacji, możemy utworzyć wiele takich buforów, które zostaną potem umieszczone w tzw. łańcuchu przełączania. Jeśli mamy już utworzony taki łancuch, to odpowiednimi funkcjami powodujemy to, że zawartość tych buforów jest po kolei wyrzucana na ekran i otrzymujemy upragnioną animację. Sam ekran też jest buforem bo, jak powszechnie wiadomo, obraz jaki widzimy na monitorze, to też jest fragment pamięci. Dla nas widzialny fragment pamięci będzie nazywany buforem przednim (ang. *front buffer*). Wrzucając do niego odpowiednie dane będziemy mogli zmieniać zawartość ekranu (animować). Jak napisałem wcześniej, buforów mamy kilka. Częstym błędem, który popełniany jest przy próbie animowania np. w Windows, jest rysowanie bezpośrednio w przednim buforze. Powoduje to nieprzyjemne migotanie obrazu i, co za tym idzie, niezbyt miłe wrażenia wizualne. Dlatego też będzie nam potrzebny co najmniej jeden bufor, w którym będziemy rysować, ale nie będzie to bufor przedni. Będziemy nazywać go buforem tylnym (ang. *back buffer*) i rysować w nim kolejne klatki naszej animacji. Jest on niewidoczną częścią naszej pamieci i aby zobaczyć jego zawartość, musimy ją przekopiować do bufora przedniego. Znajdziemy oczywiście odpowiednie funkcje, które się tym zajmą. Kopiowanie takie będzie zsynchronizowane z odświeżaniem obrazu, co spowoduje, że obraz nie będzie nam migał. W Direct3D możemy takich buforów tworzyć wiele, w zależności od możliwości karty grafiki, ale przeważnie dwa wystarczają w zupełności. Dodatkowo będziemy używać też innych rodzajów buforów (koloru, bufora Z itd.), ale to później. Na razie rozchodzi się o podstawy. Każdy bufor, o którym mówiłem wcześniej, będzie przez nas nazywany powierzchnią. Jest to nic innego, jak pewien liniowy fragment pamięci, który będzie zawierał interesujące nas dane. DirectX Graphics daje nam do ręki interfejs, dzięki któremu będziemy mogli zarządzać takimi powierzchniami, mając możliwość nawet bezpośredniego dostępu do pamięci zajmowanej przez taką powierzchnię (w rzeczywistości pamięci karty graficznej lub systemowej). Będzie oczywiście kilka pułapek w takim sposobie dostępu, ale o tym wszystkim zdążymy jeszcze sobie powiedzieć. Powierzchie będą tworzone w różnoraki sposób, dlatego bardzo ważną sprawą będzie ich format, zależny głównie od trybu graficznego, jaki ustawimy dla naszych potrzeb oraz od sposobu pracy aplikacji (pełnoekranowa lub okno). Powierzchni będziemy używać do wyświetlania obrazu jak i do przechowywania tekstur. Powierzchnie przechowujące tekstury też tworzy się w specjalny sposób, ale zaznaczam ponownie: nie wszystko na raz, bo się pogubimy.

Prostokąty (ang. *rectangles*)

Pewnie wielu z Was widziało okrągłe okienka w Windows i zastanawiało się jak ludzie robią takie rzeczy. Jest to bardzo proste i jak kogoś interesuje, to napiszemy tutorial o tym, ale miałem mówić o czymś innym. Dlaczego wspominam o prostokątach? Jest to podstawa w programowaniu w Windows. Każda powierzchnia, jaką stworzymy, będzie prostokątnym obszarem, który będzie posiadał swoje wymiary (lewy górny róg i prawy dolny). Zastosowanie prostokątnych obszarów jest podyktowane szybkością działania Windows. Po prostu takie obszary kopiuje się najprościej i najszybciej. Jednak zastosowanie pewnych sztuczek, takich jak regiony w GDI i klucze kolorów w DirectDraw, powoduje że nie stanowi wielkiej przeszkody zaprogramowanie ciekawszych kształtów niż tylko prostokątne okienka. Jednak musimy pamiętać, że

wszystko, co wyświetlamy na ekranie, to prostokąt (chodzi oczywiście o powierzchnię). Ponieważ jest to figura tak samo potrzebna jak prosta, Windows sam w sobie zawiera już potrzebną strukturę, która bardzo się przyda w naszej nauce. Wygląda tak:

```
typedef struct tagRECT
{
  LONG left;   // This is the top-left corner x-coordinate.
  LONG top;    // The top-left corner y-coordinate.
  LONG right;  // The bottom-right corner x-coordinate.
  LONG bottom; // The bottom-right corner y-coordinate.
} RECT, *PRECT, NEAR *NPRECT, FAR *LPRECT;
```

Co oznaczają poszczególne pola chyba nie trzeba nikomu tłumaczyć. Struktura ta jest częścią GDI (ang. ***G****raphics* ***D****evice* ***I****nterface*) i będzie jedną z kilku rzeczy, z których często będziemy korzystać w DirectX Graphics. Część funkcji pobiera jako swoje argumenty zmienne o typie RECT lub pochodnym. Na tym kończymy opis tego, co jest niezbędne na poczatek. Następny tutorial potraktuje już bezpośrednio o aplikacji, będzie przykładowy kod, opis obiektów oraz sposób ich wykorzystania.