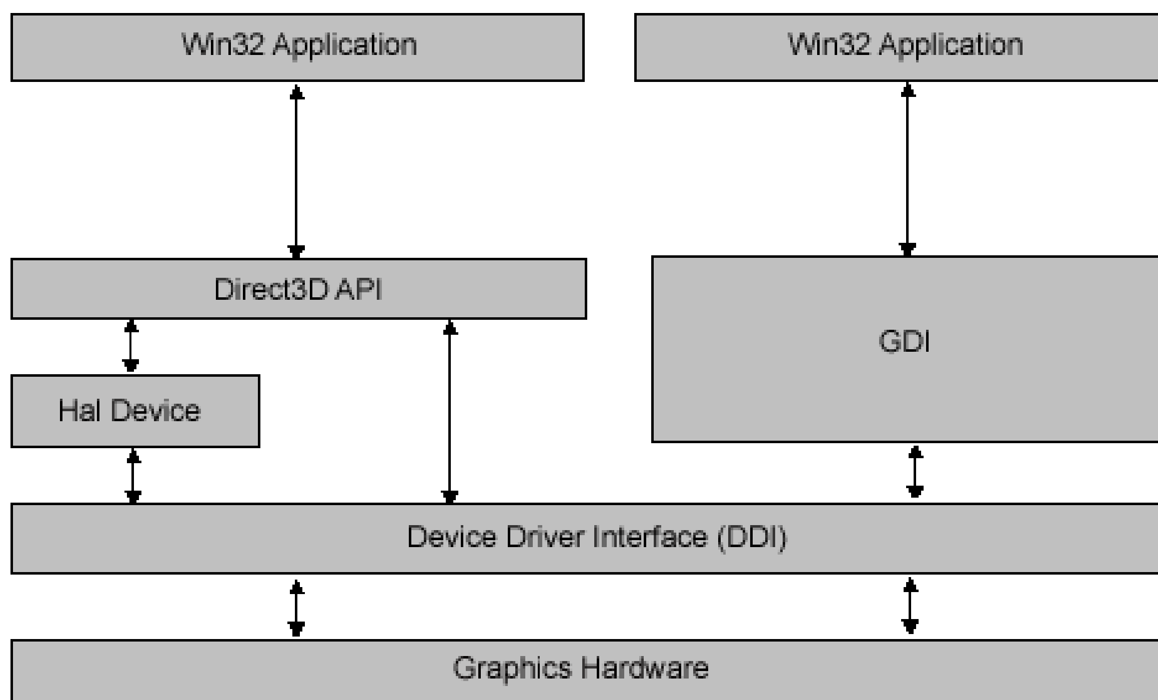
2. Podstawy *Direct3D*

Zajmiemy się teraz podstawami biblioteki graficznej *Direct3D*. Przedstawimy jej budowę oraz najważniejsze pojęcia w niej występujące.

2.1 *DLL* i *COM*

DLL (*dynamic link library*) to biblioteki łączone dynamicznie które są ładowane do pamięci przez program który ich potrzebuje. Zawierają one zazwyczaj zestaw różnych funkcji tak jak zwykłe biblioteki statyczne. Główną zaletą bibliotek *DLL* jest możliwość poprawy działania programu przez zmianę biblioteki a nie przez przekompilowywanie całego programu. Cały system *Windows* jak i aplikacje pod niego pisane korzystają z bibliotek *DLL*.

COM (*component object model*) Jest technologią wprowadzona przez *Microsoft* w czasie pracy nad *OLE* (*Object linking and embedding*). Głównym problemem eksportowania klas z bibliotek *DLL* jest fakt, że w pliku nagłówkowym musimy umieszczać oprócz metod które świadczą o funkcjonalności klasy, także elementy składowe klasy: zmienne publiczne i prywatne. Musimy tak robić gdyż różne kompilatory budują kod na różny sposób. Model *COM* umożliwia nam udostępnienie klasy – zwanej teraz już *interface*, która implementuje same metody. Jest to podobne do sytuacji gdybyśmy eksportowali samą klasę wirtualną której wszystkie metody byłyby *PURE* i po której dziedziczyła by klasa która tak naprawdę chcemy eksportować.



Rysunek 1 - W jaki sposób *Windows* komunikuje się ze sprzętem, Źródło *DirectX SDK*

Jak widać na rysunku 8 system *Windows* nie komunikuje się bezpośrednio ze sprzętem. Nie dzieje się tak ponieważ różne implementacje sprzętu mogą różnić się od siebie. *Device Driver Interface* zapewnia standard współpracy sprzętu z systemem. Sterownik musi być zgodny z *DDI*. Specyfikacja *DDI* zmienia się w zależnosci od wymogów stawianych sterownikowi. *DDI* dla kompatybilności z *DirectX9* różni się od tego dla *DirectX8* itd. Programistów którzy używają biblioteki *DirectX9* interesuje kompatybilność sterownika tą właśnie biblioteką. *D3D* zapewnia możliwość sprawdzenia z jakim sterownikiem mamy do czynienia.

2.2 Biblioteka *DirectX*

DirectX jest to zbiór różnego rodzaju bibliotek obsługujących multimedia w systemie Windows:

- *Direct Graphics:*
  - *Direct Draw,*
  - *Direct 3D,*

- *Direct Audio:*
  - *Direct Sound,*
  - *Direct Music,*
- *Direct Input,*
- *Direct Play,*
- *Direct Show,*
- *Direct Setup,*

Direct Graphics obsługuje grafikę i dzieli się na *Direct Draw* odpowiedzialny za grafika 2D i *Direct3D* który zajmuje się grafika przestrzenną. *Direct Audio* obsługuje dźwięk, w jego skład wchodzi niskopoziomowa biblioteka *Direct Sound* oraz wysokopoziomowa *Direct Music*. *Direct Input* obsługuje urządenia wejścia wyjścia takie jak myszka, kierownica, klawiatura. *Direct Play* zapewnia połączenie między komputerami. *Direct Show* zajmuje się odtwarzaniem danych strumieniowych takich jak filmy. *Direct Setup* to biblioteka przy pomocy której możemy przeprowadzić instalacje całego pakietu *DirectX* z poziomu napisanej aplikacji.
Biblioteka *Direct3D* tak jak cały *DX* jest zbudowana na *COMach*. *D3D* daje nam możliwość sprawdzenia ile kart graficznych jest w systemie, sprawdzenia każdej karty pod względem potrzebnych funkcji oraz przede wszystkim wprawienie karty graficznej „w ruch”.

2.3 Jak pracuje biblioteka *D3D9*

W czasie pracy z *D3D* będziemy musieli zainicjalizować dwa interfejsy: jeden odpowiada za komunikacje z biblioteką a drugi bezpośrednio z kartą graficzna. Pierwszy będzie nam służył do sprawdzenia ile kart mamy w systemie, przepytania każdej z nich pod względem potrzebnych funkcji oraz utworzenie interfejsu który będzie rozmawiał już bezpośrednio z wybraną kartą graficzna.

Podstawową koncepcją przy pomocy której będziemy informować kartę jak ma pracować to *Render States* (celowo nie będziemy używać polskich prób tłumaczenia pojęć istniejących w *DX*, dlatego że na całym świecie używa się pojęć angielskich). Listing 13 przedstawia przykład użycia *SetRenderState* w celu poinformowania *Direct3D*, że ma obcinać obiekty które wyszły poza ekran.

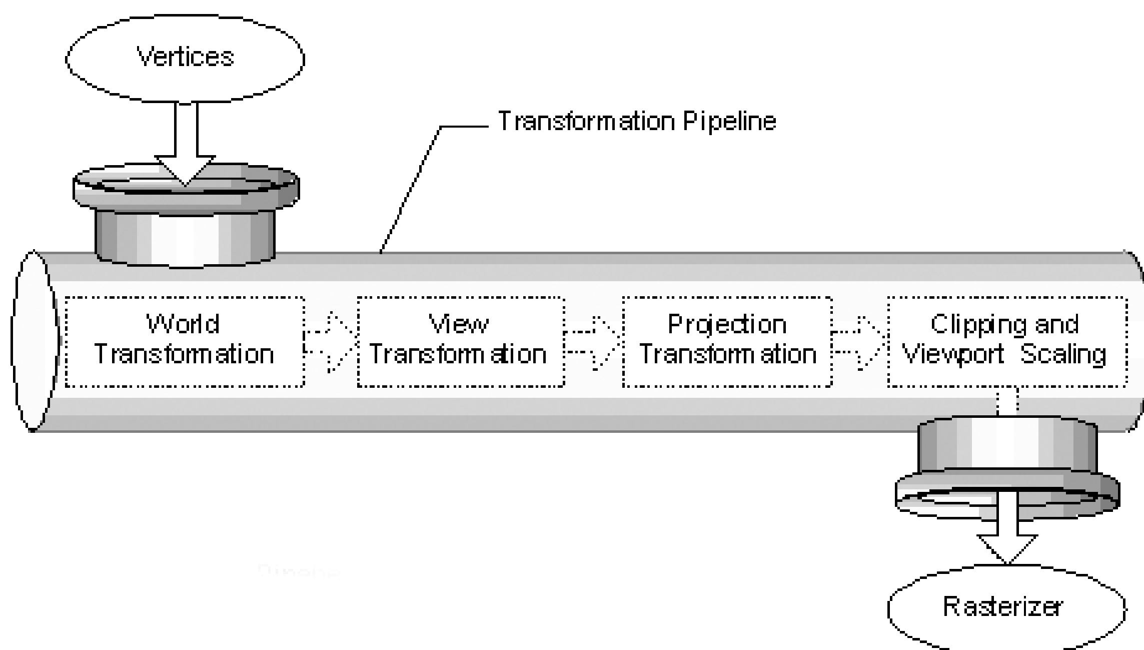
```
m_p3dDevice->SetRenderState( D3DRS_CLIPPING, TRUE );
```

Listing 1 – *Direct3DDevice9::SetRenderState*, Źródło własne

gdzie *m_p3dDevice* to interfejs *Direct3D* przypisany do karty graficznej.

Podstawowe pojęcia w grafice *3D*:

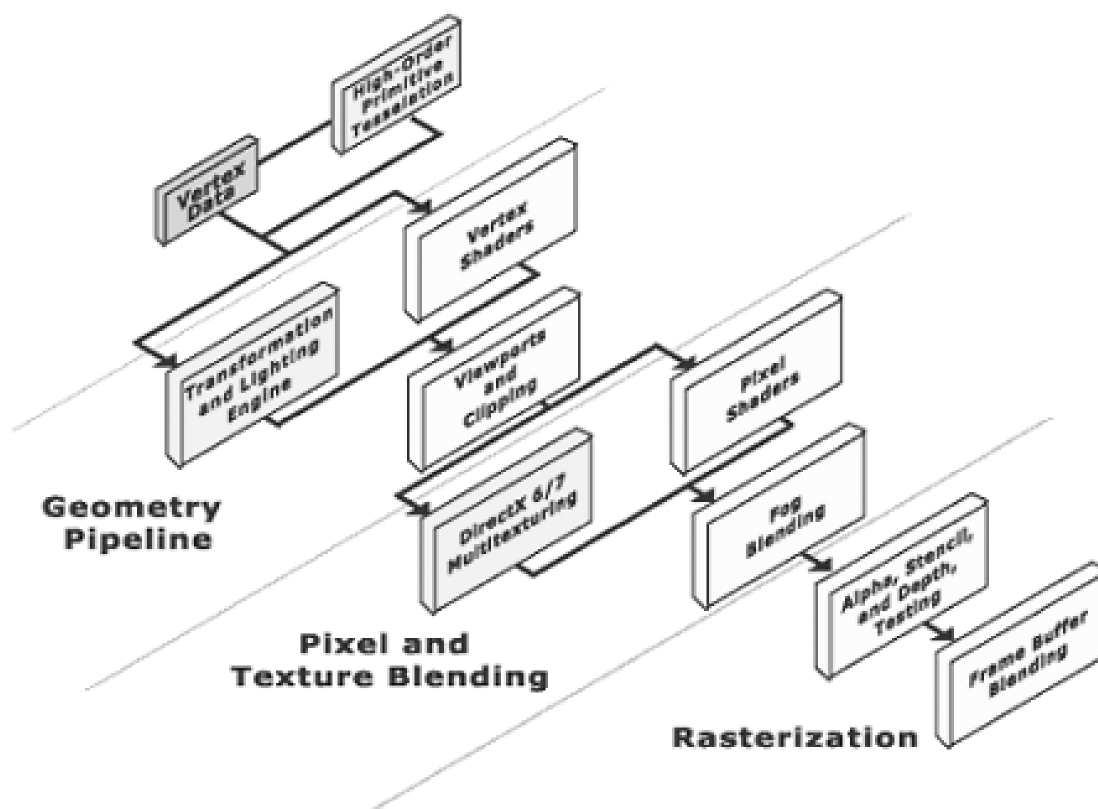
- Definicja 1 - *Vertices*, liczba pojedyncza *vertex* – inaczej punkt. Jest to podstawowa jednostka przekazywania danych karcie na temat geometrii obiektu który ma być narysowany. W *D3D* programista sam decyduje jak ma wyglądać format *vertexa* i informuje o tym kartę.
- Definicja 2 - *Rendering*. Zamiast „rysowanie” będziemy posługiwać się tym pojęciem
- Definicja 3 - *Pixel*. Podstawowa jednostka ekranu np. w rozdzielczość *640x480* pikseli.
- Definicja 4 - *Texel*. Podstawowa jednostka tekstury, odpowiednik piksela w ekranie.
- Definicja 5 - *Fixed Function Pipeline*. Jest to sposób przekształcania wierzchołków w karcie posiadającej wsparcie dla *TnL* (*transformation and lighting*) albo w emulacji programowej biblioteki *D3D*. Tym sposobem będziemy się zajmować. Drugim sposobem przekształcania są tzw. *vertex shaders*. Jest to całkowicie programowalna jednostka przekształcania wierzchołków. Ze względu na zaawansowanie tej tematyki nie będziemy się nią zajmować. Rysunek 9 przedstawia schemat pracy *Fixed Function Pipeline.*



Rysunek 2 **- Fixed Function Pipeline, Źródło DirectX SDK**

- Definicja 6 - *Multitexturing mode*. Multitexturowanie to technologia nakładania wielu tekstur na jeden piksel. Opatentowana przez *3dFX*, obecnie w każdej nowszej karcie graficznej. Jest to tryb, gdzie informujemy kartę ile tekstur ma nakładać na każdy piksel oraz jakie operacje mają być wykonywane między kolorami zawartymi w tych teksturach. Inna metodą jest tzw. *pixel shader*. Jest to adekwatnie do vertex shaderów programowalny sposób przekształcania kolorów i współrzędnych tekstur. Oczywiście ze względu na ograniczoność naszych ćwiczeń zajmiemy się tylko zwykłym *multiteksturowaniem*.

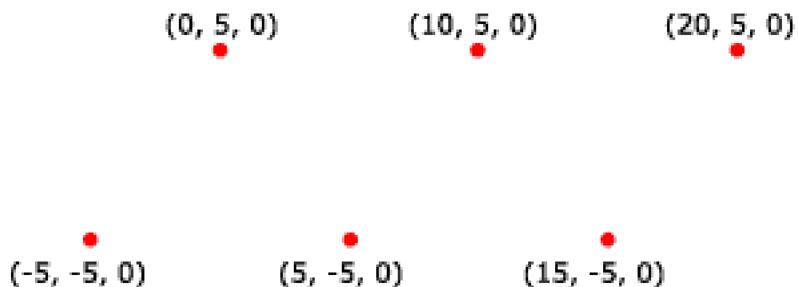
Na rysunku 10 przedstawiony jest schemat pracy jednostki *TnL* odpowiedzialnej za transformacje i oświetlenie.



Rysunek 3 - *Fixed Function* a *Vertex Shader*, *Multitexturing* a *Pixel Shader*, Źródło *DirectX SDK*

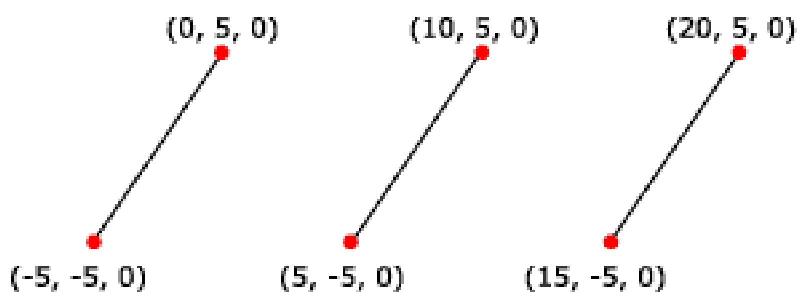
Jak przekazujemy informacje o geometrii do karty. Biblioteka *D3D* ma wiele sposobów przekazywania geometrii:

- możemy przekazywać listę punktów, gdzie każdy element listy będzie wyrenderowany jako oddzielny punkt, przykład: rysunek 11



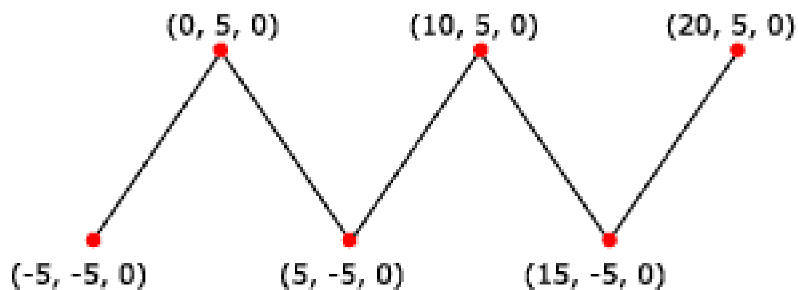
Rysunek 4 - Zbiór punktów, Źródło *DirectX SDK*

- możemy przekazywać listę linii. Każde dwa punkty będą narysowane w postaci linii łączącej je, przykład: rysunek 12



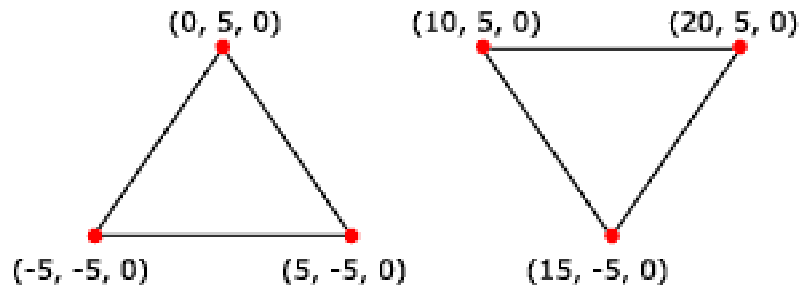
Rysunek 5 - Zbiór linii, Źródło *DirectX SDK*

- kolejną formą jest tzw. *line strip*, punkty są połączone ze sobą liniami, przykład rysunek 13



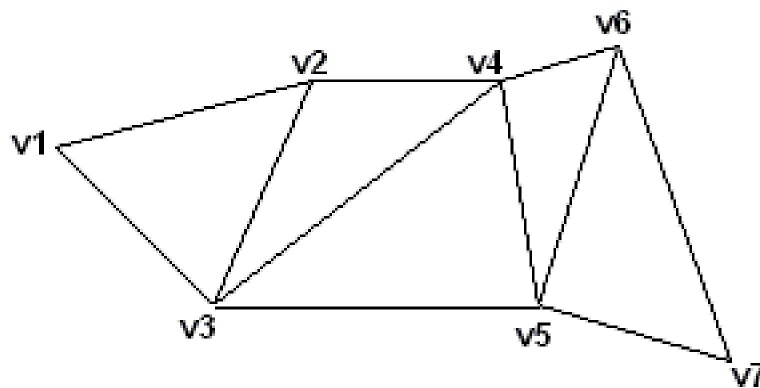
Rysunek 6 - Zbiór połączonych linii, Źródło *DirectX SDK*

- możemy przekazywać też trójkąty, każde kolejne trzy punkty tworzą trójkąt, przykład rysunek 14



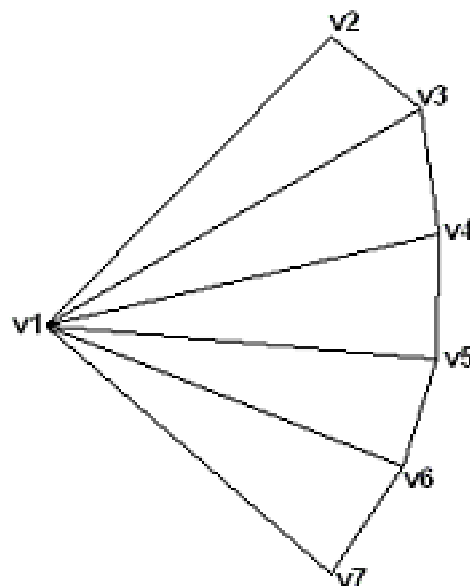
Rysunek 7 - Zbiór trójkątów, Źródło *DirectX SDK*

- *triangle strips* których przykład znajduje się na rysunku 15



Rysunek 8 - *Triangle Strip*, Źródło *DirectX SDK*

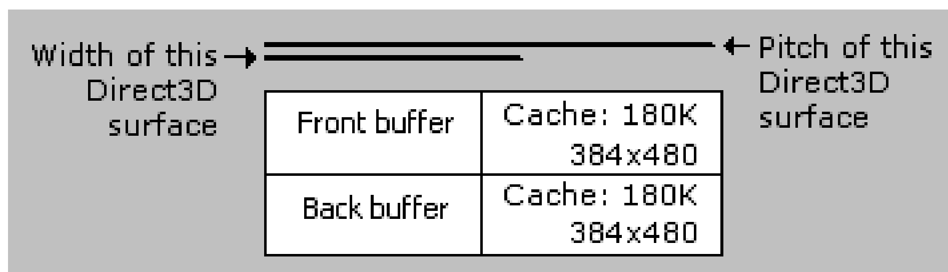
- *triangle fans* których przykład znajduje się na rysunku 16



Rysunek 9 - *Triangle Fan*, Źródło *DirectX SDK*

2.4 Podstawowe pojęcia w *D3D.*

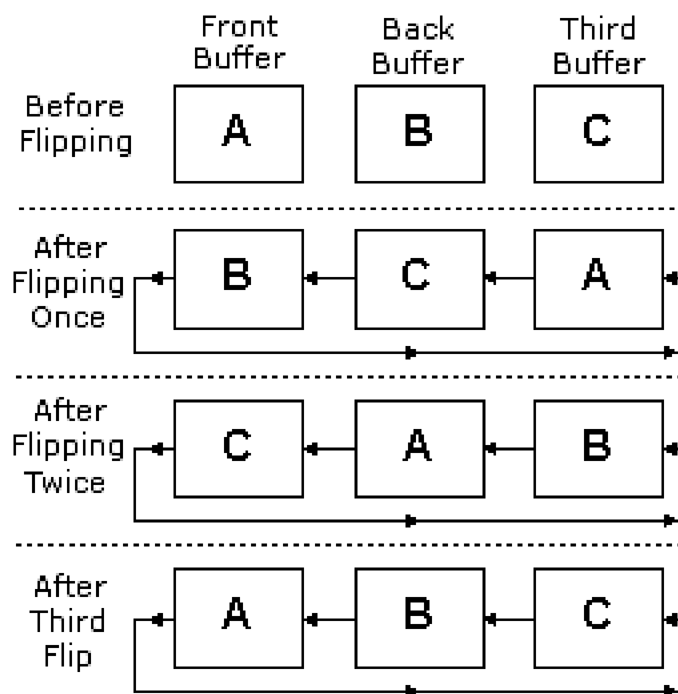
- Definicja 7 - *Surface* jest podstawowym pojęciem dotyczącym alokacji pamięci. Każdy element taki jak lista trójkątów czy tekstura jest *surface'em*. Ekran też jest *surface'em*. Surface odpowiada więc pewnemu obszarowi pamięci, w pamięci lokalnej komputera albo w pamięci karty graficznej. W *D3D* istnieje różnica między szerokościa fizyczną (*pitch*) surface'a a jego szerokością logiczną (*width*). Szerokość logiczna to szerokość jaką chcieliśmy zaalokować. Szerokość fizyczna to szerokość z jaką został *surface* zaalokowany w celu wyrównania danych w pamięci. Karty graficzne „lubią" potęgi dwójki. Tak więc np. *640x480 może być zaalokowany jako 1024x480*. Jest to związane z łatwością przeliczania adresów przy pomocy przesunięć bitów. Rysunek 17 ilustruje taki przypadek.



Rysunek 10 - *Width* a *Pitch*, Źródło *DirectX SDK*

- Definicja 8 - *Front buffer* jest to obszar zaalokowanej przez nas pamięci (*surface*), który jest widoczny na ekranie. Wszystko co narysujemy do front buffer będzie widoczne na ekranie.
- Definicja 9 - *Back buffer* jest to tzw. drugi bufor. W grafice nigdy nie rysuje się bezpośrednio do *front buffer*, gdyż może wystąpić miganie ekranu w czasie takiej operacji. Zawsze rysuje się do drugiego bufora, żeby w momencie odświeżania pionowego wysłać całość danych do pierwszego bufora. W praktyce ignoruje się moment odświeżania ekranu i wysyła dane do pierwszego bufora zaraz po zakończeniu renderingu. W prawdziwych aplikacjach graficznych nie kopiuje się danych z drugiego bufora do pierwszego. Zamienia się ich wskaźniki tak, że raz obszarem pamięci do którego będziemy *renderować* jest *front bufor* a raz *back*. Często stosuje się *dwa tylne bufory*. Ma to sens, kiedy synchronizujemy rysowanie z momentem odświeżania ekranu. Wtedy jeden bufor jest prezentowany, drugi jest czyszczony, a na trzecim rysujemy. Jest to najefektywniejsza metoda, gdyż nie marnujemy cennego czasu na oczekiwanie powrotu plamki na ekran. W aplikacjach,

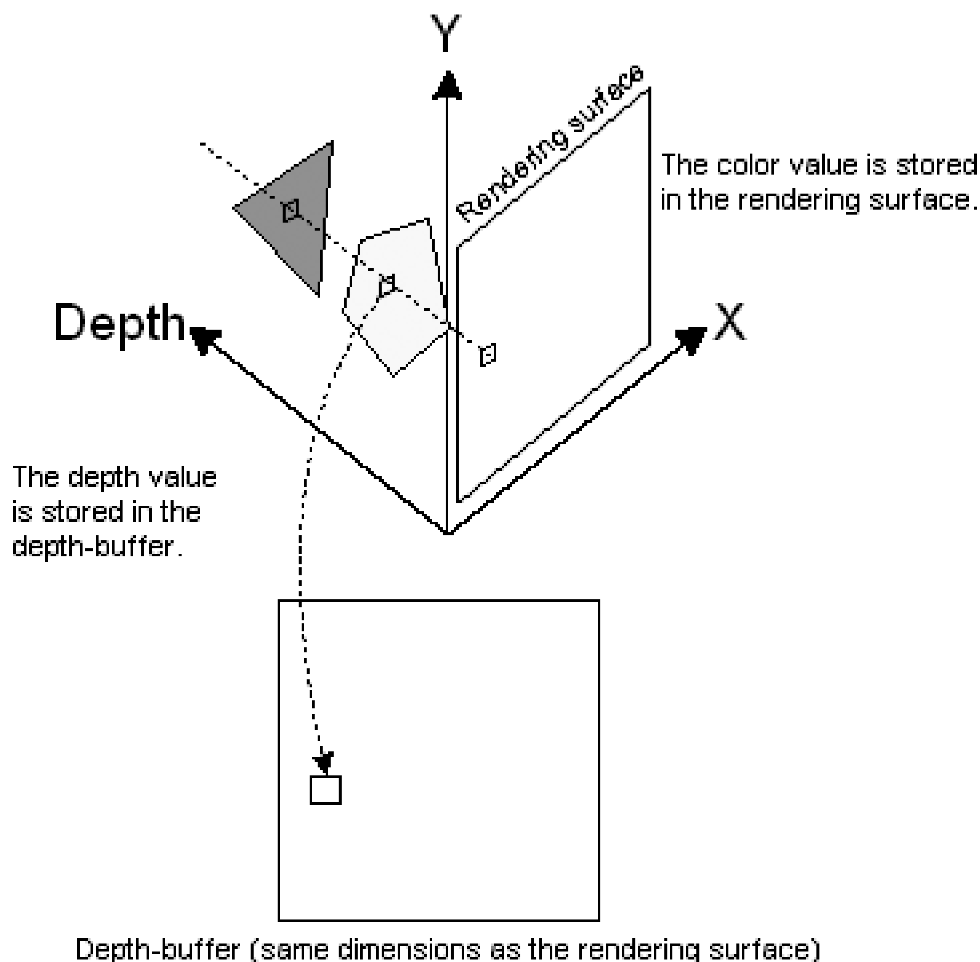
które nie synchronizują się, wystarczy użyć jeden bufor tylny. Działanie trzech buforów przedstawia rysunek 18.



Rysunek 11 - *Buffer Flipping*, Źródło *DirectX SDK*

- Definicja 10 - *Depth buffer* (*Zbuffer*). Jest to sposób na wyznaczanie powierzchni widocznych, najprostszy do zaimplementowania sprzętowego, który zdobył największą popularność. Polega na tym, że trzymany jest dodatkowy bufor o wymiarach ekranu, w którym zapisujemy informacje o odległości od kamery danego piksela, czyli po prostu wartości Z w układzie kamery. Między innymi dla bufora *Z* stosowane są płaszczyzny obcinania w kamerze. Aby mieć lepszą dokładność danych w *Zbuforze*, należy ustawić rozsądnie *Near Clipping Plane* i *Far Clipping Plane*. Dane w *Zbuforze* przyjmują wartości od *0.0* do *1.0*. Tak więc *Near Clipping Plane* jest mapowana jako *0.0*, a *Far Clipping Plane* jako *1.0*. W zależności od ustawienia *Zbufora* (*16 bitowy*, *24 bitowy*, *32 bitowy*) mamy odpowiednią dokładność. Jeżeli nie ustawiamy płaszczyzn obcinania rozsądnie, może się zdarzyć, że *16 bitowy Zbufor* nie będzie dokładnie wyznaczał powierzchni widocznych. Praca *Zbufora* polega na tym, że w czasie renderowania każdy piksel poddawany jest testowi *Z*. Jeżeli wartość w *Z buforze* informuje, że jest narysowany piksel na ekranie jest bliżej, niż ten, który chcemy narysować, po prostu nie rysujemy. Jeżeli jest dalej, to rysujemy nasz piksel

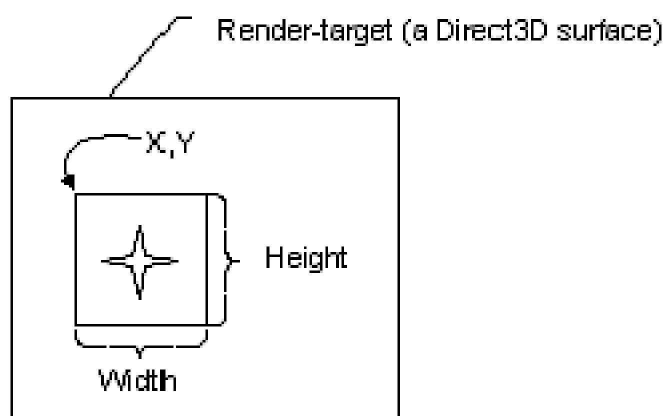
i uaktualniamy wartość *Z*. Oczywiście wszystkie te operacje dzieją się w karcie graficznej, my mamy tylko możliwość kontrolowania ich. Rysunek 19 przedstawia działanie *Bufora Z.*



Rysunek 12 - Zasada działania *Depth-buffer* (*ZBuffer*), Źródło *DirectX SDK*

- Definicja 11 - *Stencil bufor*. Tzw bufor szablonu. Służy do uzyskiwania zaawansowanych efektów. Jest on *łączony z Zbuforem*. *Zbufor* zazwyczaj zajmuje *24 bity*, zaś *stencil 8 bitow*. Razem *32* bity, czyli wyrównanie do potęgi dwójki.

- Definicja 12 - *Viewport* informuje *D3D* jaki obszar ekranu ma być renderowany. Rysunek 20 przedstawia przypadek kiedy *viewport* jest mniejszy niż tylny bufor. *Render-target* to obszar pamięci gdzie będziemy *renderować*. *X*, *Y* oznaczają lewy górny róg *viewport*, natomiast *height* to szerokość w *pikslach* a *width* to wysokość w *pikslach* od lewego górnego rogu.



Rysunek 13 - *Viewport*, Źródło *DirectX SDK*

Zazwyczaj *ustawiamy viewport na cały obszar ekranu*, np. na *640x480*. Implementacje takiego przypadku przedstawia listing 14.

```
m_vp.X = 0;
m_vp.Y = 0;
m_vp.Width = m_pConfig->m_wWidth;
m_vp.Height = m_pConfig->m_wHeight;
m_vp.MinZ = 0.0f;
m_vp.MaxZ = 1.0f;

if ( FAILED( m_p3dDevice->SetViewport( &m_vp ) ) )
      AnErrorMsg("D3D9 Renderer Error","Can't Set Viewport");
```

Listing 2 – Ustawianie *Viewport*, Źródło własne

- Definicja 13 - *Surface Locking*. Gdy pragniemy posiadać dostęp do danych trzymanych np. w teksturze, albo w buforze *vertexów*, możemy dokonoać *lockowania surface'a*, aby odczytać jego dane. Zazwyczaj nie praktykuje się czytania z *surface'ów*. *Lockowanie* służy po to, aby można było coś zapisać do nich, np. dane geometrii czy teksture odczytaną z pliku. Wiele *surface'ów* tworzonych jest z parametrem *WRITE_ONLY*, który znacznie przyspiesza prace karty graficznej z jednostką *TnL*. *Lockowanie surface'a* przedstawia listing 15.

```
DWORD *dwWhere;
D3DLOCKED_RECT d3drect;
rectTxt.top = 0;
rectTxt.left = 0;
rectTxt.right = 2;
rectTxt.bottom = 1;

if ( FAILED( hr = pSurface->LockRect( &d3drect, &rectTxt, NULL ) ) )
      AnErrorMsg("Error While Loading Texture","Can't Lock Surface");

dwWhere = (DWORD*)d3drect.pBits;

dwWhere[0] = 0x0;
dwWhere[1] = 0xffffffff;

pSurface->UnlockRect();
```

Listing 3 – Wypełnianie *surface'a*, Źródło własne

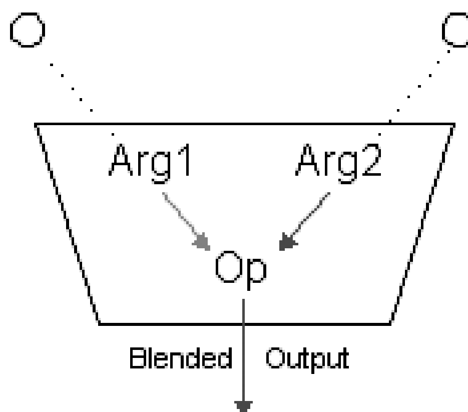
- *Kolor*. *D3D* może trzymać kolor w różnych formatach. Niektóre formaty nadają się dla ekranu, niektóre dla tekstur. W czasie *renderowania* konwersja między formatami wykonywana jest automatycznie w karcie.

D3DFMT_R8G8B8	24 bitowy format z 8 bitami na składową koloru
D3DFMT_A8R8G8B8	32 bitowy format z 8 bitami na składową koloru i 8 bitów na składową alfa
D3DFMT_X8R8G8B8	32 bitowy format z 8 bitami na składową koloru i 8 bitów zarezerwowanych
D3DFMT_R5G6B5	16 bitowy format z 5 bitami dla składowej czerwonej, 6 bitów dla zielonej i 5 dla niebieskiej
D3DFMT_A1R5G5B5	16 bitowy format z 5 bitami dla składowej czerwonej, 5 bitów dla zielonej, 5 dla niebieskiej i jednego bitu dla składowej alfa
D3DFMT_X1R5G5B5	16 bitowy format z 5 bitami dla składowej czerwonej, 5 bitów dla zielonej, 5 dla niebieskiej i jednego bitu zarezerwowanego
D3DFMT_A4R4G4B4	16 bitowy format z 4 bitami dla składowej czerwonej, 4 bitów dla zielonej, 4 dla niebieskiej i 4 bitów dla składowej alfa
D3DFMT_A8	8 bitowy format z 8 bitami dla składowej alfa
D3DFMT_R3G3B2	8 bitowy format z 3 bitami dla składowej czerwonej, 3 bitami dla składowej zielonej i 2 dla niebieskiej

D3DFMT_A8R3G3B2	16 bitowy format z 3 bitami dla składowej czerwonej, 3 bitami dla składowej zielonej, 2 dla niebieskiej i 8 bitów dla składowej alfa
D3DFMT_X4R4G4B4	16 bitowy format z 4 bitami dla składowej czerwonej, 4 bitów dla zielonej, 4 dla niebieskiej i 4 bitów zarezerwowanych
D3DFMT_A2R10G10B10	32 bitowy format z 10 bitami na składową koloru i 2 bitami na składową alfa

Jak widać, niektóre formaty zawierają *składową alfa*, która jest często używana do oznaczenia przezroczystości danego *teksela* (może też służyć do innych celów, wszystko zależy od programisty).

- Definicja 14 - *SetRenderState*. Tak jak wspomnieliśmy wcześniej, *render states* służą do kontroli operacji wykonywanych przez kartę graficzną. Można przy ich pomocy kontrolować wszystkie zachowania karty.
- Definicja 15 - *SetTextureStageState*. Stany tekstur służą do informowania karty jakie operacje mają być wykonywane między teksturami, żeby uzyskać żądany efekt. Rysunek przedstawiający schemat wykonywania operacji znajduje się na rysunku 21.



Rysunek 14 - Działanie texture stage, Źródło *DirectX SDK*

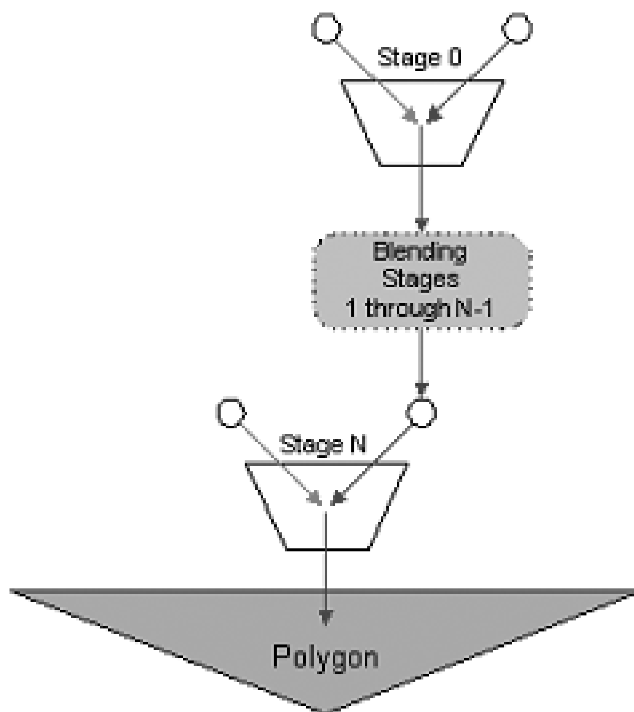
Przykładowa sekwencja ustawienia stanów operacji na *teksturze* znajduje się na listingu 16.

```
SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_MODULATE );
SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_DIFFUSE );
SetTextureStageState( 0, D3DTSS_ALPHAARG2, D3DTA_TEXTURE );
SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_MODULATE );
SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_DIFFUSE );
SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_TEXTURE );
```

Listing 4 – Przykładowe *SetTextureStageState*, Źródło własne

Kolor z tekstury pierwszej ma być przemnożony przez wartość oświetlenia. To samo dzieje się ze współczynnikiem *alfa*. Jak można zauważyć, *D3D* oddzielnie operuje na kolorze, oddzielnie na *alfa*. Jednak zależne jest to od karty, na której ten kod zostanie wykonany. Niektóre karty nie pozwalają na różne operacje na kolorze i na składowej alfa.

W przypadku *multitexturing mode*, którym będziemy się zajmować, proces nakładania koloru rozszerza się na tyle *tekstur*, ile włączymy, zatem schemat z rysunku 21 będzie wykonywany tyle razy ile *tekstur* nałożymy co przedstawia rysunek 22.



Rysunek 15 - Multitexturing, Źródło *DirectX SDK*

Przykład ustawienia stanów operacji dla wielu *tekstur* znajduje się na listingu 17.

```
SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_MODULATE );
SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_DIFFUSE );
SetTextureStageState( 0, D3DTSS_ALPHAARG2, D3DTA_TEXTURE );
SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_MODULATE );
SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_DIFFUSE );
SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_TEXTURE );
SetTextureStageState( 1, D3DTSS_ALPHAOP, D3DTOP_MODULATE );
SetTextureStageState( 1, D3DTSS_ALPHAARG1, D3DTA_CURRENT );
SetTextureStageState( 1, D3DTSS_ALPHAARG2, D3DTA_TEXTURE );
SetTextureStageState( 1, D3DTSS_COLOROP, D3DTOP_MODULATE );
SetTextureStageState( 1, D3DTSS_COLORARG1, D3DTA_CURRENT );
SetTextureStageState( 1, D3DTSS_COLORARG2, D3DTA_TEXTURE );
```

Listing 5 – Przykładowe *SetTextureStageState* multitexturingu, Źródło własne

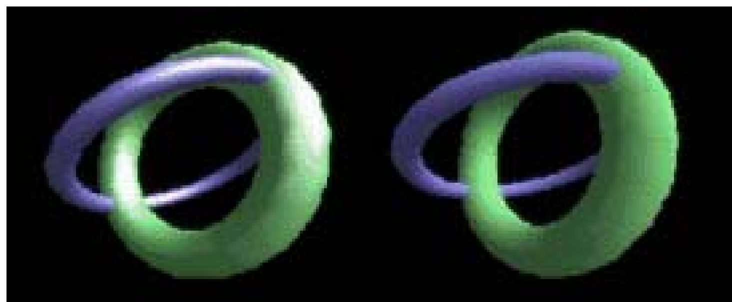
W pierwszej *teksturze*, jej kolor zostanie przemnożony przez kolor oświetlenia. W drugiej kolor zostanie przemnożony przez wynik operacji z pierwszej *tekstury*. Operacje między kolorami kontrolujemy przez *COLOROP*, zaś każdy *COLOROP* to *operacja* przynajmniej *dwuargumentowa*. *D3DTA_CURRENT* to kolor z operacji z poprzedniego *stage'a*.

- Definicja 16 - *Wektory Normalne*. Każdy *vertex* musi mieć wektory normalne. Są one potrzebne między innymi do oświetlania. Algorytm do wyznaczania wektorów normalnych poznamy później.
- Definicja 17 - *Materiał*. *D3DMATERIAL* określa, jak bardzo dany obiekt odbija, absorbuje światło, a dokładnie jego kolor. Oko ludzkie widzi kolory otaczającego nas świata przez to, jak obiekty odbijają światło tzn. jaki kolor światła został odbity. Naprzykład obiekt czerwony w świetle niebieskim będzie czarny. Pola struktury *D3DMATERIAL*

Member	Value
Diffuse	(R:1, G:1, B:1, A:0)
Specular	(R:0, G:0, B:0, A:0)
Ambient	(R:0, G:0, B:0, A:0)
Emissive	(R:0, G:0, B:0, A:0)
Power	0

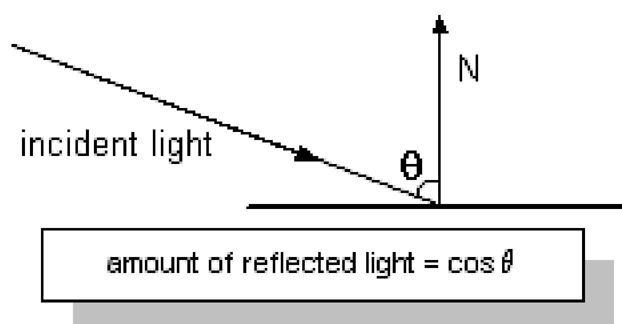
- Definicja 18 - *Ambient* to współczynnik odbijania *światła otaczającego*. Definicja 19 - *Diffuse* to współczynnik odbijania światła z innych świateł. Definicja 20 - *Specular* określa stopień odbijania rozbłysków światła. Definicja 21 - *Power* określa moc tych

rozbłysków. *Emissive* określa stopień emisji światła. Zazwyczaj większość materiałów ma *Ambient* i *Diffuse* ustawione na *1.0, 1.0, 1.0, 1.0* i kolor światła przemnażany przez kolor tekstury. Wtedy tekstura „decyduje" o stopniu odbijania światła. Przykład wpływu parametrów na oświetlenie widać na ilustracji 7.



Ilustracja 1 - Obiekt z właściwością *specular* i obiekt bez, Źródło *DirectX SDK*

Obiekt po lewej posiada wartość specular ustawioną. Obiekt po prawej nie posiada tej wartości. Natężenie światła obliczane jest na każdy *vertex* przy pomocy iloczynu skalarnego między kierunkiem padania światła a wektorem normalnym *vertexa*. Do tego modyfikowany jest przez odległość. Zachowanie natężenia światła względem odległości jest oczywiście modyfikowalne i będziemy się tym zajmować dalej.
Rysunek 23 przedstawia zależność między kątem padania światła a wektorem normalnym powierzchni na która pada.



Rysunek 16 - Liczenie natężenia światła, Źródło *DirectX SDK*

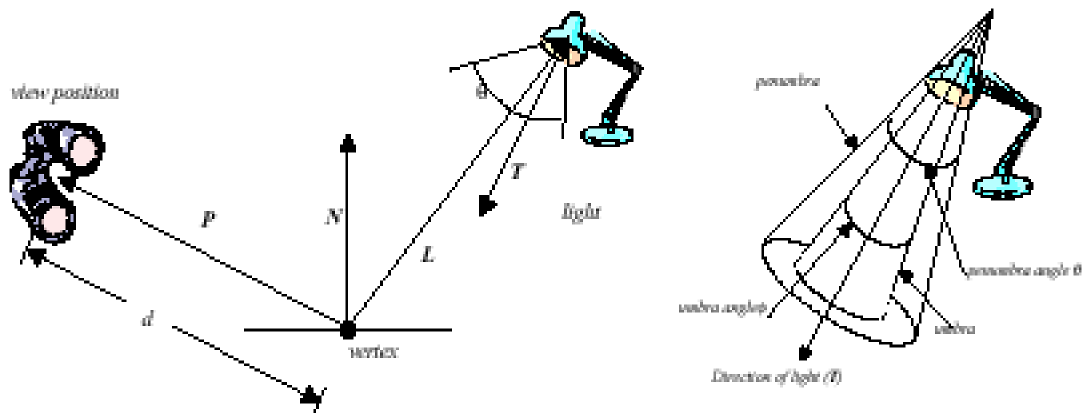
- *D3DLIGHT*. Jest to struktura przy pomocy której opisujemy właściwości światła.

Member	Default
Type	D3DLIGHT_DIRECTIONAL
Diffuse	(R:1, G:1, B:1, A:0)
Specular	(R:1, G:1, B:1, A:0)
Ambient	(R:1, G:1, B:1, A:0)
Position	(0, 0, 0)
Direction	(0, 0, 1)
Range	0
Falloff	0
Attenuation0	0
Attenuation1	0
Attenuation2	0
Theta	0
Phi	0

- *Type* – to typ światła, *D3D* posiada 3 typy światła: *DIRECTIONAL* – kierunkowe, np. słońce padające na ziemię, *POINT* – punktowe, np. światło żarówki rozchodzące się we wszystkich kierunkach, *SPOT* – światło miejscowe, np. światło latarki,
- *Diffuse* – współczynnik *Diffuse* światła, zazwyczaj używa się go do określenia mocy światła, w czasie liczenia natężenia światła przemnażany jest przez wartość *Diffuse* materiału,
- *Ambient* – współczynnik *Ambient* światła, nie używa się go, zostawiając ten współczynnik dla światła otaczającego (występującego w scenie),
- *Specular* – wartość *specular* światła, przemnażane przez wartość *specular* materiału w czasie liczenia oświetlenia,
- *Position* – aktualna pozycja światła (nie dotyczy świateł *DIRECTIONAL*),
- *Direction* – kierunek padania światła (nie dotyczy świateł *POINT*),
- *Range* – zasięg światła,
- *FallOff* – określa zanik światła między kątami *Theta* i *Phi* w świetle *SPOT*,
- *Atteniation0, 1, 2* – parametry zaniku światła wraz z odległością, pierwszy – stały, drugi – liniowy, trzeci – kwadratowy (patrz wzór oświetlenia na stronie 43),

- *Theta* – w radianach, określa stożek wewnętrzny światła *SPOT*,
- *Phi* – w radianach, określa zewnętrzny światła *SPOT*.

Wzory na natężenie oświetlenia znajdują się na rysunku 24,



Rysunek 17 - Przeliczanie oświetlania w *D3D*, Źródło *DirectX SDK*

Opis wektorów z rysunku 24:

- L – wektor jednostkowy od punktu do światła,
- T – kierunek światła,
- N – jednostkowy wektor normalny powierzchni,
- P – jednostkowy wektor od punktu do kamery,
- H – *halfway* jednostkowy wektor, dla świateł punktowych: $P + L$, dla świateł kierunkowych i spot: $P - T$,
- A_0, A_1, A_2 - właściwości światła do zanikania,
- θ – kąt półcienia,
- ϑ – kąt cienia,
- $\cos\alpha = -L \bullet T$,
- d – odległość od punktu do światła.

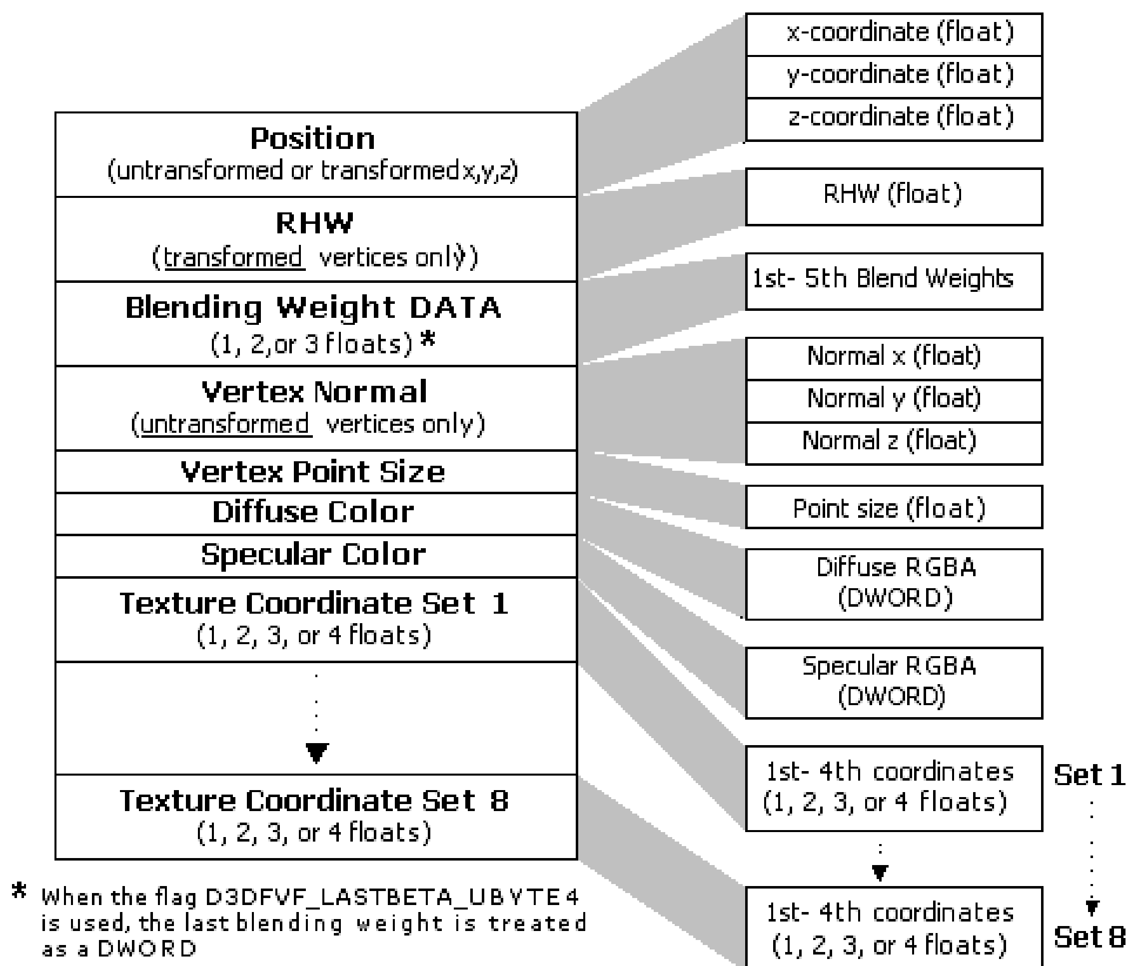
$$spotlight_effect = \begin{bmatrix} 1 \quad if \quad not \quad spot \quad light \\ \left(\max\left\{ \left(\dfrac{\cos\alpha - \cos \vartheta/2}{\cos \theta/2 - \cos \vartheta/2} \right), 0 \right\} \right)^{falloff} \end{bmatrix}$$

$$diffuse_color = emissive_{mat} + ambient_{global} \cdot ambient_{mat} +$$
$$\sum_{i=0}^{numlights-1} \left(\frac{1}{A_0 + A_1 \cdot d + A_2 \cdot d^2} \right) \cdot (spotlight_effect) \cdot$$
$$\left[ambient_{light} \cdot ambient_{mat} + (\max\{L \bullet N, 0\}) \cdot diffuse_{light} \cdot diffuse_{mzt} \right]$$

$$specular_color = \sum_{i=0}^{numlights-1} \left(\frac{1}{A_0 + A_1 \cdot d + A_2 \cdot d^2} \right) \cdot (spotlight_effect) \cdot$$
$$\left[(\max\{H \bullet N, 0\})^{power} \cdot specular_{light} \cdot specular_{mat} \right]$$

Jeśli $(N \bullet L) <= 0$, *specular_color*=0, ponieważ światło jest po drugiej stronie obiektu.

- *Verteksy***.** Aby poinformować kartę jak wygląda *vertex* będziemy używać flag *FVF* (*Flexible Vertex Format*). Przy użyciu *FVF* mamy możliwość wyboru pól opisanych na rysunku 25.



Rysunek 18 - *Flexible Vertex Format*, Źródło *DirectX SDK*

Opis pól z rysunku 25:

- *Position* – nieprzetransformowany pozycja (np. układ obiektu),
- *RHW* - *reciprocal homogeneous W* – musimy go podawać kiedy sami wykonujemy transformacje, a chcemy żeby karta obcieła trójkąty,
- *Blending Weight* – wagi,
- *Vertex Normal* – nieprzetransformowany wektor normalny,
- *Vertex Point Size,*
- *Diffuse Color* – wartośc składowej *diffuse* koloru oświetlenia, używane kiedy sami będziemy wykonywać oświetlenie,
- *Specular Color* – jak wyżej tyle, że dla składowej *specular*,

- *Texture Coordinate Set* – zestaw współrzędnych tekstury, *D3D* obsługuje maksymalnie 8 zestawów.

- Definicja 22 - *Indeksy*. Dla efektywnego *renderowania* geometrii nie podaje się listy *vertexów* z których każde trzy kolejne *vertexy* tworzą trójkąt. Podaje się listę punktów oraz listę *indeksów*, które *są referencjami do vertex bufora*. Trzy *indeksy* tworzą trójkąt. Jest to o wiele efektywniejsze gdyż *vertex* o indeksie *„n”* może być użyty wiele razy bez konieczności wilokrotnego podawania go w liście wierzcholków.
- Definicja 23 - *Vertex bufor*. Jest to bufor trzymający listę wierzchołków. W przypadku kart ze sprzętowym *TnL vertex bufor* umiejscowiony jest w pamięci karty graficznej.
- Definicja 24 - *Index bufor*. Jest to bufor trzymający listę indeksów do wierzchołków. W przypadku kart firmy *ATI* z serii *Radeon (R2xx/R3xx)* oraz wyższych *indeks bufor* umiejscowiony jest w pamięci karty graficznej. W innych kartach w pamięci *RAM*.
- Definicja 25 - *IDirect3D9*. Jest to interfejs odpowiedzialny za komunikowanie się z biblioteką graficzną *D3D*.
- Definicja 26 - *IDirect3Ddevice9*. Jest to interfejs odpowiedzialny za komunikowanie się z konkretnym urządzeniem – kartą graficzną.

Tryby pracy urządzenia *D3D*:

- Definicja 27 - *Hardware Vertex Processing*. Jest to tryb pracy urządzenia *D3D* (karty) kiedy transformacje wierzchołków, oświetlanie i obcinanie wykonywane jest przez kartę graficzną. Tryb *fixed funtion vertex processing*, akcelerowany sprzętowo, posiadają następujące układy: *NVIDIA*: *GeForce, GeForce2 (GTS/MX), GeForce3, GeForce4 (MX/TI), GeForce FX*, *ATI*: *Radeon7500, Radeon8500, Radeon9000, Radeon9100, Radeon9500, Radeon9700, Radeon9800* oraz *Matrox Parhelia*. Tryb *programmable vertex processing* o różnym stopniu zaawansowania posiadają: *NVIDIA*: *GeForce3, GeForce4 TI, GeForceFX, ATI*: *Radeon8500, Radeon9000, Radeon9100, Radeon9200, Radeon9500, Radeon9700, Radon9600, Radeon9800* oraz *Matrox Parhelia*.
- Definicja 28 - *Software Vertex Processing*. To tryb pracy *D3D*, kiedy transformacje wierzchołków, oświetlanie i obcinanie wykonywane jest przez bibliotekę *D3D* czyli przez *CPU*.

- Definicja 29 - *Reference Rasterizer*. To tryb pracy *D3D* całkowicie programowy, bez żadnej akceleracji sprzętowej. Nawet rysowanie trójkątów jest programowe. Służy do testowania sterowników i szukania błędów.
- Definicja 30 - *Pure Device*. To tryb pracy *D3D*, kiedy biblioteka nie zapewnia żadnego mechanizmu zapamiętywania wydanych poleceń. Teoretycznie najszybszy, gdyż wszystkie polecenia przekazywane są bezpośrednio do karty. Niestety żadne z poleceń typu *GET* nie będzie działać, ze względu na brak zapamiętywania.